TTK4145 – Real-time Programming

Sondre Bø Kongsgård
sondrebk@stud.ntnu.no

Contents

1 Concurrency & Parallelism

This chapter is based on (more or less equal to) the solution to exercise 1.

Concurrency vs. parallelism "Concurrency is the composition of independently executing processes, while parallelism is the simultaneous execution of (possibly related) computations. Concurrency is about dealing with lots of things at once. Parallelism is about doing lots of things at once."

Reasons for concurrency

- The need to model that happen "simulatenously".
- The need to exploit multicore hardware.
- The need to utilize hardware efficiently (even a single core).

Degree of difficulty to use concurrency Creating concurrent programs is easier if the system consists of multiple (somewhat) independent parts, like

- Two periodic functions with different periods.
- Handling multiple network requests.
- Polling different independent hardware, and generating separate events for each.
- Things that are "embarrassingly parallel".

Creating concurrent programs is harder if the concurrent parts have to interact, as we need some way to share data safely – communication and/or synchronization:

- The $i = i + 1$ problem demonstrates that strange things may happen if we don't share data safely.
- All threads share the same processor and memory (for common CPU architectures, anyway), so this also creates a form of interaction: Running one thread means we're not running another. This is a problem for real-time (ie. timing-sensitive) applications.

Different kinds of concurrent execution

- *Process*
  - An executing program (as opposed to just the code itself)
    - Executable code
    - One or more threads
    - An address space – the OS and hardware enforces that we cannot access the memory of one process from another
- *Thread*
  - A "unit of execution"
    - Has its own stack
    - Scheduled by the OS
    - Contains data structures for saving the current state of execution (the "context"), so its execution can be paused and resumed
  - Share memory with the same process
    - Some languages/extensions (such as Go/D) offer "Thread-local storage" of the heap, such that also the heap is not shared
- *Green thread*
  - Managed & (preemptively) scheduled by the runtime (ie. not OS-managed)
    - A scheduler is part of the core library of that language
    - Often mapped N:M over several threads
  - May or may not share memory
- *Co-routine*
  - Cooperatively (ie. manually, non-preemptively) scheduled by the user/programmer. No runtime, not OS-managed
    - All co-routines must call "yield"/"reschedule" (or its equivalent), else other co-routines will not run (hence the need to be "cooperative")
    - Sometimes mapped N:M over several threads, but usually just multiplexed over a single thread
  - Usually share memory
  - May not need to save/restore the execution context, saving significant management overhead

Thread spawning functions

- `pthread_create` (C++/POSIX) → Creates an OS-managed thread
- `threading.Thread` (Python) → Creates an OS-managed thread
- `go` (Go) → Creates a co-routine