# week 7 - multi-core architecture & intro to multi-threaded software

References:
**Essential readings: GPBBHL 1.1, 1.2**
Further readings: P&H 5.8, OO "Processes and Threads"

---

# Chapter 1.1

**Problem with old computers** - they used to run code sequentially from start to end where a new task begins after the current one is done. it is inefficient and expensive use/waste of scarce computer resources

**Solution** - concurrency (programs can run simultaenously) - how? threads!

Factors leading to development of OS that allow multiple programs to run simultaneously:

- **Resource utilisation** - programs sometimes have to wait for external operaations e.g. I/O, and cannot do useful work while waiting. More efficient to **use that I/O wait time to let other programs run**
- **Fairness** - let multiple users/programs share the computer via finer-grained **time slicing** than letting obne program run to completion and start another
- **Convenience** - easier/more desirable to write several programs that each perform a single task and have them coordinate w each other as necessary than write a single program that performs all of the tasks

Asynchronous tasks - tasks that perform in the background e.g. you can do something else while you wait for a kettle to boil when making tea (kettle boiling is an asynchronous task)

These factors also contribute to the development of threads (also known as *lightweight processes*)

**Threads**

- basic unit of scheduling as deemed by modern operating systems
- allow multiple streams of program control flow to coexist within a process
- share process-wide resources, e.g. memory, file handles, but each thread has its own PC, stack and local variables.

- enable hardware parallelism in multi-processor systems where multiple threads within the same program can be scheduled simultaneously on multiple CPUs (cores)
- they execute simultaneously and asynchronously relative to each other.
- as threads share the memory address space, all threads within a process have direct access to the same variables and allocate objects fron the same heap.
- if threads are not explicitly synchronised, they can **modify variables that other threads are currently using** (i.e. race conditions), resulting in unpredictable/unintentional outcomes - output can be different to what is expected. this behaviour is known as **concurrency bugs**

# Chapter 1.2

**What can threads do**

- reduce development / maintenance costs
- improve performance of complex applications
- easier to model how humans work/interact where asynchronous tasks are turned to sequential workflows
- code inside a thread can be more straightforward and easier to read, write and maintain
- improve responsiveness of user interface in GUI
- improve resource utilisation/throughput in server applications\

**Main benefits of threads**

- **exploit multiple processors**
  - programs with multiple active threads can execute simultaneously on multiple processes in a single chip (multiprocessors), as one thread will run on at most one processor at a time
  - another thread can run while first thread is waiting for I/O to complete, so application can make progress even with a blocking I/O call (its like reading a newspaper while waiting for water to boil rather than the opposite), even if mutliple threads are running in a single processor
  - if designed effectively, they can improve resource throughput as they utilise available processor resources more effectively
- **simplicity of modelling**
  - idea is that easier to handle one type of task to perform than many at once
  - hence a program that handles one task at a time is easier to write, less error-prone and easier to test than a program that handles multiple tasks at once
  - assigning a thread to each type of task provides illusion of sequentiality
  - allows decomposition of a complicated, asynchronous workflow into a simpler, synchronous workflows each running in a separate thread where they only interact

    with each other at specific synchronisation points
  - application of that: servlets (when `service` method is called, it creates a synchronous workflow to process a new web request as if its a singlethreaded program)
- **simplified handling of asynchronous events**
  - each socket connection is allocated its own thread and allowed to use synchronous I/O, as it is easier to develop a server application that accepts socket connections from multiple remote clients
  - if each request has its own thread, blocking I/O (when application tries to read from a socket and there is no data available) doesn't affect processing of other requests.
  - otherwise, other requests cannot be processed if the current request encounters blocking I/O, creating a bottleneck in the application
  - gives rise to the thread-per-client model even for large number of clients on some platforms
- **more responsive user interfaces**
  - GUIs used to be single threaded and that application code was run indirectly via a "main event loop" - which is replaced in modern GUIs by event dispatch thread (EDT)
  - Allows long-running tasks to be executed where each task is given their own thread as if they perform in event thread, it can lead to lag, causing unresponsiveness (app freezing). Putting them into threads allow event thread to process UI events in the meantime, making the GUI more responsive