# Systems Programming Coursework 2

Robert De Venny (2892269d)

## Introduction

The purpose of this report is to outline the concurrency, synchronisation, and memory management features of 3 programming languages. This is with the aim to provide a recommendation for an approach for writing the software for the new *SpaceY* satellites. These satellites have large numbers of sensors gathering a diverse array of information which is required to be processed by the software. This primary requirement is why the language chosen for the implementation of the satellite software must have effective, robust concurrency, synchronisation, and memory management features to ensure the accuracy of the collected data. This report will cover the evaluation of C, Python and Java.

## C Overview

~~C is a lower-level language more associated with operating systems and system software rather than higher-level languages primarily used in modern application software. This leads to the concurrency and synchronisation features linking closely to the concepts themselves without being hidden behind a higher level of abstraction. The concurrency aspects of C utilise the `pthreads` library which is included in most C distributions and is a common piece of learning for developers working with C.~~

### Concurrency

Concurrency is achieved through multithreading and multiprocessing which have some key differences. Threads can be thought of as smaller units of execution within a process, sharing the same memory space and hardware resources. They are much lighter weight compared to processes as they have their own isolated memory.

Multithreading allows multiple threads to run concurrently within a single process. In C, threads are typically implemented using the POSIX Threads (*pthreads*) library. Threads can be created and later joined within the main thread. Joining ensures that the main thread waits for child threads to complete their tasks before continuing execution. Additionally, joining allows the return values of threads to be collected back into the main thread, which is useful for managing results from concurrent operations. We can still run into issues when programming concurrently in C when we don’t also ensure synchronisation.

### Synchronisation

One of the main challenges faced when concurrently programming is race conditions, which occur when threads access shared resources unsafely. This can be prevented by means of synchronisation. C has supports two common mechanisms: mutexes and semaphores accessed using the *pthread* and *semaphore* libraries. These libraries provide an easy-to-use API for managing the locking and control of shared resources within threads. However, these APIs are very manual and rely on the programmer to avoid issues like **deadlocks**, where threads wait indefinitely for resources held by each other.

## Python Overview

### Concurrency

Python also supports concurrency through multithreading, implemented via its *threading* module. Unlike C, Python includes a Global Interpreter Lock (GIL), which ensures only one thread executes Python bytecode at a time. This limits true parallelism for CPU-bound tasks but is less restrictive for I/O-bound tasks, which naturally involve pauses in execution.  
Python’s higher-level design eliminates the need for manual thread creation and joining, as the threading module handles these operations automatically, reducing the risk of errors.

We are, however, able to overcome these restrictions using Python’s *multiprocessing* module which allows the creation of separate processes with their own memory space. These processes bypass the GIL, making multiprocessing a better choice for CPU-bound tasks requiring heavy computation.

### Synchronisation

Locking is also available in python but in many flavours and at a higher level of abstraction than C. These include simple locks, reentrant locks (used in recursive methods) and semaphores which are all part of the *threading* module.

Unlike C, Pythons mechanisms are more straightforward due to their higher level, clean API’s. Reentrant locks are a good example of this as they would need to be manually constructed to allow a thread to acquire the same lock multiple times without deadlock. This would increase the risk of deadlocks as it relies on the competency of the programmer.

## Java Overview

### Concurrency

Java takes concurrency to an even higher level with its *concurrent* package which includes many tools including the *Thread* class. This can be extended in other classes or similar functionality can be achieved by implementing the *Runnable* interface which will be familiar for most Java developers.

Java is suitable for more scalable applications; its *Executor* framework gives high level abstractions for managing pools of threads. Using this mechanism we can reuse threads, reducing CPU overhead and avoiding the inefficient practice of repeatedly creating and destroying threads.

Introduced in Java 8, the *Stream* API further enhances concurrency by supporting parallel streams. These allow us to process large datasets in parallel without manually managing threads due to the high-level API. Parallel streams are particularly useful for data-intensive applications where operations on collections can be distributed across multiple threads with minimal effort.

### Synchronisation

Java enforces thread safety through its *synchronised* keyword, which locks methods or sub-programs to prevent race conditions. By default, these are reentrent allowing multiple acquisitions without causing deadlocks. Java also has a full locking framework which expands on this.

Java also has specific thread-safe data structures such as *ConcurrentHashMap* which automatically restrict concurrent access to the data inside it. This reduces overhead as there is no need to manually implement locks around these variables.