Louis Pouzin

CTSS was developed during 1963 and 64. I was at MIT on the computer center staff at that time. After having written dozens of commands for CTSS, I reached the stage where I felt that commands should be usable as building blocks for writing more commands, just like subroutine libraries. Hence, I wrote "RUNCOM", a sort of shell driving the execution of command scripts, with argument substitution. The tool became instantly most popular, as it became possible to go home in the evening while leaving behind long runcoms executing overnight. It was quite neat for boring and repetitive tasks such as renaming, moving, updating, compiling, etc. whole directories of files for system and application maintenance and monitoring.

Louis Pouzin, André Bensoussan

In the same vein, I also felt that commands should be usable as library subroutines, or vice versa. This stemmed from my practice (unique at the time) of writing CTSS commands in MAD (Michigan Algorithm Decoder), a simplified Algol-like language. It was much faster and the code was more maintainable than the IBM 7094 assembly code. Since I needed MAD friendly subroutine calls to access CTSS primitives, I wrote in assembly code a battery of interface subroutines, which very often mimicked CTSS basic command functions. Or I wanted to make commands out of subroutines which handled common chores. I felt it was an awkward duplication of effort. However, I did not go further in the context of CTSS.

Then in 64 came the Multics design time, in which I was not much involved, because I had made it clear I wanted to return to France in mid 65. However, this idea of using commands somehow like a programming language was still in the back of my mind. Christopher Strachey, a British scientist, had visited MIT about that time, and his macro-generator design appeared to me a very solid base for a command language, in particular the techniques for quoting and passing arguments. Without being invited on the subject, I wrote a paper explaining how the Multics command language could be designed with this objective. And I coined the word "shell" to name it. It must have been at the end of 64 or beginning of 65. (See The SHELL: A Global Tool for Calling and Chaining Procedures in the System and RUNCOM: A Macro-Procedure Processor for the 636 System)

The small gang of Multics wizards found it a sleek idea, but they wanted something more refined in terms of language syntax. As time left to me was short, and I was not an expert in language design, I let the issue for them to debate, and instead I made a program flowchart of the shell. It was used after I left for writing the first Multics shell. Glenda Schroeder (MIT) and a GE man did it.

Time-sharing was a misnomer. While it did allow the sharing of a central computer, its success derives from the ability to share other resources: data, programs, concepts. It cracked a critical path bottleneck for writing and debugging programs. In theory this could have been achieved as well with a direct access approach. In practice it could not.

Direct access hems users in a static framework. Evolution is unfrequent and controlled by central and distant agents. Creativity is out of the user's hand.

Time sharing, as it became popular, is a living organism in which any user, with various degrees of expertise, can create new objects, test them, and make them available to others, without administrative control and hassle. With the internet experience, this no longer need be substantiated.

Posted to feb_wwide 25 Nov 2000