Great Ideas in Computing

University of Toronto CSC196
Winter/Spring 2019

Week 12: December 7-December 9 (2020)

Announcements

- Today will be our last meeting.
- Assignment 4 should have been submitted. Regrade requests must be made by Monday, December 14.
- Marta and I will be busy grading and submitting grades. We will try to get the grading done as fast as we can. Feel free to email me if you would like to talk about anything to do with the course or even about future CS courses.
- I think there is still time to submit your evaluation of the course and I urge everyone to do this. Beyond the evaluation form, please feel free to send me suggestions on topics you think should have been included and what topics you would rather not have in this course.

Some great ideas we did not discuss

There are many great ideas that we have not had time to discuss. In our remaining time, I would like to at least mention some great ideas. Namely, I want to briefly briefly the first two great ideas in the list below.

- Programming languages (and compilers) in contrast to machine code and assembly languages.
- Operating systems in contrast to submitting jobs individually
- Graphical user interfaces (GUIs) (and the mouse) in contrast to a command line interface
- Packet routing in contrast to virtual circuit routing and the internet
- Personal computers in contrast to main frames

If all of these seem like "duh, what else would you do", that is perhaps the best evidence of a great idea; that is when it becomes so common place, that it is hard to imagine anything else. But these great ideas often came with resistance.

Fortran, the first commercially successful programming language and the Fortran optimizing compiler

Please read "The History of Fortran, I, II, and III" by John Backus. Backus is a Turing award winner for Fortran and programming languages. See also Jon Nebel's history of Fortran.

Last week when we were ending the class, I mentioned programming languages in contrast to machine code and assembly language. Someone said **compilers**. That was a very important comment.

Fortran was developed at IBM by a team led by John Backus. More specifically, Fortran and the initial Fortran complier were designed for the IBM 704.

Fortran, the first commercially successful programming language and the Fortran optimizing compiler

Please read "The History of Fortran, I, II, and III" by John Backus. Backus is a Turing award winner for Fortran and programming languages. See also Jon Nebel's history of Fortran.

Last week when we were ending the class, I mentioned programming languages in contrast to machine code and assembly language. Someone said **compilers**. That was a very important comment.

Fortran was developed at IBM by a team led by John Backus. More specifically, Fortran and the initial Fortran complier were designed for the IBM 704.

As Backus says "**We certainly had no idea that languages almost identical to the one we were working on would be used for more than one IBM computer, not to mention those of other manufacturers**"

The state of "automatic programming" in 1954

Although one can find proposals for programming languages before Fortran, the state of the "art of automatic programming" was basically to use some primitive form of assembly language or interpreters to partially overcome tedious aspects of machine code.

Programming required considerable care and skill in order to overcome the limitations of existing computers; namely,

- Very small memories
- The lack of index registers (e.g,, to access elements of an array)
- The lack of floating point operations
- Limited instruction sets
- Rudimentary input/output facilities.

A programmer had to deal with modifying memory addresses, implement floating point operations, and implement loop constructs. Assembly language allowed symbolic names for variables and some macros to help overcome the limitations of early machines and the absense of programming languages.

The prevailing view: compiled source code would be too slow

In the early 1950's, "computer cycles" and access time to a computer were precious commodities. Early experience with automatic programming systems led to the concensus that compiled code could not compete with the efficiency of hand crafted machine code.

The claim was that efficient coding required the "human intelligence" of experienced programmers. Where have we heard before that something required human intelligence but is now done basically without human intelligence?

The prevailing view was that compiled code would increased computing time by a factor of five or ten. And even a slowdown by a factor of two would not be acceptable.

Even visionaries can be wrong

John von Neumann was a brilliant mathematician (in many fields). He introduced game theory and was a computing visionary (e.g. the von Neumann model).

In the 1950's von Neumann was employed as a consultant to IBM to review proposed and ongoing advanced technology projects. One day a week, von Neumann "held court" at 590 Madison Avenue, New York. On one of these occasions in 1954 he was confronted with the FORTRAN concept. Backus remembered von Neumann being unimpressed and that he asked "why would you want more than machine language?" He dismissed the whole development as "but an application of the idea of Turing's short code."

One of von Neumann's Princeton students (Donald Gillies) observed that graduate students were being used to hand assemble programs into binary (probably for the Institute for Advanced Studies IAS machine). Gillies took time out to build an assembler, but when von Neumann found out about the assembler he was very angry, saying (paraphrased), "It is a waste of a valuable scientific computing instrument to use it to do clerical work. "

Sucsess beyond expectation

But Backus and his team were successful. The created the Fortran I language and that language has survived decades of modifications and imrovements and is still used for scientific applications on generations of diffefrent (von Neumann model) computer architectures.

The early success of Fortram was made possible by their optimizing compiler that was competitive with programmer machine code.

The new language and compiler caught on quickly; “programs computing nuclear power reactor parameters took now hours instead of weeks to write, and required much less programming skill”.

Moreover, as we implied above, programs now became portable. Fortran won the battle against Assembly language and machine language codingr.

Further reflections by Backus

Backus states that he and his team were "hopelessly optimistic" that "Fortran should virtually elimninate debugging". They also thought that Fortran programs could be *reverse engineered* to create the specification for a problem.

The article by Backus is refreshingly honest as on the one hand he admits to not initially understanding the long term significance of what they were doing but on the other hand being overly optimistic.

In a final set of comments Backus argues against "von Neumann languages" in that they "create enormous, unnecessary intellectuall;y roadblocks in thinking about programs and in creating the higher-level combining forms required in a powerful programming methodology".

He goes on to say that von Neumann languages "constantly keep our noses pressed in the dirt of address computation and the separate computation of single words".

I would say that modern programming languages have evolved to escape this"dirt" but can still be thought of as von Neumann languages.

A one slide overview of operating systems

Earlier this term Eyal del Lara discussed virtualization and the multi layers of operating systems that are used in say cloud computing.

But lets go back to say the 1960s and the great idea of operating systems without the added complications of mutli-core and parallel systems.

An operating system (OS) is the software that controls the usage and sharing of resources in a computing system. An OS can be for a single user or for multiple users. But even a single user can be executing many simulataneoaus processes. What are these resources?

A one slide overview of operating systems

Earlier this term Eyal del Lara discussed virtualization and the multi layers of operating systems that are used in say cloud computing.

But lets go back to say the 1960s and the great idea of operating systems without the added complications of mutli-core and parallel systems.

An operating system (OS) is the software that controls the usage and sharing of resources in a computing system. An OS can be for a single user or for multiple users. But even a single user can be executing many simulataneoaus processes. What are these resources?

- Computing cycles (i.e. how many cycles to give each process
- I/O Management
- Memory management; virtual memory and paging/caching
- File systems

Of course, a one slide overview doesn't reflect the complexity of operating systems as all of these resources have to be mananged simulataneoulsy. And while it was not an initial concern, modern operating systems need to contend with security issues.

Virtual memory managment

To elaborate on just one aspect of an OS, lets consider memory managment. There are at least two main challenges for memory manaagement.

- An OS has to respond to dynamically changing memory requirements of the processes it is managing. For example, consider an array that grows and shrinks according to insertions and deletions.
- An OS has to manage a hieracrhy of memory; that is, different levels of memory: small but very fast registers and caches, random access memory RAM, disk memory.

While again one might say "of course, memeory management has to be done by an OS and not by individual programmers". But this was not always the case. Virtual memory is one aspect of virtualization as discussed by Eyal de Lara and some people believed that to efficiently use the memory the programmer had to know the application.

Moving data between different levels of the memory is simply referred to as *paging* or *caching*.

Paging and LRU

Fast memory (the cache) does not contain that many words. The RAM is much larger but much slower to access (lets say by a factor of 10). And disk memory is again much larger (than RAM) and again much slower.

A paging system determines how to move information between these diffefrent levels of the memory hierarchy. One can move individual words (into and out of the cache) or blocks of memory.

It turns out that variants of a simple algorithm **LRU** (least recently used) works well in practice and analytically can be shown to have desireable properties.

The underlying idea for efficient virtual memory management is to exploit *locality of reference*. In most applications, memory accesses are not random but the next memory request is often determined by the most recent accesses. LRU works in phases, In a phase LRU marks pages in the cache whenever accessed and when a *page fault* occurs it removes an unmarked page. When all pages are marked it removes all the marks (except for the new request) and starts a new phase.