Multiprocessor Scheduling

UNIkeEN

April 5, 2023

Contents

1 Introduction

In response to the need for more computing performance, modern computers often use multiprocessor systems or further multiprocessor systems. Compared to single processors, multiprocessor systems allow multiple programs to execute in parallel, and their scheduling algorithms are therefore more complex than single processors.

Multiprocessor scheduling is to reasonably allocate these processes to each processor according to a certain scheduling strategy, so that the processes can be executed orderly on each processor to achieve the goal of reducing the total execution time and improving the execution efficiency.

There are two main process allocation methods for multi-core processor: single queue scheduling strategy and multi-queue scheduling[1]. They may cause load imbalance and other problems. To solve these problems, scheduling algorithms such as work stealing were born.

Furthermore, it is necessary to consider the existence of heterogeneous environments and dependencies between processes.

2 Basic Strategy: Extended From Single Processor

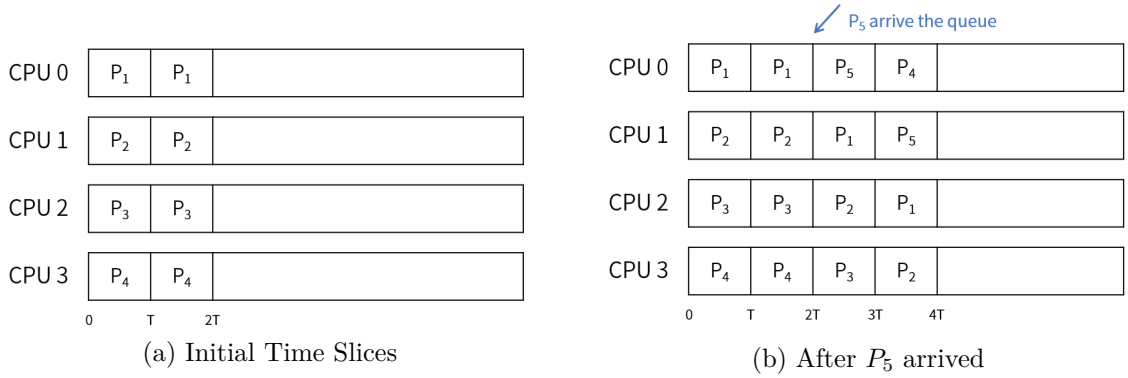
This section is based on the introduction of the symmetric multiprocessing (SMP) system and mainly focuses on isomorphic systems.

2.1 Single-Queue Scheduling

The simplest multiprocessor scheduling method is single queue scheduling, abbreviated as **SQMS**. All processes are in the same ready sequence, reusing various methods of single processor scheduling, such as FCFS, SJF, RR, etc. Each logical processor can be considered to be located in a resource pool, and if a processor is idle or completes its time slice, it is allocated to processes in the queue.

One of the problems with this strategy is the issue of cache affinity.

Suppose we have four processors, and there are four processes in the global queue initially. Using the RR algorithm, see **Fig. 1a** for the Gantt chart of the initial time slices. At this point, P_1 to P_4 have loaded caches in each processor to improve efficiency.



(a) Initial Time Slices

(b) After P_5 arrived

Fig. 1. Gantt Chart

However, suppose P_5 arrives at the sequence in a certain time slice, it will preempt CPU0. The next scheduling Gantt chart is shown in **Fig. 1b**. Because each CPU simply picks the next task to run from the single globally shared queue, each task ends up bouncing around from CPU to CPU, thus doing exactly the opposite of what would make sense from the standpoint of cache affinity.[1]

2.2 Multi-Queue Scheduling

Unlike the global single queue scheduling strategy, the multi queue multiprocessor scheduling (short for **MQMS**) strategy typically maps and allocates all processor cores and process queues to each other, generating a local set of tasks mapped to the kernel. This means that all tasks are divided into multiple independent queues, with each processor corresponding to a queue.

Each queue can use different scheduling rules, such as RR or FCFS algorithm. In this way, each processor schedule is independent of each other. This can avoid the problem of tasks repeatedly switching between processors in the aforementioned single queue scheduling method, and fully utilize local cache. The disadvantage is that it may lead to load imbalance and insufficient global utilization of the processor.[1, 2]

The above disadvantages can be exemplified as follows.

Assuming there are two processors and four tasks, P_1 and P_2 are initially assigned to CPU0, and P_3 and P_4 are assigned to CPU1. Both queues use RR, and no new processes come during the process.

If P_1 and P_2 are completed earlier than P_3 and P_4, it will cause CPU0 to be completely idle and CPU1 to work at full capacity for a certain period of time, resulting in load imbalance and low CPU utilization.

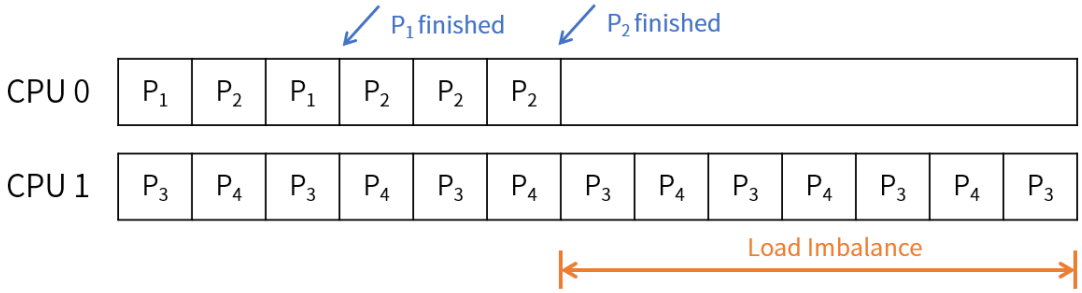


Fig. 2. Load Imbalance

2.3 Load Balance and Migration

When encountering such load imbalance in multi-queue scheduling, we can use migration methods to adjust the number of processes in each queue to achieve load balance.

Taking the above situation as an example, when CPU0 is idle, we can migrate any process on CPU1 to CPU0 as **Fig. 3** shown.

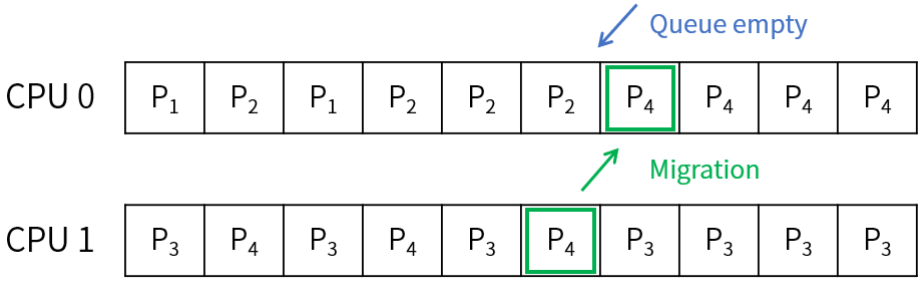


Fig. 3. Migration

The basic approach for system to enact such a migration is usually divided into push migration and pull migration.[3] For push migration, periodically check the load on each processor, and if it is not balanced, push the process from the heavily loaded processor to the smaller or idle processor.

Pull migration refers to the idle processor pulling down waiting tasks from the busy processor, similar to the **work stealing**[4], which was originally used for multithreaded scheduling. In multiprocessor process scheduling, a (source) queue that is low on jobs will occasionally peek at another (target) queue, to see how full it is. If the target queue is more full than the source queue, the source will "steal" one or more jobs from the target to help balance load.[1]

2.4 Load Balance Implementation in Linux

In the Linux kernel, load balancing is implemented through a scheduler. The scheduler is a subsystem of the Linux kernel that manages and schedules processes in order to achieve load balancing across multiple CPU cores. Inside the scheduler, there are some key functions that implement the specific logic of load balancing. The following are explanations of these functions:

- rebalance_tick(): This function is a key function in the Linux kernel that runs on every CPU core. Its role is to update the load on the current core and periodically check if some tasks need to be moved from other cores to the current core to achieve load balancing. The detailed code are as follows.

  (In the newer version, this function has renamed to run_rebalance_domains. The code below is from linux kernel-2.x)

```
static void rebalance_tick(int this_cpu, runqueue_t *this_rq,
             enum idle_type idle)
{
    unsigned long old_load, this_load;
    unsigned long j = jiffies + CPU_OFFSET(this_cpu);
    struct sched_domain *sd;

    /* Update our load */
    ...

    for_each_domain(this_cpu, sd) {
        unsigned long interval;
        ...
        interval = sd->balance_interval;
        if (idle != SCHED_IDLE)
            interval *= sd->busy_factor;

        /* scale ms to jiffies */
        interval = msecs_to_jiffies(interval);
        if (unlikely(!interval))
            interval = 1;

        if (j - sd->last_balance >= interval) {
```

```
            if (load_balance(this_cpu, this_rq, sd, idle)) {
                /* We've pulled tasks over so no longer idle */
                idle = NOT_IDLE;
            }
            sd->last_balance += interval;
        }
    }
}
```

The loop traverses the scheduling domain in each processor (A scheduling domain is a logical combination of a set of processors that can switch processes with each other). The interval interval for load balancing is calculated. if the processor is currently running a task, the interval is multiplied by the scheduling domain's preset parameter busy_factor to quantify how busy the processor is. The interval is then converted from milliseconds to jiffies to fit the Linux kernel's time units.

Next, if it has been more than the interval since the last time load balancing was performed, the load_balance function is called to perform load balancing. Finally, the timestamp of the last load balancing is updated.

- load_balance(): This function is responsible for moving tasks from other cores to the current core. It will first select a target core and move some tasks from the source core to the target core in order to achieve load balancing.
- load_balance_newidle(): This function will move tasks on one core to another core in order to achieve load balancing. It also has the ability to check if other free cores are available and move tasks to the free core.

Together, these functions implement the load balancing mechanism in the Linux kernel. They use complex algorithms and data structures to ensure that tasks can be rationally distributed among different CPU cores to maximize system performance and responsiveness.

3 Independent Tasks Scheduling Algorithms

For scheduling problems with multiple processors, in addition to considering which process to allocate to the processor (As mentioned in this section above), it is also necessary to consider which processor to allocate the process to, especially in heterogeneous processor system.

There are some algorithm design ideas for static task scheduling in heterogeneous environments, including heuristic algorithms such as OLB, MET, MCT, Min-min, A* algorithm, meta heuristic algorithm, etc. These design are also used in the field of distributed computing and so on.

3.1 Opportunistic Load Balancing

Opportunistic Load Balancing (short for **OLB**) assigns each task, in arbitrary order, to the next processor that is expected to be available, regardless of the task's expected execution time on that processor. The intuition behind OLB is to keep all processors as busy as possible. The main advantage of OLB is its simplicity.[5, 6]

3.2 Minimum Execution Time

In contrast to OLB, Minimum Execution Time (short for **MET**) assigns each task to the processor with the best expected execution time for that task, regardless of that processor's availability. The motivation behind MET is to give each task to its best processor. However, this can cause a severe load imbalance.[5, 6] If one processor do everything faster than any other processors and the process arrives slowly, all processes will choose one processor.

3.3 Min-Min Algorithm

The idea of the algorithm is to first map the small tasks and map them to the processors that execute fast.

The execution process is as follows:

1. Establish a task scheduling queue and initialize it as a collection of all tasks.
2. For all tasks in the task scheduling queue, calculate the earliest completion time mapped to all available machines.
3. Determine whether the task scheduling queue is empty. If it is not empty, execute (4); otherwise, execute step (8).
4. Find the task v_i with the earliest completion time and the corresponding processor p_j.
5. Map task v_i to machine p_j and remove it from the task scheduling queue.
6. Update the expected ready time of machine p_j.

7. Update the earliest completion time of each task on the task scheduling queue on machine p_j; Return to (3).

8. End scheduling and exit the program.

The time complexity of the algorithm is $O(p \times v^2)$ where p represents the number of heterogeneous processors and v represents the number of tasks.

The Min-Min algorithm essentially balances the matching principle and load balancing principle appropriately. The first Min refers to the minimum value of the fastest completion time for all unscheduled tasks, which is a calculation process that balances the workload of all tasks. The second Min refers to the minimum completion time of a task, which is the process of pursuing a perfect match for a single task.

Although the Min-Min algorithm has made efforts in load balancing, this directly leads to the particularly poor performance of the Min-Min algorithm in both cases[5, 7].

- There is a significant difference in performance among several processors on a processor chip. In this case, tasks are always assigned to the faster processor, while other processors are always idle, resulting in load imbalance and greatly extending the completion time of all tasks.
- There are many short assignments. The Min-Min algorithm will schedule long tasks only in the end. The completion time of all tasks will be too long and may cause starvation.

3.4 Max-Min Algorithm

The steps are roughly the same as Min-min, but select the task with the earliest and longest completion time for scheduling.

4 Task Scheduling with Dependencies

One of the great benefits of multi-core processor is that they support parallel computing. There may be dependencies between parallel computing processes, and the execution of processes may be sequential. There is communication overhead between the predecessor and successor processes.[8] The following will introduce a multiprocessor process scheduling method with dependency relationships.

4.1 Process Model

When conducting process scheduling research, it is necessary to establish a model that includes four parts: process model, processor model, scheduling algorithm, and process

mapping diagram.[9]

A common process model is a directed acyclic graph (short for **DAG**), where nodes represent processes and edges represent dependencies and communication relationships between connected processes.

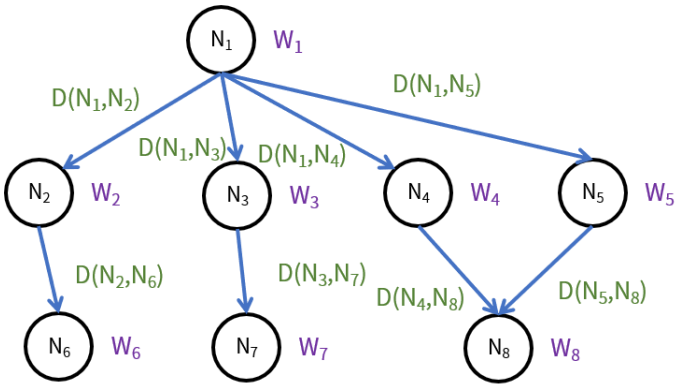


Fig. 4. Sample of DAG Graph

Each node in the graph represents a task, The matrix representing the execution time of the task on each processor is represented by the M corresponding to each node. and the directed edge represents the dependency relationship between the two connected processes. The directed edge from N_i to N_j indicates that process N_i can only execute process N_j after execution. The weight of the directed edge represents the communication cost required for transmitting messages and data between processes. If two processes are assigned to the same processor core for execution, their communication cost is zero. Otherwise, the communication cost is the weight of the directed edge.

4.2 HEFT Algorithm

The HEFT algorithm is a static task scheduling algorithm that belongs to the Bounded Number Processor (BNP) class. BNP algorithms are scheduling algorithms for the system that has a small number of heterogeneous processors with a static topology.

This type of algorithm is divided into two main steps:

- Task selection: calculates the priority of all tasks and sorts them to select the task with the highest priority.
- Processor Selection: involves scheduling the tasks selected in the previous step to the most suitable processor

The HEFT algorithm analyzes the DAG task model from bottom to top during the task selection process, calculates the priority of each task from bottom to top, and is

defined as $BRank$.

The $BRank$ consists of three parts: one part inherits from the successor node, one part represents the average execution time of the task on diffrent processors, and one part represents the average cost of switching between the task and its successor nodes on different processors.

$$BRank(v_i) = \overline{w_i} + \max_{v_j \in succ(v_i)} \{BRank(v_j) + \overline{c(i,j)}\}$$

In the process of selecting processors, the principle adopted is to allocate the tasks to be scheduled to the processors that can complete the tasks earliest, which is the principle of Early Finish Time (short for **EFT**). For most task scheduling algorithms, the earliest available time for processor P is when processor P completes the last task assigned to it.

Now, here are the algorithm.

Algorithm 1 HEFT Algorithm

1: Initialize a ready task queue with the entry task.
2: Calculate BRank values for all tasks in the given DAG starting from vexit.
3: Sort the tasks in the ready task queue in descending order of their BRank values.
4: **while** ready task queue is not empty **do**
5: Select the highest priority task v_i from the ready task queue.
6: **for** each processor $p_k \in P$ **do**
7: Calculate the earliest finish time of task v_i on processor p_k, denoted as $EFT(v_i, p_k)$.
8: **end for**
9: Assign v_i to the processor p_j that minimizes EFT.
10: Remove v_i from the ready task queue.
11: **end while**

The disadvantage of HEFT is that it only calculates the priority once and cannot dynamically adjust the priority of ready tasks based on existing task allocation schemes.

5 Analysis and Summary

Early processor performance development relied heavily on increases in individual processor frequency or instruction-level parallelism, but the growth from this approach gradually tapered off. People started to introduce multiprocessor and even multicore systems to improve computing efficiency.

The simplest multiprocessor systems are homogeneous and symmetric, and this is where my learning began. We can reuse ready queues and scheduling methods such as FCFS, SJF, RR, etc. in the single processor, while treating multiple processors as if they were placed in a single resource pool. This part includes both global single-queue and independent multi-queue implementations. The single-queue allows all processes to be located in the same ready queue, and any processor will be assigned a new task when it is free or completes a time slice, which may lead to cache affinity being broken. It has been shown that this is when using the simplest scheduling approach such as FCFS will achieve better performance.[1, 2] The multi-queue approach, on the other hand, initially places tasks in a separate queue of processors with fewer tasks, which can lead to uneven load, and the solution to this problem is migration.

The actual implementation of the OS will be more complex. 'rebalance_tick' function exists in the early Linux scheduler class, which executes on timer timing interrupts, determines whether the current processor's load balancing time point has been reached, traverses the busyness of each processor in the x-relevant scheduling domain, and moves tasks from the busier processor to the idle processor's waiting queue. This idea is similar to the work stealing algorithm, where a low-load processor regularly checks in and 'steals' its tasks at carefully designed intervals.[1, 4] The interval for this 'view' operation in Linux is variable and can vary with the busyness of the processor itself. At the same time, cache consistency issues should also be considered during the migration, which were not addressed in this study.

Another feature of multiprocessor systems is that they can be real-time. Single-processor architectures are also real-time, but they are based on slicing tasks and scheduling them in turn using time-slice rotation, which is not truly real-time. With a multi-core architecture, different processors can carry out tasks in parallel, which truly achieves the real-time requirement. At the same time, heterogeneous multiprocessor systems are also evolving, and in recent years processors such as Intel, Apple, and Qualcomm have started to adopt the 'E and P core' approach[10, 11], where different processor cores on the same physical chip have different performance and energy consumption, and the algorithms used to schedule tasks in parallel and heterogeneous situations are more complex. This leads me to the second part of my research.

Currently, task scheduling algorithms in heterogeneous processor environments can be divided into two main categories: static scheduling algorithms and dynamic scheduling algorithms. Static task scheduling uses some scheduling information (such as the execution time of different tasks on different processors, the amount of data transfer for switching between tasks, the dependencies between tasks, etc.) obtained by the application during the compilation process to develop a suitable allocation method for scheduling tasks to

the appropriate processor.[7] Static task scheduling takes place during the compilation of the program. On the contrary, dynamic task scheduling assigns tasks according to the current working condition of the system at the time of task assignment, and it occurs during the program runtime.

Depending on the task model used, static task scheduling algorithms can be considered in two categories, namely independent task scheduling algorithms and dependent task scheduling algorithms. Independent task scheduling algorithms have no dependencies between individual tasks, and whether each task is scheduled or not is only related to the scheduling algorithm and not to whether other tasks are scheduled or not. Dependent task scheduling algorithms have dependencies between tasks, similar to the critical path and task network learned in the data structure course, and the tasks enter the ready state only if their predecessors have finished executing.[7]

Independent task scheduling algorithms are divided into two types: heuristic algorithms and random search algorithms. Heuristic algorithms include MET algorithm, MCT algorithm, Min-Min algorithm, etc. The most important representative of randomized search algorithm is genetic algorithm. The Min-Min algorithm selects the task with the shortest completion time and the earliest completion time processor pair among all unscheduled tasks for scheduling, and although it may generate a starvation problem when there are more short tasks, it becomes the best overall performance and has become a benchmark for comparison of task scheduling algorithms. A study comparing the Min-Min algorithm with the genetic algorithm shows that the genetic algorithm scheduling effect is less improved under the condition of 16 processors with 512 independent tasks, but it takes 600 times more time than the former.[5]

For tasks with dependencies, the DAG graph can be used to represent the dependencies between processes, which can be implemented by methods such as CPOP and HEFT algorithms by finding critical paths and calculating the earliest completion time.

In summary, this study extends from single-processor to multi-processor systems, from homogeneous to heterogeneous, from independent to dependent tasks, and studies the task scheduling problem under multi-processors from shallow to deep. Each algorithm has its advantageous situation and disadvantageous situation, and there is no best algorithm, but only algorithms that are better in most situations. Some parameters such as the load balancing interval mentioned earlier require delicate balancing design, and such situations exist in various scheduling problems of operating systems such as IO scheduling, memory scheduling, etc. It can be seen that operating system design is a delicate and sophisticated discipline. During the research process, I also benefited from the knowledge of graph theory that I learned in data structures, algorithms, discrete mathematics, etc.

In this paper, I mostly use the term ‘task’ instead of ‘process’ to describe the scheduling

problem, because in the design of task scheduling algorithms, the unit of scheduling is just a dummy. In actual operating system implementations, scheduling may be done in terms of threads rather than processes as described in textbooks. Multi-processor systems are not limited to the familiar multi-core processors, but may also be multi-computer clusters in distributed computing.

Also, energy consumption is taken into account in the actual scheduling of the operating system, such as MacOS that place easier tasks on energy-efficient cores on its M-series chips (called Icestorm)[10] so that compared to the previous generation of x86 architecture products, the M1 Macbook Air has a significant improvement in battery life. I believe that in the future, heterogeneous multiprocessor systems will further develop rapidly, using different processing units in different work scenarios to maintain a balance between endurance and computing power.

References

[1] R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau, *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books, 1.00 ed., August 2018.

[2] J. T. Meehean, *Towards Transparent CPU Scheduling*. PhD thesis, University of Wisconsin–Madison, 2011.

[3] A. Silberschatz, P. Galvin, and G. Gagne, *Operating System Concepts*. Wiley, 2021.

[4] M. Frigo, C. E. Leiserson, and K. H. Randall, "The implementation of the cilk-5 multithreaded language," in *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pp. 212–223, 1998.

[5] "A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems," *Journal of Parallel and Distributed Computing*, vol. 61, no. 6, pp. 810–837, 2001.

[6] R. F. Freund, M. Gherrity, S. Ambrosius, M. Campbell, M. Halderman, D. Hensgen, E. Keith, T. Kidd, M. Kussow, J. D. Lima, *et al.*, "Scheduling resources in multi-user, heterogeneous, computing environments with smartnet," in *Proceedings Seventh Heterogeneous Computing Workshop (HCW'98)*, pp. 184–199, IEEE, 1998.

[7] K. Liu, *Study on Task Scheduling Problem in Heterogeneous Environment*. PhD thesis, Hangzhou Dianzi University, 2009.

[8] J. Wang, *Research on Process Scheduling Algorithm Based on Multi-core Processors*. PhD thesis, Harbin Engineering University, 2013.

[9] M.-Y. Wu, W. Shu, and H. Zhang, "Segmented min-min: A static mapping algorithm for meta-tasks on heterogeneous computing systems," in *Proceedings 9th Heterogeneous Computing Workshop (HCW 2000)(Cat. No. PR00556)*, pp. 375–385, IEEE, 2000.

[10] H. Oakley, "Making the most of apple silicon power: 1 m-series chips are different." `https://eclecticlight.co/2022/10/03/making-the-most-of-apple-silicon-power-1-m-series-chips-are-different/`, 2022.

[11] Intel, "Brief: 12th gen intel® core™ desktop processor." `https://www.intel.sg/content/www/xa/en/products/docs/processors/core/12th-gen-core-desktop-brief.html/`, 2021.