Declaration

I declare that the material submitted for assessment is my own work except where credit is explicitly given to others by citation or acknowledgement. This work was performed during the current academic year except where otherwise stated. The main text of this project report is NN,NNN words long, including project specification and plan. In submitting this project report to the University of St Andrews, I give permission for it to be made available for use in accordance with the regulations of the University Library. I also give permission for the report to be made available on the Web, for this work to be used in research within the University of St Andrews, and for any software to be released on an open source basis. I retain the copyright in this work, and ownership of any resulting intellectual property.

word count

Chapter 1

Background

1.1 Processes and Threads

The scheduling problem has been around for a while. The earliest papers which discuss scheduling date back to 1962 [2], with hard real-time scheduling (i.e. scheduling for systems where time limits *must* be met) dating back to 1973 [5], and preemptive scheduling being discussed in 1975 [4]. In this section I will give an overview of the scheduling problem, explaining some key concepts and problems, basing a lot on Silberschatz et al.'s book on operating systems [6].

Code running is represented as a *process*. Processes contain the code being run, information such as what process started the current process (its *parent*), what process(es) the current process has created (its *children*), the state of the process (e.g. 'running', 'waiting', 'ready'), and also some information about the current memory and scheduling information about the process's priority, how long it has been running for, etc. A process may contain multiple *threads*. A thread is a simple unit of computation or work being done. A thread has an ID and contains a stack, a register set, and a program counter. The other parts, e.g. the code or memory information, is stored in the process running the thread and is shared between all the process's threads. Hence, creating new threads and inter-thread communication is cheaper than creating new processes and inter-process communication. This is useful for processes which want to do different, often independent things in parallel, e.g. a web browser fetching data from the internet whilst also handling the user's keystrokes. It can also be useful if the problem being solved can be broken up into smaller parts which can then be done in parallel.

In most major operating systems, MacOS, Windows, Solaris, and Linux, there are both user threads and kernel threads. Kernel threads, as the name implies, are managed directly by the operating system kernel. User threads, on the other hand,

are created and managed by the user/programmer. Kernel threads are more expensive to create than user threads and as such, most operating systems map user threads to kernel threads during runtime. This means that the user/programmer does not have to worry about how many threads they create, and that the operating system can reuse kernel threads for various applications, thereby improving performance. When it comes to scheduling, the scheduler manages kernel threads.

1.2 Scheduling

When talking about scheduling, the terms "process scheduling" and "thread scheduling" are often used interchangeably. To avoid confusion with the previous section and to be consistent with commonly used Linux terminology, I shall use "task" to refer to anything, be it a process or a thread, which needs to be scheduled. Scheduling is the problem determining what tasks need to run and for how long. Most tasks do not require constant computation. Instead, computation happens in bursts and the other time is, for example, spent on waiting for I/O to happen, e.g. reading a value from memory. So in order to maximise CPU usage, a scheduler swaps different tasks on the CPU(s). Another reason scheduling is required, is that if the scheduler did not swap the tasks, a single CPU-intensive task could hog the CPU for a long time. On interactive systems, this could result in loss of interactivity (aka. "freezing") until the heavy task finishes its computations. By swapping tasks, the scheduler tries to keep the CPU-usage as fair as possible. Ideally, the perfect scheduler would let the task that has the least time left run first, thereby minimising the wait time for the other tasks. However, this would require the scheduler to know the future (i.e. when the various tasks would stop) and is therefore unfortunately impossible. Instead, there exist various scheduling algorithms which are used to determine what task to run next. One way to determine this is to assign the tasks a priority and execute the highest-priority task first. This works fine if the high-priority jobs finish. However, there is a very real risk that a low-priority job may never be run as it is kept at the back of the queue by higher-priority jobs. As such, priority scheduling is rarely used in its naïve version, with modern scheduling algorithms changing the priority of a task over time, based on various variables. One example of this is the Linux "Completely Fair Scheduler" (CFS) which was introduced in version 2.6.23 [1].

1.2.1 The Completely Fair Scheduler

Typically, the tasks that are waiting to be executed on the CPU are stored in a so-called "ready queue". However, the CFS stores the tasks in a data structure known as a Red-Black tree (RB-tree). RB-trees were first introduced in 1978 by Guibas and

Sedgewick [3]. An RB-tree is a type of self-balancing binary tree

explain RB-trees

The problem of how to schedule the various jobs running on a computer is complex, and has only gotten more complex as multicore processors (i.e. chips with multiple CPU cores on them) became commonplace. With the introduction of simultaneous multithreading, the ability to run more than one thread per CPU core, the problem has gotten further complex.

mention asymmetric single-ISA

- explain perfect way (shortest job first) and why it doesn't work
- mention that the problem has gotten more complex as we now have multicore processors and simultaneous multithreading (explain these)
- explain priority scheduling and its problems
- explain CFS

1.3 ARM big.LITTLE architecture

- what is an ISA?
- what is single-ISA means and why it is helpful?
- mention that ARM big.LITTLE is an example of this
- what is a big core and what is a LITTLE core?
- why do we have this split?
- mention that this poses new challenges to the already complex problem that scheduling is

Chapter 2

Literature Review

- it is only recently that people have tried to make the scheduler/OS aware of big.LITTLE (WASH+COLAB)
- it is only very recently that people have tried to make the scheduler/OS aware of energy (Basireddy/Reddy + linux kernel/ARM)
- something about why this dissertation is still relevant...

Bibliography

[1] *CFS Scheduler — The Linux Kernel documentation*. URL: https://www.kernel.org/doc/html/latest/scheduler/sched-design-CFS.html (visited on 03/26/2020).

[2] Corbató, F. J., Merwin-Daggett, M., and Daley, R. C. "An Experimental Time-Sharing System". In: *Proceedings of the May 1-3, 1962, Spring Joint Computer Conference*. AIEE-IRE '62 (Spring) (May 1962), p. 10. DOI: 10.1145/1460833.1460871.

[3] Guibas, L. J. and Sedgewick, R. "A dichromatic framework for balanced trees". In: *19th Annual Symposium on Foundations of Computer Science (sfcs 1978)*. 19th Annual Symposium on Foundations of Computer Science (sfcs 1978). Ann Arbor, MI, USA: IEEE, Oct. 1978, pp. 8–21. DOI: 10.1109/SFCS.1978.3. URL: http://ieeexplore.ieee.org/document/4567957/ (visited on 04/02/2020).

[4] Kleinrock, L., Gail, R., and Kleinrock, L. *Computer applications*. Queueing Systems Leonard Kleinrock; Richard Gail ; Vol. 2. OCLC: 299649623. New York, NY: Wiley, 1976. 549 pp. ISBN: 978-0-471-49111-8.

[5] Liu, C. L. "Scheduling Algorithms for Multiprogramming in a Hard- Real-Time Environment". In: *J. ACM* 20.1 (Jan. 1973), p. 16. DOI: 10.1145/321738.321743.

[6] Silberschatz, A., Galvin, P. B., and Gagne, G. *Operating system concepts*. 9. ed., internat. student version. OCLC: 854717328. Hoboken, NJ: Wiley, 2014. 829 pp. ISBN: 978-1-118-09375-7.