

Basic Definitions

multiple tasks.

Concurrency Cheat Sheet

A quick reference guide to concurrency concepts and tools, covering threads, processes, synchronization primitives, and common concurrency patterns

Deals with managing multiple

about structure. Tasks may not

necessarily run simultaneously.

Deals with actually executing

multiple tasks simultaneously.

Requires multiple cores or processors. It's about

Parallelism enhances

execution.

concurrency.

tasks at the same time. It's



Fundamentals of Concurrency

multiple tasks seemingly simultaneously.

Process: An instance of a program being

executed, with its own memory space.

Concurrency vs Parallelism

Concurrency

Parallelism

Concurrency

parallelism.

enables

Benefits of Concurrency

- Improved Performance: Parallel execution can reduce overall execution time.
- Responsiveness: Keeps the application responsive by offloading long-running tasks to background threads.
- Resource Utilization: Makes better use of available CPU cores.

process, sharing the same memory space. Context Switching: The process of switching the CPU's focus between different threads or processes.

Thread: A lightweight unit of execution within a

Concurrency: The ability of a program to execute

Parallelism: The actual simultaneous execution of

Threads and Processes

Thread Management

Process Management

Creating Threads Thread	Use threading libraries (e.g., threading) in Python, (java.lang.Thread) in Java) to create and start new threads. New -> Runnable -> Running ->	Creating Processes	Use process creation mechanisms (e.g., multiprocessing in Python, fork() in C) to spawn new processes.	 Threads: Lightweight, share memory space, faster context switching, but susceptible to race conditions. Processes: Heavyweight, isolated memory space, slower context switching, more robust.
Lifecycle	Blocked/Waiting -> Terminated.	Inter-Process Communication	Use techniques like pipes, message queues, shared	
Priorities	priorities, but relying on them for correctness is not recommended.	(IPC)	memory, and sockets for communication between processes.	Choose threads for I/O-bound tasks and processes for CPU-bound tasks to maximize concurrency and parallelism.
Joining Threads	Waiting for a thread to complete its execution using a join() method.	Process Isolation	Processes have their own memory space, providing isolation and preventing direct memory access from	

other processes.

Synchronization Primitives

Locks and Mutexes		Semaph	
Mutex (Mutual Exclusion)	A synchronization primitive that provides exclusive access to a shared resource. Only one thread can hold the mutex at a time. Prevents race conditions.	Definiti	
Lock	Often used interchangeably with		
(Similar to Mutex)	mutex, providing exclusive access.	Usage	
Usage	Acquire the lock before accessing the shared resource, and release it afterward.		
Example		Example	
(Python)	<pre>import threading</pre>	(Pythor	
	<pre>lock = threading.Lock()</pre>		
	with lock:		
	# Access shared		
	resource		

Semaphores

Definition	A synchronization primitive that controls access to a shared resource using a counter. Can allow more than one thread to access the resource concurrently (up to the counter's limit).
Usage	Initialize the semaphore with a counter value. Threads decrement the counter when acquiring the resource and increment it when releasing.
Example (Python)	<pre>import threading semaphore = threading.Semaphore(2) # Allow 2 threads concurrently with semaphore: # Access shared resource</pre>

Condition Variables

Threads vs. Processes

Definition	A synchronization primitive that allows threads to wait for a specific condition to become true. Always used in conjunction with a lock.
Usage	Threads acquire the lock, check the condition, and wait if the condition is false. Another thread signals the waiting thread(s) when the condition becomes true.
Methods	<pre>wait() , notify() , notify_all()</pre>
Example (Python)	<pre>import threading condition = threading.Condition() with condition: condition.wait() # Wait for a signal condition.notify() # Signal another thread</pre>

Common Concurrency Patterns

Producer-Consumer Pattern

Producers generate data and place it into a shared buffer. Consumers retrieve data from the buffer and process it. Synchronization is crucial to prevent race conditions and buffer overflows/underflows.

Use locks and condition variables to manage access to the buffer and signal when data is available or space is available.

Reader-Writer Lock

Description	Allows multiple readers to access a shared resource concurrently, but only one writer at a time. Improves performance when reads are much more frequent than writes.
Implementation	Can be implemented using a combination of mutexes and condition variables.
Prioritization	Reader-preference or writer- preference can be implemented to control fairness.

Thread Pool

A pool of worker threads that are created at the start of the program and reused to execute multiple tasks. Reduces the overhead of creating and destroying threads for each task.

Use a queue to submit tasks to the thread pool. Worker threads retrieve tasks from the queue and execute them.

Asynchronous Programming

Definition	A programming paradigm that allows tasks to be executed independently without blocking the main thread. Improves responsiveness and scalability.
Techniques	Use asynchronous constructs like futures, promises, async/await, and callbacks.
Benefits	Improved responsiveness, scalability, and resource utilization.