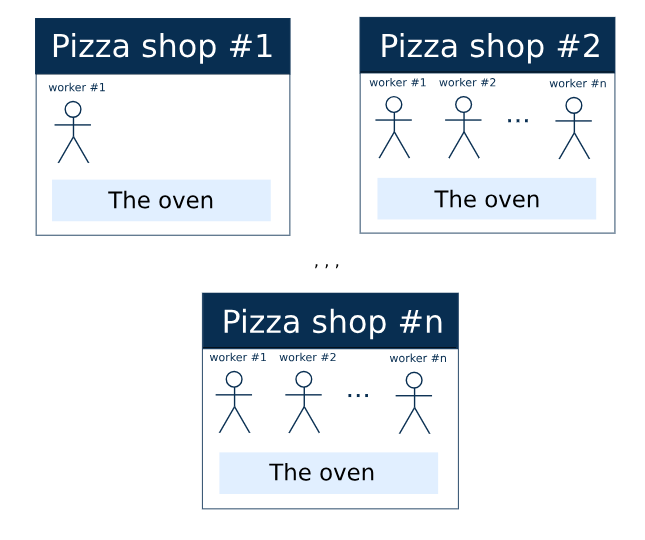
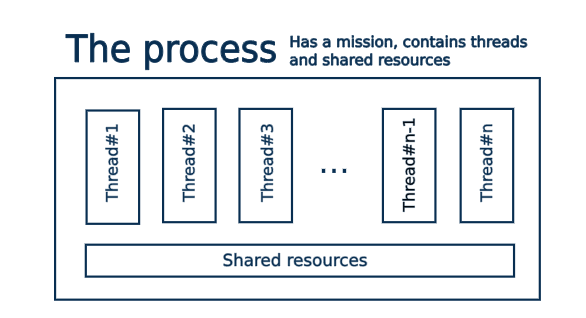
Imagine that you come to a food court during lunchtime, and you see a line of pizza shops there. Each shop's mission is to sell pizza and each of them has several workers to accomplish this. So, each worker's purpose is to sell pizza, but they can't sell it by themselves without the equipment provided by the shop. Likewise, any pizza shop can't sell anything without its workers. That is, there has to be at least one worker in a pizza shop to do the job. Meaning, the workers rely on the shop's equipment to do their jobs, just as the shop depends on these workers to function.

It's similar to how a computer runs applications and manages multitasking and parallel execution. To delve deeper and understand it better, let's explore the concepts such as *processes* and *threads*, drawing parallels between these computer science concepts and the dynamics of a pizza shop.



**Process**

A **process**is a self-contained unit of execution that includes everything necessary to complete its tasks. In short, a process is the container for its threads, encompassing all necessities for their operation and their shared resources. It's cheaper to arrange access to shared resources once than to do so each time a new thread is spawned. Every process must have at least one thread, as they perform all the work. There is no such thing as a thread without its process or a process without at least one thread.



If we look at the pizza business, a single *pizza shop* would serve as an analogy for the *process*. It provides all the environment and equipment required for a worker to perform their job. Equipment is expensive, so it's cheaper and more efficient when workers share it. There is no need for each worker to acquire personal equipment. On the other hand, a shop cannot function without its workers; it is crucial to have at least one worker, as, without them, all the equipment would remain idle. Together, these elements constitute the process of making and selling pizza.

**Thread**

In computer science, a **thread**of execution is a sequence of instructions within a process that can be scheduled and run independently. Each thread has its own executor, which can manage only one thread at a time. Multiple threads within the same process can operate *concurrently*(switching between tasks) or in *parallel*(simultaneously, if multiple executors are available), depending on how they are scheduled and the resources available.

To understand what the term *thread* means, think of employees in a pizza shop. They perform various tasks according to their job descriptions, following the rules set by the shop and utilizing shared resources provided by the shop.

In this analogy, workers in a pizza shop represent thread executors, and the tasks they perform are the threads within the pizza shop "process".

**Concurrency and parallelism**

**Concurrency** and **parallelism** are key concepts in computing that describe different methods for handling multiple tasks efficiently.

* **Concurrency**: Imagine a chef in a kitchen preparing two dishes simultaneously. The chef starts by chopping vegetables for a salad, then while those vegetables are chilling, begins grilling chicken for another dish. The chef isn't working on both dishes at the exact same moment but switches between tasks, advancing both dishes without completing one before starting the other. This is concurrency, which involves managing multiple tasks by alternating between them to maximize efficiency.
* **Parallelism**: Now picture a large kitchen where two chefs are working at the same time, one grilling chicken and the other preparing a salad. Each chef works independently on their dish, and both dishes are being prepared at the same time. This scenario exemplifies parallelism, where multiple tasks are truly happening simultaneously, each handled by separate resources.

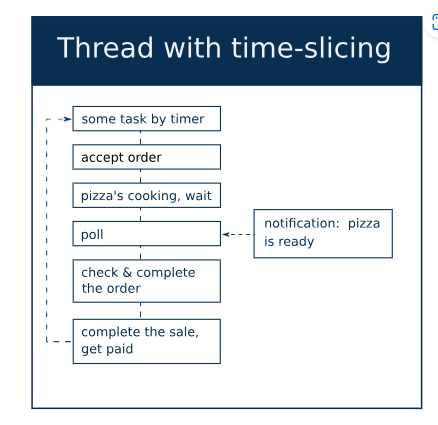
This should help clarify the distinction between *concurrency*, which involves switching between tasks to give the appearance of simultaneous progress; and *parallelism*, where tasks genuinely occur at the same time, utilizing multiple resources. Both concepts aim to optimize the execution time and resource utilization in multitasking environments, but they achieve this in different ways. Concurrency is about dealing with many tasks through quick switching, while parallelism is about doing many tasks exactly at the same time.

**Internal or lightweight concurrency**

In some cases, workers (threads) can perform multiple roles within the same pizza shop (process). For instance, a worker might serve as both a cashier and a cook at different times. This kind of concurrency isn't about multiple workers doing tasks simultaneously, but about a single worker switching between roles efficiently. These roles typically involve tasks that are quick and do not demand significant time or shared resources, classifying them as **lightweight**.

If tasks are lightweight and require minimal shared resources except the executor's time and attention, there is no need to run them in separate threads. It is more efficient to manage their concurrent execution through *time-slicing within a single thread*, where the executor switches between tasks quickly enough that they appear to be happening simultaneously. This form of concurrency is often referred to as **internal** or lightweight due to the minimal nature of the tasks involved.

The following image illustrates an example of a worker's thread featuring lightweight concurrency through time-slicing:



**Conclusion**

* **Processes** are like pizza shops. They serve as containers for worker's threads, shared resources, and parameters necessary for completing tasks. Every process must have at least one thread.
* **Threads** are independent units of execution within a process; they can operate concurrently or in parallel with one another.
* Concurrent tasks that compete only for the executor's time and don't require a lot of resources can run concurrently within the same thread. These tasks are called *lightweight*, and this type of concurrency is known as **internal**or **lightweight**concurrency. This is more resource-efficient than creating new threads for each task. Execution within threads can be synchronous or asynchronous but never parallel.

By understanding these concepts through the pizza shop analogy, you can better grasp how processes and threads work together in computer systems.

What does the term "process" mean in computer science?

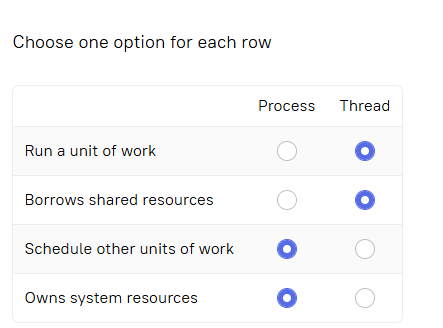
an independent unit containing an execution environment, shared resources, and threads of its tasks

## Process's and thread's properties

Mark suitable properties for each kind of container.

Hint by

Process is the set of instructions to get something done, it delegates the tasks to the Threads, which can use resources provided.



Which problems does the process (restaurant) concern?

The correct options, based on the metaphor and hint, are:

* ✅ **arranging access to scarce shared resources (ingredients)**
* ✅ **organizing the runtime environment (the kitchen for pizza workers)**
* ✅ **supporting inter-thread communications (communications between pizza workers)**

### Explanation:

* A **process** in computing manages **resources**, sets up an **execution environment**, and coordinates **communication between threads**—just like a restaurant manager coordinates access to ingredients, manages the kitchen environment, and ensures workers communicate.
* The **hint** ("Delegate and watch, staying useless") suggests that the process itself doesn’t do the work (like baking pizzas) but **delegates** it (to threads), which eliminates:
  + ❌ **making all the work by itself (baking pizzas)**

What does the term "thread" mean in computer science?

Hint by

the process can be divided into n parrallel subprocesses

✅ **Correct answer:**  
**A stream of instructions inside a process that can be scheduled and run independently**

### Explanation:

* A **thread** is the **smallest unit of execution** within a process.
* Threads share the same memory space of their parent process but can **run independently** and in **parallel**, allowing for concurrent execution.
* The hint about dividing a process into n parallel subprocesses refers to **multithreading**, where a single process can have multiple threads.

## Internal concurrency

 Report a typo

Which of the following tasks can run concurrently inside the same thread?

**Tip:**

Think of shared resources. If there is no need for them or no need to share them, then tasks can run concurrently inside the same thread.

**Hint**by

You can talk while you walk ;-)

✅ **Correct options:**

* **Slicing pizza and telling the cook to make one more pepperoni**
* **Serving customers and chatting with friends on the phone**

### Explanation:

These tasks **can run concurrently inside the same thread** because:

* They **don’t need exclusive access** to shared resources.
* They represent actions that can be **interleaved** or handled without strict synchronization.

❌ **Incorrect options:**

* **Getting payments from two customers for the same slice of pizza**  
  → This involves a **shared resource (the same slice)** and would require coordination—**not safe** in a single thread without locking.
* **Cooking in two different pizzerias at the same time**  
  → Implies **physically being in two places at once**, which a single thread (like a person) cannot do.

Where does the internal concurrency take place?

Hint by

"If tasks are lightweight and don't require access to any shared resources except the executor's time and attention, there's no need to run them in different threads. It's cheaper to arrange their concurrent execution with time-slicing inside one thread. "

Select one option from the list

inside the same thread

Which roles do threads play in a program?

**Hint**

Review: <https://hyperskill.org/learn/step/9420#thread> Think of a thread as an employee in a job like mcdonalds (I'll be using the example of a cook).- Can the cook keep the results of their execution? (i.e. can the cook keep their own food after cooking it? or does the food get sent to the customer?)- Does the cook get access to scarce shared resources? (i.e. does the cook have access to shared cooking utilities such as a frying pan? or does the cook have to bring their own cooking utilities such as a frying pan?)- Can a cook control its own schedule time? (or does the manager control the cook's schedule time?)- Is the cook responsible for executing tasks? (i.e. Is the cook responsible for the task assigned to it which is cooking food? or is the waiter the one responsible for cooking food?)

Select one or more options from the list

✅ **Correct answers:**

* **getting access to scarce shared resources**
* **executing the tasks**

### ❌ Incorrect:

* **keeping the results of the execution after completion**  
  → Threads usually **don’t retain results independently**; results are passed back to the process or stored in shared memory.
* **control its schedule time**  
  → Thread **scheduling is controlled by the OS or thread manager**, not the thread itself.

### Summary (Cook Analogy):

* The **cook (thread)** executes the **task (cooking)** ✅
* The cook uses **shared tools (frying pans, ingredients)** ✅
* The cook **doesn’t decide the schedule** ❌
* The food **goes to the customer (not kept by cook)** ❌