System Software - Assemblers / Compilers / Linkers

Videos

https://youtu.be/lp4ib9WoqjU - Lect-22-2150-pt1.mp4
https://youtu.be/ea6ibgXmJBA - Lect-22-2150-pt2.mp4
https://youtu.be/3taUHQnKqjQ - Lect-22-2150-pt3.mp4

From Amazon S3 - for download (same as youtube videos)

http://uw-s20-2015.s3.amazonaws.com/Lect-22-2150-pt1.mp4
http://uw-s20-2015.s3.amazonaws.com/Lect-22-2150-pt2.mp4
http://uw-s20-2015.s3.amazonaws.com/Lect-22-2150-pt3.mp4

There will be a pt4 and others for Friday's Lecture out later today.

Overview

One of the big developments in systems software (Chapter 8) was the development of high level languages. Please read Chapter 8 in the book.

The basic idea of system software is to build something that is farther away from the machine and easier for humans to understand and use.

Research in this is ongoing with companies like IOHK developing domain specific languages for FinTeck (Financial Technology). If this kind of a thing interests somebody -then- it makes a decent masters project to develop a DSL for things like the insurance industry.

How do Assemblers work.

Most assemblers are 2 pass systems.

Pass 1: Read the source and figure out where all the symbols are located. In this pass you pretend to do the assembly - but all you are looking for is the address of all the symbols.

```
        Load X
        Store Y
        Halt
```

```
  X,      Dec 10
  Y,      Dec 0
```

Start at pc = 0

So you fake generate a Load instruction, increment pc to 1.

Fake generate Store, increment PC to 2.

Fake Generate Halt, increment PC to 3.

Read Label X - store this as having an address of 3. Fake Dec 10, increment PC to 3.

Read Label Y - store this as having an address of 4. Fake Dec 0, increment PC to 4.

Find the end of file - so done with pass 1.

Star at top - keeping the symbol table with X, address 3, Y Address 4.

Generate Load X - with a load instruction of 0x1000 and the address of X - we know that now to be 3. So 0x1003.

Generate the Store Y - with a store instruction of 0x2000 and the address of Y - we know that now to be 4. So 0x2004.

Generate the Halt, 0x7000.

Generate the Dec 10, 0x000a.

Generate the Dec 0, 0x0000.

Done.

How do compilers work.

Compilers took a lot longer to build. They have lots of fun parts. The original compiler, Fortran was originally build on circuit boards where the boards had to be put directly into the system. The 1's and 0's were generated by cutting diodes out of the board. I bet the developers got blisters using wire cutters.

Fortran is still around - and still used - but it has "evolved" and is a much more useful language. I learned Fortran in college - and still occasionally program in it. Specifically I made fixes to some genetic analysis code last week for looking at the evolution/mutation rate in COVID-19.

The big developments in compilers involved parsing of languages into trees and the formal specification of languages. Compilers today are super-reliable. In 1989 when you had a defect in your code the question was always is the compiler failing or is my code failing. Also compilers tend to produce really nice error messages today. I used a Fortran-66 compiler that if you had a syntax error it produced the message “syntax error.” - but did not even tell you the line number that the error was on.

So ...

How do copiers work. – The 1990’s easy view – Later we will look at how a modern compiler works.

1. Symbol Table - Symbols are not just address - they have type and size information with them.
2. Scanner - This takes the input and coverts it into “tokens” so an “id” is a single thing to the next phase.
3. Parser - This converts the scanned tokens into a tree representation of the code. This is often called an Abstract Syntax Tree (AST).
4. Code Generator - This takes the AST and generates code for it in assembly Language
5. Optimizer - This takes the code and re-writes it to be better code.
6. Assemble the output into a binary.
7. Link multiple binaries together into a final program.

Interpreters.

Interpreted languages like “java” and “python” were built to run the AST directly - at least in the original form of the language. Later this AST was translated into a P-CODE - an assembly for a virtual machine that did not exist. This made the execution faster. Then things like optimizes could work on the code. Also porting to new machines just meant re-writing the emulator. Then they started Just-In-Time compiling the code to make them faster. This meant doing the final step of the compile - to translate the P-Code into machine code - as the program ran. This made them faster.

More modern languages like Go and Rust have avoided this - by doing all the work up front.

The most common data storage and analysis systems are interpreters. Microsoft Excel for example.

System Software - Batch Processing Code

Most software from the 1960s to the 1990s for business ran on a “bach” processing model. It worked a lot like an assembler - but in this case it worked on financial data.

Banks still use this - the IRS still uses this. If you get a mortgage - then the bank will use software that works this way. Telephone companies and your utility company still work this way.

Often a database persists some of the data - but the basic model is:

1. Read in a record of data
    - Validate the record
    - Do some processing on the record (save in a database, update a database)
    - Produce some output for this record
    - Summarize this data - for final reports
2. Move to next record (back to (1))

System Software - with GUI interfaces (Microsoft Excel, Photoshop etc)

The 1980 to 2000 saw the development of “desk top” applications like Excel and Photoshop. The are what-you-see-is-what-you-get applications with a GUI interface. You click and things change. A large number of compiled languages were built during this time to reflect the difficulty in developing these applications. Often the GUI interface in these is 90% or more of the code in the entire application. Also applications moved from the Mainframe (batch processing) to the Desktop (GUI with a mouse).

Good Side: GUI is really good for drawing and fast changing of data. This is fantastic for individual creativity and one-off operations.

Bad size: There is no way to re-use or build a higher-level abstraction with a GUI. If I am processing an image to add a watermark with Photoshop. Then it takes about 21 clicks to open image, open a layer, drag the “water mark” layer on top of the image, combine the lawyers, save the resulting image. This is all good when you have 1 image to work on. When you have 100,000 images a week this is a huge waste of time. The same task cam be implemented in shell-script (programming language) in 8 lines of code and do all 100,000 in the same time as it takes a human to do 4 images. Also the shell script will get the “water mark” in the same correct location every time, not in a variable location.

Good Side: Very little training is required.

Bad Side: The code is really really hard to test. This means that there are lots of defects left in it.

Good Side: Well designed applications are a joy to use.

Bad Side: Most GUI applications are very poorly designed. Figuring out how users actually work is incredibly hard. Apple develops 9 different to finish interfaces for every change that they make to a user interface then picks the best one. That is very costly and most business apps don't have that luxury.

Good Side: it is quick to make changes.

Bad Side: The application is "real time". Anything longer that about 10ms in response to a mouse move or click makes the interface very hard to use. Not everything that we want computers to do is can be done in 10ms. This introduces concurrency into the code - and that is very hard to work with.

```
Start a loop waiting for an event.
        On Event – Propogate the change to the region of the screen handler for that ev
                Given the context (region) of the vent – then take action to update the
                Redraw the screen based on the changed data in the model.
```

The real difficulty in this is that every region of the screen can now do changes to the "model" for every mouse / keyboard or other event in any order. Every chunk of code also has to make the changes in real time. Also users frequency make mistakes and want to un-do the changes that they just made.

More modern GUI applications

The web is the replacement for most desktop applications. This moves the GUI interface to a client and the processing to a server. This has all sorts of advantages - but it also makes the development more distributed. The biggest advantage is that the users will not install any software on a local system to get it to work. The biggest disadvantage is security.

The last lecture on web-lecture is on how they work.

System Software - Databases

The NASA Saturn V project saw the development of a truly new kind of system software. This was the IBM Information Management System. This was the first "database" system ever developed. IBM followed this with a number of years of research and finally developed a "query" language that allows for the concise storage and retrieval of data. This was standardized and is called SQL for Standard Query Language. It has it's problems but usually works fairly well.
One of the people, Larry Ellison, that worked at IBM on this left and purchased a small software company, renamed it Oracle and started producing a database based on this.
The two most important storage systems for data are Microsoft Excel, and the Oracle Database.

First thing is that databases store the data in a common location - and - the software remotely accesses the data. This means that the data is not local to the user and the database has to support a client-server architecture. Most of the web's client server system design came from the building of databases. Being client-server this means that the database has to have security that works. Some systems, Oracle, PostgreSQL have succeeded at this. Some like Microsoft SQL-Server(T-SQL) have known problems that have persisted in security for years. Also applications that run on top of these have to be security-aware and prevent things like malicious code injection.

Database have evolved and de-evolved. In the 1990s SQL and relational took over from the Network and Graph databases because it could do everything that they could do - and do it better and faster. Now in the 2020s we see a resurface in Network (Neo4j) and Graph databases (Mogo DB). My testing indicates that everything that you can do in Neo4j and MogoDB you can do in PostgreSQL at about 4 to 10 times faster - and - you can also do rational queries and a bunch of other things. There are some special applications like Time Series databases that work really well. PostgreSQL has a TimeSerise package - that appears to me to have more features and be the best time series database in existence. There are special databases like Google F1 built on top of BigTable. This is a relational database running on a distributed block share. It uses atomic clocks for synchronization of data. It is the worlds first "world scale" database. This has some really fantastic uses like "gmail" and google "docs". There is an open-source version of the same technique called Cockroach DB that is a port of PostgreSQL to use the distributed block share and not rely on atomic clocks. There are special network databases that Facebook uses. This is called Cassandra. At scale it makes sense. Scale being over 5,000,000 computers running it at the same time. Most people don't need 5 million computers. One of the non-relational databases that I use is called Redis - it is an in-memory database that can easily do 250,000 transactions per second on a single box and - it can scale to 3 or 4 million transactions with replication. This is a fantastic way to build data-queues and caching. I also recognized it's limitations. I don't try to store user-accounts and accounting records in an in-memory system.

Copyright