Survey on Scheduling Algorithms for Multiprocessing Systems

Dayton Bishop
Department of Computer Sciences
University of Salzburg, Austria
dbishop@cosy.sbg.ac.at

ABSTRACT
The goal of this paper is to give an overview of the different types of scheduling algorithms for Multiprocessing systems. To better understand the purpose and difficulties concerning schedulers running on multiprocessor systems, the first part of this paper will give a rudimentary introduction to the thematic by explaining scheduling on a single processor. Then the transition to the multiprocessor systems will be made in order to compare the different approaches that were made multiprocessing systems.

Next a detailed analysis of the work stealing algorithm will be presented, comparing it to the other known approaches.

Finally, the systems the incorporate the presented work stealing algorithms will be presented.

Why Scheduling in general

The basic functionality of software on a system is that code is stored in memory that should be executed on the processor. The operating system handles the requests of different programs for the execution. The part of the operating system that is responsible for the deciding what task to execute is the scheduler.

Since at any one time there are multiple tasks that the processor should compute, the idea to switch between these tasks in order to allow all the tasks to make progress on their executions arose, thus called scheduling. Some of these tasks may require that another tasks be executed first, a so called dependency. Therefore some tasks could be executed simultaneously, where as others would have to executed in file.

Also there are some tasks that should be executed immediately, they would be given a higher priority by the user, and others that can take more time to finish, a lower priority task. Since basically every system would like to differentiate between these two types of tasks, most schedulers are priority driven.

So far the consideration was directed at single processor systems. Of course there are new considerations for the implementation for a scheduling algorithm running on a multiprocessor system. Multiprocessors have been accepted as means to improve the speed, performance and enhance reliability or availability of the computation. These systems however require a new approach that adds a whole new level of complexity to the operating system design and implementation.

Scheduling algorithms for a single processor

This is a presentation of the concepts of the scheduling algorithms. These are the basis for the designs of the scheduling algorithms for multiprocessor systems. There are three rudimentary types [36] of scheduling in an operating system.

The first is known as the long term scheduler. This scheduler decides what tasks can be scheduled in order to meet certain time constraints.

The second is the most common and are known as the mid term scheduling algorithms.

These algorithms swap processes in and out of the memory when processes have been blocked, are waiting or require more space than available in the memory. Therefore these schedulers usually are implemented with virtual memory.

The last is the short term scheduler that decides after each interrupt (clock, IO or a system call), what process to execute next. Therefore this scheduler will latest at then end of each time slice decide what process to execute next.

A combination of these different types is also thinkable. Each of these different types can also be implemented as a preemptive or a non preemptive scheduler meaning, that the processes running can be interrupted in order to execute another process.

Here are a few examples of common scheduling designs [37]:

- Earliest Deadline First
- Round Robin
- FIFO
- Priority
- SJF (Shortest Job First)

Scheduling algorithms for a multi processor

Based on the scheduling algorithms for single processors, new algorithms were developed for the multi processor environment [31]. These new designs are confronted with new requirements such as scalability, work load balancing [23, 30] and even the consideration of heat produced by the system [19].

The scalability issue is discussed in the PhD thesis by J. Mohan [20] where the problem of designing a flexible runtime structure is addressed. The question analyzed is whether it is possible to have any number of run time queues used by an arbitrary number of processors.

Work load distribution [35] is also the task of the scheduler in the multiprocessor environment. In order to fully utilize the potential of a multiprocessor system, all the processors should be shouldering an equal workload.

Basically the process scheduling on parallel machines is a NP - hard problem [22].

The schedulers on multiprocessor systems can be static or dynamic, distributed or centralized in their implementation.

In the static version, each process is assigned a fixed amount of processors it receives each time it is executed.

The dynamic version assigns the number of processors, each time the process is scheduled. The tradeoff that the dynamic version makes, is the overhead needed to assign the processors anew each time. The static implementation is, on the other hand, not always able to use the full processing power at hand.

A comparison was made [8, 42] with the result, that the dynamic implementation is better if the overhead is small and the workload is high.

A centralized scheduler means that the run - time scheduler resides on a dedicated processor. The distributed scheduler can be invoked from different processors [8].

For the basic understanding of the different approaches of schedulers, it is necessary to realize that in comparison to single processor systems, there are multiple ways to construct multiprocessors. For scheduling we are mainly interested in the type of memory access used in the system. There are NUMA (non uniform memory access) and UMA (uniform memory access) access methods to the memory.

A NUMA architecture, in contrast to the UMA that requires the memory not only be accessed in the same fashion but also in the same time therefore being limited to around 32 processors [37], therefore leads to totally different memory management design choices. Memory consistency is only guaranteed for local memory and caches, or must be specifically enforced by shared memory. This architectural form exposes the existing memory architecture to programmers. The benefit is that the programmer can address what memory access is to be used, the more expensive remote memory accesses or the fast local access. Therefore also the number of context switches can be minimized and potential contention can be avoided, whereas these considerations do not include the cache memory [29].

Considerations for a scheduler

Even though scheduling design has been well researched, an evaluation of the scheduling algorithms is hard to come by. Most papers have no comparison at all, where as other papers at least compare themselves to one of the other known schedulers. The trouble in analyzing the schedulers is that maximization of the system utilities is not directly observable. However the byproduct of a quality scheduler can be measured such as performance, fairness and predictability [41]. These measures however are interdependent thus only allowing weak statements like “one of the factors has increased, whereas the others are held equal”. The representation is often to be found as a matrix.

Performance is the most common quality evaluated in the comparison of schedulers. Performance is often measured using variation of response time [9, 33]. That is the time a task needs, before its completion. Users prefer a quick response time, even though the exact correlation is not identifiable [10]. This quickly becomes clear, when we use a scheduler that exclusively executes fast running jobs when possible may easily achieve a quick response time thereby creating a low productivity in starving the heavier processes.

Therefore the second measure is fairness that should be supported on a chip multiprocessor architecture, meaning that each task is treated equal and no tasks are starved. This effect is difficult to measure but often implemented by queues [21] or time slices [11] that are assigned to each of the processes or threads. Fairness is often a tradeoff to performance, even though the relationship is not explicit.

Predictability is required by real time systems and is the users expected execution time versus the actual execution time [40]. In order achieve predictability, resources have to be assigned and job execution times have to be anticipated.

Here the functionality of the scheduling algorithms is explained that evolved from the designs current on the single processors.

The Single shared ready Queue

This design is that all processes are stored in a global ready queue [22]. Each processor then retrieves a process from the ready queue when free and returns the processes that are blocked or waiting. This approach requires a UMA, but simplifies the implementation of scheduling policies so far, that approaches used single processors, such as FIFO (first in first out), FCFS (first come first serve) and SJF(shortest job first) can be used [41].

Backfilling

The backfilling algorithm [7, 17] divides the jobs into categories by giving the jobs a priority and certain predefined guidelines. There are different approaches in the implementation of the backfilling algorithm.

The non - backfilling simply executes the job with the highest priority that is able to execute. If one of the processors of the highest priority job is blocked then the scheduler waits, until the resources are available.

The classical backfilling scheduler is similar to the non - backfilling scheduler, except if the higher priority job is blocked, the resources are filled with lower priority jobs.

The preemptive – backfill version now is also allowed to preempt the lower priority tasks to execute the higher priority tasks. There are three basic approaches in deciding what task to execute. The first is the highest priority task that uses the available resources is used. The second is to take the job that uses the resources most effectively without considering the priority. The third is to combine different jobs in order to maximize effectivity.

The multiple queue backfilling algorithm [18] additionally introduces partitions. These partitions are each assigned different execution times, for example the first partition may accept jobs that need 100 ms to 1s. Each partition has its own priority listing. Now if a job from the longest execution partition is not able to execute because of the lack of resources, the next lower partition is considered.

The relaxed backfill [39] algorithm is similar to the preemptive backfilling version, except that the preemption is executed after the higher priority process has waited a predefined amount of time.

Gang Scheduling

The gang scheduling algorithm is used to achieve a high level of parallelism for a single job. This algorithm is especially useful when the different processes of this job need to communicate often. This way the system can evade some context switches. Also it allows the processes to interact using the busy lock, thereby evading the risk of waiting for a task that is currently not running.

There are many possible ways to implement the gang scheduling algorithm. I will quickly introduce three policies employed in a distributed system [16].

The considerations that can be included in the scheduling algorithm are the different ways to schedule gangs. Gangs are all the processes of one job.

The adopted first come first serve (AFCFS) method attempts to schedule a job as soon as a assigned processor is available. If there are not enough processors to assign a large job in the front of the queue, smaller jobs are scheduled. This results in the smaller jobs being favored above the larger jobs.

The largest gang first served (LGFS) method places the largest jobs on the top of the processor queues. The job for which the assigned processors are available is executed first, whereas the job on top is considered first and then the following jobs on the work queue. This method is especially beneficiary for massively multiprocessing environments such as supercomputer centers

First come first serve (FCFS) method is fair to the jobs in the wait queue, however it mostly results in suboptimal performace.

Shortest time first (STF) method chooses the jobs that require the shortest I/O time. Since this method chooses the fastest I/O services this method is expected to yield the most throughput. This assumes that the I/O times are known in advance and therefore can only be executed if this information is available.

There have been many different policies for gang scheduling. To name a few Feitelson and Rudolph [11], Feitelson and Rudolph [12], Feitelson and Jette [13], Karatza [15], Karatza [16], Sobalvarro and Weihl [32], Squillante [34] and Wang [38].
As parallel programs become widespread, it will become increasingly difficult to keep parallel program execution well adjusted, so that users will demand the ability to use the CPUs with maximum efficiency while executing at high multiplicity. A more in depth research on this scheduling design was conducted by Ousterhout[24] who proposed the different subcategories of this design, the matrix, continuous and undivided approach.

Round Robin

There are two different versions of multiprocessor round robin schedulers [22]. The first version is based on a shared memory system, where the round robin scheduler is used in much the same way as the round robin scheduler on the single processor system. The new processes that arrive are put at the end of the global queue and the tasks are then executed. The second schedules jobs instead of processes, where just as before each job execution gets a time slice. This version is often implemented so that all processors execute the one job the scheduler has assigned the time slice to.

For a more detailed explanation of the second round robin approach there are a few additional considerations that have to taken into account. First we have the number of processes *q* in a job. The second is we have a certain number of processors *p* at our disposal. If the number of processes in a job is less or equal the number of processors, the solution is simple, each processor is give on of the processes.

If the number of processes exceeds the number of processors, we have two different ways in which to resolve the conflict. The first is that the larger number of processes is devided equally amoung the processors.

The second is that an additional round robin scheduling is introduced, that distributes *p* processes from the job for one quanta of execution time.

The main problems that have to be solved when using this round robin are that frequent context switches can occur, when the number of processes is very high.

On the other hand if the number of processes does not exceed the number of processors, the affinity for a certain processor may become a problem.

Hands off scheduling

Another consideration that is made in the choice of a scheduler is if the system should support deadlines (e.g. in a real time system). The hands off scheduler [22] is a kernel level scheduler, that can be manipulated directly by the user.

There are two fashions in which the user can address the scheduler. The first is the discouragement hint, that hints that the particular thread should be discouraged when considering what task to execute. There are three levels, that can be given, mild, strong or weak.

The second fashion in which the user can influence, is the hands - off hint. It is used to have the scheduler run a certain thread, where as the current thread hands off the processor to another thread without creating a scheduler interference.

These again are split into three subcategories:

1. Hard real time systems
2. Firm real time systems
3. Soft real time systems

Distributing the work load is of course the goal every multiprocessor environment. Load balancing is in general not worth the extra effort, as only a small gain in the execution time of tasks can be achieved and is mostly outweighed by the effort expended to maintain the load balance. More reasonable approaches are the work sharing and work stealing algorithms.

The work sharing algorithm

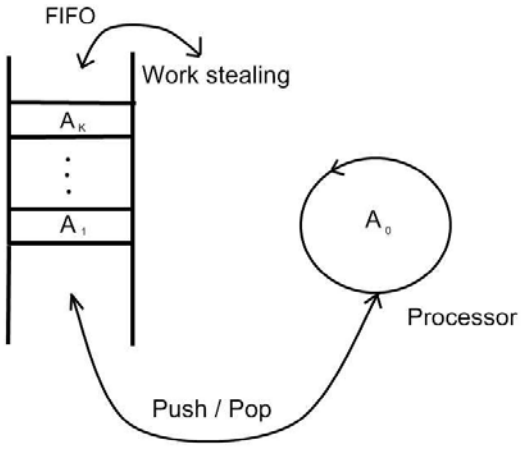
The work sharing algorithm performs load sharing by using a global scheduling. If processors have too many tasks to handle, the load balancer offloads some of the work to processors less busy or even in idle [4].

The work stealing algorithm

The functionality of the work stealing algorithm [3] is that instead of all the processors trying to share the processes held in a central queue each processor has its own queue. At the beginning of a job, one of the queues of the processors receives the first process of a job. The processor then starts the execution of the processes thereby spawning other processes.

As soon as the first process is initialized, the other processors start trying to steal the work that is on the stack of the processor that the job was initialized on.

As soon as the process being executed waits for an IO operation or waits on another process, it is pushed to the processor's own stack (picture below). Then this processor also tries to steal workload from the other processors.



Systems that incorporate the presented scheduling algorithms

There are many operating systems that have been developed for a multiprocessor environment. The systems presented here use some of the schedulers explained above and should provide a better insight into the functionality of these schedulers.

- Cilk [6] is an operating system that uses the work stealing algorithm.
- HYDRA [5, 25, 26, 27, 28] uses the round robin scheduling
- PRESTO [1, 2] uses a single shared ready queue with a pool of threads

CONCLUSION

In the field of multiprocessor scheduling there are yet many improvements to be achieved with promising results. Even though in the scheduler is such a small component when looking at an entire operating system it still is greatly responsible for the systems efficiency.

This paper provides a short overview of scheduling algorithms that have appeared in journals and conferences. I hope this survey may be of use when interested in the different approaches that have so far been suggested. I conclude by apologizing to the many system developers that I did not mention in this paper.

REFERENCES

1. B. Bershad, E. Lazowska, and H. Levy. Presto: A system for object-oriented parallel programming. Software: Practice and Experience, 18(8):713–732, August 1988.
2. B. Bershad, E. Lazowska, H. Levy, and D. Wagner. An open environment for building parallel programming systems. In Proceedings of the Symposium on Parallel Programming: Experience with Applications, Languages and Systems, pages 1–9, July 1988.
3. Robert D Blumofe and Charles E. Leiserson, Scheduling Multithreaded Computations by Work stealing, MIT Laboratory for Computer Science
4. Barbara M. Chapman, Lei Huang, Haoqiang Jin, Gabriele Jost, and Bronis R. de Supinski, Toward Enhancing OpenMP's Work-Sharing Directives
5. Jr. E.M. Chaves, P.C. Das, T.L. LeBlanc, B.D. Marsh, and M.L. Scott. Kernel-kernel communication in a shared-memory multiprocessor. Concurrency: Practice and Experience, 5(3):171–192, May 1993.
6. Cilk http://www.supertech.csail.mit.edu/cilk/manual-5.3.2.pdf
7. B. Cramer, Universität Paderborn, Single – site scheduling
8. G. Dimitriou, Simulation of static an dynamic task scheduling on multiprocessor systems, PhD thesis at the Universtiy of Illinois at Urbana – Champaign, 1994
9. C. Ernemann, V. Hamscher, U. Schwiegelshohn, A. Streit, and R. Yahyapour. On Advantages of Grid Computing for Parallel Job Scheduling. In *Proceedings of the 2nd IEEE International Symposium on Cluster Computing and the Grid (CC-GRID 2002*
10. Feitelson D. G., L. Rudolph, U. Schwiegelshohn, K. C. Sevcik, and P. Wong. Theory and Practice in Parallel Job Scheduling. In D. G. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies forParallel Processing*, pages 1–34. Springer Verlag, 1997.
11. Feitelson D. G. and L. R. L. Scheduling. Parallel Job Scheduling: Issues and Approaches. In D. G. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing – IPPS'95 Workshop*, volume 949, pages 1–18. Springer, 1995.
12. Feitelson D.G. and Rudolph L. 1996. "Evaluation of Design Choices for Gang Scheduling Using Distributed Hierarchical Control". *Journal of Parallel and Distributed Computing*, Academic Press, New York, USA, Vol. 35. Pp18-34.
13. Feitelson D.G. and Jette M.A. 1997, "Improved Utilisation and Responsiveness with Gang Scheduling". In *Job Scheduling Strategies for Parallel Processing*, *Lecture Notes in Computer Science*, Springer-Verlang, Berlin, Germany. Vol. 1291. Pp238-26.
14. Karatza H. D., Performance analysis of gang scheduling in a distributed system under processor failures
15. Karatza H.D. 1999a, "A Simulation-Based Performance Analysis of Gang Scheduling in a Distributed System". In *Proc. of the 32nd Annual Simulation Symp.* (San Diego, CA, USA, April) IEEE Computer Society, Los Alamitos, CA, USA. Pp26-33.
16. Karatza H.D. 2000a, "Gang Scheduling and I/O Scheduling in a Multiprocessor System". In *Proc. of 2000 Symp. on Performance Evaluation of Computer and Telecommunication Systems* (Vancouver, Canada, July) SCSI, San Diego, CA, USA. Pp245-252.
17. Barry G. Lawson, Evgenia Smirni Department of Computer Science College of William and Mary Williamsburg, VA 23187-8795, USA, Multiple-queue Backfilling scheduling with Priorities and Reservations for Parallel Systems.
18. Barry G. Lawson, Evgenia Smirni: Multiple–Queue backfilling Scheduling with Priorities and Reservations for Parallel Systems in Job Scheduling Strategies for Parallel Processing 8th. Workshop 2002

19. A. Merkel, F. Bellosa. EUROSYS 2006, Balancing Power Consumption in Multiprocessor Systems.
20. Joseph Mohan, Performance of parallel programs: Model and Analysis. PhD thesis, Computer science department, Carnegie – Mellon University. Pittsburgh, Pa, July 1984
21. A. Mu'alem and D. Feitelson. Utilization, Predictability, Workloads, and User Runtime Estimates in Scheduling the IBM SP2 with Backfilling. In *12th Intl.Parallel Processing Symposium*, pages 542–546, April 1998.
22. B. Mukherjee, K. Schwan, P. Gopinath, A survey of Multiprocessor Operating systems, College of computing, Georgia Institute of Technology
23. M. Norman, P. Thanisch, Models of Machines and Computations for Mapping in Multicomputers, Edinburgh Parallel computing centre, University of Edinbourgh
24. J. Ousterhout, Scheduling techniques for concurrent systems. In *Proceedings of Distributed Computing Systems Conference*, pages 22–30, October 1982.
25. M. Scott, T. Leblanc, and B. Marsh. Design rationale for psyche, a general purpose multiprocessor operating system. In Proceedings of the 1988 International Conference on Parallel Processing (V II - Software), pages 255–262, August 1988.
26. M. Scott, T. Leblanc, and B. Marsh. Evolution of an operating system for large scale shared-memory multiprocessors. Technical Report TR 309, Department of Computer Science, University of Rochester, March 1989.
27. M. Scott, T. Leblanc, and B. Marsh. Multi-model parallel programming in psyche. In proceedings of the Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pages 70–78, March 1990.
28. M. Scott, T. Leblanc, B. Marsh, T. Becker, C. Dubnicki, E. Markatos, and N. Smithline. Implementation issues for the psyche multiprocessor operating system. Computing Systems, 3(1):101–137, Winter 1990.
29. Seongbeom Kim, Dhruba Chandra and Yan Solihin, Fair Cache Sharing and Partitioning in a Chip Multiprocessor Architecture in the Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT), 2004.
30. Shirazi, Hurson, and Kavi, Scheduling and Load Balancing in Parallel and Distributed Systems [Shirazi et al., 1995].
31. Singhal and Shivaratri, Advanced Concepts in Operating Systems [Singhal and Shivaratri, 1994].
32. Sobalvarro P.G. and Weihl W.E. 1995, "Demandbased Coscheduling of Parallel Jobs on Multiprogrammed Multiprocessors". In *Job Scheduling Strategies for Parallel Processing, Lecture Notes in Computer Science*, Springer-Verlang, Berlin, Germany. Vol. 949. Pp106-126.
33. Sobalvarro P. G., S. Pakin, W. E. Weihl, and A. A. Chien. Dynamic Coscheduling on Workstation Clusters. *Lecture Notes in Computer Science*, 1459:231–256, 1998.
34. Squillante M.S., Wang F. and Papaefthymioy M. 1996, "Stochastic Analysis of Gang Scheduling in Parallel and Distributed Systems", *Performance Evaluation*, Elsevier, Amsterdam, Holland, Vol. 27&28 (4). Pp273-296.
35. Squillante M.S. and R. Nelson, Analysis of task migration in shared-memory multiprocessor scheduling, Proceedings of the 1991 ACM SIGMETRICS conference on Measurement and modeling of computer systems
36. Stallings, William (2004). *Operating Systems Internals and Design Principles (fifth international edition)*. Prentice Hall. ISBN 0-13-147954-7
37. Tanenbaum, A.S.: Modern Operating Systems Prentice Hall, Prentice Hall, 952 pages, 2001
38. Wang F., Papaefthymiou M. and Squillante M.S. 1997, "Performance Evaluation of Gang Scheduling for Parallel and Distributed Systems". In *Job Scheduling for Parallel Processing, Lecture Notes in Computer Science*, Springer-Verlang, Berlin, Germany. Vol. 1291. Pp184-195.
39. William A. Ward et al.: Scheduling Jobs on Parallel Systems Using a Relaxed Backfill Strategy in Job scheduling Strategies for Parallel Processing 8th. Workshop 2002
40. A. Wierman and M. Harchol-Balter. Classifying scheduling policies with respect to higher moments of conditional response time. In *SIGMETRICS '05: Proceedings of the 2005 ACM SIGMETRICS International conference on Measurement and modeling of computer systems*, pages 229–240, New York, NY, USA, 2005. ACM Press.
41. J. Weinberg, Job Scheduling on parallel systems
42. J. Zahorjan and C McCann, Processor scheduling in shared memory multiprocessors, In proceedings of the 1990 ACM SIGMETRICS conference on measurement and modeling of computer systems, pages 214 - 225, May 1990