Scheduling (computing)

From Wikipedia, the free encyclopedia

In computer science, **scheduling** is the method by which threads, processes or data flows are given access to system resources (e.g. processor time, communications bandwidth). This is usually done to load balance and share system resources effectively or achieve a target quality of service. The need for a scheduling algorithm arises from the requirement for most modern systems to perform multitasking (executing more than one process at a time) and multiplexing (transmit multiple data streams simultaneously across a single physical channel).

The scheduler is concerned mainly with:

- Throughput - The total number of processes that complete their execution per time unit.
- Latency, specifically:
  - Turnaround time - total time between submission of a process and its completion.
  - Response time - amount of time it takes from when a request was submitted until the first response is produced.
- Fairness - Equal CPU time to each process (or more generally appropriate times according to each process' priority and workload).
- Waiting Time - The time the process remains in the ready queue.

In practice, these goals often conflict (e.g. throughput versus latency), thus a scheduler will implement a suitable compromise. Preference is given to any one of the above mentioned concerns depending upon the user's needs and objectives.

In real-time environments, such as embedded systems for automatic control in industry (for example robotics), the scheduler also must ensure that processes can meet deadlines; this is crucial for keeping the system stable. Scheduled tasks can also be distributed to remote devices across a network and managed through an administrative back end.

Contents

Types of operating system schedulers

Operating systems may feature up to three distinct types of scheduler, a *long-term scheduler* (also known as an admission scheduler or high-level scheduler), a *mid-term or medium-term scheduler* and a *short-term scheduler*. The names suggest the relative frequency with which these functions are performed. The scheduler is an operating system module that selects the next jobs to be admitted into the system and the next process to run.

Process scheduler

Long-term scheduling

The long-term, or admission scheduler, decides which jobs or processes are to be admitted to the ready queue (in the Main Memory); that is, when an attempt is made to execute a program, its admission to the set of currently executing processes is either authorized or delayed by the long-term scheduler . Thus, this scheduler dictates what processes are to run on a system, and the degree of concurrency to be supported at any one time - i.e.: whether a high or low amount of processes are to be executed concurrently, and how the split between I/O intensive and CPU intensive processes is to be handled. The long term scheduler is responsible for controlling the degree of multiprogramming . In modern operating systems, this is used to make sure that real time processes get enough CPU time to finish their tasks. Without proper real time scheduling, modern GUIs would seem sluggish.

Long-term scheduling is also important in large-scale systems such as batch processing systems, computer clusters, supercomputers and render farms. In these cases, special purpose job scheduler software is typically used to assist these functions, in addition to any underlying admission scheduling support in the operating system..

Medium term scheduling

Scheduler temporarily removes processes from main memory and places them on secondary memory (such as a disk drive) or vice versa. This is commonly referred to as "swapping out" or "swapping in" (also incorrectly as "paging out" or "paging in"). The medium-term scheduler may decide to swap out a process which has not been active for some time, or a process which has a low priority, or a process which is page faulting frequently, or a process which is taking up a large amount of memory in order to free up main memory for other processes, swapping the process back in later when more memory is available, or when the process has been unblocked and is no longer waiting for a resource. [Stallings, 396] [Stallings, 370]

In many systems today (those that support mapping virtual address space to secondary storage other than the swap file), the medium-term scheduler may actually perform the role of the long-term scheduler, by treating binaries as "swapped out processes" upon their execution. In this way, when a segment of the binary is required it can be swapped in on demand, or "lazy loaded". [Stallings, 394]

Short-term scheduling

The short-term scheduler (also known as the CPU scheduler) decides which of the ready, in-memory processes are to be executed (allocated a CPU) after a clock interrupt, an I/O interrupt, an operating system call or another form of signal. Thus the short-term scheduler makes scheduling decisions much more frequently than the long-term or mid-term schedulers - a scheduling decision will at a minimum have to be made after every time slice, and these are very short. This scheduler can be preemptive, implying that it is capable of forcibly removing processes from a CPU when it decides to allocate that CPU to another process, or non-preemptive (also known as "voluntary" or "co-operative"), in which case the scheduler is unable to "force" processes off the CPU.

A preemptive scheduler relies upon a programmable interval timer which invokes an interrupt handler that runs in kernel mode and implements the scheduling function.

Dispatcher

Another component that is involved in the CPU-scheduling function is the dispatcher. The dispatcher is the module that gives control of the CPU to the process selected by the short-term scheduler. This function involves the following:

- Switching context
- Switching to user mode
- Jumping to the proper location in the user program to restart that program.

Dispatcher analyses the values from Program counter and fetches instructions, loads data into registers.

The dispatcher should be as fast as possible, since it is invoked during every process switch. During the context switches, the processor is idle for a fraction of time. Hence, unnecessary context switches should be avoided. The time it takes for the dispatcher to stop one process and start another running is known as the **dispatch latency**. [Galvin, 155].

Network scheduler

Main article: Network scheduler

Disk scheduler

Main article: I/O scheduling

Job scheduler

Main article: Job scheduler

Examples are cron, at, systemd.

Scheduling disciplines

Scheduling disciplines are algorithms used for distributing resources among parties which simultaneously and asynchronously request them. Scheduling disciplines are used in routers (to handle packet traffic) as well as in operating systems (to share CPU time among both threads and processes), disk drives (I/O scheduling), printers (print spooler), most embedded systems, etc.

The main purposes of scheduling algorithms are to minimize resource starvation and to ensure fairness amongst the parties utilizing the resources. Scheduling deals with the problem of deciding which of the outstanding requests is to be allocated resources. There are many different scheduling algorithms. In this section, we introduce several of them.

In packet-switched computer networks and other statistical multiplexing, the notion of a **scheduling algorithm** is used as an alternative to first-come first-served queuing of data packets.

The simplest best-effort scheduling algorithms are round-robin, fair queuing (a max-min fair scheduling algorithm), proportionally fair scheduling and maximum throughput. If differentiated or guaranteed quality of service is offered, as opposed to best-effort communication, weighted fair queuing may be utilized.

In advanced packet radio wireless networks such as HSDPA (High-Speed Downlink Packet Access) 3.5G cellular system, **channel-dependent scheduling** may be used to take advantage of channel state information. If the channel conditions are favourable, the throughput and system spectral efficiency may be increased. In even more advanced systems such as LTE, the scheduling is combined by channel-dependent packet-by-packet dynamic channel allocation, or by assigning OFDMA multi-carriers or other frequency-domain equalization components to the users that best can utilize them.

First in first out

Main article: First In First Out

Also known as ***F**irst **C**ome, **F**irst **S**erved* (FCFS), is the simplest scheduling algorithm, FIFO simply queues processes in the order that they arrive in the ready queue.

- Since context switches only occur upon process termination, and no reorganization of the process queue is required, scheduling overhead is minimal.
- Throughput can be low, since long processes can hold the CPU
- Turnaround time, waiting time and response time can be high for the same reasons above
- No prioritization occurs, thus this system has trouble meeting process deadlines.
- The lack of prioritization means that as long as every process eventually completes, there is no starvation. In an environment where some processes might not complete, there can be starvation.
- It is based on Queuing
- Here is the C-code for FCFS (http://scanftree.com/operating-system/fcfs)

Shortest remaining time

Main article: Shortest remaining time

Similar to ***S**hortest **J**ob **F**irst* (SJF). With this strategy the scheduler arranges processes with the least estimated processing time remaining to be next in the queue. This requires advanced knowledge or estimations about the time required for a process to complete.

- If a shorter process arrives during another process' execution, the currently running process may be interrupted (known as preemption), dividing that process into two separate computing blocks. This creates excess overhead through additional context switching. The scheduler must also place each incoming process into a specific place in the queue, creating additional overhead.
- This algorithm is designed for maximum throughput in most scenarios.
- Waiting time and response time increase as the process's computational requirements increase. Since turnaround time is based on waiting time plus processing time, longer processes are significantly affected by this. Overall waiting time is smaller

than FIFO, however since no process has to wait for the termination of the longest process.
- No particular attention is given to deadlines, the programmer can only attempt to make processes with deadlines as short as possible.
- Starvation is possible, especially in a busy system with many small processes being run.
- This policy is no more in use.
- To use this policy we should have at least two processes of different priority

Fixed priority pre-emptive scheduling

Main article: Fixed priority pre-emptive scheduling

The OS assigns a fixed priority rank to every process, and the scheduler arranges the processes in the ready queue in order of their priority. Lower priority processes get interrupted by incoming higher priority processes.

- Overhead is not minimal, nor is it significant.
- FPPS has no particular advantage in terms of throughput over FIFO scheduling.
- If the number of rankings is limited it can be characterized as a collection of FIFO queues, one for each priority ranking. Processes in lower-priority queues are selected only when all of the higher-priority queues are empty.
- Waiting time and response time depend on the priority of the process. Higher priority processes have smaller waiting and response times.
- Deadlines can be met by giving processes with deadlines a higher priority.
- Starvation of lower priority processes is possible with large amounts of high priority processes queuing for CPU time.

Round-robin scheduling

Main article: Round-robin scheduling

The scheduler assigns a fixed time unit per process, and cycles through them.

- RR scheduling involves extensive overhead, especially with a small time unit.
- Balanced throughput between FCFS and SJF, shorter jobs are

completed faster than in FCFS and longer processes are completed faster than in SJF.
- Good average response time, waiting time is dependent on number of processes, and not average process length.
- Because of high waiting times, deadlines are rarely met in a pure RR system.
- Starvation can never occur, since no priority is given. Order of time unit allocation is based upon process arrival time, similar to FCFS.

Multilevel queue scheduling

Main article: Multilevel feedback queue

This is used for situations in which processes are easily divided into different groups. For example, a common division is made between foreground (interactive) processes and background (batch) processes. These two types of processes have different response-time requirements and so may have different scheduling needs. It is very useful for shared memory problems.

Scheduling optimization problems

- Open-shop scheduling
- Job Shop Scheduling
- Flow Shop Scheduling Problem

Manual scheduling

Main article: Manual scheduling

A very common method in embedded systems is to manually schedule jobs. This can for example be done in a time-multiplexed fashion. Sometimes the kernel is divided in three or more parts: Manual scheduling, preemptive and interrupt level. Exact methods for scheduling jobs are often proprietary.

- No resource starvation problems.
- Very high predictability; allows implementation of hard real-time systems.
- Almost no overhead.
- May not be optimal for all applications.
- Effectiveness is completely dependent on the implementation.

How to choose a scheduling algorithm

When designing an operating system, a programmer must consider which scheduling algorithm will perform best for the use the system is going to see. There is no universal “best” scheduling algorithm, and many operating systems use extended or combinations of the scheduling algorithms above. For example, Windows NT/XP/Vista uses a multilevel feedback queue, a combination of fixed priority preemptive scheduling, round-robin, and first in first out. In this system, threads can dynamically increase or decrease in priority depending on if it has been serviced already, or if it has been waiting extensively. Every priority level is represented by its own queue, with round-robin scheduling amongst the high priority threads and FIFO among the lower ones. In this sense, response time is short for most threads, and short but critical system threads get completed very quickly. Since threads can only use one time unit of the round robin in the highest priority queue, starvation can be a problem for longer high priority threads.

Operating system process scheduler implementations

The algorithm used may be as simple as round-robin in which each process is given equal time (for instance 1 ms, usually between 1 ms and 100 ms) in a cycling list. So, process A executes for 1 ms, then process B, then process C, then back to process A.

More advanced algorithms take into account process priority, or the importance of the process. This allows some processes to use more time than other processes. The kernel always uses whatever resources it needs to ensure proper functioning of the system, and so can be said to have infinite priority. In SMP(symmetric multiprocessing) systems, processor affinity is considered to increase overall system performance, even if it may cause a process itself to run more slowly. This generally improves performance by reducing cache thrashing.

Windows

Very early MS-DOS and Microsoft Windows systems were non-multitasking, and as such did not feature a scheduler. Windows 3.1x used a non-preemptive scheduler, meaning that it did not interrupt programs. It relied on the program to end or tell the OS that it didn't need the processor so that it could move on to another

process. This is usually called cooperative multitasking. Windows 95 introduced a rudimentary preemptive scheduler; however, for legacy support opted to let 16 bit applications run without preemption.[1]

Windows NT-based operating systems use a multilevel feedback queue. 32 priority levels are defined, 0 through to 31, with priorities 0 through 15 being "normal" priorities and priorities 16 through 31 being soft real-time priorities, requiring privileges to assign. 0 is reserved for the Operating System. Users can select 5 of these priorities to assign to a running application from the Task Manager application, or through thread management APIs. The kernel may change the priority level of a thread depending on its I/O and CPU usage and whether it is interactive (i.e. accepts and responds to input from humans), raising the priority of interactive and I/O bounded processes and lowering that of CPU bound processes, to increase the responsiveness of interactive applications.[2] The scheduler was modified in Windows Vista to use the cycle counter register of modern processors to keep track of exactly how many CPU cycles a thread has executed, rather than just using an interval-timer interrupt routine.[3] Vista also uses a priority scheduler for the I/O queue so that disk defragmenters and other such programs don't interfere with foreground operations.[4]

Mac OS

Mac OS 9 uses cooperative scheduling for threads, where one process controls multiple cooperative threads, and also provides preemptive scheduling for MP tasks. The kernel schedules MP tasks using a preemptive scheduling algorithm. All Process Manager processes run within a special MP task, called the "blue task". Those processes are scheduled cooperatively, using a round-robin scheduling algorithm; a process yields control of the processor to another process by explicitly calling a blocking function such as WaitNextEvent. Each process has its own copy of the Thread Manager that schedules that process's threads cooperatively; a thread yields control of the processor to another thread by calling YieldToAnyThread or YieldToThread.[5]

Mac OS X uses a multilevel feedback queue, with four priority bands for threads - normal, system high priority, kernel mode only, and real-time.[6] Threads are scheduled preemptively; Mac OS X also supports cooperatively scheduled threads in its implementation of the Thread Manager in Carbon.[5]

AIX

In AIX Version 4 there are three possible values for thread scheduling policy :

- First In, First Out: Once a thread with this policy is scheduled, it runs to completion unless it is blocked, it voluntarily yields control of the CPU, or a higher-priority thread becomes dispatchable. Only fixed-priority threads can have a FIFO scheduling policy.
- Round Robin: This is similar to the AIX Version 3 scheduler round-robin scheme based on 10ms time slices. When a RR thread has control at the end of the time slice, it moves to the tail of the queue of dispatchable threads of its priority. Only fixed-priority threads can have a Round Robin scheduling policy.
- OTHER: This policy is defined by POSIX1003.4a as implementation-defined. In AIX Version 4, this policy is defined to be equivalent to RR, except that it applies to threads with non-fixed priority. The recalculation of the running thread's priority value at each clock interrupt means that a thread may lose control because its priority value has risen above that of another dispatchable thread. This is the AIX Version 3 behavior.

Threads are primarily of interest for applications that currently consist of several asynchronous processes. These applications might impose a lighter load on the system if converted to a multithreaded structure.

AIX 5 implements the following scheduling policies: FIFO, round robin, and a fair round robin. The FIFO policy has three different implementations: FIFO, FIFO2, and FIFO3. The round robin policy is named SCHED_RR in AIX, and the fair round robin is called SCHED_OTHER. This link provides additional information on AIX 5 scheduling: http://www.ibm.com/developerworks/aix/library/au-aix5_cpu/index.html#N100F6 .

Linux

Linux 2.4

In Linux 2.4, an O(n) scheduler with a multilevel feedback queue with priority levels ranging from 0 to 140 was used; 0–99 are reserved for real-time tasks and 100–140 are considered nice task levels. For real-time tasks, the time quantum for switching processes was approximately 200 ms, and for nice tasks approximately 10

ms.[*citation needed*] The scheduler ran through the run queue of all ready processes, letting the highest priority processes go first and run through their time slices, after which they will be placed in an expired queue. When the active queue is empty the expired queue will become the active queue and vice versa.

However, some Enterprise Linux distributions such as SUSE Linux Enterprise Server replaced this scheduler with a backport of the O(1) scheduler (which was maintained by Alan Cox in his Linux 2.4-ac Kernel series) to the Linux 2.4 kernel used by the distribution.

Linux 2.6.0 to Linux 2.6.22

From versions 2.6 to 2.6.22, the kernel used an O(1) scheduler developed by Ingo Molnar and many other kernel developers during the Linux 2.5 development. For many kernel in time frame, Con Kolivas developed patch sets which improved interactivity with this scheduler or even replaced it with his own schedulers.

Since Linux 2.6.23

Con Kolivas's work, most significantly his implementation of "fair scheduling" named "Rotating Staircase Deadline", inspired Ingo Molnár to develop the Completely Fair Scheduler as a replacement for the earlier O(1) scheduler, crediting Kolivas in his announcement.[7]

The Completely Fair Scheduler (CFS) uses a well-studied, classic scheduling algorithm called fair queuing originally invented for packet networks. Fair queuing had been previously applied to CPU scheduling under the name stride scheduling.

The fair queuing CFS scheduler has a scheduling complexity of O(log N), where N is the number of tasks in the runqueue. Choosing a task can be done in constant time, but reinserting a task after it has run requires O(log N) operations, because the run queue is implemented as a red-black tree.

CFS is the first implementation of a fair queuing process scheduler widely used in a general-purpose operating system.[8]

The Brain !@#$%^&* Scheduler (BFS) is an alternative to the CFS.

FreeBSD

FreeBSD uses a multilevel feedback queue with priorities ranging from

0-255. 0-63 are reserved for interrupts, 64-127 for the top half of the kernel, 128-159 for real-time user threads, 160-223 for time-shared user threads, and 224-255 for idle user threads. Also, like Linux, it uses the active queue setup, but it also has an idle queue.[9]

NetBSD

NetBSD uses a multilevel feedback queue with priorities ranging from 0-223. 0-63 are reserved for time-shared threads (default, SCHED_OTHER policy), 64-95 for user threads which entered kernel space, 96-128 for kernel threads, 128-191 for user real-time threads (SCHED_FIFO and SCHED_RR policies), and 192-223 for software interrupts.

Solaris

Solaris uses a multilevel feedback queue with priorities ranging from 0-169. 0-59 are reserved for time-shared threads, 60-99 for system threads, 100-159 for real-time threads, and 160-169 for low priority interrupts. Unlike Linux, when a process is done using its time quantum, it's given a new priority and put back in the queue.

Summary

Operating System	Preemption	Algorithm
Amiga OS	Yes	Prioritized Round-robin scheduling
FreeBSD	Yes	Multilevel feedback queue
Linux pre-2.6	Yes	Multilevel feedback queue
Linux 2.6-2.6.23	Yes	O(1) scheduler
Linux post-2.6.23	Yes	Completely Fair Scheduler
Mac OS pre-9	None	Cooperative Scheduler
Mac OS 9	Some	Preemptive for MP tasks, Cooperative Scheduler for processes and threads
Mac OS X	Yes	Multilevel feedback queue
NetBSD	Yes	Multilevel feedback queue
Solaris	Yes	Multilevel feedback queue
Windows 3.1x	None	Cooperative Scheduler
Windows 95, 98, Me	Half	Preemptive for 32-bit processes, Cooperative Scheduler for 16-bit processes
Windows NT (including 2000, XP, Vista, 7, and Server)	Yes	Multilevel feedback queue

See also

- Activity selection problem
- Aging (scheduling)
- Atropos scheduler
- Automated planning and scheduling
- Completely Fair Scheduler
- Computer multitasking
- Cyclic executive
- Dynamic priority scheduling
- Earliest deadline first scheduling
- Foreground-background
- Interruptible operating system
- Least slack time scheduling
- Lottery scheduling
- Multilevel feedback queue

- O(1) scheduler
- Priority inversion
- Process (computing)
- Process states
- Rate-monotonic scheduling
- Stride scheduling
- Time-utility function

References

1. ^ Early Windows (https://web.archive.org/web/*/www.jgcampbell.com/caos/html/node13.html) at the Wayback Machine
2. ^ Sriram Krishnan. "A Tale of Two Schedulers Windows NT and Windows CE" (http://web.archive.org/web/20120722015555/http://sriramk.com/schedulers.html).
3. ^ Inside the Windows Vista Kernel: Part 1 (http://technet.microsoft.com/en-us/magazine/cc162494.aspx), Microsoft Technet
4. ^ "Vista Kernel Improvements" (http://blog.gabefrost.com/?p=25).
5. ^ ***a b*** "Technical Note TN2028 - Threading Architectures" (http://developer.apple.com/technotes/tn/tn2028.html).
6. ^ "Mach Scheduling and Thread Interfaces" (http://developer.apple.com/mac/library/documentation/Darwin/Conceptual/KernelProgramming/scheduler/scheduler.html).
7. ^ Molnár, Ingo (2007-04-13). "[patch] Modular Scheduler Core and Completely Fair Scheduler [CFS]" (http://lwn.net/Articles/230501/). *linux-kernel mailing list*. http://lwn.net/Articles/230501/.
8. ^ Efficient and Scalable Multiprocessor Fair Scheduling Using Distributed Weighted Round-Robin (http://happyli.org/tongli/papers/dwrr.pdf)
9. ^ "Comparison of Solaris, Linux, and FreeBSD Kernels" (http://web.archive.org/web/20080807124435/http://cn.opensolaris.org/files/solaris_linux_bsd_cmp.pdf).

- Błażewicz, Jacek; Ecker, K.H.; Pesch, E.; Schmidt, G.; Weglarz, J. (2001). *Scheduling computer and manufacturing processes* (2 ed.). Berlin [u.a.]: Springer. ISBN 3-540-41931-4.
- Stallings, William (2004). *Operating Systems Internals and Design Principles (fifth international edition)*. Prentice Hall. ISBN 0-13-147954-7.
- Stallings, William (2004). *Operating Systems Internals and Design Principles (fourth edition)*. Prentice Hall. ISBN 0-13-031999-6.
- Information on the Linux 2.6 O(1)-scheduler (http://joshaas.net/linux/)

Further reading

- Operating Systems: Three Easy Pieces (http://pages.cs.wisc.edu/~remzi/OSTEP/) by Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. Arpaci-Dusseau Books, 2014. Relevant chapters: Scheduling: Introduction (http://pages.cs.wisc.edu/~remzi/OSTEP/cpu-sched.pdf) Multi-level Feedback Queue (http://pages.cs.wisc.edu/~remzi/OSTEP/cpu-sched-mlfq.pdf) Proportional-share Scheduling (http://pages.cs.wisc.edu/~remzi/OSTEP/cpu-sched-lottery.pdf) Multiprocessor Scheduling (http://pages.cs.wisc.edu/~remzi/OSTEP/cpu-sched-multi.pdf)
- Brief discussion of Job Scheduling algorithms (http://www.cs.sunysb.edu/~algorith/files/scheduling.shtml)
- Understanding the Linux Kernel: Chapter 10 Process Scheduling (http://www.oreilly.com/catalog/linuxkernel/chapter/ch10.html)
- Kerneltrap: Linux kernel scheduler articles (http://kerneltrap.org/scheduler)
- AIX CPU monitoring and tuning (http://www.ibm.com/developerworks/aix/library/au-aix5_cpu/index.html#N100F6)
- Josh Aas' introduction to the Linux 2.6.8.1 CPU scheduler implementation (http://joshaas.net/linux/)
- Peter Brucker, Sigrid Knust. Complexity results for scheduling problems [1] (http://www.mathematik.uni-osnabrueck.de/research/OR/class/)
- TORSCHE Scheduling Toolbox for Matlab (http://rtime.felk.cvut.cz/scheduling-toolbox) is a toolbox of scheduling and graph algorithms.
- [2] (http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=6226795) A survey on cellular networks packet scheduling

Retrieved from "http://en.wikipedia.org/w/index.php?title=Scheduling_(computing)&oldid=605490692"
Categories: Operations research | Scheduling (computing) | Scheduling algorithms | Planning

---

- This page was last modified on 23 April 2014 at 18:28.
- Text is available under the Creative Commons Attribution-ShareAlike License; additional terms may apply. By using this site, you agree to the Terms of Use and Privacy Policy. Wikipedia® is a registered trademark of the Wikimedia Foundation, Inc., a non-profit organization.