UNSW SYDNEY

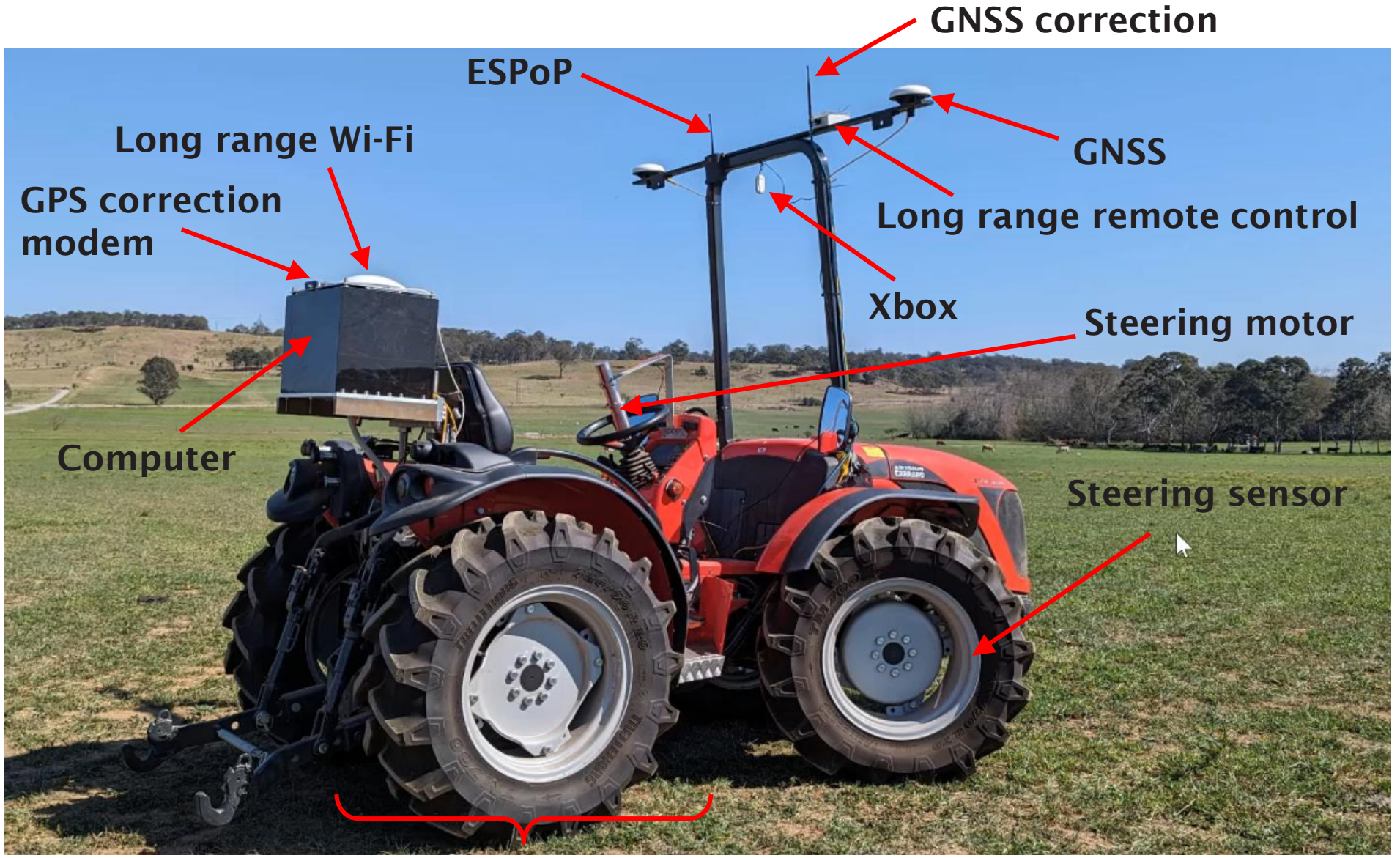
Introduction to Multithreading

A/Prof Shiyang Tang

MTRN3500

Computing Applications in Mechatronics Systems

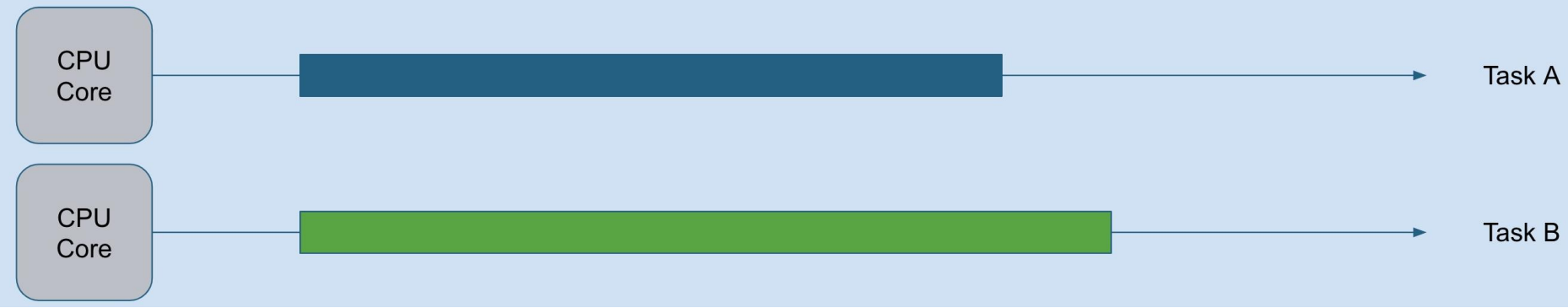
UGV Systems


GNSS correction
ESPoP
Long range Wi-Fi
GPS correction modem
GNSS
Long range remote control
Xbox
Steering motor
Computer
Steering sensor
Drive system
Other systems: LiDAR, IMU, camera

Multithreading in Single/Multi-Core Processors

Multi-core processor with multithreading

- A multi-core CPU has multiple physical cores that can each execute a thread at the same time.
- This allows for **true parallelism** – threads can literally run simultaneously on different cores.

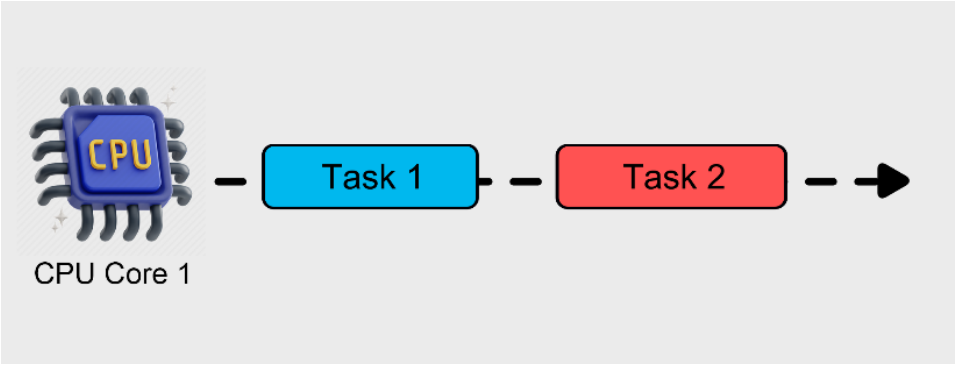


Single-core processor with multithreading

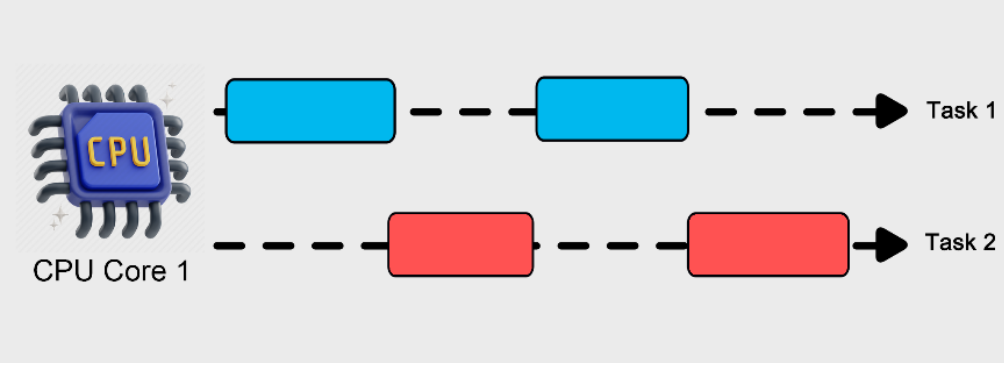
- A single-core CPU has only **one** physical execution unit; when multiple threads are created, the processor cannot truly run them at the same time.
- It uses **time slicing**: the CPU rapidly switches between threads, giving the illusion of parallelism.
- This is called **concurrency**, not true parallelism.

Concurrency vs Parallelism

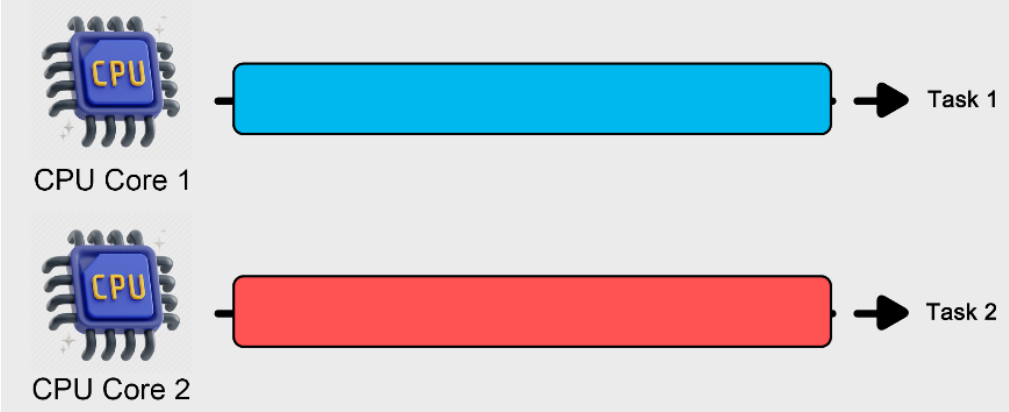
Not concurrent, not parallel



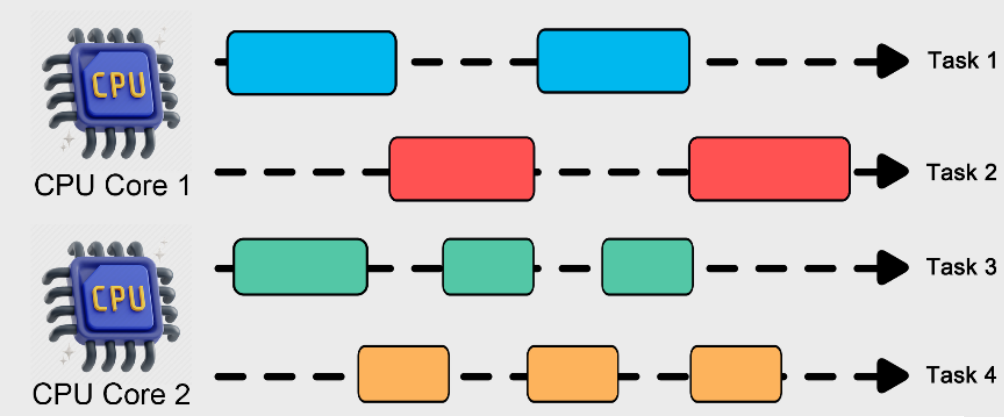
Concurrent, not parallel



Not concurrent, but parallel



Concurrent and parallel

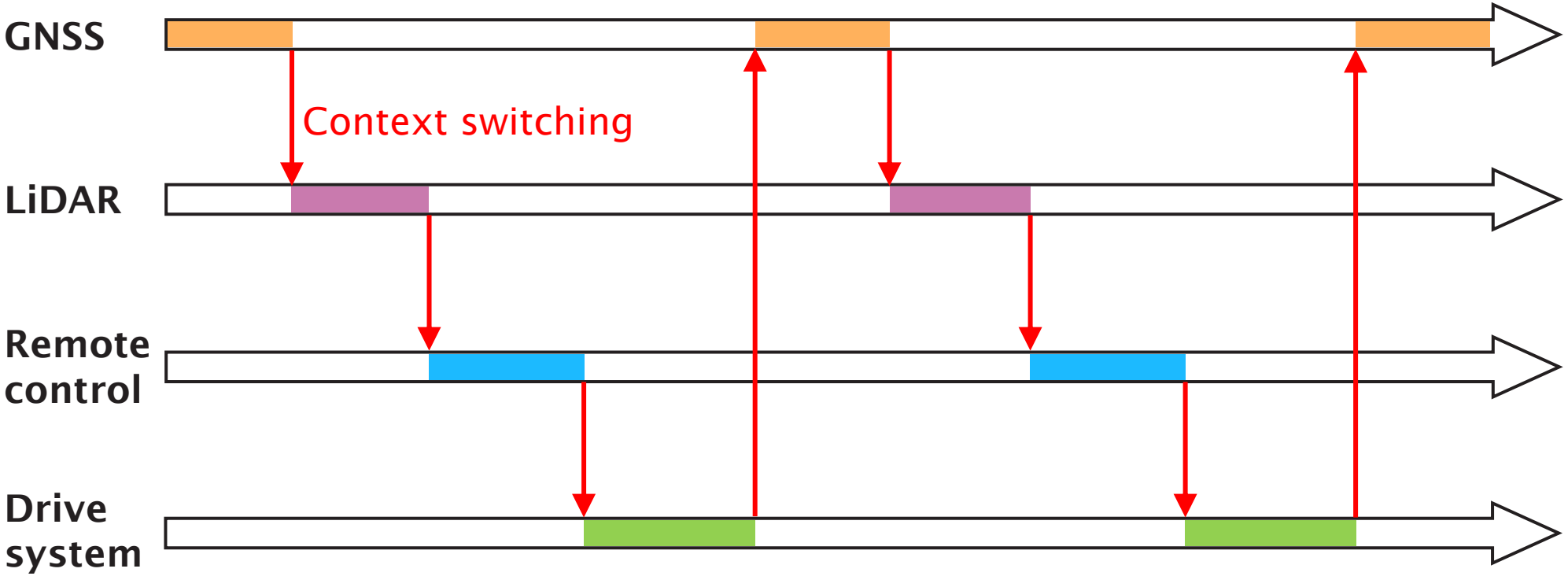


https://bytebytego.com/guides/concurrency-is-not-parallelism/

Context Switching

Context switching: the process of a processor switching from executing one thread to another.

A processor core can only actively run one thread at a time; the OS needs to "pause" the current thread, save its state, and then "load" the state of another thread so it can continue running. The saved information is called the **context**.



This switching happens very quickly (thousands or millions of times per second), so it feels like multiple programs are running simultaneously.

Context Switching

How does it work?

1. Save the current state: The registers, program counter (where it left off), and other information of the running thread are stored.
2. Load the new state: The processor loads the saved information of the next thread.
3. Resume execution: The processor continues executing the new thread as if it had never been interrupted.

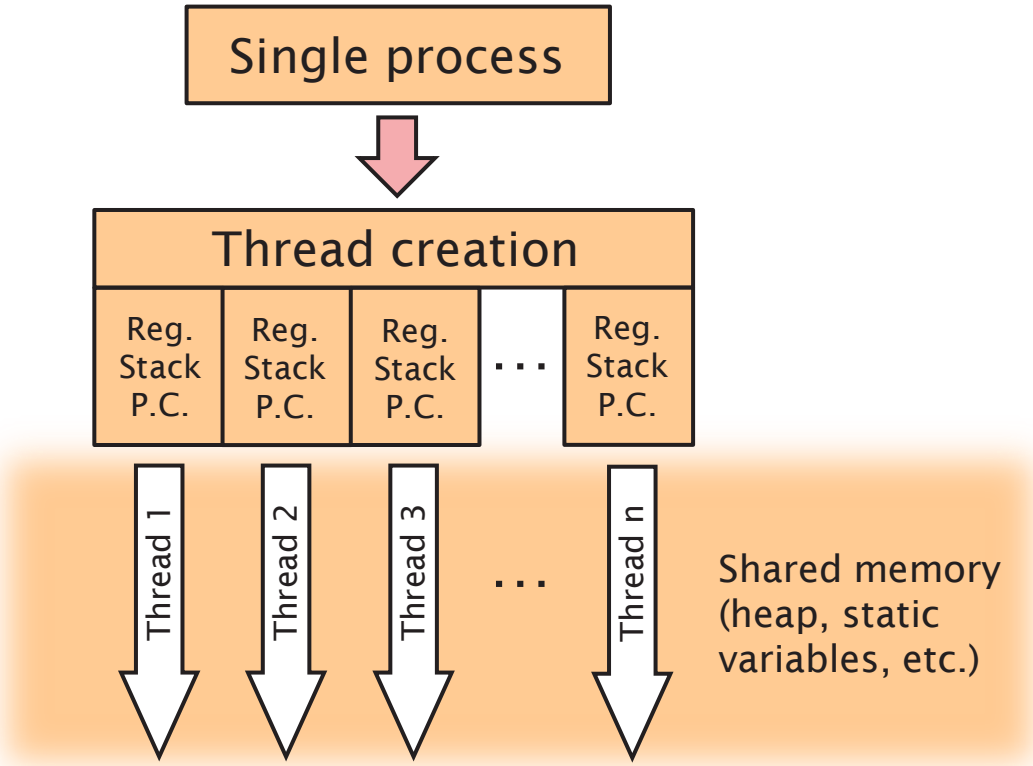
Context switching is not free:

- It takes time to save and restore states.
- Too many switches cause **overhead**, reducing performance.

Multithreading vs Multitasking vs Multiprocessing

Multithreading: multiple threads inside **one process**, sharing memory, running concurrently (time slicing *across threads inside one process*) or in parallel.

Analogy: One dish (process) is being prepared by a team of assistants (threads) working together, sharing the same kitchen tools.



Race condition: Two threads write to the same memory simultaneously.

Inconsistent state: One thread reads while another is writing.

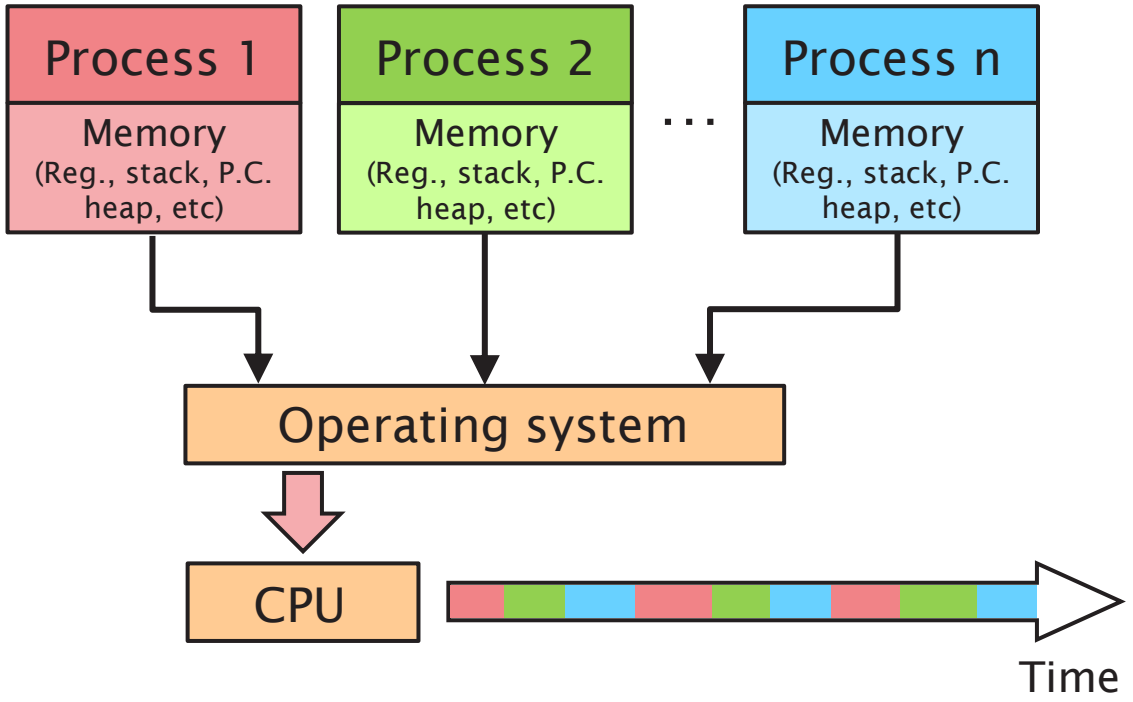
A **mutex (mutual exclusion)** ensures that *only one thread at a time* can access a shared resource.

Multithreading vs Multitasking vs Multiprocessing

Multitasking: multiple processes sharing a CPU. Time slicing *across independent processes*.

Analogy: One cook making many dishes by rapidly switching between them.

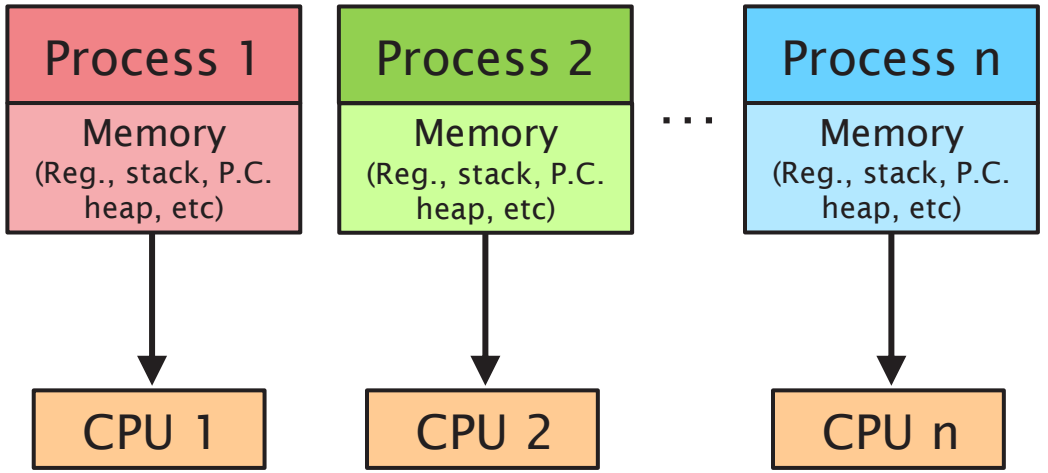
Both multitasking and multithreading on a single-core CPU use time slicing (context switching) to simulate parallelism, but they differ in what is being sliced and how resources are shared.



Multithreading vs Multitasking vs Multiprocessing

Multiprocessing: multiple CPUs/cores running processes simultaneously.

Analogy: Many cooks in the kitchen, each making a dish simultaneously (true parallelism).



Multitasking and Multiprocessing

- Processes cannot see each other's memory.
- Inter-Process Communication (IPC) is needed if they need to communicate → context switching is expensive

Multithreading vs Multitasking vs Multiprocessing

Aspect	Multithreading	Multitasking	Multiprocessing
Unit of Execution	Thread	Process	Process
Data Segment (globals/statics)	Shared among all threads in the process.	Private per process.	Private per process.
Heap (dynamic memory)	Shared heap: threads can read/write the same dynamically allocated memory.	Private per process.	Private per process.
Stack	Each thread has its own stack (separate local vars & call history).	Private per process	Private per process.
Registers	Each thread has its own registers and program counter.	Each process has its own register set and program counter.	Each process has its own register set and program counter.
Isolation	Weak: threads can overwrite each other's data if not synchronised.	Strong	Strong
Communication	Direct but requires synchronisation (mutex, monitor, semaphore).	IPC	IPC
Context Switch Cost	Low	High	High
Parallelism	Concurrency on single core; parallelism on multiple cores.	Concurrency via time slicing (single core).	True parallelism

C++/CLI for Threading

In C++/CLI, threading is managed using the .NET Framework, especially from the `System::Threading` namespace.

Key Classes for Threading in C++/CLI:

Class	Description
`Thread`	Creates and manages a new thread
`ThreadStart`	Delegate that defines the entry method for a thread
`ThreadPool`	Allows reuse of threads for short-lived tasks
`Monitor, lock`	Synchronisation primitives (to avoid race conditions)
`Mutex, Semaphore`	Advanced thread coordination tools

Challenges in UGV

Single processor → only one thread executes at a time. Responsiveness relies on **time slicing** and good scheduling, not true parallelism.

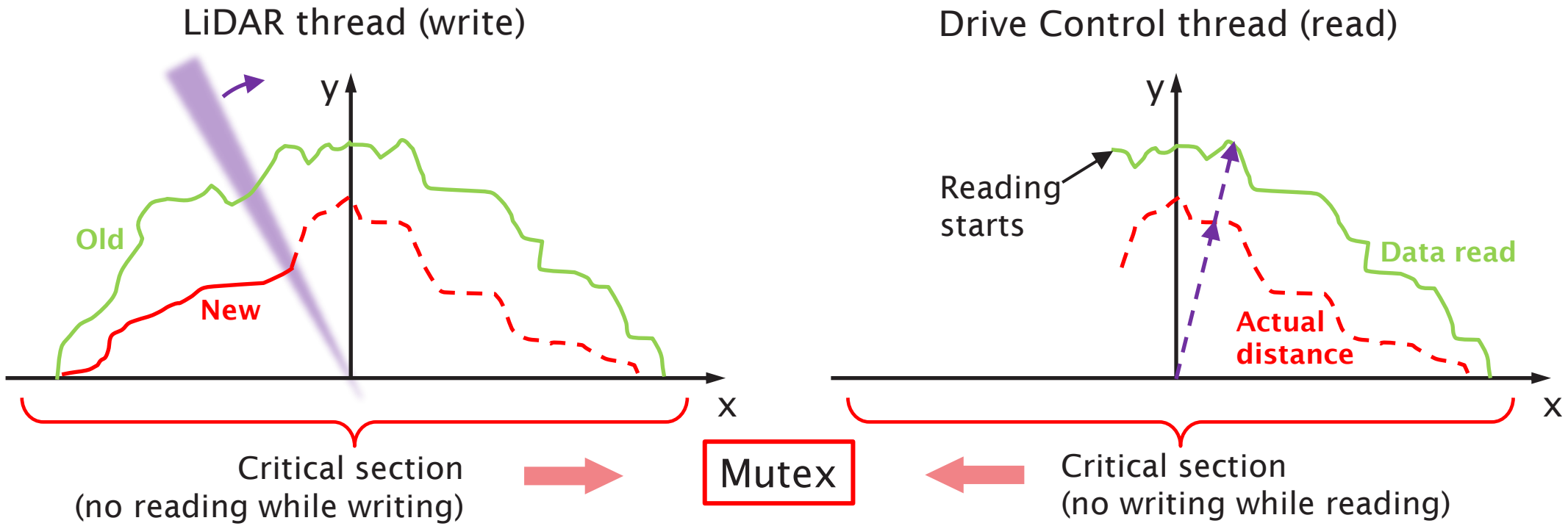
Scalability → needs to add new subsystems (e.g., Camera) without rewriting your main loop.

Maintainability → needs to "plug/unplug" threads easily.

Responsiveness → no subsystem should hog the processor; all should get fair attention.

Multithreading

Race condition: e.g., if both LiDAR and Drive Control threads update the UGV's position buffer.

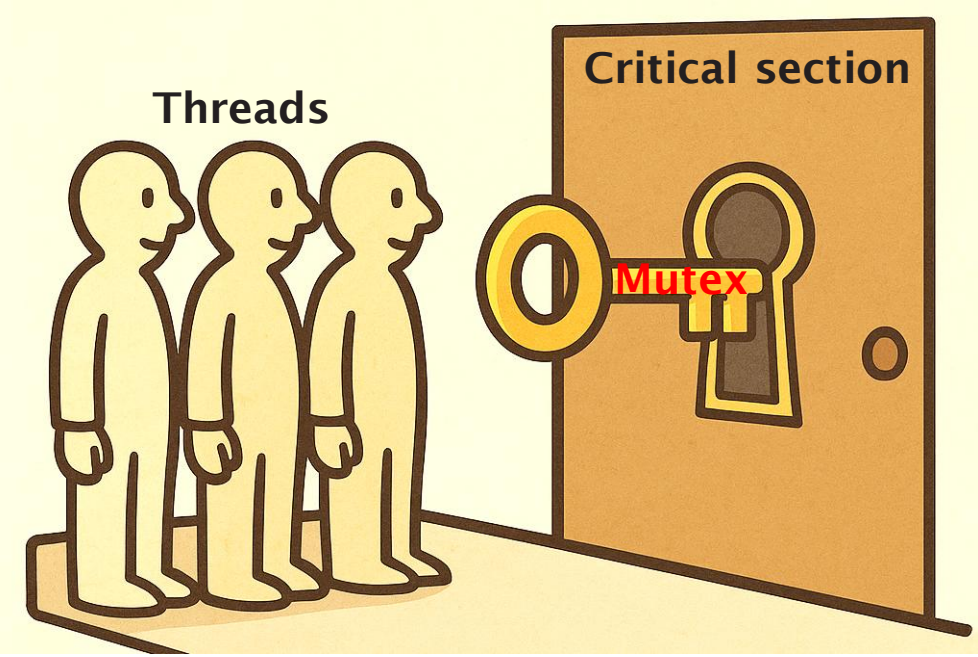


Mutex

A **mutex** (short for mutual exclusion) is a synchronisation primitive used in multithreading to ensure that only one thread at a time can access a shared resource or critical section of code.

It prevents *race conditions*, where multiple threads try to read/write the same data simultaneously.

- A mutex acts like a **key**.
- When a thread wants to enter a critical section, it must take the key.
- If another thread already has the key, the requesting thread must wait until the key is returned.
- Once the first thread finishes, it returns the key, and another waiting thread can take it.

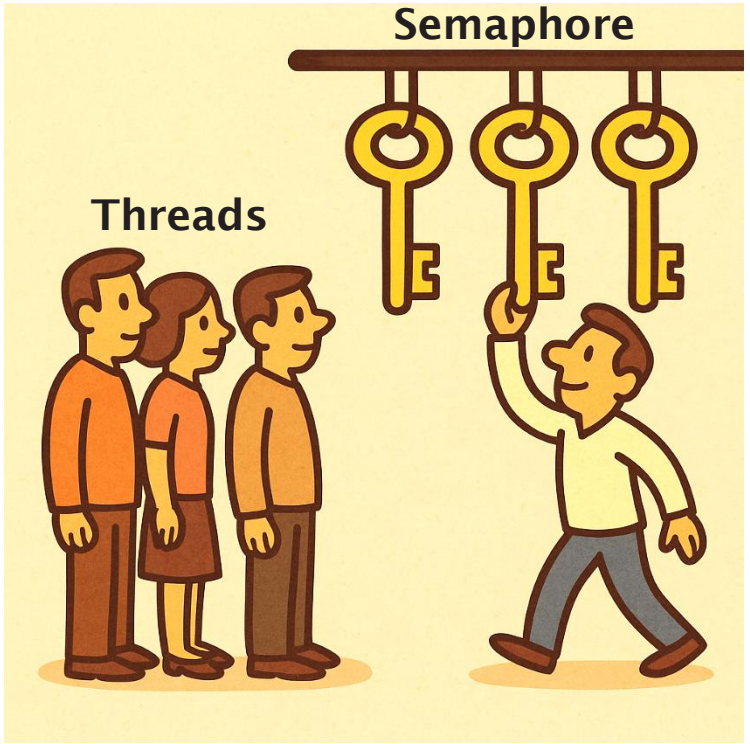


Semaphore

A **semaphore** is a synchronisation primitive used in concurrent programming to control access to shared resources. It acts like a counter for available permits – each permit represents permission for one thread to enter the critical section.

Think of semaphore as **a bunch of keys**:

- If a thread takes a key (decrement counter), it may enter.
- If no keys are left, the thread must wait.
- When the thread finishes, it returns the key (increment counter).



Monitor

A **monitor** is a high-level synchronisation construct that provides both:

1. Mutual exclusion (like a mutex – only one thread can execute inside the monitor at a time).
2. Condition synchronisation (using wait and signal/notify, so threads can wait for certain conditions and be woken up when they're met).

You can think of a monitor as a **room with one key and a bell**:

- Only one thread with the key can enter (mutual exclusion).
- Inside, threads can wait if a condition isn't satisfied.
- Another thread can ring the bell (signal) to wake waiting threads when the condition is met.

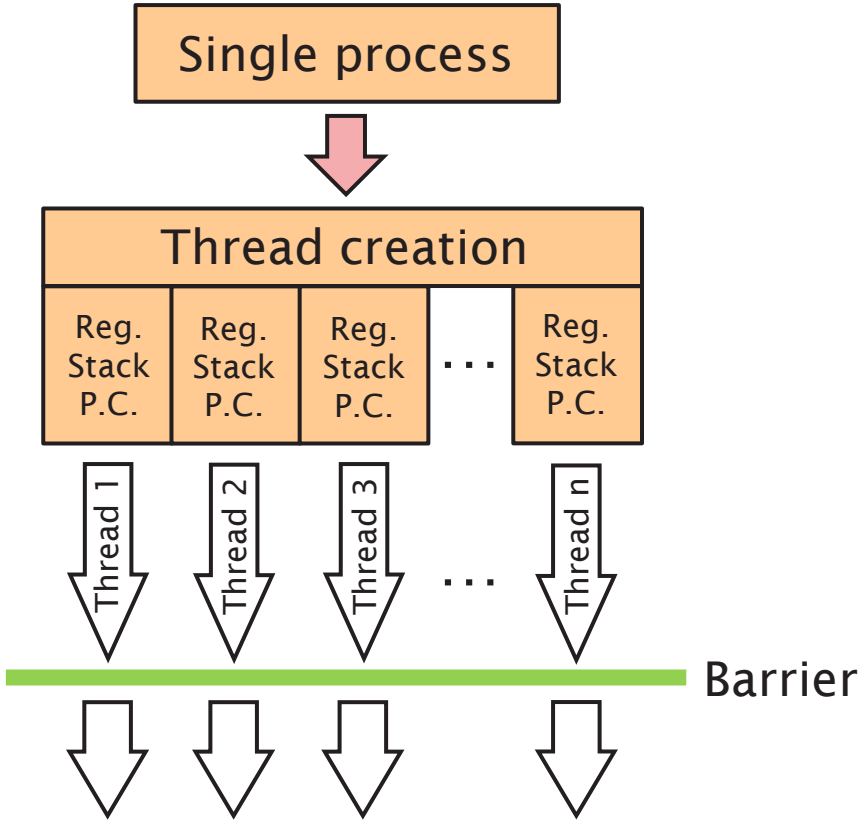
	Mutex	Semaphore	Monitor
Analogy	One key for one room	Multiple keys for limited resources	A room with one key + a bell for signalling
Mutual Exclusion	Yes (one thread at a time)	Yes; Limited number at a time	Yes (one thread inside monitor at a time)
Condition Sync.	No	No	Yes (threads can wait and be signalled/notified)
Blocking Behaviour	Threads block if mutex is already locked	Threads block if counter = 0	Threads block if monitor is busy, or if waiting for a condition
Use Cases	Protect one shared resource (e.g., no reading of LiDAR while scanning and writing)	Limit access to a pool of resources	Complex coordination (e.g., Drive Control must wait until both LiDAR and GNSS are ready.)

Thread Barriers

A **thread barrier** is a synchronisation primitive used to make a group of threads wait until all of them reach the same point in execution, before any are allowed to continue.

Think of it like a **checkpoint**:

- Threads arrive at the checkpoint.
- Each one must wait there until *everyone* in the group has arrived.
- Once all threads are at the barrier, they are released together to continue.



Pre-emptive Scheduler

A **pre-emptive scheduler** is part of the OS that decides which thread or process runs next on the CPU. Pre-emptive means the OS can interrupt (pre-empt) a running thread at any time and switch to another – ensures that no single thread can hog the CPU.

Scheduling is based on policies like:

- Time slice / quantum (round-robin scheduling)
- Priority levels (higher-priority threads get more CPU time)
- Fairness (all threads get a chance to run)

Single-core CPUs: The scheduler uses time slicing + pre-emption to switch rapidly between threads.

Multi-core CPUs: The scheduler decides which threads map to which cores and when.

Thread priorities: The scheduler can assign more CPU time to critical threads.

For example, in a UGV, pre-emptive scheduler ensures:

- LiDAR doesn't hog the CPU with heavy point cloud processing.
- GNSS gets periodic CPU time for updates.
- Drive Control always gets scheduled quickly (high priority) for safety.
- If LiDAR takes too long, the scheduler pre-empts it and gives CPU to Drive Control.