Scheduling (computing)

In computing, **scheduling** is the action of assigning *resources* to perform *tasks*. The *resources* may be processors, network links or expansion cards. The *tasks* may be threads, processes or data flows.

The scheduling activity is carried out by a process called **scheduler**. Schedulers are often designed so as to keep all computer resources busy (as in load balancing), allow multiple users to share system resources effectively, or to achieve a target quality-of-service.

Scheduling is fundamental to computation itself, and an intrinsic part of the execution model of a computer system; the concept of scheduling makes it possible to have computer multitasking with a single central processing unit (CPU).

Goals

A scheduler may aim at one or more goals, for example:

- maximizing *throughput* (the total amount of work completed per time unit);
- minimizing *wait time* (time from work becoming ready until the first point it begins execution);

- minimizing *latency* or *response time* (time from work becoming ready until it is finished in case of batch activity,[1][2][3] or until the system responds and hands the first output to the user in case of interactive activity);[4]
- maximizing *fairness* (equal CPU time to each process, or more generally appropriate times according to the priority and workload of each process).

In practice, these goals often conflict (e.g. throughput versus latency), thus a scheduler will implement a suitable compromise. Preference is measured by any one of the concerns mentioned above, 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.

Types of operating system schedulers

The scheduler is an operating system module that selects the next jobs to be admitted into the system and the next process to run. Operating systems may feature up to three distinct scheduler types: 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 their functions are performed.

Process scheduler

The process scheduler is a part of the operating system that decides which process runs at a certain point in time. It usually has the ability to pause a running process, move it to the back of the running queue and start a new process; such a scheduler is known as a *preemptive scheduler*, otherwise it is a *cooperative scheduler*.[5]

We distinguish between "long-term scheduling", "medium-term scheduling", and "short-term scheduling" based on how often decisions must be made.[6]

Long-term scheduling

The *long-term scheduler*, or *admission scheduler*, decides which jobs or processes are to be admitted to the ready queue (in 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 – whether many or few 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 general, most processes can be described as either I/O-bound or CPU-bound. An I/O-bound process is one that spends more of its time doing I/O than it spends doing computations. A CPU-bound process, in contrast, generates I/O requests infrequently, using more of its time doing computations. It is important that a long-term scheduler selects a good process mix of I/O-bound and CPU-bound processes. If all processes are I/O-bound, the ready queue will almost always be empty, and the short-term scheduler will have little to do. On the other hand, if all processes are CPU-bound, the I/O waiting queue will almost always be empty, devices will go unused, and again the system will be unbalanced. The system with the best performance will thus have a combination of CPU-bound and I/O-bound processes. In modern operating systems, this is used to make sure that real-time processes get enough CPU time to finish their tasks.[7]

Long-term scheduling is also important in large-scale systems such as batch processing systems, computer clusters, supercomputers, and render farms. For example, in concurrent systems, coscheduling of interacting processes is often required to prevent them from blocking due to waiting on each other. 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.

Some operating systems only allow new tasks to be added if it is sure all real-time deadlines can still be met. The specific heuristic algorithm used by an operating system to accept or reject new tasks is the *admission control mechanism*.[8]

Medium-term scheduling

The *medium-term scheduler* temporarily removes processes from main memory and places them in secondary memory (such as a hard disk drive) or vice versa, which 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] also called demand paging.

Short-term scheduling

The *short-term scheduler* (also known as the *CPU scheduler*) decides which of the ready, in-memory processes is 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, which is the module that gives control of the CPU to the process selected by the short-term scheduler. It receives control in kernel mode as the result of an interrupt or system call. The functions of a dispatcher involve the following:

- Context switches, in which the dispatcher saves the state (also known as context) of the process or thread that was previously running; the dispatcher then loads the initial or previously saved state of the new process.
- Switching to user mode.
- Jumping to the proper location in the user program to restart that program indicated by its new state.

The dispatcher should be as fast as possible, since it is invoked during every process switch. During the context switches, the processor is virtually idle for a fraction of time, thus unnecessary context switches should be avoided. The time it takes for the dispatcher to stop one process and start another is known as the *dispatch latency*.[7]:155

Scheduling disciplines

A **scheduling discipline** (also called **scheduling policy** or **scheduling algorithm**) is an algorithm 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.

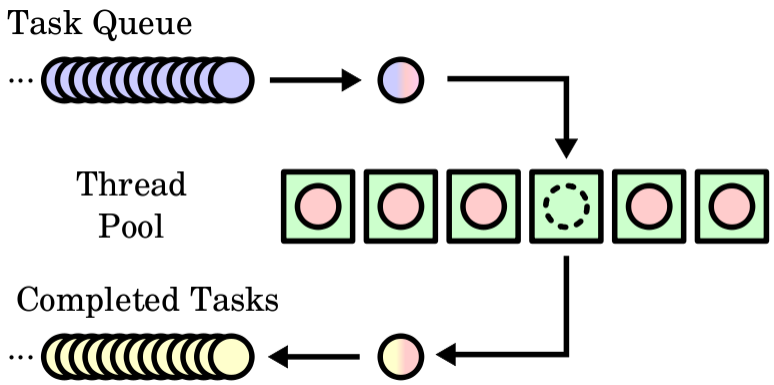
The main purposes of scheduling algorithms are to minimize resource starvation and to ensure fairness amongst the parties utilizing the resources. Scheduling deals with the problem of deciding which of the outstanding requests is to be allocated resources. There are many different scheduling algorithms. 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), proportional-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.[9]

First come, first served



A sample thread pool (green boxes) with a queue (FIFO) of waiting tasks (blue) and a queue of completed tasks (yellow)

First in, first out (FIFO), also known as *first come, first served* (FCFS), is the simplest scheduling algorithm. FIFO simply queues processes in the order that they arrive in the ready queue. This is commonly used for a **task queue**, for example as illustrated in this section.

- 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, because long processes can be holding the CPU, causing the short processes to wait for a long time (known as the convoy effect).
- No starvation, because each process gets chance to be executed after a definite time.
- Turnaround time, waiting time and response time depend on the order of their arrival and 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.

Priority scheduling

Earliest deadline first (EDF) or *least time to go* is a dynamic scheduling algorithm used in real-time operating systems to place processes in a priority queue. Whenever a scheduling event

occurs (a task finishes, new task is released, etc.), the queue will be searched for the process closest to its deadline, which will be the next to be scheduled for execution.

Shortest remaining time first

Similar to shortest job first (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 is 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.
- To use this policy we should have at least two processes of different priority

Fixed priority pre-emptive scheduling

The operating system 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 numbers of high-priority processes queuing for CPU time.

Round-robin scheduling

The scheduler assigns a fixed time unit per process, and cycles through them. If process completes within that time-slice it gets terminated otherwise it is rescheduled after giving a chance to all other processes.

- RR scheduling involves extensive overhead, especially with a small time unit.
- Balanced throughput between FCFS/ FIFO and SJF/SRTF, shorter jobs are completed faster than in FIFO 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 FIFO.
- If Time-Slice is large it becomes FCFS /FIFO or if it is short then it becomes SJF/SRTF.

Multilevel queue scheduling

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.

Work-conserving schedulers

A work-conserving scheduler is a scheduler that always tries to keep the scheduled resources busy, if there are submitted jobs ready to be scheduled. In contrast, a non-work conserving

scheduler is a scheduler that, in some cases, may leave the scheduled resources idle despite the presence of jobs ready to be scheduled.

Scheduling optimization problems

There are several scheduling problems in which the goal is to decide which job goes to which station at what time, such that the total makespan is minimized:

- Job shop scheduling – there are n jobs and m identical stations. Each job should be executed on a single machine. This is usually regarded as an online problem.
- Open-shop scheduling – there are n jobs and m different stations. Each job should spend some time at each station, in a free order.
- Flow shop scheduling – there are n jobs and m different stations. Each job should spend some time at each station, in a pre-determined order.

Manual scheduling

A very common method in embedded systems is to schedule jobs manually. 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

Choosing 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 algorithms. 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 among the high-priority threads and FIFO among the lower-priority 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 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.

OS/360 and successors

IBM OS/360 was available with three different schedulers. The differences were such that the variants were often considered three different operating systems:

- The *Single Sequential Scheduler* option, also known as the *Primary Control Program (PCP)* provided sequential execution of a single stream of jobs.
- The *Multiple Sequential Scheduler* option, known as *Multiprogramming with a Fixed Number of Tasks (MFT)* provided execution of multiple concurrent jobs. Execution was governed by a priority which had a default for each stream or could be requested separately for each job. MFT version II added subtasks (threads), which executed at a priority based on that of the parent job. Each job stream defined the maximum amount of memory which could be used by any job in that stream.
- The *Multiple Priority Schedulers* option, or *Multiprogramming with a Variable Number of Tasks (MVT)*, featured subtasks from the start; each job requested the priority and memory it

required before execution.

Later virtual storage versions of MVS added a *Workload Manager* feature to the scheduler, which schedules processor resources according to an elaborate scheme defined by the installation.

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.[10]

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. User interfaces and APIs work with priority classes for the process and the threads in the process, which are then combined by the system into the absolute priority level.

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.[11] 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.[12] Vista also uses a priority scheduler for the I/O queue so that disk defragmenters and other such programs do not interfere with foreground operations.[13]

Classic Mac OS and macOS

Mac OS 9 uses cooperative scheduling for threads, where one process controls multiple cooperative threads, and also provides preemptive scheduling for multiprocessing tasks. The kernel schedules multiprocessing tasks using a preemptive scheduling algorithm. All Process Manager processes run within a special multiprocessing 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`.[14]

macOS uses a multilevel feedback queue, with four priority bands for threads – normal, system high priority, kernel mode only, and real-time.[15] Threads are scheduled preemptively; macOS also supports cooperatively scheduled threads in its implementation of the Thread Manager in Carbon.[14]

AIX

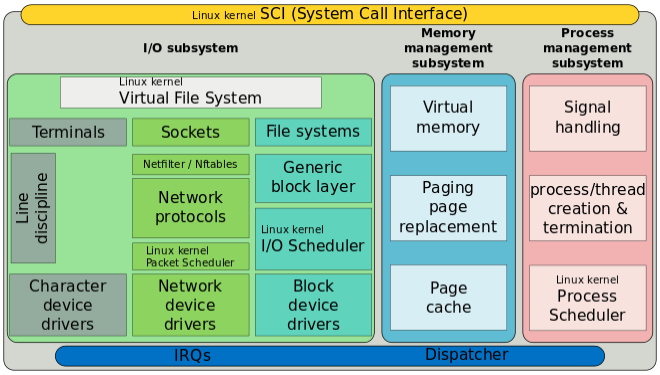
In AIX Version 4 there are three possible values for thread scheduling policy:

- 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 10 ms 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.[16]

Linux



A highly simplified structure of the Linux kernel, which includes process schedulers, I/O schedulers, and packet schedulers

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. 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

In versions 2.6.0 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' 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.[17] CFS is the

first implementation of a fair queuing process scheduler widely used in a general-purpose operating system.[18]

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.

The Brain Fuck Scheduler, also created by Con Kolivas, 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.[19]

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 between 0 and 169. Priorities 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,[19] when a process is done using its time quantum, it is given a new priority and put back in the queue. Solaris 9 introduced two new scheduling classes, namely fixed priority class and fair share class. The threads with fixed priority have the same priority range as that of the time-sharing class, but their priorities are not dynamically adjusted. The fair scheduling class uses CPU *shares* to prioritize threads for

scheduling decisions. CPU shares indicate the entitlement to CPU resources. They are allocated to a set of processes, which are collectively known as a project.[7]

Summary

Operating System	Preemption	Algorithm
Amiga OS	Yes	Prioritized round-robin scheduling
FreeBSD	Yes	Multilevel feedback queue
Linux kernel before 2.6.0	Yes	Multilevel feedback queue
Linux kernel 2.6.0–2.6.23	Yes	O(1) scheduler
Linux kernel after 2.6.23	Yes	Completely Fair Scheduler
classic Mac OS pre-9	None	Cooperative scheduler
Mac OS 9	Some	Preemptive scheduler for MP tasks, and cooperative for processes and threads
macOS	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 scheduler for 32-bit processes, and cooperative 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
- Cyclic executive
- Dynamic priority scheduling
- Foreground-background
- Interruptible operating system
- Least slack time scheduling
- Lottery scheduling

- Priority inversion
- Process states
- Queuing Theory
- Rate-monotonic scheduling
- Resource-Task Network
- Scheduling (production processes)
- Stochastic scheduling
- Time-utility function

Notes

1. *C. L., Liu; James W., Layland (January 1973). "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment". Journal of the ACM. ACM. **20** (1): 46–61. doi:10.1145/321738.321743 (https://doi.org/10.1145%2F321738.321743) . S2CID 207669821 (https://api.semanticscholar.org/CorpusID:207669821) . "We define the response time of a request for a certain task to be the time span between the request and the end of the response to that request."*
2. *Kleinrock, Leonard (1976). Queueing Systems, Vol. 2: Computer Applications (https://archive.org/details/queueingsystems00klei/page/171) (1 ed.). Wiley-Interscience. p. 171 (https://archive.org/details/queueingsystems00klei/page/171) . ISBN 047149111X. "For a customer requiring x sec of service, his response time will equal his service time x plus his waiting time."*
3. *Feitelson, Dror G. (2015). Workload Modeling for Computer Systems Performance Evaluation (http://www.cs.huji.ac.il/~feit/wlmod/) . Cambridge University Press. Section 8.4 (Page 422) in Version 1.03 of the freely available manuscript. ISBN 9781107078239. Retrieved 2015-10-17. "if we denote the time that a job waits in the queue by t_w, and the time it actually runs by t_r, then the response time is $r = t_w + t_r$."*
4. *Silberschatz, Abraham; Galvin, Peter Baer; Gagne, Greg (2012). Operating System Concepts (9 ed.). Wiley Publishing. p. 187. ISBN 978-0470128725. "In an interactive system, turnaround time may not be the best criterion. Often, a process can produce some output fairly early and can continue computing new results while previous results are being output to the user. Thus, another measure is the time from the submission of a request until the first response is produced. This measure, called response time, is the time it takes to start responding, not the time it takes to output the response."*
5. *Paul Krzyzanowski (2014-02-19). "Process Scheduling: Who gets to run next?" (https://www.cs.rutgers.edu/~pxk/416/notes/07-scheduling.html) . cs.rutgers.edu. Retrieved 2015-01-11.*
6. *Raphael Finkel. "An Operating Systems Vade Mecum" (https://www.yumpu.com/en/document/read/32199214/an-operating-systems-vade-mecum) . Prentice Hall. 1988. "chapter 2: Time Management". p. 27.*
7. *Abraham Silberschatz, Peter Baer Galvin and Greg Gagne (2013). Operating System Concepts. Vol. 9. John Wiley & Sons, Inc. ISBN 978-1-118-06333-0.*
8. *Robert Kroeger (2004). "Admission Control for Independently-authored Realtime Applications" (https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.71.3977&rep=rep1&type=pdf) . UWSpace. http://hdl.handle.net/10012/1170 . Section "2.6 Admission Control". p. 33.*

9. Guowang Miao; Jens Zander; Ki Won Sung; Ben Slimane (2016). Fundamentals of Mobile Data Networks. Cambridge University Press. ISBN 978-1107143210.
10. Early Windows (https://web.archive.org/web/*/www.jgcampbell.com/caos/html/node13.html) at the Wayback Machine (archive index)
11. Sriram Krishnan. "A Tale of Two Schedulers Windows NT and Windows CE" (https://web.archive.org/web/20120722015555/http://sriramk.com/schedulers.html) . Archived from the original (http://sriramk.com/schedulers.html) on July 22, 2012.
12. "Windows Administration: Inside the Windows Vista Kernel: Part 1" (https://technet.microsoft.com/en-us/magazine/cc162494.aspx) . Technet.microsoft.com. 2016-11-14. Retrieved 2016-12-09.
13. "Archived copy" (https://web.archive.org/web/20080219174631/http://blog.gabefrost.com/?p=25) . blog.gabefrost.com. Archived from the original (http://blog.gabefrost.com/?p=25) on 19 February 2008. Retrieved 15 January 2022.
14. "Technical Note TN2028: Threading Architectures" (https://developer.apple.com/library/archive/technotes/tn/tn2028.html) . developer.apple.com. Retrieved 2019-01-15.
15. "Mach Scheduling and Thread Interfaces" (https://developer.apple.com/library/archive/documentation/Darwin/Conceptual/KernelProgramming/scheduler/scheduler.html) . developer.apple.com. Retrieved 2019-01-15.
16. [1] (http://www.ibm.com/developerworks/aix/library/au-aix5_cpu/index.html#N100F6) Archived (https://web.archive.org/web/20110811094049/http://www.ibm.com/developerworks/aix/library/au-aix5_cpu/index.html) 2011-08-11 at the Wayback Machine
17. Molnár, Ingo (2007-04-13). "[patch] Modular Scheduler Core and Completely Fair Scheduler [CFS]" (https://lwn.net/Articles/230501/) . linux-kernel (Mailing list).
18. Tong Li; Dan Baumberger; Scott Hahn. "Efficient and Scalable Multiprocessor Fair Scheduling Using Distributed Weighted Round-Robin" (http://happyli.org/tongli/papers/dwrr.pdf) (PDF). Happyli.org. Retrieved 2016-12-09.
19. "Comparison of Solaris, Linux, and FreeBSD Kernels" (https://web.archive.org/web/20080807124435/http://cn.opensolaris.org/files/solaris_linux_bsd_cmp.pdf) (PDF). Archived from the original (http://cn.opensolaris.org/files/solaris_linux_bsd_cmp.pdf) (PDF) on August 7, 2008.

References

- 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* (https://archive.org/details/operatingsystems00stal) (fourth ed.). Prentice Hall. ISBN 0-13-031999-6.

- Information on the Linux 2.6 O(1)-scheduler (https://github.com/bdaehlie/linux-cpu-scheduler-docs/)

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 (https://web.archive.org/web/20060613130106/http://oreilly.com/catalog/linuxkernel/chapter/ch10.html)
- Kerneltrap: Linux kernel scheduler articles (http://kerneltrap.org/scheduler)
- AIX CPU monitoring and tuning (https://web.archive.org/web/20110811094049/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 (https://github.com/bdaehlie/linux-cpu-scheduler-docs/)
- Peter Brucker, Sigrid Knust. Complexity results for scheduling problems [2] (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.
- A survey on cellular networks packet scheduling (http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=6226795)
- Large-scale cluster management at Google with Borg (https://static.googleusercontent.com/media/research.google.com/en/us/pubs/archive/43438.pdf)

Last edited 27 days ago **by** DanCherek