**Simulation of multi-core scheduling**

1. **Abstract**

In this paper, we first look into some key issues and mechanisms need to be considered for multi-core scheduling based on SMP (symmetric multiprocessing) architecture, and present the implementation details of the simulator for SMP multi-core scheduling based on what we did on project2. The design of our simulator is to employ the idea of multi-queue multi-core scheduling model, which means each core maintains a group of private queues of the threads. To avoid unbalanced run-queues, threads can be migrated to other cores based on a proper algorithm. The issue of processor affinity has also been well considered in our implementation. Besides, our multi-threaded simulator is implemented in C language with POSIX thread (pthread) library as what project2 does. Pthread mutex-lock is frequently used in our project to avoid concurrent problems.

1. **Introduction**

While scheduling in multi-core systems is not a brand new problem, since many related researches have been done on this topic both theoretically and practically, but they are now still faced with the fact that they need to well parallelize their work in order to get the expected performance increment. Most traditional task scheduling mechanisms are focusing on single-core processor, and they are not able to work well for multi-core processors system. Therefore, expanding our project2 implementation from single-core scheduling to multi-core scheduling and digging into the issues in scheduling on multi-core processors is a practical way of understanding how multi-core system works and how can it be improving performance.

Partitioning the work into parallel pieces is a necessary but not sufficient condition for multi-core scheduling. When considering Scheduling for multi-core processors, we need to first decide what kinds of multicore architecture (AMP or SMP) we are going to study on, and then choose if we want to use a single queue for all cores or each core can have its own queue. If we employ the scheme that each core can have its own private queue (which is what we used in the simulation), then we need to further consider the problems of load balancing and task migration, and also processor affinity. Load balancing means to make even and full use of the processor resources in a system. Processor affinity can represent the user or program preference for the cores or be used to reduce the frequency of migration to better utilize the cache effort. A core should allocate part of the cache resource for each thread running on it so as to store the intermediate data or results, by which the execution performance can be improved. But when a thread migrates to another core, the cache allocated to it by the previous core becomes invalid and the current core has to re-allocate cache resource for it. The load balancing and Processor affinity are actually approaches with opposite efforts on the system and we need to find a balance point between them to optimize the system performance.

The rest of this paper is presented as follows: In Section 3, we talk about multi-core scheduling issues, including multi-queue scheduling scheme, load balancing and processor affinity, in detail to see how they affect the system performance. Section 4 describes how we implement our simulator based on the well consideration of the issues. Section 5 presents the simulation results and a brief conclusion.

1. **Multi-core Scheduling Issues**

Before getting into details of our discussion of multi-core scheduling issues, we need to first briefly introduce the SMP (symmetric multiprocessing) architecture we are going to based our study on. Most multiprocessor systems today use SMP architecture. SMP architecture involves symmetric multiprocessor system hardware and software architecture where two or more identical processors or cores in one processor connect to a single, shared main memory, has full access to all I/O devices, and is controlled by a single operating system instance that treats all processors equally, reserving none for special purposes. The SMP architecture would be depicted as follows (two CPUs),

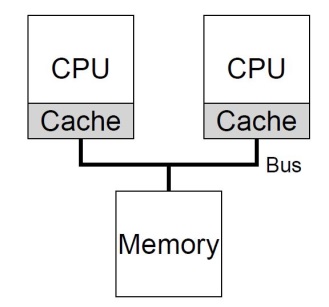


Figure 1 SMP architecture for two CPUs

While in Asymmetric multiprocessing (AMP) architecture, not all CPUs or cores are treated equally; there may exist a master core that handle all the scheduling decisions for other so called slave cores. We are not going to look into AMP in detail in our research. We will look into the issues based on AMP architecture in the following sections in detail.

**3.1 Single queue VS multi-queue multi-core scheduling**

When considering how to build a scheduler for a multiprocessor system. The most basic approach is to simply reuse the basic framework for single processor scheduling, by putting all jobs that need to be scheduled into a single queue. This is called single queue multi-core scheduling or SQMS. This approach would have a very obvious advantage of simplicity: it does not require much work to take an existing policy that picks the best job to run next and adapt it to work on more than one CPU. However, SQMS has obvious shortcomings. The first problem is a lack of scalability. To ensure the scheduler works correctly on multiple CPUs, the developers will have inserted some form of locking into the code, and these locks can greatly reduce performance, particularly as the number of CPUs in the systems grows. The second one is CPU affinity, which we would talk more in the multi-queue scheduling, Because each CPU simply picks the next job to run from the globally shared queue, each job ends up bouncing around from CPU to CPU, and this results in a poor CPU affinity.

As we can see from above, problems exist for single queue scheduling, so some systems nowadays opt for multi-queue multi-core scheduling (MQMS). In this approach, each CPU core would employ private queue, such as one queue per CPU core. Each per CPU core would follow a specific scheduling scheme or same scheduling scheme may be used for all queues. When a job enters the system, it is placed on exactly one scheduling queue, according to some heuristic (e.g., random, or picking one with fewer jobs than others). Then it is scheduled essentially independently, thus avoiding the problems of information sharing and synchronization found in the single-queue approach.

MQMS has a distinct advantage of SQMS in that it should be inherently more scalable. The number of private queues would increase as the growth of number of CPU cores. In addition, MQMS intrinsically provides cache affinity; jobs stay on the same CPU and thus reap the advantage of reusing cached contents therein.

However, problems often accompany the solutions. As you may see, there would be a new problem raised when doing MQMS, the load imbalance problem. Since each per CPU private queue does not share tasks or jobs with each other and run independently, some CPU core may run low or even out of tasks at some points while other cores are still have a busy queue to scheduling with. This could greatly affect the performance of multi-core system. Thus, we need to employ some methods to balance the loads between CPU cores and task migration would become necessary.

**3.2 Load balancing and task migration**

At this point, we would be facing the problem of doing load balancing between CPU cores. Since we are using MQMS scheme, so at some point of running, some cores would result in low on tasks or even idling while others would still have busy run queues to schedule with. How could the scheduler solve the load imbalance problem? The obvious answer to this query is to move jobs around, a technique which we refer to as migration. By migrating a job from one CPU to another, true load balance can be achieved. For example, suppose we now have a situation where one CPU is idle and the others have some jobs. In this case, the OS should simply move one or more jobs from the other CPUs to the idle one. The result of this single job migration is a much more balanced load. And the system performance would be largely improved.

There are actually many migration schemes available out there. One basic approach of balancing the load is called work stealing or pull migration, which is common in the world of multithreaded programming. Work stealing, as it is called, the core which is low on its tasks or idling would try to fetch tasks from other busier cores, and helps balancing the loads. Unlike another approach called work sharing, in which the cores would always want to share its tasks with others, the idle core or core on low tasks would initiate the migration in work stealing approach. The work stealing would result in a less migration frequency and thus a better performance compared to the work sharing approach. And our simulator would try to implement migration using work stealing. We would talk about this in later section.

* 1. **Processor Affinity**

The third issue we look into is processor affinity. Processor affinity enables the binding and unbinding of a process or a thread to a central processing unit (CPU) or a range of CPUs, so that the process or thread will execute only on the designated CPU or CPUs rather than any CPU. But why we need Processor Affinity? There are several benefits:

i. The first benefit of Processor Affinity is optimizing cache performance. As we can see in the SMP architecture, each CPU or core will have its own private cache that could build up a fair bit of state while a thread or process running on it. The next time the process or thread runs, it is often advantageous to run it on the same CPU, as it will run faster if some of its state is already present in the caches on that CPU. In this case, CPU affinity could better utilize the CPU caches by binding processes or threads on one or a range of CPUs or cores.

ii. A second benefit of Processor Affinity is a corollary to the first. If multiple threads are accessing the same data, it might make sense to bind them all to the same processor. Doing so guarantees that the threads do not contend over data and cause cache misses. This does diminish the performance gained from multithreading on SMP. If the threads are inherently serialized, however, the improved cache hit rate may be worth it.

In sum, Processor Affinity takes advantage of the fact that some remnants of a process that was run on a given processor may remain in that processor's memory state (for example, data in the CPU cache) after another process is run on that CPU. Scheduling that process to execute on the same processor could result in an efficient use of process by reducing performance-degrading situations such as cache misses.

Please note that migration and load balancing process we discussed above would work against the principle of processor affinity, and if not carefully managed, the savings gained by balancing the system can be lost in rebuilding caches. Thus we need to carefully handle the balance point between migration and CPU affinity.

In our implementation, a running time bonus mechanism is used to roughly simulate the effort of CPU cache and we will implement Processor Affinity by a user-defined input file (we will discuss it later).

1. **Multi-core Scheduling Simulation**

**4.1 Multi-core Threading Model**

Our simulator employs a thread to simulate a processor core. In each single thread of the core, the logic of scheduling in project2 is reused. Each core thread can “run” one thread at a time, but the whole multi-core simulator can “run” more than one thread concurrently if there are more than one cores.

To achieve this, we define the model of each core in the structure core\_data\_t in multicore\_sim.h. In core\_data\_t, we implement private queues, readyQ, blockQ and unblockQ for the threads which are running on the current processor core. Note that, in project2, these three queues with several mutex-lock and status are global values. In fact, one of the major modifications on project2 is that many external variables become the private members of the core\_data\_t structure. Another thing is that each core maintains a “simlock” rather than share a global “simlock”. It is notable that we still use a global “simlock” for thread dispatching and balancing among the cores (we will discuss this issue in the next section). For each core, the scheduling functions defined in project2 still works with some minor modifications in quite a few places, of which the most important modification is that the scheduling function can only touch the queue of the current core, meaning that the scheduling function cannot select a thread to run or block from other cores. To achieve this, a parameter with the type of core\_data\_t\* is added to each scheduling function. Moreover, a log printing function "core\_log" is implemented instead of the "LOG" function in project2. Since two cores may invoke "core\_log" at the same time, a global lock has to be acquired inside "core\_log" to avoid mixing up two or more lines in the log sequence. And fortunately, the performance influence of this lock will be very minor, because the duration of the lock in "core\_log" is just a few milliseconds. In addition, each core will enter an idling mode rather than termination when there is no thread in its private queues. Because our load balance module may migrate threads from other cores to the idle one. Finally, the thread of each core will terminate by checking a global flag “is\_terminated” when all the user threads finish.

**4.2 Load Balancing Simulation**

Our simulator provides a load balance module to dispatch and balance the threads among processor cores. The dispatch function (see dispatcher\_execution() function in dispatch\_func.c) will send the thread newly created to a proper core using the two ways below.

1) If there is no core binding information predefined in the core binding data file (we will discuss the usage of this file in the next section) for the current thread, the thread will be sent to the next core.

2) If there is core binding information predefined, the current thread will be sent to the core binded to the thread.

For each user thread, it will wait for the event that the dispatcher assigns a core to it, and then start its own execution. Also, the logic of status changing of each user thread is exactly following the logic defined in project2.

The dispatch function will acquire the global lock and check whether there is a new thread created from time to time, and then invoke the load balance functions below.

1) “noop”, the simulator won’t migrate the threads from one thread to another thread at all.

2) “more\_frequent\_work\_stealing”, the simulator will keep migrating the threads from busy cores to the ones that are idling. The logic is that, if a core is idling (no thread in the readyQ and no thread is running) the balance function will try to fetch a thread from another core. CPU affinity can be considered.

3) “less\_frequent\_work\_stealing”, in order to avoid too frequent thread migration, this balance function only fetches the thread from much busier cores. To achieve this, the total threads (the sum of the number of all three private queues) should be two times of the destination core. Also, this function won’t check the status of each core as often as “more\_frequent\_work\_stealing”.

The concurrent issue is very important in the implementation of the balance function. When a migration from core A to core B occurs, the migration function will try to acquire the lock (using pthread\_mutex\_trylock) of core A first. If the balance thread can get the lock successfully and the core A is idling, then the migration function will try to acquire the lock of core B. If the balance thread can get the lock of core B successfully and core B has enough thread which can be shifted out, we will do the migration. Finally, we release the lock of B, and then release the lock of A.

For each user thread, when the migration occurs, it will release the lock of the previous core and acquire the lock of the new core.

**4.3 Processor Affinity Simulation**

In out implementation, the processor affinity features includes two related parts and they are implemented as follows.

1) User can define the core binding pair in an external data file for the user threads. This is simulation of CPU affinity.

2) If a thread is running in a core for a while (longer than a predefined time period), that means the processor has enough time to load the data of this thread into the core private cache, our simulator will provide a run-time “bonus” to this thread. This is for the effort simulation of CPU caches.

Note that, the dispatch function will always follow the binding pair defined in the data file. The format of the data binding file is:

<thread id> <core id>

<thread id> <core id>

…

Each line defines a pair of thread-core binding pair. If there is no bind info for a thread to a specific core, the simulator will migrate it with no affinity. Note that if there is no any bind info, the file would be left blank.

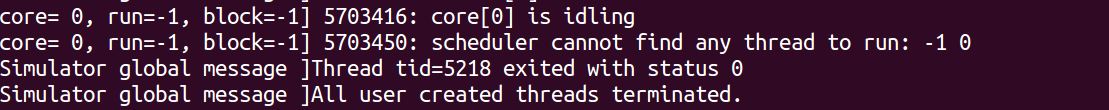
In addition, the run-time “bonus” will added when the status of a thread turns from “running” to other states.

1. **Simulation results**

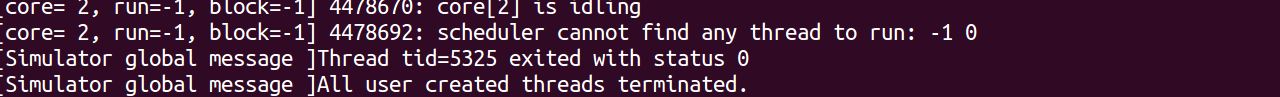
We run the simulator for 10 threads and 4 cores, as below (screenshots are result for one time run),

No balancing: (ten times run time average is ~5704400)

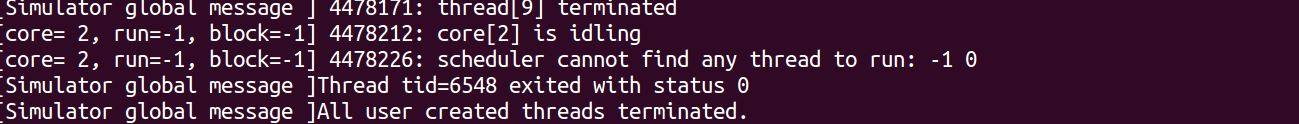




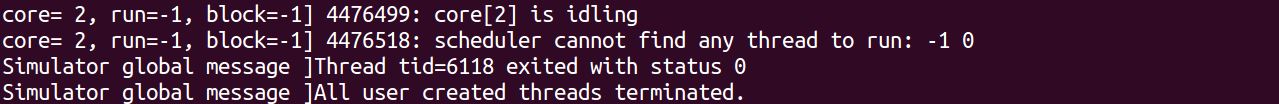
less\_frequent\_work\_stealing balancing without bind info: (ten times run time average is ~4480900)



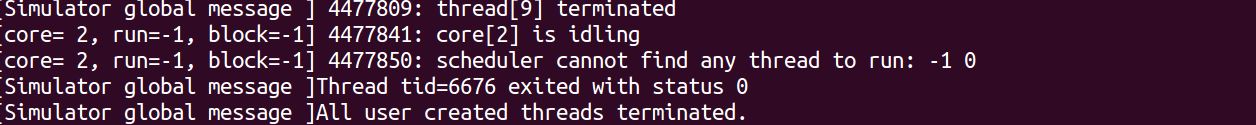
more\_frequent\_work\_stealing balancing without bind info: (ten times run time average is ~4532500)



less\_frequent\_work\_stealing balancing with bind info thread 1 to core 0: (ten times run time average is ~4479100)



more\_frequent\_work\_stealing balancing with bind info thread 1 to core 0: (ten times run time average is ~4492800)



Because the run time of threads is randomly generated, we run each of the above scenarios 10 times to get the average run time, and it is obvious to find out that balancing function can greatly shorten the total run time of the simulator. And balancing with different frequently could result in performance differences. In the 10 threads and 4 cores setting, the less\_frequent\_work\_stealing seems have a slightly better performance over the more\_frequent\_work\_stealing one. Also, the CPU affinity could further improve the performance if set properly.

1. **Conclusion**

In conclusion, our implementation realizes the concurrent run of multi-queue multi-core scheduling, the basic load balancing function and processor affinity, and helps us gain experience on these issues.

As we found out via the output of the simulator, the overall run time of multi-queue with load balancing function would be much less than the time without load balancing function. So the load balancing actually can boost the multi-core performance.

As we change the frequency of balancing, that is, using the scheme of less\_frequent\_work\_stealing will result in a run time difference with the scheme of more\_frequent\_work\_stealing. Basically, the run time of using less\_frequent\_work\_stealing scheme would have a better performance than using more\_frequent\_work\_stealing. More balance means more loss of cache effort, so the frequency of balancing should be well considered.

Moreover, the performance would be further improved if we bind one or two threads properly and then apply the less\_frequent\_work\_stealing scheme, and we found out that this is generally a good balance point for migration and CPU affinity.

Although the running time of our simulator could include overheads and may not be very accurate in collecting the actual simulation time of all threads, our simulator could reflect if not very accurate, the performance gain of employing load balancing and CPU affinity.

**References**

<http://pages.cs.wisc.edu/~remzi/OSTEP/cpu-sched-multi.pdf>

Wu Jia-Jun, “Research On Task Scheduling for Multi-core and Multi-thread processor,” Institute of Computing Technology of the Chinese Academy of Sciences, 2006.

<http://www.sei.cmu.edu/cyber-physical/research/timing-verification/multicore-scheduling.cfm>

<http://en.wikipedia.org/wiki/Processor_affinity>

<http://www.linuxjournal.com/article/6799>

[http://en.wikipedia.org/wiki/Load\_balancing\_%28computing%29](http://en.wikipedia.org/wiki/Load_balancing_(computing))

<https://software.intel.com/en-us/articles/load-balancing-between-threads>

<http://en.wikipedia.org/wiki/Amp>

<http://www.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/5_CPU_Scheduling.html>

<http://en.wikipedia.org/wiki/Asymmetric_multiprocessing>