The Trainer's Friend, Inc.

6790 E. Cedar Ave., Suite 201
Denver, Colorado 80224
U.S.A.

Telephone:	(303) 393-8716
E-mail:	trainers@trainersfriend.com
Internet:	www.trainersfriend.com

z/OS, Language Environment, and UNIX

How They Work Together

Preface

This document came about as a result of writing my first course for UNIX on the IBM mainframe. I "grew up", professionally speaking, on the mainframe. In the years I worked for IBM, UNIX was considered "the enemy". Now, you can run UNIX applications on the mainframe.

How did this happen?

More than anything else, I believe it was the emergence of the Internet as a force in corporate computing. The Internet is important, even essential, for businesses today: It provides a friendly way to reach customers and prospects, it can, potentially, be a less expensive way to handle many customer support issues, and it can give a company an advantage over its competition.

Most Internet work is done on UNIX boxes. If IBM wanted to be a player in this field, it had to support UNIX somehow. But the mainframe is still important. A large percentage of the "bread and butter" applications of the world's largest organizations are running on mainframes, taking advantage of the mainframe's legendary strengths: speed, I/O bandwidth, security, reliability.

So IBM developed a way to support UNIX applications, such as Internet work, at the same time you can run mission critical applications, all on the same mainframe box.

Still, I was reluctant to follow this development. But now, customer interest and demands have caused me to jump into creating UNIX-on-the-mainframe courses.

The UNIX way of looking at computing is, in many areas, quite different from the traditional IBM mainframe way of looking at things. So I had to find a way to re-examine computing with a fresh eye. At the same time, the z/OS version of the product is now current and I needed to put this development in some perspective.

So here are my thoughts and understandings of how things work, from an applications programmer perspective. I start out with a refresh of MVS, z/OS terms and functionality, look at UNIX from a generic sense, and then explore how IBM maps the mainframe technology to the UNIX paradigm. It turns out to be pretty amazing.

z/OS

Acknowledgements and Permissions

Although this document is copyrighted, I hereby grant permission for unlimited copying, excerpting, or abstracting, as long as correct attributions are made.

All the source materials for this publication are freely available from IBM and other sources on the Internet (the World Wide Web, usenet newsgroups and listservs). Sometimes it is a little obscure (both in the sense of where to find what you need and in the sense of understanding what is being said). Sometimes you have to ask; sometimes you have to write some code and do some testing. All I can lay claim to is selection, organization, and presentation.

While a great many people have helped me by looking at this document and suggesting changes, any errors contained here are strictly my own, due to errors in understanding, interpretation, or typing.

I hope this publication helps others in their understanding in how things work on z/OS and that they can productively apply the concepts presented here in applications design / debugging / and maintenance.

I particularly wish to acknowledge the help I received from these people:

B.V. Bixoft
Abe Kornelis

Chicago Soft
Mike Shaw

Cole Software
David Cole

IBM
Greg Dyck
Rich Fabish
Walt Ferrell
Peter Hunkeler
Bill Schoen

The Trainer's Friend, Inc.
Hunter Cobb

TWA
Matthew Wheaton

z/OS

❒ **Part 1: Background**

- ♦ **Introduction**
- ♦ **Numbers and Numeric Terms**
- ♦ **The Road to z/OS**
- ♦ **The Story of You**
- ♦ **Address Spaces**
- ♦ **SubSystems**

Introduction

- ❐ **When you start out to learn a foreign language, especially for the first time, you end up learning a lot about your native language**
  - ♦ **Examining syntax and grammar of a familiar language allows you to abstract the concepts and then apply them to the new, unfamiliar language**

- ❐ **The same might be said about learning a new operating system**
  - ♦ **Learning a new operating system is eased by first getting a deeper understanding of a familiar operating system**

- ❐ **Our ultimate goal, here, is to learn how to use IBM's UNIX System Services on the mainframe - almost like a new operating system**
  - ♦ **We focus on using UNIX System Services under z/OS, z/OS being IBM's premier 64-bit operating system on the mainframe, capable of working with 64-bit addresses**

- ❐ **We begin our journey by re-examining how things work in z/OS**
  - ♦ **Many applications programmers who work on MVS, OS/390, and z/OS systems may find this new, not a re-examination**
    - ✗ But at least the starting point is relatively familiar, so we hope to explore z/OS then move on to the newer, stranger world

Introduction, continued

❒ **Our exploration will be trying to walk a tricky path**

- ♦ **Lay enough foundation for our ultimate goal**
  - ✗ While not bogging down in details that are not relevant, even if they are interesting

❒ **So, some caveats, especially for technical people who already have a deeper understanding of z/OS (or its predecessors)**

- ♦ **The intended audience is experienced MVS and OS/390 applications programmers**

- ♦ **We are simplifying the story by omitting details of system-level interfaces, such as how does IPL/NIP work, what about scheduling using SRBs, the impact of system exits, and that level of detail**

- ♦ **But we are including high-level discussions of the existence and role of many z/OS control blocks the applications programmer will need to understand to grasp some of the subtleties of using UNIX System Services**

❒ **Note: UNIX System Services is officially not abbreviated USS (that acronym had an earlier meaning), and we will try to use the simple shorthand of "UNIX", meaning specifically "UNIX System Services on z/OS"**

- ♦ **While "USS" is in widespread casual use, it is better to use the shorthand "z/OS UNIX"**

Numbers and Numeric Terms

❒ The numbers possible with 64-bit integers and addresses are large, so we find we may need to refresh / introduce the proper numeric terms, at least as far as American English goes

- For counting ...

For this many digits	We use the term
9	units
99	tens
999	hundreds
9,999	thousands
9,999,999	millions
9,999,999,999	billions
9,999,999,999,999	trillions
9,999,999,999,999,999	quadrillions
9,999,999,999,999,999,999	quintillions

❒ The range of binary number in 64 bits is, in decimal:

- 0 - 18,446,744,073,709,551,615 if unsigned
- -9,223,372,036,854,775,808 to +9,223,372,036,854,775,807 if signed

Numbers and Numeric Terms, 2

- ❒ But, in terms of measurements (for example number of bits or bytes of memory or disk space) the language has gotten trickier

- ❒ First, we have historically used these terms

  - ♦ Kilobit, Kilobyte (Kb, KB) - 1,024 bits or bytes
  - ♦ Megabit, Megabyte (Mb, MB) - 1,048,576 bits or bytes
  - ♦ Gigabit, Gigabyte (Gb, GB) - 1,073,741,824 bits or bytes
  - ♦ Terabit, Terabyte (Tb, TB) - 1,099,511,627,776 bits or bytes
  - ♦ Petabit, Petabyte (Pb, PB) - 1,125,899,906,842,624 bits or bytes
  - ♦ Exabit, Exabyte (Eb, EB) - 1,152,921,504,606,846,976 bits or bytes

- ❒ But these numbers are based on powers of 2, while engineers have historically worked with powers of 10 and use these terms:

  - ♦ Kilobit, Kilobyte (Kb, KB) - 1,000 bits or bytes
  - ♦ Megabit, Megabyte (Mb, MB) - 1,000,000 bits or bytes
  - ♦ Gigabit, Gigabyte (Gb, GB) - 1,000,000,000 bits or bytes
  - ♦ Terabit, Terabyte (Tb, TB) - 1,000,000,000,000 bits or bytes
  - ♦ Petabit, Petabyte (Pb, PB) - 1,000,000,000,000,000 bits or bytes
  - ♦ Exabit, Exabyte (Eb, EB) - 1,000,000,000,000,000,000 bits or bytes

- ❒ These differences can cause real incompatibility problems in designs or standards, and thus manufacturing, costing, and pricing

Numbers and Numeric Terms, 3

❐ In 1998, the International Electrotechnical Commission (IEC) defined a standard set of terms and prefixes to be used to distinguish powers of two from powers of ten

❐ So these are the terms one should use when referencing numbers based on powers of two that describe quantities:

- ♦ Kibibit, Kibibyte (Kib, KiB) - 1,024 bits or bytes
- ♦ Mebibit, Mebibyte (Mib, MiB) - 1,048,576 bits or bytes
- ♦ Gibibit, Gibibyte (Gib, GiB) - 1,073,741,824 bits or bytes
- ♦ Tebibit, Tebibyte (Tib, TiB) - 1,099,511,627,776 bits or bytes
- ♦ Pebibit, Pebibyte (Pib, PiB) - 1,125,899,906,842,624 bits or bytes
- ♦ Exbibit, Exbibyte (Eib, EiB) - 1,152,921,504,606,846,976 bits or bytes

❐ The point of all this is that, for example, 65,536 bytes is 64KiB, not 65KiB, and 18,446,744,073,709,551,616 bytes is 16EiB, not 18EiB

❐ It's recommended that the second syllable ("bi") be pronounced as "bee"; the "bi" indicates "binary" - power of two

❐ It is not clear if these standards will be widely adopted or used outside of technical areas, and we may mix the new with the old while we go through a period of transition

The Road to z/OS

- **Modern IBM mainframe operating systems originated in the mid-1960's**

- **The mainframe operating systems on S/360 class machines that led to z/OS have been ...**

  - **OS/360 - Operating System/360, which had three major variations**

    - PCP - Primary Control Program; no longer around

    - MFT - Multiprogramming with a Fixed number of Tasks; no longer around

    - MVT - Multiprogramming with a Variable number of Tasks (the base for what is to become z/OS)

- **The OS/360 systems were all real-storage systems**

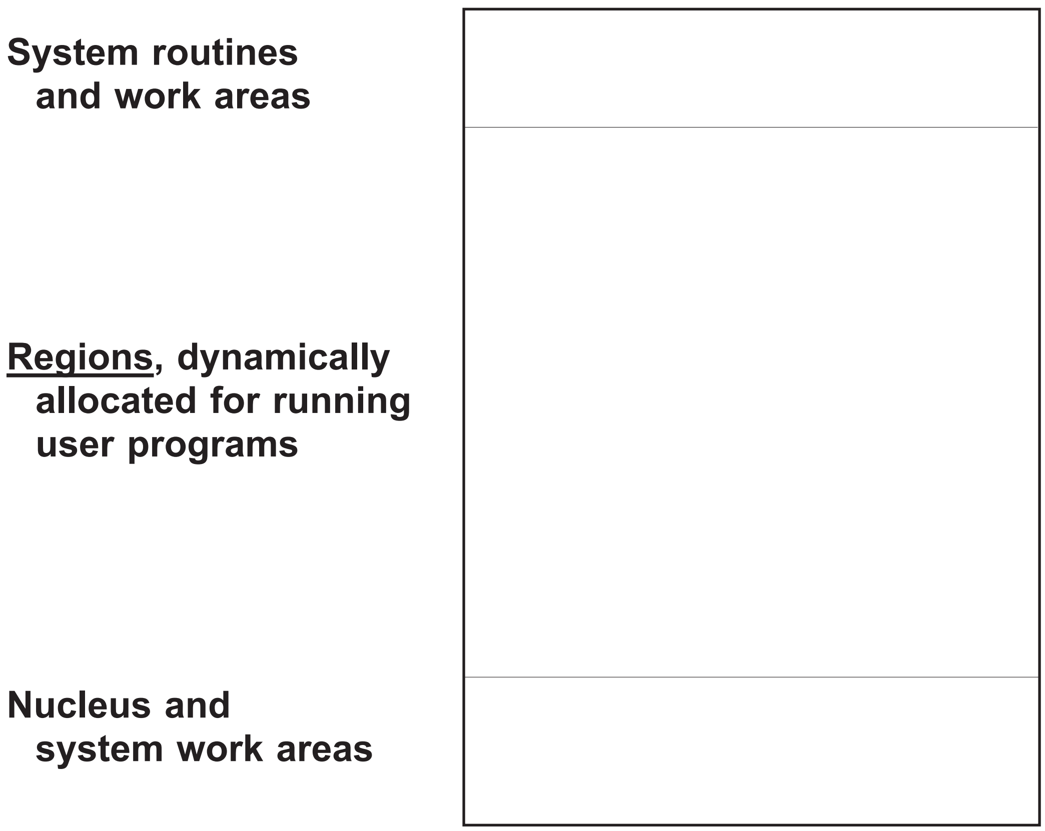
  - **No concepts of virtual storage, paging, swapping, and so on**

  - **The real storage you had was what you worked with**

  - **Mainframes running these systems often only had 24KiB of real memory; a machine with a megabyte of storage was a rarity**

The Road to z/OS, 2

❒ **MVT organized [real] storage as having (from top to bottom: that is, high address to low address):**


System routines and work areas
Regions, dynamically allocated for running user programs
Nucleus and system work areas

❒ **The maximum, theoretical size of memory was 16MiB, but few customers bought more than 4 MiB, and it wasn't uncommon in this timeframe for a mainframe to have 128KiB**

- ♦ **The memory addressing scheme used 24 bits (3 bytes) to specify a memory address, so 16,777,215 was the largest theoretical address**

The Road to z/OS, 3

❒ **In the 1970's, it became apparent there was a need to increase storage, and the first approach introduced Virtual Storage**

- ♦ **Virtual Storage uses real storage, disk as backing storage, in combination with hardware detection of address faults to present the user with the illusion of having more memory than is really present**

- ♦ **The user cannot distinguish between real storage and virtual storage**

❒ **At this time, MVT became OS/VS2**

- ♦ **OS/VS1 was the old MFT with virtual storage; you could specify the total size of virtual storage up to the 16MiB limit**

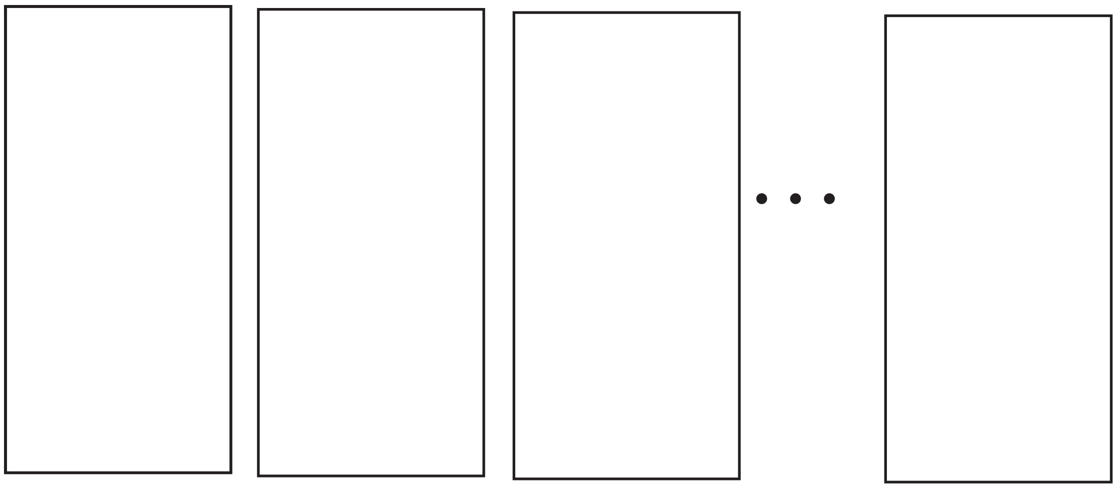
- ♦ **OS/VS2 later became called "SVS" for Single Virtual Storage**

  - ✗ SVS used a single virtual storage of the maximum size (16MiB), but the logical organization remained as it was for MVT

The Road to z/OS, 4

- ❒ **In the mid-1970's, IBM started to phase out SVS in favor of the newest operating system: MVS - Multiple Virtual Storages**

- ❒ **In this design, each user is given their own virtual storage of the maximum size (still, at this point, 16MiB):**



- ❒ **These virtual storages are called address spaces**

  - ♦ **Theoretically, there can be up to 65,537 address spaces, although the actual limit is considerably smaller**

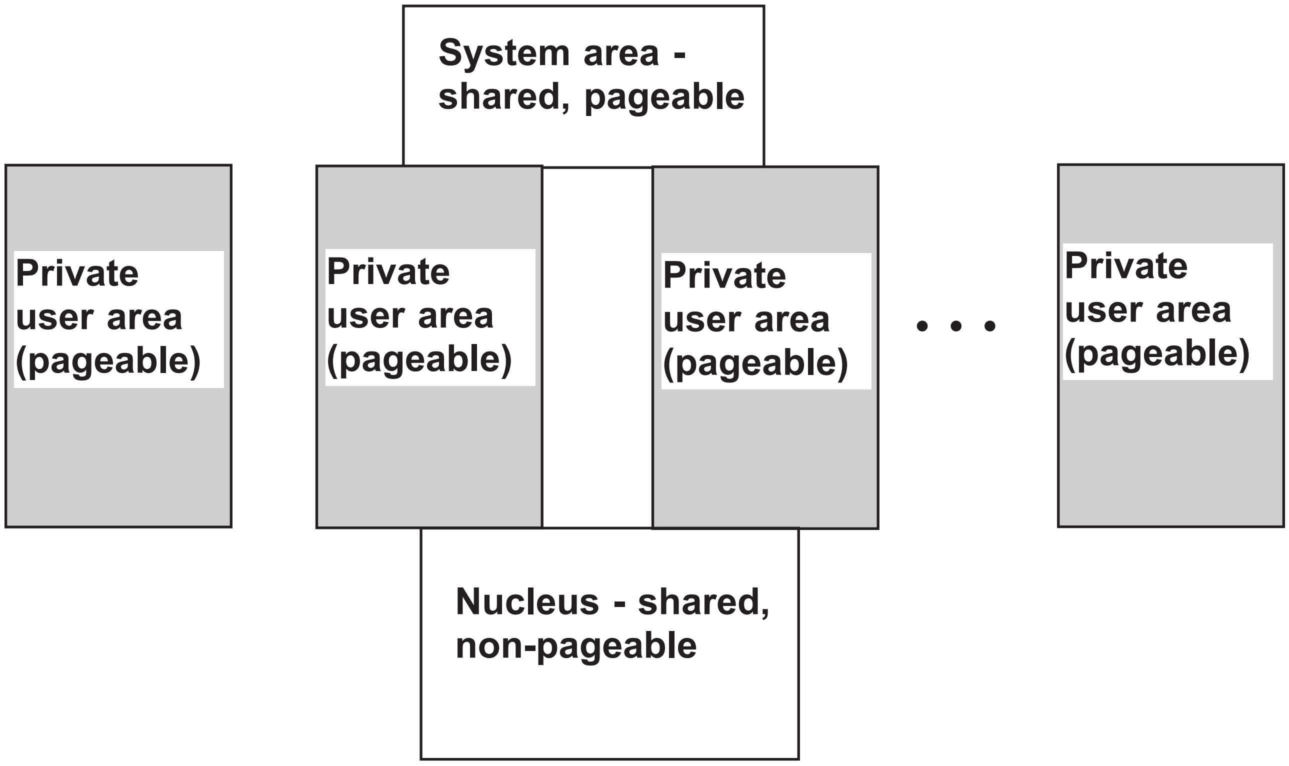
    - ✗ There are some number of system address spaces
    - ✗ Every batch initiator runs in its own address space
    - ✗ Every TSO user gets his or her own address space

The Road to z/OS, 5

❒ Because each user has their own address space, each address space needs to have a copy of the operating system

♦ Since this is the same for each user, the addressing scheme is set up to have only one, shared, copy of the nucleus area and the system area

♦ So the unique parts of an MVS system look, conceptually, like this:


System area - shared, pageable
Private user area (pageable)
Private user area (pageable)
Private user area (pageable)
• • •
Private user area (pageable)
Nucleus - shared, non-pageable

❒ Again, addresses stay at 24-bits so each address space is 16MiB in size

The Road to z/OS, 6

❒ **In the 1980's, IBM bit the bullet and extended the address space from 24 bits to 31 bits**

- **31 bits instead of 32 bits for a variety of reasons, which provides for a 2 GiB address space (2,147,483,648 bytes)**

- **This was called extended architecture, abbreviated XA, so the operating system was called MVS/XA**

- **This provides for 128 times the previous amount of virtual storage for programs to use**

- **In addition to providing a larger address space, IBM re-arranged the layout**

    - ✗ Sections of code that relied on 24-bit addresses had to remain under the 16 MiB limit (which has come to be called The Line)

    - ✗ So IBM moved as much of their code as possible above The Line (there will always have to be some code below The Line, to support older code)

- **So, the layout of an address space in MVS/XA looks like the diagram on the following page ...**

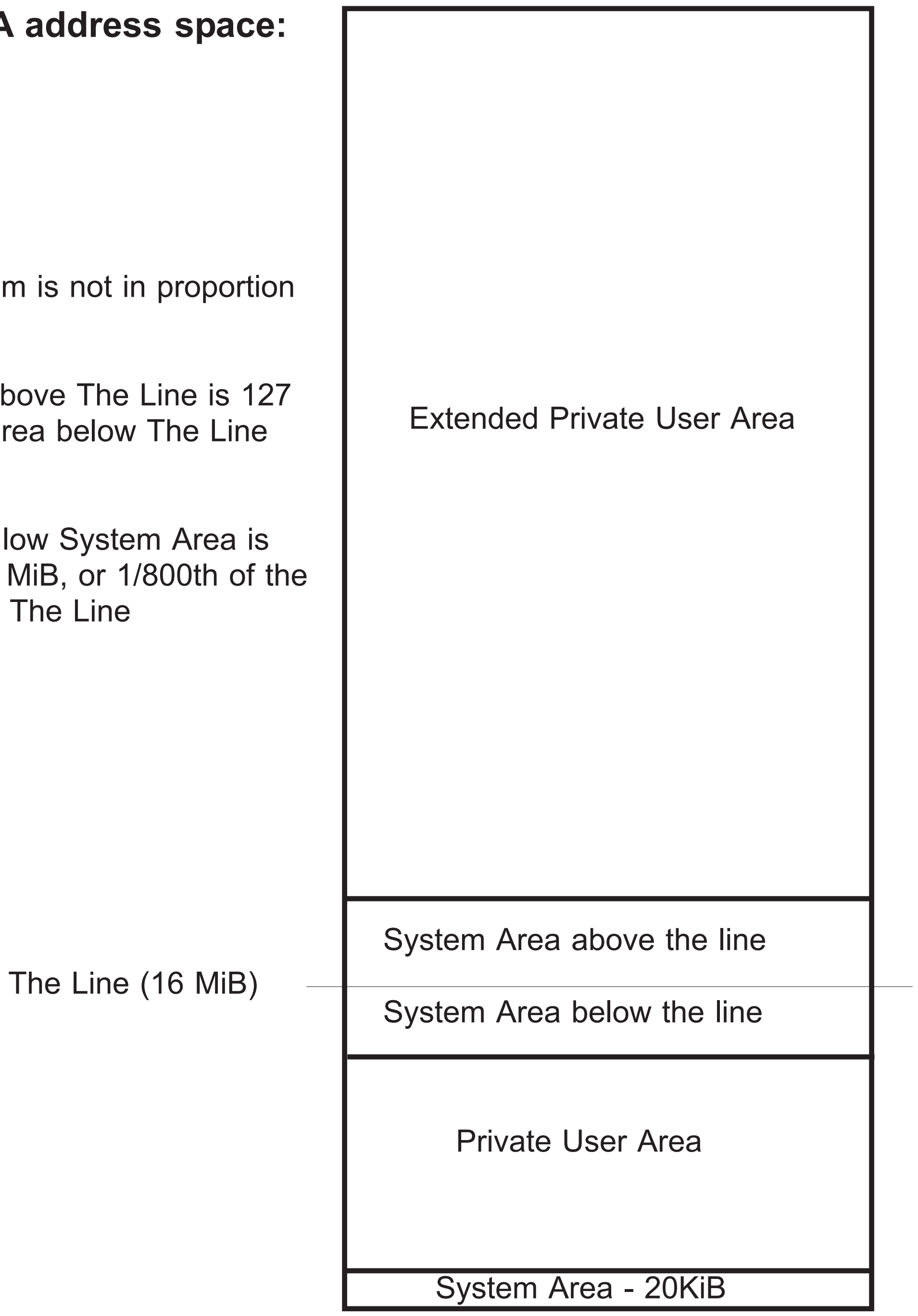
The Road to z/OS, 7

❒ **MVS/XA address space:**

This diagram is not in proportion

The area above The Line is 127 times the area below The Line

The 20KiB low System Area is 1/50th of 1 MiB, or 1/800th of the area below The Line


Extended Private User Area
System Area above the line
The Line (16 MiB)
System Area below the line
Private User Area
System Area - 20KiB

❒ **The goal is to put very little code and data below the line and to have the vast majority of programs and data reside above the line**

The Road to z/OS, 8

❒ **MVS/ESA - MVS/Enterprise Systems Architecture, introduced the concept of a** <u>data space</u>

- ♦ **A virtual storage that does not contain any copy of the operating system**
  - ✗ So, you cannot run programs in a data space
- ♦ **Use for storing data for high speed access**
  - ✗ Uses paging mechanisms which provide faster response than traditional access methods
- ♦ **Can store programs in a data space, but need to copy to an address space to run a program**
- ♦ **Data spaces can be up to 2GiB in size**
- ♦ **Data spaces can be public (shared) or private (only accessed by specifically designated address spaces)**
- ♦ **Data spaces can be system (used by MVS and its subsystems) or user (used by applications)**
- ♦ **Each address space can access up to 8,192 data spaces, for a total of 16TiB of virtual storage**

❒ **Most applications programmers, however, find it difficult to work with data spaces, so in practice, most data space usage is done by subsystems, behind the scenes**

- ♦ **For example, DB2 uses data spaces for its buffers**

The Road to z/OS, 9

❒ Other variations of MVS came along, to support enhanced hardware instructions and features, but the essence of address spaces and data spaces did not change

❒ MVS/ESA was the last release of MVS as an independent product

❒ The next step in the evolution was OS/390 (Operating System/390) which is really a packaging of components

The Road to z/OS, 10

❒ **OS/390 contains**

- ♦ **MVS/ESA code plus a number of program products as a single package**

- ♦ **Intent is to update every six months, keeping all the products in synch, thus simplifying the process of installing and upgrading systems and products (1st release was 3/96)**

- ♦ **Products included with OS/390 (among others):**
  - ✗ SMP/E (for maintenance uses)
  - ✗ TSO/E
  - ✗ ISPF
  - ✗ High Level Assembler
  - ✗ BookManager
  - ✗ DFSMSdfp
  - ✗ Language Environment (LE)
  - ✗ TCP/IP
  - ✗ DCE (Distributed Computing Environment support)
  - ✗ OpenEdition / POSIX support (UNIX under MVS!)

- ♦ **In addition, other optional products were available to be shipped in an OS/390 order, for an extra charge**

The Road to z/OS, 11

- ❐ In 2001, IBM made available new hardware, the z900 series, that supported 64-bit addresses
  - ♦ So now address spaces can be as large as 64-bit addresses allow
- ❐ A new operating system, z/OS, was announced to support the new hardware
- ❐ But z/OS is based on OS/390 - there is a solid continuity here
  - ♦ Old code can still run under z/OS, even code compiled and linked under MVT over 35 years earlier
  - ♦ To use new features, of course, you need to rewrite, recompile, and rebind
  - ♦ There are still address spaces, just larger and organized slightly differently
  - ♦ There is still an MVS component, a TSO component, and so on
- ❐ The last release of OS/390 was V2R10, available September 2000, the first release of z/OS was available March 2001
  - ♦ The announced intent is to slow the release schedule to once a year after V1R6 is available

The Road to z/OS, 12

❒ **Some of the issues around establishing a 64-bit address space are resolved this way**

- **The size of the low System Area is increased to 24KiB**

- **The previous limit of 2 GiB is now called The Bar**

  ✗ So programs or data can reside

  ➢ Below The Line

  ➢ Above The Line but below The Bar

  ➢ Above The Bar (data only, currently, no programs)

❒ **A 64-bit address space allows for a maximum address of 18,446,744,073,709,551,615**

- **That is, a 64-bit address space is 8,589,934,592 times the size of a 31-bit, 2 GiB address space**

❒ **Note that data spaces are no longer needed, but code that uses data spaces continues to run just fine**

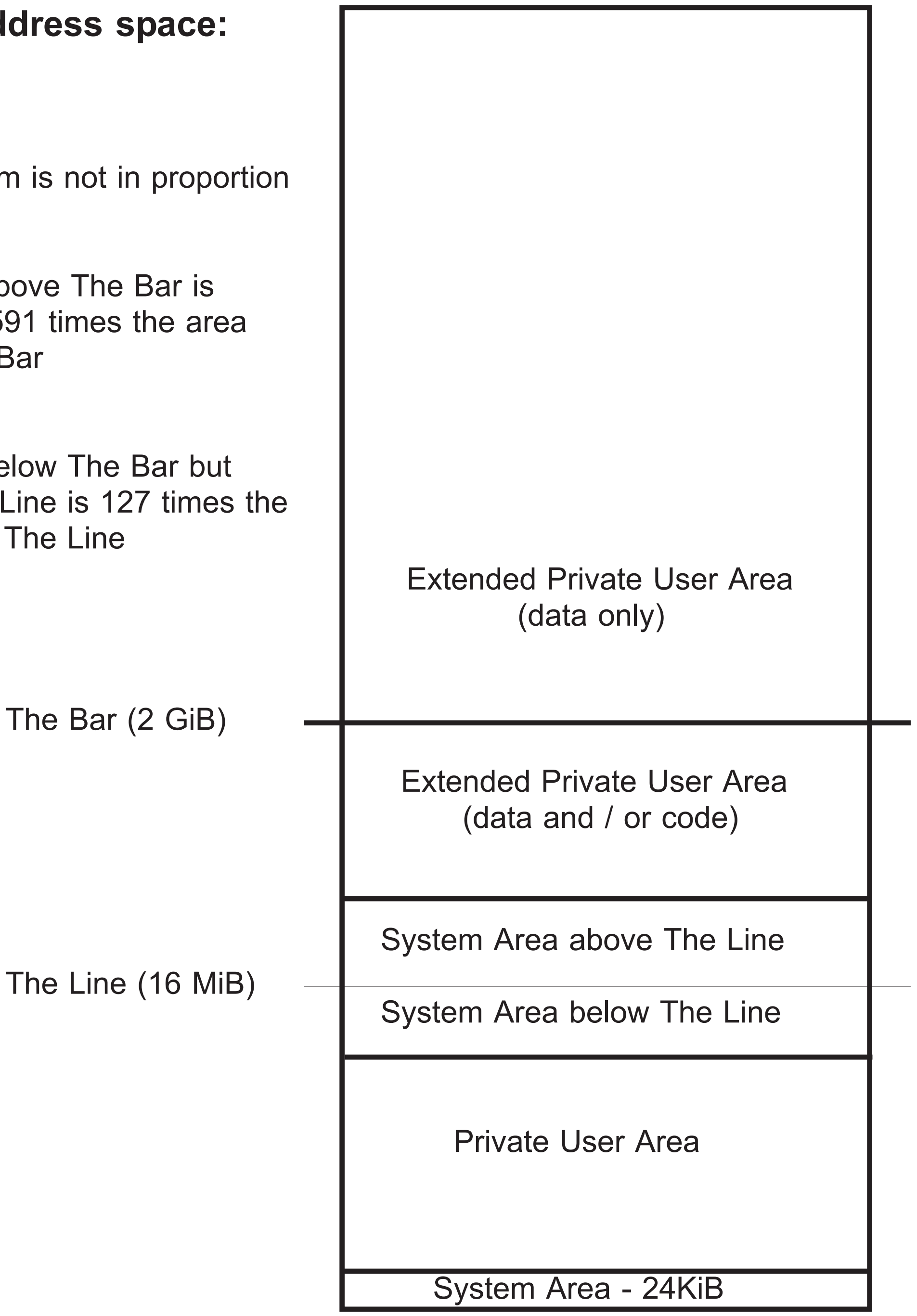
The Road to z/OS, 13

❒ z/OS address space:

This diagram is not in proportion

The area above The Bar is 8,589,934,591 times the area below The Bar

The area below The Bar but above The Line is 127 times the area below The Line


Extended Private User Area
(data only)
The Bar (2 GiB)
Extended Private User Area
(data and / or code)
System Area above The Line
The Line (16 MiB)
System Area below The Line
Private User Area
System Area - 24KiB

The Story of You

❒ To put these large numbers into some perspective, let's consider how much address space it takes to include all the information about a single person - you, for instance

- ♦ Your history (birth records, medical records, school records, driving records, credit card history, etc.)
- ♦ Your writings (letters, books, plays, technical materials, reports)
- ♦ Your productions (photographs, recordings, movies, videos, other creative work)
- ♦ Information about you (performance appraisals, reviews, references)

❒ There are a lot of variables, so first, we make some assumptions to help in our calculations (these are deliberately conservative)

- ♦ A page of text averages 500 words, and a word (English) averages 5 characters - so a text page takes about 2500 bytes, not counting formatting information
- ♦ A book averages 500 pages in length
- ♦ A photograph can be reasonably digitized to a size of 500,000 bytes, again with a lot of variables here

The Story of You, 2

❒ **So, some quick calculations**

Pages	Books	Bytes needed
1	-	2,500
2	-	5,000
20	-	50,000
200	-	500,000
500	1	1,250,000
1,000	2	2,500,000
50,000	100	125,000,000
250,000	500	625,000,000
500,000	1,000	1,250,000,000

❒ **And, just using photographs:**

Photos	Bytes needed
1	500,000
1,000	500,000,000
2,000	1,000,000,000

The Story of You, 3

- ❐ **Clearly, for most of us, all the information needed to hold our story is under 2GiB (500 books and 2,000 photographs, or equivalent), and certainly the average, world wide, is easily under 2GiB per person**

- ❐ **Now, given 6 billion people in the world, we have room for about 2.67GiB per person - in one z/OS address space**

  - ♦ **That is, <u>one</u> z/OS address space can easily hold all the information about all the people in the world today!**

    - ✗ And, there can be, theoretically, 65,368 address spaces

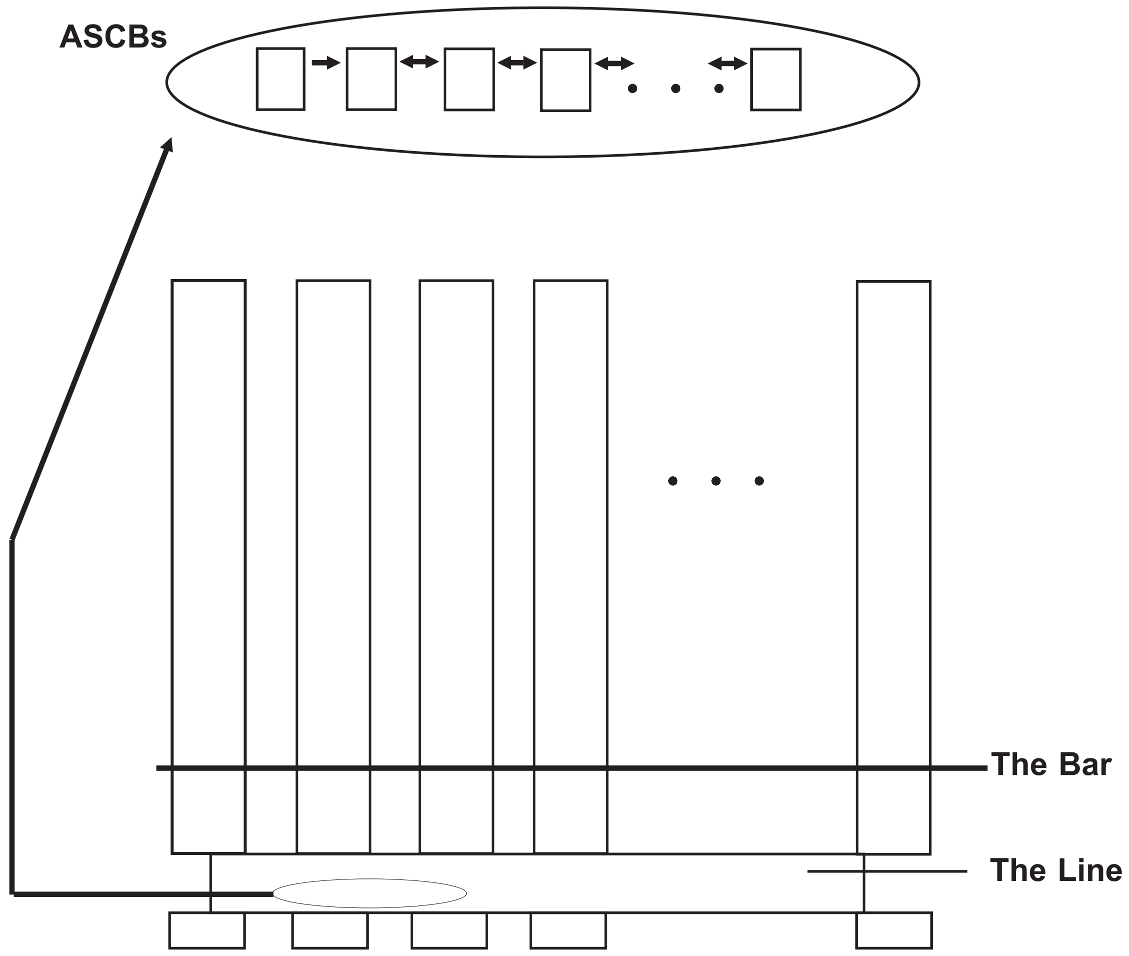
      - ➢ Per z/OS system, of which there will probably be thousands, over time

- ❐ **It really is hard to grasp the sizes we're talking about**

Address Spaces

❒ **z/OS uses a control block called an Address Space Control Block (or, of course, ASCB) to keep track of address spaces currently in existence**

- ♦ **Each address space is given a unique number, it's Address Space ID, or ASID; the ASID for an address space is stored in the address space's ASCB**
  - ✗ Along with lots of other information
- ♦ **All the current ASCBs are chained together and they are maintained in the System Area**



Address Spaces, 2

❒ **Since address spaces are potentially so large, and few applications need anywhere near the available space, z/OS actually provides only a portion of each address space**

- ♦ **This portion of an address space actually provided is called its region**

- ♦ **There is an installation default region size**

- ♦ **You can override the default region size in JCL, the TSO logon, a system exit, and other places as appropriate**

- ♦ **The point is to use resources wisely**

- ♦ **An application program can't tell it is running in a small region unless it abends due to insufficient storage space; to the application, it looks like all of an address space is available to it**

Address Spaces, 3

❒ **When z/OS is brought up (IPLed), the process provides an installation determined number of address spaces**

❒ **Additional address spaces can be created ...**

- ♦ **By operator command**
- ♦ **By TSO logon**
- ♦ **By [authorized] callable service ASCRE - Address Space Create**
  - ✗ Actually, all address space creators use ASCRE
- ♦ **Many subsystems create address spaces for their own use ...**

SubSystems

- ❐ **A subsystem is a "service provider"**
  - ♦ **A collection of one or more programs that provide services on an as-requested basis**

- ❐ **There are two types of susbsytems**
  - ♦ **The <u>primary</u> subsystem: JES2 or JES3**
  - ♦ **<u>Secondary subsystem</u> - any other subsystem**
    - ✗ Some examples of secondary subsystems provided by IBM:
      - ➢ IMS
      - ➢ DB2

- ❐ **Subsystems typically create one or more address spaces and may create and destroy additional address spaces, dynamically as needed**

z/OS

- Part 2: Context and Environments
    - Executables
    - Entry Points and Names
    - Program Components
    - Running Programs
    - Program Context

Executables

❒ **In z/OS, executable programs come in one of two formats**

Load modules

- ✗ Members in PDSs

Program objects

- ✗ Members in PDSEs
- ✗ Files in the HFS (Hierarchical File System) of UNIX System Services

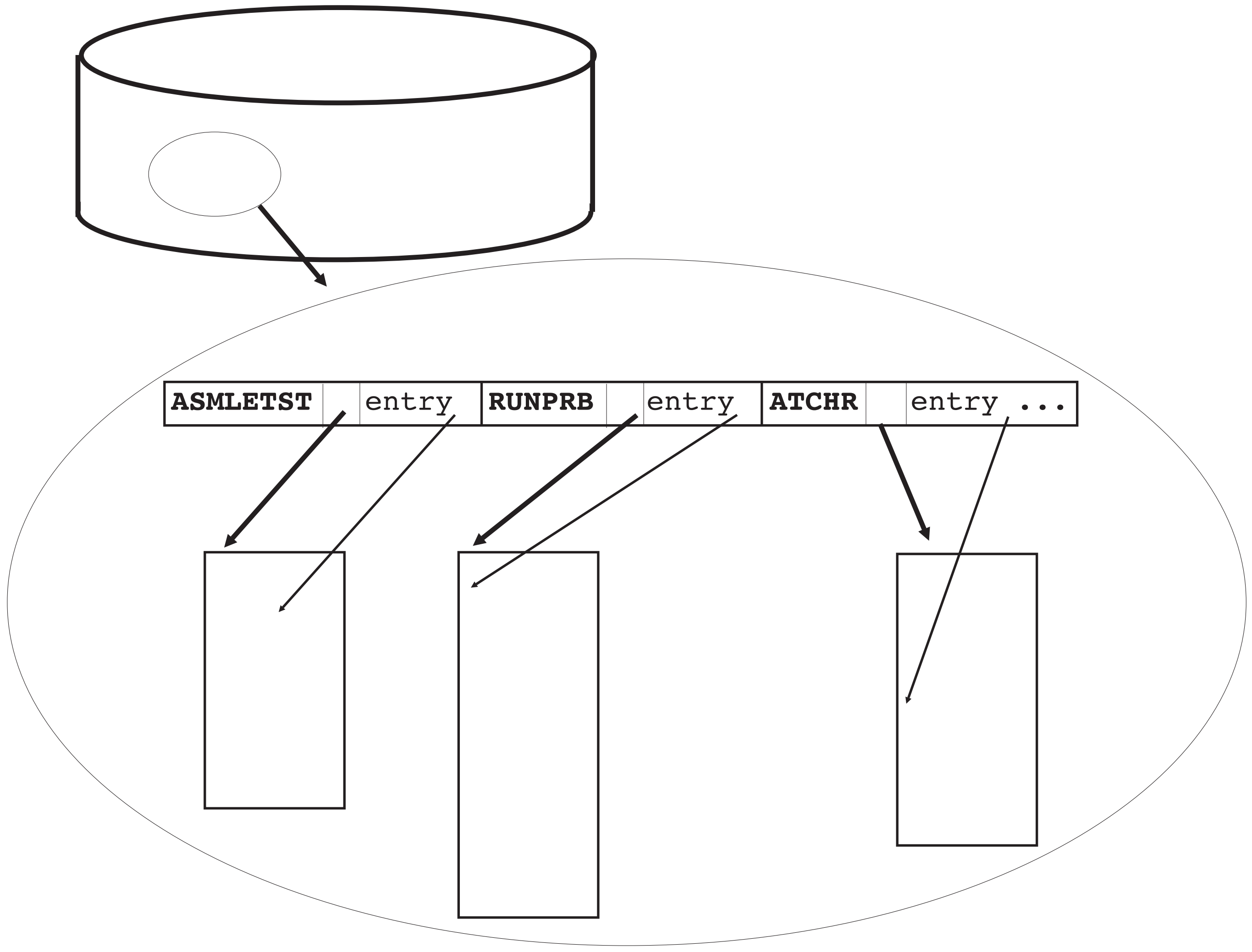
❒ **We'll use the term 'executable' or 'program' to include both load modules and and program objects**

❒ **We'll use the term 'library' to include both PDSs and PDSEs**

Entry Points and Names

❒ **An executable must have a <u>primary entry point</u>**

- ♦ **The point where execution begins if you invoke the program using its member name (//STEPx EXEC PGM=*member*)**



❒ **At the front of a library is a directory, which contains the names of each member and a pointer to where the member is found in the library's disk space**

- ♦ **The directory entry and the member itself include information about the executable, such as where all the entry points are, which is the primary, the program's AMODE and RMODE, *etc*.**

Entry Points and Names, continued

❒ **An executable may have one or more <u>alternate entry points</u>**

- **Points other than the primary entry point where execution may begin when invoked from another program (CALL, for example, or LINK); these points must be external labels**

- **You can use the binder to establish an ALIAS - a member name that relates to an alternate entry point**

  - ✗ You can also use ALIASes to relate to the primary entry point, if desired; such ALIASes are called <u>true alias</u>es

- **If you invoke an executable using an ALIAS name (EXEC PGM=*alias_name*), the same member is referenced, but entry will be at the alternate entry point**

- **In a PDS, all member names, aliases, and entry point names are limited to 8 characters**

- **In a PDSE, alternate entry point names can be up to 1024 bytes, but member names can only be 8 bytes**

  - ✗ If you have a primary entry point that is longer than 8 bytes, the name is stored as an <u>alternate primary</u> entry point, and an 8 byte primary entry point is generated by the binder

Program Components

❒ **Executable programs have these components**

Load modules

- ✗ Loadable text (instructions and hard-coded data)
- ✗ Information about where address constants are in the program
- ✗ Information about entry points
- ✗ Information about the module (size; initial addressing mode (AMODE); residency mode (RMODE); authorization; *etc.*)

Program objects

- ✗ Loadable text (instructions and hard-coded data)
- ✗ Other loadable information (for example, location of referenced DLL variables and routines)
- ✗ Information about where address constants are in the program
- ✗ Information about entry points
- ✗ Information about the object (size; initial addressing mode (AMODE); residency mode (RMODE); authorization; *etc.*)

Running Programs

❒ Every program running in a z/OS system runs in an address space, of course

❒ But within an address space, there is also a context within which the program runs - routines and control blocks to manage the system; for z/OS, the context includes ...

- ♦ Contents management
- ♦ Task management
- ♦ Workload management
- ♦ Dispatching
- ♦ Interrupt handling

❒ We give a high level overview of the context on the following pages

Program Context

❒ **<u>Contents management</u> - these routines use control blocks to keep track of programs in memory: where they are, their attributes, and how many users are sharing the program**

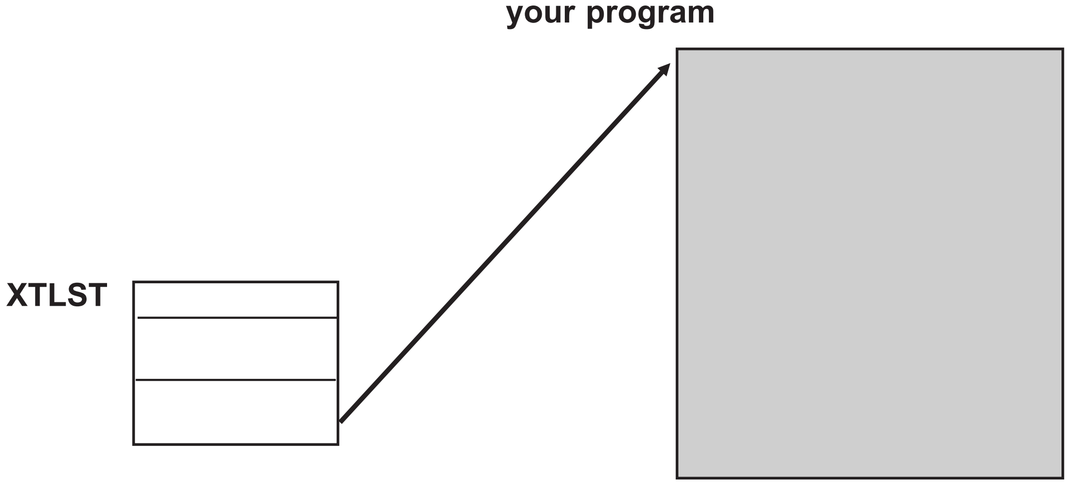
- **z/OS uses this information to see if a program needs to be loaded into memory and relocated, or if an existing copy in memory is available to use**

- **Application programs can request contents management services, too, for example to dynamically load subroutines or to find out information about an executable**

- **The primary control blocks are:**

  ✗ <u>XTLST</u> - extent list; contains starting address and length of an executable in memory

  ➢ Note: in some cases, an executable can contain part of its text above The Line and part below The Line; there will be an XTLST entry for each part (called a segment)

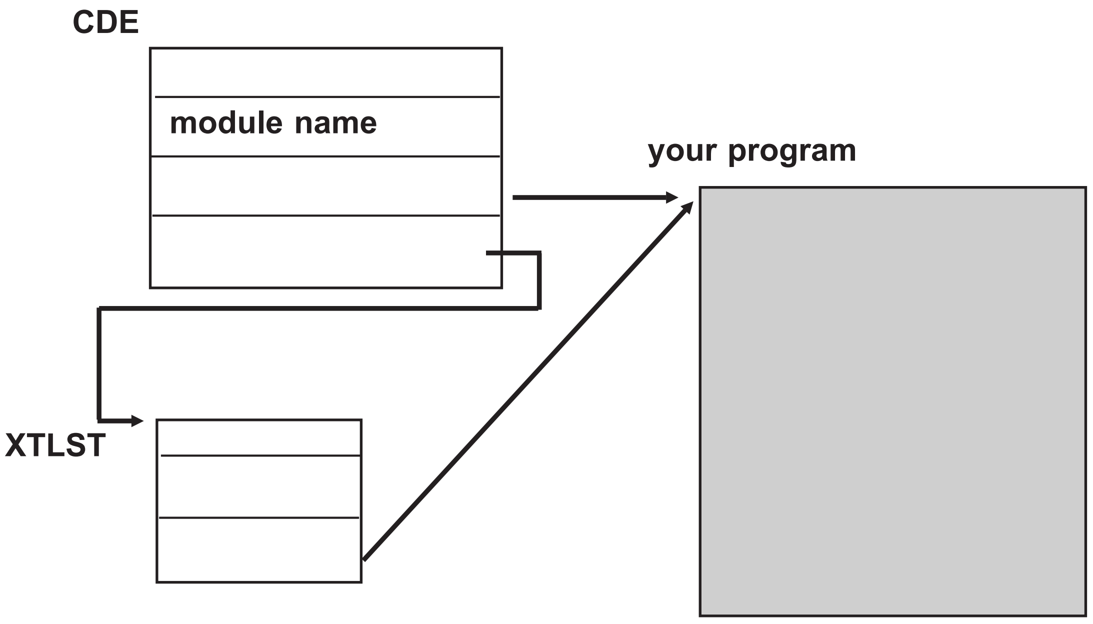


Program Context, continued

❒ **Contents management, continued**

- ♦ **The primary control blocks, continued:**

  ✗ CDE - Contents Directory Entry; contains member name, entry point address, module attributes, address of the related XTLST, count of number of current users [the use count], and address of next element on the CDE chain

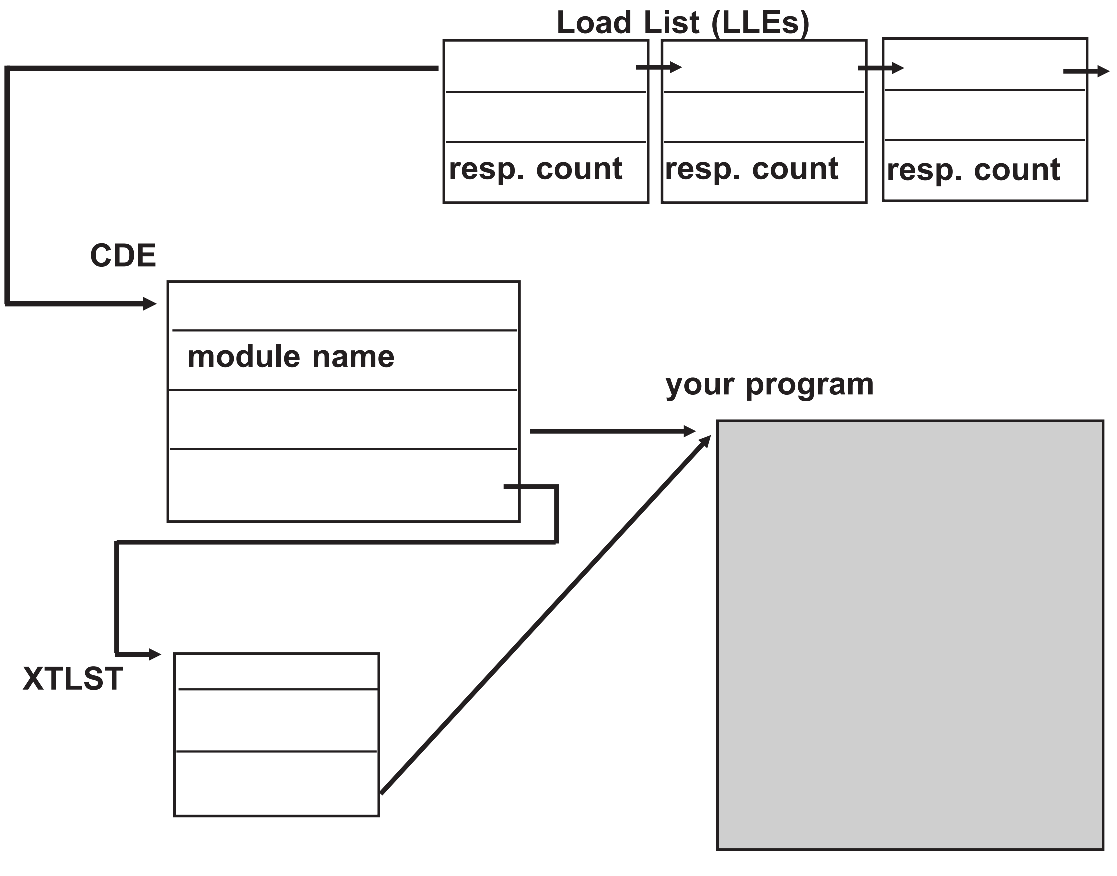


Program Context, continued

❒ **Contents management, continued**

- ♦ **The primary control blocks, continued:**

  ✗ LLE - Load List Element; contains information about executable programs LOADed by the main program (number of current LOAD requests for the program [the responsibility count], pointer to the CDE for the program, address of next element on the list)



Program Context, continued

- ❒ Task management - these routines use control blocks to keep track of dispatchable units of work: tasks

  - ♦ **Each task has a chain of one or more request blocks (RBs) that describe services and programs that have to run**

    - ✗ For example, your application program has a PRB (see below) to manage the program's status; if the program requests a system service (OPEN, for example), the service will have an SVRB (SuperVisor Request Block) to manage the progress of the service until control returns to your program

  - ♦ **So essential control blocks are:**

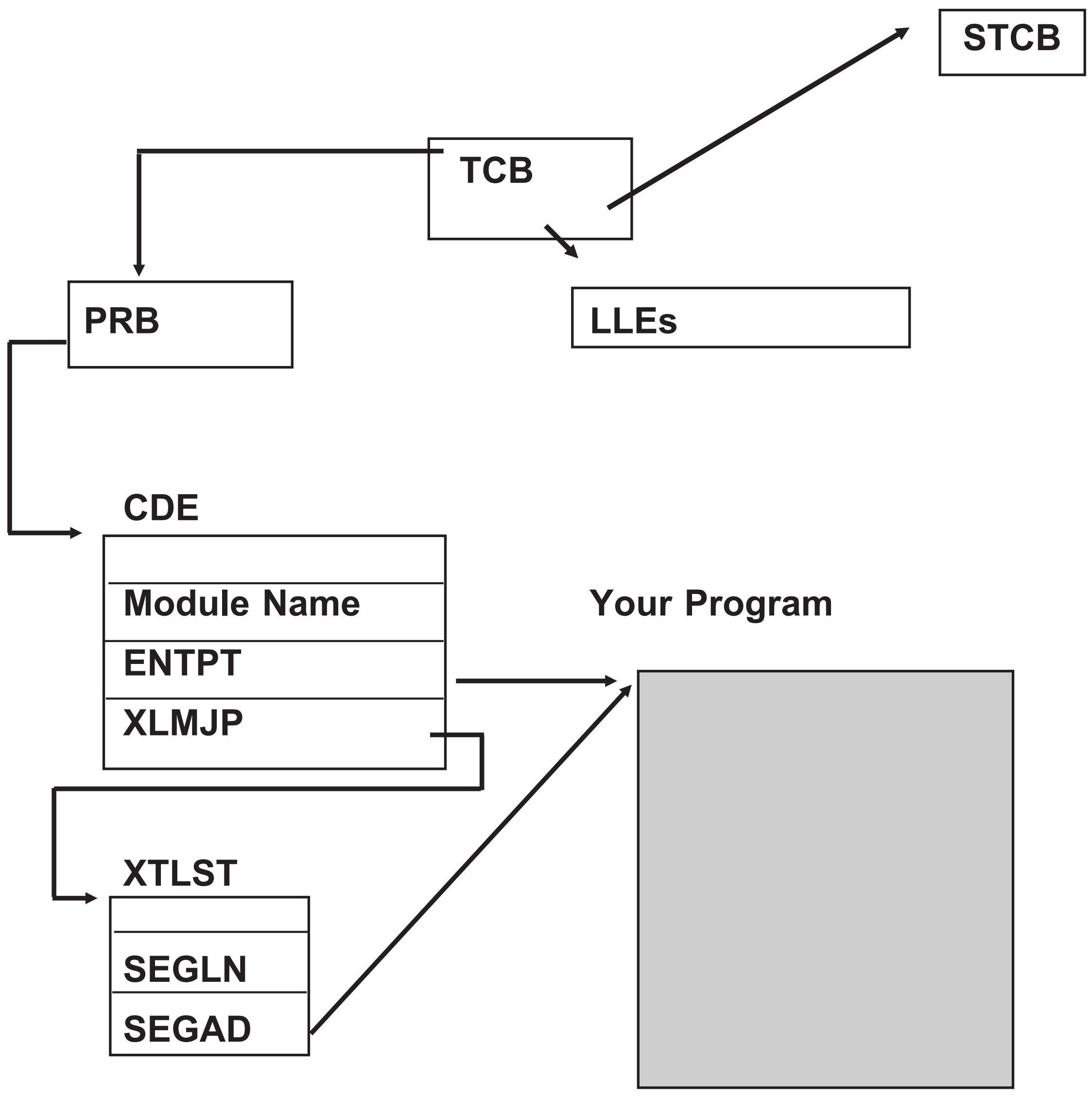
    - ✗ TCB - Task Control Block; represents a task; contains pointer to current RB on chain, pointers to related tasks' TCBs, pointers to related I/O structures, task status flags, task register save area, pointer to ECB (Event Control Block) used to post when the task is completed (optional), pointer to LLE (see previous page), pointer to related STCB

      - ➢ Note that TCBs are below The Line, STCBs are above The Line

    - ✗ STCB - Secondary TCB; save area for high words of 64-bit registers; pointers for UNIX System Services control blocks, *etc.*

    - ✗ PRB - Program Request Block; contains current status information for a program currently being run, a register save area, address of CDE for program

Program Context, continued



Program Context, continued

❒ **The other parts of the context for running programs that we focus on are discussed in terms of functionality, not control blocks**

❒ **<u>Workload Management</u> - routines that decide which program to run next (*running* means letting a processor execute the machine instructions in the executable program)**

- **The decision is based on many factors, including**
  - ✗ The current workload on the system
  - ✗ Installation specified goals for various kinds of tasks under different workloads
  - ✗ What class of work a task belongs to

- **Workload management uses this information to rearrange the priorities of work in the system dynamically**
  - ✗ Ready work is put on the Work Unit Queue in priority order

Program Context, continued

- **Dispatching - the routines that actually turn control over to the next task to run (run the program represented by the first RB on the chain of RBs)**

    - **This includes loading registers and the saved PSW (Program Status Word) as they were the last time the program was running, to pick up where it left off, as it were**

    - **A task is dispatched with a time interval, and the task is paused when the task voluntarily goes into a wait (for I/O, for example) or the time interval expires, or terminated when the task is cancelled or abends**

- **Although we are omitting a lot of details, dispatching ultimately comes down to running tasks - and this is the essence of running a program under z/OS, whether in batch or online**

Program Context, continued

- **Interrupt Handling - these routines save the status of the current task (register contents, PSW, etc.) when an interrupt occurs**

    - **If some task is waiting for that specific interrupt, that task can now be marked as available to run**

        - If not, the interrupt is handled and dispatching will continue as before

    - **Control then returns to workload manager who may readjust the priorities of all waiting work**

    - **Control then returns to the dispatcher, who dispatches the highest priority ready task, with a time interval**

Program Context, continued

<u>Note</u>

- **The computer has multiple processors, so many tasks are usually executing simultaneously**

  - **If two or more tasks can access or modify the same resource (program, memory location, external device), the system has to ensure conflicts are resolved and changes are not lost**

    - z/OS manages these issues automatically for resources it is in charge of

    - But the system can't always know about potential conflicts, since the application programmer may be making updates that are not represented in any system control blocks

  - **Facilities such as locks, latches, signals, mutexes, task synchronization, and enqueues are available for use by the application programmer**

This page intentionally left almost blank.

z/OS

❒ Part 3: Batch

- ♦ Running Programs in z/OS Batch
- ♦ z/OS Dispatching and Task Management
- ♦ z/OS Contents Management
- ♦ z/OS Essential Control Blocks
- ♦ Your Program and Subroutines: Static CALLs
- ♦ Your Program and Subroutines: Dynamic CALLs
- ♦ Program Reusability Attributes
- ♦ Program Arguments
- ♦ Program Errors
- ♦ ATTACH Options

Running Programs in z/OS Batch

- **You request a program run in the batch by submitting JCL to the JES2 or JES3 subsystem**
  - **JES, the job entry subsystem (either JES2 or JES3), reads the JCL onto the SPOOL for a batch initiator to decide when it is the job's turn to run**

- **As a brief reminder, the JCL includes one or more EXEC statements that names a program to be run, possibly passing a parameter string, and each EXEC statement is followed by DD statements that describe the files needed by the program; that is:**

```
//jobname  JOB  job_parameters
//stepname EXEC PGM=progname,PARM='parm_string'
//ddname    DD  dd_parameters
//ddname    DD  dd_parameters
.
.
.
//stepname EXEC PGM=progname,PARM='parm_string'
//ddname    DD  dd_parameters
//ddname    DD  dd_parameters
.
.
.
```

Running Programs in z/OS Batch, continued

❒ Batch initiator address spaces tend to stay up all the time, from IPL to shutdown for maintenance (or system crash)

- ♦ Although the operator can issue a 'stop' command for an initiator address space
- ♦ And the operator can issue a 'start' command to start up new initiator address spaces

Running Programs in z/OS Batch, continued

❒ **Most address spaces have the control block structure reflected on the following pages**

- ♦ **A 'start' or 'logon' command is issued (automatically at IPL or by the operator or by a TSO user logging on), which causes an address space to be created and the Region Control Task (RCT) to be set up and given control**

- ♦ **RCT builds tables necessary to support the new address space's virtual storage, then attaches two subtasks**

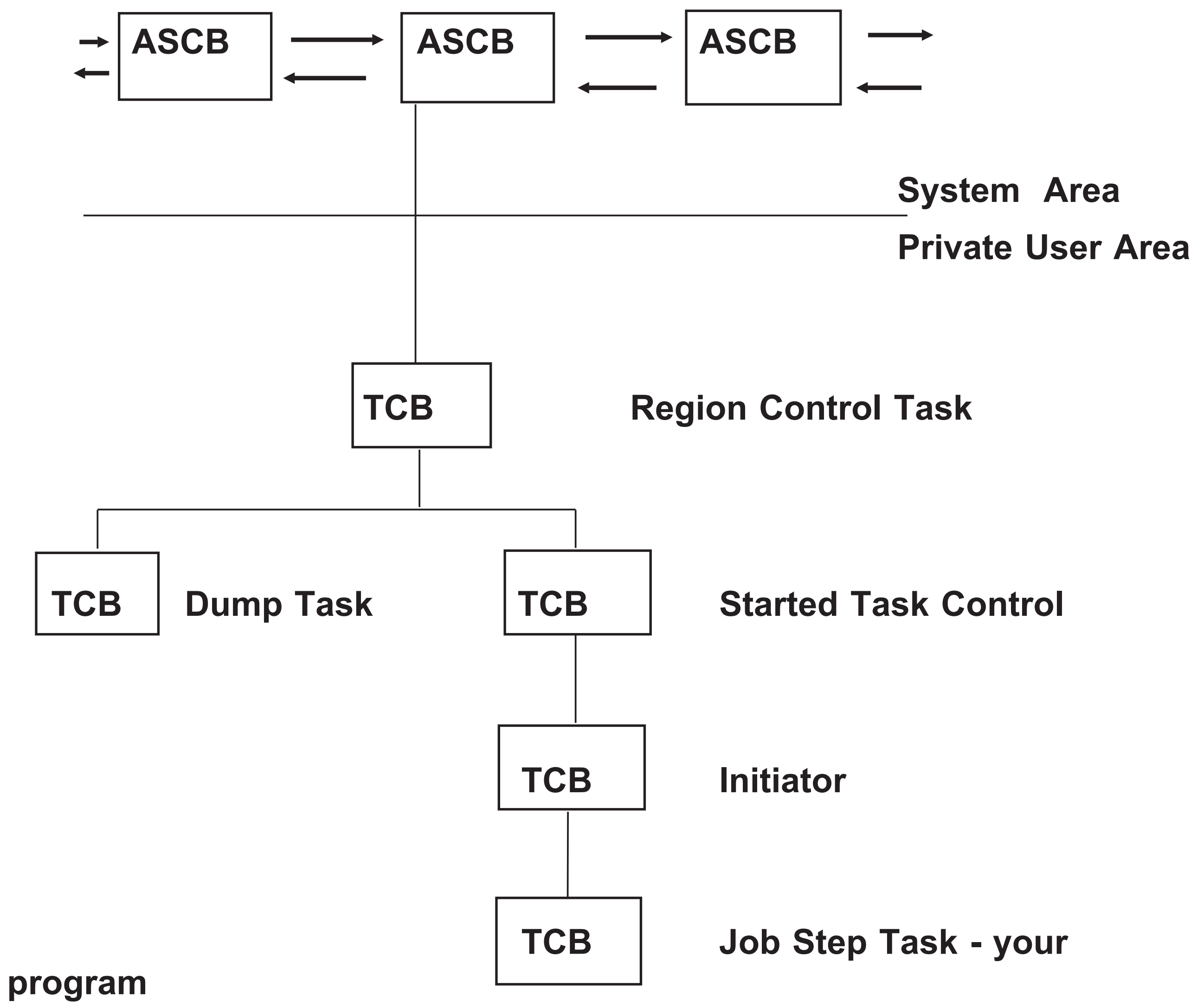
  - ✗ Dump Task - there is copy of this task in every address space, for providing a dump if the system abends or the address space abends; hopefully it is never needed

  - ✗ Started Task Control (STC) attaches the program specified on the start JCL, which, for a batch address space is the initiator task; which will remain available until the system is shut down or the operator issues a 'stop' command for the initiator

- ♦ **The Initiator takes each step in your job, one at a time, allocates the resources the program in the step will need, and ATTACHes the program as the Job Step Task - this is your program running**

  - ✗ ATTACH is a system service to create a task that is a **subtask** of the requesting task

z/OS Dispatching and Task Management



Each TCB also points to an STCB
that resides above The Line.

ASCB — Address Space Control Block

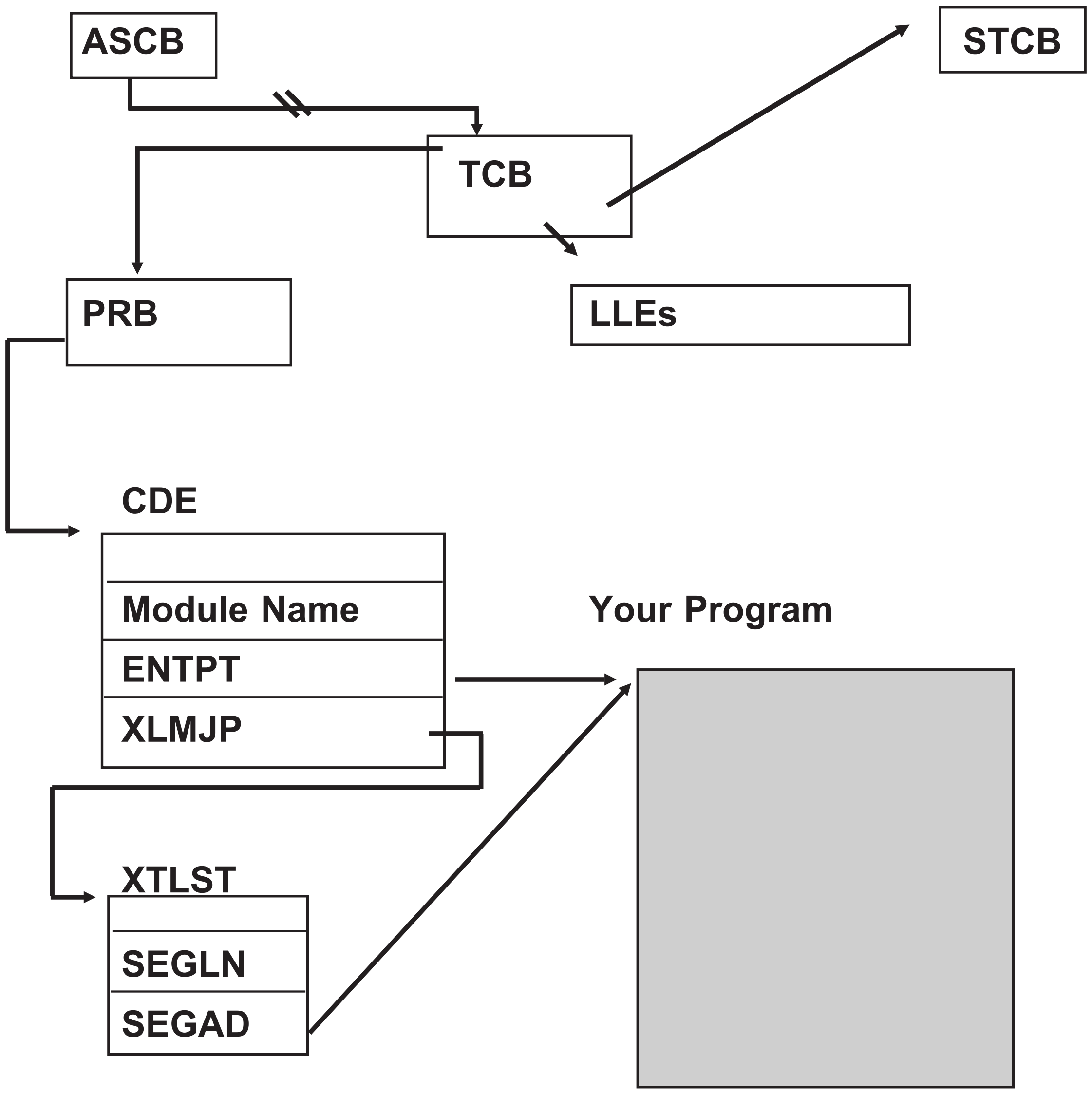
TCB — Task Control Block

STCB — Secondary Task Control Block

z/OS - Contents Management

- ❒ **The program to run is specified on the EXEC statement in your JCL; when the Initiator ATTACHes the program ...**
  - ♦ **Contents management searches to see if the program is already in your address space and available**
  - ♦ **If not, search the libraries (STEPLIB or JOBLIB (but not both), and LINKLIST libraries if necessary)**
    - ✗ If not found, abend the step and go on, usually flush the rest of the job
  - ♦ **If found, allocate memory to hold control blocks and the program; load the program into memory; relocate any address constants**
  - ♦ **Mark your task as available so it will eventually be dispatched**
- ❒ **When the job step task program terminates, it returns to the Initiator who does clean up (closes files, frees storage, detaches the task, etc.) and looks to see if there is another step**
  - ♦ **If, so, repeat the process for that step**
  - ♦ **If not, clean up for this job (final data set disposition,** *etc*.**) and go to the job queue for the next job in the queue**
- ❒ **So as each program runs under z/OS, it runs with this kind of control block structure..**

z/OS Essential Control Blocks



ASCB — Address Space Control Block

TCB — Task Control Block

STCB — Secondary Task Control Block

PRB — Program Request Block

CDE — Contents Directory Entry; LLE — Load List Entry

XTLST — eXTent LiST

SEGLN — length of module in bytes

SEGAD — address where module loaded in memory

Your Program and Subroutines: Static CALLs

- ❒ **While your program runs, it can request other programs to run on its behalf - subroutines**

- ❒ **If your program invokes a subroutine through a static CALL, the subroutine has been bound to the main program at program bind time**

  - ♦ **Both programs are physically included in the load module or program object**

  - ♦ **When the program was loaded and relocated, all its statically called subroutines were automatically brought along**

  - ♦ **Static CALLs do not require any assistance from the operating system (and so are very fast)**

    - ✗ No building of LLE, CDE, XTLST; no additional address constant relocating [after the initial loading of the executable, of course]

Your Program and Subroutines: Dynamic Calls

❐ There are issues and tradeoffs involved, but often you want to invoke a subroutine dynamically

- That is, the routine is not bound into your main program's load module or program object

- You do this by asking z/OS for assistance to find, load, and run the program you had in mind, at run-time, just when you need it

Your Program and Subroutines: Dynamic Calls, continued

❒ **There are several services available:**

- **LINK - dynamically find, load, and run the subroutine, return to the invoker, then delete the subroutine from memory**

- **LOAD - dynamically find and load the subroutine on request**

  - ✗ It is then the responsibility of the LOADing program to invoke the loaded program when it needs to

  - ✗ The loaded subroutine will stay in memory for reuse until explicitly DELETEd or the originating task ends

- **XCTL - dynamically find the requested program and replace the invoking program with the new program; the original program is no longer [logically] in storage**

- **ATTACH - dynamically find, load, and run the subroutine as a subtask**

  - ✗ The originating task ("mother task" can run independently of the subtask ("daughter task")

  - ✗ A task can ATTACH many subtasks, and each subtask may in turn ATTACH subtasks

  - ✗ A mother task may request to be notified when a daughter task completes

Your Program and Subroutines: Dynamic Calls, continued

❒ **When a LOAD request is made, program management searches for the requested program**

- ♦ **If not found in memory, search the libraries; if not found in libraries, ABEND**

- ♦ **If not found in memory but found in libraries, then,**

  - ✗ Build CDE, XTLST, and LLE

    - ➢ Put CDE on the chain of CDEs with a use count of 0, and put LLE on the chain of LLEs, with a responsibility count of 1

  - ✗ Allocate storage for program

  - ✗ Load program into storage, relocating address constants as necessary

  - ✗ Return to requester, returning the address of where the program was loaded

- ♦ **If found in memory**

  - ✗ Add one to responsibility count

  - ✗ Return to requester, returning the address of where the program was loaded

Your Program and Subroutines: Dynamic Calls, continued

- When user wants to invoke a LOADed program, use a CALL or LINK
  - CALL ends up being a direct branch; system does not know about it, no PRB is created; this is not available in high level languages
  - LINK is described shortly

Your Program and Subroutines: Dynamic Calls, continued

- **If a DELETE request is made, program management searches for the requested program in memory**
    - **If not found, return with non-zero return code**
    - **If found,**
        - Subtract 1 from responsibility count
        - If responsibility count now zero,
            - Free storage holding module
            - Free LLE and CDE (adjusting chains as necessary)

Your Program and Subroutines: Dynamic Calls, continued

❒ When a LINK request is made, program management ...

- ♦ Issues a LOAD request for the program (if not successful, ABEND)
- ♦ Builds a PRB, put at head of RB chain from TCB
- ♦ Adds one to use count (in CDE), and one to responsibility count (in LLE)
- ♦ Invokes the program
- ♦ On return, deletes the PRB (adjusting the RB chain), and issues a DELETE

❒ Note that if you will be LINKing to a program repeatedly, it is often best to do an explicit LOAD of the program first

- ♦ Then your LINKs will not delete the program on return, since the responsibility count will not be zero when the DELETE is issued after return from the program

Your Program and Subroutines: Dynamic Calls, continued

❐ **When an XCTL request is made, program management ...**

- ♦ **Issues a LOAD request for the module (if not successful, ABEND)**

- ♦ **Issues a DELETE request for the program issuing the LOAD**

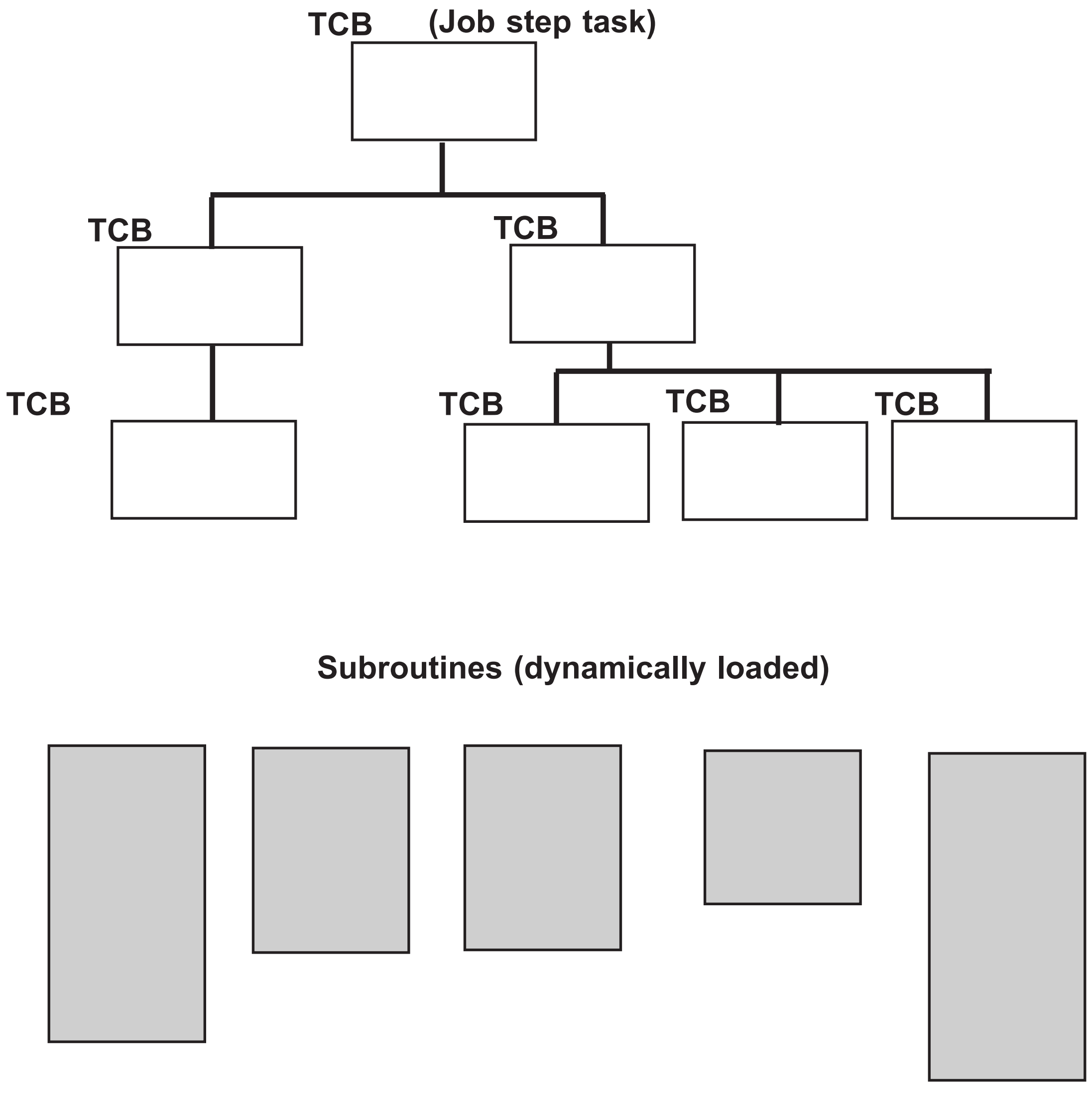
- ♦ **Uses the PRB of the program issuing the LOAD to now represent the program being XCTL'ed to**

Your Program and Subroutines: Dynamic Calls, continued

- **When an ATTACH request is made, program management**
  - **Constructs a TCB and puts the TCB on the chain of TCBs for the address space**
  - **Issues a LINK to the program being ATTACHED**
    - That is, the program will be searched for
      - If not found, ABEND
      - If found in memory, add one to the use count and responsibility count
        - If not found in memory, must be in the libraries, so allocate storage, fetch into memory, relocate address constants, build CDE and LLE
      - Build PRB and chain off the new TCB
  - **At this point, the newly created subtask is an independent unit of work, competing to be dispatched like all other tasks in the address space**
  - **This subtask may, in turn, issue requests for LOAD, DELETE, LINK, XCTL, ATTACH and other system services**

Your Program and Subroutines: Dynamic Calls, continued

❒ **In a complex multi-tasking environment, you can have many tasks and subtasks that, in turn, issue LOADs and LINKs to the same subroutine(s):**



❒ **It is in this situation that we care about program <u>reusability attributes</u>**

Program Reusability Attributes

- ❐ Every program can be assigned several reusability attributes; the ones we care about are:
  - ♦ REENTRANT or not (also, in the literature, REENTERABLE or not)
  - ♦ REUSABLE or not (also, as a phrase: SERIALLY REUSABLE)
- ❐ If a program is REUSABLE, it either always initializes itself on entry or re-initializes itself on exit
  - ♦ So, if two tasks try to LINK to a REUSABLE program, the first one to request the routine will get it; the other task(s) will wait until the first requester is done
- ❐ If a program is REENTRANT, a single copy can be run simultaneously under two (or more) tasks
  - ♦ The single copy in storage will be used for multiple requestors, simultaneously if necessary
- ❐ If a program is not reusable (the default), each time a task LINKs, XCTLs, or ATTACHes that program, a new copy is brought in to memory from disk (any previous copy in memory is deleted)
  - ♦ Note that if another task is running an existing copy (there is a PRB for the program), any other task issuing LINK, XCTL, or ATTACH must wait until the current copy is done being used

Program Reusability Attributes, continued

- ❒ **High level languages (COBOL, PL/I, and C) all have a compile time option RENT that adds the code to create a reentrant version of the program being compiled**
  - ♦ **Note that current PL/I compiler always produces reentrant code, regardless of the compiler option, and C++ always produces reentrant code**

- ❒ **For Assembler, the programmer is responsible for adding code if he or she wants a program to be reentrant**
  - ♦ **For a program to be reentrant, it must not modify itself**

- ❒ **Whether it's done for you by a compiler or you do the work yourself in Assembler, the technique to create reentrant code is the same**
  - ♦ **Basically, you obtain storage outside your load module and this is where you build constructs that must be modified (calculations, formatting lines for display or print, many system services such as I/O requests, and so on)**
    - ✗ There are several system services for dynamically obtaining storage: GETMAIN, STORAGE, CPOOL among them; our discussion here assumes GETMAIN, but you could use others
    - ✗ Each task using the subroutine will work with their own copy of the obtained storage, since this storage is managed off the TCB

- ❒ **In all cases, you must also use the binder to mark a program as reusable or reentrant, otherwise program management won't know and will assume the default (not reusable)**

Program Reusability Attributes, continued

❒ Although writing reentrant code is somewhat more difficult for Assembler programmers, the trend is for a need (or at least a strong inclination) for reentrant code

- ♦ Programs running as a UNIX process, for example, must be reentrant (more later)
- ♦ Programs working with CICS, for example, need to be reentrant, since transactions are managed as subtasks and there may be many invocations of a transaction simultaneously
- ♦ Newer machines often run code faster if it is reentrant than if it is not

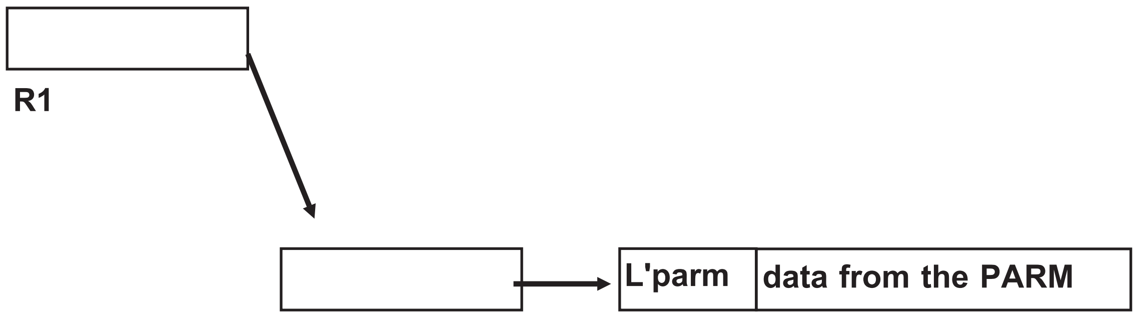
Program Arguments

❒ **When a program is ATTACHed by an initiator (that is, the job step task), a single parameter is passed, based on the PARM field from the EXEC statement**

- ♦ **Behind the scenes, the job step task initially gets control with register 1 pointing to the address of a string**

- ♦ **The string begins with a halfword binary integer that contains the length of the PARM data that was passed**

  - ✗ Followed by the PARM data:



❒ **For all other programs (programs reached through CALL, LINK, XCTL, or ATTACH), you may pass any number of arguments**

- ♦ **R1 is set up to point to a list of addresses, one for each argument being passed**

- ♦ **The details are beyond the scope of this discussion**

Program Errors

- ❒ **Eventually, any program can encounter a serious error, one that causes it to ABEND ("blow up")**

- ❒ **In z/OS, when this happens, z/OS looks to see if you have specified any error handling routine(s) to get control**

  - ♦ **If not, your task, and all your subtasks, are terminated**

- ❒ **There are two major facilities for applications programmers to use in error handling**

  - ♦ **ESPIE (Extended Specify Program Interruption Exit) - if you request it, a user-written ESPIE routine can get control when specific program checks occur**

    - ✗ An ESPIE routine can allow a terminating task to continue running (possibly after doing some fix up) or allow the task termination process to continue

  - ♦ **ESTAE (Extended Specify Task Abnormal Exit) - if you request it, a user-written ESTAE routine can get control when just about any abend occurs**

    - ✗ ESTAE routines can catch program checks not caught by an ESPIE routine, for example, or most system abends such as S013, S80A, and so on

- ❒ **Note that earlier versions of these (SPIE and STAE, respectively) are still supported, but ESPIE and ESTAE are newer and more flexible**

Program Errors

❒ **Assembler programmers create ESPIE or ESTAE environments by issuing macros**

  - ♦ **They must also supply the related error handling exit programs**

❒ **For high level languages, the compilers generate various combinations of these, depending on language, compiler options, and programmer code**

ATTACH Options

❒ **When you ATTACH a subtask, you may attach it with various options:**

- ♦ **Request to be notified when the subtask ends (or not)**
- ♦ **Specify an error handling routine at the task level (ESTAI)**
- ♦ **Specify whether all, some, or none of GETMAINed storage is shared between the mother task and the daughter task**
    - ✗ Note the terminology, which is historical: the attaching task is the <u>mother task</u>, the subtask is a <u>daughter task</u>
    - ✗ You can talk about <u>sister tasks</u>, which are all daughters of the same mother task
        - ➢ Although I have never seen it done, presumably you can talk about tasks that are cousins, aunts, and so on(!)

ATTACH Options, continued

- **After issuing an ATTACH, the mother task can continue to run independent of the daughter task, perhaps ATTACHing additional subtasks**
  - **At some time, the mother task can WAIT for one or more (or all) subtasks to complete**

- **When a subtask ends, if the mother task asked to be notified, it is notified (so if it is waiting, the mother task can now do further work knowing the subtask has completed)**
  - **If this is the case, at some point the mother task must explicitly DETACH the daughter task, or the subtask stays around until the mother task terminates**

- **If the mother task did not request notification of the daughter task finishing, the mother need not DETACH the daughter**
  - **In this case the DETACH will be done automatically when the daughter completes**
    - ✗ BUT it is an error for the mother to terminate before all its subtasks have completed

Access to Files

❒ **A batch program has the potential to access these kinds of files:**

- ♦ **Sequential files, PDSs, PDSEs (even still have some support for older file types such as BDAM)**
    - ✗ Use DCB control blocks in program and JCL for allocation (although can call system services for dynamic allocation)
- ♦ **VSAM files**
    - ✗ Use ACB control blocks in the program and JCL for allocation (although can call system services for dynamic allocation)
- ♦ **Data bases - hierarchical, network, relational**
    - ✗ Through embedded calls to product services
- ♦ **Files in the HFS (Hierarchical File System)**
    - ✗ Through DCB approach or calls to UNIX services

❒ **Generally, batch programs have no awareness of anything like a terminal**

- ♦ **Although in many languages a batch program can communicate with the console operator through system services**
    - ✗ This is generally frowned on by most installations

z/OS

❒ Part 4: TSO and ISPF

- The TSO Environment
- TSO Task Structure
- TSO Command Processors
- ISPF - Integrated System Productivity Facility
- Batch TSO

The TSO Environment

- **z/OS and its predecessors have batch processing in their blood**
  - **The original design for OS/360 only supported interactive communication between the system and the operator**
    - ✗ The bulk of the processing would be done using batch
- **TSO (Time Sharing Option) came out around 1970, and has been enhanced substantially over the decades**
- **Now, when each TSO user logs into the system, an address space is created for him or her and a communication link is established between the system and the user**
  - **The TMP (Terminal Monitor Program) is a command processor task**
    - ✗ Although users may supply their own, very few shops use anything other than the IBM-supplied TMP program, IKJEFT01, these days

The TSO Environment, 2

- **The TSO user can issue TSO commands to**
  - **Work with data sets**
  - **Submit jobs to the batch (also called the background)**
    - ✗ Note that batch jobs run in separate address spaces from the TSO session
  - **Control batch jobs (cancel, pause, work with output), assuming proper authority**
  - **Run programs in the foreground**
    - ✗ Note that the terminal user must wait until the program is done; the TMP task will attach the program and issue a WAIT for the daughter task to complete
  - **Run REXX execs and CLISTs (these are two scripting languages that allow you to save system commands and logic in a single file and invoke the whole script at one time)**
    - ✗ These two languages are interpreted, not compiled
      - ➢ The REXX or CLIST language processor examines each line at run time and executes the instructions or commands

The TSO Environment, 3

- ❐ When a command is presented to the TMP, the TMP attaches a subtask to attach the command processor - a program designated to handle that command name

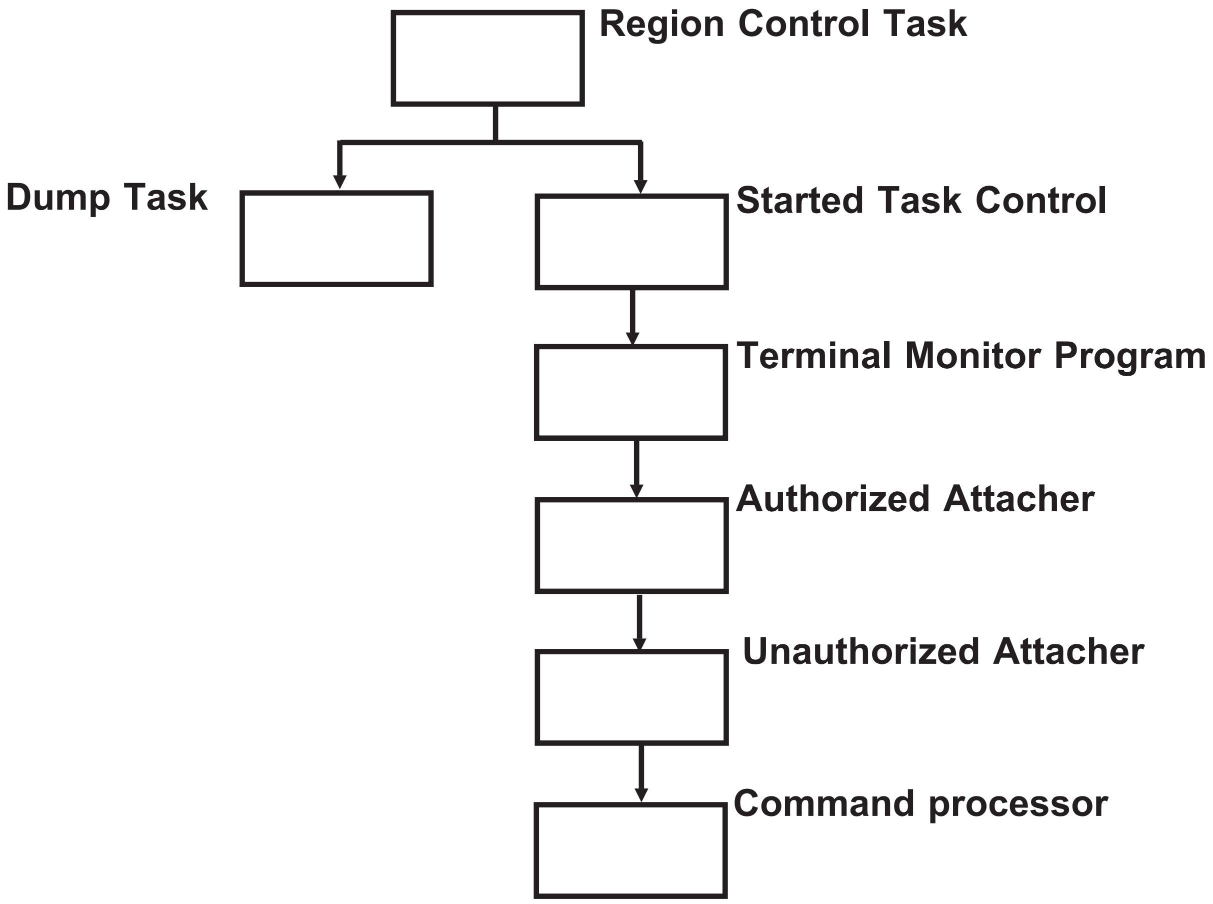
  - ♦ If the command processor is a non-authorized program, another subtask is attached to attach the command processor(!)

- ❐ A command processor may, in turn, attach other subtasks

  - ♦ The basic task structure is shown on the next page ...

TSO Task Structure

❒ So, when an unauthorized command is being processed, the TCB hierarchy looks like this:



❒ For an authorized command processor, you will not see the "Unauthorized Attacher" task

❒ In some situations, other combinations are possible

TSO Command Processors

❒ **In some situations the Command processor will attach subtasks; here are some widely used command processors ...**

- ♦ **TSO CALL command processor attaches the program being called; this is an executable program**

- ♦ **The EXEC command processor interprets REXX or CLIST scripts**

  - ✗ When a script contains a TSO command, the appropriate command processor is attached in turn

- ♦ **THE ISPSTART command processor brings up ISPF**

  - ✗ ISPF is the TSO application most programmers are familiar with

ISPF - Integrated System Productivity Facility

❒ **This facility provides a full-screen front end to TSO, providing a friendlier interface than native TSO, which is just a command line interface**

❒ **ISPF is a <u>dialog manage</u>r**

- ♦ **Providing full screen prompts or fill-in-the-blanks panels**
- ♦ **Accepting and processing data from the panels, doing any necessary behind-the-scenes work, and returning results to the user**
- ♦ **The actions may be driven by REXX execs, CLISTs, or compiled programs**

ISPF, continued

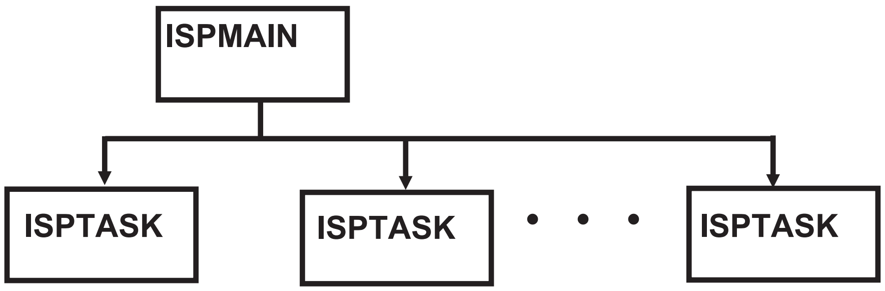
❒ **ISPF supports "split screen", whereby up to 32 separate tasks can be going on**

- ♦ **The ISPMAIN task attaches a copy of ISPTASK for each split screen**

- ♦ **Coordination among screens is done by ISPMAIN**

  - ✗ So that even though you may be highly multi-tasked, only one task is actually being executed at any one time

❒ **So the TCB hierarchy of the ISPF portion of an address space might look like this:**



❒ **Each of the ISPTASK TCBs represents a split screen, and may be doing any of the myriad tasks possible under ISPF**

- ♦ **Edit, view, browse, working with data sets, looking at job output, and so on**

ISPF, continued

❐ Note that in split screen mode, only one task can actually be doing work

- ♦ Each task (screen) can be swapped to, and the user can work in that screen

- ♦ When another screen is swapped to, the previous screen is inactive until control swaps back to it

Batch TSO

❒ **Finally, you can run TSO in the batch(!)**

- ♦ **Just submit a jobstep that executes program IKJEFT01**

❒ **Of course, there's no real interaction, but you can bundle a stack of commands and have them run with no user intervention**

- ♦ **Input from the terminal is simulated by the TMP reading from a DD statement of SYSTSIN**

- ♦ **Line-oriented output to the terminal is simulated by writing to a DD statement of SYSTSPRT**

- ♦ **You can run REXX execs and CLISTs in the batch, which is very useful when you have a script that might run a long time and doesn't need any interaction**

  - ✗ Which, come to think of it, is the essence of batch work

❒ **Since ISPF is just a task under TSO, one of the commands you can pass to a batch TSO session is ISPF**

- ♦ **Of course, displays are not useful here, but there are a whole raft of ISPF services that are very useful in batch, so many programmers construct jobs to run ISPF in the batch this way**

Language Environment

❐ **Part 5: The Influence of Language Environment**

- ♦ **What is Language Environment?**
- ♦ **The Impact of Language Environment**
- ♦ **LE Initialization**
- ♦ **Major Control Blocks with LE Initialized**
- ♦ **The Impact of Language Environment, concluded**

What Is Language Environment?

❒ **Language Environment (also called just "LE") is a component of z/OS that has an enormous impact on applications**

❒ **It was added to the operating system with an eye towards providing support for what was then called "Open Edition MVS" (now called "UNIX System Services for z/OS")**

- **But LE provides services for a broader audience of users than just C/C++ users using UNIX**
  - ✗ Non-UNIX applications, for example, written in any language
  - ✗ Especially applications developers with components written in multiple languages

What Is Language Environment, continued

❐ **LE ...**

- **Provides a single run-time environment for multiple languages**

    - ✗ Previously, applications that included programs written in a mixture of languages always had conflicts since each language wanted to establish its own environment at run time

    - ✗ Now, all supported high level languages from IBM use this single run-time package

    - ✗ Assembler programmers can also take advantage of LE if they wish (need to code in an "LE-conforming" style)

- **Provides a suite of useful callable services**

    - ✗ Dynamic storage handling, user-written condition handling, math services, international services (formatting of dates and times, for example, and the use of locales), and many more

        - ➢ All LE routine names begin with "CEE" (for Common Execution Environment)

- **Supports UNIX System Services for z/OS**

    - ✗ "Generally, that which could be reasonably done in LE was, and that which was either very difficult in LE or could not be done there was done in the [UNIX] kernel [address space]."
      - Bill Schoen, IBM Open
      Edition Developer

The Impact of Language Environment

- ❒ **If you compile using a supported IBM compiler (C, C++, COBOL, PL/I) the resulting object code will include a LINK to LE initialization**
    - ♦ **If you use Systems Programming C, SAS/C, Dignus C, or the SYSTEM option of PL/I you can request not to get LE initialization (but the intent of those options is to work with systems level code, not applications level)**
    - ♦ **If you code in Assembler, you have a choice of creating LE-conforming Assembler or not**
        - ✗ If you code non-LE-conforming Assembler, you can't use any of the LE-provided facilities and services
            - ➢ But you can use many (not all) of the callable UNIX services (more later)
- ❒ **Old load modules, compiled under earlier, pre-LE versions of the compilers, will still work, so you don't have to have an LE environment established**
    - ♦ **But then you can't use any of the LE-provided facilities and services**

The Impact of Language Environment, continued

❒ **LE initialization routines will build an environment to support the LE callable services**

- **Remembering that contents management and task management need to work as before, so the existing control block structure needs to be supplemented, not replaced**

❒ **LE has a program management model that consists of four levels**

Region - collection of processes

Process - collection of enclaves

Enclave - program(s) and resources for running thread(s)

Thread - a task

LE Initialization

❒ **The LE initialization process has many steps, including**

- **LOADing CEEBINIT and CEEPLPKA (this last is the bulk of LE, containing many entry points and functions)**

  ✗ So, build LLEs, CDEs, and XTLSTs, and load the modules

- **Region initialization - although only really relevant in CICS applications, every LE environment has a top-most level Region Control Block (RCB) to manage processes**

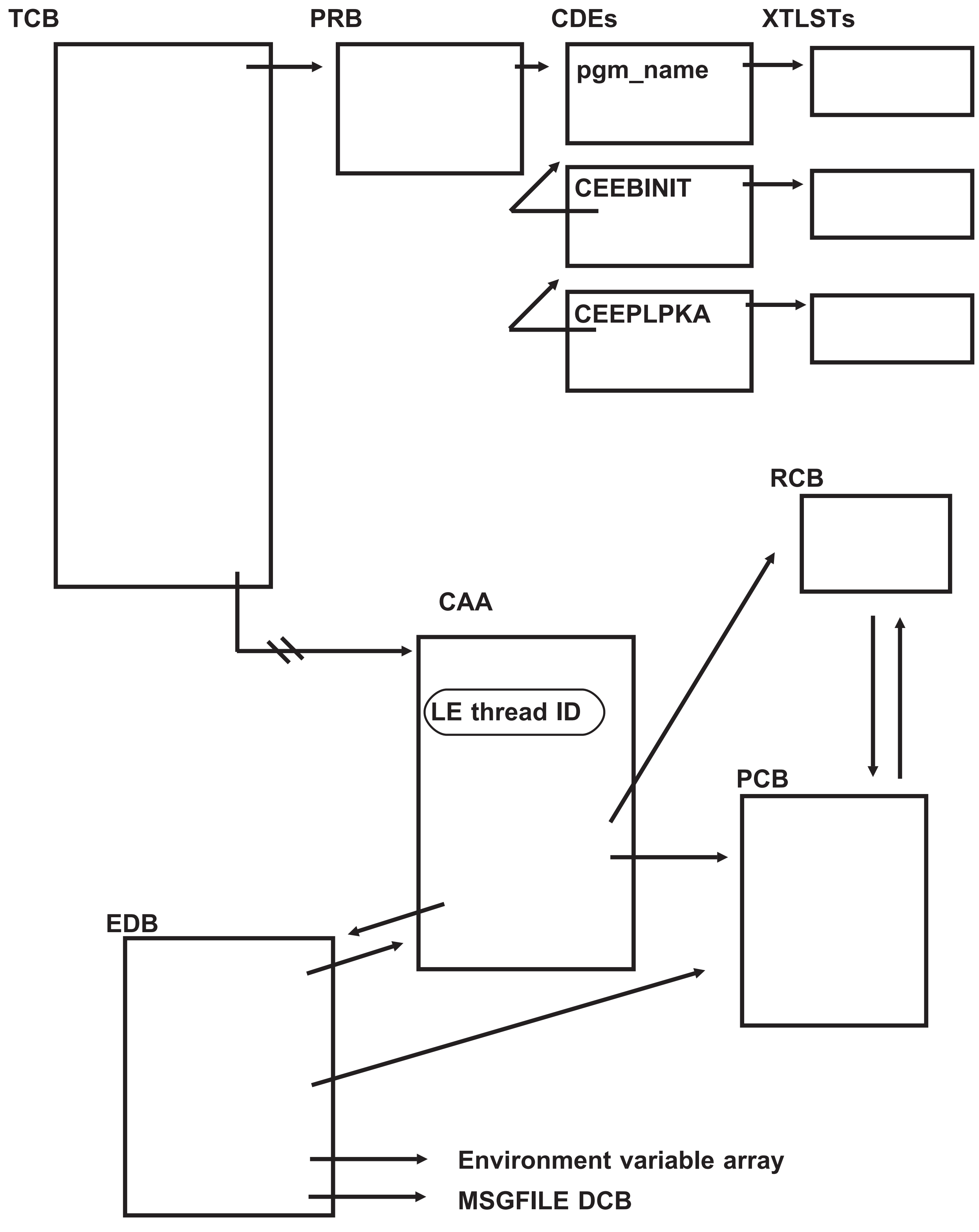
- **Process initialization - for online applications, think transaction; for batch a process maps to an address space; a process contains one or more enclaves, and is represented by a Process Control Block (PCB)**

- **Enclave initialization - each enclave is represented by an Enclave Data Block (EDB) and it provides resources for threads, including heap storage, run-time options block, set up for environment variables; create and open a DCB for the message file**

  ✗ The load module is examined for language specific signature CSECTs and the initialization routines are run for each language found in the executable

- **Thread initialization - a thread is represented by a Common Anchor Area (CAA); thread initialization includes acquiring a stack (storage for the save area chain), and establishing two entry points in CEEPLPKA as ESPIE and ESTAE routines (unless run-time options override this behavior)**

Major Control Blocks with LE Initialized



Another Look at Language Environment

❒ **To better understand the LE terminology, you might find this approach useful:**

- ♦ **A thread is the basic unit of work, of dispatchability**
  - ✗ A thread is a task, that is, the running of a program, enhanced with its own condition manager and certain thread-level resources such as a stack (save area chain)
- ♦ **An enclave is the basic unit of contents management: for each main program, there is an enclave**
  - ✗ The enclave includes the main routine, all its subroutines, heap storage used by the main and its subroutines, run-time settings, and so on
- ♦ **A process is a higher level of contents management that allows for sharing resources among all enclaves**
  - ✗ For example, the message file is available for use by all enclaves in a process
- ♦ **A region is the highest level of resource management**
  - ✗ Primarily sharing the run-time library among all processes in the region
  - ✗ This allows different regions to use different run-time libraries, perhaps, for example, due to release dependencies

The Impact of Language Environment, concluded

❒ **Now, as the program runs (the thread executes), any program in the enclave can request (call / invoke)**

- ♦ **All callable LE services**
- ♦ **Most C functions (depending, for example, if installation has opted to pay for the C/C++ run-time or not)**
- ♦ **All UNIX services (if UNIX System Services installed above the minimum level)**

❒ **In particular, LE provides the support for**

- ♦ **Creating / updating / accessing environment variables**
- ♦ **Invoking DLL services**
- ♦ **Dynamically invoking routines from PDSEs or the HFS**
    - ✗ Since things still have to work in the context of existing structures, if an executable is loaded from a PDSE or the HFS, the CDE for the executable indicates there is a system-maintained hash table entry available to find the full name of the executable (CDEs only have room for 8 byte names)

❒ **If an error occurs ("a condition is signaled" in LE parlance) ...**

- ♦ **The Language Environment-specified ESPIE or ESTAE routines get control, and they may offer the condition up to installation-written condition handling routines**

This page intentionally left almost blank.

UNIX

❒ Part 6: The UNIX Model

- ♦ The UNIX Model — Introduction
- ♦ The UNIX Standard(s)
- ♦ UNIX — The Operating System
- ♦ UNIX User ID
- ♦ UNIX Group ID
- ♦ The UNIX Program Management Model
- ♦ The UNIX File Model: the Hierarchical File System (HFS)

The UNIX Model - Introduction

- ❒ **At this point of this exploration, I want to step away from the control block approach to things**

- ❒ **Here I want to explore the UNIX way of looking at computing - concepts and terms as UNIX users use them**

- ❒ **Then in the next, and final chapter, we'll put everything together to explore how IBM has mapped the UNIX paradigm on top of the z/OS implementation already in place**

  - ♦ **An approach that sometimes works very well and other times doesn't work well at all**

    - ✗ But over all, a pretty amazing job

- ❒ **But first, UNIX ...**

- ❒ **UNIX has its history intertwined with the old Bell Labs, part of AT&T, back in the late 1960's and early 70's**

  - ♦ **If you are interested in a detailed history of UNIX, go the web and start here:**

    - ✗ http://www.bell-labs.com/history/unix

The UNIX Standard(s)

- **One of the reasons UNIX has been so successful is the ongoing process of defining a standard**
  - **So that vendors can have their products validated by the standard**
    - So that customers will know there is at least a minimum level of consistency across UNIX products
  - **Of course, each vendor offers extensions to the standard, so they can differentiate their product in the marketplace**
  - **But at least there is a common starting point**
- **Unfortunately, there is more than one common starting point(!)**
  - **The X/Open company's most recent standard is called XPG4**
  - **The IEEE's (Institute for Electrical and Electronic Engineers) most recent standard is called POSIX.2**
- **Fortunately, there is a consortium of companies, called The Open Group, that have developed what is called the Single UNIX Specification, combining these two standards**
  - **The Open Group is accessible on the web at**
    - http://www.opengroup.org
  - **You can download the specification in a variety of formats from this site**

UNIX - The Operating System

❐ **So, IBM mainframers should now temporarily put aside their previous vocabulary of terms and concepts; we'll tie it all together later**

❐ **"UNIX" is not an acronym**

- ♦ **Although it is correctly spelled in all capital letters**

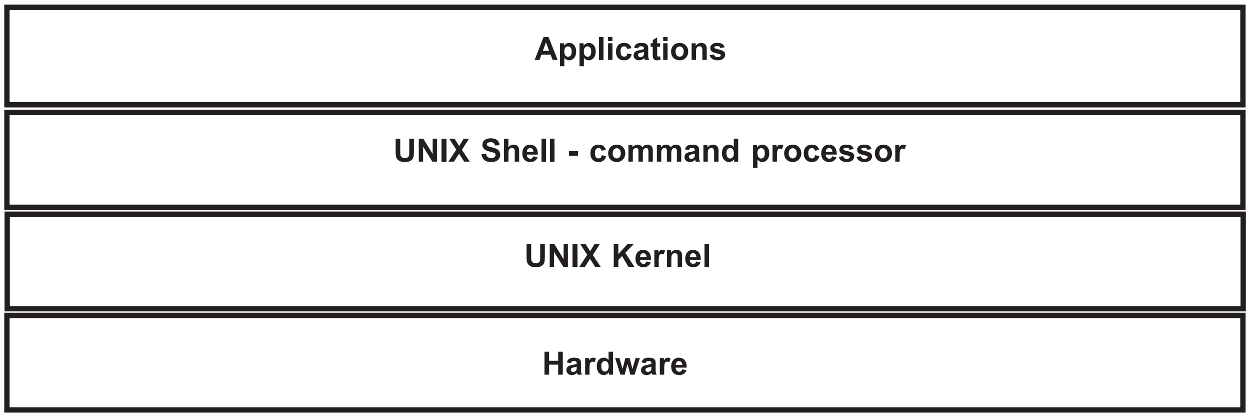
❐ **The basic underlying philosophies of UNIX are these:**

- ♦ **There are three major software components**

  - ✗ The kernel, the basis of the operating system, that provides services for managing the underlying hardware and the scheduling of work (processes)

  - ✗ The shell, a program that accepts, interprets, and processes commands (user or program requests); you have more than one shell to choose from

  - ✗ Applications software: tools, utilities, other programs



UNIX - The Operating System

❒ **The basic underlying philosophies of UNIX are these (continued):**

- ♦ **The standard way for a user to interface to the system is through a terminal**

  - ✗ Historically, a teletype terminal ("tty"), which was a primitive keyboard / printer, with no video screen

- ♦ **Commands are the primary way to get work done for users**

- ♦ **Commands are implemented as programs**

- ♦ **Programs should handle text streams, rather than machine-dependent data types and structures**

- ♦ **Programs should be considered tools that can be combined to solve complex problems**

  - ✗ The connection between these tools is the pipe (discussed later)

- ♦ **Program, command, variable, and file names should be case sensitive**

  - ✗ For example, a command named Cmd is not the same as a command named CMD

- ♦ **There are no arbitrary restrictions on the maximum length of names: any such maximums are implementor-defined**

UNIX User ID

- ❒ **A user of a UNIX system must have a UNIX user ID**
    - ♦ **In the literature, the term "UID" is used to specify a UNIX user ID**
- ❒ **UID's are assigned by a system administrator, and must follow these rules**
    - ♦ **A <u>UID</u> is a positive integer (maximum value is implementation dependent)**
    - ♦ **A <u>Userid</u> is a name that is known to the system for login purposes**
        - ✗ Case sensitive with a maximum length determined by the UNIX implementation
    - ♦ **A user (person) may have multiple Userids (user names), each with one UID; and a given UID may be assigned to multiple Userids (although this is not recommended)**
    - ♦ **UID of 0 (zero) is special: a** <u>superuser</u>
        - ✗ Users with superuser status can access and change all system resources
        - ✗ A UID of 0 may be assigned to multiple users
- ❒ **Any given UID may be logged into UNIX multiple times at once**

UNIX User ID, 2

❐ **A** user database **is maintained that contains at least the following for each user:**

- **Userid (user name)**
- **UID**
- **Initial GID (see next page)**
- **Initial working directory (if omitted, default is implementation defined)**
- **Initial user program (if omitted, default is implementation defined; usually a shell program)**

UNIX Group ID

- ❒ **Every UNIX system also has a set of groups**
  - ♦ **Collections of related UIDs, authorizations, and permissions**
- ❒ **Each group is identified by a positive integer (maximum value is implementation dependent)**
  - ♦ **This is called the** <u>Group ID</u> **(sometimes "**<u>GID</u>**")**
- ❒ **Every group also has a** <u>Group name</u> **(alphanumeric)**
- ❒ **Every user belongs to at least one group (perhaps a department or an application group) and often many groups**
  - ♦ **Access to files is at least partially controlled by what GID(s) a UNIX user belongs to**
- ❒ **The system maintains a** <u>group database</u> **that contains at least the following for each group:**
  - ♦ **Group name**
  - ♦ **GID**
  - ♦ **List of users in the group**

The UNIX Program Management Model

- ❒ **In the UNIX sense of the word, a "program" is either of these**
  - ♦ **An executable program (called a "binary" sometimes)**
  - ♦ **A shell script**

- ❒ **A** process **is a collection one or more threads, along with the resources the threads are using**
  - ♦ **Memory / storage**
  - ♦ **External files**
  - ♦ **Control blocks**
  - ♦ **Environment variables and options (defined shortly)**

- ❒ **A** thread **is a single flow of control within a process - the execution of a program**
  - ♦ **Each thread has its own** thread ID
  - ♦ **All threads in a process have access to the same memory items**

- ❒ **An** address space **is the set of memory locations that can be referenced by a process or the threads in the process**

The UNIX Program Management Model, p.2.

❐ **Some useful terms**

- **Environment variable - named location in memory containing a value that influences the behavior of a process**

  - ✗ Example: the **PATH** environment variable contains the default search path for programs

- **Environment option - named switch indicating restrictions / permissions for the current environment**

  - ✗ Example: **noglob** if set on prohibits the use of wildcard characters in file names used in commands

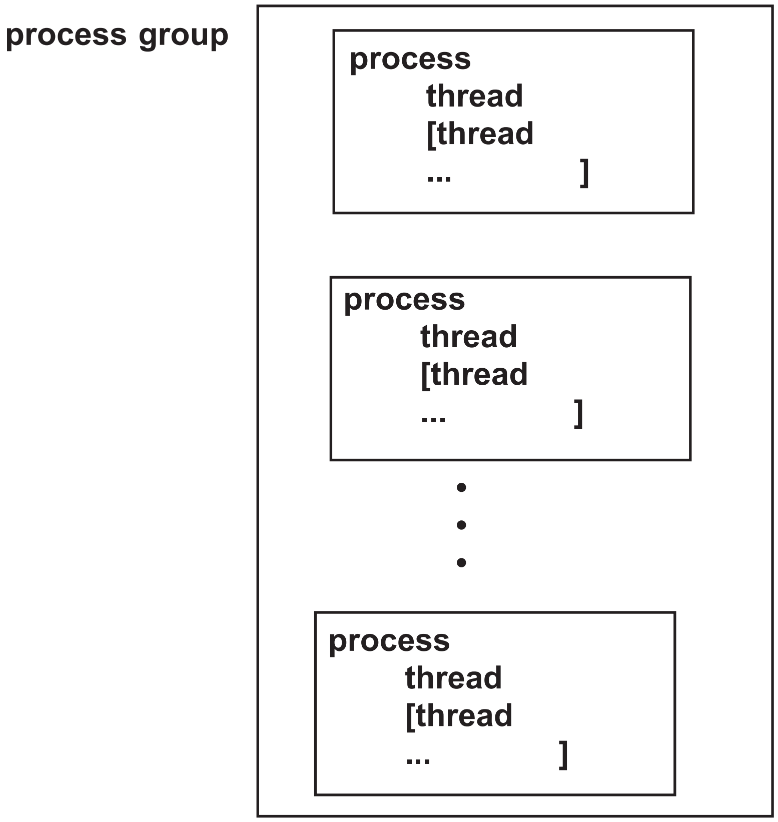
- **Shell script - file containing UNIX (kernel) commands, shell commands, and possibly other script names; invoked by simply naming the file**

The UNIX Program Management Model, p.3.

❒ **Every process has a number that uniquely identifies it: a** process ID **(**PID**)**

❒ **Every process belongs to a** process group**, which also has a unique identifying number, the** process group ID **(**PGID**)**

- **Note that a process group may have a single process**
- **All the processes in a group can communicate with each other**
- **One of the processes in a process group is the** process group leader**; the PID of the process group leader = the PGID of the group**



The UNIX Program Management Model, p.4.

❒ **When a process creates another process, the creating process is called the** parent **process, the created process is called the** child **process**

- **A process creates a child process using the** *fork()* **service**
- **Every process has a** Parent Process ID **(**PPID**) which identifies its parent process**

The UNIX Program Management Model, p.5.

❒ **When a user logs in to a UNIX system, the login process ...**

- ♦ **Creates a pseudo-terminal (also ptty or, sometimes, pTTY)**

  - ✗ This provides a focus for terminal input and output for the current session

  - ✗ A pseudo-terminal is used because UNIX supports multiple sessions per terminal, so each session will get its own pseudo-terminal

- ♦ **Establishes links to the ptty: file descriptors 0, 1, and 2 are tied to files known as** stdin, stdout**, and** stderr**, respectively**

  - ✗ A file descriptor is a UNIX handle to relate to an open file; details beyond the scope of this document

- ♦ **Creates a process (allocates storage and sets initial values for environment variables and options)**

- ♦ **Runs a profile script (this sets user-specific environment variables and options; these will override any system level default values for same-named variables and options)**

  - ✗ The initial shell is called a login shell because it runs the profile script; subsequent sub-shells (shells invoked from the current shell) do not normally run the profile

- ♦ **Loads a shell program which reads from** stdin**, waiting for commands from the user**

The UNIX Program Management Model, p.6.

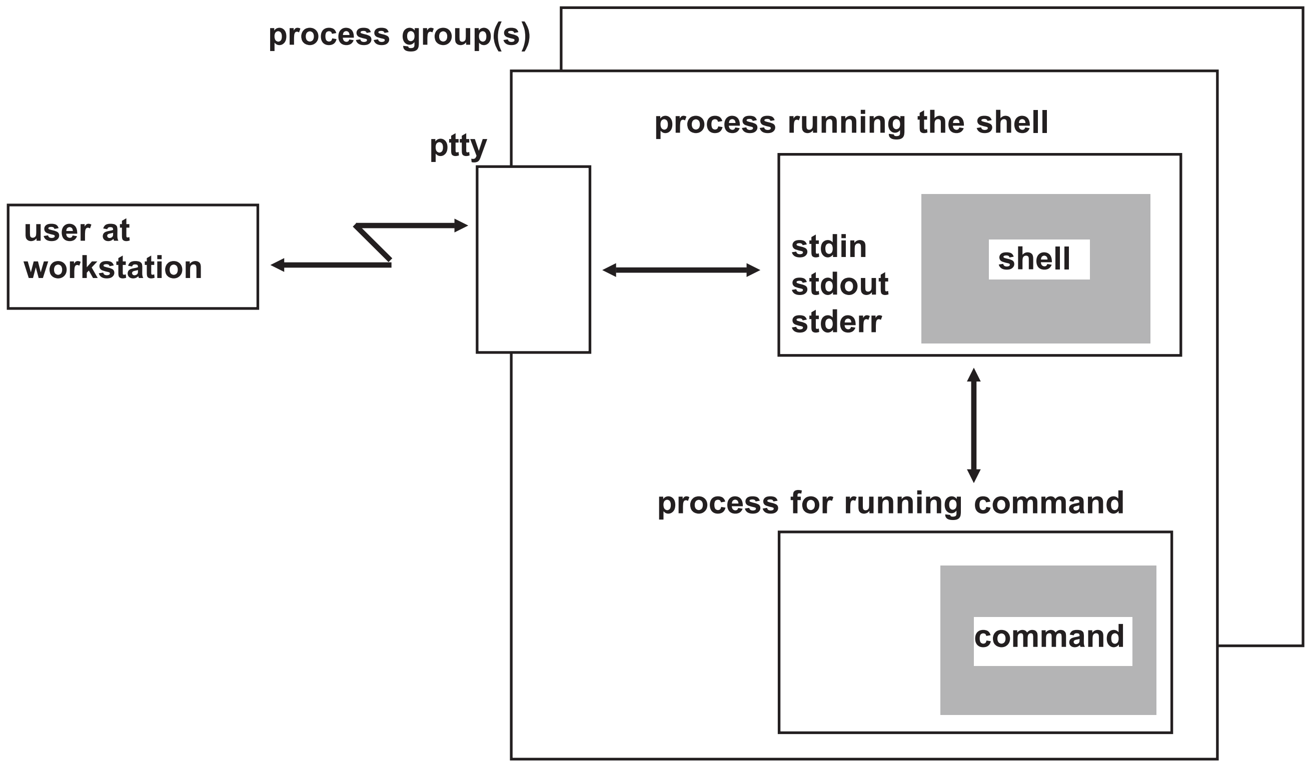
- ❐ **A shell, as mentioned before, is a command processor**
  - ♦ **It accepts a command from the user and parses it into its parts, then finds the command and runs it**
    - ✗ A command may be a built-in command or an external command; for non-built-in commands there is a well-defined search process
    - ✗ Each non-built-in command is run in a separate process
  - ♦ **If the command being run is a script, it may contain invocations of shell commands, subcommands, or further programs or scripts**
- ❐ **The shell can pass parameters to the invoked command, doing variable substitution as needed**
- ❐ **Normally, any "printed" (usually displayed) output from a command goes to** stdout, **which is usually tied to the terminal**
- ❐ **Similarly, any error messages go to** stderr, **which is also usually tied to the terminal**
- ❐ **If a command requires user input, by default it gets it from** stdin, **which is usually the keyboard**

The UNIX Program Management Model, p.7.

❒ If a command is a shell built-in command, the command is executed in the shell's environment

❒ But if the command is not a built-in, the shell creates a new process to run the command in

❒ Pictorially ...



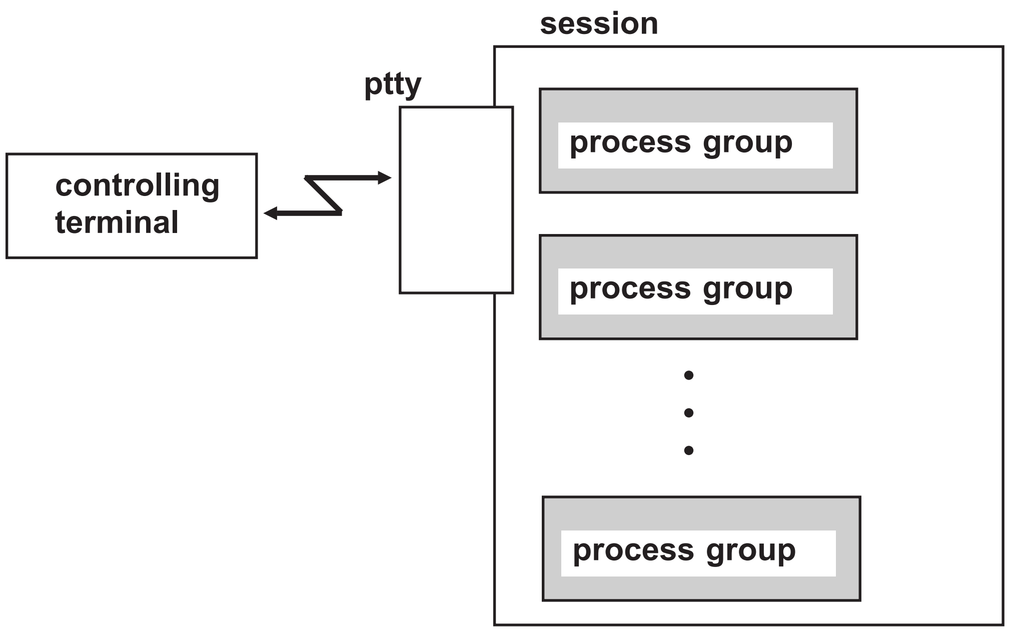
❒ Notice that running a command in a separate process protects the invoking process from damage

- For example, environment variables in the parent process are inherited by the child process, but if the child process changes the values, the new values are not reflected back in the parent's address space

The UNIX Program Management Model, p.8.

❒ **A session is a collection of process groups established for job control purposes**

- **Every session has an identifier, the session ID (SID)**
- **Each process group is a member of exactly one session**
  - ✗ Each process in a process group is considered to be a member of the session
- **The process that creates a session is the session leader; once the session leader process ends, no other process in the session can communicate with the ptty**
- **One process group in a session may be the foreground process group, all other process groups in the session are background process groups**
- **A session may have a controlling terminal; if so, every process in the session has that terminal as its controlling terminal**



The UNIX Program Management Model, p.9.

- **By issuing a** setpgid() **command, a child process can start its own process group in the same session**

- **By issuing a** setsid() **command, a process starts a new session; the process issuing this command will be the session leader of the new session and no longer a member of the old session**

  - **A process relinquishes its controlling terminal when it creates a new session**

    - Other processes remaining in the old session that had that controlling terminal continue to have it

The UNIX Program Management Model, p.10.

- **So initially, at login, there is one session with a single process group containing a single process running a single thread**

- **If that process does a** *fork()*, **it creates a child process in the same process group**

    - **If it does multiple** *fork()***s, you have multiple processes in the same process group**

    - **If any child process does a** *fork()*, **its children, too, belong to the original process group**

- **When a process issues a** *setpgid()*, **that creates a new process group with a single process, in the current session**

    - **From now on, if that process issues** *fork()*, **these children (and their descendants) belong to the new process group**

- **If a process issues** *setsid()*, **that starts a new session with a single process group containing a single process running a single thread**

    - **The new session starts out without a controlling terminal**

    - **Whether it is possible to obtain a controlling terminal is implementation dependent, but if it is possible it is done by issuing** *open()* **to a terminal device by the process that is the session leader**

The UNIX Program Management Model, p.11.

❒ **You may enter a string of commands connected using various special symbols, for example (we only show three commands here, but it can be any number):**

```
command  ; command  ;  command
```

- **execute the commands one after the other**

```
command  ||  command  ||  command
```

- **run the second command only if the first command fails; run the third command only if the second command fails**

```
command  &&  command  &&  command
```

- **run the second command only if the first command is successful; run the third command only if the second command is successful**

```
command  &&  command  ||  command
```

- **run the second command only if the first command is successful; run the third command only if the second command fails**

```
command  ||  command  &&  command
```

- **run the second command only if the first one fails; run the third command only if the second command is successful**

The UNIX Program Management Model, p.12.

❒ **You can code a command string including the use of pipes, for example:**

```
command | command | command
```

- **run the first command, having output of that command become input to the second command**
- **then run the second command, having output of that command become input to the third command**
- **then run the third command**

❒ **This works because every command uses** *stdin* **for input and** *stdout* **for output, so the pipe tells UNIX (or the shell, more accurately) to send the** *stdout* **of one command to the** *stdin* **of the next command**

The UNIX Program Management Model, p.13.

- **In all of the situations on the previous two pages, the command string will be run in the foreground process group, and your terminal will wait until the last command is done**

- **If you put an ampersand (&) at the end of your command string, the shell will create a new process group for this command string and run the string of commands in the background**

  - **For example:**

```
command  |  command  |  command  &
```

  - **create a background job (process group) to run this series of commands**

- **You can do this even if you only have one [long running] command to run**

- **Once you enter the command, the shell sets up the background job and returns to your foreground process group**

  - **Now you can enter more commands, and any background jobs you've started are running independently of your foreground work**

The UNIX Program Management Model, p.14.

- **Note that a pipelined command string job will set up multiple processes that run simultaneously, coordinating inputs and outputs as necessary**
  - **That is, the system will not wait for one process to end before starting the next process**
  - **They are all started at once and downstream processes sit waiting for input from upstream processes**
    - And data is processed and passed on as soon as it is available

The UNIX Program Management Model, p.15.

- **Instead of always using** *fork()* **and creating a new process, there is an option of using** *exec()*

  - **The** *exec()* **service essentially reuses the current process but runs a different program**

    - ✗ Thus saving some of the overhead of creating a whole new process

  - **The shell will sometimes do this for the last command of a pipeline, or connected command string, although this is implementation dependent**

The UNIX Program Management Model, p.16.

❒ **The POSIX standard introduced multi-threading**

- ♦ **The ability to take the work of a process and divide it up into multiple, independently dispatched units of work that run in paralllel**
- ♦ **The central concept here is of a** <u>POSIX thread</u>**, or** <u>pthread</u>

❒ **There are three kinds of pthreads described**

- ♦ **<u>Heavy weigh</u>t pthreads - each pthread competes for the CPU at the same level, independently**
- ♦ **<u>Medium weight</u> pthreads - there is a pool of work to be done on pthreads but only a subset of them will run at one time; the others must wait until some other running pthread in the process finishes**
- ♦ **<u>Light weight</u> pthreads - the system dispatching is not involved here: user code determines what code runs when**

❒ **There are a large number of services defined for working with pthreads, includng** *pthread_create()***, which creates a pthread**

The UNIX Program Management Model, p.17.

❒ **The actual implementation details are left to the individual software developers, recognizing their dispatching routines may have differing granularity:**

- ♦ **Process level only dispatching**
- ♦ **Thread level only dispatching**
- ♦ **Hybrid dispatching: threads within processes**

The UNIX File Model: the Hierarchical File System (HFS)

- ❐ **We spend a little time on HFS here, since it will impact some notes in the next section**

- ❐ **Each UNIX filesystem contains <u>files</u>, <u>directories</u>, and an <u>inode table</u>**

  - ♦ **Each file is given an <u>inode number</u>, unique within its filesystem**

    - ✗ A file's inode table entry contains its inode number and information about the file (location, size, *etc.*), but not its name

  - ♦ **Each file also has a <u>device number</u> (or <u>file serial number</u>), and the combination of its inode number and device number are unique throughout all the file systems in the UNIX system the file is mounted in**

    - ✗ We make these distinctions because it is possible for a remote UNIX system to <u>mount</u> all or part of its filesystem to appear to belong to the local filesystem

  - ♦ **A <u>directory</u> is a file that contains a list of filenames and their corresponding inode numbers**

  - ♦ **If a file found in a directory is itself a directory, we say it is a <u>subdirectory</u>**

Hierarchical File System (HFS), continued

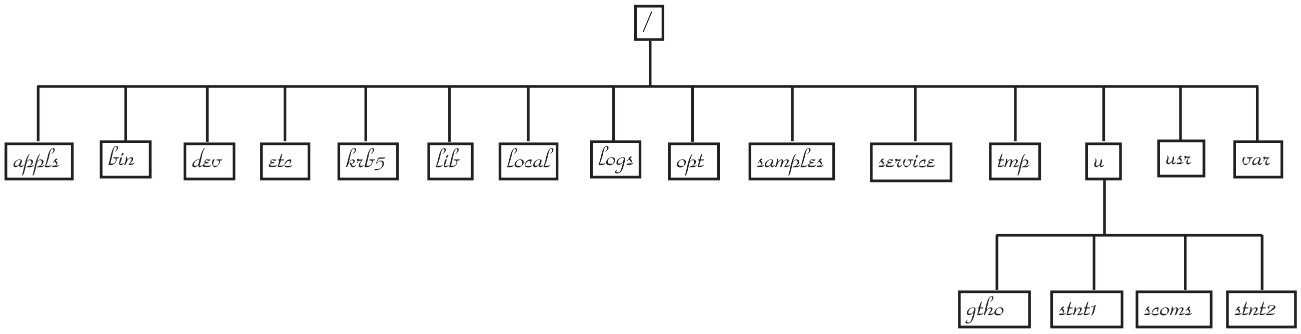
❒ **UNIX supports these kinds of files:**

- **Regular files (for data or programs)**
- **Directories (for other directories and filenames)**
- **Device files (all devices look like files)**
  - Character special files
  - Block special files
- **Link files (aliases and pointers to other files)**
  - Hard links
  - Symbolic links
  - External links
- **Pipe files (files used to transport data between programs)**
- **Sockets (used for network communications)**

Hierarchical File System (HFS), continued

❒ **UNIX organizes directories, subdirectories, and files in a hierarchy, that can be described as an upside-down tree**

- ♦ **The top of the hierarchy is called the** <u>root</u> **and is indicated by a slash ("/")**
- ♦ **The hierarchy of directories and subdirectories looks something like this:**



❒ **Each of these directories can have subdirectories which can, in turn, have subdirectories, and so on**

- ♦ **Ultimately you get down to files**
- ♦ **Of course, a directory may include both files and subdirectories**

Paths

❒ **The route to get to a file is called the** path**: the absolute or relative specification of the location of a file in the hierarchy**

❒ **A user running a UNIX session is always assumed to be "in" some directory - the** current directory **(also:** working directory**, also** current working directory**)**

- **Also, each user has a** home directory **specified in their profile in the user database**

- **At login time, the home directory is established as the current directory**

❒ **An** absolute path **always begin with the root directory, for example: /u/scoms/cbl/cobex01**

❒ **A** relative path **begins with the current directory and specifies subdirectories down to the file**

- **For example if the current directory were /u/scoms then you could specify cbl/cobex01 to get to the same file as in the example above**

Paths, 2

- In many situations, you specify a path as a list of directories to search to find a file, the entries in the list separated by colons (:)
    - The system performs pathname resolution to resolve a pathname to a particular file in a file hierarchy
    - The system searches from entry to entry in your path list in order to find the file
    - This search pathname can be set and changed using commands, although the details are beyond the scope of this document

- For example, you may set your search path to be '/bin:/usr/bin:/usr/etc'
    - Then search will look first in /bin
        - then in /usr/bin
            - then in /usr/etc

UNIX System Services

❒ Part 7: UNIX on z/OS

- UNIX on z/OS
- Dubbing
- z/OS Control Blocks for UNIX System Services
- Processes
- pthreads
- Running Under OMVS
- The BPXBATCH Utility
- z/OS UNIX Control
- I/O Issues
- Practical Applications

UNIX on z/OS

- ❐ **z/OS UNIX System Services provide a UNIX environment**
  - ♦ **which passes the validation tests to be certified as conforming to the POSIX 1003.2 and XPG4 1995 standards**

- ❐ **Users running on z/OS can access UNIX services and files and / or traditional services and files**
  - ♦ **Run programs in batch - traditional or under a non-interactive shell**
  - ♦ **Run programs and commands, including REXX execs, in traditional TSO and TSO/ISPF**
  - ♦ **Interact with the system under traditional TSO READY or TSO/ISPF**
  - ♦ **Interact with the system under an interactive shell - from TSO READY or TSO/ISPF**

- ❐ **Note that currently CICS and IMS are explicitly excluded from using UNIX System Services**

UNIX on z/OS, 2

- **There are many pieces that collectively support this facility**
  - **The security interface, which supports security packages such as RACF, ACF2, Top Secret, and so on, is used to supply the user database and group database functions**
    - Groups and users must be defined in a UNIX segment in order for users to use UNIX System Services
  - **HFS files are supported, for system functions and user functions**
    - Typically each user will have their own mountable filesystem; access to HFS files is supported through JCL, commands, and many utilities
  - **Language Environment**
    - LE provides support for many UNIX capabilities, such as: environment variables, ability to work with executables with long names and case-sensitive names, DLL support, locales support
      - LE also makes this support available even to programs that don't use UNIX services
  - **A UNIX kernel address space**
    - Keeps track of all UNIX processes going on and provides services in support of these processes

UNIX on z/OS, 3

- ❒ **There are many pieces that collectively support this facility, continued**

  - ♦ **TSO commands**

    - ✗ For managing HFS constructs (create directories and subdirectories, for example; MKDIR, MKNOD; MOUNT, UNMOUNT, OSTEPLIB)

    - ✗ For transferring data between the traditional file structures and HFS, where possible and appropriate (OCOPY, OGET, OGETX, OPUT, OPUTX)

    - ✗ For invoking shell scripts, commands, and executable files (BPXBATCH, OSHELL)

      - ➢ Notice, by the way, that components whose names begin with the letters "BPX" are, generally speaking, part of UNIX System Services

    - ✗ To invoke the z/OS shell, which is based on UNIX System V shell with some features of KornShell (OMVS)

      - ➢ Or you can run the tcsh shell, which is an enhanced version of the Berkley UNIX C shell called csh

    - ✗ To edit or browse HFS files (OBROWSE, OEDIT)

    - ✗ To provide an ISPF-like interface (the ISPF shell) for doing support work (ISHELL)

    - ✗ To provide access to online publications (OHELP)

UNIX on z/OS, 4

- **There are many pieces that collectively support this facility, continued**

  - **Extensions to REXX to access UNIX services**

    - Two new address environments, SYSCALL and SH, are available; using SYSCALL commands, you can issue many UNIX commands from an exec running under a shell, in batch, or under native TSO

  - **A large set of callable services (hundreds of services), BPX... routines**

    - These can be called from Assembler programs (LE-conforming or not) or high level languages

      - However, the functions these services implement are often available as part of the C/C++ library, so the C/C++ user would typically use the C/C++ library function, which will issue the appropriate service call on behalf of the user

      - But the COBOL or PL/I user who wants to use these services can call the services directly

UNIX on z/OS, 5

❒ **The objectives of the IBM UNIX System Services development team are pretty focused:**

- ♦ **"The full set of UNIX services are defined and intended for LE enabled C/C++ programs"**

❒ **But because so much of the UNIX capability is embedded in LE ...**

- ♦ **And because all high level languages compiled with supported compilers generate LE-enabled code**
  - ✗ Which allows, for example, COBOL, PL/I, and [LE-conforming] Assembler programs to call most C library functions directly (that is, there is no need for a whole C subroutine)
- ♦ **And because the non-LE UNIX services are available as callable subroutines to just about anybody ...**
  - ✗ The full set of UNIX services is pretty much available to all programs and programmers

❒ **And because all programs and scripts are, ultimately, running under z/OS ...**

- ✗ The full set of z/OS capabilities is available to all programs and programmers

UNIX on z/OS, 6

- ❒ So you can create programs that ...

    - ♦ Are not LE-enabled and do not use UNIX services (Assembler or non-supported compilers only)

    - ♦ Are LE-enabled and do not use UNIX services (LE-enabled Assembler and all supported high level languages)

    - ♦ Are not LE-enabled but do use UNIX services (Assembler or non-supported compilers only)

    - ♦ Are LE-enabled and do use UNIX services (LE-enabled Assembler and all supported high level languages)

- ❒ Programs that use UNIX System Services may or may not run under a shell program (for example, they may run in batch or under TSO)

    - ♦ However, traditional UNIX programs expect to be running under a shell

- ❒ Programs can access data on the HFS or in traditional z/OS files

Dubbing

❐ **When a program is run in z/OS, either batch or TSO [with or without ISPF], it starts out like any traditional program in a traditional z/OS address space**

- ♦ **If the program is LE-enabled, LE initialization takes place and the program now has access to all the LE services we've talked about and alluded to**

  - ✗ An LE thread ID is assigned to the task, which is still a z/OS task

- ♦ **If the program calls no UNIX services, then there is no new behavior**

Dubbing, 2

- **But the first time the program calls any z/OS UNIX service, the call is sent to some routine in the UNIX kernel address space**
  - ✗ At this point, the task becomes dubbed, which means the UNIX kernel is now aware of the task as a UNIX thread
  - ✗ As a minimum, dubbing causes z/OS UNIX to ...
    - ➢ Obtain the user UID and the GID for the initial group the user belongs to, and locate the initial directory
      - ➤ Note that if there is no UNIX segment defined for the user in the security software, the request will fail
    - ➢ Assign a UNIX thread ID (which will not be the same as the LE thread ID)
    - ➢ Assign a UNIX PID (Process ID), PGID (Process Group ID), and PPID (Parent Process ID)
    - ➢ Build some new control blocks off the TCB (see next page)
  - ✗ Note that this does not mean the program is running under a shell, it just means the UNIX kernel can now respond to kernel requests because the requestor has various ID's

❒ **If a batch program is LE-enabled and has an LE-run-time parm that includes POSIX(ON), the LE thread starts out dubbed, with the LE thread ID the same as the UNIX thread ID**

- **Still, no shell is provided automatically**

z/OS Control Blocks For UNIX System Services

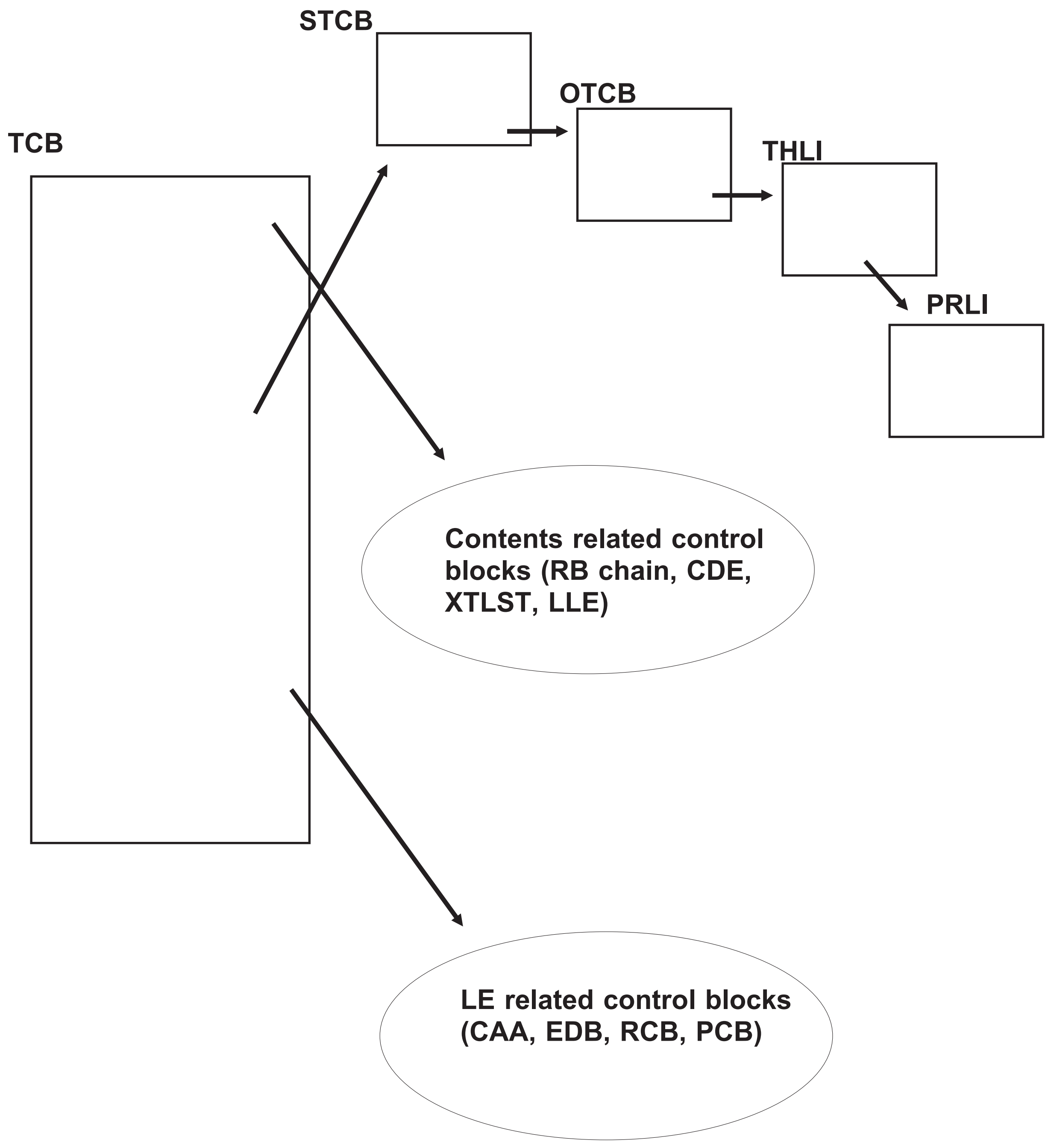
❒ **Many of the control blocks for UNIX type entities (such as directories, process groups, etc.) are maintained in the UNIX kernel address space**

❒ **But these control blocks are maintained in the program's address space, off the Secondary TCB:**

- **OTCB - Open systems Task Control Block**
  - ✗ Holds UID and GID, points to kernel services and lots of other places, including the THLI control block
- **THLI - THread Level Information**
  - ✗ Holds flags and pointers to UNIX thread related information, including the address of the PRLI control block
- **PRLI - PRocess Level Information**
  - ✗ Contains PID, flags, process return code, other pointers

z/OS Control Blocks For UNIX System Services, 2

❐ So, visually, the connections are something like this:


STCB
OTCB
TCB
THLI
PRLI
Contents related control blocks (RB chain, CDE, XTLST, LLE)
LE related control blocks (CAA, EDB, RCB, PCB)

Processes

- ❒ **A z/OS UNIX process maps, roughly, to a z/OS address space**
  - ♦ **We say "roughly" because**
    - ✗ In certain circumstances, you can request processes to share an address space
      - ➢ Address space creation is expensive in z/OS (in terms of processing time and resources)
        - ➤ so you can gain some performance benefits by sharing an address space among two or more processes
        - ➤ but you lose exact UNIX behavior replication
    - ✗ Also, in z/OS UNIX, every process must have at least one thread (task), in order to match the z/OS dispatching paradigm
      - ➢ But the minimum, strict UNIX model does not require threads *per se*
        - ➤ Multi-threading is a POSIX component
- ❒ **Notice the implication, then: a <u>process group</u> is a <u>collection of address spaces</u>**
  - ♦ **This is correct; the kernel keeps the relationships in mind and enables cross-process work (such as sharing the controlling terminal) to work as expected in a native UNIX environment**

pthreads

- **z/OS UNIX uses a hybrid dispatching scheme, so pthreads are implemented this way**
  - **Heavy weight pthreads become sub-tasks in the process address space**
  - **Medium weight pthreads run under a special subtask called the Initial Pthread Task (IPT)**
    - The IPT attaches medium weight threads as subtasks to itself
    - When the maximum allowed number of pthreads have been attached, the IPT waits until one finishes before dispatching another
  - **Light weight pthreads are not supported**
    - But that is defined to be user-controlled dispatching anyway
- **One of the implications is that you cannot issue** *fork()* **from a process that is using (or has used during its run) any pthread work**
  - **It is, at best, ambiguous how to copy the task structure in a new,** *fork()***-ed address space in this situation**

Running Under OMVS

- **When you issue the OMVS command from TSO or ISPF 6, OMVS is attached as any other TSO command processor - a subtask**
  - **The OMVS command processor initializes an LE environment, requests dubbing, and attaches the z/OS shell program**
    - OMVS has a UNIX thread ID and an LE thread ID, and they are the same value
    - OMVS itself does not have any environment variables
  - **The shell program runs any initialization script(s) that are called for, to establish options and set environment variables, and waits for user input**
    - This is a login, interactive shell

- **The OMVS command has an optional operand that allows you to establish multiple sessions when you start up**
  - **In which case, multiple process groups are set up**

Running Under OMVS, continued

- ❒ **As you issue commands, the shell creates processes to run those commands in**
  - ♦ **Or shares the address spaces with multiple processes if you are set up that way**

- ❒ **Basically, the shell runs a** *fork()*, **which creates a duplicate address space, followed by an** *exec()* **service request, which replaces the shell program by the program that runs the command**

- ❒ **An alternative is** *spawn()*, **which is a** *fork()* **and** *exec()* **in a single command**
  - ♦ *spawn()* **is part of POSIX.4b standard; a variation,** *spawn2()* **is part of the XPG4.2 standard**
  - ♦ **Whether** *spawn()* **will try to create a new process in the current address space is determined by the setting of the environment variable _BPX_SHAREAS**
    - ✗ The ability to create multiple processes in a single address space is probably unique to the z/OS and OS/390 versions of UNIX, so expecting this behavior in a script or program will be non-portable
  - ♦ **Note that** *spawn()* **will go ahead and create a new address space if, for example, there is not enough virtual storage in the current address space**

Running Under OMVS, continued

- **At this point, the OMVS user gets a sense that they are sitting at a UNIX terminal**
  - **Commands and responses are pretty much the same as one expects from any other UNIX terminal**
    - Well, there are some exceptions
      - Some of these are extensions (access to z/OS files for instance)
      - Some of these are behaviors that are common among UNIX users but not part of the standard (maybe not even written down formally)
      - z/OS UNIX System Services is new at the UNIX game, so it's still growing into it

The BPXBATCH Utility

❒ **A major component supplied with z/OS UNIX is the program called BPXBATCH**

- ♦ **This utility allows you to run a program under a (non-interactive) UNIX shell**
  - ✗ In batch
  - ✗ From TSO or TSO/ISPF
  - ✗ From REXX execs
  - ✗ From a program

z/OS UNIX Control

❐ **When an address space is created using** fork()**, we say the address space is dubbed**

- ♦ **Recall that a task within an address space is dubbed, whether or not the address space is dubbed, if a program running under the task invokes a call to a BPX service or was started with the LE run-time option POSIX(ON)**

❐ **Dubbed address spaces are seen by job control and the system operator as having a jobname inherited from the process that** *fork()***ed the address space**

- ♦ **When the process ends, the Workload Manager keeps the address space around for 30 minutes before cleaning up and removing the address space**

- ♦ **In this way, if another process comes along needing to be run, it can use the already existing one and save time and computing resources**

- ♦ **During the time of non-use, the address space has a name of BPXAS**

I/O Issues

❒ **One area that is quite different between traditional z/OS and UNIX styles of working is the management of files and buffers**

❒ **To work with files, a z/OS program traditionally creates a DCB or ACB to represent the file; in this control block is a ddname field**

- ♦ **At step allocation time, the data set is located and allocated to the job step on a shared or exclusive basis**
- ♦ **At open time, the OPEN routines look through the JCL supplied with the step for a ddname that matches, and the connection is made to an external file**
- ♦ **OPEN carves buffers for the file out of the address space the program is running in**

❒ **UNIX uses external variables to hold file names**

- ♦ **Furthermore, UNIX programs generally use file descriptors to represent files; in z/OS UNIX, a file descriptor is simply a fullword binary integer**
- ♦ **The kernel manages file descriptors, and all buffers are managed by the kernel**
  - ✗ Allowing, potentially at least, a higher degree of file sharing between processes

Practical Applications

- **So how do we expect programmers and developers to use z/OS UNIX System Services?**
  - **Porting existing applications from other UNIX systems**
  - **Creating new UNIX applications to port to other UNIX systems**
  - **Developing new UNIX applications to run on the mainframe**
    - Most likely Internet-based applications
      - With the added strengths of
        - accessing centrally stored corporate data
        - Using existing, already programmed business rules for your specific business
  - **Extending existing applications to take advantage of features and capabilities unique to UNIX (or at least not usually found on z/OS), such as**
    - 'grep' kinds of searching capabilities
    - Pipes (without the extra charge for IBM's BatchPipes program)
    - DLLs (actually available without using UNIX)
    - More natural use of multi-threading / multi-tasking

Being Practical

- ❒ **It doesn't make sense to use UNIX services where they are not needed or appropriate**
  - ♦ **Continue to develop, maintain, and extend applications where their 'natural' environment draws on traditional mainframe strengths or that have already captured the business needs for the company**
    - ✗ Large, long running batch jobs with output spinoff and step restart and checkpoint / restart capabilities
    - ✗ High volume online transaction processing
    - ✗ High data bandwidth applications that access very large databases
    - ✗ Application development tools
      - ➢ Yes, you can use UNIX tools such as the vi editor, but most mainframe developers are simply going to remain more productive with the TSO/ISPF-based tools

- ❒ **The reality is that it is not an "either" / "or" situation, but an environment where customers and users have an astoundingly wide range of useful options to choose from**

This page intentionally left almost blank.

Index

Index

Index

Index