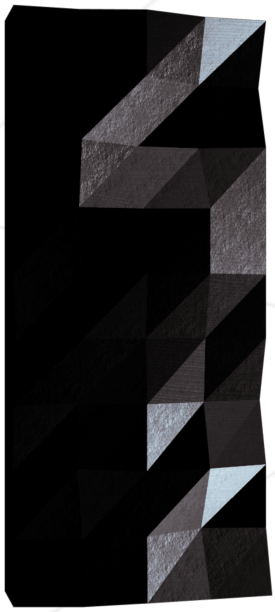

Codexion

Master the race for resources before the deadline masters you

Summary: Race against time in this thrilling concurrency challenge! Orchestrate multiple coders competing for limited USB dongles using POSIX threads, mutexes, and smart scheduling—master resource synchronization before burnout strikes.

Version: 1.1

Contents

Chapter I

Introduction

Computer programming (often simply called "coding") is the process of designing and building executable software to accomplish specific computing tasks. It involves analysis, generating algorithms, profiling algorithms' accuracy and resource consumption, and implementing these algorithms in a chosen programming language. The earliest computers were programmed directly in machine code, sequences of binary instructions understood by the hardware. Over time, higher-level languages were developed, allowing coders to express ideas more abstractly.

Coding culture extends beyond the purely technical. It blends problem-solving,creativity, and collaboration in shared environments ranging from open-source communities to coworking spaces and hackathons. Within these hubs, coders not only write code but also debug, refactor, and share tools. Some resources in such environments are plentiful (e.g., coffee), while others — like specialized hardware dongles or licensed development environments — are limited and must be shared carefully to avoid bottlenecks.

Just as ancient philosophers debated truth and existence, modern coders debate programming paradigms, design patterns, and ethical software practices. Is it better to optimize for speed or maintainability? Should one refactor now or after the product proves itself? How can a team ensure fair access to shared development resources without stalling progress?

Historically, programming was often a solitary pursuit, but the evolution of collaborative tools — version control systems, real-time editors, and distributed issue trackers — has made coding an increasingly social and cooperative process. Alongside this, concurrency and synchronization have become vital skills: knowing how to let many developers or processes work without stepping on each other's toes is as important as writing the code itself.

In the modern era, with teams spread across time zones and projects using time-limited or scarce hardware, the challenge is not only to write correct code, but also to design fair and efficient protocols for sharing resources. This simulation models such a scenario: coders working in a shared space, requiring two rare USB dongles to compile their quantum code. They must coordinate, avoid deadlocks, and prevent burnout — because in the world of collaborative coding, as in life, access and timing are everything.

Chapter II

AI Instructions

Context

During your learning journey, AI can assist with many different tasks. Take the time to explore the various capabilities of AI tools and how they can support your work. However, always approach them with caution and critically assess the results. Whether it's code, documentation, ideas, or technical explanations, you can never be completely sure that your question was well-formed or that the generated content is accurate. Your peers are a valuable resource to help you avoid mistakes and blind spots.

Main message

- Use AI to reduce repetitive or tedious tasks.
- Develop prompting skills — both coding and non-coding — that will benefit your future career.
- Learn how AI systems work to better anticipate and avoid common risks, biases, and ethical issues.
- Continue building both technical and power skills by working with your peers.
- Only use AI-generated content that you fully understand and can take responsibility for.

Learner rules:

- You should take the time to explore AI tools and understand how they work, so you can use them ethically and reduce potential biases.
- You should reflect on your problem before prompting — this helps you write clearer, more detailed, and more relevant prompts using accurate vocabulary.
- You should develop the habit of systematically checking, reviewing, questioning, and testing anything generated by AI.
- You should always seek peer review — don't rely solely on your own validation.

Phase outcomes:

- Develop both general-purpose and domain-specific prompting skills.
- Boost your productivity with effective use of AI tools.
- Continue strengthening computational thinking, problem-solving, adaptability, and collaboration.

Comments and examples:

- You'll regularly encounter situations — exams, evaluations, and more — where you must demonstrate real understanding. Be prepared, keep building both your technical and interpersonal skills.
- Explaining your reasoning and debating with peers often reveals gaps in your understanding. Make peer learning a priority.
- AI tools often lack your specific context and tend to provide generic responses. Your peers, who share your environment, can offer more relevant and accurate insights.
- Where AI tends to generate the most likely answer, your peers can provide alternative perspectives and valuable nuance. Rely on them as a quality checkpoint.

✓ Good practice:

> I ask AI: "How do I test a sorting function?" It gives me a few ideas. I try them out and review the results with a peer. We refine the approach together.

✗ Bad practice:

> I ask AI to write a whole function, copy-paste it into my project. During peer-evaluation, I can't explain what it does or why. I lose credibility — and I fail my project.

✓ Good practice:

> I use AI to help design a parser. Then I walk through the logic with a peer. We catch two bugs and rewrite it together — better, cleaner, and fully understood.

✗ Bad practice:

> I let Copilot generate my code for a key part of my project. It compiles, but I can't explain how it handles pipes. During the evaluation, I fail to justify and I fail my project.

Chapter III

Common Instructions

- Your project must be written in C.
- Your project must be written in accordance with the Norm. If you have bonus files/functions, they are included in the norm check, and you will receive a `0` if there is a norm error.
- Your functions should not quit unexpectedly (segmentation fault, bus error, double free, etc.) except for undefined behavior. If this occurs, your project will be considered non-functional and will receive a `0` during the evaluation.
- All heap-allocated memory must be properly freed when necessary. Memory leaks will not be tolerated.
- If the subject requires it, you must submit a `Makefile` that compiles your source files to the required output with the flags `-Wall`, `-Wextra`, and `-Werror`, using `cc`. Additionally, your `Makefile` must not perform unnecessary relinking.
- Your `Makefile` must contain at least the rules `$(NAME)`, `all`, `clean`, `fclean` and `re`.
- To submit bonuses for your project, you must include a `bonus` rule in your `Makefile`, which will add all the various headers, libraries, or functions that are not allowed in the main part of the project. Bonuses must be placed in `_bonus.{c/h}` files, unless the subject specifies otherwise. The evaluation of mandatory and bonus parts is conducted separately.
- If your project allows you to use your `libft`, you must copy its sources and its associated `Makefile` into a `libft` folder. Your project's `Makefile` must compile the library by using its `Makefile`, then compile the project.
- We encourage you to create test programs for your project, even though this work **does not need to be submitted and will not be graded**. It will give you an opportunity to easily test your work and your peers' work. You will find these tests especially useful during your defence. Indeed, during defence, you are free to use your tests and/or the tests of the peer you are evaluating.
- Submit your work to the assigned Git repository. Only the work in the Git repository will be graded. If Deepthought is assigned to grade your work, it will occur

after your peer-evaluations. If an error happens in any section of your work during Deepthought's grading, the evaluation will stop.

Chapter IV

Overview

Here are the things you need to know if you want to succeed in this assignment:

- One or more **coders** sit in a circular inclusive co-working hub.
  In the center, there is a shared **Quantum Compiler**.
- The coders alternatively **compile**, **debug**, or **refactor**.
  While compiling, they are not debugging nor refactoring;
  while debugging, they are not compiling nor refactoring;
  and, of course, while refactoring, they are not compiling nor debugging.
- There are **USB dongles** on the table. There are **as many dongles as coders**.
- Compiling quantum code requires two dongles plugged in simultaneously,
  one in each hand: a coder takes their left and right dongles to compile.
- When a coder finishes compiling, they put both dongles back on the table and start debugging.
  Once debugging is done, they start refactoring. The simulation stops when a coder **burns out** due to lack of compiling.
- Every coder needs to compile regularly and should never burn out.
- Coders do not communicate with each other.
- Coders do not know if another coder is about to burn out.
- Needless to say, coders should avoid burnout!

Chapter V

Global rules

You have to write **one program** that complies with the following rules:

- **Global variables are forbidden!**
- Your program must take the following arguments (all mandatory):
  `number_of_coders time_to_burnout time_to_compile time_to_debug time_to_refactor number_of_compiles_required dongle_cooldown scheduler`

  - `number_of_coders`: The number of coders and also the number of dongles.
  - `time_to_burnout` (in milliseconds): If a coder did not start compiling within `time_to_burnout` milliseconds since the beginning of their last compile or the beginning of the simulation, they burn out.
  - `time_to_compile` (in milliseconds): The time it takes for a coder to compile. During that time, they must hold two dongles.
  - `time_to_debug` (in milliseconds): The time a coder will spend debugging.
  - `time_to_refactor` (in milliseconds): The time a coder will spend refactoring. After completing the refactoring phase, the coder will immediately attempt to acquire dongles and start compiling again.
  - `number_of_compiles_required`: If all coders have compiled at least this many times, the simulation stops. Otherwise, it stops when a coder burns out.
  - `dongle_cooldown` (in milliseconds): After being released, a dongle is **unavailable** until its cooldown has passed.
  - `scheduler`: The arbitration policy used by dongles to decide who gets them when multiple coders request them.
    The value must be exactly one of: `fifo` or `edf`.
    `fifo` means *First In, First Out*: the dongle is granted to the coder whose request arrived first.
    `edf` means *Earliest Deadline First* with deadline = `last_compile_start + time_to_burnout`.
- Each coder has a number ranging from `1` to `number_of_coders`.

- Coder number `1` sits next to coder number `number_of_coders`.
  Any other coder number `N` sits between coder number `N - 1` and coder number `N + 1`.

Reminder: All arguments are mandatory. Reject invalid inputs such as negative numbers, non-integers, or a scheduler other than fifo or edf.

About the logs of your program:

- Any state change of a coder must be formatted as follows:
  - `timestamp_in_ms X has taken a dongle`
  - `timestamp_in_ms X is compiling`
  - `timestamp_in_ms X is debugging`
  - `timestamp_in_ms X is refactoring`
  - `timestamp_in_ms X burned out`

  Replace `timestamp_in_ms` *with the current timestamp in milliseconds and* `X` *with the coder number.*
- A displayed state message should not be mixed up with another message.
- A message announcing that a coder burned out should be displayed no more than 10 ms after the actual burnout.
- Again, coders should avoid burning out!

Example of the expected log format:

```
0 1 has taken a dongle
1 1 has taken a dongle
1 1 is compiling
201 1 is debugging
401 1 is refactoring
402 2 has taken a dongle
403 2 has taken a dongle
403 2 is compiling
603 2 is debugging
803 2 is refactoring
1204 3 burned out
```

Precision requirement: Burnout logs must be displayed within 10 ms of the actual burnout time. Allow a minimal tolerance when testing, as hardware and OS scheduling may slightly affect measured timings.

Timing consideration: To reduce hardware impact on performance measurements, consider using CPU usage time instead of real-time clock when feasible. However, for this project, real-time measurements using gettimeofday() are acceptable and recommended for simplicity.

Chapter VI

Mandatory part

Program Name	`codexion`
Files to Submit	`Makefile, *.c, *.h in directory coders/`
Makefile	`NAME, all, clean, fclean, re`
Arguments	`number_of_coders time_to_burnout time_to_compile time_to_debug time_to_refactor number_of_compiles_required dongle_cooldown scheduler`
External Function	`pthread_create, pthread_join, pthread_mutex_init, pthread_mutex_lock, pthread_mutex_unlock, pthread_mutex_destroy, pthread_cond_init, pthread_cond_wait, pthread_cond_timedwait, pthread_cond_broadcast, pthread_cond_destroy, gettimeofday, usleep, write, malloc, free, printf, fprintf, strcmp, strlen, atoi, memset`
Libft authorized	`No`
Description	`Coders with threads and mutexes (C)`

The specific rules for the mandatory part are:

- Each coder must be represented by a thread (using `pthread_create`).
- There is one dongle between each pair of coders. Therefore, if there are several coders, each coder has a dongle on their left side and a dongle on their right side. If there is only one coder, there should be only one dongle on the table.
- To prevent coders from duplicating dongles, you must protect each dongle's state with a mutex (`pthread_mutex_t`). A condition variable (`pthread_cond_t`) may be used to manage waiting queues.
- **Dongle cooldown is mandatory**: after a coder releases a dongle, the dongle cannot be taken again until `dongle_cooldown` milliseconds have elapsed.
- **Fair arbitration is mandatory**: when multiple coders request the same dongle, the dongle must grant access according to `scheduler`.

With `fifo`, serve requests in arrival order.
With `edf`, serve the coder with the earliest burnout deadline (i.e., `last_compile_start + time_to_burnout`).

- The program must guarantee liveness: coders should not starve under `edf` scheduling, provided the parameters are feasible.
- A separate **monitor** thread must detect burnout precisely and stop the simulation. The burnout log must be printed within 10 ms of the actual burnout time.
- Logging must be serialized so that two messages never interleave on a single line (use a mutex to protect output).
- The simulation stops either when a coder burns out or when every coder has compiled at least `number_of_compiles_required` times.
- Your code must compile with `-Wall -Wextra -Werror -pthread`.
- You must implement a priority queue (heap) for FIFO/EDF scheduling (C89 has no standard library for this).
- All memory must be properly allocated and freed (no memory leaks).

Example of simulation run:

```
0 1 has taken a dongle
2 1 has taken a dongle
2 1 is compiling
202 1 is debugging
402 1 is refactoring
405 2 has taken a dongle
406 2 has taken a dongle
406 2 is compiling
606 2 is debugging
806 2 is refactoring
900 3 has taken a dongle
902 3 has taken a dongle
902 3 is compiling
1102 3 is debugging
1302 3 is refactoring
1505 4 burned out
```

This example illustrates the sequence of actions for multiple coders. Note how each "compiling" action is preceded by two "has taken a dongle" lines, and how the "burned out" message appears at the moment a coder misses their deadline.

Chapter VII

Readme Requirements

A `README.md` file must be provided at the root of your Git repository. Its purpose is to allow anyone unfamiliar with the project (peers, staff, recruiters, etc.) to quickly understand what the project is about, how to run it, and where to find more information on the topic.
The `README.md` must include at least:

- The very first line must be italicized and read: *This project has been created as part of the 42 curriculum by <login1>[, <login2>[, <login3>[...]]].*
- A "**Description**" section that clearly presents the project, including its goal and a brief overview.
- An "**Instructions**" section containing any relevant information about compilation, installation, and/or execution.
- A "**Resources**" section listing classic references related to the topic (documentation, articles, tutorials, etc.), as well as a description of how AI was used — specifying for which tasks and which parts of the project.

- **Additional sections may be required depending on the project** (e.g., usage examples, feature list, technical choices, etc.).

Any required additions will be explicitly listed below.

For this project, the `README.md` must also include:

- A "**Blocking cases handled**" section describing all the concurrency issues addressed in your solution (e.g., deadlock prevention and Coffman's conditions, starvation prevention, cooldown handling, precise burnout detection, and log serialization).
- A "**Thread synchronization mechanisms**" section explaining the specific threading primitives used in your implementation (pthread_mutex_t, pthread_cond_t, custom event implementation) and how they coordinate access to shared resources (dongles, logging, monitor state). Include examples of how race conditions are prevented and how thread-safe communication is achieved between coders and the monitor.

`Your README must be written in English.`

Chapter VIII

Submission and peer-evaluation

Turn in your assignment in your `Git` repository as usual. Only the work inside your repository will be evaluated during the defense. Don't hesitate to double-check the names of your files to ensure they are correct.

Recode instructions

During the evaluation, a brief **modification of the project** may occasionally be requested. This could involve a minor behaviour change, a few lines of code to write or rewrite, or an easy-to-add feature.

While this step may **not be applicable to every project**, you must be prepared for it if it is mentioned in the evaluation guidelines.

This step is meant to verify your actual understanding of a specific part of the project. The modification can be performed in any development environment you choose (e.g., your usual setup), and it should be feasible within a few minutes — unless a specific time frame is defined as part of the evaluation.
You can, for example, be asked to make a small update to a function or script, modify a display, or adjust a data structure to store new information, etc.

The details (scope, target, etc.) will be specified in the **evaluation guidelines** and may vary from one evaluation to another for the same project.