OPERATING SYSTEMS

SCHEDULING

6 Scheduling

2

When a computer is multi-programmed, it frequently has multiple processes or threads (in the ready state) competing for the CPU. If only one CPU is available, a choice has to be made which process to run next. The part of the operating system that makes the choice is called the scheduler, and the algorithm it uses is called the scheduling algorithm.

6.1 Introduction to Scheduling

3

In personal computers, there are two situations for scheduling. First, most of the time there is only one active process. For example, an user entering a document on a word processor is unlikely to be simultaneously compiling a program in the background. When the user types a command to the word processor, the scheduler does not have to do much work to figure out which process to run the word processor is the only candidate.

6.1 Introduction to Scheduling

4

Second, since the CPU is not a scarce resource, even compilations (a major sink of CPU cycles in the past) take just a few seconds in most cases nowadays. Even when two programs are actually running at once, such as a word processor and a spreadsheet, it hardly matters which goes first since the user is probably waiting for both of them to finish. As a consequence, scheduling does not matter much on simple PCs.

6.1 Introduction to Scheduling

5

When we turn to networked servers, scheduling matters again since multiple processes often do compete for the CPU. For example, when the CPU has to choose between running a process that gathers the daily statistics and one that serves user requests, the users will be a lot happier if the latter gets first at the CPU.

6.1 Introduction to Scheduling

6

In addition, the scheduler has to worry about making efficient use of the CPU because process switching is expensive. To start with, a switch from user mode to kernel mode must occur. Then the state of the current process must be saved for reloading later. Next another process should be selected by running the scheduling algorithm. Finally, the new process must be started. Overall, doing too many process switches per second can cause spending a substantial amount of CPU time.

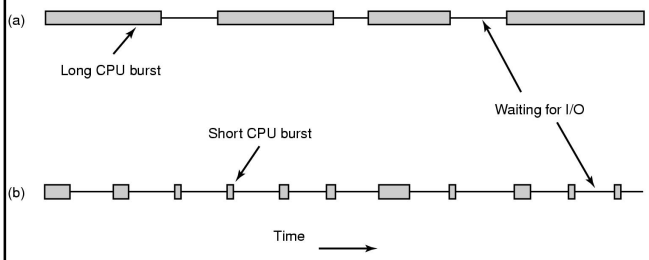
6.1.1 Process Behavior

7

A process mostly alternates computing and I/O requests. Typically the CPU runs for a while without stopping, then a system call is made to access a file. When the system call completes, the CPU computes again until it needs data. Note that some I/O activities count as computing. For example, when the CPU copies bits to a video RAM to update the screen, it is computing, not doing I/O, because the CPU is in use. I/O in this sense is when a process enters the blocked state waiting for an external device to complete its work.

6.1.1 Process Behavior

8



Bursts of CPU usage alternate with periods of I/O wait
(a) a CPU-bound process (b) an I/O bound process

6.1.1 Process Behavior

9

The process in (a) spend most of its time computing (called compute-bound), while other in (b) spend most of its time waiting for I/O (called I/O-bound). Note that the key factor is the length of the CPU burst, not the length of the I/O burst. I/O bound processes are I/O bound because they do not compute much between I/O requests, not because they have especially long I/O requests.

6.1.1 Process Behavior

10

As CPUs get faster, processes tend to get more I/O bound. This effect occurs because CPUs are improving much faster than disks. As a consequence, the scheduling of I/O-bound processes is likely to become a more important subject in the future. The basic idea here is that if an I/O-bound process wants to run, it should get a chance quickly so that it can issue its disk request and keep the disk busy.

6.1.2 When to Schedule

11

A key issue is when to make scheduling decisions:

First, when a new process is created, a decision needs to be made whether to run the parent process or the child process. Since both processes are in ready state, the scheduler can choose to run one of them next.

Second, a scheduling decision must-be made when a process exits. So some other processes must be chosen from the set of ready processes.

6.1.2 When to Schedule

12

Third, when a process blocks on I/O, another process has to be selected to run. Sometimes the reason for blocking may play a role in the choice. For example, if A is an important process and it is waiting for B to exit its critical region, letting B run next will allow it to exit its critical region and thus let A continue. The trouble, however, is that the scheduler generally does not have the necessary information to take this dependency into account.

6.1.2 When to Schedule

13

Fourth, when an I/O interrupt occurs, a scheduling decision may be made. If the interrupt came from an I/O device that has now completed its work, some process that was blocked waiting for the I/O may now be ready to run. It is up to the scheduler to decide whether to run the newly ready process, the process that was running at the time of the interrupt, or some third process.

6.1.2 When to Schedule

14

If a hardware clock provides periodic interrupts at a frequency (like 60 Hz), a scheduling decision can be made at each clock interrupt or at every k-th clock interrupt. Scheduling algorithms can be divided into two categories with respect to how they deal with clock interrupts:

- preemptive scheduling algorithms
- non-preemptive scheduling algorithms

6.1.2 When to Schedule

15

A non-preemptive scheduling algorithm picks a process to run and then just lets it run until it blocks (either on I/O or waiting for another process) or until it voluntarily releases the CPU. Even if it runs for hours, it will not be forcibly suspended. In fact, no scheduling decisions are made during clock interrupts. After clock interrupt processing has been completed, the process that was running before the interrupt is resumed, unless a higher-priority process was waiting for a now satisfied timeout.

6.1.2 When to Schedule

16

In contrast, a preemptive scheduling algorithm picks a process and lets it run for a maximum of some fixed time. If it is still running at the end of the time interval, it is suspended and the scheduler picks another process to run (if one is available). Doing preemptive scheduling requires having a clock interrupt occur at the end of the time interval to give control of the CPU back to the scheduler. If no clock is available non-preemptive scheduling is the only option.

6.1.3 Categories of Scheduling Algorithms

17

Different kinds of operating systems have different goals. In other words, what the scheduler should optimize for is not the same in all systems. Three environments worth distinguishing are

1. Batch
2. Interactive
3. Real time

6.1.3 Categories of Scheduling Algorithms

18

Batch systems are still in widespread use in the business world for doing some periodic tasks. Because there are no users waiting in batch systems, preemptive or non-preemptive algorithms with long time periods for each process, are often acceptable. This approach reduces process switches and thus improves performance.

6.1.3 Categories of Scheduling Algorithms

19

In an environment with interactive users, preemption is essential to keep one process from hogging the CPU and denying service to the others. Even if no process intentionally ran forever, one process might shut out all the others indefinitely due to a program bug. Preemption is needed to prevent this behavior. Servers also fall into this category, since they normally serve multiple users, all of whom are in a big hurry.

6.1.3 Categories of Scheduling Algorithms

20

In systems with real-time constraints, preemption is, oddly enough, sometimes not needed because the processes know that they may not run for long periods of time and usually do their work and block quickly. The difference with interactive systems is that real-time systems run only programs that are designed to advance the application at hand. Interactive systems are general purpose and may run arbitrary programs that are not cooperative or even malicious (harmful).

6.1.4 Scheduling Algorithm Goals

21

- All systems
  - Fairness - giving each process a fair share of the CPU
  - Policy enforcement - seeing that stated policy is carried out
  - Balance - keeping all parts of the system busy
- Batch systems
  - Throughput-maximize jobs per hour
  - Turnaround time-minimize time between submission and termination
  - CPU utilization - keep the CPU busy all the time
- Interactive systems
  - Response time - respond to requests quickly
- Real-time systems
  - Meeting deadlines - avoid losing data

6.1.4 Scheduling Algorithm Goals

22

The fairness is an important concept for comparable processes. Giving one process much more CPU time than an equivalent one is not fair. Of course, different categories of processes may be treated differently. Think of safety control and doing the payroll at a nuclear reactor's computer center. Sometimes fairness is related to the system's policies. If the local policy is that safety control processes get to run whenever they want to, even if it means the payroll is 30 sec late, the scheduler has to make sure this policy is enforced.

6.1.4 Scheduling Algorithm Goals

23

Another general goal is keeping whole system busy. If the CPU and I/O devices can be kept running at the same time, more work gets done per second. Running CPU-bound and I/O-bound processes in memory together is a better idea than running all the CPU-bound jobs first and then all the I/O-bound jobs. If the latter strategy is used, the disks are idle while all processes are fighting for the CPU, or the CPU is idle in otherwise. Better to keep the whole system running at once by a careful mix of processes.

6.1.4 Scheduling Algorithm Goals

24

The managers of large computer centers that run many batch jobs typically look at three metrics to see how well their systems are performing: throughput, turnaround time, and CPU utilization. Throughput is the number of jobs per hour that the system completes. Turnaround time is the statistically average time from the moment that a batch job is submitted until the moment it is completed. It measures how long the average user has to wait for the output.

6.1.4 Scheduling Algorithm Goals

25

A scheduling, algorithm that maximizes throughput may not necessarily minimize turnaround time. For example, given a mix of short jobs and long jobs, a scheduler that always ran short jobs and never ran long jobs might achieve an excellent throughput (many short jobs per hour) but at the expense of a terrible turnaround time for the long jobs. If short jobs kept arriving at a fairly steady rate, the long jobs might never run, making the mean turnaround time infinite while achieving a high throughput.

6.1.4 Scheduling Algorithm Goals

26

CPU utilization is often used as a metric on batch systems, actually though it is not such a good metric. What really matters is how many jobs per hour come out of the system (throughput) and how long it takes to get a job back (turnaround time). On the other hand, knowing when the CPU utilization is approaching 100% is useful for knowing when it is time to get more computing power.

6.1.4 Scheduling Algorithm Goals

27

For interactive systems, different goals apply. The most important one is to minimize response time, that is, the time between issuing a command and getting the result. On a personal computer where a background process is running, a user request to start a program or open a file should take precedence over the background work. Performing all interactive requests first will be perceived as good service.

6.1.4 Scheduling Algorithm Goals

28

Real-time systems are characterized by having deadlines that must be met. For example, if a computer is controlling a device that produces data at a regular rate, failure to run the data-collection process on time may result in lost data. Thus the foremost need in a real-time system is meeting most deadlines.

6.2 Scheduling in Batch Systems

29

Now we will look at some algorithms used in batch systems:

1. First-Come First-Served
2. Shortest Job First
3. Shortest Remaining Time Next

6.2.1 First-Come First-Served

30

In non-preemptive first-come first-served, there is a single queue of ready processes. When the first job enters the system from the outside at the start, it is started and allowed to run as long as it wants to. As other jobs come in, they are put onto the end of the queue. When the running process blocks, the first process on the queue is run next. When a blocked process becomes ready, like a newly arrived job, it is put on the end of the queue.

6.2.1 First-Come First-Served

31

The great strength of this algorithm is that it is easy to understand and equally easy to program. It is also fair in the same sense. With this algorithm, a single linked list keeps track of all ready processes. Picking a process to run just requires removing one from the front of the queue. Adding a new job or unblocked process just requires attaching it to the end of the queue.

6.2.1 First-Come First-Served

32

First-come first-served also has a disadvantage. Suppose that there is one compute-bound process that runs for 1 sec at a time and many I/O-bound processes that use little CPU time but each have to perform 1000 disk reads. The compute-bound process runs for 1 sec, then it reads a disk block. All the I/O processes now run and start disk reads. When the compute-bound process gets its disk block, it runs for another 1 sec, followed by all the I/O-bound processes in quick succession.

6.2.1 First-Come First-Served

33

The net result is that each I/O-bound process gets to read 1 block per second and will take 1000 sec to finish. With a scheduling algorithm that preempted the compute-bound process every 10 msec, the I/O-bound processes would finish in 10 sec instead of 1000 sec, and without slowing down the compute-bound process very much.

6.2.2 Shortest Job First

34

For example, people in an insurance company can predict how long it will take to run a batch of 1000 requests, since similar work is done every day. When several equally important jobs are sitting in the input queue, the scheduler picks the shortest job first. In figure, we find four jobs A, B, C, and D with run times of 8, 4, 4, and 4 minutes, respectively. By running them in that order, the turnaround time for A is 8 minutes, for B is 12 minutes, for C is 16 minutes, and for D is 20 minutes for an average of 14 minutes.

6.2.2 Shortest Job First

35

An example of shortest job first scheduling

6.2.2 Shortest Job First

36

Consider running these four jobs using shortest job first as in Fig.(b). The turnaround times are now 4, 8, 12, and 20 minutes for an average of 11 minutes. The case of four jobs are with run times of a, b, c, and d, respectively. The first job finishes at time a, the second finishes at time a + b, and so on. The mean turnaround time is (4a + 3b + 2c + d)/4. It is clear that a contributes more to the average than the other times, so it should be the shortest job, with b next, then c, and finally d as the longest as it affects only its own turnaround time.

6.2.3 Shortest Remaining Time Next

37

A preemptive version of shortest job first is shortest remaining time next. With this algorithm, the scheduler always chooses the process whose remaining run time is the shortest. Again here, the run time has to be known in advance. When a new job arrives, its total time is compared to the current process' remaining time. If the new job needs less time to finish than the current process, the current process is suspended and the new job started. This scheme allows new short jobs to get good service.

6.3 Scheduling in Interactive Systems

38

The algorithms that can be used in interactive systems;

1. Round-Robin Scheduling
2. Priority Scheduling
3. Shortest Process Next
4. Guaranteed Scheduling
5. Lottery Scheduling
6. Fair-Share Scheduling

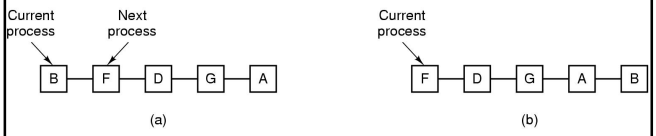
6.3.1 Round-Robin Scheduling

39

One of the oldest, simplest, fairest, and most widely used algorithms is Round Robin. Each process is assigned a time interval, called its quantum, during which it is allowed to run. When the process uses up its quantum, it is put on the end of the list and the CPU is given to another process. If the process has blocked or finished before the quantum has elapsed, the CPU switching is done. All the scheduler needs to do is maintain a list of runnable processes.

6.3.1 Round-Robin Scheduling

40



(a) list of runnable processes (b) list of runnable processes after B uses up its quantum

6.3.1 Round-Robin Scheduling

41

The only interesting issue with round robin is the length of the quantum. Switching from one process to another requires a certain amount of time for saving, loading, and so on. Suppose that this process switch takes 1 msec. Also suppose that the quantum is set at 4 msec. With these parameters, after doing 4 msec of useful work, the CPU will have to waste 1 msec on process switching. Thus 20% of the CPU time will be thrown away on administrative overhead.

6.3.1 Round-Robin Scheduling

42

To improve the CPU efficiency, we could set the quantum to, say, 100 msec. Now the wasted time is only 1 %. But consider what happens on a server system if 50 requests come in within a very short time interval and with widely varying CPU requirements. Fifty processes will be put on the list of runnable processes. The first one will start, the second one may not start in 100 msec. The unlucky last one may have to wait 5 sec before getting a chance. With a short quantum they would have gotten better service.

6.3.1 Round-Robin Scheduling

43

The conclusion can be formulated as follows:

setting the quantum too short causes too many process switches and lowers the CPU efficiency, but setting it too long may cause poor response to short interactive requests.

A quantum around 20-50 msec is often a reasonable compromise.

6.3.2 Priority Scheduling

44

Round-robin scheduling makes the implicit assumption that all processes are equally important. Frequently, the people who own and operate multiuser computers have different ideas on that subject. The need to take external factors into account leads to priority scheduling.

The basic idea is simple: each process is assigned a priority, and the runnable process with the highest priority is allowed to run.

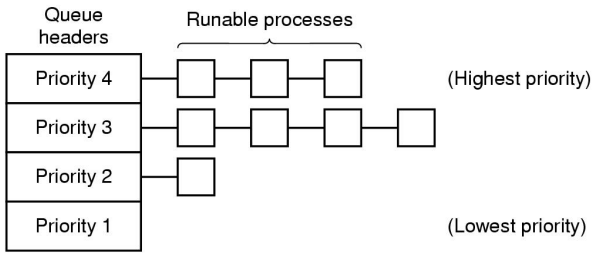
6.3.2 Priority Scheduling

45

To prevent high-priority processes from running indefinitely, the scheduler may decrease the priority of the currently running process at each clock tick. If this action causes its priority to drop below that of the next highest process, a process switch occurs. Alternatively, each process may be assigned a maximum time quantum that it is allowed to run. When this quantum is used up, the next highest priority process is given a chance to run.

6.3.2 Priority Scheduling

46



A scheduling algorithm with four priority classes

6.3.2 Priority Scheduling

47

It is often convenient to group processes into priority classes and use priority scheduling among the classes but round-robin scheduling within each class. In a system with four priority classes (as in figure), the scheduling algorithm is as follows: as long as there are runnable processes in priority class 4, just run each one for one quantum, round-robin fashion, and never bother with lower-priority classes. If priority class 4 is empty, then run the class 3 processes round robin. If priorities are not adjusted occasionally, lower priority classes may all starve to death.

6.3.3 Shortest Process Next

48

Because shortest job first always produces the minimum average response time for batch systems, it would be nice if it could be used for interactive processes as well. Interactive processes generally follow the pattern of wait for command, execute command, wait for command, execute command, and so on. If we regard the execution of each command as a separate "job," then we could minimize overall response time by running the shortest one first. The only problem is figuring out which of the currently runnable processes is the shortest one.

6.3.3 Shortest Process Next

49

One approach is to make estimates based on past behavior and run the process with the shortest estimated running time. Suppose that the estimated time per command for some terminal is T_0. Now suppose its next run is measured to be T_1. We could update our estimate by taking a weighted sum of these two numbers $T_2=aT_0 + (1-a)\ T_1$. Through the choice of "a" we can decide to have the estimation process forget old runs quickly, or remember them for a long time. After three new runs, the weight of T_0 in the new estimate has dropped to a^3.

6.3.3 Shortest Process Next

50

The technique of estimating the next value in a series by taking the weighted average of the current measured value and the previous estimate is sometimes called aging.

6.3.4 Guaranteed Scheduling

51

Another approach to scheduling is to make real promises to the users about performance. One promise is this: If there are n users logged in while you are working, you will receive about 1/n of the CPU power. Similarly, on a single-user system with n processes running, all things being equal, each one should get 1/n of the CPU cycles. That seems fair enough.

6.3.5 Lottery Scheduling

52

In lottery scheduling, the idea is to give processes lottery tickets for system resources such as CPU time. Whenever a scheduling decision has to be made, a lottery ticket is chosen at random, and the process holding that ticket gets the resource. When applied to CPU scheduling, the system might hold a lottery 50 times a second, with each winner getting 20 msec of CPU time as a prize.

6.3.6 Fair-Share Scheduling

53

Assume that if User1 starts up 9 processes and User2 starts up 1 process, with round robin or equal priorities, User1 will get 90% of the CPU and User2 will get only 10% of it. To prevent this situation, some systems take into account who owns a process before scheduling it.

6.3.6 Fair-Share Scheduling

54

Consider a system with two users, each of which has been promised 50% of the CPU. User1 has four processes, A, B, C, and D, and User2 has only 1 process, E. If round-robin scheduling is used, a possible scheduling sequence is

AEBECEDEAEBECEDE...

If User1 is entitled to twice as much CPU time as User2, we might get

ABECDEABECDE…

6.4 Scheduling in Real-Time Systems

55

A real-time system is one in which time plays an essential role. For example, the computer in a compact disc player gets the bits as they come off the drive and must convert them into music within a very tight time interval. If the calculation takes too long, the music will sound strange. Having the right answer but too late is often just as bad as not having it at all.

6.4 Scheduling in Real-Time Systems

56

Real-time systems categorized as hard or soft. In hard real-time, absolute deadlines must be met, in soft real time, missing an occasional deadline is undesirable but nevertheless tolerable.

For a real-time system, the events can be categorized as periodic (occurring at regular intervals) or aperiodic (occurring unpredictably).

6.4 Scheduling in Real-Time Systems

57

Real-time scheduling algorithms can be static or dynamic. The former make their scheduling decisions before the system starts running. The latter make their scheduling decisions at run time. Static scheduling only works when there is perfect information available in advance about the work to be done and the deadlines that have to be met. Dynamic scheduling algorithms do not have these restrictions.

6.5 Policy versus Mechanism

58

Sometimes one process has many children running under its control. For example, a database system process may have many children working on a request or performing a specific function. The main process has a task to find which child is the most important and which the least. Unfortunately, none of the schedulers discussed above accept any input from user processes about scheduling decisions. As a result, the scheduler rarely makes the best choice.

6.5 Policy versus Mechanism

59

The scheduling algorithm is parameterized in some way, but the parameters can be filled in by user processes. For example, suppose that the kernel uses a priority-scheduling algorithm but provides a system call by which a process can set and change the priorities of its children. In this way the parent can control in detail how its children are scheduled, even though it itself does not do the scheduling. Here the mechanism is in the kernel but policy is set by a user process.

6.6 Thread Scheduling

60

In multi-thread systems, we have two levels of parallelism present: processes and threads. Scheduling in such systems differs substantially depending on whether user-level threads or kernel-level threads are supported.

6.6 Thread Scheduling

61

Let us consider user-level threads first. Since the kernel is not aware of the existence of threads, it operates as it always does, picking a process, say, A, and giving A control for its quantum. The thread scheduler inside A decides which thread to run, say A1. Since there are no clock interrupts to multi-program threads, this thread may continue running as long as it wants to. If it uses up the process entire quantum, the kernel will select another process to run.

6.6 Thread Scheduling

62

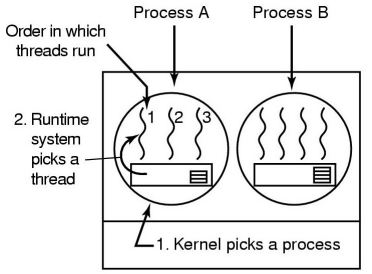
Now consider the case that A's threads have relatively little work to do per CPU burst, for example, 5msec of work within a 50msec quantum. Consequently each one runs for a little while, then yields the CPU back to the thread scheduler. This might lead to the sequence

A1, A2, A3, A1, A2, A3, A1, A2, A3, A1

before the kernel switches to process B.

6.6 Thread Scheduling

63



Possible: A1, A2, A3, A1, A2, A3
Not possible: A1, B1, A2, B2, A3, B3

Possible scheduling of user-level threads
(50-msec process quantum, threads run 5 msec/CPU burst)

6.6 Thread Scheduling

64

The scheduling algorithm used by the run-time system can be any of the ones described above. In practice, round-robin scheduling and priority scheduling are most common. The only constraint is the absence of a clock to interrupt a thread that has run too long.

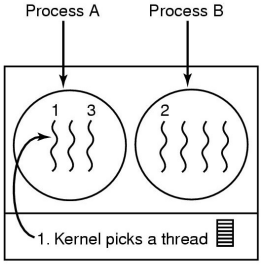
6.6 Thread Scheduling

65

Now consider the situation with kernel-level threads. Here the kernel picks a particular thread to run. It does not have to take into account which process the thread belongs to, but it can if it wants to. The thread is given a quantum and is forcibly suspended if it exceeds the quantum.

6.6 Thread Scheduling

66



Possible: A1, A2, A3, A1, A2, A3
Also possible: A1, B1, A2, B2, A3, B3

Possible scheduling of kernel-level threads
(50-msec process quantum, threads run 5 msec/CPU burst)

6.6 Thread Scheduling

67

A major difference between user-level threads and kernel-level threads is the performance. Doing a thread switch with user-level threads is very simple in contrast to kernel-level.

On the other hand, with kernel-level threads, having a thread block on I/O does not suspend the entire process as it does with user-level threads.

6.6 Thread Scheduling

68

Since the kernel knows that switching from a thread in process A to a thread in process B is more expensive than running a second thread in process A (due to having to change the memory map and having the memory cache spoiled), it can take this information into account when making a decision.

6.6 Thread Scheduling

69

User-level threads can employ an application-specific thread scheduler.

Consider a Web server that a worker thread has just blocked and the dispatcher thread and two worker threads are ready. The run-time system can easily pick the dispatcher to run next, so that it can start another worker running. This strategy maximizes the amount of parallelism in an environment where workers frequently block on disk I/O.