Using LE Services in z/OS

Using LE Services in z/OS - Course Objectives

On successful completion of this class, the student, with the aid of the appropriate reference materials, should be able to:

1. Code programs using one or more of these LE-conforming compilers:

   Enterprise COBOL for z/OS and OS/390, COBOL for OS/390 & VM

   z/OS C/C++, XL C/C++

   Enterprise PL/I for z/OS and OS/390, PL/I for MVS & VM,
   Visual Age PL/I for OS/390

   or Assembler language

2. If appropriate, include FORTRAN programs in the mix, even though none of the current FORTRAN compilers are LE-conforming

3. Invoke LE message management services to create and issue user run time messages (in multiple natural languages, if necessary)

4. Use LE storage management services for holding large tables or entire files in virtual storage

5. Use LE condition handling routines to intercept and handle appropriate conditions under user control

6. Use LE date and time services for working with dates, times, and durations.

Assembler programmers may explore using preintialization and Library Routine Retention services.

Note: z/OS V1R6 and later support 64-bit LE-conforming applications in C/C++ and Assembler. 64-bit support is now also available for COBOL. 64-bit issues are not discussed in this course.

Using LE Services in z/OS - Topical Outline

Day One

Course Survey - what languages to include in the discussion

Using LE Services in z/OS - Topical Outline, p.3.

Course Survey - Languages Selection

❒ **This course is multi-lingual, but we don't talk about programming languages you will not be encountering**

❒ **So here is the time for you to specify which languages you are interested in exploring during this class**

❒ **Based on your selection(s) we will omit parts of lecture and labs that are not relevant to your work**

Language

_____ **Assembler**

_____ **C**

_____ **COBOL**

_____ **PL/I**

Section Preview

- **Introduction to Language Environment (LE)**
  - **What Is LE?**
  - **LE Services**
  - **Benefits of Using LE**
  - **Invoking LE Services**
  - **Tokens**
  - **LE Program Management**
  - **POSIX and z/OS UNIX**
  - **LE Program Management, Second Pass**
  - **LE and 64-bit Processes**
  - **Introduction to XPLINK**

What Is Language Environment?

❒ **The run-time library support for the versions of IBM mainframe compilers that will continue to be enhanced**

IBM Enterprise COBOL for z/OS is the current COBOL compiler for LE

✗ It supports the ANSI '89 amendment to the ANSI '85 standard (primarily intrinsic function support), along with selected extensions and some features from the 2002 standard

The current PL/I compiler for LE is IBM Enterprise PL/I for z/OS

The LE C/C++ compiler is and XL C/C++

Visual Age for Java, Enterprise Edition for z/OS

— These are often called the "LE conforming compilers"

❒ **Note that some earlier versions of these compilers are also supported, depending on the release of z/OS you are running**

❒ **LE provides support for Assembler**

❒ **LE also provides run-time support for VS FORTRAN (versions 1 and 2), FORTRAN IV H Extended, and FORTRAN IV G1**

✗ Although none of these are LE-conforming compilers

What Is Language Environment? (continued)

❒ **LE is a set of programs that provide the following capabilities**

A common run-time environment for many languages

✗ COBOL, PL/I, C, C++, Assembler, Java, FORTRAN

A set of callable routines that provide useful services for applications written using LE conforming compilers

✗ Date and time, storage management, mathematical, etc.

LE is used to support z/OS UNIX System Services, also called, more simply, z/OS UNIX

✗ Including support for UNIX file structures (directories, subdirectories, ... , files) and standard UNIX calls and services

LE supports DLLs (Dynamic Link Libraries) both under UNIX and native z/OS

✗ A DLL is a program object that contains one or more functions (subroutines) and variables that can be accessed from applications dynamically

✗ The latest C, C++, COBOL, and PL/I compilers support creating and accessing DLL functions and variables, as does HLASM (the High Level Assembler)

LE Conforming Programs

- A program is “LE conforming” if it establishes or runs under the LE run-time environment and follows LE conventions

- Programs compiled using the compilers designed for the LE environment are automatically LE conforming:

  IBM Enterprise COBOL for z/OS

  IBM Enterprise PL/I for z/OS

  XL C/C++

  Visual Age for Java, Enterprise Edition for OS/390

- These compilers automatically generate dynamic calls to the Language Environment initialization routines

  In fact, programs compiled and bound using these compilers must run under the LE run-time environment

- Of course, Assembler programs can also be written to invoke the LE initialization routines, but the Assembler doesn't automatically generate the linkages to these routines

  More on Assembler linkages later

LE Services

❒ **As an overview, the services available to Language Environment conforming programs fall into the following categories**

Storage Management - obtain and free memory dynamically

Condition Handling - detect errors and other conditions, and handle conditions in a consistent manner

Messaging Services - define message files that can be shared by many programs; issue messages, including

- ✗ substituting variables from programs
- ✗ route messages to various target locations

Date and Time Services - get and store date and time in various formats; convert between formats

Debugging Services - retrieve / set error codes; generate dumps; invoke a debug tool

Mathematical Services - Trigonometric functions; exponential and logarithmic functions; etc.

International Services - retrieve / set country, language, currency, and similar attributes, including support for locales

Using LE Services

- **Language Environment services are accessed using CALL statements (or CALL-like mechanisms, such as function references in C/C++)**

  All Language Environment services are subroutines

  All these subroutine names begin with “CEE”

- **A program using Language Environment services must be compiled using the appropriate compilers**

  Just inserting CALLs to these services and then compiling with an earlier compiler won't work because the service calls assume the LE environment has been established

- **However, note that non-LE conforming programs can run under the LE run-time environment (a COBOL II load module, for example, can be called by an Enterprise COBOL main program)**

  Details on mixing environments are discussed later

Portability

❒ **Language Environment conforming programs are portable to other platforms in the following sense**

Source code ports between z/OS, and z/VM and z/OS UNIX

LE supports executables generated by releases of older compiler versions, for example

- ✗ z/OS C/C++, OS/390 C/C++, C/C++ for MVS/ESA, C/C++ forz/VM
- ✗ Enterprise COBOL for z/OS, Enterprise COBOL for z/OS and OS/390, COBOL for z/OS and OS/390
- ✗ Enterprise PL/I for z/OS, Enterprise PL/I for z/OS andOS/390, Visual Age PL/I

Notes

Services unique to the z/OS environment do not port; these service names all begin with the string "CEE3" (there is often a corresponding service for other platforms, but not always)

Assembler language programs do not port to non-Z hardware platforms

❒ **The CEEGPID and CEE3INF services can be used to return the version of LE, the operating system platform a program is running under, and the environment (CICS, TSO, batch,** *etc.***)**

So applications can be designed to be portable at the source level to many different operating systems and environments

Benefits of Using LE

- **The LE compilers provide the latest language features**

  For example, Enterprise COBOL supports all COBOL II features plus the intrinsic functions added to the COBOL ANSI '85 standard in the 1989 amendment, along with Unicode and XML support and much more

- **LE provides a single, common run-time library supporting multiple languages**

- **LE provides a suite of useful services callable by all LE conforming programs**

- **LE programs may port across platforms, including the use of LE services**

- **Debugging LE programs can be more efficient than debugging non-LE programs, especially in a multiple language environment**

  LE provides a single dump formatting service, for example, and a single debug tool interface that can debug applications written in multiple languages

- **Programs may run significantly faster than programs compiled using previous compilers**

See following page

Performance Improvements Using LE

❒ **Multiple language applications have a single run-time environment and library**

And a streamlined path for inter-language communications

❒ **LE environment has optimized the CALL interface for all uses**

Gaining performance benefits when using system services

❒ **Programs can gain performance benefits because of new capabilities - options that simply haven't been available**

❒ **These combinations can result in improvements from 5% to 30%, depending on the application and amount of tuning done**

One company's experience (COBOL II to COBOL for MVS & VM)

✗ 5-10% improved performance for batch programs with no recompile (use LE runtime libraries)

✗ Up to 23% improved IMS transaction performance with recompile

Another company's experiences

✗ 9% faster performance moving from COBOL II R2 to COBOL for MVS & VM (production batch)

✗ 29% performance improvements in program that calls C service routines

❒ **On the other hand, some companies have experienced performance losses**

Invoking LE Services

❒ Assembler Language

If the main program for an LE application is Assembler, this program must invoke the Language Environment initialization code (this is discussed later)

Standard CALL syntax applies to invoking services, for example

```
Call   CEEMSG,(in_token,dest2,fc_token)
```

On return, check the contents of “fc_token”, not R15, to see if request was successful

Invoking LE Services, 2

❒ COBOL

Standard CALL syntax applies to invoking services, for example

```
Call 'CEEMSG' using in-token, dest2, fc-token
```

On return, check “fc-token”, not RETURN-CODE

Calls may be either static or dynamic

Invoking LE Services, 3

❒ **PL/I**

There is a source INCLUDE file, CEEIBMAW, shipped with the compiler, containing declarations for all the callable services

✗ Coding a "%INCLUDE CEEIBMAW" statement in your program saves you the effort of declaring any of the callable services

You must invoke Language Environment services using CALL, not a function reference, for example

```
%include CEEIBMAW;
.
.
.
call ceemsg (in_token, dest2, fc_token);
```

Check "fc_token" to test result of the service request, not PLIRETV

Invoking LE Services, 4

❒ **C/C++**

The “leawi.h” header file shipped with C/C++ contains declarations of all LE callable services and *OMIT_FC* (which can be used to omit the feedback code token parameter)

You must invoke LE services as procedures, you cannot invoke them as function calls

✗ LE services have the return type of *void*, and thus do not return values

Example

```
#include <leawi.h>
.
.
int main(void)
{
.
.
CEEMSG(&intoken,&dest2,&fc_token);
.
.
}
```

Note that input strings used as parameters are not null terminated

Check the value in “fc_token” to see how the request went

Invoking LE Services, 5

❒ FORTRAN

Although FORTRAN programs can't directly call LE services, there are two services provided that can act as intermediaries between a FORTRAN program and most of the LE callable services

✗ AFHCEEF - Call an LE service, providing a feedback field

Example

```
EXTERNAL CEEMSG
.
.
.
CALL  AFHCEEF (CEEMSG, INTOKEN, DEST2, FCTOKEN)
```

✗ AFHCEEN - Call an LE service, omitting the feedback field

Example

```
EXTERNAL CEEMSG
.
.
.
CALL  AFHCEEN (CEEMSG, INTOKEN, DEST2)
```

Arguments and Parameters

❒ **When your program issues a CALL to an LE service, the CALL must include one or more <u>arguments</u> passed to the service**

From the perspective of the service, the passed values are called <u>parameters</u>

The data are the same, it's simply the perspective:

✗ From the perspective of the calling program, the values are called arguments

✗ From the perspective of the called program, the values are called parameters

✗ You pass arguments, you receive parameters

❒ **This is the traditional nomenclature used by PL/I**

Most other languages have used the term “parameter” from both perspectives

It is not a big deal: this is just to provide you with a clue to these terms as used in the IBM documentation

Tokens

- A token is a string of data that represents an object or a situation (condition)

- In Language Environment, most services return a condition token that indicates how the service request went

- Some LE services also take a condition token as input

  For example, the CEEMSG service issues the message that relates to a condition token

  The CEEDCOD service takes a condition token returned from some previous service and breaks out its component parts

Condition Tokens and Feedback Codes

❒ **The last argument passed to a call of an LE service (next-to-last argument for LE math services) is the name of a variable into which the service places a condition token**

A 12-byte area of memory used for holding feedback information

✗ So this last argument is often called the fc argument, for feedback code

❒ **Condition tokens are classified as Case 1 or Case 2, and have the following layouts:**

Case 1

Severity	Msg_no	Flags	Facility ID	ISI
2 bytes	2 bytes	1 byte	3 bytes	4 bytes

Case 2

Class	Cause	Flags	Facility ID	ISI
2 bytes	2 bytes	1 byte	3 bytes	4 bytes

❒ **Flag bits have the following meaning:**

2 bits for case ('01' or '10'); identify condition token type

3 bits for severity ('000' through '100' (0-4))

3 bits for control ('001' -> facility ID assigned by IBM;
'000' -> facility ID assigned by user)

More on LE Tokens

❒ **Generally speaking, LE generates Case 1 condition tokens while application programs can generate Case 1 or Case 2 condition tokens**

You might use Case 2 tokens, for example, if you didn't want to get wrapped up in the message numbering / naming scheme set up for Case 1 tokens and most messaging services

❒ **The ISI (Instance Specific Information) field contains zero (no further information) or an ISI block number**

LE maintains a number of ISI blocks for each thread

Conditions that need to specify more information grab an ISI block and put the block number in the ISI field of the condition token

When a condition is handled, its ISI blocks may be reused

In some instances the data found in the ISI contains a "q_data_token" (“q_data” for qualifying data)

✗ This is the address of a list of addresses of more data (the details are beyond the scope of this course)

Tokens and Severity Codes

- **One of the pieces of information in a condition token is the severity of the condition**

  A severity code of 0 means the request was satisfied with no unusual results; in this case the entire token is returned as binary zeros

  A severity code of 1 means a Warning level: things are probably OK (example: you issued a message that was too long, so it was truncated)

  A severity code of 2 means an Error was detected, correction was attempted, and the service completed, probably correctly (example: you requested a date to be formatted and the current country code is invalid; the default format was used)

  A severity code of 3 means a Severe error was detected, the service could not complete, there may be negative side effects (example: you asked for a date to be formatted, but the date was not in the acceptable range)

  A severity code of 4 means a Critical error was detected, the service was incomplete, and a condition is signaled (example: storage management services control information was found to be damaged)

Reacting to Conditions

- **When you check a condition token or detect a user-defined condition in your program, you can take any of several actions**

  Continue running

  Signal a condition to be handled by the condition manager (and thus, perhaps, be routed to a user-provided condition handling routine)

  Build a message and save it in memory or send it to a destination immediately

  Handle anticipated situations as appropriate to the application

- **If a condition is signaled, the token representing the condition is presented to all condition handlers available to the thread**

Including the FC Token Argument

❒ **If you call an LE service and include the fc token argument, that LE service might detect an error condition**

If the severity code of the condition is less than 4, control is returned to your program with the fc parameter containing the severity code value in the condition token

✗ Your program can now check the code or not

➢ Some services need not be checked because they will either work correctly or you'll never get control back to your program: the condition manager will be signaled directly

If the severity code is 4, control passes directly to the condition manager for processing using standard LE condition handling procedures

Omitting the FC Token Argument

❒ **If you call an LE service and omit the fc token argument ...**

If the resulting severity code is 0, resume execution at the next sequential instruction

If the severity code from the service is greater than 0, the condition represented by the token is automatically signaled

Thus your program can omit fc token arguments, and the checking the value of these tokens, and have LE automatically signal conditions of severity greater than 0

- ✗ Control will pass to the thread condition manager, which will pass the condition to all registered condition handlers:
  - ➢ If some condition handler handles the condition, resume execution at the point specified (details later)
  - ➢ If the condition is unhandled, if the severity code is 1, resume execution at the next sequential instruction
  - ➢ If the condition is unhandled, if the severity code is greater than 1, add 1 to an error counter
    - ➤ If the ERRCOUNT run-time parameter is 0, or if ERRCOUNT is greater than 0 and the error counter is less than ERRCOUNT, re-drive the condition manager stack one more time, indicating a condition of termination imminent (if condition remains unhandled, terminate the thread)
    - ➤ If the ERRCOUNT run-time parameter is greater than 0, and if the error counter exceeds this value, terminate the thread immediately, with Abend code 4091 and reason code 11

Omitting the FC Token Argument, continued

❒ **All languages may omit the fc argument:**

Assembler: do not code the argument; code 'VL' after argument list

C/C++: code the fc argument as NULL or OMIT_FC

COBOL: code the fc argument as OMITTED

✗ Note: if you specify the fc argument as OMITTED and the service is successful, the RETURN-CODE special register is set to zero; if the service is un-successful, RETURN-CODE is not modified; if you supply an fc argument, RETURN-CODE is always set to 0, regardless of the success or failure of the service

PL/I: code the fc argument as *

FORTRAN: invoke the service using a call to AFHCEEN

❒ **We will always supply this argument in our examples, but it is an option to keep in mind**

❒ **Details of condition handling routines and their possibilities are discussed later in this course**

LE Program Management

❒ **Language Environment manages programs and resources using a model that recognizes**

Thread - the execution of an application's program(s); think "task" in traditional z/OS terms

Enclave - programs and storage used by one or more related threads; an enclave consists of: a single main program, any number of sub-programs (subroutines), and storage shared among the programs; think "run-unit"

Process - one or more related enclaves and their shared resources: a message file and the runtime library (for batch, think: a logical chunk of an address space containing related programs, data, and control blocks; for online programs, think: transaction)

❒ **When you run an LE main program (LE-conforming Assembler or LE-conforming high level language compiler), LE initializes the run-time environment (process) by initializing an enclave and an initial thread**

Enclave initialization acquires an initial heap storage and establishes the starting values of attributes such as the country and language settings and the century window

Thread initialization acquires a stack, enables a condition manager, and launches the main program

❒ **You can modify initialization by running a user exit**

LE Program Management, continued

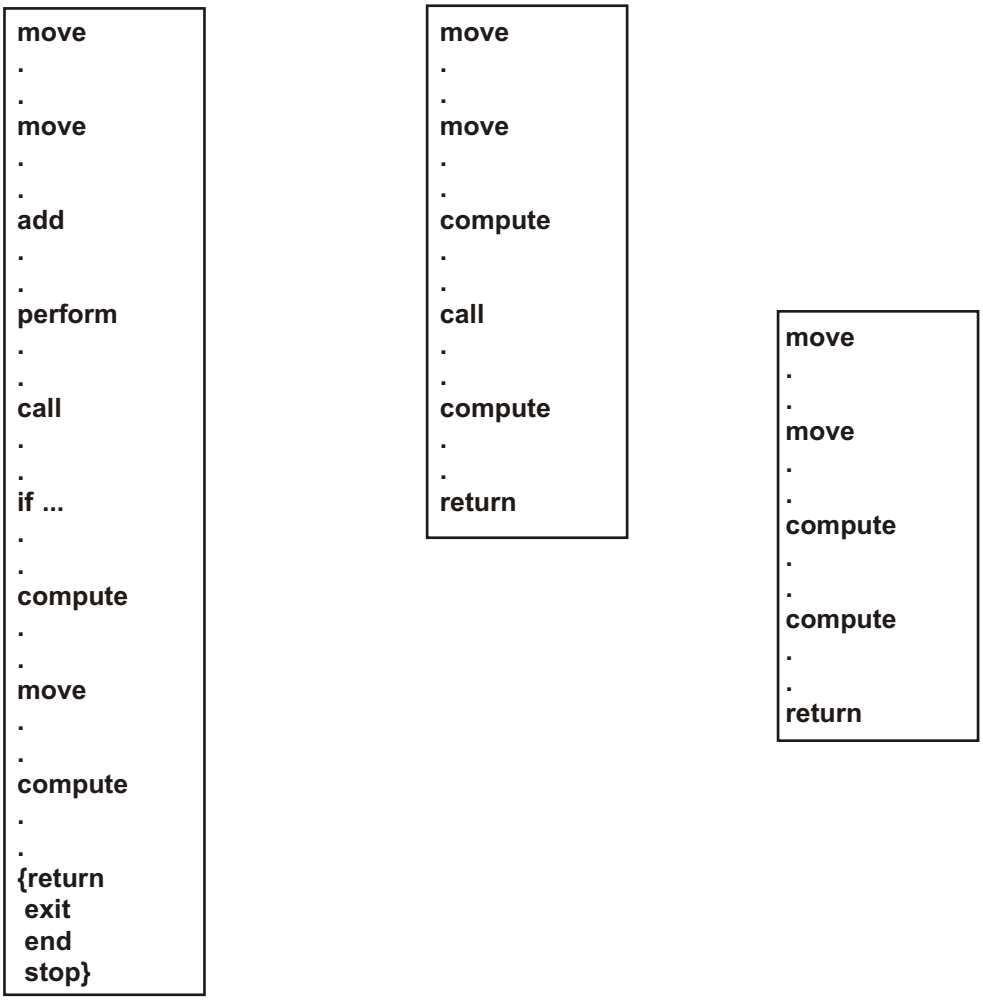
- Let's examine this program management model a little more closely

- Start with the enclave: this is really the most familiar concept for most programmers:

  A mainline and the subroutines it calls (including subroutines called by subroutines, etc.)

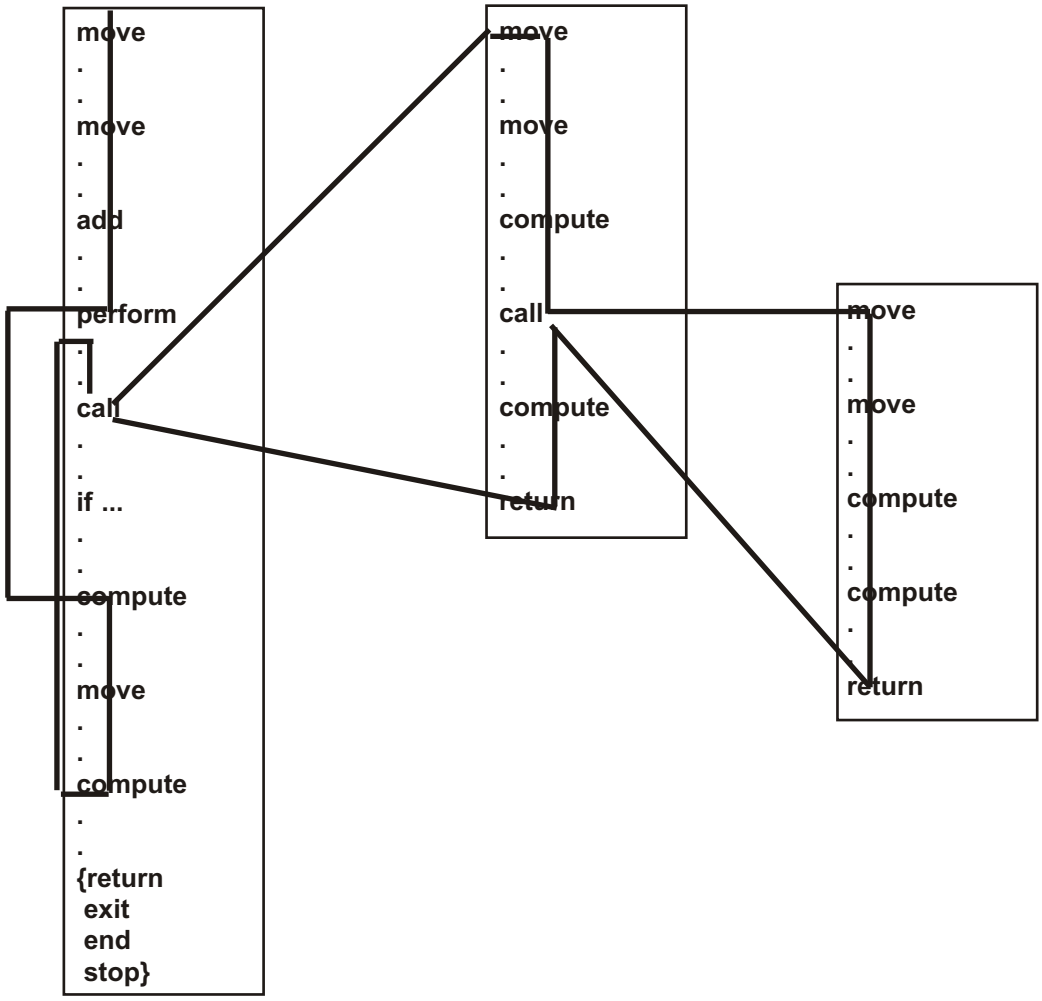
  The subroutines may be called statically or dynamically

- An enclave



LE Program Management, continued

❒ **Now, as the program executes, if we could trace its progress we might see a line of execution something like this:**



❒ **This line of instruction execution is called a thread**

Note that although there are three programs here, there is a single thread

LE Program Management, continued

❒ **Multiple Threads**

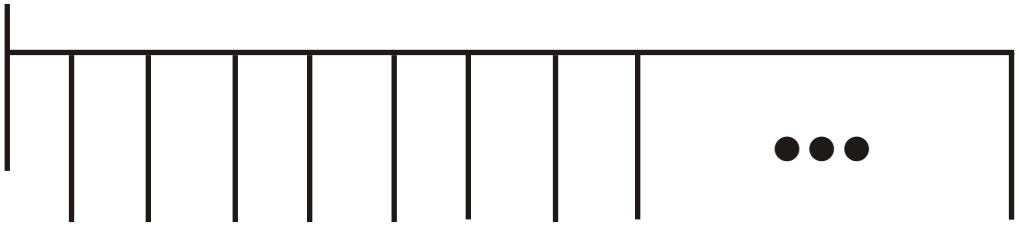
If a program wanted to start another, independent, program to do some work on its behalf, we would have a multi-threaded enclave

For example, a program might be written to analyze a customer data base for logical consistency in areas where the data base definition can't

For each row in the customer table, it would be nice to start a thread to follow all the cross-table connections for that customer

The initial thread could just be responsible for “kicking off” each sub-thread to work independently and then go on to start the next one

A schematic might look like this:



❒ **Here we see a representation of an enclave with multiple threads, each independent of the other**

❒ **Threads are created and dispatched by the kernel**

Each thread has its own copy of the condition manager and its own set of thread-level resources

LE Program Management, continued

❒ Multiple Enclaves

A thread in an enclave can start up a whole separate enclave within a process

- ✗ Since an enclave may only contain a single 'main', the introduction of a new main program requires the initialization of a new enclave, either in the current process or in a new process

For example, a program might be passed a set of parameters that describe

- ✗ A data base to analyze
- ✗ A set of relationships to analyze
- ✗ Where to pass or record the analysis

The program could then kick off a separate enclave for each such request

Each such enclave could, in turn, be multi-threaded

- ✗ Then we would have a multi-enclaved, multi-threaded process

LE Program Management, continued

❒ **Multiple Processes**

A thread can create a whole new process

✗ **A group of processes is managed by a hidden construct called a region or application (these words are not used in their traditional sense here)**

❒ **When any process is started, it goes through process initialization, enclave initialization, and initial thread creation**

And things progress from there

❒ **So we could end up with an application containing multiple processes, each of which contains multiple enclaves, with each enclave's main program (and related subroutines) running multi-threaded**

❒ **For current LE versions, only C programs can initiate multi-threading**

And these programs have to be POSIX-conforming, that is UNIX programs ...

z/OS UNIX, and POSIX

- **z/OS provides optional features that collectively are called z/OS UNIX System Services (or simply "z/OS UNIX")**

- **z/OS UNIX evolved based originally on the POSIX interface**

  Portable Operating System Interface - defined by IEEE (Institute of Electronic and Electrical Engineers, Inc.)

  Defines a minimum set of standards required to support applications across various flavors of UNIX

  At one time, z/OS UNIX (or maybe one of its predeccessor versions) was fully UNIX-branded (not clear about current POSIX status or UNIX-branding) - point being: this is very similar to classic UNIX (some purists deny this, but the differences don't concern us here)

- **Effectively, the presence of z/OS UNIX provides the functionality of UNIX under TSO and in batch and also as connected by rlogin or telnet**

  UNIX-conforming applications can mostly port between UNIX systems, including z/OS UNIX

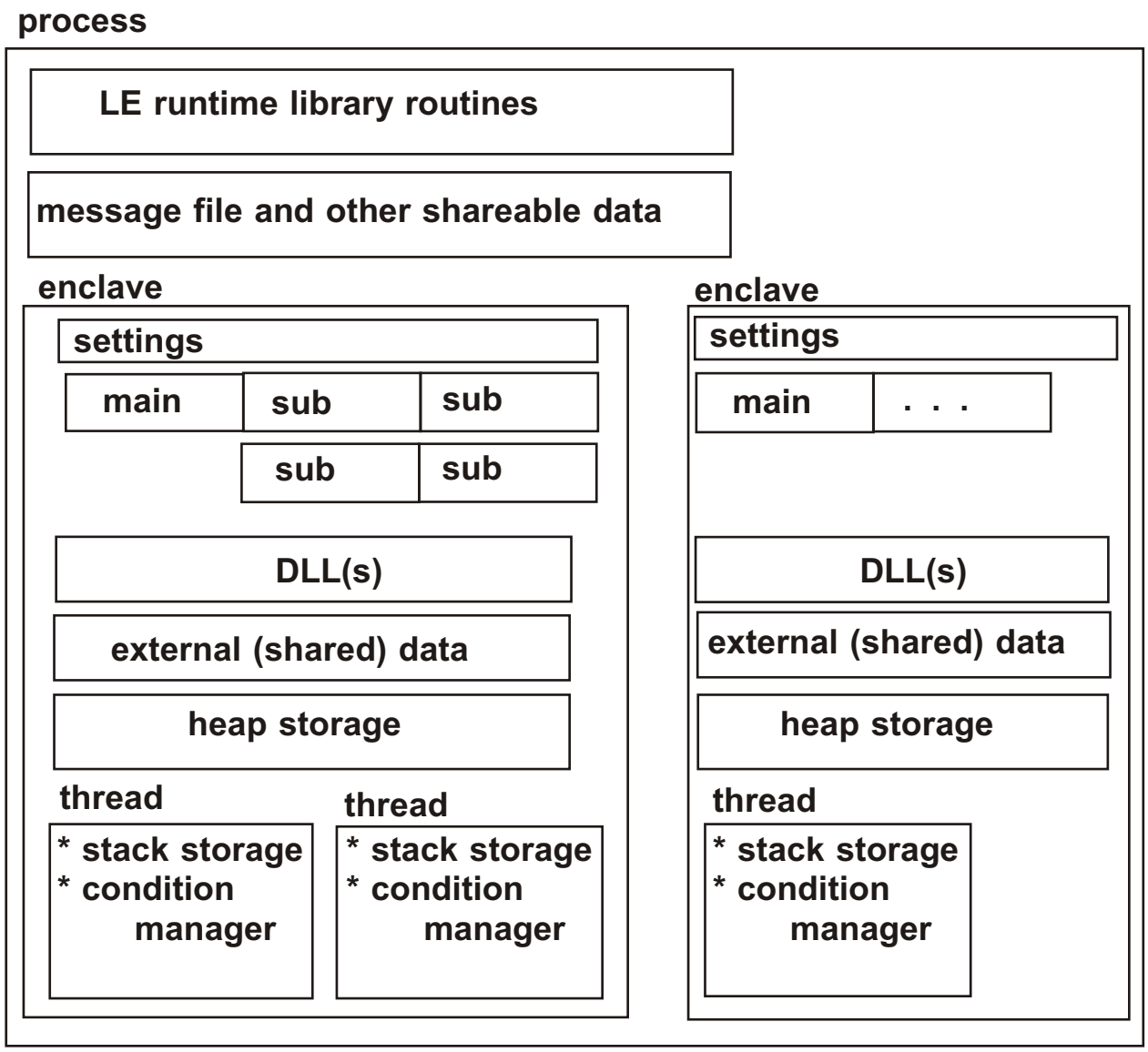
- **z/OS UNIX support requires Language Environment**

z/OS UNIX and POSIX, continued

- C applications may be POSIX-conforming or not
  - C POSIX-conforming applications may also use all LE functions
  - C POSIX-conforming applications can perform pthread ("POSIX-thread") thread management functions
  - C applications that are not POSIX-conforming cannot perform these thread management functions
- POSIX is explicitly not supported under CICS or CMS
- PL/I and COBOL programs compiled using the LE compilers can run under z/OS UNIX
- LE-conforming Assembler programs can run under z/OS UNIX
- In this course, we only mention UNIX / POSIX tangentially, where required for completeness or accuracy
  - Full discussion of z/OS UNIX takes place in other courses
- When z/OS is IPL'ed, z/OS UNIX is also always started
  - If you do not use it, you must still allow it to initialize in a minimal mode: there are parts of z/OS that expect UNIX to be available

LE Program Management, Second Pass

- Aside from z/OS UNIX, then, we can visualize some of the interrelationships this way:



- Notes

  This represents the full model, which is not all implemented in the current version of LE

  A thread can create another thread in the current enclave, another enclave in the current process, or another process in the current application

LE Program Management, Second Pass, p.2

❒ **To recapitulate, LE manages programs and resources using this conceptual model**

Process

Consists of: one or more enclaves, a message file, and the LE runtime library routines

✗ There are no LE-supplied services for creating multiple enclaves in a process in the current version of LE, but some CICS processes and some Assembler processes can create multiple enclaves in a process (discussed later)

✗ In the model, a thread in a process can create other processes, although processes are independent of one another (no hierarchical relationships)

❒ **The resources managed at the process level, include**

Message file

The Language Environmen run-time library

LE Program Management, Second Pass, p.3

❒ **C programs can create a new process using** *fork()* **or** *spawn()*

The new process is in the same address space or a new address space, depending on various environment variable settings

✗ Perform process initialization

✗ Perform enclave initialization

✗ Perform thread initialization

❒ **Running with POSIX(ON), a C program can create a new thread in the current enclave using** *pthread_* **services**

When a thread runs, it runs under a TCB (Task Control Block)

There can be more threads than TCBs (tasks) if the threads are asynchronous: when a TCB finishes running one thread it can run another thread; a thread waits until there is an available TCB

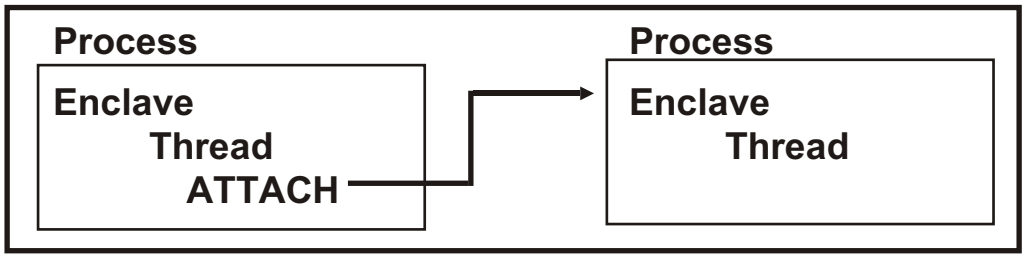
✗ This is the dispatching the z/OS UNIX kernel does

❒ **Note that POSIX(ON) only affects** *pthread_* **services, signaling services, and in some situations, the order of search for DLLs**

LE Program Management, Second Pass, p.4

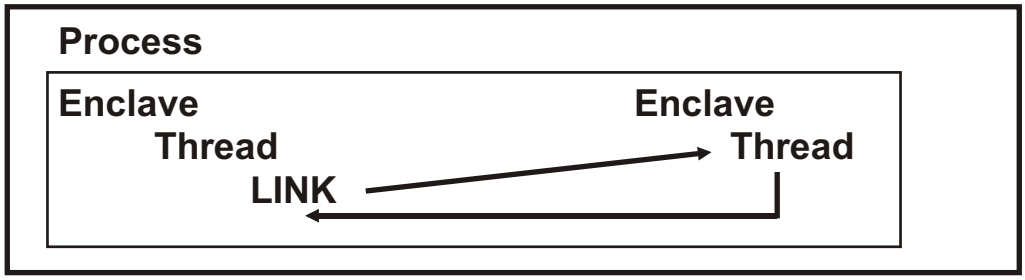
❒ **Aside from C programs, for applications only a single address space is used:**

Assembler ATTACH service creates multiple processes

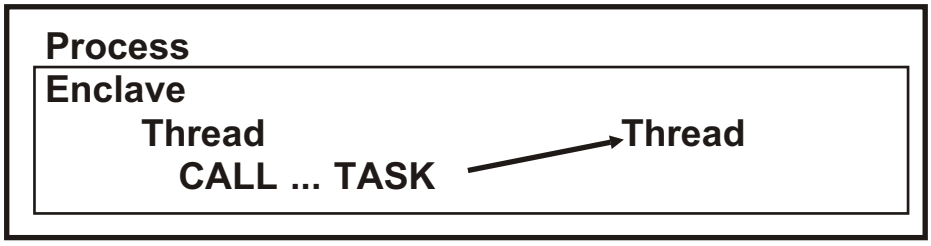


✗ Actually creates new region, new process, new enclave, and new thread; the attached program must be an LE MAIN program or a non-LE program that takes no arguments

Assembler LINK and EXEC CICS LINK services create multiple enclaves (they are not multi-tasked: serial execution):



PL/I multi-tasking must run POSIX(OFF) but uses POSIX services behind the scenes to create a new thread, no new enclave:



❒ **Note: Enterprise PL/I does not support multi-tasking but it does support mulit-threading, when running with POSIX(ON)**

LE Program Management, Second Pass, p.5

❒ **The LE program management conceptual model, continued**

Enclave

One MAIN routine and zero or more subroutines, including HEAP storage, external files, DLLs, and shareable 'EXTERNAL' data

- ✗ the first invoked routine in an enclave is the main routine, all subsequent routines are subroutines
- ✗ file sharing is not managed by LE (except for the message file managed at the process level)
- ✗ file sharing across languages is not currently supported in LE
- ✗ heap storage is shared among all routines in an enclave

❒ **Resources managed at the enclave level include**

Default date and time formats

Current country setting

National language setting

Currency symbol, decimal separator, and thousands separator

Setting of the century window

The user area fields

Heap storage

❒ **If any thread sets / changes these values, they are changed for all threads in the enclave**

LE Program Management, Second Pass, p.6

❒ **The LE program management conceptual model, concluded**

Thread

The basic line of instruction execution; each thread has its own storage stack and condition manager

- ✗ A thread is created during enclave initialization, with its own instruction counter, registers, stack and condition handling mechanism
- ✗ Each thread shares all resources of an enclave; it does not own its own storage but can address all storage in an enclave

❒ **Resources managed at the thread level include**

Stack storage

Condition manager

Registers

Parameters passed on invocation - initial thread only

Message inserts

Day of week, date and time values

Platform id

Random number generator seed

❒ **If a thread changes any of these, it does not affect the values for these on any other thread**

❒ **A thread is like an enhanced z/OS task**

Multiple Processes, Enclaves, Threads

❒ **Note that the full ability of multi-threading applications is not available to all languages in the current version of LE**

❒ **But there is a variety of language-specific techniques:**

Multi-tasking in Assembler, using ATTACH

✗ Create multiple new processes in the address space

Multi-processing in C using *exec*, *fork*, **or** *spawn*

✗ May create a new address space

Multi-threading in C using *pthread_create* **and related services**

✗ Requires POSIX(ON), creates new threads in current enclave

Multi-tasking in PL/I using CALL specifying one or more tasking options (releases prior to Enterprise PL/I)

✗ Creates TCB-based tasks; requires POSIX(OFF)

Multi-threading in Enterprise PL/I using ATTACH

✗ Creates POSIX threads; requires POSX(ON)

❒ **There are some special cases where you can create what is called a nested enclave (discussed later), but this is not true multi-enclave or multi-thread behavior**

❒ **There are no plans to provide multi-threading services other than through the C and Enterprise PL/I interfaces (that is, no CEE... services)**

LE and 64-bit Processes

❒ **z/OS is IBM's premier 64-bit operating system for zArchitecture machines**

❒ **To take advantage of 64-bit addressing, there are currently two options:**

Assembler code can run in 64-bit addressing mode (AMODE) whether or not it is LE-conforming Assembler

C /C++ can run under either 64-bit LE or non-64-bit LE

✗ That is, there is a 64-bit version of LE and it currently supports only C, C++, and Assembler

✗ Currently there is no communication allowed between 64-bit LE applications and 31-bit LE applications

❒ **We shall only reference 64-bit LE, and related topics, in passing**

Introduction to XPLINK

- C, C++, and related languages, were originally designed to run on machines that had a storage stack automatically implemented in hardware

  This automatically stores environment information and parameters for fast entry and exit to functions

- IBM mainframes do not include such a hardware assist, so each subroutine and function is responsible for saving and restoring register contents

  And for providing a storage area for called routines to use for their saving and restoring logic, and so on

- For languages such as C/C++ which tend to have a large number of short functions, this means the overhead of initialization and termination for each function tend to take up a large percent of the time spent in a function

- To provide more efficient linkages in this environment, IBM introduced XPLINK ("eXtra Performance LINK) conventions that use non-traditional module / function linkages

  Currently XPLINK is only used in certain environments, and using it requires special versions of many routines

- All 64-bit LE support uses XPLINK

  Although 64-bit Assembler code does not need to use XPLINK

  In this course, we only reference XPLINK incidentally

Section Preview

- **LE Message Services and Running LE Programs**

  Language Environment Message Handling Services

  The CEEMOUT Service

  Compiling and Binding LE Programs

  Running LE Programs

  A First Encounter With Language Environment (Machine Exercise)

Language Environment Message Handling Services

- **The first of the Language Environment services we'll look at explicitly deal with message handling**

  This is useful in debugging and, later, writing condition handling routines

  This is also a good introduction to coding Language Environment callable services requests, and provides an opportunity to compile, link, and run an LE conforming program

- **Messages dispatched by LE go to the LE MSGFILE data set**

  This data set has a DDname of SYSOUT, by default; the DDname may be changed with a run-time option

  Messages dispatched by LE while running CICS go to the transient data queue CESE

LE Message Handling Services, Introduction

- **Non-LE services still work, such as**

  DISPLAY (COBOL)

  PUT (PL/I)

  WTO ... ROUTCDE=(11) (Assembler)

  printf **(C/C++)**

- **COBOL programs compiled using older compilers (e.g.: OS/VS COBOL) routines, running as subroutines to an LE program, may still use TRACE and EXHIBIT statements, and this output is directed to the MSGFILE destination**

- **The default MSGFILE DDname is SYSOUT, which is also the default DDname for COBOL DISPLAY statements**

- **If a COBOL program is compiled with OUTDD different from SYSOUT, DISPLAY output goes to the DDname specified in OUTDD**

- **User-directed messages in PL/I programs are sent to the PL/I SYSPRINT STREAM PRINT file; to send this to MSGFILE, you need to set MSGFILE to be SYSPRINT (discussed with run-time options later)**

LE Message Handling Services, continued

- **C standard stream outputs (LE message requests, C library messages, and error messages (using** perror() **or** fprintf() **) are normally routed to the MSGFILE DD statement, but** printf **output goes to** stdout

  stdout **goes to one of these DD statements (C tries to open them, in this order; if not found, C generates a DD name of SYS0000***n* **and uses that): SYSPRINT, SYSTERM, SYSERR**

  To direct printf **output to the MSGFILE use the C** freopen **function or invoke redirection services at run time (**1>&2**)**

- **FORTRAN WRITE statements to a unit identifier with the same value as the FORTRAN error message unit, and output from FORTRAN dump services calls are directed to the LE message file**

- **Run time messages generated by the COBOL, PL/I, C, C++, and FORTRAN language support routines all go to the MSGFILE destination**

More on Message Handling Services in LE

❒ **Language Environment provides a common set of message handling services usable by all programs in an application, regardless of the language each program was coded in**

Within a process, all messages dispatched by LE messaging routines go to the same MSGFILE data set

Thus messages are found in a single place, and in the order they were issued

❒ **The text of messages can be**

<u>Text that resides in the program</u> as a "varying length character string" (also called a "halfword prefixed string"):

- ✗ 2-byte length prefix, indicating the length of the following text
- ✗ The text as character string

<u>Text in one or more loadable text files</u> shared by all programs in the application

- ✗ Thus ensuring consistency in wording, message codes, severity codes, and formatting
- ✗ Messages in a loadable text file can be
  - ➢ Static text
  - ➢ Text with room for one or more inserts - values that will be determined at run time

❒ **We examine the text-in-the-program ability first**

The CEEMOUT Service

- **The CEEMOUT LE service dispatches a message to the message file data set**

Syntax - COBOL

```
Call   'CEEMOUT' using   msg,   dest,   fc
```

Syntax - PL/I

```
Call   ceemout (msg, dest, fc);
```

Syntax - C/C++

```
CEEMOUT (&msg,&dest,&fc);
```

Syntax - Assembler

```
Call   ceemout,(msg,dest,fc)
```

- **Later, we'll just list the arguments and assume you can convert them into the correct format (note: generally speaking, case is not significant except for C/C++; we'll point out when it matters)**

The CEEMOUT Service, 2

- **The arguments for the CEEMOUT service are:**

  msg

  denotes a halfword prefixed string containing the text of the message being sent

  dest

  denotes the destination, as a binary fullword integer

  currently only a value of '2' is allowed, which says “route the message to the Language Environment message file”

  fc

  denotes a twelve byte area for the service to return a feedback code condition token; the fc argument

- **The service sends the message text to the message file and then fills in the fc argument**

  The fc argument is seldom checked after this service, and in most cases it may safely be omitted

CEEMOUT - Examples

❒ Some code fragments, primarily showing how to describe arguments

COBOL

```
Working-storage section.
01  Msg.
    05  msg-length   pic s9(4) binary value 10.
    05  msg-text     pic x(10)
                         value 'Here I am!'.
01  Dest              pic s9(9) binary value 2.
01  Fc                pic x(12) value low-values.
. . .
Procedure division
. . .
     call 'ceemout' using msg, dest, fc
```

PL/I

```
Declare CEEMOUT entry options(asm) external;
Declare msg char(10) varying init('Here I am!');
Declare dest fixed bin(31,0) init(2);
Declare fc char(12) init(low(12));
. . .
call ceemout (msg, dest, fc);
```

CEEMOUT - Examples, p.2.

C/C++

```
#include <leawi.h>.
. . .
int main(void)
{
  _VSTRING msg;
  _INT4 dest;
  _FEEDBACK fc;
strcpy(msg.string,"Here I am!");
msg.length = strlen(msg.string);
dest = 2;
. . .
CEEMOUT(&msg,&dest,&fc);
}
```

Assembler

```
. . .
          call  ceemout,(msg,dest,fc)
. . .
          ds    0F
msg       ds    0CL12
          dc    H'10'
          dc    CL10'Here I am!'
dest      dc    F'2'
fc        dc    3F'0'
```

Compiling and Binding LE Programs

❒ **Compiling programs written to use LE services is essentially the same as compiling and linking any other program**

Need to make sure you invoke the appropriate compiler (check program name and STEPLIB statement in compile step; make sure LE run-time library available as well)

❒ **At program bind time, you need to make sure certain LE data sets are available in your SYSLIB concatenation, depending on your environment and usage of services; the names below are usually the low level qualifier**

SCEELKED - all non-XPLINK applications

SCEELKEX - most C, C++ non-XPLINK applications

SCEECPP - C++ applications containing non-XPLINK programs

SCEEOBJ - z/OS UNIX, non-XPLINK apps

CSSLIB - apps calling UNIX callable services

SCEEBIND - XPLINK apps; migrate to using SCEEBND2 instead

SCEELIB - XPLINK side decks

✗ Note that the first four files are only for non-XPLINK applications; for XPLINK applications you should use SCEEBIND (or SCEEBND2) and SCEELIB instead

Running LE Programs

❒ **You run LE programs just like any other executable program**

❒ **However, you need to validate the following JCL statements**

STEPLIB - include library where program is as well as language-supplied dynamic subroutine libraries, application dynamic subroutine libraries, and the Language Environment run-time libraries SCEERUN and (for C, C++ and some COBOL apps) SCEERUN2 (unless these modules are in the Link Pack Area or the linklist)

Message file DD statement (usually DD name SYSOUT)

CEEDUMP - to hold any LE dump services output

✗ Although LE will dynamically allocate this statement if needed

CEESNAP - to hold dumps generated when using PLIDUMP

SYSUDUMP - to hold any z/OS generated dump output

Language specific DD statements, if their use is implied in your program (for example: SYSIN, SYSOUT for COBOL; SYSIN, SYSPRINT for PL/I, etc.)

Application specific DD statements, as before

Specify run-time parameters on the EXEC statement correctly

✗ More on this later

Computer Exercise 1: A First Encounter With Language Environment

This machine exercise is designed to provide setup for all the remaining class exercises.

First, you need to run M512STRT, a supplied REXX exec that will prompt you for the high level qualifier (HLQ) you want to use for your data set names; the exec uses a default of your TSO id, and that is usually fine. Then the exec creates data sets and copies members you will need.

The allocated data sets:

<hlq>.TR.CNTL	for all your JCL
<hlq>.TR.COBOL	for all COBOL source code
<hlq>.TR.EXEC	for holding the LE REXX exec CEEBLDTX
<hlq>.TR.SOURCE	for all other source code
<hlq>.TR.PDSE	for executable programs (program objects)

To run the M512STRT exec, from ISPF option 6, on the command line, enter:

```
===> ex     '________.train.library(m512strt)'       exec
```

*** more ***

Computer Exercise: A First Encounter With Language Environment, p.2.

We have provided a working program for you to enhance by using Language Environment services. The program name is based on the language you will be using for the labs: LEASM, LEC, LECOBOL, or LEPLI. A listing of these programs is found in Appendix A to this handout.

COBOL programmers note: depending on the version of your compiler, you may have to change the process statement TEST parameter from test(sym,none) to just test.

For this exercise, you are to modify your program to add code necesary to invoke the CEEMOUT service to display the message:

Got to main program _____

where _____ is your program's name (LExxxx).

You can find the places where you need to add code by doing a find on 'exercise 1' (for COBOL and Assembler) or on 'exercise number 1' (for C and PL/I).

After the job is run, your message should appear on the message file data set in your GO step (SYSOUT DD statement).

To Assemble or compile, bind, and run this program, use the provided JCL. Use member TESTASM, TESTC, TESTCOB, or TESTPLI, based on the language you are using, for all your programming tests in this course.

***** more *****

Computer Exercise: A First Encounter With Language Environment, p.3.

Exercise stretch:

We have supplied an Assembler subroutine (already assembled and linked) for use in displaying token data, if you wish to explore how LE is setting tokens. The routine name is FC2HEX; call it with two parameters: the name of the data to examine, and a 26-byte field for the hex representation of 12 bytes to be placed. Your JCL is already set up to automatically include this subroutine, if you invoke it, at link edit time or run time. Sample code for invoking FC2HEX is included in your program, currently commented out.

(Note: FC2HEX may be called from programs written in any language.)