CPU and OS

The **CPU** has always been the most important and the most coveted resource in a computer system.

Since the introduction of multiprogramming, it is the norm that there are several processes competing for the right to use the CPU. When the OS gets the CPU (for whatever reason[a]) it performs the needed actions and tries to give the CPU back to a process.

Thus, the last instruction of every OS module is an unconditional **goto** to a dedicated handler called the **scheduler**. If the system has more than one CPU the scheduler may exist in several copies.

[a]Some interrupt handling excepted.

Scheduler in action

A scheduler can be non–preemptive, in which case if simply returns the **CPU** to the process that was using it last, if the process is ready. If the process is not ready anymore, a non–preemptive scheduler acts like a preemptive one.

When a preemptive scheduler is invoked it performs two actions:

Picks a process: only ready processes are considered. If there are several ready processes, the OS pick the most worthy of them using a scheduling algorithm.

Gives the CPU to the selected process: this operation, called dispatching, involves a full context switch.

In the past, a great deal of work has gone into devising clever and efficient scheduling algorithms.

A bit of history

In the early days of computing, scheduling was done manually (only one process was served at a time).

When multiprogramming was introduced (in OS/360 or maybe earlier) it became necessary to automate this task. When interactive users appeared w(with timesharing) the scheduling algorithm became one of the most important components of an OS.

the relevance of scheduling came to an abrupt end when personal computers became widespread. Today's operating systems are prepared to handle many processes concurrently, but they face an average ready queue size that is close to 0 and seldom exceeds 1. In such circumstances any correct decision is perfect.

Scheduling remained relevant in two specialised domains: supercomputers and network servers. Both domains are populated by CPU–hungry processes that have to be forced to share the CPU.

REDMOND, WAIn what CEO Bill Gates called ”an unfortunate but necessary step to protect our intellectual property from theft and exploitation by competitors,” the Microsoft Corporation patented the numbers one and zero Monday.

At a press conference beamed live to Microsoft shareholders around the globe, Bill Gates announces the company’s patenting of the binary system.

With the patent, Microsoft’s rivals are prohibited from manufacturing or selling products containing zeroes and ones–the mathematical building blocks of all computer languages and programs–unless a royalty fee of 10 cents per digit used is paid to the software giant.

The sad life of a scribe

Before proceeding, let us examine the prospects of two beginner processes: Crunch and Scribe, both exec'ed into being at the same time:

Crunch: reads in one integer n, computes the smallest prime number larger than 2^n and prints its last 10 digits. The central part–the computation–takes about $n^6 \mu s$.

Scribe: reads in two strings $\mathcal{F}_1$ and $\mathcal{F}_2$ and copies file $\mathcal{F}_1$ onto file $\mathcal{F}_2$ (assuming $\mathcal{F}_1$ exists). The central part–the copying–takes about $5ms$ per block of data (let the block be 1 KB).

Both processes start at almost the same time (why not exactly the same?). The OS gives the CPU for 1 second; the longest waiting process gets it (that's fair!).

The value read by Crunch is 20; the filename provided to Scribe names a file of 100 blocks. Given these inputs, Crunch needs about 70 seconds to finish (in an ideal world) while Scribe should be done after about a second (100 reads+100 writes).

Alas, reality does not match theory. Crunch will indeed finish in less than 70 seconds but the poor Scribe will take slightly longer than Crunch in spite of needing only a second of disk i/o time and practically no CPU time at all.

Who stole the kishka?

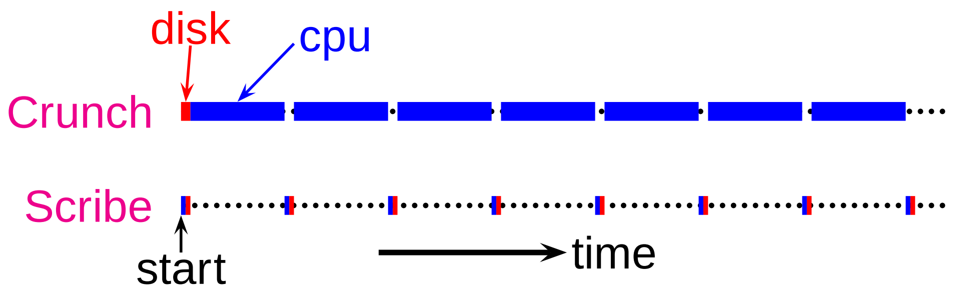
To find the cause of the strange conspiracy against the Scribe process, we might start by decreasing the time slice from 1s to 100ms. With this timeslice Crunch sees no difference in speed while Scribe finishes in about 20 seconds.

Reducing the timeslice even more drastically, to 10ms, will see Scribe finish in slightly over 2 seconds while the time taken by Crunch still is around 70s.

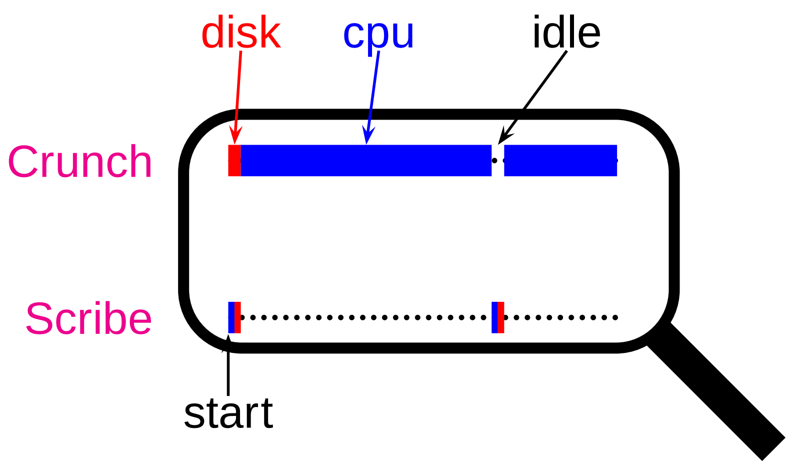
The problem must therefore be with the time that Scribe spends waiting for the CPU. But Scribe doesn't need the CPU!

The CPU is needed for everything

When a process wants to execute a system call (any system call) it must have the CPU to do so. This is because a system call is a CPU instruction.



Through a magnifying glass:



Moral

The two processes behave very differently:

Crunch: needs the CPU and practically nothing else. Its execution time is proportional to the speed of the CPU. Such processes are called **CPU**–bound.

Scribe: does only input–output and needs the CPU only to execute the next system call. Its execution is proportional to the access time to the i/o device needed (disk in most cases) and the speed of the CPU is of no relevance. such processes are called I/O–bound.

It should be clear that while the speed of the CPU matters little to an I/O–bound process, the availability of the CPU is of great importance.

Scheduling criteria

The main goals of a general–purpose scheduling algorithm:

Fairness: comparable processes should get similar treatment.

Balance: keep in use as many resources as possible (probably weighed by their importance).

Response time: take as little time as possible to start a process.

Predictability: it is increasingly important that the variance (departure from average) of scheduling behaviour be limited. This is in interactive systems (response time), real–time systems especially involving audio–visual aspects (they are synchronous).

With the above goals duly noticed, other criteria are used, mainly CPU utilisation; this is a leftover from prehistoric days.

Scheduling algorithms

Many algorithms were used in the days of large mainframes. These included:

First–Come–First–Serve: a process is given the CPU for an infinite timeslice. When a process gives the CPU away (only voluntarily) it is placed at the end of the ready queue, as if it was just created. The most hated scheduling policy of all.

Priority scheduling: this policy exists in 2 forms: static priorities (as good as feudalism was) or dynamic priorities (also known as priorities with aging) which is fairly reasonable.

Round Robin: this is the most commonly used scheduling algorithm. It has no serious weaknesses.

Multilevel Feedback Queues: A combination of **RR** and dynamic priorities where priorities are derived from the past behaviour of the process. Good but complex and easy to dupe by enterprising students.

Round Robin

This is *“one of the oldest, simplest, fairest, and most widely used algorithms”* according to Tanenbaum.

The process at the front of the ready queue is given the CPU for a fixed timeslice (also called quantum). When the timeslice expires, the process goes to the end of the ready queue.

The time quantum is the only parameter of RR. When too large, RR loses its predictability; when too small, it wastes much CPU time on context switches (but is excellent in other respects).

The perfect time quantum? Nobody knows, but the duration of a disk i/o operation seems to be a logical lower bound (5–10 ms).

Feedback queues

The general idea is to give priority to processes that need the CPU mostly to issue input–output system calls (these are interactive processes and i/o–bound processes).

The scheduler maintains multiple ready queues ordered from highest $\mathcal{Q}_0$ to lowest $\mathcal{Q}_n$ priority ($n > 0$).

Each queue is a **RR** queue with a timeslice that increases with the queue number (typically, timeslice of $\mathcal{Q}_i$ is twice the length of the timeslice of queue $\mathcal{Q}_{i-1}$).

When the CPU is to be given away, the scheduler picks the first process from the highest non–empty queue and dispatches it for a time equal to the timeslice of that queue.

Suppose that the CPU is given to $\mathcal{P}$ from queue $\mathcal{Q}_i$. The timeslice is t_i.

Three possibilities exist:

- $\mathcal{P}$ uses the whole timeslice t_i. It is demoted to the next lower queue, $\mathcal{Q}_{i+1}$
- $\mathcal{P}$ relinquished the CPU after a period of time less than $t_i/2$. $\mathcal{P}$ becomes blocked; when it becomes ready again, it is placed at the end of $\mathcal{Q}_{i-1}$ (promoted) if $i \neq 0$.
- $\mathcal{P}$ relinquished the CPU after a time period no shorter than $t_i/2$. $\mathcal{P}$ becomes blocked; when it is ready again, it is placed at the end of the queue that it came from, $\mathcal{Q}_i$.

This algorithm promotes processes that use resources other than the CPU which makes sense, as it keeps the whole system busy, not just the CPU.

Linux scheduler

Some references for the truly interested:

- The basic paper by Josh Aas
- An overview by Amit Gud
- Another site with the basic paper by Josh Aas
- Paper on process management in Linux
- Anonymous lecture notes from Freie Universität Berlin