Analysis of SMP Kernel Scheduling

Jnaneshwar Weibel

April 17, 2018

Abstract

As algorithms become larger and more resource intensive, computer hardware must become increasingly efficient to effectively run such programs. In particular the capability for multiprocessing allows systems to distribute work across multiple CPUs. Modern operating systems and their respective schedulers have the opportunity to make policy decisions, such as load balancing and locality of access to shared memory, when distributing programs that can influence the performance of both the kernel and user programs.

1 Introduction

Hardware has evolved to become increasingly more complex and efficient, through areas such as processor frequency and memory latency. Amdhal's law gives diminishing returns on the impact of parallelizing computation. However, with respect to symmetric multiprocessing (SMP) in the kernel, Gustafon's law proves as now as a **scheduler?** can run "in the same time with a larger workload" [3]. Thus as new system architectures take advantage of these processors. They also require new approaches for system software and applications that run on top of this hardware [1].

Within an operating system, the task scheduler is responsible for running user tasks and determining an execution order. Yet with multiprocessing the scheduler offers an opportunity to distribute work across multiple cores. In a preemptive system, the scheduler may migrate tasks between different cores which generally will incur a local penalty in cache performance yet may improve global performance. Thus for this report several variations on the classical task scheduler will be compared to their impact on system performance and task scheduling.

A run-queue will consist of a FIFO list of tasks and will be protected by a run-queue specific lock. The performance of several tasks will be measured using a single global run-queue, a per-core run-queue without pull migration, and a per-core run-queue with pull migration. Pull migration refers to periodically re-balancing the load on two run-queues. In addition, some tasks will be constrained to a smaller set of CPUs and thus may be fixed to a single process or not undergo push migration. This paper will demonstrate the performance implications that various scheduling algorithms have on both local and global task execution.

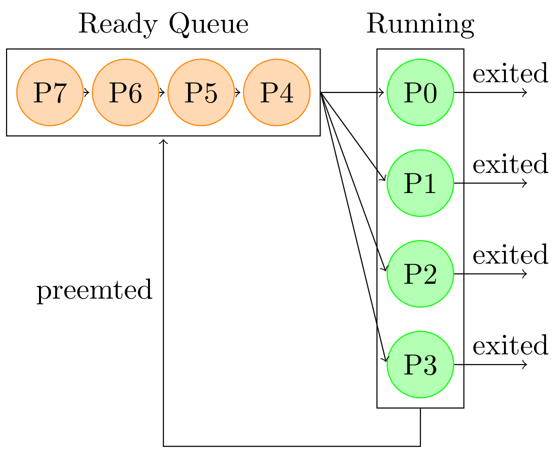
2 Environment?

The OS will be compiled for the ARMv8 (aarch64) instruction set and will be executed on the Raspberry Pi 3 B. The device includes a 1.2 GHz 64-bit quad-core ARM Cortex-A53 processor and consists of a 16KB L1 data cache, a 16KB L1 instruction cache, and a 128KB L2 cache. Intra-core cache coherency is enabled and uses the MOESI protocol [2]. The memory address space is configured using a linear two level translation table with accessible memory configured as normal outer and inner write-back, write-allocate and device register memory configured as device nGnRnE. Performance will be measured using the ARM Performance Monitor Unit (PMU) and the core timer. The timer executes at a frequency of 19.2 MHz [6] and each core maintains its own core timer which will preempt the current task each MS. Both the kernel and pseudo-user processes will execute in EL1; however, the kernel will use SPx and the user processes will use SP0.

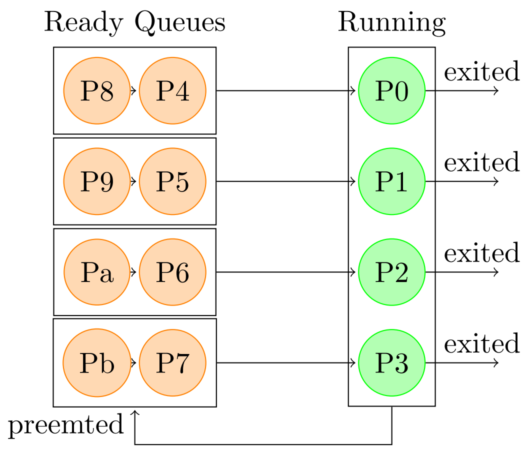
3 Definitions?

A ready queue can be defined as a generic data structure with two primary operations: ready and next. For its internal manipulation a ready queue is protected by a spinlock. Internally there are several queues corresponding to different priority levels allowing for processes with different priorities to be pushed onto the queues with no contention. However, for discussion we will assume that all processes run at the same priority.

A naieve implementation for a SMP scheduler involves a global ready queue which all cores pull processes from. This naieve implementation acts identical that in a single core system. On SMP systems this data structure is benificial since it guarantees equal distribution of workload across all cores in addition to implementation simplicity. Additionally, this workload distribution is not affected by a rapid stream of exiting processes. However, the approach suffers from kernel lock contention since all cores must synchronize access to the queue.



(a) Global Ready Queue



(b) Core Ready Queue

Figure 1: Scheduler Structures

Under the current scheduler, a process, if eligible, is migrated during preemption. In general the scheduler will opt to keep a process on the same CPU unless another core is sufficiently empty or its affinity set dictates otherwise. In essence this rebalancing means that as processes are rescheduled, the cores will tend towards a balanced configuration. However, it is also possible that a core will have a stream of processes terminate leading to a load imbalance between ready queues (fig. 3).

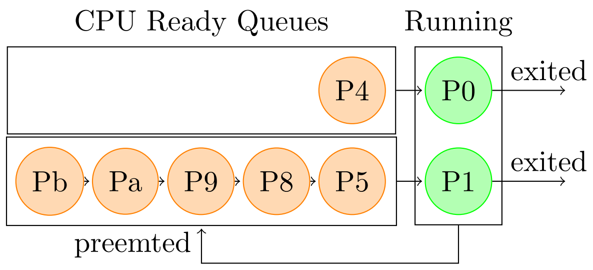


Figure 2: Unbalanced Scheduling

Pull migration is an effort to improve the re-balancing latency for queues which exhibit the behaviour in fig. 3. Periodically, an idle core will search for an overloaded core and pull eligible processes from the tail of its queue. This task requires locking the busy queue for a longer period of time; while, the idle queue tends to have have its lock acquired more frequently. This task leads to high lock contention for both queues which can negatively impact the time spent in the kernel. Specifically, while iterating, the busy core will be unable to retrieve the next runnable process. Thus it is important to pick a large enough interval to do the re-balancing that it does not cause much strain on performance. However, in most cases, if a core is not sufficiently loaded the pull migration task will instead exit prematurely instead of locking both cores.

TODO: Process Fairness

4 Performance

Different user programs were tested against the variations of the OS scheduler described previously. In general the tests consist of performing a scalar multiply for a specified runtime. These tests aim to simulate process cache access and utilization and provide repeatable results for user and kernel event counters. For example, some examples heavily utilize memory while others operate on smaller subsets of data. Additionally other processes are designed to run for longer or shorter periods of time to represent variability in task runtime.

While the actual measured runtime is presented in jiffies, these measurements should rarely be compared as they are rough measurements based on the CPU timer. Additionally, tasks which may have a variable amount of IO with the FTDI output device are likely to see unrepresentative results compared to the actual execution of tasks. Finally it is also important to note that all data fits into the L2 cache and thus pipeline stalls involved when fetching from main memory are limited the initial cold start. **TODO: talk about the L2 cache in conclusion as it amplifies performance penalty**

To describe the behaviour for execution some terms should be defined.

- **Work**: the local size of data available for execution in a single thread, specifically related to the dimensions of the L1 data cache.
  - uint64_t[20][25] = 25% L1 data cache
  - uint64_t[40][50] = 100% L1 data cache
  - uint64_t[80][100] = 400% L1 data cache
- **Samples**: the global number of work entries available for execution by **all** threads
  - 4 independent work entries = high spatial locality
  - 16 independent work entries = low spatial locality
- **Difficulty**: a relative measure of bounded execution time by a single thread
  - 10^2 repeated computations = easy
  - 10^3 repeated computations = normal
  - 10^4 repeated computations = hard

Figure 3: single core baseline comparison

Type	Total	User	Kernel
Instrs	467617788	466380022	1237766
Cycles	693267321	691411106	1856215
Access	264291472	263665636	625836
Refill	1539	1340	199
Runtime	45374884	-	-

(a) single core - matrix 1x utilization

Type	Total	User	Kernel
Instrs	1858506765	1853566311	4940454
Cycles	2768987855	2760678046	8309809
Access	1050488228	1047990323	2497905
Refill	1389770	1293188	96582
Runtime	178608294	-	-

(b) single core - matrix 4x utilization

These initial examples were designed to demonstrate the performance penalty of poor cache utilization in user programs. The first example will fill up the L1 data cache while the second will overfill the cache, thus requiring a refill on each iteration. It is important to note that while the cycle count is 4x larger as expected, the refill rate is larger by nearly 900% with the overfilled cache. The most notable performance comparisons comes from the number of instructions per cycle in the kernel being roughly 11% slower, due to frequent thrashing of kernel memory. Thus, the overloaded example will be executed in a trivially parallelizable manner for the following two tests.

TODO: note about how to improve global runtime in userspace programs

Figure 4: strided matrix multiplication running against different schedulers

Type	Total	User	Kernel
Instrs	354249559	353629318	620241
Cycles	527538552	526403550	1135002
Access	200305644	199963039	342605
Refill	125708	116722	8986
Runtime	41175374	-	-

(a) multi core - global queue

Type	Total	User	Kernel
Instrs	375176881	374755150	421731
Cycles	556693572	556049701	643871
Access	212114934	211903384	211550
Refill	15332	14735	597
Runtime	40603738	-	-

(b) multi core - cpu queue (affinity)

Extending the previous examples to a multicore system serves as a demonstration to the benefit of well thought out parallelization. A single sample operating on 400% of a processors cache at hard difficulty was executed against the global scheduler and the core scheduler with process affinity to separate computation onto each of the four cores. Despite having a 94% APR improvement in userspace, the difference in IPC is nearly insignificant. However, in kernel execution there is a 76% improvement in cycles which results in a cumulative 20% improvment in kernel IPC. This is likely due to predictable cache access patterns leading to much fewer kernel refills and hence fewer wasted cycles. Additionally, the number of kernel instructions is 47% less when running with a per core runqueue, owing to no lock contention between cores during a context switch.

Figure 5: 60 threads consisting of 48 hard processes and 12 easy processes

Type	Total	User	Kernel
Instrs	1543135939	1396850683	146285256
Cycles	3146576481	2076522276	1070054205
Access	845516444	789349384	56167060
Refill	7653514	112360	7541154
Runtime	152657484	-	-

(a) without pull migration (high locality)

Type	Total	User	Kernel
Instrs	1468299596	1398454567	69845029
Cycles	2552848279	2079318879	473529400
Access	816022596	790256656	25765940
Refill	3126952	140213	2986739
Runtime	156804356	-	-

(b) with pull migration (high locality)

Type	Total	User	Kernel
Instrs	1613944771	1404552990	209391781
Cycles	3491698429	2086636583	1405061846
Access	873214573	793699999	79514574
Refill	9792188	28820	9763368
Runtime	152321428	-	-

(c) without pull migration (low locality)

Type	Total	User	Kernel
Instrs	1520575475	1445743126	74832349
Cycles	2574353547	2147872630	426480917
Access	843643737	816978128	26665609
Refill	2283806	34930	2248876
Runtime	161631896	-	-

(d) with pull migration (low locality)

To test the behaviour of the system under a rapid stream of terminating processes difficulty was split into 12 hard threads and 60 easy threads designed to quickly terminate. Each matrix would operate on 25% of the cache and the examples were under each of the sample data locality cases.

For these examples, the impact of pull migration is immediately evident in the reduced cycle count of 75-80% and a maximum reduction to 25% of the original l1 refill rate to the same tests running without push migration. Despite the apparent cache benefit it is important to note that the cycles spent in userspace tend to remain around the same meaning that the more important improvement ends up in the kernel. When investigated more, it is noticable that the kernel cycle count and number of instructions executed drops significantly with pull migration. Since the change in the kernel refill rate more insignificant this decrease in kernel overhead seems to be highly related to decreased lock contention between the cores.

This contention is derived from the scenario of multiple busy queues attempting to rapidly push work onto an idle queue. In these cases, the added lock contention will inhibit processes from being scheduled on all cores since they are all effectively being serialized by the idle queue runqueue lock. With pull migration, instead of having many repeated requests to the idle cores runqueue, a busy core elects to offload a larger amount of work until the idle core is no longer considered idle. This prevents other cores from offloading work unnecessarily leading to improved userspace cache behaviour for those cores as well as reduced lock contention and minimal serialization. Additionally, pulling processes from the end of the runqueue; instead of migrating recently executed processes, provides the opportunity to exhibit temporal locality and less cache pollution from more processes being executed.

TODO: cleanup these paragraphs

Figure 6: even distribution of easy, normal and hard threads

Type	Total	User	Kernel
Instrs	2945961318	2829092653	116868665
Cycles	4881259967	4206238780	675021187
Access	1642115209	1598721424	43393785
Refill	4570467	298213	4272254
Runtime	175864992	-	-

(a) without pull migration

Type	Total	User	Kernel
Instrs	2727027846	2625810066	101217780
Cycles	4421745433	3903078399	518667034
Access	1520521397	1483845439	36675958
Refill	3226996	222090	3004906
Runtime	171909144	-	-

(b) with pull migration

While the previous example demonstrated the immediate benefit of pull migration, this case utilizes an even difficulty distribution of 25% of the L1 data cache, with low spatial locality across all cores to demonstrate long term scheduling benefits. Specifically, despite lower lock contention with pull scheduling, the processors are able to minimize the migration overhead relating to cache invalidation. Since tasks are migrated from the end of the runqueue, it is highly probable that a tasks would have its cache fully polluted on a busy core by the time it was next executed. Thus migrating to an idle queue helps distribute evenly distribute memory access across all cores reducing the impact on any one individual core. Additionally, the higher cache utilization leads to fewer stalled cycles and thus more opportunity to complete work earlier in userspace leading to fewer preemption context switching.

Figure 7: comparison between thread pool and fully threaded execution

Type	Total	User	Kernel
Instrs	836416320	753346040	83070280
Cycles	1724034904	1123196400	600838504
Access	456170314	426134109	30036205
Refill	4027452	231590	3795862
Runtime	165276806	-	-

(a) threads (high locality)

Type	Total	User	Kernel
Instrs	811243405	741653152	69590253
Cycles	1789217832	1113471653	675746179
Access	444450115	419398612	25051503
Refill	4540858	379326	4161532
Runtime	66534692	-	-

(b) pooled (high locality)

In Fig. 7, a work sized at 25% of the L1 data cache is distributed to 64 threads scheduled into independent threads assigned work **or** 16 worker threads pulling work from a protected queue. They run under the same difficulty; however, locality between the two classes of samples is evaluated separately.

TODO: try this with semaphore TODO: remeasure runtimes? repeat tests with different MD case? TODO: mention that these were unexpected, was expecting much better performance with the pooled - md This example has ipc larger in both cases. However, one noticeable factor is that for larger sample size there is less l1 thrashing relevant in the APR (contributing factor for larger runtimes). Instead for the smaller case the caches are able to be fully utilized.

Type	Total	User	Kernel
Instrs	819345284	755845609	63499675
Cycles	1672284752	1125235476	547049276
Access	450878090	427546852	23331238
Refill	3740754	148469	3592285
Runtime	167143162	-	-

(a) threads (low locality)

Type	Total	User	Kernel
Instrs	840132648	753613514	86519134
Cycles	1849770029	1127098953	722671076
Access	457128942	426166739	30962203
Refill	4435166	143616	4291550
Runtime	66158114	-	-

(b) pooled (low locality)

5 Lessons Learned

Since the development of the operating system was done from the ground up there were plenty of learning opportunities along the way.

Measuring success or failure in a bare metal environment can prove to be quite difficult; thus it became important to progressively escalate to better debugging utilities. Initially, bare minimum assembly required to toggle an LED gave limited success or failure output. The jump to FTDI was helpful in debugging simple conditions and CPU registers. Finally, for tracing complex execution, having a proper debugging protocol with JLink + OpenOCD + GDB enabled better debugging for context switching, exception handling and multicore execution.

While the intial JLink setup was incredibly difficult due to incompatability with the current version of OpenOCD, mismatched toolchain and microcontroller runtime, and a poorly configured microcontroller, the end result was incredibly important to establishing a sucessfull development workflow. Instead of having to perform a careful SD card dance between my computer and the Pi, the ability to directly flash to RAM allowed for much quicker development. Initially, I had looked into alternative bootloaders (UBoot, GRUB) which can work over ethernet, WIFI, or serial connections; however, in general the end result of having proper host debugging utilities proved invaluable.

Relating to some of the issues of toolchain incompatability; initially code was compiled using the `arm-none-eabi` toolchain which is used for cross compiling to 32-bit arm. While, the Pi offers compatability with the older architecture, the generic debug interface is less compatible with the 32-bit toolchain. Thus it is important to recognize that despite older systems often having more documentation; the complexity of developing on a compatability layer is not worth the extra effort. In addition, in the aarch64 architecture proved to have clear and more thought out semantics especially when relating to exception handling. The exception model for aarch64 is standardized across four types; whereas, the arm32 architecture requires storing and restoring a different subsets of registers as well as having different methods to return from interrupts. Additionally, the addition of a unified exception status register (ESR) was much more helpful with determing the cause of a processor fault.

Newlib itegration

6 Conclusion

Conclusions here [4] [5].

7 Future Work

References

[1] Holdren, J. P., Lander, E., and Varmus, H. Report to the president and congress designing a digital future: Federally funded research and development in networking and information technology, 2010. `https://www.cis.upenn.edu/~mkearns/papers/nitrd.pdf`.

[2] Ltd, A. Arm cortex-a53 mpcore processor technical reference manual — data cache coherency – arm developer, 2018. `https://developer.arm.com/docs/ddi0500/e/level-1-memory-system/cache-behavior/data-cache-coherency`.

[3] McCool, M., Robison, A., and Reinders, J. *Structured parallel programming: patterns for efficient computation.* Morgan Kaufmann, 2012.

[4] McKusick, M. K., Neville-Neil, G. V., and Watson, R. N. M. *The Design and Implementation of the FreeBSD Operating System.* Addison-Wesley, 2015.

[5] Schimmel, C. *UNIX Systems for Modern Architectures: Symmetric Multiprocessing and Caching for Kernel Programmers.* Addison-Wesley, 1994.

[6] Van loo, G. Arm quad a7 core, 2014. `https://www.raspberrypi.org/documentation/hardware/raspberrypi/bcm2836/QA7_rev3.4.pdf`.