Threads

A **thread** is the smallest unit of execution, a sequence of instructions, that the operating system can schedule and run on a CPU.

A **process** is an executing program with its own memory space and resources, and it contain one or more **threads**.

Processes have separate memory space and IO resources, threads of the same process share the same memory space and IO resource.

Running a Thread

Threads need to use the CPU to be executed. A thread running on a CPU have two different ways to progress:

- **Concurrency**: Multiple threads switch turns to make progress (one or more CPU cores)
- **Parallelism**: Multiple threads literally run in parallel (require multi-core CPUs)

When the OS switches to current running thread to a different one, it is called a **context switch**. Switching involves saving the current thread's registers, program counter, and other CPU states; and loading in the next thread's state.

Threads that have been switched out need to **wait**.

A thread that can guarantee progress in its program and pass it own to others is called to be **live**.

> "Eventually, something good will happen"
>
> - Liveness Property

Thread States

Threads have multiple states that indicates what should the OS do with it.

- **New** → being created, waiting to be admitted by the OS scheduler.
- **Ready** → waiting to be schedule on the CPU.
- **Running** → currently executing.
- **Blocked** / **Waiting** → paused, waiting for I/O or synchronization.

- **Terminated** → finished execution, waiting to have its resources free by the OS.

Threads that never terminate are called **zombie threads**.

Coordination

A thread that waits for a resource or another thread is called **synchronous** to that thread. Otherwise, if it does not wait and moves on to do more work is called **asynchronous** relative to that thread.

Threads share resources and memory inside a process, each thread needs to coordinate (or being synchronized) with other threads to execute properly. It needs Synchronization primitives.

You may check out the Synchronization primitives guide for a closer look.

Mutual Exclusion Locks (Mutexes or Locks) enforce synchronization to all threads that have it. The first thread that acquire the lock have exclusive right to a section of code and the memory it may access.

This is called a **critical section** of the code, and it forces others to **pause** execution if they acquire the lock in their run time until the

Thread that try to get the lock and are said to have **lock contention**. Any other resource is called **thread contention** and are forced to wait.

Threads needs to make memory consistent also across caches when they update a variable. These functionality are provided with atomic variables ** through memory barriers.

Concurrency Bugs

Errors while designing for multi-threading is hard to debug, for it has **emergence** properties. One thread interacting with another can create emergent and interesting behaviors.

> A multi-threaded program is more than the sum of its parts.

Often, these are symptoms of bad programming. Check out Concurrency Bugs for a list.

Here are the three major concurrency bugs:

- When two threads overwrite the progress of another unexpectedly, it is called a **race condition**. Broadly, it is a bug where sequence of threads lead to inconsistent or unexpected behavior.

- When a thread that has acquired the mutual lock crashes, the rest of the threads that waits for it will halt forever. This is called a **deadlock**.
- When a thread will never get access to a needed resource (a file for example) while other threads constantly accesses it, that thread will wait forever. This thread is under **starvation**.

Once all concurrency bugs are deemed squashed, it is called **thread-safe**.

Operations and variables that are thread-safe are called **atomic**. Executing or using them are guranteed to be safe to use.

Concurrency Design Patterns

Programming in multi-threading can be difficult because of said emergent properties, but there are strategies to mitigate this.

First, do problem analysis to lay down what is the nature of the problem, and what should the solution account for.

Then, use design architecture (or design patterns) help make programming easier laid out for it be to coded, to debug, to reason about.

These are some useful design pattern terminology:

- **Worker Thread** - a thread or object that performs tasks
- **Thread pool** - a group of reusable worker threads
- **Dispatcher** (or **Scheduler**) - code that assigns work from a queue to workers
- **Load Balancer** - code that distributes work evenly among workers
- **Pipeline** - a sequential order of threads where one's result is the other's task.

Check out the next guides:

- Concurrency Problem Analysis
- Concurrency Design Architecture