COMP 3400 Mainframe Administration

Christian Grothoff
christian@grothoff.org

http://grothoff.org/christian/

¹These slides are based in part on materials provided by IBM’s Academic Initiative.
Today

- Application Programming on z/OS:
  - Software Engineering Review
  - Common languages on z/OS
  - Build process
Applications Are Everywhere

Internet

Enterprise Network

Central Site

e-business
Browser

Web Server
Browser

Appl. Server

Business Systems

Databases

Client-Server

GUI Front-End

Terminal Processing

"Dumb" Terminal

Business Systems

Front End

Internet

Enterprise Network

Central Site

e-business with Legacy Systems
Browser

Web Server

Browser

Web Server

Personal Computer

Server

Personal Computer

Business Systems

Applications

Business Systems

Front End

Internet

Enterprise Network

Central Site
Roles in Application Development

• Application Designer
• Application Programmer
Application Designer

Determines the best programming solution for a business requirement using his understanding of:

- Business objectives of the company
- Company’s hardware and software
- Other roles in the mainframe IT organization

⇒ Must have a global view of the entire project
Types of Requirements

- Accessibility
- Interoperability
- Usability
- Managability
- Performance
- Serviceability
- Portability
- Availability
- Recoverability
- Fault-tolerance
Key Decisions

- Online processing or batch processing?
- Storage model: Database, tape, flat file?
- Programming language: Java, Cobol, PL/1, Assembler?
- Platform: z/OS, Linux, UNIX, Windows?
- Hardware: type of server, capacity?
- Develop or purchase or both?
The Waterfall Model

1. Gather requirements
2. Analysis
3. Design
4. Code & test
5. User, System tests
6. Go production
7. Maintenance
Application Developer

• Builds, tests and delivers applications (for end users)
• Works from application designer’s specifications
• Uses tools to change code, compile, build and test applications
The Grind (of the Application Developer)
Traditional Application Development

1. Edit source and make modifications
2. Submit compile job to JCL (verifies syntax, compiles, links, runs tests)
3. View job output in SDSF, check for errors
4. If there are errors, go back to step 1
5. Save source code in repository
Programming tools

- ISPF Editor (or remote using WebSphere/Eclipse/Rational)
- Repository for source code (PDS, SCLM or other repository)
- Job monitoring and viewing (SDSF)
- Debugging tools (WebSphere, Rational, etc.)

IDEs accelerate development – also on the mainframe!
IDE features

• Edit source on workstation
• Compile on workstation or on platform
• Remote debugging (program on mainframe, debugger on workstation)
• Support for many programming languages
• Integration with source code repositories
Testing

Before production use, code must pass many types of tests:

- Unit testing
- Functionality and acceptance testing
- Performance (stress) testing
- Integration testing
Moving to Production

- Final tested programs
- Promote To production
- Production Repository
More than code

- Document operational procedures
- Provide training manuals (for users and administrators)
- Implement change control process
- Help handing code over to system operators
- Responsibility for maintenance may change to others or stay with developers
Maintenance

- Maintenance and enhancement is the primary role of most application programmers for mainframes.
- Existing applications are often written in COBOL and PL/1.
- New applications are in Java, COBOL and PL/1. Note that COBOL and PL/1 continue to be enhanced.
Types of Programming Languages

- Low-level languages: Assembler and C – direct correspondence to physical machine

- High-level languages: COBOL, PL/1, Java – require complex compilation and/or runtime environments

- Special-purpose languages: RPG, CSP, QMF, SQL – usually used for a specific subproblem

- Scripting languages: Perl, REXX – fast development, write-only code
Choosing a programming language

- Performance requirements
- Interaction with code in other languages
- Knowledge of development team
- Scope of the project
- Tool support
Using Assembler on z/OS

Assembler is not usually used for application development, but for:

- Accessing system control blocks
- High performance subroutines where extreme execution efficiency is needed
COBOL on z/OS

- Traditional language for business applications
- Can be integrated with web-oriented business processes
- Interoperability with Java
- Support for XML and Unicode
PL/1 on z/OS

- Used for system programming
- Used for engineering and scientific applications
- Less verbose and English-like compared to COBOL
- Can use symbolic file names just like COBOL

\[2\text{In fact, all z/OS languages support the use of symbolic file names, even Assembler.}\]
C/C++ on z/OS

- Used for system-level code, text processing, graphics, etc.
- z/OS is POSIX compliant!
- C language is standardized, but z/OS uses EBCDIC
  ⇒ C strings are in EBCDIC, not ASCII!
- The z/OS C compiler is not gcc!
  ⇒ No gcc extensions, for example, // is not a comment in C!
Java on z/OS

- Interfaces with COBOL and PL/1
- Interfaces with DB2 and IMS
- Support for JNI (interface with C/C++ and other languages)
- Good IDE support (Eclipse, WebSphere)
CLISTs on z/OS

• Interpreted language
• Most basic CLISTs are lists of TSO/E commands
• Commonly used for writing ISPF panels
• Commonly used for one-time quick solutions for small problems
REXX on z/OS

• Can be compiled or interpreted
• More expressive than CLIST
• Also available on other platforms (GNU/Linux, W32, z/VM)
• Used for routine tasks (submitting TSO/E commands), ISPF panels, system programming, etc.
Questions

?
Question!

Which programming language is the best to use?
z/OS Language Environment

z/OS Language Environment product provides a common environment for all conforming high-level language (HLL) products:

• Common language development and execution environment for application programmers on z/OS

• Consolidates runtime library functions previously provided in individual library products

⇒ Similar to Microsoft’s .NET framework, but without IR
Advantages of z/OS Language Environment

Having a common run-time environment for all participating HLLs...

• allows programs to seamlessly call one language from another

• avoids replication of essential run-time services such as message handling and storage management

• provides consistent interfaces across programming languages
Language Environment Overview

- C/C++ language specific library
- Java language specific library
- COBOL language specific library
- FORTRAN language specific library
- PL/I language specific library

Language Environment callable service interface, common services, and support routines
Compiling on z/OS

• A source program is divided into logical units called modules

• Each source module is assembled or compiled by the respective language translator

• The compiler generates an object module

• Object modules are processed by the binder to create a load module which can be executed
Compilation Overview

Source module → Precompiler → Compiler → Object module → Binder → Load module
Separate Compilation and Relocation

When compilers translate source code into object code, they:

- Assign *relative* addresses to all instructions, data elements and labels, starting from zero
- Run-time addresses are in the form of a base address plus a *displacement* (to allow programs to be relocated)
- References to external programs or subroutines are left as *unresolved*
Relocatability

Even the final load modules are relocatable:

- The code can be located at any address in virtual storage\(^3\)

⇒ An identical copy of a program can be loaded in many different address spaces at different starting addresses

⇒ Physical copies in memory can be reused!

\(^3\)Within the confines of the residency mode
Source modules

- Source code written in the respective programming language
- Source programs are often stored in a PDS known as a source library
- A copybook is a source library containing prewritten text; it is used to copy text into a source program, at compile time.

⇒ copybook ≈ /usr/include/
Object modules

• Collection of one or more compilation units
• Contains machine code (and program data) in relocateable format
• Contains control dictionaries to resolve cross-references between sections of different modules
• Not executable
• Multiple object modules can be stored in an object library
Load modules

- Contains machine code (and program data) in relocateable format
- Contains *control dictionaries* to resolve cross-references between sections of different modules
- Can be loaded into virtual storage and relocated by the *program manager*

⇒ “executable” (with help of program manager)
Load libraries

Load libraries contain programs ready to be executed (usually load modules processed by the binder or linkage editor).

We distinguish:

- **System libraries** – unless a job specifies a private library, the system libraries are searched when a job specifies `//step EXEC PGM=program-name`

- **Private libraries** – user-written programs, searched only when JCL specifies DD statement defining JOBLIB or STEPLIB

- **Temporary libraries** – see IGYWCLG (later)
Binder

- Assigns virtual storage addresses to sections of the module and resolves references between modules
- Can process traditional data sets (PDS, PDSE) and z/OS UNIX files
- An older, more restricted version of the binder was called the linkage editor
- The batch loader and the program management loader can also be used to create a load module to execute only (but not to store the result in a library)
- The batch loader is replaced by the binder in later releases of z/OS
Compilation Overview II

Flowchart:

1. Source modules
   - Assembler or compiler
   - Object modules
   - Program management binder
     - Program object PDSE program library
       - Load modules in virtual storage ready for execution
       - Batch loader
         - Load modules in PDS program library
           - Linkage Editor
             - Load modules in PDS program library
               - Program management loader
Translating Assembler Code

Assembler language source statements

High Level Assembler

Messages and listings

Binder

Machine language version of the program

Executable load module
Translating COBOL Code

1. **HLL Source statements**
2. **HLL compiler**
3. **Messages and listings**
4. **Binder**
5. **Machine language version of the program**
6. **Executable Load module**
Ways for compiling COBOL

- Use a batch job (JCL), often with cataloged procedures
- Use TSO/E commands in CLISTs or ISPF panels
- Use the `cob2` command in the z/OS UNIX shell
IGYWCLG is a three-step cataloged procedure to:

- Compile (step is called “COBOL”)
- Link-Edit and
- Run (step is called “GO”)

a COBOL application. You must supply:

```
//COBOL.SYSIN DD *
```
Example: Compiling COBOL

//MYJOB JOB
//STEP1 EXEC IGYWCLG
//COBOL.SYSIN DD *

... INPUT-OUTPUT SECTION.
FILE-CONTROL.
   SELECT INPUT ASSIGN TO INPUT1 ...
   SELECT DISKOUT ASSIGN TO OUTPUT1 ...
...
/*
//GO.INPUT1 DD DSN=MY.INPUT,DISP=SHR
//GO.OUTPUT1 DD DSN=MY.OUTPUT,DISP=OLD
Example: Compiling PL/1

//MYJOB JOB
//STEP1 EXEC IBMZCLG
//PLKED.SYSIN DD *
OPEN FILE=INPUT1
OPEN FILE=OUTPUT1
READ FILE=INPUT1 ...
WRITE FILE=OUTPUT1 ...
CLOSE FILE=INPUT1
CLOSE FILE=OUTPUT1

/*
//GO.INPUT1 DD DSN=MY.INPUT,DISP=SHR
//GO.OUTPUT1 DD DSN=MY.OUTPUT,DISP=OLD
Questions

?
Tasks!

- Find IGYWCLG on your z/OS system!
- Do the exercises in section 10.9