Scheduling Algorithms

A scheduling algorithm is the algorithm which dictates how much CPU time is allocated to Processes and Threads. The goal of any scheduling algorithm is to fulfill a number of criteria:

- no task must be starved of resources - all tasks must get their chance at CPU time;
- if using priorities, a low-priority task must not hold up a high-priority task;
- the scheduler must scale well with a growing number of tasks, ideally being O(1). This has been done, for example, in the Linux kernel.

Contents

Interactive Scheduling Algorithms

Round Robin

Round Robin is the simplest algorithm for a preemptive scheduler. Only a single queue of processes is used. When the system timer fires, the next process in the queue is switched to, and the preempted process is put back into the queue.

Each process is assigned a time slice or "quantum". This quantum dictates the number of system timer ticks the process may run for before being preempted. For example, if the timer runs at 100Hz, and a process' quantum is 10 ticks, it may run for 100 milliseconds (10/100 of a second). To achieve this, the running process is given a variable that starts at its quantum, and is then decremented each tick until it reaches zero. The process may also relinquish its quantum by doing a blocking system call (i.e. I/O), like in other preemptive algorithms.

In the Round Robin algorithm, each process is given an equal quantum; the big question is how to choose the time quantum.

Here are some considerations: The smaller the quantum, the larger the proportion of the time used in context switches.
Interactive processes should do I/O before being preempted, so that unnecessary preemptions are avoided.
The larger the quantum, the "laggier" the user experience - quanta above 100ms should be avoided.

A frequently chosen compromise for the quantum is between 20ms and 50ms.

Advantages of Round Robin include its simplicity and strict "first come, first served" nature. Disadvantages include the absence of a priority system: lots of low privilege processes may starve one high privilege one.

Priority-Based Round Robin

Priority scheduling is similar to Round Robin, but allows a hierarchy of processes. Multiple process queues are used, one for each priority. As long as there are processes in a higher priority queue, they are run first. For example, if you have 2 queues, "high" and "low", in this state:

"high": X
"low": xterm, vim, firefox

The first process to run would be X, then if it blocked (for I/O, probably), the state would be:

"high":
"low": xterm, vim, firefox

The next process that would run would be xterm. If process "kswapd" is added to "high", it would then get the next quantum:

"high": kswapd
"low": vim, firefox, xterm

There are usually between four and sixteen queues used in a priority scheduler.

Advantages of this algorithm are simplicity and reasonable support for priorities. The disadvantage (or possible advantage) is that privileged processes may completely starve unprivileged ones. This is less of a problem than it appears, because processes (especially daemons, which are usually privileged) are usually blocked for I/O.

Let's have a look on the round robin scheduler with three processes in the queue: A B C:

```
A(time 0) B(time 10) C(time 10)  A's time slice is zero: let's do round robin scheduling:
B(time 10) C(time 10) A(time 10) ... one clock tick occurs ... the next one ...
B(time 8) C(time 10) A(time 10)  ... several clock ticks occur ... b's time slice is worn out ...
C(time 10) A(time 10) B(time 10) ... ten clock ticks later ...
A(time 10) B(time 10) C(time 10) ... now A has its share of CPU time again.
```

SVR2 Unix Implementation

Classical UNIX systems scheduled *equal-priority* processes in a round-robin manner, each running for a fixed time quantum. If a higher priority process becomes runnable, it will preempt the current process (if it's *not* running in kernel mode, since classical UNIX kernels were non-preemptive) even if the process did not finish its time quantum. This way, high priority processes can possibly starve low-priority ones. To avoid this, a new factor in calculating a process priority was introduced: the 'usage' factor.

This factor allows the kernel to vary processes priorities dynamically. When a process is not running, the kernel periodically increases its priority. When a process receives some CPU time, the kernel reduces its priority. This scheme will potentially prevent the starvation of any process, since eventually the priority of any waiting process will rise high enough to be scheduled.

All user-space priorities are lower than the lowest system priority. The usage factor of a user-process is calculated by the amount of compute time to real-time consumed by the process. A process that has used a lot of compute time in the last real-time unit is assigned a low user priority. Because interactive processes are characterized by low ratios of compute to real time, interactive response is maintained without any special arrangements.

If there are no processes eligible for execution, the CPU idles till the next interrupt, which will happen *at most* after one clock tick. After handling the interrupt, the kernel again attempts to schedule a process to run.

References

Ken Thompson, "UNIX Implementation", 2.3 - Synchronization and Scheduling, Bell Laboratories

Maurice J. Bach, "The Design of the UNIX Operating System", Chapter 8 - Process Scheduling and Time, Prentice Hall

Shortest Process Next

Version of SRTN (Shortest Remaining Time Next) for interactive systems. The Problem here is that we can't say what the user's next command will be. This algorithm needs prediction :)

Lottery Scheduling

Lottery Scheduling is a simple algorithm that statistically guarantees a variable fraction of processor time to each runnable process. The concept is much like a lottery. At each scheduling decision, each runnable process is given a number of "lottery tickets". Then a random number is generated, corresponding to a specific ticket. The process with that ticket gets the quantum.

Although there is no absolute guarantee that processes will be treated equally, the frequency of scheduling events in a preemptive multitasking system means that it comes very close to doing so. The number of tickets given to a process divided by the total number of tickets is the statistical

fraction of the quanta given to that process. For example, if these processes are given this number of tickets:

foo - 5
bar - 7
bash - 4

The fractions of processor time given to each should be:

foo - 5/16 - 31%
bar - 7/16 - 44%
bash - 4/16 - 25%

As you can see, it is trivial to create a fine grained priority system: just give higher priority processes more tickets.

Advantages of Lottery Scheduling include fine grained priorities and statistical fairness. Disadvantages include the need for a reliable random number generator and non-absolute guarantees, especially on systems with large quanta.

You need to implement a Random Number Generator to use this algorithm.

Batch Scheduling Algorithms

First Come First Served

This scheduling method is used on Batch-Systems, it is NON-PREEMPTIVE. It implements just one queue which holds the tasks in order they come in.

The order the tasks arrive is very important for the Turnaround-Time:

Task1(24) Task2(6) Task3(6)
avg. Turnaround-Time = (24 + 30 + 36) / 3 = 30 time units (this assumes all tasks arrive at time 0)

Task1(6) Task2(6) Task3(24)
avg. Turnaround-Time = (6 +12 +36) / 3 = 18 time units (this assumes all tasks arrive at time 0)

Strengths:

-Simple
-Fair

Problems:

-Convoy Effect
-Order of task arrival is very important for average Turnaround time

Shortest Job First (SJF)

Is also NON-PREEMPTIVE. It selects the shortest Job/Process which is available in the run queue. This scheduling algorithm assumes that run times are known in advance.

Strengths:

-nearly optimal (Turnaround Time)

Problems:

-Only optimal if all jobs/process are available simultaneously
-Usually run times are not known ...

Shortest Remaining Time Next

Preemptive version of SJF (Shortest Job First). Schedulre picks the job/process which has the lowest remaining time to run.

Strengths:

-probably optimal (Tournaround Time)

Problems:

-again run times must be known

Highest Response Ratio Next

This page is a stub.
You can help the wiki by *accurately* adding more contents to it.

Real-Time Scheduling Algorithms

Real-Time Scheduling Algorithms are a special class of algorithms of which it is required that they can guarantee a process will be done before its deadline. The only way these algorithms can work is if they at least know when the deadline for a process is, and how much the process takes of the system. Only if the system is not overloaded (subjective term) can the threads be guaranteed to finish before their deadline.

Each task has to be scheduled X_t times a second, or every Y_t milliseconds ($Y_t = 1000 / X_t$). Each run of that task takes at most Z_t milliseconds. This task then creates a load of $L_t = Z_t / Y_t$.

The system as a whole has a load L, which is the sum of all task-loads: $L = \mathrm{E}\, L_t$. If the system load exceeds 0.7 (in some rare cases it can be slightly larger, but we don't count them) the system is unschedulable using Rate Monotonic Scheduling. If this system load exceeds 1.0 it is unschedulable for any real-time system. Note that for normal systems any load is possible, including the ones that are extremely large. They will make the system very unusable though.

Rate Monotonic Scheduling

Rate Monotonic Scheduling is a way to schedule Real-Time threads in such a way, that can be guaranteed that none of the threads will ever exceed their deadline.

The load of the system may vary, but there is a utilisation-based test that, if satisfied, guarantees that the system will always be schedulable. As an example the utilisation limit for a system with one process is 100% (as there is no need for preemption). The utilisation limit for a system with 3 processes is approximately 69%.

The utilisation based test is *sufficient* but not *necessary*. If a process set passes the utilisation based test, it will be schedulable. However, process sets can be constructed that fail the utilisation test but are in fact (trivially) schedulable.

The RMS scheduling works by assigning each task a priority based on its interval. The task with the shortest interval gets the highest priority and the task with the largest interval gets the lowest priority (still real-time though). The tasks are then run similar to a prioritized preempting [#Round-Robin]. This means, any task that can run runs, and if a task runs but a task with a higher priority is available, the higher one runs instead.

If your system is based on a Round-Robin scheduler, this is the easiest way to do Real-Time scheduling.

Earliest Deadline First

Each task in an EDF scheduler is assigned a _deadline_ (e.g. a moment in the future at which the task _must_ be completed). Every time a task is inserted in the system or completed, the scheduler looks for the task which has the closest deadline and selects it for execution. In order to ensure that the scheduler is still able to meet each deadline, a '*monitor* must evaluate if each new task doesn't overload the system and deny execution if it will do so.

In order to implement EDF-based system, one will have to know both the _deadline_ of the task (which could optionally be computed as "no more than X ms in the future") and the expected time needed to perform the task (required by the monitor). QoS network routers usually implement variants of EDF scheduling.

Again, there is a utilisation based test for EDF systems. The limit is simpler however - it is always 100%, no matter how many processes are in the set. This makes dynamic analysis of schedulability easier. Not only that, but the EDF utilisation test is both *sufficient* and *necessary*, so can be relied on to provide an accurate indication of schedulability.

For more information, see "Real time systems and programming languages" by Burns & Wellings.

See Also

Articles

- Real-Time Systems

Threads

- Real-Time Scheduling

External Links

- Linux 2.6 scheduler, under FDL (https://web.archive.org/web/20070818053107/http://josh.trancesoftware.com/linux/)
- Wikipedia:Scheduling (computing)