Theoretical and Experimental Approach to SMP Scheduling

Jnaneshwar Weibel

Abstract. Over the years computer hardware has grown increasingly efficient; 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 about this distribution that may influence the performance of user programs. Shared memory and locality of data ...

Keywords. Word1, word2, word3.

1 Introduction

Modern computer hardware has grown increasingly efficient to improve the performance of various systems by improving areas such as processor frequency and memory latency. While instruction throughput can be improved with optimizations to a CPU pipeline, multiprocessing is also increasingly important for improving throughput. To improve shared memory latency, processors have adopted smaller memory subsystems, or caches, that allows quicker data access.

As the hardware evolves, software must be able to take advantage of these changes. Within an operating system, the scheduler is responsible for running user tasks and determining an execution order; however, with multiple processors these tasks can be distributed across cores. In a preemptive system, a task may migrate between different processors and this migration will generally impact cache performance; however, globally this migration my improve performance. Thus several distinct policy decisions relating to task scheduling will be compared.

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 cache and a 128KB L2 cache. Intra-core cache coherency is enabled and uses the MOESI protocol [**?**]. 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 and each core maintains its own core timer which will preempt the current task each MS. [**?**]

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 internal 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 a ready queue is a singular list of processes protected by a spinlock.

A naieve implementation for a SMP scheduler involves a global ready queue which all cores

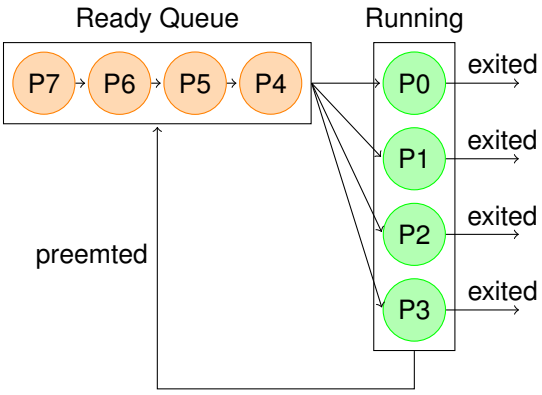

Fig. 1. Global Ready Queue

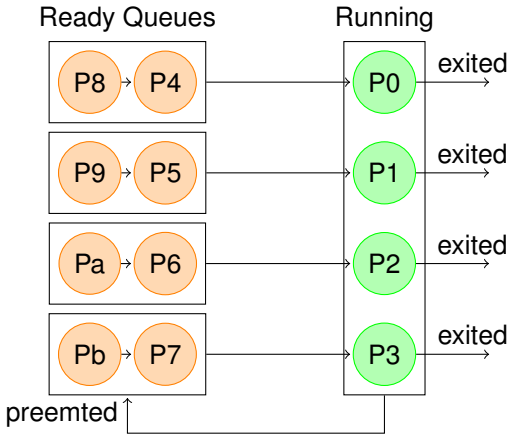

Fig. 2. Per Core Ready Queue

pull processes from. This naieve implementation acts identical to a ready queue implentation 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. (TODO possibly example)

Under the current scheduler, a process, if eligible, is migrated at its preemption. In general the scheduler will opt to keep a process on the same CPU unless another core is sufficiently empty. 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).

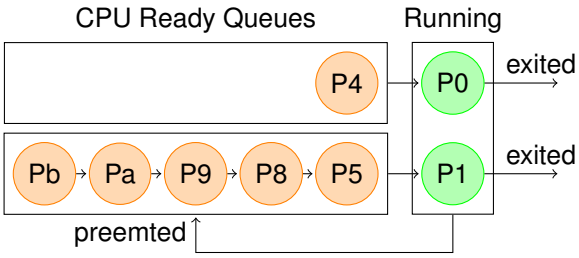

Fig. 3. Unbalanced Scheduling

4 Performance Tests

Various user programs were tested against different variations of the scheduler. In general the tests were performed on processes tasked with performing a scalar multiply on a data set with a size relative to the L1 cache. As a baseline, the OS was configured to run on only a single core and a matrix multiplication was performed on a matrix exactly the size of the cache. For more interesting results, a data set 4x the size of the cache was tested with the global run-queue and the per-core run-queue with evenly distributed CPU affinity. In these cases each process would work on an independent part of the data set.

The impact of per-core scheduler with and without pull migration was analyzed with tasks with varying runtime. Three classes of processes were defined as short, medium and long; each, with 10x the runtime of the previous case. An even distribution of each type of task and the data set was tested against these schedulers. Additionally the schedulers were tested against many short runtime processes compared to the few long runtime processes.

Finally, a common pattern; in resource intensive and parallelizable tasks, is to create a smaller subset of OS tasks and pull work off a shared queue. Otherwise known as a thread pool, the difference between allowing the OS to manage all the tasks compared to semi-manual work queueing was examined. Work was divided into 64 threads

with 16 workers for the pooled case. Additionally the locality of the data being worked on was constrained to both 4 (sm) and 16 (lg) matrices of a smaller size than the L1 cache. All tests ran on the per-core scheduler with pull migration.

	Total	User	Kernel
Instrs	0	0	0
Cycles	0	0	0
Access	0	0	0
Refill	0	0	0
Runtime	0	0	0

Table 1. table caption

4.1 Linguistic Features

Features.

4.1.1 Lexical Features

Other features.

$$H(X|Y) = -\sum_{y \in Y} P(y_i) \sum_{x \in X} p(x_i|y_i) \log p(x_i|y_i) \tag{1}$$

5 Conclusion and Future Work

Conclusions here.

Acknowledgements

We would like to thank..
This work is funded by...

References

John Smith is professor ...

Juan Perez is researcher...

Ivan Petrov is researcher...

Article received on 06/12/2012; accepted on 16/01/2013.