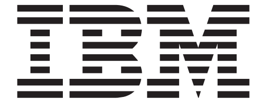
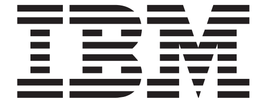
z/OS UNIX System Services



Porting Guide

z/OS UNIX System Services



Porting Guide

Note!

Before using this information and the product it supports, be sure to read the general information under "Notices" on page 123.

March 2001 Edition

This edition applies to Version 1 Release 1 of z/OS UNIX System Services Porting Guide and to all subsequent releases and modifications until otherwise indicated in new editions.

Order publications through your IBM representative or the IBM branch office serving your locality. Publications are not stocked at the address below.

IBM welcomes your comments. A form for readers' comments may be provided at the back of this publication, or you may address your comments to the following address:

International Business Machines Corporation
Department 55JA, Mail Station P384
2455 South Road
Poughkeepsie, NY 12601-5400
United States of America

FAX (United States & Canada): 1+845+432-9405
FAX (Other Countries):
Your International Access Code +1+845+432-9405

IBMLink (United States customers only): IBMUSM10(MHVRCFS)
Internet e-mail: mhvrcfs@us.ibm.com
World Wide Web: http://www.ibm.com/servers/eserver/zseries/zos/webqs.html

If you would like a reply, be sure to include your name, address, telephone number, or FAX number.

Make sure to include the following in your comment or note:
- Title and order number of this book
- Page number or topic related to your comment

When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

© Copyright International Business Machines Corporation 1998, 2001. All rights reserved.
US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

Chapter 1. An Introduction to the Porting Guide . . . 1
Using the PDF file . . . 1
Feedback . . . 2

Chapter 2. Choosing a UNIX application to port . . . 3
Owning the code . . . 3
ANSI C . . . 3
A small program . . . 3
No vendor APIs . . . 3
Database access . . . 3
3270 emulation . . . 4
HLLAPI . . . 4
COBOL considerations . . . 4
C++ considerations . . . 4
Non-Standard interfaces/functions . . . 5
Freeware. . . . 5
Performance of your first ported application. . . . 5

Chapter 3. Sizing the port . . . 7
How portable is the code? . . . 7
How much effort is involved? . . . 7
How long will the port take? . . . 8
Does the application have RAS? . . . 8
Porting centers . . . 8

Chapter 4. Setting Up to Port . . . 9
Tuning the system for optimum performance . . . 9
Creating an HFS data set on the z/OS system . . . 10
  Allocating MVS DASD Space . . . 10
Setting up security . . . 11
Getting access to the shell . . . 11
Editing . . . 11
ASCII-EBCDIC Issues . . . 12
  Background . . . 12
  Code pages . . . 12
  ASCII-like application environment . . . 13
  Application-to-application communication . . . 13
  Effect of ASCII/EBCDIC on collating sequence 13
Using TCP/IP FTP to transfer archive files . . . 14
Using an NFS Windows Client . . . 15
Data exchange and access . . . 16
Customizing and using the shells . . . 17
  Environment variables. . . . 18
  Using square brackets . . . 18
The magic value. . . . 19
set . . . 20
Testing for character strings . . . 20
Arithmetic expressions inside parentheses . . . 21
Checking your environment setup. . . . 21
Finding tools and utilities . . . 21
Online help . . . 22

Chapter 5. Assorted porting topics. . . . 23
Language Support . . . 23
  C/C++ . . . 23
  Assembler . . . 23
  COBOL. . . . 23
C/C++ Portability . . . 24
  Header Files . . . 24
  C++ Function Pointers for X11 callbacks. . . . 25
  Error Handling . . . 26
Developing a dynamic link library (DLL) . . . 26
  Building a C DLL . . . 27
  Building a C++ DLL . . . 28
X-Windows support . . . 28
man pages. . . . 28
gnu utilities . . . 29
Time management . . . 29

Chapter 6. Security considerations . . . 31
Users and passwords . . . 31
Security implications of programs running in the HFS . . . 33
  Authorizing individual programs . . . 33
Daemon program setup . . . 34
  Vendor-written programs that need daemon authority . . . 34
Enabling thread-level security for servers . . . 35

Chapter 7. Compiling . . . 37
Using make . . . 38
Libraries for functions and headers . . . 38
Ordering options and operands. . . . 39
Exporting functions and variables . . . 39
Compiler Options . . . 39
  Extension options . . . 40
Conditional compilation . . . 41
c89 access to socket header files . . . 41

Chapter 8. Debugging . . . 43
Runtime Environment . . . 43
Debugging . . . 43
  ASCII characters and strings. . . . 43
  Debugging a running program . . . 44
  Debugging authorized programs . . . 44
  Other debug methods . . . 44
Dumps . . . 44

Chapter 9. The hierarchical file system 47
An Introduction to the Hierarchical File System . . 47
The Root File System and Mountable File Systems 50
Files . . . 50
Executable Modules in the File System . . . 51
Memory-mapped Files. . . . 51
Pathnames. . . . 51
  Requirement for an Absolute Pathname . . . 52

© Copyright IBM Corp. 1998, 2001

Code Page . . . 52
Data Conversion . . . 52
Security for the File System . . . 53
Power Failures and the File System . . . 53
Sharing Files . . . 53
    Using the Network File System Feature . . . 54
    LANRES and LAN Server . . . 54
File Locking . . . 55
Opening MVS data sets from an z/OS UNIX environment . . . 55

Chapter 10. Process management . . . 57
Processes . . . 58
    Forking a New process . . . 59
    Spawning a new process . . . 60
    Replacing the program in a process . . . 61
    UID/GID Assignment: Process Authorization . . . 61
    Process groups and job control . . . 62
    Process priorities . . . 62
Threads . . . 63
    Limitation on the number of threads . . . 63
    Stopping Threads . . . 64
    Porting applications with pthreads . . . 64
Interprocess communication (IPC) . . . 64
    Shared memory . . . 65
    Message queues . . . 66
    Semaphores . . . 66
    Memory mapping . . . 67
Signals . . . 67
    Supported Signals - POSIX(OFF) . . . 68
    Supported Signals - POSIX(ON) . . . 69

Chapter 11. Networking . . . 71
TCP/IP . . . 71
AnyNet . . . 71
Sockets in the z/OS UNIX Environment . . . 72
    Sockets in z/OS . . . 72
    Writing a socket application . . . 73
    Integrated sockets PFS . . . 75
    Common INET PFS . . . 76
C/C++ resolver configuration data . . . 78
    Resolver configuration data . . . 80
    Protocol configuration data . . . 81
    Service configuration data . . . 81
    Hosts . . . 81
    ASCII-EBCDIC translation table . . . 82
    gethostid and gethostname calls . . . 82
    Where to place the resolver configuration data . . . 82
    Environment variables and the C/C++ resolver . . . 83

Chapter 12. Server models . . . 85
Iterative server programs . . . 85
Concurrent server programs . . . 86
The listener program . . . 86
The InetD generic listener program . . . 88
Starting listener programs . . . 89
Security for server programs . . . 90

Chapter 13. Database migration . . . 91

Chapter 14. After the port, focus on performance . . . 93
Use spawn() rather than fork() . . . 93
Use a threading model instead of a process model . . . 95
File I/O and Memory . . . 95
Character I/O . . . 95
Character set conversion . . . 95
Shared memory . . . 96
Do not use spins with serialization . . . 97
Compile your production application with optimization . . . 97
For large load modules, consider using LPA or VLF . . . 97
pthread_yield() calls in mainline paths . . . 98
Using HEAPPOOLS for malloc and free requests . . . 98

Chapter 15. Packaging for z/OS installation . . . 99

Chapter 16. Appendix . . . 101
Portable header files . . . 101
Porting: ASCII to EBCDIC conversion . . . 102
    Typical problem areas . . . 102
    Functions that support ASCII input/output . . . 103
    Setting a variable to convert text files in an archive . . . 103
    Commands and functions that handle conversion . . . 104
Porting services and resources . . . 104
    S/390 Partners in Development program . . . 104
    Porting centers . . . 104
    Books . . . 105
    Tools and Toys . . . 106
    Products . . . 106
Performance: tuning targets for UNIX System Services . . . 106
    Memory . . . 107
    Putting frequently used modules in the LPA . . . 107
    RACF UIDs and GIDs . . . 107
    File System . . . 108
    APPC initiators . . . 108
    Shell variables . . . 108
    Prevent propagation of TSO/E or ISPF STEPLIB data sets . . . 108
    The next step . . . 109
Two hot shell environment variables . . . 109
    _BPX_SHAREAS . . . 109
    _BPX_SPAWN_SCRIPT . . . 110
z/OS UNIX Setup Verification . . . 110
    Downloading and Running the Program . . . 111
    Feedback . . . 111
Porting with pthreads . . . 111

Chapter 17. CHARMAP source for IBM-1047 . . . 115

Notices . . . 123
Trademarks and Service Marks . . . 123

Index . . . 125

Chapter 1. An Introduction to the Porting Guide

An outgrowth of internal IBM porting work, z/OS UNIX System Services **Porting Guide**, is available from our Porting web page (**http://webdev10.pok.ibm.com/servers/eserver/zseries/zos/unix/bpxa1por.html**). The **Porting Guide** is designed to minimize the effort involved in porting an application to the zSeries platform, and it is also useful for programmers who want to write a new UNIX-style application for the zSeries platform. The **Porting Guide** points out the key differences between z/OS UNIX and other UNIX environments, providing practical advice on how to address these differences.

Over the past few years, IBM teams working on porting projects such as Lotus Notes, JAVA, Domino Go Webserver, and Digital Library have learned much about the UNIX world and the effort involved in porting a UNIX application to z/OS. The learning curve actually began with the development of z/OS UNIX System Services (known then as OpenEdition MVS). In 1989, when IBM set its sights on developing UNIX equivalency in what was then its MVS product, we needed to implement the programming APIs, provide a filesystem and networking capabilities, and deliver a function-rich UNIX shell and utilities package. With this common UNIX base in place, our customers, business partners, and we ourselves could develop or port applications for the zSeries platform.

Because new tools and applications were being ported to z/OS UNIX from other UNIX platforms, we wanted to document approaches and guidelines for these porting teams, many of which had little or no z/OS UNIX skills at the outset. We chose the web as the delivery medium for this porting information because we recognized that this information would be dynamic. By using the web medium, we can easily update the information. Users can browse various web pages and print selected topics, or download the entire document in a PDF file that can be printed.

The **Porting Guide's** content will continue to change and grow. We expect to keep updating it so that all of our customers and ISVs can continue to benefit from what we learn.

Using the PDF file

As you read through the PDF file created from the **Porting Guide** web pages, you will notice some **text that is underlined**. On the real web page, that text is a hyperlink to another web page.

When creating the PDF file, **we handled hyperlinks this way:**

- If the link is to an z/OS UNIX web page that is **not** part of the **Porting Guide**, we included the web page in the **Appendix** of the **Porting Guide**. There are a few exceptions to this — for example, the Tools and Toys page and the Compiler page were not included because they change frequently and are easy to find on the web site.
- We did not include pages from other web sites.

© Copyright IBM Corp. 1998, 2001

Feedback

After you've used the **Porting Guide**, we want your feedback. You can send in comments or suggestions from our home page (**http://webdev10.pok.ibm.com/servers/eserver/zseries/zos/unix/**) or e-mail them to **gorbsky@us.ibm.com**. So let us hear from you!

Chapter 2. Choosing a UNIX application to port

Porting an application from a UNIX platform to z/OS UNIX System Services can be very simple or it can be a major undertaking that requires reengineering of the application design. **This is a guideline for picking an application that will port quickly, in order to prove the viability of UNIX applications on z/OS UNIX.** Information is included that discusses the factors that make porting more difficult and the reasons for the difficulty. The factors discussed here are not all-inclusive, but they are examples. If an application possesses one of the difficulty factors it does not mean that the application is not a good candidate for porting, but that the port will take a little longer in order to handle that factor.

Owning the code

Any organization wanting to port an application will reduce the difficulty if it owns all of the code. The reason for this is that you can change any part of it without requiring approval of a vendor or the developer of that code.

ANSI C

This language is the most straightforward to port. It is the most widely used language on z/OS to date and, therefore, the most tested and understood. Resources for performing C code work are also more plentiful.

A small program

Small is a relative term, but in this case we recommend that the number of lines of source code be less than 100,000. This should keep the port to less than 20 person weeks of effort. The optimal porting team size is 2 to 3 people, so that the time for the port ends up at 10 weeks or fewer. This includes the move of the code, the compile, and unit testing. The team should plan for additional time for function and acceptance testing. These test times will vary in each installation.

No vendor APIs

Requiring ISV products causes delays, because in most cases the ISV product is not licensed and the pricing for that product is an additional cost for a project that is evaluating the ability of z/OS UNIX to support UNIX applications. The other exposure with ISVs is encountered if the particular ISV product needed has not been ported to z/OS UNIX. This requires IBM and/or the customer to enter into negotiations with the ISV to get a commitment to port the product. When the product is received, it will probably be the first time it is being used in a real application, so the probability of encountering additional problems during testing is increased, along with the difficulty of identifying the cause of the failure.

Database access

This subject has two parts:

© Copyright IBM Corp. 1998, 2001

1. No database at all is the preferable choice. Flat files are fine, but when you introduce a database into the port, the number of areas that could cause problems increases, and the size of the project gets bigger also.
2. If every application requires a database, then you should select an application that uses one of the databases available on z/OS. At this time, those are:
   - DB2
   - Oracle
   - IDMS
   - Adabase

3270 emulation

No one ever thought that a 3270 would have to be emulated on S/390 or zSeries. Why would anyone ever want to do that? It turns out that many UNIX applications use 3270 emulation packages to log on to mainframe systems and extract data needed to perform functions of the application. The portion of the application that uses emulator APIs will require a rewrite in order to function in z/OS.

HLLAPI

HLLAPI is not supported in z/OS. Applications using HLLAPI will require a rewrite to use Host on Demand (HODAPI).

COBOL considerations

The majority of COBOL applications port with very little effort. There is no direct interface to the COBOL compiler, which requires the compile to be done as an MVS batch job. The executable can be stored in the HFS or other library, and be executed from a UNIX shell or called by another program.

For the most part, applications written in Micro Focus COBOL can be recompiled using the z/OS COBOL compiler. This is true as long as the code uses standard functions. Exploitation of any extensions to the standard have to be evaluated to determine if the function is available or if modifications need to be made to provide equivalent results.

However, there are two functions implemented in Micro Focus COBOL that are not supported in z/OS COBOL. These are the screen scraping function and an indexed base structure on top of the POSIX file system. The solution in the screen-scraping case is to code callable subroutines that can perform screen manipulation for the application.

C++ considerations

For the most part C++ ports very well. The exposure here is that many C++ applications require class libraries(for example, RogueWave) in order to compile. Each class library requires an investigation to determine if it has been ported to z/OS.

z/OS supports C++ X-Windows applications; however, some modifications to the application’s X-Window callback function declarations may be required. The z/OS

X-Window library currently only supports static-linked (non-exported) C-style X-Window callback functions. For information on how to work around this, see our discussion of how to declare X-Windows callbacks using z/OS C++.

Non-Standard interfaces/functions

If an application exploits platform specific functions (those not part of the X/OPENstandards), they will have to be modified in order to run on z/OS UNIX.

Freeware

Many installations use freeware as part of their normal development and production environment. Many of these freeware applications have been ported to z/OS UNIX and are available from MKS via the web. They can be used on z/OS UNIX and service can be obtained from MKS. To determine if the freeware you are looking for is available for z/OS UNIX, check the web at http://www.mks.com/s390/gnu/index.htm.

Performance of your first ported application

As a rule of thumb, it is better to start with an application that does not require high performance. This is because tuning an application on z/OS UNIX is much different than tuning an application on other UNIX platforms. The architecture of the zSeries is much different than other UNIX systems; changes used to improve performance on other UNIX systems may no longer produce the same improvement on z/OS UNIX. The use of caching is one of these areas. This is because a cache miss on z/OS does not have as big an impact as it does on other systems. Because cache misses on z/OS do not cause a significant degradation, the performance changes made for traditional UNIX would not result in improved performance on the zSeries.

For some applications, there may be coding changes you can make to optimize performance. For details about these changes, see the chapter ″After the Port, Focus on Performance″ in the **Porting Guide**.

And, finally, after a function-oriented application has been ported, you can use the various measurement and tuning facilities of z/OS to tune the application to perform at maximum efficiency.

Chapter 3. Sizing the port

Having the correct skills before starting a porting project is essential. For example, C/C++ programming and debugging skills are critical. And, of course, knowledge of UNIX externals is also useful. When source code needs to be modified to get an application or tool running on another platform, it helps if the programmer doing the port also knows the application (and, even better, has experience porting this code to other platforms).

To determine how portable the code is, size the port in advance. This requires some analysis of both the application source code and the functionality provided by z/OS UNIX System Services (z/OS UNIX). You may discover simple changes to C header files are needed or you may identify significant design implications.

How portable is the code?

A code checker tool can be used in advance of a porting project to identify code that will need to be changed. Some of these tools are quite sophisticated and allow you to easily map application programming interface usage to industry programming standards (for example, POSIX, X/OPEN). One example is Code Integrity, a tool from Mortice Kern Systems of Waterloo, Canada, which analyzes an application to determine if there are non-portable functions used. It has specific information for z/OS UNIX. Code checkers can provide a useful checklist if no one with porting experience or application knowledge is available.

In many cases, however, the reason that a particular piece of code does not port to a new environment has more to do with the way that a function has been used rather than the function itself.

How much effort is involved?

If the program is XPG4-compliant, then the likelihood of having a smooth port increases significantly. However, there are many legacy UNIX applications that are coded to non-standard interfaces, so you need to look at the services, functions, and APIs that the application requires.

1. **system calls:** Make a list of all system calls that the application makes. Compare this list with the z/OS C/C++ Run-Time Library functions. Any functions that the z/OS C/C++ RTL does not supply will have to be addressed during the port.

   To help make a list of system calls, you might try using the **nm** utility on the platform where the executable(s) are already built. While the list may not be perfect, it can often point out trends in API usage.
2. **header files:** You will uncover header files that have not been ported. We have accounted for the fact that some non-standard header definitions will be encountered, by providing compiler options for deciding that let you compile a given application against a specific Open standard (for example, _OPEN_SOURCE) or against historical UNIX (_ALL_SOURCE). Also, for some of these missing headers, there are workarounds.
3. **ASCII-to-EBCDIC** conversion: Analyze the code for ASCII dependencies to determine how much work this may involve. Remember, z/OS adheres to a EBCDIC code-page environment and most of the rest of the world lives in an

© Copyright IBM Corp. 1998, 2001

ASCII environment. If the application was originally developed without this understanding then programmer shortcuts (such as comparisons that assume ASCII A < a, whereas in EBCDIC A > a) may introduce issues for your port. The good news is that most of these errors are easy to spot and to correct (for example, use the strcoll () function).

How long will the port take?

It can take 10 minutes to recompile a simple utility downloaded from the internet, and we've seen many applications that fit into this simple recompile and run category. Larger applications have ported easily, but some elect to build additional functionality, scalability, or reliability into the solution by exploiting traditional z/OS platform strengths. Additional redesign, development, large system testing can easily add months, if not seasons, to the completion date.

Does the application have RAS?

To obtain Reliability, Availability and Serviceability (RAS) characteristics consistent with those expected by z/OS customers, a level of investment must be planned. z/OS customers require that products remain continuously available. This means that software products must recover from failures, even if some requests are caused to fail. Useful diagnosis information about product failures must be captured the first time they occur so that problems can be identified and fixes applied (if possible) to minimize reoccurrences. This automatic gathering of information is called "first failure data capture," and promotes serviceability. The resulting actions that are required include:

- Requiring descriptive messages with unique message identifiers, and returning erroneous requests with unique error codes
- Applying application recovery techniques within your programs, like driving thread termination and retry in the applicable cases (similar to z/OS capabilities)
- For each call to a service that can return error indicators, testing those indicators
- Capturing enough diagnostic data in the event of an error to ensure that the cause can be isolated and repaired (first failure data capture)
- Allowing for the delivery of individual fixes for problems
- Providing good serviceability (production diagnosis/debugging) documentation

Application recovery processing and unique error messages, ABEND codes, and reason codes are examples of information that can result in improved availability characteristics of the application environment because it can be better managed. Other forms of serviceability, such as tracing and error logging, must be enabled in the application, just as would be done when developing z/OS functions to enable rapid identification of the root causes of problems.

For more "how to" information on this topic, see z/OS Reliability, Availability, and Serviceability Guidelines for Ported Software on the z/OS UNIX Porting web page.

Porting centers

You may want to investigate the IBM porting centers and the support they can provide for porting applications.

Chapter 4. Setting Up to Port

While porting your application to z/OS's UNIX System Services (z/OS UNIX) environment, you can work in a familiar environment, using a UNIX shell that has the same look and feel and utilities as other UNIX environments. The z/OS UNIX shell is modeled after the UNIX System V shell with some of the features found in the KornShell. Also, the **ksh93** shell has been ported to z/OS, and is available for download.

In addition to the z/OS UNIX shell, there is also the tsch shell, which was ported to z/OS and provided with OS/390 V2R9. tsch is an enhanced, but completely compatible version of the Berkeley UNIX C shell, csh. This port provides Unix System Services users a means to run their C shell scripts, and offers users a more flexible environment with the addition of a second shell. tcsh also provides many features, including programmable word completion and spelling correction, which are not available in the z/OS UNIX shell.

If you want to be able to work with your workstation tools, and you have the Network File System feature on your workstation, you can mount a hierarchical file system (HFS) directory on your workstation's file system. Using NFS to mount an z/OS UNIX hierarchical file system to a workstation file system lets you use your workstation tools for tasks such as source code control, editing, and code quality checking.

These are the topics we will discuss:
- Tuning the system for optimum performance

- "Creating an HFS data set on the z/OS system" on page 10
- "Setting up security" on page 11
- "Getting access to the shell" on page 11
- "Editing" on page 11
- "ASCII-EBCDIC Issues" on page 12
- "Using TCP/IP FTP to transfer archive files" on page 14
- "Using an NFS Windows Client" on page 15
- "Data exchange and access" on page 16
- "Customizing and using the shells" on page 17
- "Checking your environment setup" on page 21
- "Finding tools and utilities" on page 21
- "Online help" on page 22

Tuning the system for optimum performance

Tuning the system is critical for performance. On our web site, we have a list of release-specific tuning tactics, including a list specially designed for compile-intensive systems. For example, the compiler, critical parts of the Language Environment Run-Time Library, and other key z/OS UNIX executables need to be placed in the Link Pack Area (LPA). Check with your MVS systems programmer to be sure your system has been tuned before you begin your port.

© Copyright IBM Corp. 1998, 2001

Creating an HFS data set on the z/OS system

Ask your friendly MVS systems programmer to give you an HFS data set mounted at /u/userid. You will store your source files in /u/userid, your home directory.

HFS files are organized in a hierarchy. All files are members of a directory, and each directory is a member of another directory at a higher level in the hierarchy. The highest level of the hierarchy is the root directory. z/OS UNIX views an entire file hierarchy as a collection of hierarchical file system data sets (HFS data sets). An HFS data set is a new type of MVS data set created for z/OS UNIX. Each HFS data set is a mountable file system.

A file in the hierarchical file system is called an HFS file. An HFS data set can contain one or many HFS directories (it will always have at least one) and zero or more files. The highest level directory of an HFS can be thought of as the root for that HFS, but be careful not to confuse this with the root of the entire file hierarchy. Note that when an HFS is initially created, its root will have permission 700, which may not be what you intended. You may want to change this to 755, 775, or even 777 after mounting it the first time.

DFSMShsm (Data Facility System-Managed Storage Hierarchical Storage Manager)provides automatic backup facilities for HFS data sets. ADSTAR Distributed Storage Manager (ADSM) provides backup function for HFS files. There are two types of backup: incremental, in which all new or changed files are backed up, and selective, in which the user backs up specific files. If HSM backs up an HFS data set that has not been used recently, it removes the data set from the current DASD volume. This frees up space of more active data sets. On the other hand, ADSM will perform the backup of files, but it does not free up the space for the file.

Allocating MVS DASD Space

In the zSeries world, disk space has traditionally been allocated in units of tracks or cylinders even if allocation in units of blocksize has been possible. The disadvantage of this method is the size of a track or cylinder often is different depending on the type of disk hardware.

More recently, it is possible to allocate space in terms of megabytes (MB 1024*1024 = 1048576 bytes) or kilobytes (KB 1024 bytes). Programmers from the UNIX world are more familiar with this method of allocating space and can better estimate how much space should be allocated in the HFS to hold their source code.

Below is a short generalized table showing DASD allocation sizes for most commonly used DASD families:

DASD type	Track size	Cylinder size	Tracks/Cylinder
3390	56KB	850KB	15
3380	47KB	712KB	15
9340	46KB	697KB	15

An HFS data set can have up to 123 secondary extents, just like a PDS/E.

To give you an idea of how much disk space you may want to reserve, here are a few examples of the differences in the size of source and object files in Linux and

z/OS UNIX. These numbers were derived using the "du -ks" command. The object files were compiled with the debug (-g) option and without optimization.

Directory	Linux Source	Linux Object	UNIX Source	z/OS UNIX Object
substrate	7371	48530	9168	71400
ms	7457	122781	12444	180660
tas	2653	54061	3636	91892
ui	14377	62618	17268	174816

Setting up security

Because UNIX System Services is an add-on to z/OS, functions such as security and system maintenance already existed and were not reinvented. To do security setup and userid maintenance, etc., you must revert to the 'native' interface, namely TSO. This is MVS systems programmer territory. To aid in the tasks typically needed, see *z/OS UNIX Planning* for information on setting up security, file systems, users, and many other tasks.

Getting access to the shell

Before you can move your data files to the z/OS UNIX system, you will need a user ID and password on the z/OS UNIX system; the user ID must contain an OMVS RACF segment. Have your MVS systems programmer set this up for you.

After you have a user ID and password, you can access the shell in one of these ways:

- With the OMVS command, from a logged on TSO/E user ID using a 3270 terminal or using a workstation running a 3270 emulator
- With the rlogin command, from a workstation running TCP/IP
- With the telnet command, from a workstation running TCP/IP

  Note: Be sure to check with your systems programmer for the system name and port address you should use for telnet. There are two telnet servers on MVS (one for TSO and one for the shell) and you need to get to the one for the shell. On many systems, the shell server is listening to port 623, instead of the well-known port 23 that all clients use by default.

Editing

These UNIX editors are supported: **vi**, **sed**, and **ed**.As of May 1999, the z/OS UNIX port of **GNU Emacs 19.34** is available from Mortice Kern Systems, Inc. You can download it and try it out. To improve performance, ask your systems programmer to place the editor (for example, /bin/vi) in the Link Pack Area.

If you have X-Windows support on your workstation (for example, Exceed's Hummingbird), you can use these X-Windows editors:

- Visual SlickEdit from MicroEdge, Inc., a highly extensible programmer's editor that runs on the Windows platform and on z/OS UNIX.
- nedit (download), an X-Windows based editor, with a GUI interface much like you find on Mac and Windows systems.

You can use **vi** and **Emacs** only from rlogin or telnet sessions, which are raw-mode sessions.

To ease the ASCII/EBCDIC issue, you can download and use viascii, an ASCII version of **vi**.

The ISPF editor is also available from the shell if you logged in via TSO/E; you can use the ISPF editor only during an OMVS session (a line-mode session), not with rlogin or telnet. You invoke ISPF file edit using the oedit shell command or OEDIT TSO/E command; you can invoke file browsing using the obrowse shell command or OBROWSE TSO/E command.

Of course, if you use NFS to mount HFS files to your workstation, you can use your favorite workstation editor.

ASCII-EBCDIC Issues

Probably the most significant consideration when porting UNIX applications to z/OS UNIX is that z/OS UNIX uses a completely different set of hexadecimal character representations than UNIX systems. There is a requirement at times to translate between the two different systems and between variations of each. Often this is done automatically; sometimes it is a programmer's responsibility.

Programmers who are porting need to keep a clear head as to where they are and who is responsible for translation. When debugging, it is always worth asking yourself if the problem could be an ASCII/EBCDIC issue first.

Background

Given 8 bits to a byte, then there are 256 different possible bytes available for representing characters. Each different byte can then arbitrarily be assigned to a particular character. This process is called character encoding (or decoding).

IBM defined EBCDIC (Extended Binary-Coded Decimal Interchange Code) as one particular character encoding scheme for use in its computers and the American National Standards Institute (ANSI) defined a different code called ASCII (American National Standard Code for Information Interchange).

All UNIX and PC systems use ASCII in one form or another, but IBM mainframes, among others, continue to use EBCDIC. Therefore, when porting from any UNIX platform to z/OS UNIX, take the ASCII and EBCDIC dependencies into account. This is a unique consideration for z/OS UNIX and something to which even experienced porters will not be accustomed.

Code pages

A code page for a specific character set determines the graphic character produced for each hexadecimal encoding. The code page used is determined by the programs and national languages being used. For internal processing, z/OS UNIX uses EBCDIC. To be specific, it uses the character set in the EBCDIC Latin 1/Open Systems Interconnection Code Page 01047. Any text to be used in shell processing must be converted to code page 01047. Depending on its origins, a file could be in any one of a number of different code pages, both ASCII and EBCDIC. For example, a tar file would normally be stored using an ASCII code set. See the *z/OS UNIX User's Guide* for more information.

ASCII-like application environment

The libascii functions and V1R3.0 C/C++ __STRING_CODE_SET=″ISO8859-1″ predefined macro provide an ASCII-like application environment on z/OS. libascii supports ASCII input and output characters by performing the necessary iconv() translations before and after invoking the C/C++ run-time library functions. The __STRING_CODE_SET=″ISO8859-1″ predefined macro generates ASCII characters, constants, and strings.

As of OS/390 V2R8, the libascii functions are integrated into the base of Language Environment. For earlier releases, the libascii package is available from an z/OS UNIX Tools and Toys Page .

An application with a large number of ASCII characters defined in hex or octal can be ported by compiling the code in ASCII and using the libascii calls to convert to EBCDIC in system calls which must be passed EBCDIC.

Application-to-application communication

When two processes using different code pages communicate, it is the responsibility of the application to code the translation. It is also the application's responsibility to determine which data needs to be translated. That is, text data is translated, binary data is not.

If your application already has code to translate between different ASCII encodings on the client and the server, it can be a simple matter to code the EBCDIC translation. By using two tables (one for ASCII to EBCDIC and one for EBCDIC to ASCII), the code can use the character as the index into the table to look up the translated character. All that is required is a translation table between the two code pages (for example, ASCII ISO 8859/1 and EBCDIC Code Page 1047).

There is an ASCII/EBCDIC translation table supplied with the TCP/IP software. The dataset name is:

```
TCPIP.SEZATCPX(OEMVS311)
```

An alternative is to use the iconv() function. This provides translation between two specified character sets or code pages.

To handle ASCII APIs from other systems, you can use our libascii functions.

Effect of ASCII/EBCDIC on collating sequence

Some functions will give different results on z/OS than in ASCII-based systems. When a function uses the relative position of characters to do a sort or compare, the results will differ depending on whether you are running on a platform with the ASCII encoding or running that same function on a platform based on the EBCDIC encoding. Examples are:

```
String compare   (strcmp (s1,s2))
String uncompare (strucmp (s1,s2))
Memory compare   (memcmp (s1,s2))
```

To illustrate in ASCII, when comparing the character ″A″ to the character ″a″, ″A″<″a″, but in EBCDIC, ″a″<″A″. This will result in different return codes on compares and different output sequences on sorts.

The order of letters and numbers differs in ASCII and EBCDIC, as well as the relationship of upper and lowercase:

- EBCDIC: Lowercase letters occur before the uppercase letters, which occur before the numerals.
- ASCII: Numbers occur before uppercase letters, which occur before lowercase letters. This order is the opposite of EBCDIC.

So applications which have dependencies on a given collating and/or sorting order must be modified.

It is useful in the early stages of a port to identify all such usage. A code checker can be helpful. Decisions then have to be made as to whether to recode the logic of the application or to perform a translation immediately prior to affected usages.

Summary

Translation to and from EBCDIC is a requirement unique to porting to some selected platforms including z/OS UNIX. However, conceptually it is no different than translating between different ASCII character representations. So long as the application logic does not rely on a particular encoding for any character, then EBCDIC translation need not be a major issue.

Attention should be paid at installation time to ensuring that all required characters can be correctly entered, displayed and printed from all workstations.

For additional information on how this issue affects porting, see the web page Porting: ASCII to EBCDIC Conversion.

Using TCP/IP FTP to transfer archive files

If TCP/IP is installed on both the workstation and the z/OS system, you can use the File Transfer Protocol (FTP) facility to transfer your source data. If transferring single-byte data, FTP will convert the data from ASCII to EBCDIC for you. Binary data can also be sent using FTP if the binary parameter is specified.

Files are often bundled together in single files by utilities like pax, tar, and cpio. When these files are bundled into a single file, it is called an archive file. By convention, the file name of the archive indicates the utility that was used when the file was built. For example, a file named mvsport.tar indicates that the tar utility was used. The three utilities provide basically the same function: reading and writing of archive files. The important thing to know is tar and cpio can only read and write files of their respective formats, but pax can read or write in either format. So given a pax, tar, or cpio file, you can use pax from the shell to ″explode″ or unwind the archive into its individual files.

To save disk space and transmission time, archive files can also be compressed by using the -z option with the tar, cpio, and pax utilities. The naming convention that is generally used for a compressed archive file is to end the filename with a .Z — for example, mvsport.tar.Z.

With the TCP/IP from the Communications Server for z/OS you can ftp files into or from that system’s HFS.

Note: Be sure to check with your systems programmer for the system name and port address you should use for ftp. Prior to V2R5, there were two ftp servers on MVS (one for MVS datasets and one for HFS files). On many systems, the HFS

server that comes with the applications feature is listening to port 621, instead of the well-known port 21 that all clients use by default. As of V2R5, the two servers are combined into one that listens on the well-known port 21.

An FTP client forthe shell and utilities, **ncftp**, is available to download. Be sure to read the ″readme.MVS″ file before you unwind the tar file.

Note: If your browser does not work with the above link, use this URL instead:

`ftp://ftp.s390.ibm.com/u/ftp/os390/oe/port/ncftp.tar.Z`

With archive files, we recommend using the BINARY transfer option on FTP PUT and GET commands. If you don't use BINARY mode everywhere, your archive will most likely be corrupted and unusable.

Using an NFS Windows Client

The NFS server supports two authorization protocols:

- An IBM-only protocol: For NFS connection between IBM platforms, such as OS/2 and z/OS. This requires **mvslogin**.
- PCNFS: Used for non-IBM platforms, this is the UNIX standard and you will use it for a Windows client. It is the equivalent of mvslogin. This protocol flows the userid and password on the connection request. For more information, read the Appendix ″Using PCNFSD Protocol″ in *z/OS NFS Customization & Operation Guide*.

When you are accessing data in the HFS using NFS from another system, in text processing mode the data is converted between ASCII and EBCDIC by NFS. The default translation table (internal to NFS) converts between EBCDIC code page 0037 and ISO 8859-(ASCII). For accessing UNIX System Services files, the OEMVS311 translation table is specified either in the mount command or in the *xlat* processing attribute. User defined tables may also be specified. See *z/OS NFS Customization and Operation* for further information.

Here are some recommendations for using NFS:

- **Text and binarymountpoints:** The NFS development team strongly suggests using two mountpoints, one for **binary** files and one for **text** files. z/OS uses EBCDIC and its data is record-oriented (not byte-oriented like workstation data). For binary data flowing between z/OS and an ASCII workstation, NFS can keep the data format as it is. However, for text data, NFS has to convert the data from ASCII to EBCDIC, and the byte-oriented data stream to record-oriented data.

  If you are not generating a lot of small binary files, you could simply use a text mountpoint. For transferring large binary files between workstation and server, you could then use ftp. However, if you are running tools on your workstation that generate lots of small binary files, then use a separate binary mountpoint and write to the HFS.
- **SymbolicLinks:** Currently we have not found a Windows-based NFS client that recognizes symbolic links on z/OS UNIX. You can try this approach to organizing your directories if you use symbolic links:
  1. Segregate your **source parts** into a directory (for example, **src**) for source parts only.
  2. For symbolic links to other parts, create a separate directory to hold the links to both R/O source and your own local source. For example, you could have

a **bld** directory that has links to the files in your own local source directory and links to other source directories (owned by people on your team). Do your builds in this directory.

- **End-of-Linedelimiters:**

  The NFS translation table OEMVS311 converts EBCDIC X'15' (NL) to ASCII X'0A' (LF) and vice versa. But for a Windows workstation, you need to translate NL to CRLF. There are several available workarounds to this problem.
- **Authentication/security:** To share an HFS among a group, put everyone in the same RACF group. This means that all files created by group members will have the same group ID, and everyone in the group will have access to the files. The permissions for the directories that everyone will share should allow group read and group execute. Members of the group can use a common **umask** setting that grants group read and execute access to the files.

Data exchange and access

The implementation of the HFS is transparent to the type of data to be stored. Data may be text or binary; the implementation does not care about this. However, you may need to know what kind of data you are working with when you exchange data with other systems:

- Text Data

  In a client/server environment where z/OS UNIX participates, you have to consider how to do ASCII-EBCDIC translation. At first this might seem to be only a matter of two translation tables, but it goes much further than that. As soon as you have to deal with different countries, you have to handle translations using the appropriate code page. z/OS UNIX services provide:
  - The iconv command in the shell environment
  - The iconv() routine for programs coded in C

  Both allow you to convert according to the code pages that are applicable.
- Binary Data

  Binary data is much more complicated to handle than text data. Integer variables may be stored differently according to the byte order in storage (little endian (PC) versus big endian (zSeries)). Different representations of floating point variables exist. There is no single simple solution for handling such differences. Each situation has to be analyzed to find a suitable solution.

In an environment where zSeries provides the server platform, you normally can make a trade-off based on what kind of coding is mostly needed. If workstation users store and fetch data in the HFS based on NFS or FTP and further processing on the MVS system is limited, then you may decide to store all data in the representation it has on the workstations.

To transfer files in and out of your z/OS UNIX system, you can use the FTPD server from TCP/IP. With this server, you can transfer files directly in and out of the HFS, as well as in and out of traditional MVS data sets. You can use the FTPD server for transferring both binary data and text data (C source code). You can transfer single files, tar files, cpio files, and pax files. The latter are collections of files which make it easier to transfer a bunch of files and even allow you to reconstruct the directory structure. Use tar, cpio, or pax files when exchanging data which span several directories. The drawback is that you have to separate text and binary data.

Archive files: As of OS/390 V2R8, pax and tar can read and write archive files that reside in MVS sequential or partitioned data sets.

Unpacking a tarfile: As we mentioned before, file archives of tar, cpio, or pax format are very convenient to transfer files that are organized in a directory tree. The pax shell command allows you to expand the various kinds of archive file formats (cpio or tar). It allows you to translate data from ASCII to EBCDIC or vice versa at the same time. Here is an example of how to extract a tar type archive and translate from ASCII to EBCDIC at the same time:

```
pax -o from=ISO8859-1,to=IBM-1047 -rf  your_file.tar
```

Copying databeween data sets and the file system: Data exchange between traditional MVS data sets and the HFS can be accomplished through the shell commands cp and mv (as of OS/390 V2R8)or the TSO/E copy commands that are provided with z/OS UNIX: OCOPY, OPUT, OGET, OPUTX and OGETX.

- As of Release 8, the cp and mv shell commands can be used to move files between UNIX and MVS data sets. The commands support text, binary, and executable files; and partitioned and sequential data sets.
- The TSO/E commands:
  - OCOPY is based on DDNAME allocation. So you have to allocate your input and output DDNAMEs to MVS data sets or HFS files before you invoke the OCOPY command.
  - OGET, OPUT, OGETX and OPUTX give you the opportunity to specify the MVS data set name and HFS file name directly on the command invocation.
  - OGETX and OPUTX support partitioned data sets, allowing you to copy all members of a partitioned data set in one command invocation. In addition you may apply a suffix to the new files. If, for example, you want to copy all members of your partitioned library *hlq.project.c* to the HFS as files in a specific directory, you can have all member names changed to file names of the form *membername.c* while you copy.

The TSO/E commands support ASCII to EBCDIC conversion.

When you copy from a traditional MVS data set that contains binary data, you should use the BINARY option of the OPUT or OPUTX command:

```
OPUT GZIP.EXE '/u/hdm/gzip' BINARY
```

To copy a data set that contains text type data in ASCII encoding and convert the data to EBCDIC while copying to the HFS, use this command format:

```
oput love.letter '/u/hdm/love.letter' binary CONVERT((BPXFX111))
```

Note: You need both parentheses for the CONVERT keyword parameter. The BINARY option is important, because it tells OPUT to handle the input data set as a string of bytes, not as a collection of records.

Customizing and using the shells

The z/OS UNIX shell is modeled after the UNIX System V shell, with some of the features of the 1988 KornShell. The shell conforms to the XPG4 standard.

ksh93is ported to z/OS; for information on how to obtain it, go to our Tools and Toys web page (http://webdev10.pok.ibm.com/servers/eserver/zseries/zos/unix/bpxa1toy.html).

Tcsh is an enhanced but completely compatible version of the Berkeley UNIX C shell, csh, that was also ported to z/OS.

If you are using the z/OS UNIX shell or tcsh for the first time, you might need to do some customization before you start to compile and link your code.
- "Environment variables"
- "Using square brackets"

These are some areas where the behavior of the z/OS UNIX shell may differ from the UNIX shell that you are accustomed to:
- The magic value
- The set command
- Testing for character strings
- Arithmetic expressions inside parentheses

Environment variables

You will want to customize environment variables associated with the c89/cc/c++ utility and environment variables in your $HOME/.profile.

The c89/cc/c++ command is the interface to the z/OS C/C++ compiler, the prelinker, and the linkage editor for z/OS UNIX. The c89/cc/c++ command can be invoked directly from the shell or a batch job.

There are a couple of environment variables that are used to point the c89/cc/c++ utility to the header files. For example,
- Environment variable {_INCDIRS} is set, by default, to search the /usr/include directory
- Environment variable {_CLIB_PREFIX} is set, by default, to search the MVS data sets that contain the C/C++ compiler header files and the compiler messages.

Our Compiler web page and the *z/OS UNIX Command Reference* have more information about the environment variables that affect the c89/cc/c++ utility.

For improved shell performance, there are two shell variables to customize: _BPX_SHAREAS and _BPX_SPAWN_SCRIPT. See the web page that discusses them.

For more information on customizing environment variables, see *z/OS UNIX Planning* and *z/OS UNIX User's Guide* . For more information on c89/cc/c++, see *z/OS UNIX Command Reference*.

Using square brackets

When you use OMVS functions from 3270 terminals, you will encounter problems with the square brackets ([ ]). This is true even in a native US environment that uses host code page 037. The differences between the code pages that z/OS and z/OS UNIX System Services (code page 1047) become apparent when you are using square brackets and some other characters. To be able to enter, display, and print square brackets correctly, you need to customize the keyboard mapping on your workstation, and also ensure that z/OS UNIX and any 3270 emulation package that you might be using are both using the same code page.

This topic of variant characters whose hexadecimal coding may vary between code pages is discussed in the *z/OS UNIX User's Guide*. There is no general solution. Each 3270 emulator is likely to require slightly different customization. For the ISPF edit environment, you can select ISPF option 0 and then set the terminal type to 3278A.

A different approach may be to customize an z/OS UNIX user conversion table, BPXFXnnn. But, in this case you have a solution only in the OMVS shell environment, not in the ISPF edit environment.

If you are usingPersonal Communications/3270 on your workstation, you can re-map square brackets for an OEDIT session, so that you can display/create square brackets in C and Java programs.

The square bracket characters are significant in both C code and shell scripts. It is therefore important that:

- They are displayed correctly at the workstation being used.
- The user has the ability to enter them from the workstation.
- Printed output shows them correctly.

It is recommended that you determine the CECP (country extended code page) used in your z/OS installation (ask your friendly z/OS system programmer) and the differences between it and Code Page 1047.

Using the Convert Option on the OMVS Command:By default, the OMVS command uses a null character conversion table, (BPXFX100). This does no translation between code pages. The OMVS command has a CONVERT option that lets you specify a conversion table for converting between code pages. The table you want to specify depends on the code pages you are using in z/OS and in the shell. For example, if you are using code page 0037 on your z/OS system and code page 1047 in the shell, specify the following when you enter the OMVS command:

```
OMVS CONVERT((BPXFX111))
```

Conversion table BPXFX111 will display square brackets correctly for operations that are performed in the shell. For example, square brackets will be displayed correctly in a file that is processed with the `cat` shell command. Using the `ed` editor from the shell will also display the square brackets correctly if the correct CONVERT option is used. This technique works from both 3270 type terminals and workstations running a 3270 emulator. For more information, see the OMVS command description in *z/OS UNIX Command Reference*.

The magic value

Many UNIX systems support an executable text file that contains a ″magic number″ (also known as ″pound bang″, or ″shbang″) This is a text file beginning with `#!pathname`, for example:

```
#!/usr/bin/perl
```

In OS/390 V2R8, the z/OS UNIX shell supports the magic value. In earlier releases, it was not supported. Tcsh also supports the magic value.

For systems that do not support it, Wall and Schwartz in *Programming Perl* recommend replacing

```
#!/usr/bin/perl
```

with this:

```
eval 'exec /usr/bin/perl -S $0 ${1+"$@"}'
    if 0;
```

set

The set command in z/OS UNIX returns all shell variable values enclosed in double quotes. This behavior, documented in the *z/OS UNIX Command Reference* may require modification of shell scripts that rely on either no quotes or single quotes in the returned data.

Both set and export display even simple variable values enclosed in double quotes:

```
ac_cv_newval="abc"
 ac_cv_otherval="zyx"
```

Scripts that save a configuration value with the use of set or env might at first save values in this way:

```
ac_cv_newval=${ac_cv_newval='"abc"'}
ac_cv_otherval=${ac_cv_otherval='"zyx"'}
```

Executing the script a second time might result in:

```
 ac_cv_newval=${ac_cv_newval='"\"abc\""'}
 ac_cv_otherval=${ac_cv_otherval='"\"zyx\""'}
```

And a third time:

```
ac_cv_newval=${ac_cv_newval='"\"\\\"abc\\\"\""'}
ac_cv_otherval=${ac_cv_otherval='"\"\\\"zyx\\\"\""'}
```

If a configuration file is generated by a script, you might avoid this sort of problem by careful use of a sed filter to remove the unnecessary double quotes:

```
 ... | sed "s/=\'"'"\(.*\)"'"'/='\1'/" | ...
```

Testing for character strings

Different shells provide different means to test for character strings. As of OS/390 V2R8, the z/OS UNIX shell supports ″double brackets.″ For earlier releases when they were not supported, you can do things like:

```
if [ "$resp" = yes ]
 then
   .
   .
   .
 fi
```

Alternatively, you could use either of the following condtions to accomplish the same thing

```
 if [ X$resp = Xyes ]
 if [ "X$resp" = Xyes ]
```

Also, test cannot do lexical greater-than or less-than comparisons, as in this example:

```
 a="abc"
 b="def"
 if [[ $a > $b ]]
 then
```

```
    print "a bigger than b"
  else
    print "a smaller than b"
  fi
```

This sort of script would have to be rewritten as:

```
a="abc"
b="def"
if expr $a \> $b
then
  print "a bigger than b"
else
  print "a smaller than b"
fi
```

Arithmetic expressions inside parentheses

By default the z/OS UNIX shell does not accept statements like:

```
((x=y+z))
```

You must either:

- Use ″let″ (the command for which ((...)) is often synonymous):

  ```
  let x=y+z
  ```

or

- Turn on the ″korn″ shell option:

  ```
  set -o korn
   ((x=y+z))
  ```

Checking your environment setup

After you have set up your z/OS UNIX environment, you can test it with a simple port, such as gzip. Visit our web page that has instructions for porting gzip to z/OS UNIX.

Finding tools and utilities

Each development group has its own standards and favorites among the many tools and utilities widely used in the UNIX environment. Some are available on z/OS UNIX, some are easily ported, and others can be used in the development environment at your workstation.

On our tools and toys web page (http://webdev10.pok.ibm.com/servers/eserver/zseries/zos/unix/bpxa1toy.html), there are a number of tools and toys you can download, and the list keeps growing. Ported versions of gnumake and other gnu utilities are available from the MKS web site.

Cscope,available on the AIX and Solaris platforms, can be used examine C programs interactively. This handy tool helps you learn how a C program works, without endless flipping through a thick listing. It can locate the section of code that needs to be changed to fix a bug without having to learn the entire program.

If you are using an NFS-mounted z/OS file system, you can use Cscope on your development platform.

Online help

For the shell commands, there are man pages.

The TSO/E OHELP command provides a similar capability to the man shell command. OHELP displays online reference information about commands, C functions, callable services, and messages issued by the shell and dbx. You must have Bookmanager Read installed to use OHELP. For more information about using online help, see *z/OS UNIX User's Guide* .

Chapter 5. Assorted porting topics

In this chapter, we discuss assorted porting topics:

- “Language Support”
- “C/C++ Portability” on page 24
- “Developing a dynamic link library (DLL)” on page 26
- “X-Windows support” on page 28
- “man pages” on page 28
- “gnu utilities” on page 29
- “Time management” on page 29

Language Support

z/OS UNIX System Services (z/OS UNIX) supports the z/OS C/C++, Assembler, and Cobol languages.

C/C++

z/OS UNIX suppports the z/OS C/C++ run-time library. OS/390 Version 1 Release 2 achieved the XPG4 UNIX Profile Brand. We have web pages that list the X/Open Single UNIX Specification (UNIX 95) C functions that z/OS provides, arranged alphabetically and by functional category.

Assembler

Generally, z/OS UNIX uses an assembler callable service whenever the C RTL (run-time library) needs to do something that requires being in an authorized state. Some C functions are passed on by the C language support to the kernel address space via the assembler callable services (also known as syscalls).

These callable services are documented in *z/OS UNIX Programming: Assembler Callable Services Reference*.

COBOL

There is no explicit COBOL support for z/OS UNIX. The COBOL for z/OS product does not exploit z/OS UNIX functions such as the HFS and multiple threads. (The compiler uses standard MVS datasets for source, COPY books, listings, and for object programs that are the target of DYNAMIC CALL.) However,

- COBOL can call routines using standard MVS linkage conventions, so a COBOL program could make use of many of the z/OS UNIX callable services.

  Note that if you have a UNIX COBOL program that never existed on zSeries before and was written with Micro Focus COBOL for UNIX or an X/Open-compatible COBOL compiler, the program may use COBOL syntax that is not supported by the IBM COBOL compiler. For example, Micro Focus COBOL for UNIX and other X/Open-compatible UNIX COBOLs have language for screen handling, for line-sequential files, and for record locking that the IBM

© Copyright IBM Corp. 1998, 2001

COBOL compiler does not support. Prior to compiling such a program with the IBM compiler, any usage of these language elements must be recoded to use supported language.

- At IBM's International Technical Support Organzation in Poughkeepsie, they have successfully compiled COBOL code, using batch, into an HFS executable. In this case, they are using COBOL as a web server CGI program, with a C wrapper to do the STDIN and STDOUT.

There is no debugger for COBOL that runs under z/OS UNIX. The COBOL Debug Tool supports C and C++ applications running under OE, when Debug Tool is being used by the C/C++ VisualAge Debugger. We do not anticipate any significant problems that would prevent it from debugging COBOL applications under z/OS UNIX too when running with VisualAge for COBOL Standard V2.1. However, because the Debug Tool has not been tested with COBOL/OE programs, there currently is no support offered.

C/C++ Portability

Here are some portability and compatibility considerations.

There are some **C/C++ language differences** that you might encounter when using z/OS's C/C++:

- Theconst qualifier may be regarded more strictly when considering if two function types are compatible.
- char versus. unsigned char or signed char are not considered equivalent. That is, IBM's default is that char is an unsigned char, yet char and unsigned char are not always compatible. You can control it with the preprocessor directive:

  ```
  #pragma chars (signed)
  ```

  or

  ```
  #pragma chars (unsigned)
  ```

- You cannot initialize static or extern variables, except with constant expressions, which can also include references to the address of previously defined static or extern variables. This is only a C problem;, you can use other types of initializers in C++.
- Template support:
  - IBM does not ship a Standard Template Library, but has several versions downloadable for free. IBM has a set of C++ Class Libraries.
  - IBM's template support generally works much more satisfactorily when using automatic template generation (to prevent getting large modules). This requires the templates be implemented using a specific file naming convention.
  - Symbol resolution from autocall libraries will not replace a generalization (a function built using a template) with a matching specialization. That is, the generalization satisfies the reference, and so autocall is not performed. If the specialization is brought in for another reason (either explicitly or via autocall), it should have higher precedence over the generalization.

Header Files

If an application on a UNIX system is not POSIX- or XPG4-compliant, then you may not be able to just move it to an z/OS system and expect it to compile. Applications that are not POSIX- or XPG4-compliant may include headers that are

not supported on z/OS. Porting an application that does not conform to those standards requires that you inspect any headers that may not be present on an z/OS system and determine whether or not the application really requires them. As you know, headers can include all kinds of things, from macros that simply exist for convenience to prototypes for functions that may or may not exist on a particular UNIX system.

We have accounted for the fact that some non-standard header definitions will be encountered, by providing compiler options for deciding that let you compile a given application against a specific Open standard (for example, _OPEN_SOURCE) or against historical UNIX (_ALL_SOURCE).

Also, we have a web page that lists headers that are not available on an z/OS system and some possible workarounds.

C++ Function Pointers for X11 callbacks

Editor's note: Here is advice from a programmer who is porting an application that uses an X-Windows GUI.

Our application uses a lot of C++ code, much of which is used for X11 callback procedures (and other things such as event handler procedures). The application compiles and links quite nicely but fails miserably with a segmentation violation on any callback/event handler/etc. call. We learned that IBM's X11 support requires the callback type procedures to have C linkage. Here are some possible solutions for a situation where C++ code needs the compiler to use C linkage:

- Declare the functions extern ″C″. Using an extern ″C″ wrapper around the function declarations is more portable. If you are writing code for multiple platforms, use this approach.

  Make the function pointer a typedef with extern ″C″ wrapped around it. Then use the typedef in the structure.
- Force C linkage to these procedures by using the **__cdecl** modifier in the function declarations and function definition. This approach is more convenient because it requires fewer code changes, but it is less portable.
- If the funtion is a member of a class, declare the function as static __cdecl. The static declaration should be required on most platforms to keep the class instance's ″this″ from being passed as an argument to the function. __ cdecl is required on z/OS to tell the compiler to generate a function that uses standard C argument and stack manipulation conventions.

For an example of using **__cdecl**, in my C++ module I would change this declaration from:

```
void myActivatePBCallback(Widget w, XtPointer cd, XtPointer cb);
```

to this coding:

```
void __cdecl myActivatePBCallback(Widget w, XtPointer cd, XtPointer cb);
```

Note that you will have to typecast the function in the XtAddCallback() call as XtCallbackProc. You can extrapolate what to do for event handler procedures and the like.

Use of **__cdecl** is allowed on member and non-member functions that are static or non-static, according to the *OS/390 V2R4 C/C++ Language Reference*. This __cdecl support works for OS/390 V2R4 and R5. It was added to OS/390 V1R3 via a PTF (UQ04420).

If you declare the functions extern ″C″ and they are declared static (having C++ file scope), you will probably have to create a separate function to call the static callback function. For example:

```
/*—–-
Use  (XtCallbackProc)myCallback in the XtAddCallback call because it
will have C linkage. Have myCallback call the original callback
(orig_c++_callback) in the C++ module.  Then X11 will be happy,
otherwise we get a segmentation violation when the callback is called.
—–-*/
static void orig_c++_callback(Widget w, XtPointer cb, XtPointer cd);
extern "C" void myCallback(Widget w, XtPointer cb, XtPointer cd)  {
    orig_c++_callback(w,cb,cd);
}
```

For details about **__cdecl**, please refer to the *OS/390 V2R4 C/C++ Language Reference*.

Error Handling

Editor's note: Here is a nugget from a programmer who was porting code that can be called by another program.

Combining both C++ exception handling (try/catch blocks) and C error handling (signals) in your code can give unpredictable results. Because our customers can write C++ code with try/catch error handling that calls our product code, we decided to use the C setjmp() and longjmp() functions to do internal error handling — instead of using signal handlers. Fortunately, we already have had to do this for other platforms, so it was not a difficult configuration change to make.

For more information about this, see the *C/C++ Programming Guide* and the *C/C++ Language Reference*, where they discuss exception handling.

Developing a dynamic link library (DLL)

A program can be made up of more than one executable, each separately built and linked. A dynamic link library (DLL) is an executable containing functions and/or variables that are made acessible to other executables (via export). The executable that are to use those functions and variables identify at link-edit time that they are to be resolved dynamically (import). All the executables that are dynamically linked together comprise a DLL application.

In a DLL application, external function and variable references are resolved dynamically at run-time, rather than statically at link-edit time. All that is done at link-edit time is to identify the DLL from which they are to be resolved. The actual resolution of the function or variable happens at run-time.

The object code generated by the z/OS C++ compiler is always DLL code. To generate DLL object code with the C compiler, use the DLL option. For more information about compiler options for DLLs, see the *z/OS C/C++ User's Guide*.

Building a C DLL

To build a simple C DLL,

1. Write code using the #pragma export directive to export specific external functions and variables as shown here:

   ```
   #pragma export(bopen)
   #pragma export(bclose)
   #pragma export(bread)
   #pragma export(bwrite)
   int bopen(const char* file, const char* mode) {
     ...
   }
   int bclose(int) {
     ...
   }
   int bread(int bytes) {
    ...
   }
   int bwrite(int bytes) {
    ...
   }
   #pragma export(berror)
   int berror;
   char buffer[1024];
    ...
   ```

   Note: An alternative to using #pragma export in your code is to use the EXPORTALL compiler option to export all defined functions and variables with external linkage in the compilation unit. With EXPORTALL, all the defined functions and variables with external linkage will be accessible from this DLL and by all users of this DLL. However, exporting all functions and variables has a performance penalty, especially with IPA.

2. Compile with the DLL compiler option (whether you use #pragma or exportall). For example,

   ```
   c89   -W c,dll
   ```

   When you specify the DLL compiler option, the compiler generates special code when calling functions and referencing external variables. It is best to have all parts of the DLL application compiled DLL. You need the DLL option to import — that is, reference functions and variable using dynamic linkage.

3. Binding: When you link, use this option:

   ```
   c89   -W l,dll
   ```

   This produces a .x file as well as a DLL executable. The .x file, a definition side-deck, contains the IMPORT control statements that let DLL applications reference the exported functions and variables. For example:

   ```
   c89  -o  pgmb -Wc,dll,exportall -Wl,dll pgmb.c
   c89  -o  pgma -Wc,dll pgma.c pgmb.x
   ```

   You must provide this generated definition side-deck to all users of the DLL.

   Without the -W l,dll option (which allocates the definition side-deck), if you have exported symbols, you get a warning and no .x file.

Building a C++ DLL

To build a simple C++ DLL,

1. Write code using the _Export keyword or the #pragma directive to export specific functions and variables. Ensure that classes and class members are exported correctly, especially if they use templates.

   For the _Export keyword:

   - Do not inline the function if you apply the _Export keyword to the functions declaration.
   - Always export constructors and destructors when using _Export
   - Apply the _Export keyword to a class

   Note: An alternative to the _Export keyword or the #pragma directive is the EXPORTALL compiler option. If you use EXPORTALL, you do not need to include #pragma export or _Export in your code. Exporting all functions and variables has a performance penalty, especially with IPA. There is some bind-time overhead and some larger structures are built for the DLLs (import-export table). Subsequently, initialization of a program importing from the DLL takes longer, since a bigger table has to be searched to dynamically link to the functions/variables. Compiler options are described in the *C/C++ User's Guide*, and #pragma directives are described in the *C/C++ Language Reference*.
2. Compile as you would any C++ program. For example,

   ```
   cxx  -W c
   ```
3. Binding your code: See the discussion above for binding C code.

For a complete discussion of building and using Dynamic Link Libraries (DLL), see the *C/C++ Programming Guide*.

X-Windows support

A user accessing the shell from a workstation or an X-terminal running an X-Window server can run an X-Window application from the shell. An X-Window application needs the TCP/IP address and display identifier for the workstation or X-terminal. For further information, see *z/OS SecureWay Communications Server: IP Programmer's Reference*, formerly known as the *IBM TCP/IP for MVS: Programmer's Reference*, SC31-7135.

man pages

z/OS UNIX does not have an nroff formatter. If your program has man pages, you will need to:

1. Take the nroff source to another system (for example, AIX) and format it there.
2. Install the formatted files in the /usr/man/C/cat1 directory (for example, /usr/man/C/cat1/xxxxx.1).

   Note: If you want to install the pages in another directory (for example usr/local/man/cat1/xxxxx.1), then users will need to set MANPATH accordingly. For example:

   ```
   export MANPATH="/usr/man/%L:/usr/local/man"
   ```

Prior to OS/390 Release 7, system man pages were distributed in post-processed (formatted) form. As of OS/390 Release 7, most of the z/OS UNIX system man pages are distributed as softcopy book files.

gnu utilities

IBM's business partner Mortice Kern Systems Inc. (MKS) provides and services several of the key GNU products and some other UNIX utilities. The MKS GNU products should be equivalent with GNU products running on other platforms.

Time management

z/OS does not support its time being changed via some external application request. z/OS does support the External Time Reference (sysplex timer) which is used to keep all the clocks for systems in the sysplex synchronized. In general, when z/OS is included in a network, the sysplex timer gets the time from a government radio transmission and keeps the sysplex in synch with official time. The z/OS can be used as the time source for other systems connected to it in the network. Some options are:

- DCE running under z/OS UNIX provides some capability of logically synching time between applications running on different platforms.
- **inetd** alone gives you a time service (see RFC868). After it is set up (mainly /etc/services), you can use setclock reading the time from an MVS or z/OS system.

The Network Time Protocol, used on the Internet to distribute accurate time information that can then be used by systems on the internet to set their clocks, is not implemented on z/OS.

Chapter 6. Security considerations

Certain functions in most UNIX systems require that special privileges be set before a user is authorized to execute these functions. Because z/OS UNIX System Services (z/OS UNIX) use's the traditional MVS security products along with UNIX permission-bit security, there are additional things that you need to be aware of.

If your application performs a particular function that MVS deems to be privileged, your application will have to be executed out of a special MVS data set called an MVS authorized program library. Many MVS system functions, such as entire supervisor calls (SVC) or special paths through SVCs, are sensitive. To avoid compromising the security and integrity of the z/OS system, access to these functions must be restricted to authorized programs. Programs that use the setuid(), seteuid(), and the passwd() functions have to be link-edited and executed from an MVS authorized program library.

z/OS UNIX security handling is different from typical UNIX security in these areas:

- Users and passwords
- Security implications of programs running in the HFS
- Daemon program setup
- Enabling thread-level security for servers

There are also differences in implementation of networking security (for example, no rhost support). For more information, see *z/OS IBM Communications Server: IP Migration Guide*, and *z/OS IBM Communications Server: IP Configuration Guide*.

Users and passwords

UNIX uses the /etc/passwd file to keep track of every user on the system. /etc/passwd contains the username, real name, identification information, and basic account information for each user.

With z/OS UNIX system services, the security product (for example, RACF, Top Secret, or ACF2) is used to keep track of every user on the system. There is no /etc/passwd file.

Users and their UIDs and passwords, and groups and their GIDs are defined to the security product. Each z/OS UNIX user has an OMVS segment defined, which allows the user to invoke the shell.

Table 1 provides an overview of how security for user identity is handled on z/OS UNIX.

Table 1. Managing user identities

Category	UNIX	z/OS UNIX
User identity	Users are assigned a unique UID, a 4-byte integer, and user name.	Users are assigned a unique user ID with an associated UID.

© Copyright IBM Corp. 1998, 2001

Category	UNIX	z/OS UNIX
Security identity	UID	UID for accessing traditional UNIX resources and the user ID for accessing traditional z/OS UNIX resources
Login ID	Name used to locate a UID	Same as the user ID
Special user	Typically on UNIX systems, there is a single ROOT userid. All users that need to have ROOT authority know and share the password for the ROOT user.	Multiple user IDs can be assigned a UID of 0 or users can be permitted to BPX.SUPERUSER.
Data set access	Superuser can access all files.	Superuser can access all HFS files; data sets controlled by RACF profiles.
Identity change from superuser to regular user	Superuser can change the UID of a process to any UID using **setuid()** or **seteuid()** functions.	There are two options: 1. If the BPX.DAEMON FACILITY class profile is not defined, the superuser can change the UID of a process to any UID using **setuid()** or **seteuid()** functions. 2. Or, the superuser must be permitted to the BPX.DAEMON FACILITY class profile in order to change UIDs.
Identity change from regular user to superuser	**su** shell command allows change if the user provides root's password.	**su** shell command allows change if the user is permitted to the BPX.SUPERUSER FACILITY class profile or if the user provides the password of a user with a UID of 0.
Identity change from regular user to regular user	**su** shell command allows change if user provides password.	**su** shell command allows change if user provides password.
Terminate user processes	Superuser can kill any process.	Superuser can kill any process.
Multiple logins	Users can login to a single user ID multiple times.	Users can rlogin multiple times to a single user ID and logon once to TSO/E at the same time.
Login daemons	**inetd**, **rlogind**, **lm**, and **telnetd** process user requests for login. A process is created with the user identity (UID).	Users can log on to TSO/E or login using one of the login daemons. In all cases, an address space is created with both an MVS identity (user ID) and a UID.

Security implications of programs running in the HFS

On z/OS, executable programs are generally categorized as coming from authorized or unauthorized libraries.

Programs in authorized libraries are considered safe for anyone to run. That is, the code should be free of viruses and should uphold the integrity and security classification of the operating system.

Programs in unauthorized libraries can be further divided into:

- System-controlled libraries, which are protected from general user modification. A system-controlled library could be any library with a profile that prevents the average user from modifying it. This is different from ″security-product controlled,″ which is a concept provided only by RACF, which calls it ″program-controlled″.
- Libraries that are not system-controlled. Libraries that are not system controlled are not considered safe for anyone to run. This code is generally a local version of a program that the owner has created or modified. Users with special privileges must use caution when running such programs. If a programmer with RACF SPECIAL authority or authority to update APF-authorized libraries runs a program from an unauthorized library, it is possible for the program to take advantage of the caller's authority to compromise the integrity of the system.

In addition to the basic concepts described above, there are further considerations when combining traditional z/OS services and z/OS UNIX.

Authorizing individual programs

The entire HFS is considered to be an unauthorized library.

You can authorize individual programs within the HFS as APF-authorized (authorized by the Authorized Program Facility) by setting the APF extended attribute for the file. HFS programs that are APF-authorized behave the same as other programs that are loaded from APF-authorized libraries. If a program running in an APF-authorized address space attempts to load a program from the HFS that does not have the APF-extended attribute set, the load is rejected. This applies to non-jobstep exec, local spawn, attach_exec, and DLL loads.

This is consistent with the way that Contents Supervisor rejects requests to LINK, LOAD, or ATTACH unauthorized programs from an authorized environment.

In order to run a program from the HFS in an APF-authorized address space, you have two choices:

- You can link-edit the program into an APF-authorized library and use the chmod command to turn on the sticky bit for the file in the HFS. When the sticky bit is set for a file, UNIX System Services searches for the program in the user's STEPLIB, the link pack area, or the link list (LNKLST) concatenation.
- You can use the extattr shell command for regular files to authorize a program as capable of running APF-authorized

APF-authorized libraries are not necessarily in the user's search order. You need to talk to the systems programmer about copying the program into an authorized library, and making the authorized library available in a STEPLIB, the link pack area (LPA), or the linklist data set.

If an APF-authorized program is the first program to be executed in an address space, then you also need to set the Authorization Code to 1 (AC=1) when your program is link-edited. If a program is loaded into an APF-authorized address space but is not the first program to be executed it should not have the AC=1 attribute set.

Previously, **dbx** could not be used on programs running in an APF-authorized address space. With OS/390 V2R5, having permission to the BPX.DEBUG FACILITY class profile allows you to debug APF-authorized programs, using ptrace (via **dbx**).

Authority checks for HFS files: To check a user's authority to access HFS files, the system uses:

- The user's effective UID
- The user's effective GID
- The GIDs for the supplemental groups, if list-of-group checking is active. When RACF list-of-groups checking is active, a user can access an z/OS UNIX resource if it is available to members of any group the user is connected to, if the group has a GID in its RACF group profile. The additional groups are called supplemental groups. To activate the RACF list-of-groups checking, specify the GRPLIST parameter on the RACF SETROPTS command. The maximum number of supplemental groups that can be associated with a process is 300.

The system sets the UID and GID of a file when it is created:

- The UID is set to the effective UID of the creating process
- The GID is set to the GID of the owning directory

Daemon program setup

You may be porting a program that uses daemons. On z/OS UNIX, daemons run authorized (they have superuser authority) and can issue authorized functions such as the following to change the identity of a user's process:

- setuid()
- seteuid()
- __spawn()

You have the choice of running daemons with regular UNIX security or with z/OS UNIX security. If you require a high level of security in your z/OS UNIX system and do not want superusers to have access to such z/OS UNIX resources as SYS1.PROCLIB, contact your system programmer. The MVS user ID that will run the daemon needs to be RACF-permitted to the BPX.DAEMON FACILITY class profile. The chapter "Setting Up for Daemons" in *z/OS UNIX Planning* explains how the system programmer can set up a BPX.DAEMON or BPX.SERVER FACILITY class for more strict security and control over superusers.

Vendor-written programs that need daemon authority

Daemon authority is required only when a program does a setuid(), seteuid(), setreuid(), or spawn userid to change the current UID without first having issued a __passwd() call to the target. If a program comes from a controlled library and knows the target userid's password, it can change the UID without having daemon authority. See *z/OS C/C++ Run-Time Library Reference* for more information about the __passwd() function.

If you are a vendor shipping code that needs daemon authority, your responsibilities are as follows:

- Create the daemon program that invokes setuid() or seteuid().
- In your installation logic, install the executable program in the HFS. Provide directions on how to use the **extattr** command to mark the program in the HFS as program-controlled. If it is an SMP/E installation, SMP/E can set the program-controlled attribute for the program in the HFS.

  If your program uses DLLs, also install the DLL executables in the HFS and mark them as program-controlled.

  An alternative approach is to install the program in both the HFS and a load library that is marked program-controlled. Set the program-controlled attribute on the HFS file. Again, if your program uses DLLs, the DLL executables also need to be program-controlled.

  Programs in the Link Pack Area (LPA) are automatically program-controlled. Placing large modules in LPA can improve performance and reduce consumption of system resources. See the topic ″Moving HFS Executables into the Link Pack Area″ in *z/OS UNIX Planning* for details on this approach.

Documentation suggestions:

1. Document the requirement to assign to the daemon a userid that has a UID of 0.
2. Document the requirement to permit the daemon to the BPX.DAEMON FACILITY class profile.
3. Document how to use the **extattr** command to mark the executables installed in the HFS as program-controlled.
4. Document how to start the daemon from /etc/rc or as a started procedure.

Enabling thread-level security for servers

For a discussion of how many threads you can run in an address space, see the topic ″Limitation on the number of threads″ in the Process Management chapter.

If you decide to run the clients on threads, and you want the code to run with the client's identify, you need to use the proprietary function pthread_security__np(). This function creates or deletes a thread-level security environment for the calling thread. UNIX does not have the concept of thread security.

If a server application uses pthread_security_np(), your system programmer needs to authorize the application to create thread-level security environments. z/OS UNIX provides services for servers written in C to create task-level security without being APF-authorized. The chapter ″Enabling Thread-Level Security for Servers″ in *z/OS UNIX Planning* details the steps the system programmer must take.

Chapter 7. Compiling

After z/OS UNIX System Services has been installed, ask your systems programmer:

- To run our setup checker, which can be downloaded from the z/OS UNIX Web site at http://www.ibm.com/s390/zos/bpxalsvp.html. Among other things, the setup checker verifies that you can compile and run a program.
- To follow the tuning targets guidelines for compile-intensive systems for the release of z/OS that you are using.

  For example, to improve performance, make sure that cc/c89/c++ do not use a shared address space (local spawn). As described in detail in the Performance chapter of *z/OS UNIX Planning*, you can either use extended attributes to mark the cc/c89/c++ files as ineligible for local spawn, or put them in LPA.

The c89/cc/c++ command is the interface to the z/OS C/C++ compiler, the prelinker, and the binder for z/OS UNIX. The c89/cc/c++ command can be invoked directly from the shell or a batch job. For each release of z/OS, there is a default compiler for c89, cc, or c++ (cxx). Optionally, you can use a non-default compiler.

If you have loaded the source code into the HFS, customized the c89/cc/c++ utility for your shell session, and taken a look at your makefiles and made any necessary changes, it is time to start compiling your code. You can run your code through a code checker or lint filter that uses the z/OS UNIX header files to possibly flag any nonconforming and unsupported constructs.

If you run with the c89/cc/c++ default settings, the following are true:

- C source file names end with the .c suffix
- C++ source file names end with the .C suffix
- Archive file names end with the .a suffix
- The -c option of the c89/cc/c++ command specifies that only compilations will be done
- Unless you name the executable file with a -o file, the default name is a.out.

The following topics explain more about compiling in the z/OS UNIX environment:

- “Using make” on page 38
- “Libraries for functions and headers” on page 38
- “Ordering options and operands” on page 39
- “Exporting functions and variables” on page 39
- “Compiler Options” on page 39
- “Conditional compilation” on page 41
- “c89 access to socket header files” on page 41

We have a compiler web page that lists various compiler topics, and hints and tips for using the compiler, and a Compiler FAQ web page.

© Copyright IBM Corp. 1998, 2001

Using make

Larger applications will probably have a collection of makefiles that are used to build the object files and executables from source files.

make is a recipe-driven utility for managing the compilation process. The programmer specifies relationships between programs; make ensures that the compiles, links, etc. are done correctly and at the right time. The make command calls the c89/cc/c++ utility by default to compile and link programs specified in your makefiles. make implementation varies across UNIX platforms. z/OS UNIX make tends to be less tolerant of non-standard files than other makes. With each new release of z/OS, there are fewer differences between z/OS UNIX make and other makes. See *z/OS UNIX Programming Tools* for more information on the make command, and *z/OS UNIX Command Reference* for the syntax of the make command.

.POSIX makefile special target: In your makefile, specify the .POSIX special target. This causes make to process the makefile as specified in the POSIX.2 standard. This special target must appear before the first noncomment line in the makefile. Do not associate any prerequisites or recipes with this target. The .POSIX target:

- Causes make to use the shell when running all recipe lines (one per shell).
- Disables any brace expansion (set with the .BRACEEXPAND special target).
- Disables metarule inferencing.
- Disables conditionals.
- Disables make's use of dynamic prerequisites.
- Disables make's use of group recipes.

A ported version of gnumake is available from the MKS web site. gnumake executes commands in a makefile to update one or more target names, where name is typically a program.

Libraries for functions and headers

Language Environment Runtime Library functions are kept in MVS data sets rather than an HFS archive library like /lib/libc.a. In z/OS UNIX, the default directory to search for archive libraries set by the {LIBDIRS} environment variable is /lib followed by the /usr/lib directory. This default is set to be consistent with other UNIX implementations, but the library functions are not contained in those directories. Instead, the MVS data sets that are installed with Language Environment are used to resolve library functions. For more information on which Language Environment data sets are searched, see the description of {_PLIB_PREFIX} in *z/OS UNIX Command Reference*.

Functions used by a program should be declared in the program. This is true whether you create them or you use system calls provided by the platform If you do not declare the functions, the C compiler will create default declarations that may or may not operate correctly.

System calls do not need to be declared explicitly, as their declarations are part of the header and include libraries that you specify to gain access to system calls. These libraries are generally installed in the directory /usr/include in files with a .h suffix on most systems. z/OS UNIX header files are installed in MVS data sets, but generally they are also installed in the HFS and you can grep them. There are

a few exceptions, such as headers for C++ classes, which are not installed in the HFS for OS/390 V2R5M0 and below; for OS/390 V2R6M0 the headers for C++ classes are generally installed in the directory /usr/lpp/ioclib/include.

Ordering options and operands

In accordance with the POSIX.2 and XPG4 standards, options and operands of utilities cannot be mixed: all options must appear before operands. You cannot put the -o option at the end of the command. The X/Open compliance suites specifically test to ensure that c89 does not allow the mixing of operands and options.

Because the mixture of operands and options is a common practice on Unix platforms, z/OS UNIX lets you enable this with an environment variable (a different one for each utility). If you typically use this syntax or use makefiles that imply this practice, you may want to add one or more of these variables to your $HOME/.profile:

```
export _C89_CCMODE=1
export _CXX_CCMODE=1
export _CC_CCMODE=1
```

Exporting functions and variables

To avoid exposing unnecessary external variables and functions, selectively export external variables and functions by using **#pragma export** or the **_Export** keyword for C++ instead of the EXPORTALL compiler option.

If you export variables or functions unnecessarily, it has the following effects:

- It increases the size of your DLL, thus increasing the load and initialization costs when the DLL is first referenced by another program.
- It severely limits IPA optimization (global variable coalescing and pruning of unreachable or 100% inlined code do not occur).

Compiler Options

LANGLVL(EXTENDED): The C library contains several functions that are extensions to the SAA CPI Level 2 definition. These library functions are available only if the LANGLVL(EXTENDED) compile-time option is in effect.

LANGLVL(EXTENDED) can also make the compiler more ″forgiving″ and allow you to compile code that it ordinarily would complain about (type mismatches, etc.).

To specify this option to c89/cc/c++, use:

```
-W0,langlvl(extended)
```

but take care to protect the parentheses from shell interpretation. One technique to avoid this problem is to specify this option implicitly for all compiles (either in your profile or from the command line):

```
export _CC_OPTIONS='-W0,langlvl(extended)'
```

For enhanced diagnostic messages: use the -g option for compile/linkedit.

For automatic inlining: use either the -O or -2 option for compile/linkedit. This is recommended for any performance-sensitive final code.

When using DLLs: it is recommended that you use the compile option `-W0,dll`

For listings: use the -V option for compile/linkedit.

To indicate that symbols required by POSIX.1, POSIX.1a, POSIX.2 are made visible: use -D_OPEN_SYS. Any symbols defined by the _OPEN_THREADS macro are also made visible. Additional symbols can be made visible if any of these standards explicitly allows the symbol to appear in the header in question or if the symbol is defined to be an z/OS UNIX extension.

For more information about the feature test macros, see the *z/OS C/C++ Library Reference*.

Extension options

These are the extension options that c89 uses:

Option	Meaning
-0, -1, -2	Representing optimization levels. The standard only calls for -O.
-V	To produce "listing/map" type information.
-v	-v is commonly "verbose". AIX also has a -v. c89 -vv (-v more-than-once) is like AIX -#.
-+	Indicates all files are C++. It is common for C++'s to have an option that says all files are C++, regardless of their names (even file.c). AIX also has this option.
.C	Capital C for C++ is also used on AIX. OS/2 couldn't without requiring HPFS; Windows couldn't until Windows 95, so .CPP is typical on those (FAT file) systems.

Option	Meaning
-W	c89 lets you pass anything on -W to underlying programs (compiler, assembler, Binder...). This is what it was intended to do! c89 does have some non-standard sub-args, however. The X/Open ones are: • **p = C/C++ preprocessor** z/OS does not have a separate preprocessor program from the compiler, so this isn't used on z/OS. That is, c89 invokes the compiler driver program, and doesn't distinguish preprocessor from optimizer options. • **0 = C/C++ compiler** • **2 = C/C++ optimizer**. We don't have a separate optimizer program from the compiler, so this isn't used on z/OS. (Same as p). • **a = assembler** • **l = linker** c89 allows: • **c = compiler**. Identical to 0; AIX has c but not 0. • **I = enable IPA optimization**. Enables optimization and passes subsequent options as IPA (sub)options.

Conditional compilation

One of the problems programmers have is writing code that can work on many different machines. In theory, C code is portable; in reality, many machines have little differences that must be accounted for. The compiler allows the programmer great flexibility in changing the way code is generated through the use of conditional compilation. If you find that a function works differently or a header file declaration is different under z/OS UNIX, you can insert #ifdef and #endif statements in the code. For example:

```
#ifdef __MVS__
#include <stdlib.h>
#else
#include <malloc.h>
#endif
```

The __MVS__ macro is available in all the z/OS C/C++ compilers. It was introduced with IBM C/C++ C++/MVS V3R1 and in IBM C/C++ C/MVS V3R2. It is not available in AD/Cycle C/370 V1R2 or in IBM C/C++ C/MVS V3R1.

c89 access to socket header files

To get access to the z/OS UNIX socket header files when compiling, consider using the following options when you invoke the c89/cc/c++ utility:

- -DMVS

This enables logic in include files that is unique to MVS (such as referencing variables defined in the DLL).

- _OE_SOCKETS

  Defines a BSD-like socket interface for the function prototypes and structures involved. This option is useful if you are porting a BSD4.3 conforming application to z/OS UNIX. Because XPG 4.2 sockets, _XOPEN_SOCKETS, contain some prototypes that require const on some input parameters, using this option instead would save you from editing the ported source to changes some of the socket or IP address resolution calls. _OE_SOCKETS can be used with _XOPEN_SOURCE_EXTENDED 1 and the XPG4.2 socket interfaces will be replaced with the BSD-like interfaces. An application cannot specify _OE_SOCKETS with _OPEN_SOCKETS. A compile time error message will be generated.

- _OPEN_SOCKETS

  Defines a BSD-like socket interface for the z/OS UNIX System Services Application Services (FMID HOT11x0). This option cannot be used with either _XOPEN_SOURCE_EXTENDED 1, _OE_SOCKETS, or _OPEN_SOURCE=2. A compile time error message will be generated.

  Use of this feature test macro implies that the application is using the z/OS UNIX System Services Application Services (FMID HOT11x0) product implementation of the sockets runtime library, and must comply with several other requirements, such as header concatenation and inclusion of HOT11x0 unique headers.

- _ALL_SOURCE

  Defines all of the functionality that is currently available with z/OS UNIX, including XPG4, XPG4.2, and all of the z/OS UNIX extensions. In addition, defining _ALL_SOURCE makes a number of symbols visible that are not permitted under ANSI, POSIX or XPG4, but which are provided as an aid to porting C-language applications to z/OS UNIX.

Note: If a source program can be ported to z/OS UNIX just by defining _ALL_SOURCE, then it is possible to set this option on the command line invocation of the compiler:

```
c89 -D _ALL_SOURCE ....
```

If you do not specify any option, you will be using Universal UNIX (UU) sockets that are compatible with C++.

Chapter 8. Debugging

The topics we will discuss are:
- Runtime environment
- Debugging
- Dumps

Runtime Environment

When you are ready to test an executable, but before you start the application, check the CEE runtime definitions and define appropriate environmental variables. See *Language Environment for z/OS Programming Reference* for the definition of runtime options and how to specify them.

Debugging

When application testing begins, you will need to run it under dbx. You change the makefile, adjust compiler optimization from -2 , -1, or -0 to -g and remake the application.

You need some preparation to run under dbx. To use dbx as a source code debugger, you must tell it where the source code resides. You can enter this each time, or create a file containing the information to be read in by dbx. Assuming that the previous problem was a bad parameter to a runtime function, you can set a breakpoint at the API call. Then you can examine and modify parameters to determine the cause of the error. After repairing the error, you repeat the debugging process as necessary.

The dbx debugger can be configured to display ASCII coded text, which was indispensable when compiling in ASCII mode and using the libascii package.

To debug application daemons started by inetd, you can use dbx's ability to attach to a running process.

See the dbx home page for more information on dbx. Even undocumented commands such as storage displays in ASCII are there.

ASCII characters and strings

By default the dbx ″p″ command assumes characters and strings are encoded in EBCDIC. However, it is possible to use ASCII encoding in a program via a compiler option and to use the libascii library from the Tools and Toys page, and it can be a useful approach in porting.

If your program is compiled in ASCII mode and you want to get meaningful output from the ″p″ command, you can tell dbx interpret strings and characters as ASCII with:

```
set $asciichars
set $asciistrings
```

© Copyright IBM Corp. 1998, 2001

Here is an extract from a ~/.dbxinit file that makes it easier to switch back and forth between ASCII and EBCDIC:

```
# Change to ASCII interpretation.
alias asc\
 "print 'Setting ASCII mode...'; set $asciistrings; set $asciichars"
# Change to EBCDIC interpretation.
alias ebc\
 "unset $asciistrings; unset $asciichars; print 'Setting EBCDIC Mode...'"
# Start in EBCDIC.
asc
```

Debugging a running program

You can debug a running program with

```
wdbx -a <pid>
```

where pid is the process id of the program. This is very useful for debugging daemons.

Debugging authorized programs

In OS/390 V1R2 authorized programs have to be loaded from an MVS dataset rather than a HFS executable. This is accomplished by setting the sticky bit on the HFS executable, which causes an MVS program of the same name to be loaded instead. dbx isn't supposed to work with these programs, but the beta version doesn't detect this situation, so it may appear to be working and run for a while before failing mysteriously. You can easily be fooled into thinking there is something wrong with your application.

In OS/390 Release 7, dbx provides a sticky bit debug flag.

Other debug methods

There are other methods that can be used in the debugging effort. You can use the standard printf() to output information at critical points. If the application is running in a mode where stdout or stderr is unavailable, you can open a file and use fprintf() or similar functions.

Dumps

You may need to gather additional information that is not available to one of the debuggers. Fortunately, there are a few other methods to gather diagnostic information. Allocating a file to the SYSMDUMP, SYSABEND, or SYSUDUMP indicates to z/OS UNIX that it is to take a dump in the event of an error. You need to include TERMTHDATA(UADUMP) in the CEE runtime options. This will cause the runtime to take a CEEDUMP to $HOME, then issue a 4094, 4091, or 4039 user abend.

You can include the CHNGDUMP command (z/OS Change dump command) to add more areas in the dump. A system programmer or operator must issue this command. In cases where additional data not normally available to user programs needs to be dumped, it may be necessary to use the SLIP command to set a SLIP trap to issue a SDUMP during a particular event (such as instruction fetch, storage modification, or abend with filtering). Like the change dump command, the slip command is also an authorized command. (For information about these commands, see *z/OS MVS System Commands, GC28-1781*. A user abend such as a 4094, 4091, or 4039 can be used as a trigger for the SDUMP via the SLIP command.

If there are z/OS UNIX problems associated with the problem being tracked, you may need to include z/OS UNIX information in the dump. You can print and read the SYSUDUMP and SYSABEND dumps without additional processing. (See *z/OS MVS Diagnosis: Tools and Service Aids* for information on getting a dump, and *z/OS MVS Diagnosis: Procedures* for information on diagnosing the dump.) However, you must process SYSMDUMP and SDUMP (slip) dumps using IPCS. For z/OS UNIX-related problems such as signaling, process synchronization or other problems, you may find it necessary to check the status of the application's processes and threads. You can use the OMVSDATA IPCS subcommand to give indications of process status. This command is documented in *z/OS MVS Diagnosis: Reference*.

Chapter 9. The hierarchical file system

Files in the z/OS UNIX file system are organized in a hierarchy. All files are members of a *directory,* and each directory is in turn a member of another directory at a higher level in the hierarchy. The highest level of the hierarchy is the *root directory*.

Features of the hierarchical file system include:

- A hierarchical directory structure:
  - Directory support
  - Current directory and home support
  - Absolute and relative pathnames
  - Hard links and symbolic links
- Stream files
- POSIX.1 APIs that perform file system operations on directories and stream files
- UNIX-style permissions support for security
- Local sockets support
- Extended attributes support
- Data conversion support
- Save and restore support

The file system is POSIX.1 compliant.

We will discuss these topics:

- “The Root File System and Mountable File Systems” on page 50
- “Files” on page 50
- “Executable Modules in the File System” on page 51
- “Memory-mapped Files” on page 51
- “Pathnames” on page 51
- “Code Page” on page 52
- “Setting up security” on page 11
- “Power Failures and the File System” on page 53
- “Sharing Files” on page 53
- “File Locking” on page 55
- “Opening MVS data sets from an z/OS UNIX environment” on page 55

An Introduction to the Hierarchical File System

MVS views an entire file hierarchy as a collection of *hierarchical file system data sets (HFS data sets)*. Each HFS data set is a mountable file system. DFSMS/MVS facilities are used to manage an HFS data set, and DFSMS Hierarchical Storage Manager (DFSMShsm) is used to back up and restore an HFS data set.

With DFSMS/MVS 1.5, the new HFS multivolume support works the same as any other multivolume non-VSAM SMS-managed data set. HFS data sets can now span

© Copyright IBM Corp. 1998, 2001

up to 59 volumes, with up to 255 total extents for all volumes. As users add files and extend existing files, an HFS data set can increase in size to a maximum of 123 extents:

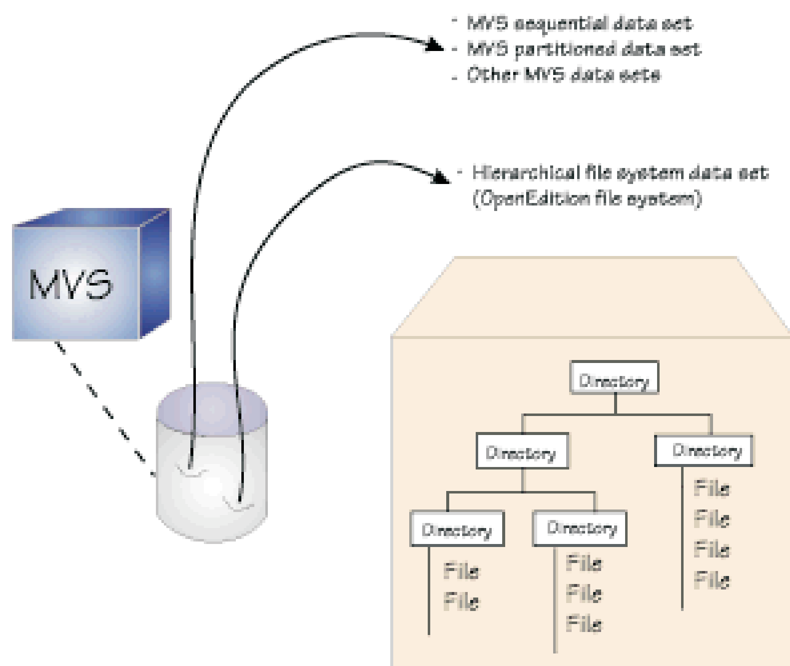
- If you allocate the HFS with a secondary allocation, it should extend when the current allocation is filled.
- If you do not allocate the HFS with a secondary allocation, it will not automatically extend, but you can use the confighfs shell utility (new in Release 7) to manually extend it.

With Release 9, there is now support for sharing a hierarchical file system in a parallel sysplex.

The root file system is the first file system mounted. Subsequent file systems can be mounted on any directory within the root file system or on a directory within any mounted file system.

A file in the hierarchical file system is called an *HFS file*. HFS files are byte-oriented, rather than record-oriented, as are MVS data sets. You can copy HFS files into MVS data sets (sequential data set, partitioned data set, or PDSE), and you can copy MVS sequential data sets or partitioned data set members into a hierarchical file system.

Figure 1. The Hierarchical File System

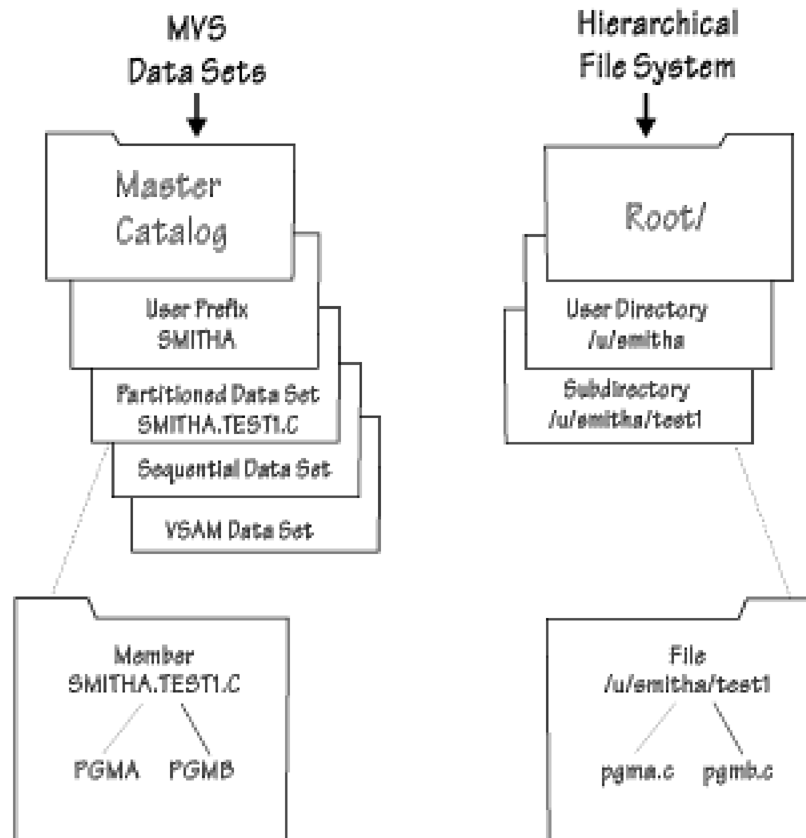


The maximum file size is the size of the largest DASD volume, minus some administrative overhead. This means you can create a file that is larger than 2 GB. From C, you cannot lseek() past 2 GB, but you can sequentially process it. The kernel assembler interfaces are geared to 8 byte length and pointer fields, but the C RTL has not yet done this.

The z/OS UNIX shell typically imposes a *line orientation* on the byte-oriented files. A *line* is a stream of bytes terminated with a <newline> character. A line terminated by a <newline> character is sometimes referred to as a record. So, there

is a single <newline> character between every pair of adjacent records. Text files use the <newline> character to delimit lines; binary files do not.

Figure 2. Comparison of MVS Data Sets and a Hierarchical File System



In Figure 2, you see that:

- The MVS master catalog is analogous to the root directory in a hierarchical file system.
- The user prefix assigned to MVS data sets is an organizer analogous to a user directory (**/u/smitha**) in the file system. Typically, one user owns all the data sets whose names begin with his user prefix. For example, the data sets belonging to the TSO/E user ID SMITHA all begin with the prefix SMITHA. There could be data sets named SMITHA.TEST1.C, SMITHA.TEST2.C, SMITHA.TEST1.LIST, and SMITHA.TEST2.LIST.

In the file system, SMITHA would have a user directory named **/u/smitha**; under that directory there could be subdirectories named **/u/smitha/test1** and **/u/smitha/test2**. We recommend that each user file system be a separate HFS data set (mountable file system).

- Of the various types of MVS data sets, a partitioned data set (PDS) is most akin to a user directory in the file system. In a partitioned data set such as SMITHA.TEST1.C, you could have members PGMA, PGMB, and so on—for example, SMITHA.TEST1.C(PGMA) and SMITHA.TEST1.C(PGMB). Likewise, a subdirectory such as **/u/smitha/test1** can hold many files, such as pgma.c, pgmb.c, and so on.

All data written to the hierarchical file system can be read by all programs as soon as it is written. Data is written to a disk when a program issues an fsync().

The Root File System and Mountable File Systems

Taken as a whole, the *file system* is the entire set of directories and files, consisting of all HFS files shipped with the product and all those created by the system programmer and users. The system programmer (superuser) defines the *root file system*; subsequently, a superuser can mount other *mountable file systems* on directories within the file hierarchy. Altogether, the root file system and mountable file systems comprise the file hierarchy used by shell users and applications.

Several types of file systems can be mounted within the file hierarchy:

- Hierarchical File System (HFS)
- System (NFS): Using NFS client on z/OS UNIX, you can mount a file system, directory, or file from any system with an NFS server within your user directory. You can edit or browse the files.
- File System (DFS): A DCE component, DFS joins the local file systems of several file server machines making the files equally available to all DFS client machines. DFS allows users to access and share files stored on a file server anywhere in the network, without having to consider the physical location of the file.
- File System (TFS): The TFS is an in-memory physical system that delivers high-speed I/O. To take advantage of that, the system programmer (superuser) can mount a TFS over the **/tmp** directory so it can be used as a high-speed file system for temporary files. (Normally, the TFS is the file system that is mounted instead of the HFS if the z/OS UNIX are started in minimum setup mode.) The TFS became available with OS/390 V1R3.

Files

There are four types of files that can exist in the HFS, in addition to directories:

- A regular file can be C source code, text, or a a printer-formatted document.
- A character special file.
- A FIFO special file is a file typically used to send data from one process to (FIFO). A FIFO special file is also known as a *named pipe*.
- A symbolic link reference to the original file. Only the original pathname is the real name of the original file. You can create a symbolic link to a file or a directory.

An *external link* is a type of symbolic link, to an MVS data set. An external link lets an NFS client use a pathname to transparently access an MVS data set. A program using the exec() family of functions or callable services can use an external link to access an MVS data set.

Users and programs create regular files, FIFO special files, symbolic links, and external links.

Files not in the HFS: There are two types of unnamed files that you may be aware of, but they do not exist in the HFS:

- An unnamed pipe.

A program creates a pipe with the pipe() function. A pipe typically sends data from one process to another; the two ends of a pipe can be used in a single program task. A pipe does not have a name in the file system, and it vanishes when the last process using it closes it.

- A socket.

A program creates a socket with the **socket()** function. A socket provides communication in two directions, in contrast to pipes, which allow communication in only one direction. The processes using a socket can be on the same system or on different systems in the same network.

Executable Modules in the File System

You can have an executable module in the HFS. To run a binary executable, a user only needs execute permission to the file. To run a shell script, REXX exec, or perl script, a user must have read and execute permissions to the file. Use chmod to set the permissions.

- For frequently used programs in the HFS, you can use the chmod command to set the sticky bit on. This reduces I/O and improves performance. When the bit is set on, the system searches for the program in the user's STEPLIB, the link pack area, or the link list concatenation. The main reason for turning on the sticky bit would be if you are using the module in many address spaces and you want the module to come out of LPA. By having the module come out of LPA, you save pagable storage by having only 1 copy and you also save the overhead of copying the module on fork. For a module like the shell, which does lots of forks, this saves the copying of 4 MB for each shell command that gets forked.
- command is used to set, reset and display extended attributes for files to allow executable files to be marked so they run APF– authorized, or as a program controlled executable, or not in a shared address space. To use this command, you must be permitted to BPX.FILEATTR.APF to mark a file as APF authorized, or permitted to BPX.FILEATTR.PROGCTL to mark a file as program controlled.

The ls shell command has an -E option which will display these attributes for a file.

- You can copy executable modules between MVS and HFS, using the OPUT, OPUTX, OGET, or OGETX commands.

Memory-mapped Files

z/OS UNIX provide memory-mapped files: the data of a file is mapped to a particular area of memory. You can access it directly in memory instead of calling read() or write(). The following functions are used to implement memory mapping:

- mmap() maps the file into memory
- msync() ensures that updates to the file map are flushed back to the file.
- mprotect() sets the protection of memory mapping
- munmap() frees the mapped file data

Pathnames

A pathname can be up to 1023 characters long, including all directory names, filenames, and separating slashes. For pathnames and filenames, use characters from the POSIX portable character set. Using DBCS data in these names is not recommended; it may cause unpredictable results.

The system performs *pathname resolution* to resolve a pathname to a particular file in a file hierarchy. The system searches from element to element in a pathname in order to find the file.

Requirement for an Absolute Pathname

In some situations, an absolute pathname is required. Table 1 (page 52) shows the absolute pathname requirements for job control language (JCL) and some TSO/E commands. These absolute pathnames require an MVS data set name to be specified in a certain way. In these situations, the maximum length of the absolute pathname is 255 characters.

Table 1. Absolute Pathname Requirements

	Pathname	Dataset name
JCL	Absolute, in single quotes	Fully qualified (no quotes needed).
ALLOCATE command	Absolute, in single quotes	Fully qualified in single quotes. If specified without quotes, the TSO/E prefix is added to the data set name. Normally the TSO/E prefix is the TSO/E user ID (this can be changed with the PROFILE PREFIX( ) command).
OEDIT, OBROWSE commands	Absolute, unless you are working in your home directory	Not applicable
OPUT, OGET commands	Absolute (unless you are working in your home directory), in single quotes	Fully qualified in single quotes. If specified without quotes, the TSO/E prefix is added to the data set name. Normally the TSO/E prefix is the TSO/E user ID (this can be changed with the PROFILE PREFIX( ) command).
OPUTX, OGETX commands	Absolute (unless you are working in your home directory)	Fully qualified in single quotes. If specified without quotes, the TSO/E prefix is added to the data set name. Normally the TSO/E prefix is the TSO/E user ID (this can be changed with the PROFILE PREFIX( ) command).

Code Page

The default code page for the shell and utilities is IBM-1047, an EBCDIC code page. There is information in the *z/OS UNIX User's Guide* on how to change the code page for the shell and utilities to a different EBCDIC code page.

Data Conversion

There are many options for converting data from one code page to another:

- Copying files between the file system and an MVS data set: The OCOPY, OGET, OGETX, OPUT, and OPUTX commands all have a CONVERT option.
- Working in the shell: you can use the iconv shell command to convert data.
- Working with C/C++: you can use the C iconv() function to convert data from one code page to another. This is described in the *z/OS C/C++ Programming Guide*.
- Working with tar files: the pax command has an option for converting between code pages.
- Using the Network File System feature: text files are automatically converted between the EBCDIC code page used in the z/OS UNIX shell and the ASCII code page used at your workstation.
- FTP automatically converts data when copied in text mode, similar to the Network File System feature.

Security for the File System

The HFS uses UNIX-style permissions for file security; there are also some extensions that are unique to z/OS UNIX. The *z/OS UNIX User's Guide* has a chapter on handling security for your files.

Your enterprise can use the SecureWay Security Server for z/OS (RACF), or an equivalent security product to provide security.

Power Failures and the File System

Should there be a power failure, you might lose recent data that is still buffered, but the file system structures, directories, inodes and such, will not be damaged. A shadow writing technique is used to ensure structural changes are always committed atomically. The HFS does its own repair, as needed, on each mount of a file system. This is based on records it keeps of changes in progress.

command and the HFS was designed so that it is not needed. The **fsck** utility generally ensures structural integrity, not data integrity.

Of course, there is always a possibility that user data, critical file system data, or the media can be damaged, so prudent backup procedures are always warranted.

Sharing Files

There are several ways to access HFS files from your workstation or mount them there. As of Release 9, there is support for sharing a hierarchical file system in a parallel sysplex. With z/OS NFS Client, an HFS can be mounted in read/write mode to multiple systems. However, performance may be an issue.

The file-sharing options are:
- “Using an NFS Windows Client” on page 15
- “LANRES and LAN Server” on page 54
- DCE Distributed File Service (DFS)
- Samba for z/OS

Using the Network File System Feature

z/OS NFS with OS/390 2.6 supports PC-NFS and NFS V3. The performance enhancements with this release make it the preferred choice over LAN Server NFS for OS/390 2.6 and beyond.

Using the Network File System feature, you can mount HFS files on an empty directory at your workstation.

To access the hierarchical file system, you first enter the **mvslogin** command, which gives you permission to use NFS (NFS is a trademark of Sun Microsystems, Inc.).

Then you enter the **mount** command to make a connection between a mount point on your local file system and a directory or file in the hierarchical file system. After a directory is mounted, you can create, delete, read, or write to a file in or below that directory in the file hierarchy; generally, you can treat a file in or below that directory as a member of your own workstation file system.

- For text files, the Network File System feature handles conversion between the EBCDIC code page used in the z/OS UNIX shell and the ASCII code page used at your workstation.
- RACF checks the authority of a workstation user to access HFS files at the host. This is based on the authority of the MVS user ID specified on the **mvslogin** command.

Locking

Locking is local to the system you are working on; it is coordinated with other local users. If remote users are accessing a file at the same time as local users, locking is not coordinated between local and remote users.

External Links

An external link is a type of symbolic link that you can use to associate an MVS data set or PDS member with an HFS pathname. The external link lets the NFS client user transparently access an MVS data set using a pathname. A program using the **exec()** family of functions or the BPX1EXC (exec), BPX1LOD (load) or BPX1SPN (spawn) callable services can also access an MVS data set using an external link.

The data set appears in a mounted HFS directory with HFS files. If you are working with both MVS data sets and HFS files at the workstation, with an external link you can have one directory for both the data sets and the files—for example, **/host**, instead of **/host/ds** for the data sets and **/host/hfs** for the files.

LANRES and LAN Server

LANRES provides disk serving, print serving, data distribution, and central administration for NetWare LAN clients connected to MVS.

Using NFS, LAN Server lets you view data from the server system. LAN Server provides file serving to OS/2 and Network File System (NFS) servers connected to zSeries servers. NFS servers can be AIX or UNIX. LAN Server also provides the capability for OS/2 and NFS clients to share a common data repository with full update capability.

LAN Services for z/OS provides rapid communication between MVS and the LAN and is transparent to the LAN clients. The LAN servers can communicate with MVS through TCP/IP or ESCON- or parallel-attached channels. LAN servers can

also use APPC as the communication protocol. The MVS system becomes a high-speed file server and allows data to be stored in a VSAM data set, which can be shared among all LAN servers. Thus disk space can be freed up on the workstation-based LAN servers, and the large capacity of MVS can relieve the constraints of those LAN servers. Users access files on the host system as if the files were on their LAN server or local disk.

LAN Server NFS is the fastest most scalable solution for NFS serving prior to OS/390 2.6. It supports NFS V2 and PC-NFS, but will never support NFS V3.

File Locking

File locking is a means to coordinate access to a file, when two or more processes access it at the same time. The hierarchical file system supports advisory locking, not mandatory locking.

Advisory locking does not prevent access to the file. Advisory locks work if each participating process ensures that it locks the file before accessing it. If the file is already locked, either the lock request fails and the requesting process has to handle the condition or the process blocks until it gains the lock. Because the locking is advisory, not mandatory, nothing forces a program to use locks or even pay attention to locks that another process might have on a file.

File locking can only be performed on file descriptors that refer to regular files.

For more information, see the section on File Locking in the fcntl() function description in the *z/OS C/C++ Library Reference* .

Opening MVS data sets from an z/OS UNIX environment

There are times when you may prefer to put your data in an MVS data set instead of the HFS.

If your application does frequent I/O, you may be able to improve its efficiency by using MVS datasets rather than HFS files. If you are doing streams I/O, you can change the fopen() to point directly to an MVS data set. This will cause the Language Environment RTL to allocate and open the dataset. Then fread() and fwrite() calls will use the appropriate MVS access method (QSAM, BSAM, VSAM) to read or write data.

The *C/C++ Programming Guide* provides all the details about how fopen() determines if a filename refers to an MVS data set or a UNIX file, but here are three rules that apply in most situations:

- If the filename begins with exactly two slashes (that's the first two bytes) but no more (not 3, 4, ... slashes), the filename is unambiguous and refers to an MVS dataset

or

- If the filename contains a slash anywhere in it, it's unambiguous and refers to an HFS file

or

- The name is ambiguous and the POSIX runtime option (and no other criteria is used, such as whether a dataset name is valid) is used to determine whether a filename is an MVS dataset or an HFS file:
- In the shell, the POSIX option is usually ON and the name refers to an HFS file.
- In TSO or BATCH, the POSIX option is OFF by default and the name refers to an MVS dataset.

If you are using the pax or tar utilities, as of OS/390 V2R8 pax and tar can read and write archive files that reside in MVS sequential or partitioned data sets.

Chapter 10. Process management

In order to set up, configure, and possibly debug z/OS UNIX application programs, such as a Webserver, you need to have a basic understanding of the process model that is used within the z/OS UNIX environment.

In z/OS, we have the following basic categories of work:
- Started tasks (MVS operator start command)
- Batch jobs (submit via JES)
- TSO/E user (logon)
- APPC/MVS transactions (CPI-C allocate)
- IMS (a started task)
- Other server or system address spaces

All work that is created through one of the above requests finally ends up in an address space that holds every piece of information that is required to describe the work at any moment in time (for example, storage control blocks, program(s) to execute, opened data sets, etc.). Though representing the current work, the address space is not the unit of work MVS dispatches. MVS dispatches a TCB (task control block), or an SRB (service request block). They are all dispatchable units that represent work that runs in an address space.

Basically this does not change in the z/OS UNIX environment, but we work with a slightly different terminology that is based on the concepts of a *process* and a *thread*.

Note: Do not run z/OS UNIX applications from CICS. Support for this capability is under consideration. You also need to be careful about using z/OS UNIX services from an APPC multitransaction environment. Both of these environments cause problems for z/OS UNIX because they change the security environment (ACEE) without giving z/OS UNIX a chance to react.

During this discussion, we will be referring to C functions and assembler callable services (also known as syscalls). Generally, whenever the C RTL (Run-time Library) needs to do something which requires being in an authorized state, a callable service is used. The C language support passes the C function to the kernel address space via an assembler callable service. The z/OS UNIX callable services are documented in *z/OS UNIX Programming: Assembler Callable Services Reference*.

When discussing process management, we will consider:

- “Processes” on page 58
- “Threads” on page 63
- “Interprocess communication (IPC)” on page 64
- “Signals” on page 67

© Copyright IBM Corp. 1998, 2001

Processes

A process maps into an MVS address space and an MVS task environment exists for the process, in terms of a task control block (TCB) and related control blocks. In addition to the TCB, the kernel address space maintains a number of control blocks that represent a process. These control blocks exist within the kernel address space; they are created when an existing standard MVS program begins using z/OS UNIX System Services (z/OS UNIX) or when a new process is created by the z/OS UNIX process creation functions. When an address space begins using z/OS UNIX services, we say the MVS address space is *dubbed* as an z/OS UNIX process. From a UNIX perspective, this means z/OS UNIX assigns a PID (process ID) to the process. Control blocks in common storage and the kernel address space are built to represent this piece of work. These control blocks build the infrastructure that z/OS UNIX uses to keep track of all you do. In addition to the process control blocks, task level control blocks are created in the user address space and in the kernel. Each task is treated the same as a UNIX thread.

There are situations where multiple processes may exist within the same MVS address space, and in such a case a process may be running as the job step task or a subtask.

An z/OS UNIX program may use a number of z/OS UNIX services to create new processes or to enable multithreading within the process itself. There are no means to prohibit creation of new processes by an application programmer (although BPXPRMXX parmlib settings can limit the number of processes).

To control processes, the following basic services are available:

- **fork() function and BPX1FRK service — Create a New Process**
  fork() replicates the current process into a child process. After the fork(), the child process is running in a new address space, with a single task and a single RB, regardless of the task environment of the parent. Key 8 virtual storage has been copied from the parent to the child address space.
- **spawn() function and BPX1SPN service — Spawn a Process**
  spawn() starts a new process, but the new child process is started with another program in the hierarchical file system (HFS), as indicated by the parent process on the spawn() call. After the spawn() call, the two processes continue as independent processes.

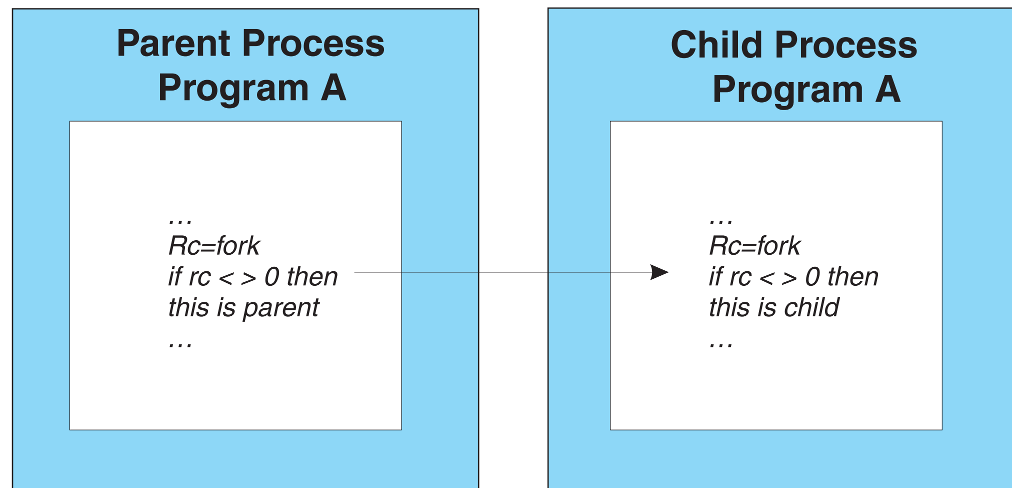
  spawn() can create a new process in a separate address space or in the same address space, depending on the setting of the environment variable _BPX_SHAREAS=YES|NO.

  If your application creates a lot of processes and you want better performance, use spawn(). Similar to fork() and exec, spawn() runs much faster and saves resources because it does not have to copy the address space. In fact spawn() can optionally place the new process in the same MVS address space, even further saving system resources. If your application is multithreaded you must use spawn() instead of fork().
- **exec family of functions and BPX1EXC service — Run a Program**
  This does not start a new process, but replaces the program in the current process with another program as indicated on the exec call.
- **BPX1ATX service — Attach a Program Residing in the HFS**
  This attach_exec service will create a new process in the same address space and pass control to a program in the hierarchical file system. No C function is provide for this service. The equivalent function is provided by spawn() with _BPX_SHAREAS=YES.

- **BPX1ATM service — Attach an MVS Program**
  This attach_execmvs service creates a new process in the same address space and passes control to a program in the normal MVS search order (Job Pack Queue, STEPLIB, LPALIB, LINKLIB).
- **BPX1SDD service — Set the Dub Default Service**
  You can call the set_dub_default service and set the default level to Process. Then any subtask making a BPX1xxx call will get dubbed as a separate process. As a separate process, each task does not share any z/OS UNIX resources with the other processes in the address space.

Forking a New process

To start a new process, a process may use the fork() service. Forking is a very well-known concept in UNIX environments, but it is not a function that directly maps into any traditional MVS system services.



If you have a process running that is executing, for example, program A, and this program calls the fork() function, the kernel address space initiates the following actions:

1. Creates a child address space. Prior to OS/390 V2R4, this was accomplished using APPC initiators. As of OS/390 V2R4, it is done using an internal Workload Manager (WLM) interface. The WLM interface will create a new address space only if the system can tolerate the additional load.
2. Copies user recovery routines and contents supervisor structures from the parent process address space to the child process address space. Private storage (stack, heap, and programs) is propagated from the parent address space to the child address space. The security environment is also propagated.
3. Returns control to the instruction following the fork() call in the parent and child process.

Just after a fork() call, the two processes are almost identical; the program is the same, they have access to the same storage, and the user security environment is the same. Any HFS file descriptors or socket descriptors that were opened by the parent process are also open and available to the child process. (However, if a file is marked as fdclose on a fcntl() call, the file descriptor will be missing and not propagated to the child.) Positions established by the parent process in sequentially processed files before the fork() call are maintained and preserved in the child process. In fact, the only difference between the two processes is the return value

from the fork() call; the parent process receives the process ID (PID) of the child process, while the child process receives a return value of zero. It is important to understand that control is given to the child process at the instruction following the fork() call and not at the program's main entry point, as you, based on traditional MVS experience, might have expected.

One important aspect of this inheritance concept applies to DD-name allocations of any kind. If the parent process had made a DD-name allocation (using JCL, TSO ALLOC, or dynamic allocation services) before the fork() call, this allocation is *not* inherited by the child process — unless it is the STEPLIB DD. The STEPLIB DD is propagated to the child process.

In most implementations, the parent process will go on doing what it has to do, and the child process will most likely do cleanup and pass control (exec) to a child-specific program that will do whatever the child process has to do.

Spawning a new process

The spawn service is another mechanism for starting a new process. This service works very much like a fork(), except for the fact that the new process is not a copy of the parent process. On the spawn() call, the parent process specifies the name of an HFS program to start in the child process, and the new process is started with this new program being given control at its main entry point.

The default action is to propagate file and socket descriptors to the child process, as is done with a fork(). However, with spawn() the application can specify an fd_map to remap file descriptors, so that the child gets different file descriptors from the parent. If a process is spawned in the same address space, all the DD's are still available. Parent and child in the same address space can use the same DD as long as they understand the rules laid out by Allocation. For example, they which has a DD with DISP=SHR. The rules are complex.

By setting the _BPX_SHAREAS environment variable, the parent process can control if a spawn() call will result in a process being started in another address space or as a task within the same address space as the parent process itself. If _BPX_SHAREAS is set to YES before the spawn() call is initiated, the child process starts within the same address space as the parent process. There are some exceptions where, despite _BPX_SHAREAS=YES, a non-local spawn() (child process starts in another address space) is done. A non-local spawn() is done in any of these cases:
- The program spawned has sticky bit on
- The program spawned is an external link
- The program spawned is a setuid or setgid program
- The address space has exhausted its private storage

Some applications allow you to set a configuration variable that is used by the application to control whether new processes are started within the same address space or within a new address space. Where such configuration options are available, it is generally a good idea to turn them on. Starting a new process within the same address space as the parent process requires much less processing and will in general perform better than starting the new process in another address space.

If your application uses pipes or shared memory and you switch to using spawn(), we have a web page that lists differences to be aware of.

Replacing the program in a process

If a program in a process wants to replace itself with another program, it can use the exec service. This service will preserve the current process environment, but completely replace the program that is running within that process. A successful exec call (the exec family of functions) will never return control to the calling program, but control will be passed to the main entry point of the new program that is specified on the exec call.

The exec service is typically used after a fork() by the child process to replace the parent process program with a child-specific program to perform the child-related functions of the application.

UID/GID Assignment: Process Authorization

As we mentioned earlier, the first attempt to use z/OS UNIX services *dubs* the MVS address space as an z/OS UNIX process. This adds new information to the current address space, of which the UID/GID assignment probably is the most important:

Real UID
: At process creation, the real UID identifies the user who has created the process.

Effective UID
: Each process also has an *effective UID*. The effective UID is used to determine *owner* access privileges of a process.

  Normally this value is the same as the real UID. It is possible, however, for a program that resides in the hierarchical file system to have a special flag set that, when this program is executed, changes the effective UID of the process to the UID of the owner of the program. A program with this special flag set is said to be a *set-user-ID* program. This feature provides additional permissions to users while the set-user-ID program is being executed.

Real GID
: At process creation, the real GID identifies the current connect group of the user for which the process was created.

Effective GID
: Each process also has an effective group. The effective GID is used to determine *group access* privileges of a process.

  Normally this value is the same as the real GID. A program can, however, have a special flag set that, when this program is executed, changes the effective GID of the process to the GID of the owner of this program. A program with this special flag set is said to be a *set-group-ID* program. Like the set-user-ID feature, this provides additional permission to users while the set-group-ID program is being executed.

The real UID/GID tells us who we really are; the effective UID and GID are used for file access permission checks; the saved values of UID and GID are stored by the exec function.

See Table 1 (page 62) for the relations between the values described above and how they are manipulated by various function calls.

Note: Although we only reference the setuid() function, the same applies to the GID as handled by the setgid() function.

Table 1. Summary Showing How the UID May Be Changed

ID	exec set-uid-bit off	exec set-uid-bit on	setuid() superuser
real user ID	unchanged	unchanged	set to UID
effective user ID	unchanged	set from owner UID of program file	set to UID
saved set-user ID	copied from effective UID	copied from effective UID	set to UID

Process groups and job control

In addition to having a process ID, each process belongs to a process group. A process group is a collection of one or more processes. Each process group has a unique process group ID. The most important attribute of a process group is that it is possible to send a signal to every process in the group just by sending the signal to the process group leader. (When sending a kill, you must put a negative sign (-) before the PID of the process group leader in order to have the signal be propagated to the group.)

Each time the shell creates a process to run an application, the process is placed into a new process group. When the application spawns new processes, these are members of the same process group as the parent.

Some process identifiers are used for job control. The several types of process identifiers associated with a process are:

- PID: A process ID. A unique identifier assigned to a process while it runs. The PID is not returned to the system until the parent issues a wait(). Until the wait() is issued, a terminated process still has a PID and its status is ZOMBIE. Each time you run a process, it has a different PID (it takes a long time for a PID to be reused by the system). You can use the PID to track the status of a process with the ps command or the jobs command, or to end a process with the kill command.
- PGID: Each process in a process group shares a process group ID (PGID), which is the same as the PID of the first process in the process group. This ID is used for signaling related processes. If a command starts just one process, its PID and PGID are the same.
- PPID: A process that creates a new process is called a parent process; the new process is called a child process. The parent process (PPID) becomes associated with the new child process when it is created. The PPID is not used for job control.

Process priorities

Process priorities are handled as follows:

- For an MVS address space that gets dubbed, its priority has already been established based on whether it is a batch job, TSO work, started task, etc. Getting dubbed does not change the priority.
- Forked and spawned processes are places in the OMVS subsystem category. Installations control the performance attributes of the OMVS subsystem using SRM or WLM mechanisms. Child processes do not inherit their priorities from

the parent. Instead, they are treated as a member of the OMVS subsystem category, which can further be tuned by account number or userid in the appropriate SRM or WLM controls. This is discussed in *z/OS UNIX Planning*.

Threads

If a program within an z/OS UNIX System Services (z/OS UNIX) process needs to work with more dispatchable units of work, but does not need the advanced functions of fork or spawn, it can use the POSIX threading services that are part of the z/OS UNIX services.

The application program can create and manage threads via a range of special threading system service calls. To create a new thread, an application program will use the pthread_create() service. If you are writing code in assembler, it is easier to do an ATTACH and let the tasks be dubbed as threads.

There are, in general, three types of threads:

1. **Light-weight threading**

   z/OS UNIX does not implement light-weight threads. Light-weight threads are not assigned to individual subtasks, but are managed within one task. This is also sometimes referred to as pseudo-subtasking.
2. **Medium-weight threading**

   With medium-weight threading services, an application does pthread_create() and after the thread does a pthread_exit(), z/OS UNIX reuses the task. An example of an z/OS UNIX application that uses medium-weight threading is the z/OS Internet Connection Server.
3. **Heavy-weight threading**

   If the application uses heavy-weight threads, a new MVS subtask is created every time the application requests a new thread to be created. The subtask is terminated when the thread terminates.

z/OS UNIX resources, such as HFS files, sockets, pipes, and so forth, are available to all threads within a process.

POSIX threads and MVS subtasking are similar in many respects, except for the following: MVS subtasks are normally used to run a piece of code that may run on its own (a separate load module), while the pieces of code that are running on POSIX threads are part of one and the same load module.

Limitation on the number of threads

At the current time, the limiting factor for the number of C threads that you can run in a single address space is the storage below the 16M line. Each thread is an MVS TCB. Each task has a TCB, XSB (extended status block), at least 1 RB (request block), a first save area, and possibly a few other things. These MVS control blocks take up about 1K below the 16M line. Additionally, Language Environment and the C RTL take up about 12K below the line. This is considered a problem and is being worked on.

We generally recommend that the maximum number of concurrent threads in an address space be 100-200. For many applications with large numbers of threads, contention can become a problem. All the work in an address space can drive

contention on Language Environment latches, the local lock, and the RSM address space lock for paging activity. Also, debugging can be more difficult.

However, some customer applications are designed in such a way that they successfully use many more threads per process. For example, with proper system tuning, we have seen Java applications that run with well over 1000 concurrent threads in an address space. The largest number of concurrent threads we've seen in a single address space is about 2600 (OS/390 V2R6). But, in this case careful tuning was done, and we ran a simple testcase in which all the threads were sleeping.

If you want to increase the number of threads, read our suggestions for setting Language Environment runtime options and other steps to take.

Stopping Threads

An application can stop threads within their process by using pthread_kill() or pthread_cancel().

An operator can terminate an z/OS UNIX thread, without disrupting the entire process. The syntax of the MODIFY command to terminate a thread is:

```
F BPXOINIT,{TERM}=pid[.tid]
           {FORCE}
```

where *tid* indicates the thread id (TID) of the thread to be terminated. The TID is 16 hexadecimal characters as displayed by the following command:

```
D OMVS,PID=pppppppp
```

In some situations, you may want to want to terminate a single thread when the thread represents a single user in a server address space. Although random termination of a thread usually causes a process to hang or fail, using the MODIFY command to terminate a thread will not cause the process to terminate. Note that some servers, such as Lotus, do not terminate individual threads; rather, all processes are terminated if one terminates.

Porting applications with pthreads

If you are porting pthreads or mutexes, we have a web page that lists some differences you will encounter.

Interprocess communication (IPC)

Four interesting functions in z/OS UNIX System Services (z/OS UNIX) come under the heading of interprocess communication:

- "Shared memory" on page 65
- "Message queues" on page 66
- "Semaphores" on page 66
- "Memory mapping" on page 67

These forms of interprocess communication extend the possibilities provided by the simpler forms of communication between processes: pipes, named pipes or FIFOs, signals, and sockets.

The IPC mechanisms of shared memory, semaphore and message queues are all persistent. To identify abandoned IPC constructs, use the **ipcs** shell command. To remove them, the owner or superuser can use the **ipcrm** shell command.

Shared memory

Shared memory is good for sharing data, and it can also be useful for keeping track of resources shared across multiple processes. It saves you from moving the data multiple times, as is done for pipes, message queues, and sockets.

Shared memory provides an efficient way for multiple processes to share data, such as control information that all processes require access to. The processes use semaphores to take turns getting access to the shared memory. For example, a server process can use a semaphore to lock a shared memory area, then update the area with new control information, use a semaphore to unlock the shared memory area, and then notify the sharing processes. Each client process sharing the information can then use a semaphore to lock the area, read it, and then unlock it again for access by other sharing processes. In general, an application would want to have multiple readers and one writer which can only write when it knows the readers are blocked. This can be done with semaphores or with read/write locks. Read/write locks make this much simpler to accomplish.

Shared memory will persist even after all users detach from it. For example, if one process is attached to a shared memory segment and it terminates (either normally or abnormally) without detaching the segment, the segment does not go away.

If you want to use shared memory from C, you can use the C functions: shmget(), shmat(), and shmctl(). For assembler, you can use the services:
- shmget() (BPX1MGT) — Create/Find a Shared Memory Segment
- shmat() (BPX1MAT) — Attach to a Shared Memory Segment
- shmctl() (BPX1MCT) — Perform Shared Memory Control Operations
- shmdt() (BPX1MDT) — Detach a Shared Memory Segment

shmget() or the BPX1MGT function allows you to define how big an area of shared memory you want. This starts at 4K and the upper limit is controlled by the BPXPRMxx parmlib limit and the amount of storage available in the user region. When you use this function, it reserves space in a kernel data space. Shared memory is permanent, until explicitly freed or detached (just like common storage, CSA).

When you call shmat() or BPX1MAT, it does a getmain for the amount of space in the user private area and uses an RSM service IARVSERV to create page sharing groups between the getmained pages and the pages in the data space. All users (with appropriate permission) that attach to this shared memory segment are similarly connected to these same pages. Hence, after the attach, you share this memory with the other processes which also did the attach. This is regular key 8, user storage which can be used for any existing MVS service. You can read into it, write out of it, and it is shared between your multiple address spaces, but no one else can see it.

If you are using shared memory, be aware of its extended system queue area (ESQA) requirements, and how to reduce the real storage requirements. A number of z/OS UNIX System Services (z/OS UNIX) use base z/OS functions that consume ESQA storage. This storage is fixed, consuming main memory rather than

only virtual storage. Installations having constraints on virtual storage or main memory can control the amount of ESQA storage consumed.

If you specify the __IPC_MEGA option on shmget() to request segment-level sharing, it results in significant real storage savings and reduced ESQA usage, especially as number of shares increases. The resulting shared memory segment will be allocated in units of segments instead of units of pages.

Message queues

XPG4 provides a set of C functions that allow processes to communicate through one or more message queues. A process can create, read from, or write to a message queue. Each message is identified with a "type" number, a length value, and data (if the length is greater than zero).

Messages queues are good for handling small messages that are fed to a server. The intended design is that the message queue never get too deep.

A message can be read from a queue based on its type rather than on its order of arrival. Multiple processes can share the same queue. For example, a server process can handle messages from a number of client processes and associate a particular message type with a particular client process. Or the message type can be used to assign a priority in which a message should be dequeued and handled.

If you build up deep queues and use multiple message types for categorizing, this can affect performance.

Message queues are persistent for the duration of the current IPL. You can write a message to a queue and another job or address space can react to it right away or next week. Messages waiting in the queues are kept in kernel data spaces until they are received.

Messages can be very small (1 byte) or quite large (megabytes). For larger messages, consider using shared memory instead. One process can put something in shared memory and a message on a queue can point to it. This avoids moving the data.

The C functions for using message queues are msgget(), msgrcv(), and msgsnd(). The callable services are:

- msgget (BPX1QGT) — Create or Find a Message Queue
- msgrcv (BPX1QRC) — Receive from a Message Queue
- msgsnd (BPX1QSN) — Send to a Message Queue

Semaphores

Semaphores, unlike message queues and pipes, are not used for exchanging data, but as a means of synchronizing operations among processes. Semaphores provide the ability to perform serialization on resources. A semaphore value is stored in the kernel and then set, read, and reset by sharing processes according to some defined scheme.

Typical uses for semaphores are serialization of shared memory, resource counting, and file locking. Frequently, semaphores are used to serialize hunks of shared memory.

The semget() function creates a semaphore set or locates an existing one. Semaphores are convenient for C programmers, but for assembler programming, ENQ is simpler to use and has better recovery and serviceability characteristics.

To serialize between a C program and an assembler program, you can either write an assembler stub for C to do the ENQ or you can have the assembler program code a syscall interface to use semaphores.

You can optimize the behavior of binary semaphores, whereas trying to optimize the behavior of counting semaphores is probably impossible.

- A counting semaphone can range in value from 0 to a large number. Counting semphores are serialized with a GRS latch, which can inject additional contention into the application.
- A binary semaphore has a value of either 1 or 0, where 0 usually means the semaphore is held and 1 means that the semaphore is available. You can only have one owner of a binary semaphore.

To improve the performance of binary semaphores, use the __IPC_BINSEM option for semget(); this tells the kernel that the application will be using the semaphores only in a binary semaphore fashion. semop() then uses the PLO instruction to atomically update the semaphores in the set, and the GRS latch is eliminated from the semop( ) processing. This implementation is ideally suited to environments where there is heavy semaphore usage and contention. This option was introduced in OS/390 V2R6 and rolled back to OS/390 V2R4 via APAR. See *z/OS C/C++ Programming Guide* for further information on semaphore performance.

Memory mapping

In z/OS UNIX, a programmer can arrange to transparently map an HFS file into process storage. The use of memory mapping can reduce the number of disk accesses required when randomly accessing a file.

The mmap(), mprotect(), msynch(), munmap(), __map_init() and __map_service() functions provide memory mapping. The callable services are:

- mmap (BPX1MMP) — Map Pages of Memory
- mprotect (BPX1MPR) — Set Protection of Memory Mapping
- msynch (BPX1MSY) — Synchronize Memory with Physical Storage
- munmap (BPX1MUN) — Unmap Previously Mapped Addresses
- __map_init (BPX1MMI) — Designate a Storage Area for Mapping
- __map_service (BPX1MMS) — Set Memory Mapping Service

Signals

The basis for error handling in z/OS UNIX C/C++ application programs is the generation, delivery, and handling of signals. Signals can be generated and delivered as a result of system events or application programming. You can code your application program to generate and send signals, and to handle and respond to signals delivered to it. These types of signal handling are supported for catching signals: ANSI C, POSIX.1, BSD, and additional functions provided by XPG4.

Examples of ways signal functions can be used are:

- **Maintenance:** Most UNIX systems send a signal to the process in the event of invalid pointers or other indications of a bug in the program. Depending on

how the signal handling is set up, this can cause a core dump to be generated and used for debugging purposes by developers.

- **Communication Events:** When two programs are communicating with each other over a file descriptor (it could be a networking (IP) program, a pipe, or something else), if the recipient side of a conversation terminates (normally or abnormally), the sending party receives a SIGPIPE event. Other network related signals are SIGIO and SIGPOLL that indicate an asynchronous I/O event.
- **Timer Functionality:** Either the alarm() or the setitimer() functions cause the signal SIGALRM to be generated.
- **User/tty Interrupts:** Usually the interactive user can cause a signal to be generated by using certain key sequences. For example, Ctrl-C generates a SIGINT and Ctrl-Z causes the SIGTSTP signal to be sent to the process.
- **Interprocess Communication:** We do not recommend using signals for communication. The biggest disadvantange is that multiple signals of the same type can be lost. For general purpose communication, use shared memory, semaphores, messages queues, sockets, and pipes.
- **Process Tracking:** A parent process can have a signal catcher for SIGCHLD, so that it is notified when a child process terminates.

Each process has an action to be taken in response to each signal defined by the system. During the time between the generation of a signal and the delivery of a signal (when the actual action is performed), the signal is said to be pending. It is valid for the process to block it. If a signal that is blocked is generated for a process and the action for that signal is either the default action or to catch the signal, the signal remains pending for the process until the process either unblocks the signal or changes the action to ignore the signal.

A signal can be specified to be blocked in the sigprocmask() and sigsuspend() functions. Each thread has a signal mask that defines the set of signals currently blocked from delivery, and the mask is inherited by a child from its parent. The signal mask is inherited across fork(), exec(), spawn() and pthread_create().

Supported Signals - POSIX(OFF)

For z/OS C/C++ with POSIX(OFF), these signals are supported:

- **SIGABND:** System abend.
- **SIGABRT:** Abnormal termination (software only).
- **SIGFPE:** Erroneous arithmetic operation (hardware and software).
- **SIGILL:** Illegal or invalid instruction.
- **SIGINT:** Interactive attention interrupt by raise() (software only).
- **SIGIOERR:** Serious software error such as a system read or write. You can assign a signal handler to determine the file in which the error occurs or whether the condition is an abort or abend. This minimizes the time required to locate the source of a serious error.
- **SIGSEGV:** Invalid access to memory (hardware and software).
- **SIGTERM:** Termination request sent to program (software only).
- **SIGUSR1:** Reserved for user (software only).
- **SIGUSR2:** Reserved for user (software only).

Supported Signals - POSIX(ON)

For z/OS C/C++ with POSIX(ON), these signals are supported:

- **SIGABND:** System abend.
- **SIGABRT:** Abnormal termination (software only).
- **SIGALRM:** Asynchronous timeout signal generated as a result of an alarm().
- **SIGBUS:** Bus error.
- **SIGCHLD:** Child process terminated or stopped.
- **SIGCONT:** Continue execution, if stopped.
- **SIGDCE:** DCE event.
- **SIGFPE:** Erroneous arithmetic operation (hardware and software).
- **SIGHUP:** Hangup, when a controlling terminal is suspended or the controlling process ended.
- **SIGILL:** Illegal or invalid instruction.
- **SIGINT:** Asynchronous Ctrl-C from the shell or a software generated signal.
- **SIGIO:** Completion of input or output.
- **SIGIOERR:** Serious software error such as a system read or write. Assign a signal handler to determine the file in which the error occurs or whether the condition is an abort or abend. Minimize the time required to locate the source of a system error.
- **SIGKILL:** An unconditional terminating signal.
- **SIGPIPE:** Write on a pipe with no one to read it.
- **SIGPOLL:** Pollable event.
- **SIGPROF:** Profiling timer expired.
- **SIGQUIT:** Terminal quit signal.
- **SIGSEGV:** Invalid access to memory (hardware and software).
- **SIGSTOP:** Stop executing.
- **SIGSYS:** Bad system call.
- **SIGTERM:** Termination request sent to program (software only).
- **SIGTRAP:** Debugger event.
- **SIGTSTP:** Terminal stop signal.
- **SIGTTIN:** Background process attempting read.
- **SIGTTOU:** Background process attempting write.
- **SIGURG:** High bandwidth is available at a socket.
- **SIGUSR1:** Reserved for user (software only).
- **SIGUSR2:** Reserved for user (software only).
- **SIGVTALRM:** Virtual timer expired.
- **SIGXCPU:** CPU time limit exceeded.
- **SIGXFSZ:** File size limit exceeded.

Chapter 11. Networking

Open systems support considerable distributed network capabilities. In the z/OS UNIX System Services (z/OS UNIX) environment, both client and server socket applications can use the Berkeley socket interface (formerly called the OpenEdition socket interface) to communicate over the network (using AF_INET sockets) or communicate with other local z/OS UNIX socket applications (using AF_UNIX sockets).

There are many products that support AF_INET sockets; two examples are SecureWay Communications Server (IP), formerly known as eNetwork Communications Server(TCP/IP) and AnyNet. The Common INET prerouting function will allow multiple AF_INET providers to be concurrently active. These may be either different products or multiple releases or versions of the same product.

These are the topics we will discuss:

- "TCP/IP"
- "AnyNet"
- "Sockets in the z/OS UNIX Environment" on page 72

TCP/IP

For many years now, the TCP/IP communications stack has been included in most UNIX operating systems. Therefore, most UNIX client/server applications use it. TCP/IP is the transport provider when users rlogin or telnet from a workstation directly into the shell.

TCP/IP (Transmission Control Protocol/Internet Protocol) is a term that is generally used to refer to a specific set of protocols that allow computers to share resources in a network. Prior to OS/390 Release 4, the TCP/IP Version 3 for OpenEdition MVS Sockets Applications Feature was available. With OS/390 Release 4, TCP/IP for OS/390 UNIX (formerly called OpenEdition) became a base element of z/OS, as part of the eNetwork Communications Server.

AnyNet

AnyNet software, based on an X/Open standard, delivers multiprotocol combinations on the z/OS platform. With AnyNet, your SNA, TCP/IP, IPX, and NetBIOS applications and networks can be integrated.

AnyNet is a family of software products consisting of multiprotocol access nodes and multiprotocol gateway nodes that are based on the multiprotocol transport networking architecture.

Traditionally, networking APIs are tied to one particular network protocol family. For example, if you develop a program that uses the sockets API, such a program is traditionally tied to the TCP/IP protocol stack. If you develop a program that uses the CPI-C API, such a program is traditionally tied to the SNA protocol stack. Multiprotocol transport networking removes the tie between a particular API and a

© Copyright IBM Corp. 1998, 2001

particular network protocol family, allowing your socket programs to use an SNA network (sockets over SNA) and your CPI-C programs to use a TCP/IP network.

In an z/OS environment, AnyNet,VTAM, and TCP/IP now comprise the SecureWay Communications Server(IP). AnyNet functions are available via MPTF (Multiple Protocol Transport Facility) or via the AnyNet MVS feature of VTAM/ESA. The first VTAM/ESA version to support AnyNet MVS was VTAM/ESA V3.4.2.

- In VTAM for MVS/ESA V4.2, APAR OW10895 / PTF UW17057 supplied support for integrated sockets and thus enabled AnyNet MVS as an AF_INET transport provider in an z/OS UNIX environment.
- The AnyNet MVS feature of VTAM V4.3 for MVS/ESA includes support for Common INET.

Sockets in the z/OS UNIX Environment

Any program that uses a socket application programming interface must be compiled and linked with a socket library. A socket library consists of one or more of the following components:

- Header files, include files, or copy structures that are used during compilation of the socket application program. These files define commonly used data structures and interfaces to socket-related functions.
- Link library with object modules to be statically linked with the socket application program during linkage editing processing.
- Run-time library with run-time programs to support the socket calls that are being used by the application program.

These are the sockets topics we will discuss:

- “Sockets in z/OS”
- “Writing a socket application” on page 73
- “Integrated sockets PFS” on page 75
- “Common INET PFS” on page 76
- “C/C++ resolver configuration data” on page 78

Sockets in z/OS

In an z/OS environment you have a number of socket libraries that are available for your use. Most socket libraries are language-specific. The most common socket programming language is C, but other programming languages are supported, too.

If you write your socket program in C for an z/OS environment, using the z/OS C/C++ socket library is recommended. Because our intention has been to converge on a single C sockets interface, all new function has been added to the z/OS C/C++ socket library. For that reason, you are encouraged to use the z/OS C/C++ socket library instead of the other interfaces: The TCP/IP C socket library or the AnyNet z/OS C socket library. To provide for a transition, there will be multiple socket libraries for a while.

The z/OS C/C++ socket library offers two choices:

- **X/Open Sockets**

  X/Open sockets meet the UNIX95 brand interface. These sockets are recommended for new applications.
- **Berkeley sockets (formerly OpenEdition sockets)**

Berkeley sockets meet only the BSD interface. They provide a porting path for existing applications that use BSD sockets.

To accesss the **socket header files**, there are various options you can use when you invoke the c89/cc/c++ utility to compile. See the section on c89 access to socket header files in the ″Compiling″ chapter of this **Porting Guide**.

Writing a socket application

When you write socket application programs to be used with the C socket libraries from TCP/IP and AnyNet, you have to use a separate set of C functions to access files and another set of C functions to access sockets. To read data from a socket, you can use the read() call, and to read data from a file, you can use the fread() call.

A C program works with handles that represent resources, such as files or sockets. Such a handle is called a *descriptor*, and is a 16-bit integer that holds a descriptor number, which represents the resource in question. In a C program that uses the TCP/IP or AnyNet socket libraries, one part of the run-time environment manages the descriptors that represent, for example, files (file descriptors), and another part, the socket library, manages the descriptors that represent sockets (socket descriptors). In such a program, there is no common management of descriptor numbers, which is why the programmer has to use different functions for accessing a file or a socket. The C run-time environment is not able to determine if a descriptor is a file descriptor or a socket descriptor, so the programmer must indicate that via the functions that are used to access the resources.

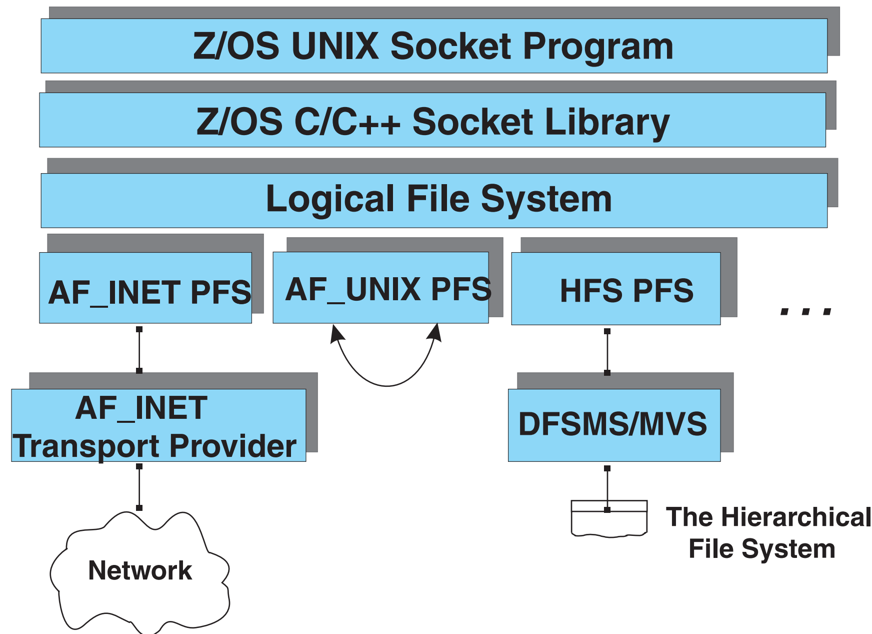
In a POSIX-compliant program, the programmer does not have to make such a distinction. In such a C program, the programmer uses the same functions to access files, pipes, sockets and other resources that are used by the C program. The C run-time environment must therefore be able to determine dynamically what kind of a descriptor is passed on the individual function calls, it must have information available for all the descriptors that are in use and manage assignment of all new descriptors across all the supported resource types.

In z/OS UNIX, this is accomplished through the use of the following components, as shown in Figure 1:

1. An z/OS C/C++ socket library
2. A component that is called the logical file system (LFS)
3. A set of components that are called physical file systems (PFS)

All C calls that use descriptors are passed to the logical file system. The logical file system manages assignment of new descriptors and maintains information about the type of resource that is represented by the individual descriptors.

Figure 1. Sockets in z/OS UNIX



Based on the type of resource, individual function calls are passed to the associated physical file system for execution. There are a number of physical file systems in z/OS UNIX, including:

- The hierarchical file system PFS.

  This PFS takes care of requests that are related to resources in the hierarchical file system, such as traditional files or special character files.
- The AF_UNIX PFS.

  If the descriptor represents an AF_UNIX socket, the request is handled by this PFS. AF_UNIX sockets are so-called local sockets that can be used by two z/OS UNIX application programs on the same system to communicate with each other.
- The AF_INET PFS.

  If the descriptor represents an AF_INET socket, the request is handled by this PFS. An AF_INET socket is what is used on a TCP/IP-based network and is generally known as a network socket. If an z/OS UNIX application program wants to communicate with a socket program on a TCP/IP host on an attached IP network, the program will open an AF_INET socket for that purpose.

From a TCP/IP point of view, it is the AF_INET physical file system that is of most interest:

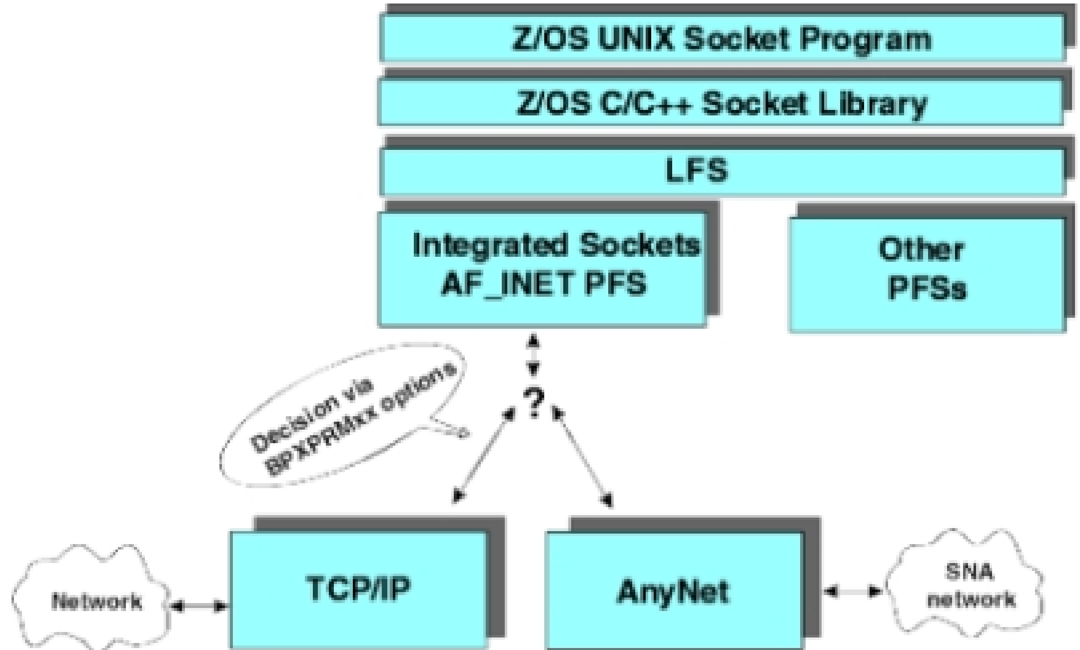
- OS/390/ESA SP 5.1 introduced the integrated sockets AF_INET PFS.
- In MVS/ESA SP 5.2.2 the Common INET PFS was introduced; sometimes, for short, called CINET.

In an MVS/ESA SP 5.2.2 or z/OS system, you can choose which of the two AF_INET physical file systems you want to use: integrated AF_INET or Common INET.

Integrated sockets PFS

The integrated sockets PFS supports one AF_INET transport provider. This transport provider can either be a TCP/IP stack or it can be an AnyNet stack, but only one stack at a time is supported. All AF_INET socket calls are handled by this one AF_INET transport provider, as shown in Figure 2.

Figure 2. Integrated sockets



You specify in the BPXPRMxx parmlib member if you want TCP/IP or AnyNet as the AF_INET transport provider. If you start more TCP/IP stacks on your z/OS system, you cannot use the BPXPRMxx parmlib member to specify which of the TCP/IP stacks you want as AF_INET transport provider. The first TCP/IP stack that connects to the kernel address space becomes the AF_INET transport provider. You can control, via parameters in your TCP/IP PROFILE configuration data set, if a TCP/IP stack should try to connect to the kernel address space as a transport provider. By default, any TCP/IP stack will try to connect to z/OS UNIX. You can prevent a stack from doing so by entering the NOOE configuration parameter in that stack's PROFILE configuration data set:

```
**************************************************************
; * Do NOT attempt to connect to OE from this TCP/IP stack     *
; **************************************************************
;
NOOE
;
```

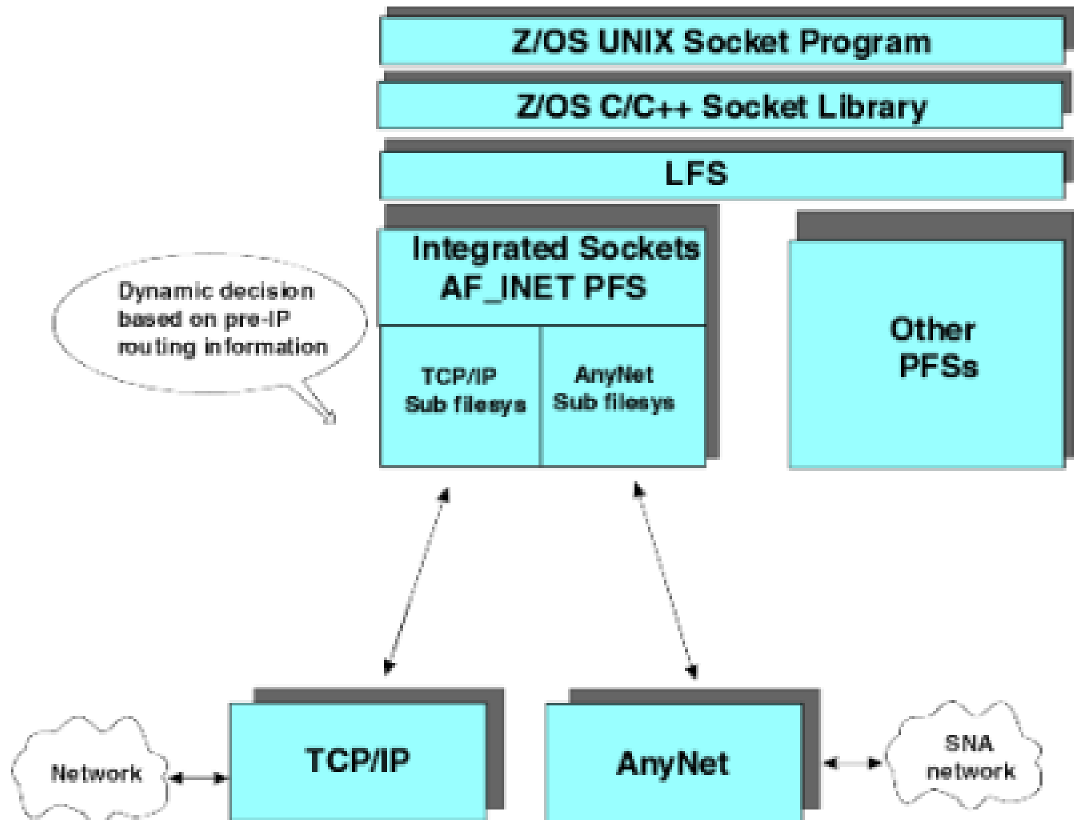
You have a similar configuration option in the AnyNet environment parameter data set. The default for an AnyNet stack is to try to connect to z/OS UNIX, but you can prevent an AnyNet z/OS stack from doing so by coding the following in the AnyNet environment parameter data set:

```
# **************************************************************
# * Do NOT attempt to connect to OE from this AnyNet stack     *
# **************************************************************
#
OPEN_EDITION=NO
#
```

Common INET PFS

In MVS/ESA SP 5.2.2 and in z/OS, the Common INET AF_INET physical file system was introduced; this is referred to as Common INET. This PFS allows more AF_INET transport providers to be connected to the kernel address space at the same time, supporting concurrent access from more AF_INET stacks, for example, a TCP/IP stack and an AnyNet stack, as shown in Figure 3.

Figure 3. Common sockets



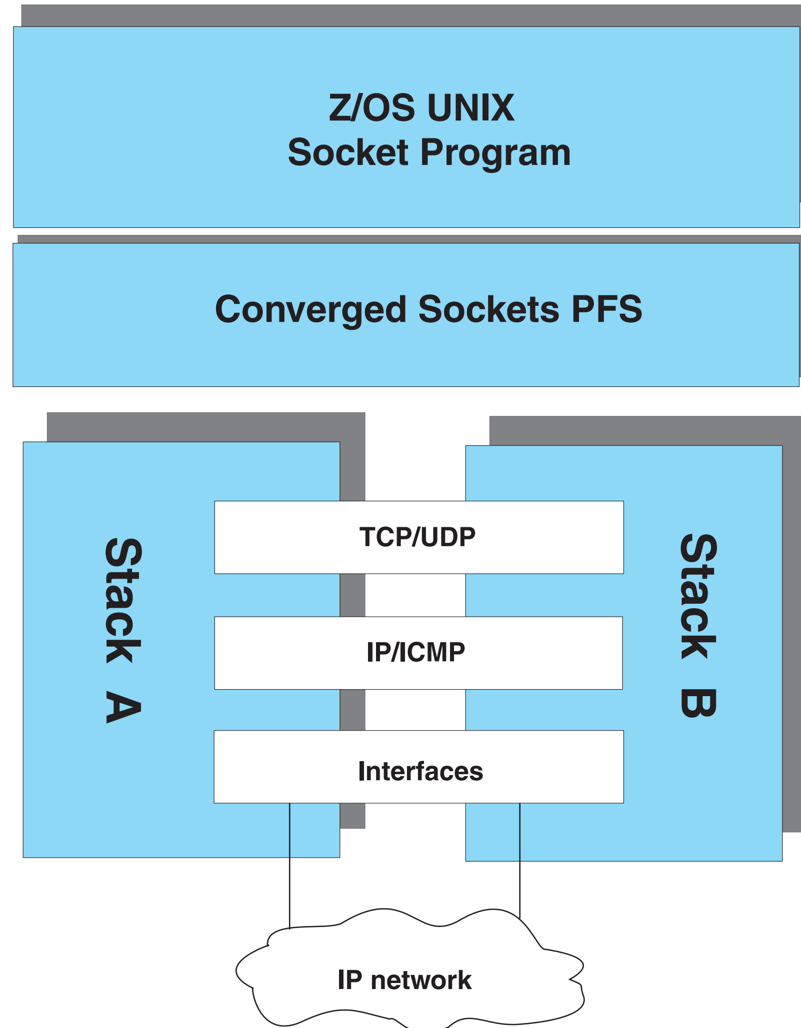
One instance of an AF_INET socket program that runs in the z/OS UNIX environment can concurrently service client requests that arrive over a TCP/IP stack and requests that arrive over an AnyNet stack.

The Common INET PFS includes a so-called prerouter, which determines how individual socket calls are passed to the connected transport providers. Some calls are routed directly to a single transport provider while other calls are propagated across all the connected transport providers. In addition to the prerouter, the Common INET PFS consists of one subfilesystem per connected AF_INET transport provider.

An AF_INET socket program does not have any knowledge of the existence of one or more transport providers. From a Berkeley socket program point of view, there is one AF_INET stack with a number of network interfaces. In reality, the individual network interfaces may be associated with different stacks, but the Berkeley socket program does not know that. The Common INET PFS gives the appearance of a single converged TCP/IP stack.

Each transport provider has a separate set of interfaces, separate networking layer with IP and ICMP, and separate transport protocol layer with TCP and UDP. But the application layer is considered to be shared among all connected stacks, as shown in Figure 4.

Figure 4. Converged Stacks As Perceived by z/OS UNIX Applications



When a remote socket client connects to an z/OS UNIX server program, that connection request arrives over one of the AF_INET transport providers, and all socket calls from the z/OS UNIX server program for that connection are routed directly to that one AF_INET transport provider. But there are socket calls that cannot be routed to a single transport provider. One example of this is a stream socket server program (TCP protocols) that starts up in the z/OS UNIX environment. Such a server program issues a number of initial socket calls that establish the program as a server program that will accept requests from clients on the connected internets. The sequence of calls is:

1. socket() - Open a socket. This call is propagated to all connected AF_INET transport providers in order to open a socket in each transport provider's transport protocol layer.
2. bind() - Bind the socket to a local server port number. This call is propagated to all connected AF_INET transport providers in order to establish the server program's identity on all stacks.

3. listen() - Prepare to receive client connection requests. This call is propagated to all stacks in order to signal the server program's intent to receive client requests over all connected transport providers.
4. accept() - Wait for the next client to connect. As the server accepts client requests over all connected transport providers, this call is also propagated to all transport providers.

When a client actually connects through one of the transport providers, the succeeding socket calls for that one connection are routed only to that one stack over which the connection was established.

Consider another type of socket program: a client program that starts in the z/OS UNIX environment. If this program is a stream socket client program, it issues a connect() call where the client program specifies the internet address of the server host and the port number on which the server program is running on that destination host. When this request comes down to the Common INET PFS, the prerouter must determine which of the connected transport providers has the best route to the requested destination IP address. This decision is made based on a copy of each transport provider's IP layer routing table. When a stack connects to the Common INET PFS, the prerouter queries the newly connected stack for a copy of its IP layer routing table. Each time the transport provider updates its IP layer routing table, the prerouter receives a signal from the transport provider and initiates a process to obtain a new copy in order to keep an up-to-date accurate copy of all the destinations that are supported by the connected transport providers.

A stack may update its IP layer routing table in more ways. It may do so as the result of a manual update of the routing tables, for example, via a TCP/IP OBEYFILE command that replaces the static route definitions for a TCP/IP stack, or for an AnyNet stack via an execution of the ISTSKRTE utility program to manually update the AnyNet IP layer routing table. IP layer routing tables in TCP/IP may also be updated via ICMP redirects or via a dynamic update from the RouteD server program based on new route information received from other RouteD or GateD servers on the connected IP networks.

It is important to emphasize that the Common INET prerouter does not make any IP level routing decisions. The prerouter only uses its copy of the IP layer routing tables to *select* an appropriate stack for certain socket calls. When the transport protocol layer in the selected stack has constructed an IP datagram and passed this IP datagram to that stack's IP layer, normal IP layer routing decisions are being made by that IP layer based on its IP layer routing table. Note that this means there is no IP layer routing taking place via the Common INET PFS between stacks. If we have two stacks connected to the Common INET PFS, these two stacks cannot route an IP datagram between them via the Common INET PFS. If the two stacks need to route IP datagrams between them, they will either have to use an IUCV link between them (if both stacks are TCP/IP stacks) or they will have to be connected to the same physical IP network.

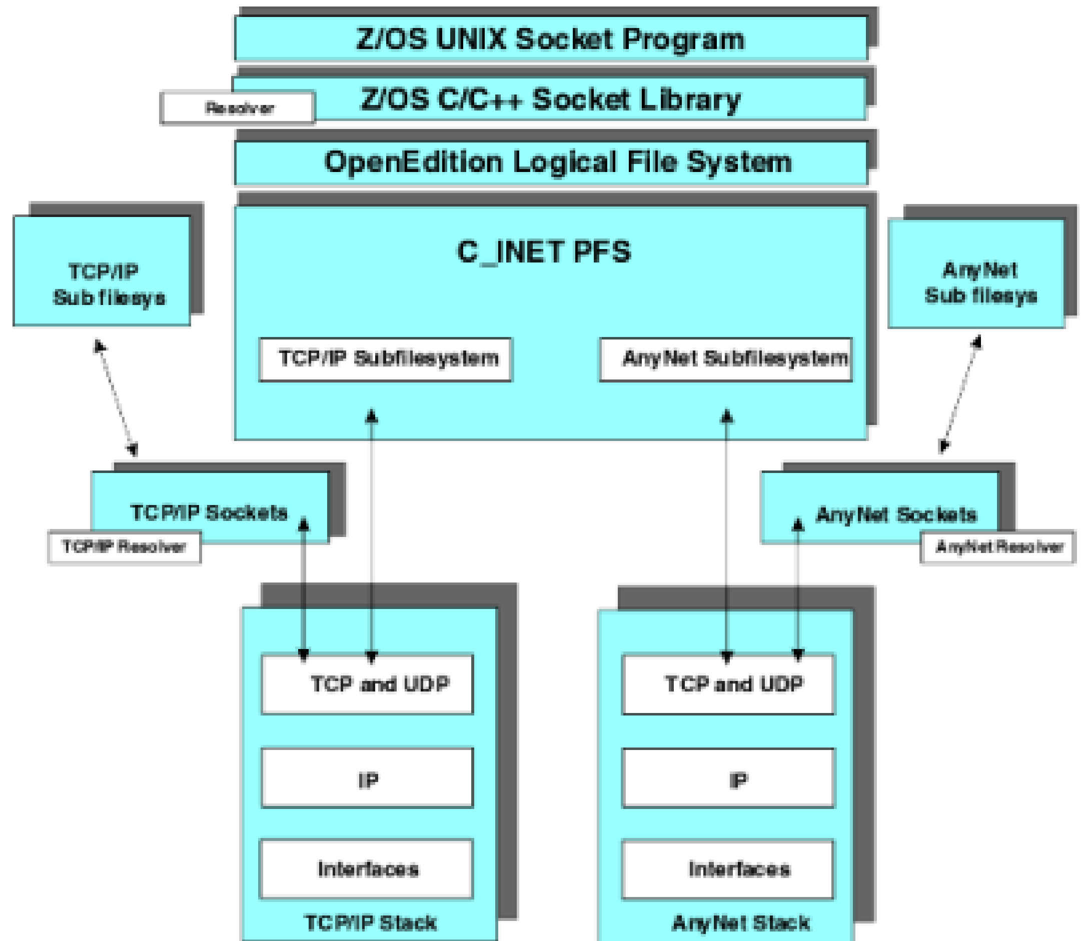
C/C++ resolver configuration data

One of the components you will find in a socket library is the *resolver*. The resolver in the z/OS C/C++ socket library is delivered as part of the run-time library and is used by all z/OS UNIX socket programs. The function of the resolver is to service a number of application calls to obtain, for example, the local port number for a given server program or to resolve an IP host name into one or more IP

addresses. Some of the calls to the resolver are serviced via looking up information in a number of local configuration data sets or files, while other calls are serviced by sending requests to remote server programs, generally known as domain name servers.

All socket libraries have a resolver component. This also means that if you have a TCP/IP stack and an AnyNet stack connected to your z/OS UNIX environment, you have three different resolver components, each requiring a set of resolver configuration data sets or files, as shown in Figure 5.

Figure 5. Resolver Components and Related Configuration Information



The main configuration data set for any resolver function is the resolver configuration data set or file:

- In a UNIX system, this file is normally located in the /etc/resolv.conf file.
- In a TCP/IP system, the resolver configuration data set is called TCPIP.DATA and can be located various places of which the most commonly used is SYS1.TCPPARMS(TCPDATA).
- In an AnyNet z/OS environment, the resolver configuration data set is pointed to via the AnyNet z/OS environment data set. The keyword is RESOLV, and it points to a fully qualified z/OS data set. In general, AnyNet is able to work with the same resolver configuration data set formats as TCP/IP is using.

The C/C++ resolver needs access to the following information:

- Resolver configuration

- Protocols supported
- Services supported
- Locally known host names
- ASCII-EBCDIC translation table for the resolver

Resolver configuration data

The contents of this configuration data set or file are compatible with the TCP/IP TCPIP.DATA configuration data set. The only parameter in an existing TCPIP.DATA configuration data set that has no meaning in z/OS UNIX environment is the TCPIPJOBNAME or TCPIPUSERID keyword. A z/OS UNIX socket program does not contact the TCP/IP system address space directly, but leaves that communication to take place in the physical file system component. The AF_INET PFS uses other techniques to decide which TCP/IP system address space to use for a given socket call.

The search order for other configuration data sets may include a step where a search is made for a z/OS data set with a specific high-level qualifier: *datasetprefix*. The value of datasetprefix comes from the DATASETPREFIX keyword in the resolver configuration data set or file. If no DATASETPREFIX keyword is found in the resolver configuration data set or file, a default of TCPIP is used by the resolver.

The resolver uses the following search order to locate the actual resolver configuration data set or file to use:

1. The z/OS data set or HFS file pointed to by the RESOLVER_CONFIG environment variable

   If the environment variable RESOLVER_CONFIG has been defined, the resolver uses the value of this environment variable as the name of a z/OS data set or HFS file to access the resolver configuration data. The syntax for a z/OS data set name is:

   `//’mvs.dataset.name’`

   . The syntax for an HFS file name is:

   `/dir/subdir/file.name`

   .
2. /etc/resolv.conf file

   This file is the preferred place in an z/OS UNIX system to place the resolver configuration data.
3. Any z/OS data set preallocated to a DDname of SYSTCPD

   We discourage anyone from using this technique in an z/OS UNIX environment because of the restrictions for DDname allocations during fork() processing.
4. userID.TCPIP.DATA or jobname.TCPIP.DATA
5. SYS1.TCPPARMS(TCPDATA)
6. TCPIP.TCPIP.DATA

   If during TCP/IP installation, you ran the EZAPPRFX installation job to zap a default high-level qualifier into numerous TCP/IP modules, this zap does *not* apply to the C/C++ resolver. This resolver always uses a high-level qualifier of TCPIP in this last search step for a TCPIP.DATA data set.

Protocol configuration data

The resolver uses the following search order for a protocol configuration data set or file:

1. /etc/protocol
2. userID.ETC.PROTO or jobname.ETC.PROTO
3. *datasetprefix*.ETC.PROTO

   datasetprefix is the value of the DATASETPREFIX keyword in the resolver configuration data set or file.

Service configuration data

The services data set or file contains the relationship between service names (servers) and port numbers in the z/OS UNIX environment. Many server programs query this configuration data set or file via a getservbyname() call to the resolver function. The resolver accesses this data set or file to find the requested service name and returns the port number to use. When you configure /etc/inetd.conf you specify service names, such as telnet and exec. INETD uses the getservbyname() call to find out which port numbers are assigned to these two services before it begins processing requests.

The following search order is used to find the services data set or file:

1. /etc/services
2. userID.ETC.SERVICES or jobname.ETC.SERVICES
3. *datasetprefix*.ETC.SERVICES

Hosts

If the z/OS UNIX system does not use a domain name server to resolve host names into IP addresses, the resolver needs access to local hosts tables it can use to resolve host names into IP addresses and IP addresses into host names.

In a UNIX system this is normally accomplished via a flat text file called hosts in the /etc directory: /etc/hosts. The syntax used in this file is generally referred to as BSD-formatted.

In a TCP/IP system, this is normally accomplished via two data sets, HOSTS.SITEINFO and HOSTS.ADDRINFO, which are built from a flat text data set called HOSTS with a utility program called MAKESITE.

The resolver may use both techniques. The search order for the local host tables is:

1. The z/OS data sets pointed to by the X_SITE and X_ADDR environment variables

   If the environment variables X_SITE and X_ADDR have been defined, their values will be used as reference to two fully qualified z/OS data sets containing the output from the TCP/IP MAKESITE utility program, HOSTS.SITEINFO and HOSTS.ADDRINFO. If these two environment variables have been defined, they must point to z/OS data sets that are output from the MAKESITE utility. If the environment variables are defined, but the resolver cannot open the data sets, the resolver assumes that you want to use the TCP/IP HOSTS.ADDRINFO and HOSTS.SITEINFO approach and skips step 2 in the search order, going directly from 1 one to step 3.

2. /etc/hosts
3. userID.HOSTS.xxxxINFO or jobname.HOSTS.xxxxINFO
4. *datasetprefix*.HOSTS.xxxxINFO

ASCII-EBCDIC translation table

The resolver has to translate, for example, an EBCDIC host name into an ASCII host name before it sends a request to a name server. When it receives the response from the name server, it has to translate the response from ASCII to EBCDIC before handing over the result to the z/OS UNIX application program. To perform these translations, the C/C++ resolver uses the TCP/IP translation table format, and it searches for a translation table data set using the following search order:

1. The z/OS data set or HFS file pointed to by the X_XLATE environment variable

   If the environment variable X_XLATE has been defined, it must refer to either a fully qualified z/OS data set name or an HFS file name that has been built with the TCP/IP CONVXLAT utility program.
2. *datasetprefix*.STANDARD.TCPXLBIN
3. An internal default ASCII-EBCDIC translate table

   If the C/C++ resolver does not find a translate table file or data set, it uses an internal default ASCII-EBCDIC translate table.

gethostid and gethostname calls

In general, the resolver code handles all the get-type calls. There are two calls that require a small comment in this context: gethostid() and gethostname().

The gethostid resolver call

The gethostid() call returns the default HOME IP address of this TCP/IP host. This information does not exist in any of the resolver configuration data sets, so the resolver passes this call down to the AF_INET transport provider to obtain the default HOME IP address of the stack. If you use the integrated sockets PFS there is only one stack to pass the request to. If you use the Common INET PFS, this call is passed to the stack that is listed with the DEFAULT keyword in your BPXPRMxx parmlib member (or the first subfilesystem listed, if you did not specify any DEFAULT keyword).

The gethostname resolver call

The gethostname() call returns the host name of this TCP/IP host. Currently this call is passed down to either the one stack that is used with integrated sockets or the default stack if you use Common INET. If the default stack is a TCP/IP stack, the host name that is returned is the host name from that stack's TCPIP.DATA configuration data set and *not* the host name of your z/OS UNIX resolver configuration data set or file. As this may change in the future, we recommend that you use the same host name in your resolver configuration data set or file as you use in your default stack's TCPIP.DATA configuration data set. If your default stack is AnyNet, the host name that is returned is the host name from the AnyNet environment data set listed with the keyword HOSTNAME.

Where to place the resolver configuration data

In general we recommend that you do not share resolver configuration data among the various resolver components, but that you create a separate set of configuration data sets or files per resolver you plan to use.

It is, however, possible to share configuration data sets between, for example, the TCP/IP resolver and the resolver, but you have to be aware of the following limitations for non-z/OS-UNIX applications' access to HFS files:

- To access a file in the hierarchical file system, an address space must execute with a user ID that has an z/OS UNIX UID and GID.
- The application program must access the HFS file using z/OS UNIX services.
- With z/OS UNIX and the Network File System client, an unchanged legacy z/OS program can use BSAM, QSAM or VSAM to create, read and write HFS files and pipes. The z/OS NFS client can access any server on a TCP/IP network that supports the SUN NFS Version 3 or Version 2 protocols.

None of the TCP/IP programs that you would normally use, such as NETSTAT or PING, are currently able to access, for example, TCPIP.DATA in the z/OS UNIX hierarchical file system. This it is not enough to do an explicit allocation to an HFS file via the PATH JCL keyword; the application still needs to use z/OS UNIX services to read the HFS file. If you want to be able to use NETSTAT or any of the TCP/IP client applications from your TSO session for your z/OS UNIX TCP/IP stack, you have to create a TCPIP.DATA z/OS data set.

Environment variables and the C/C++ resolver

Some of the C/C++ resolver configuration options can be customized by environment variable settings. If you decide to use environment variables for the resolver configuration files or data sets, the environment variables are:

1. RESOLVER_CONFIG - the resolver configuration data set or file.
2. X_SITE and X_ADDR - the HOSTS.SITEINFO and HOSTS.ADDRINFO data sets or files.
3. X_XLATE - the ASCII-EBCDIC translate table data set or file built by the TCP/IP CONVXLAT utility.

If the above environment variables have been defined, they take precedence over any other alternatives for locating the resolver configuration data, which suggests the following scheme:

1. Define the systemwide resolver configuration data sets or files so the resolver, by default, will locate them without using environment variables, for example:
   - /etc/resolv.conf
   - /etc/protocol
   - /etc/services
   - /etc/hosts
   - *datasetprefix*.STANDARD.TCPXLBIN
2. If a program or a user needs to override the systemwide resolver configuration data, the environment variables can be used on an individual basis—for example, by passing the environment variables to the program in the EXEC PARM field or by setting them in the user's $HOME.profile.

Chapter 12. Server models

During installation and customization of various server programs in the z/OS UNIX environment, you have to make decisions about a number of issues that are related to the way these server programs have been developed and are supposed to run.

We will discuss these topics:
- Iterative server programs
- Concurrent server programs
- The listener program
- The InetD generic listener program
- Starting listener programs
- Security for server programs

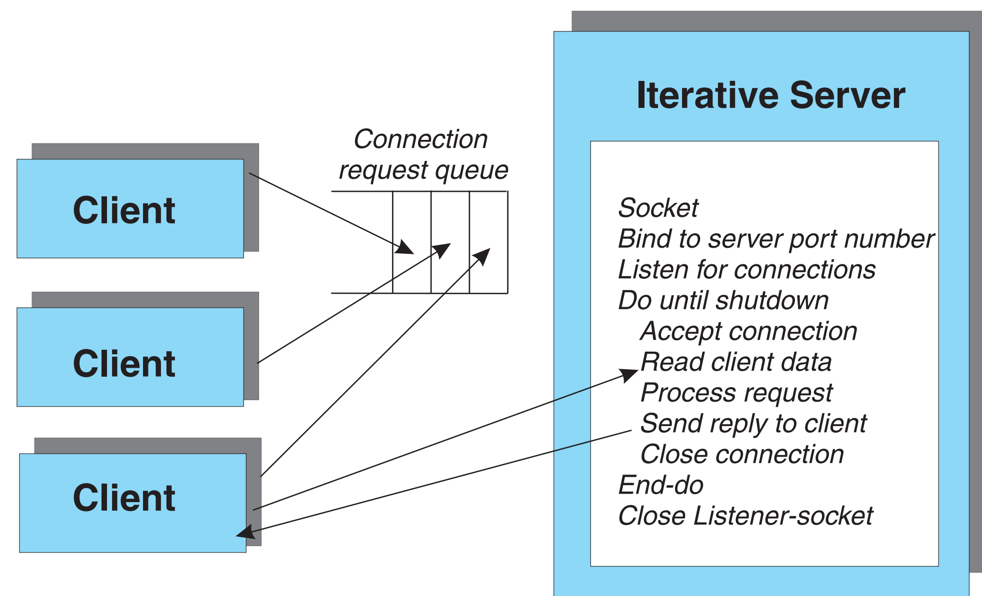
Iterative server programs

A socket server program can be developed so it will process requests from clients one at a time in a serial fashion. The server program will finish one client request before it is able to receive the next client request. Such a server is called an *iterative server*, and works as shown in Figure 1.

If the number of client requests is small and the processing needed to complete one client request is of a limited duration, an iterative server is simple to develop and works well.

Figure 1. Iterative Server Structure

An ITERATIVE server processes client requests serially, one at a time



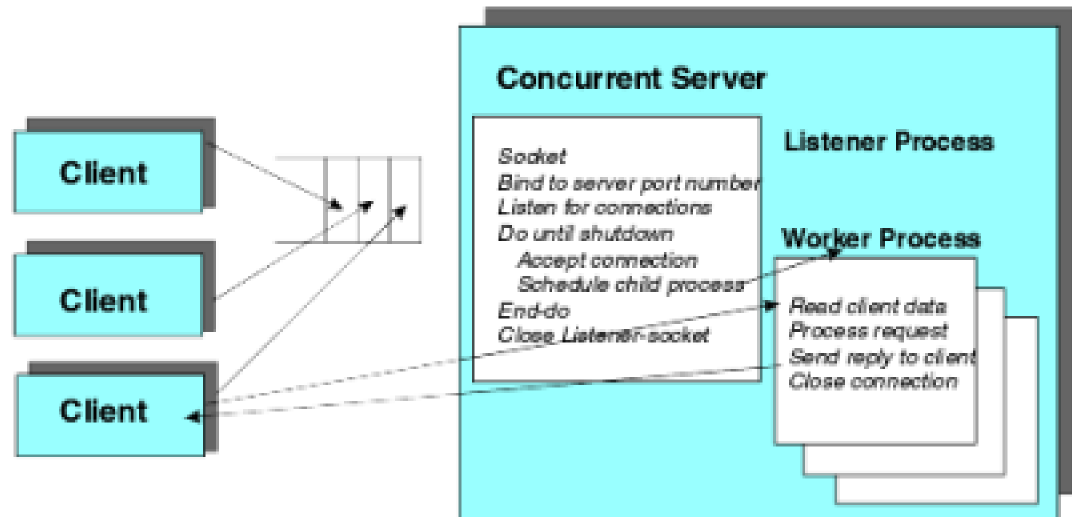
© Copyright IBM Corp. 1998, 2001

Concurrent server programs

If the number of client requests is high and/or the time to process individual client requests is of varying length, an iterative server is not an efficient implementation. A *concurrent server* will give better overall performance. The structure of a concurrent server is shown in Figure 2.

Figure 2. Concurrent Server Structure - Process Model

A CONCURRENT server is able to process a number of client requests concurrently.



A concurrent server consists of two programs:

1. A *listener* program

   This program is actually a small iterative server program, but instead of processing each client request itself, the listener program schedules a worker program for every request it receives, and immediately prepares to receive the next client request.

2. A *worker* program

   When the listener receives a client request, it schedules the worker program. Each instance of the worker program processes one client request and then terminates.

The technique used by concurrent servers allows a high degree of parallel processing, where a number of worker programs can execute concurrently, each serving one client.

The listener program

The listener program can be designed to handle connection-oriented or transaction-oriented work.

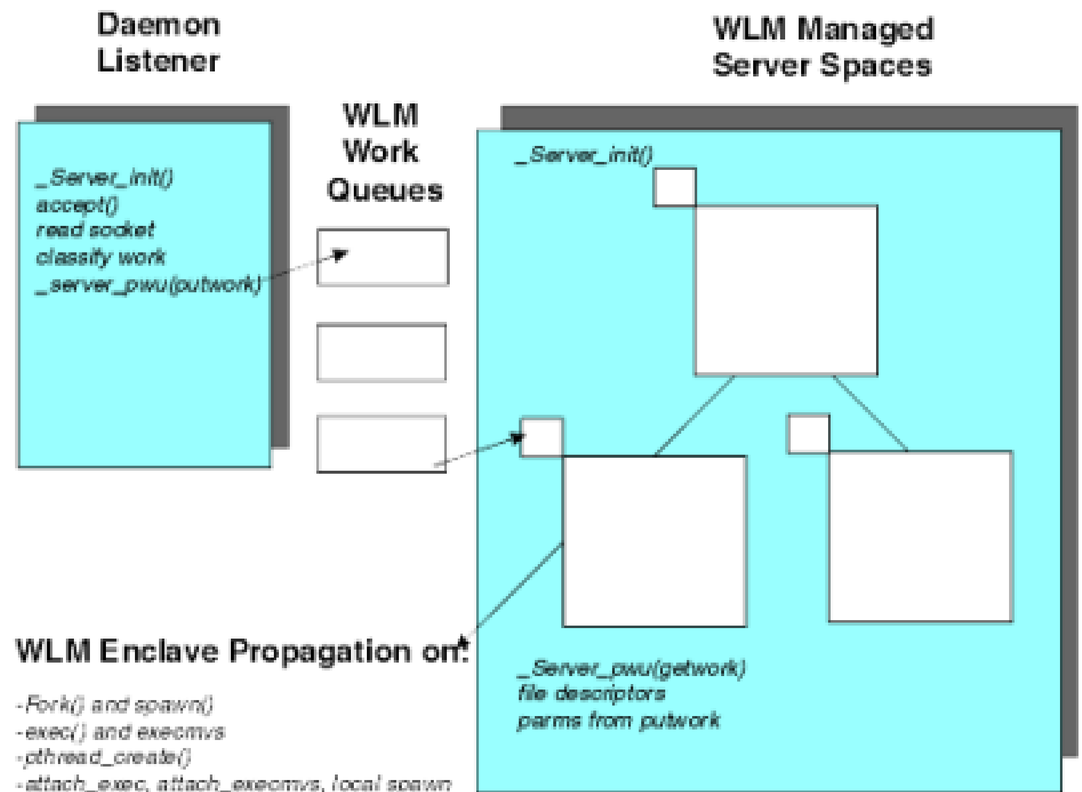
- Connection-oriented work is the type where the user logs in during the morning and stay connected for hours — for example, Lotus, SAP, or rlogin. Performance during initialization is not that critical. Here are two models:

– The listener program may start the worker program as a new process via a fork() call followed by an exec() call, or via a spawn() call. This technique is often used where the individual worker processes are supposed to execute for a longer period of time, entering a dialog with the client program. An example is rlogin. The rlogin client end-user enters a series of requests in one session before the session is terminated on request from the end-user. Because socket descriptors are inherited by the worker process, the socket that represents the connected client program is available to be used in the worker process.
– The listener program and worker program communicate via an AF_UNIX socket. With the AF_UNIX socket, the listener daemon can use the sendmsg()-recvmsg() protocol to transfer the socket to the server that will do the work. Lotus is an example of this approach.

- Transaction-oriented work typically involves short transactions: a user accesses the server with one or two requests, as with a web server. For transaction-oriented work, the server needs to be fast and able to support large numbers of concurrent clients.

  For the Internet Connection Server (ICS), IBM created a transaction-oriented model that uses new proprietary services for Workload Manager (WLM). WLM goals are defined for the web work: its priority, its response time, and so forth. In this model, WLM dynamically manages the creation of server address spaces, as needed. The daemon listener program accepts the work, classifies it, and then puts it on a WLM work queue for the worker program to handle. In the work queue, a WLM enclave is created for the work, containing the data and the socket descriptor. The WLM enclave provides a means for managing each task as an individual entity, controlling the amount of resources it consumes. The work is then put into a WLM-managed address space, which receives the data and socket descriptors, and does the work.

Figure 3. Using the WLM Server Services

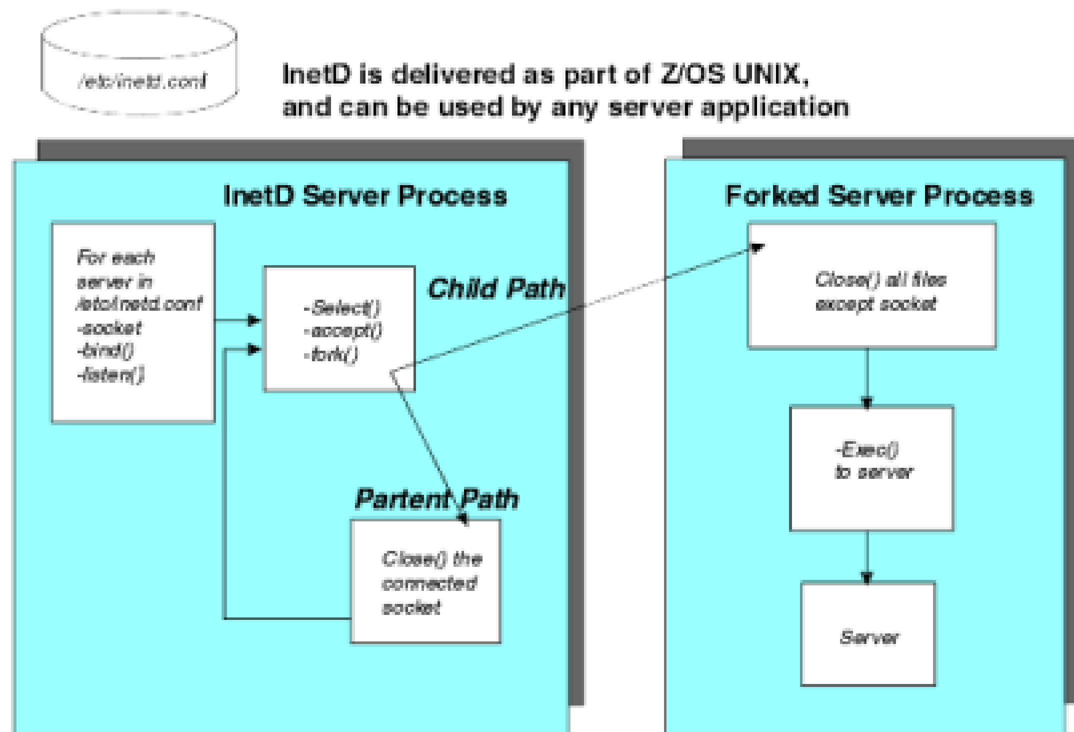


The number of address spaces and the number of threads per address space that the application requires depends on the amount of of resources used by each process, the size of the machine you are running on, and whether there are other applications competing for the system resources.

The InetD generic listener program

Listener programs are very much alike. The real difference between servers may be seen in the worker programs, because it is here the actual application-specific tasks are being performed. In fact the listener program has been generalized to an extent that many different server applications share one and the same listener program. This generic listener program is called *INETD* and is being used by such servers as TelnetD, REXECD, and RSHD. Other servers have implemented their own listener programs and cannot use InetD. For example, the TCP/IP FTP server and the Internet Connection Server do not use InetD, but supply their own listener programs.

Figure 4. INETD Overview



To specify the server applications that INETD is supposed to act as listener for, you update the INETD configuration file, which by default is located in /etc/inetd.conf.

In /etc/inetd.conf you specify the names of the services that INETD supports in your environment. The service names must exist in your /etc/services file, because INETD uses the getservbyname() call to find out which protocol (TCP or UDP) and port number is assigned to the server application. INETD opens sockets, binds them to the port numbers in question, and enters a loop to accept new client requests. When a client request arrives, INETD uses the fork() function to start a new process, where the forked INETD program uses the exec() function to start the server-specific program. The name of this program is specified in the inetd.conf file together with other information for the individual servers.

If you have to pass any run-time options to the server programs that are started via INETD, you have to specify these options in inetd.conf along with the server program name. If you, for example, want to enable tracing in the telnetD server, you can specify the telnet service line in inetd.conf as follows, where the -D flag is the telnet debug flag and the -t flag is the telnet trace flag:

```
telnet   stream tcp nowait OMVSKERN /usr/sbin/otelnetd otelnetd -l -m -D all -t
```

Table 1 is an overview of the listener programs that are used for servers in the z/OS UNIX System Services (z/OS UNIX) environment:

Table 1. Listener Program Overview

Server function	Listener program	Server worker program	Worker program environment
Remote login	InetD	/usr/sbin/rlogind	Separate process
Telnet	InetD	/usr/sbin/otelnetd	Separate process
Remote shell	InetD	/usr/sbin/rshd	Separate process
Remote execution	InetD	/usr/sbin/rexecd	Separate process
Echo, Discard, Chargen, Daytime and Time	InetD	Served internally in InetD	
FTP	FTPD	/usr/sbin/ftpdsrv	Separate process
Web Server	IMWHTTPD	Served internally in IMWHTTPD	Separate thread

Starting listener programs

You need to start your listener programs during z/OS UNIX startup. Here is a sample shell script to start all your daemons:

```
#
# Example to Start Daemons
#
syslogd
inetd
ftpd
sleep 120
```

The sleep ensures that, if you are in batch, all the daemons are started before the script exits, otherwise you kill them when you exit.

If no AF_INET transport provider is connected to z/OS UNIX when you start your listener programs, the listener programs will get an error when they try to bind their listener socket to a port number.

- Some listener programs, such as INETD, handle this situation by writing out a message to syslog:

  ```
  EDC5112I Resources temporarily unavailable
  ```

  The servers then wait a little while before they retry the failed bind() socket call. If an AF_INET transport provider had connected in the meantime, the bind() call is now successful and processing continues normally.
- Other listener programs, such as FTPD, stop processing in this situation and you have to manually restart the FTPD listener process a little later, when your AF_INET transport provider has connected to z/OS UNIX.

Depending on how your listener programs handle the above-described situation, you may use different techniques to start them:

- If the listener program includes retry logic, you can start your listener program during z/OS UNIX initialization from the /etc/rc shell script that is executed during z/OS UNIX initialization.
- If the listener program does not include retry logic, it may be a better technique to postpone starting the program until after your AF_INET transport provider has connected. Then, either start it manually or develop some automation logic to start the listener program after having received a message that a stack has connected to z/OS UNIX.

Security for server programs

When setting up security, you can choose to have a client task run under:

- The server identity, as it does in Lotus.
- A client user ID, which provides task-level ACEE. There are two modes:
  - Authenticated mode, where the user provides an ID and password. Although other UNIX systems do not provide the capability to associate a client user ID on threads, z/OS UNIX provides a proprietary pthread_security_np service that lets each request run with the identity of the user.
  - Surrogate mode, where the server is set up to act as a surrogate for the client. Some servers process user requests that come from generic user IDs representing anonymous users (as with anonymous FTP), or use a method of authentication other than a user ID and password combination.

rhosts.data file: Some RSHD servers implement an rhosts.data file on the RSHD server host to authorize certain client users on certain remote hosts to execute commands on the RSHD server without authentication in terms of verifying a password. This implementation has, for security reasons, not been ported to the z/OS UNIX RSHD server, and its RSHD server does not process any remote **rsh** or **rexec** commands without a valid MVS user ID and password. For example, an RSH client would use this command:

```
rsh mvs18o -l userid/password ls -al
```

On the *-l* parameter, the end user has to enter both an MVS user ID and password with the user ID and password separated by a slash (/).

For more information about thread-level security, go to "Enabling thread-level security for servers" in the Security chapter.

Chapter 13. Database migration

UNIX database applications are typically migrated to DB2 or to Oracle as part of any port to the z/OS UNIX System Services (z/OS UNIX) environment.

- **DB2**

  z/OS UNIX applications can access the existing DB2/MVS product.

  – **Migrating an Application from Oracle to DB2** is an experience report from the IBM Software Migration Project Office.

  – Migrating a DB2/6000 database application to DB2/MVS is documented in **Porting a DB2/6000 Program**, a chapter from the redbook *Porting Applications to the OpenEdition MVS Platform*, GG24-4473. Although this book is based on MVS 5.1, the DB2 porting information is up-to-date.

  – **Porting an AIX DB2 Application to z/OS UNIX** is an experience report from a team that ported a C/C++ AIX 4.2 UNIX application using DB2 to OS/390 Version 1.2. It discusses porting issues raised by z/OS DB2 and how they were resolved.

  Move for Servers, a data extract, transformation and migration tool that now supports DB2 for z/OS databases is available from Princeton Softech, a wholly owned subsidiary of Computer Horizons Corp. Move for Servers enables users to create and exchange relationally intact data subsets between Oracle, DB2 Universal Database and DB2 for z/OS databases, in preparation for cross-platform migration, application testing, staging data for production, archiving production data, or creating realistic, time-shifted data for Year 2000 testing.

- **Oracle**

  With Oracle7 for MVS Version 7.1.6 or higher, Oracle applications in C or Assembler can be executed from an z/OS UNIX environment. A UNIX application running under z/OS UNIX Services can access the Oracle database on z/OS. Therefore, when an application using an Oracle database on a UNIX platform is ported to z/OS UNIX, it will run with the Oracle database on z/OS UNIX. (The Oracle database will, of course, need to be loaded from the UNIX platform to z/OS UNIX). Refer to the *Oracle for z/OS User's Guide*.

- **Sybase**

  Sybase's UNIX relational database manager is SQL Server. Sybase does not have an MVS relational database manager and has no announced plans for one. Sybase's MVS strategy is to use their OpenClient and OpenServer gateway products. A Sybase client writes to the OpenClient API to pipe SQL statements to DB2 on MVS. In this way, a Sybase client application that previously accessed SQL Server can access DB2 with no changes.

  You can migrate a Sybase database to DB2. For DB2 migration, the IBM Software Migration Project Office (SMPO) has a services offering with ManTech Systems Engineering Corporation (MSEC) for the SQL Conversion Workbench, set of tools that automate conversion. Contact Dominic Marrese (dmarrese@us.ibm.com) for more information.

- **Informix, Ingres**

  There are currently no plans to port Informix or Ingres databases to z/OS. This decision was made for business reasons, not for technical reasons. If you have a requirement for Informix or Ingres on z/OS, send it to CALLS390 (calls390@us.ibm.com).

© Copyright IBM Corp. 1998, 2001

ManTech Systems Engineering Corporation (MSEC) has the the SQL Conversion Workbench available to perform Informix-to-DB2 conversions for customers. Contact Dominic Marrese (dmarrese@us.ibm.com) for more information. The SQL Conversion Workbench is available by license or franchise.

Porting an Ingres Database Application is a first-person account about porting an HP/Ingres transaction server application that handles stock transactions to an z/OS UNIX-DB2 environment.

Chapter 14. After the port, focus on performance

After the ported application is working, you can optimize performance, if necessary. To run the application efficiently, you may need to extend it to exploit facilities unique to z/OS. Here are some recommendations to consider. Keep in mind that if it was poorly written code on the original platform, it is still poor code. As an example, in one poorly performing application, we discovered that the application was opening and closing a file inside of a loop. Once we changed the code to move the open and close outside the loop, the performance jumped dramatically.

To improve performance in the z/OS environment, some recommendations are:

- “Use spawn() rather than fork()”
- “Use a threading model instead of a process model” on page 95
- “File I/O and Memory” on page 95
- “Character I/O” on page 95
- “Character set conversion” on page 95
- “Shared memory” on page 96
- “Do not use spins with serialization” on page 97
- “Compile your production application with optimization” on page 97
- “For large load modules, consider using LPA or VLF” on page 97
- Scrutinize any pthread_yield() calls in mainline application paths
- “Using HEAPPOOLS for malloc and free requests” on page 98

There is a profiling tool available that can provide detailed information about where an application is spending most of its instructions. Close examination can reveal questionable programming practices. You can further optimize high usage routines to improve performance.

There are memory leak detection tools available.

Use spawn() rather than fork()

The z/OS platform has some performance characteristics that are not common to other UNIX platforms. A fork() causes z/OS UNIX to create another address space and clone the running application. This is an expensive operation that can be avoided by changing the application if warranted. If the fork()’ed process are long running processes, fork() performance will be acceptable. If they are not long running, you should replace the fork() should with either a local spawn() or a pthread_create(). Both of these substitutions are nontrivial, except for the case of a fork() followed by an exec(). In this case, the substitution to a local spawn is simple.

In the spawn() case, spawn’s argument is the name of an executable module as well as other arguments. Since spawn() does not clone the heap or stack, you must pass data needed by the spawned module or move it to a shared memory segment. You need to change program logic to make another new main() that can be spawned or change the existing main() to get to the point of the fork(). In either

© Copyright IBM Corp. 1998, 2001

case, you will have to initialize data areas since the heap and stack are not cloned. The temptation of passing in the parent's heap should be ignored. Heaps are not made to be shared by multiple processes.

If your application creates many processes, to improve performance set the environment variable _BPX_SHAREAS to YES or REUSE and use spawn(). Similar to fork() and exec(), spawn() runs much faster and saves resources because it does not have to copy the address space. However, if you do not set the environment variable _BPX_SHAREAS to YES or REUSE, spawn will do exactly what fork() and exec() do, and there will be no performance improvement.

If your application is multithreaded you *must* use spawn() instead of fork().

If your application is designed to create multiple copies, with each running the same program, then spawn() might not be useful. Many applications rely on having program initialization performed once by the parent process and propagated via fork() to all the child processes. The spawn() function only propagates a few things like open file descriptors. spawn()'s assumption is that the new process will run a *different* program, not another copy of the same one.

__spawn() provides the ability to do a spawn that has additional data in the inheritance structure. You can specify the userid, cwd, umask and some other things as well. For example, when used with a web server, __spawn() allows a web worker thread that has used pthread_security_np() to spawn a CGI script which will be set up with the correct security identity.

If your application uses pipes or shared memory and you switch to using spawn(), read the following:

- **Applications that use pipes**

  After changing from using fork() to spawn(), an application that uses pipes can appear to hang. Often one process will work correctly for a while, then get stuck in a blocking read of the pipe.

  A pipe consists of two file descriptors (fd) such that data written to ″fd B″ of the pipe can be read from ″fd A″ of the pipe. When a process forks, the pipe gets copied as well. Data written to ″fd B″ in the parent can be read from ″fd A″ in the child. When using fork(), the parent and child both close their copy of the unused pipe file descriptor. Normally, when data flows from parent to child, the parent closes ″fd A″ and the child closes ″fd B″. When data flows the other way, from child to parent, the parent closes ″fd B″ and the child closes ″fd A″. In either case, each process uses and leaves open only one half of the pipe.

  With spawn() there is no explicit way for the child to close its unused half of the pipe. Because both ends of the pipe are open in the child process, the child will never see EOF on a read of ″fd A″ — the write half or ″fd B″ is open in the child. EOF is detected only on a read of ″fd A″ when the pipe is empty and all copies of ″fd B″ are closed.

  The solution is for the parent to mark the file descriptor for its half of the pipe (normally ″fd B″) to be closed-on-exec. If this is done then ″fd B″ will not be open in the child. When the parent closes its ″fd B″ then EOF will be detected in the child after all available data has been read from ″fd A″.

- **Applications that use shared memory**

  When using spawn(), an application that uses shared memory may find that shmat() with a specified shmaddr returns -1 if both processes are in the same address space, although this appeared to work on previous tests. The problem

with a returned -1 would only occur if the application had previously used spawn() with _BPX_SHAREAS=NO and then switched to spawn() with _BPX_SHAREAS=YES or REUSE.

If you require the shared memory to be at the same address, you have two choices:

– Run the two processes in separate address spaces and do the shmat(), specifying the same starting address.
– When running in the same address space, pass the address of the shared memory from process 1 to process 2. Then in process 2, just use the shared memory and do not do the shmat(). If only 2 processes are involved, then regular memory will suffice, and a malloc() in the first process can replace the shmget(). Then just pass the address of the heap storage to the second process.

Use a threading model instead of a process model

Threads are a good alternative because they can be started and stopped more efficiently than processes. z/OS UNIX supports heavy-weight threads — if you are using multiple threads, each thread can run on a different processor at the same time.

One limiting factor is the number of threads in the address space. For a discussion of how many threads you can run in an address space, see the topic ″Limitation on the number of threads″ in the Process Management chapter. See our web page that has suggestions for how to increase the number of threads in an address space. This includes information on the HEAPPOOLS runtime option.

File I/O and Memory

When doing file I/O, keep these guidelines in mind:

- Do your work in memory rather than in temporary files.
- If your application extensively uses temporary files to save data, consider replacing this logic to use memory instead. On some UNIX platforms, memory is limited, so some applications use temporary files to avoid out-of-memory errors. Take advantage of z/OS UNIX's abundant memory to do work.
- Use larger buffers for file I/O. For peak performance, use buffers sized in the range 64K to 256K.
- Don't open a file unless you are going to read or write to it, and don't close a file until you have finished working with it.

Character I/O

Many UNIX applications read data from files one byte at a time. For z/OS UNIX, consider changing the application to read ″lines″ or ″records″ instead of characters.

Likewise, many UNIX applications read data from terminals one byte at a time. If possible, consider reading ″lines″ instead of characters.

Character set conversion

The guidelines for efficient data conversion are similar to those for efficient I/O:

- iconv_open() and iconv_close() should be done at the same time as fopen() and fclose(), that is during application initialization and termination. iconv_open() and iconv_close() services are expensive and are intended to be part of program initialization and termination. Sometimes, to simplify code development, iconv_open() and iconv_close() calls are issued every time translation is needed. We have seen performance greatly enhanced in some cases, when an application was changed to do iconv_open() only once (during initialization) and iconv_close() only once (during termination).
- Buffer as many bytes of data as possible on calls to iconv(). For example, if a line of data is read, the entire line should be passed to iconv().

  Many UNIX programs being ported to the S/390 platform were written to read and write a byte of data at a time. Hence, iconv() would be called for each byte. The overhead to call iconv() and set up for conversion of a buffer of data is fairly high (on the order of 100 instructions per call), whether there is one byte or many bytes in the buffer. However, once the setup is done, it only takes 5 or 6 instructions per byte for iconv() to convert buffered data.

Shared memory

Shared memory — shmat() — is typically used between server processes or used by server address spaces to communicate with clients.

On z/OS UNIX, shared memory is as efficient as any other type of memory access. When you use it, you need to be aware of its impact on the extended system queue area (ESQA) storage requirements. ESQA storage is in common and page fixed, which causes it to consume real memory. A number of z/OS UNIX System Services use base z/OS functions that consume ESQA storage. Installations having constraints on virtual storage or main memory can control the amount of ESQA storage consumed. Ensuring the appropriate size of ESQA and extended common service area (CSA) storage is critical to the long-term operation of the system.

For each real page of shared storage, a 32-byte anchor block is allocated in ESQA. In addition, for every address space accessing that page, an additional control block is allocated — let’s call it a page block for this discussion. The anchor block and the page block are very similar in structure (both 32 bytes), but their fields are different. Both anchor blocks and page blocks are allocated in fixed ESQA storage and they consume real memory.

Example of shared-memory consumption of ESQA:

A server that allocates 8MB of shared memory and has 500 clients connected to it will consume the equivalent to 33MB of ESQA:

```
8MB * 256 pages/MB * 503 connections * 32 bytes/page
    or 33MB of ESQA
```

The 503 comes from 500 clients, 1 server, 1 anchor block, and 1 connection to a kernel data space used to manage the storage.

For information about controlling the use of ESQA, see *z/OS UNIX Planning*.

If you are using memory mapping with large files or large shared memory segments, OS/390 V2R6 provides new programming options that reduce the real storage requirements. The shmget() and mmap() C functions have the new options

that require the storage to be allocated in megabyte multiples and reside on megabyte boundaries. All processes sharing these megabytes have the same access to the storage.

- The __IPC_MEGA option of shmget() (BPX1MGT callable service) allows applications to use large quantities of shared memory without excessive system overhead.
- The __MAP_MEGA option of mmap() (BPX1MMP callable service) allows applications to map very large files without the overhead in ESQA.
- The functions munmap() (BPX1MUN callable service) and mprotect() (BPX1MPR callable service) have a different scope when they are used with memory maps that have been created with the __MAP_MEGA option. When munmap() is used to unmap a MAP_MEGA mapping, entire segments are unmapped. When mprotect() is used to change the access protection of a MAP_MEGA mapping, the change is system-wide. All active maps to the same file-offset range are affected by the request.

Do not use spins with serialization

If you are writing an application that runs in multiple processes or on multiple threads, it is not uncommon for these work units to need to share resources. Sharing resources also implies the need to serialize access to these resources. There are several ways to serialize access to shared resources:

- When sharing a resource across processes, use semaphores. See the explanation of the C functions semget(), semctl(), and semop().
- When sharing resources between threads, you can use mutexes or condition variables. See the explanation of the C functions pthread_mutex_init() and pthread_cond_init().

These serialization mechanisms are provided by the operating system or runtime library. Sometimes programmers feel these functions perform too slowly and create their own mechanisms to handle serialization. Avoid these common mistakes:

- Spin loops that check for a resource being available in an infinite while loop.
- Spin loops that check for a resource being available and then usleep() for a small amount of time before checking again.

In an z/OS system, these loops can consume excessive CPU cycles while preventing other users from running.

Compile your production application with optimization

For each release of the compiler, we have a web page that lists the various optimization options available to improve your application's performance. For example, using the IPA option with the the compiler puts high usage routines inline where called. This eliminates the call overhead entirely.

For large load modules, consider using LPA or VLF

To lessen the impact of very large modules when you have thousands of users, turn on the sticky bit and put the module into the link pack area (LPA). Your users can then share a single copy of the load module. This greatly reduces the working set size for each user and reduces system paging activity. If this process does any forks, the forks will be speeded up.

If you cannot put your module into LPA for any reason, but the module will be loaded into many address spaces or loaded repeatedly into a few address spaces, consider using the Virtual Lookaside Facility (VLF). To do this:

1. Turn on the sticky bit
2. Put the module into a link list data set or a steplib
3. Define the load library to VLF so that the module gets cached.

VLF will then have the module in storage and you will avoid the I/O to fetch the module each time. However, the module will still consume storage in each address space using it.

pthread_yield() calls in mainline paths

pthread_yield() (Thread.yield() in Java) is intended to allow some thread other than the current thread to get control of the processor. On some platforms, calling this service gives the processor to another thread without any fixed length delay in the calling thread. However, on z/OS UNIX pthread_yield gives the processor to another thread by putting the current thread in a timed wait. Sometimes the duration of this timed wait can cause delays in response time and drops in external throughput. Any pthread_yield() calls in mainline application paths should be scrutinized. In most cases, these pthread_yield calls should be removed from mainline paths.

Using HEAPPOOLS for malloc and free requests

If you are running a multithreaded application and doing frequent calls to malloc, free, or other heap storage functions, consider turning on the HEAPPOOLS runtime option. HEAPPOOLS is designed to manage malloc and free requests without getting a lock. It uses compare and swap logic to accomplish a malloc or free in about 50 instructions. Without HEAPPOOLS, a malloc or free will take 300 instructions; plus the lock, which may trigger a WAIT/POST. The type of application that will benefit most from HEAPPOOLS is a multithreaded application that obtains and frees lots of small (4K) pieces of storage.

Chapter 15. Packaging for z/OS installation

Packaging an application for installation on an z/OS UNIX system is different than packaging for installation on other UNIX machines. Customers on the zSeries platform are accustomed to using SMP/E, the basic tool for installing and maintaining software in z/OS systems and subsystems. SMP/E installation may be a requirement some z/OS customers place on all software installed on their mainframes.

On other platforms, products like InstallShield check product levels and update the registry. SMP/E exerts change control for a product at the element level by:

- Selecting the proper levels of elements to be installed from a large number of potential changes
- Calling system utility programs to install the changes
- Keeping records of the installed changes

If you are shipping a fix for an z/OS UNIX application, with SMP/E you can just replace parts of the application, instead of the entire product, as you would on other UNIX platforms.

	Other UNIX platforms	z/OS UNIX
Packaging medium	tar or pax file CD 4mm tape	tar or pax file CD 34xx tape 4mm tape
Installation method	ftp InstallShield	ftp SMP/E
Service philosophy	Total product replacement	Change/replace parts, not entire product

zSeries mainframes don't directly allow connections to CD-ROM readers, so the process of installing from a CD is different. The software can be transfered from a CD-ROM to a Windows or Unix machine, and then FTP'd to a file on zSeries. If this is an SMP/E-installable version of an z/OS product distributed on CD-ROM, then the next step is to install the software using SMP/E.

Learning about SMP/E

Independent Software Vendors need to have SMP/E skills in-house to offer and maintain their zSeries products, and to apply IBM maintainance to their own internal zSeries machines. Using SMP/E requires knowledge of basic z/OS Job Control Language, and knowledge of a TSO or ISPF-based editor and a job submission/monitoring mechanism.

To learn how to make your product SMP/E-installable, you can:

- Read the book MVS Software Manufacturing Standard Packaging Rules for MVS-based Products, SC23-3695. You will need to obtain a product identifier, called an FMID, for your application, which is used in the SMP/E install.

- Attend a 2-day class on the basics of SMP/E packaging. Contact Keith Tilley, krtilley@us.ibm.com for information about having a 2-day class offered at your location.

Chapter 16. Appendix

This Appendix consists of many of the web pages on the z/OS UNIX web site that are linked to from within the **Porting Guide**.

When we create the PDF file from the **Porting Guide** web pages, **we handle hyperlinks this way:**

- If the link is to an z/OS UNIX web page that is **not** part of the **Porting Guide**, we include the web page in the Appendix of the **Porting Guide**. There are a few exceptions to this — for example, the Tools and Toys page and the Compiler page were not included because they change frequently and are easy to find on the web site.
- If the link is to a page on another web site, we do not include it.

Portable header files

If an application on a UNIX system is not POSIX- or XPG4-compliant, then you may not be able to just move it to an z/OS UNIX system and expect it to compile. Applications that are not POSIX- or XPG4-compliant may include headers that are not supported by z/OS UNIX application services. Porting an application that does not conform to those standards requires that you inspect any headers that may not be present on an z/OS UNIX system and determine whether or not the application really requires them. As you know, headers can contain all kinds of things, from macros that simply exist for convenience to prototypes for functions that may or may not exist on a particular UNIX system.

Here is a list of some headers that you will not find on an z/OS UNIX system (this list is not comprehensive):

<access.h>	z/OS UNIX's equivalent interfaces are in <unistd.h>, per POSIX and XPG4.
<ar.h>	No equivalent at this time
<arpa/ftp.h>	No equivalent; you can "borrow" a file from a UNIX system and use it.
<cur01.h>	The <cur01.h> header is not standardized; replace it with <curses.h>
<dir.h>	z/OS UNIX supports <dirent.h> per POSIX and XPG4
<macros.h>	No equivalent at this time
<select.h>	Use <sys/time.h>, as per XPG4 V2. This header contains the prototype for select() and macros like FD_SET, among other things.
<sys/ldr.h>	No equivalent at this time
<sys/mntctl.h>	No equivalent at this time
<sys/mode.h>	This header is non-portable. We use <modes.h> but the standards do not specifically refer to this header. An include for <fcntl.h> is more portable.

© Copyright IBM Corp. 1998, 2001

<sys/param.h>	This header is often unnecessary. Try removing the includes for it and see what falls out.
<sys/ptrace.h>	Although we don’t have this header file, we have a kernel interface (BPX1PTR, see the Callable Services book). The main reason we have this callable service is for the dbx debugger, which was ported from AIX. Much of what AIX’s <sys/ptrace.h> defines shows up in the assembler macro BPXYPTRC. It should be possible to create a header file based on the macro.
<sys/reg.h>	No equivalent at this time
<sys/vmount.h>	No equivalent at this time
<sys/vnode.h>	No equivalent at this time
<termio.h>	z/OS UNIX supports <termios.h> per POSIX and XPG4
<usersec.h>	No equivalent at this time
<userpw.h>	No equivalent at this time

Porting: ASCII to EBCDIC conversion

- Typical problem areas
- “Functions that support ASCII input/output” on page 103
- “Environment variables” on page 18
- “Commands and functions that handle conversion” on page 104

Typical problem areas

When porting a program to z/OS UNIX, tester John Pfuntner says experience has taught him to keep an eye out for these areas where the ASCII to EBCDIC conversion may cause problems:

- **Hard-coded ASCII characters in C code as well as shell scripts**

  Avoid using hardcoded values or depending on the values of characters at all costs. For example, a program might use ’\012’ (octal) instead of ’\n’. A program might use characters as indices into arrays that were populated using the ASCII values for indices
  (for example, hash_table[’a’] —> is not the same as hash_table[0x61]), etc.

- **Using the high-order bit of a character for some special purpose**

  You can do this in ASCII because only 7 bits are necessary for all the printable characters, but that is not true in EBCDIC.

- **Assuming the alphabet (’a’...’z’) is contiguous**

  This is true in ASCII, but not in EBCDIC where there are three noncontiguous groups of letters. Even seemingly harmless code like the following probably needs to be changed: `char c; for (c=’a’; c<=’z’; c++) { ... }`

- **Using code generated by lexx or yacc**

  Often, packages contain C code that were generated by the lexx or yacc utilities. This code will probably contain ASCII dependencies and won’t work on. The code needs to be generated on z/OS UNIX by rerunning the utilities. Note that this may introduce EBCDIC dependencies making the code less portable to other systems but at least it will work on z/OS UNIX.

For example, `y.tab.c` is typically generated by yacc and there should be commands in the package's makefile instructing make how to invoke yacc to rebuild y.tab.c. There should also be a comment in `y.tab.c` that specifies the source file that yacc processed to generate `y.tab.c`.

- **Applications that talk to arbitrary remote systems via sockets (such as an ftp client)**

  These applications typically have to assume all text they receive is ASCII and they send out all text as ASCII. They have to convert the data locally as they go along.

  Consider an ftp client: a user will type a command such as

  ```
  dir foobar
  ```

  The ftp server does not want to see these characters in EBCDIC, so the client must convert the data to ASCII before they are written to the socket. Likewise, if you simply write the data coming from the server to the user's screen, it will be meaningless because it will be in ASCII. The client must first convert the data to EBCDIC. This is true even if the server is running on an EBCDIC system, such as z/OS or VM.

  However, you must be careful to convert only text data. Some applications may mix binary and text in a data stream. For instance, the server might send 2 bytes of binary data preceding a block of text to represent the number of bytes in the block, a cksum value, etc.

- **Code that relies on byte order of data may not be portable.** PC systems are ″little endian″ (that is, the leftmost byte is the most significant); however zSeries and most UNIX systems are ″big endian.″ This typically affects integer and floating point data. If an application is responsible for transferring such data between platforms, you need to either (1) write data exchange logic or (2) translate to text, transfer as text, and then recreate as binary.

Functions that support ASCII input/output

The z/OS C/C++ run-time library functions support EBCDIC characters. The libascii package and V1R3.0 C/C++ __STRING_CODE_SET=″ISO8859-1″ predefined macro provide an ASCII-like application environment on z/OS.

As of OS/390 V2R8, the libascii functions are integrated into the base of Language Environment. If you are running on an earlier release of OS/390, you can download our libascii package, which provides an ASCII interface layer for some of the more commonly used C/C++ run-time library functions. libascii supports ASCII input and output characters by performing the necessary iconv() translations before and after invoking the C/C++ run-time library functions. The __STRING_CODE_SET=″ISO8859-1″ predefined macro generates ASCII characters, constants, and strings. [More]

Setting a variable to convert text files in an archive

You can set an environment variable in your .profile to handle conversion from ASCII to EBCDIC for text files contained in archives. Here is an example showing how to set an environment variable called A2E and then use it:

```
$ export A2E='-o from=ISO8859-1,to=IBM-1047'
.
.
.
$ pax $A2E -rzf foobar.tar.Z
```

"The -o option is not pretty to look at, but once you hide it in a variable, it is easy to use and works perfectly. I have converted millions of bytes of text data this way and have not had a single conversion problem," says John.

Commands and functions that handle conversion

There are shell commands, TSO/E commands, and C functions that handle ASCII to EBCDIC conversion.

Here are two shell commands that are useful:

- **iconv**. For example, the command:

  ```
  iconv -f IBM-1047 -t ISO8859-1 words.txt >converted.txt
  ```

  converts the file words.txt from the IBM-1047 standard code set to the ISO 8859-1 standard code set and stores it in the file named converted.txt.
- **pax**. For example, the command:

  ```
  pax -wf testpgm.pax -o to=IBM-1047,from=ISO8859-1 /tmp/posix/testpgm
  ```

  backs up the /tmp/posix/testpgm directory, which is in the character set CP1047, into an archive file that is targeted to an ASCII character set(IS646).

The **TSO/E commands** OPUT, OGET, and OCOPY let you convert files between ASCII and EBCDIC.

The **C functions** __atoe(), __atoe_l(), __etoa(), and __etoa_l() also perform ASCII-EBCDIC conversion.

Porting services and resources

Here are some consulting services, books, and links to other resources that you might find helpful.

S/390 Partners in Development program

The S/390 Partners in Development program is a free program that assists Independent Software Vendors by providing the resources they need to develop, port, maintain, and market their products on S/390 platforms.

Porting centers

The IBM porting centers are ideal for ISVs or customers with no in-house zSeries system or skills. IBM provides the porting environment, the hardware, software, and access to skills. Typically, the application owner provides a programmer to conduct the port.

Gaithersburg, Maryland (for ISVs and customers):

Contact: Jim Byerly
e-mail: byerly@us.ibm.com
Phone: 301-240-8005

San Mateo, California (for ISVs):

Contact: Dale Wilson
e-mail: ldwilson@us.ibm.com
Phone: 415-312-0241 or 800-678-4249
Web page: http://www.spc.ibm.com

Waltham, Massachusetts (for ISVs):

Contact: John Terlemezian
e-mail: jterleme@us.ibm.com
Phone: 617-895-2564 or 800-678-4249
Web page: http://www.spc.ibm.com

Canada (For ISVs):

Contact: Jacques Albert
e-mail: jalbert@ca.ibm.com
Phone: 905-316-3288

Boeblingen, Germany (Customer proof of concept):

Contact: Roland Reck
email: rolreck@de.ibm.com
Phone: +49-(0)7031-162680
Fax: *49-(0)7031-163232

Stuttgart, Germany (for ISVs):

Contact: Uwe Kopf, Solution Partnership Center
email: ukopf@de.ibm.com
Phone: (0)711/785-1052

Montpellier, France (for ISVs and customers)

Contact: Michel Jan
email: mjan@fr.ibm.com
Phone: (33) 4 67 34 61 83

Hursley, UK (for ISVs and customers):

Contact: Jim Hall
e-mail: jim_hall@uk.ibm.com
Phone: 44-1962-815030

Books

- *Porting UNIX Software* by Greg Lehey, O'Reilly and Associates, Inc., 1996.
- ″Consolidating UNIX Systems onto z/OS″.
- ″Porting C Applications to Lotus Domino on S/390″ SG24-2092
- ″Bringing Windows NT Applications to z/OS″
- *Porting Applications to the OpenEdition MVS Platform*, GG24-4473, has details on one porting experience (Sysdeco) and the methodology used. This helpful book contains valuable hints and tips, and it is packaged with a diskette that contains source code and executables for the tools and examples discussed in the book. Note, however, that the porting information is based on MVS 5.1. For MVS 5.2.2, OS/390 and z/OS releases, OpenEdition support provides much more function. Most of the examples are still relevant.

  In the USA, you can order the book by calling **1-800-879-2755** or faxing 1-800-284-4721. Visa and Master Cards are accepted. If you are outside the USA, contact your local IBM office.

Tools and Toys

We have ported tools and homegrown tools here on the web for you.

Products

We have links to product web pages and information that may be useful for your port.

Performance: tuning targets for UNIX System Services

Performance specialists Bob St. John and Don Corbett say customers are surprised by the difference tuning will make: ″In most cases, the tuning improved throughput by 2 to 3 times, and response time improved by 2 to 5 times. If you are adding UNIX system services to an existing MVS system, you absolutely must take some special tuning steps — because you are combining MVS and UNIX and that's a different ballgame.″

Here's a handy list of steps to take to fine-tune performance and control resource consumption for each release of z/OS:

- OS/390 V2R10: compile-intensive systems
- OS/390 V2R10
- OS/390 V2R9: compile-intensive systems
- OS/390 V2R9
- OS/390 V2R8: compile-intensive systems
- OS/390 V2R8
- OS/390 V2R7: compile-intensive systems
- OS/390 V2R7
- OS/390 V2R6: compile-intensive systems
- OS/390 V2R6
- OS/390 V2R5: compile-intensive systems
- OS/390 V2R5
- OS/390 V2R4: compile-intensive systems
- OS/390 V2R4
- OS/390 V1R3: compile-intensive systems
- OS/390 V1R3
- OS/390 V1R2: compile-intensive systems
- OS/390 V1R2

And here is the background information on the recommended tuning changes:

- "Memory" on page 107
- "Putting frequently used modules in the LPA" on page 107
- "RACF UIDs and GIDs" on page 107
- "File System" on page 108
- "APPC initiators" on page 108
- "Shell variables" on page 108
- "Prevent propagation of TSO/E or ISPF STEPLIB data sets" on page 108

Memory

If your system is running in an LPAR or as a VM guest, the storage size should be at least 64M.

Putting frequently used modules in the LPA

You can move frequently used shell and utility routines, C/MVS runtime library routines, and c89/cc/cxx modules into the LPA. The modules are listed in the tuning recommendations for each OS/390 release (above).

Shell and utility runtime routines:
Shell and utility runtime routines are loaded and deleted as needed when a utility is run. You can improve performance by including frequently used routines in IEALPAxx parmlib member — this places them in the Link Pack Area (LPA). The only constraint is that by doing this you are reducing the amount of virtual storage available to your MVS address spaces.

C runtime library:
C/MVS runtime library routines are loaded and deleted as needed when C programs run. You can improve performance by putting frequently used runtime library routines in the Link Pack Area (LPA); you do this by including them in IEALPAxx parmlib member.

However, there are a few situations when you cannot put runtime library routines in the LPA to improve performance:

- If you already have a version of the Language Environment RTL in linklist, then that is the only version that you can put in the LPA. For example, if you have other, non-z/OS UNIX C programs that require an older version of the C runtime library, then you will not want to put the C/MVS routines in LPA.
- You may want to limit what you put in LPA because putting the routines there reduces the amount of virtual storage available to your MVS address spaces.

The RTL routines that you choose not to put in LPA can be cached in Virtual Lookaside Facility (VLF).

For more information about C/MVS routines that can be placed in the LPA, see *Language Environment for z/OS Customization*.

c89 runtime library:
For development systems where users are doing alot of compiles, be sure to use the list of modules for compile-intensive systems when you look at the tuning recommendations for each OS/390 release (above).

RACF UIDs and GIDs

Caching UIDs and GIDs in VLF:
Caching UIDs and GIDs improves performance. RACF allows you to cache UID and GID information in VLF. You can add VLF options to the COFVLFxx member of SYS1.PARMLIB to enable the caching; the options are listed in the tuning recommendations for each OS/390 release (above).

After you add the options, start VLF, specifying the updated member (in this example, COFVLF33 member) with an operator command:

```
START VLF,SUB=MSTR,NN=33
```

Ensure that all files in your file system have a valid owning UID and GID:
If you restore files from an archive and accidentally keep a UID and GID that were valid on another system, it can create problems that impact response time. For example, say there is an invalid UID associated with a file — when you use a utility that checks the UID (such as ls -l), RACF searches the entire database for the UID ... Time for a trip to the coffee machine!

File System

Our recommendations for the file system are:

- Place HFS data sets on packs that are cached with DASD Fast Write.
- Give each user a separate mountable file system. This lets you spread user file systems across multiple DASD devices, to avoid I/O contention.
- Use control unit caching with DASD Fast Write. We have seen as much as 50 percent faster ″make″ processing with this.
- For OS/390 V1R3 and higher, use the temporary file system (TFS) for **/tmp**.

APPC initiators

For OS/390 Version 1, Releases 1, 2, and 3:
If you are working in a porting environment or if OS/390 UNIX is heavily used on your system, increasing the settings for minimum and maximum APPC initiators can improve performance.

Make sure you have enough APPC initiators defined. To do this, check the console log to see if the system is constantly creating and deleting initiators as commands are run. If this is happening, then increase the minimum number of initiators. A minimum of 20 is probably enough for a few users; you may need more if you are still seeing initiators being created.

The maximum number of initiators should be large enough to prevent the system from running out of initiators.

APPC initiator definitions in the ASCHPMxx parmlib member are included in the tuning recommendations for OS/390 V1R1, V1R2, and V1R3 (above).

Shell variables

You'll see the shell and utilities perform better when you set these two environment variables:

- _BPX_SHAREAS=YES or _BPX_SHAREAS=REUSE
- _BPX_SPAWN_SCRIPT=YES

See our two hot shell environment variables web page (http://webdev10.pok.ibm.com/servers/eserver/zseries/zos/unix/bpxa1shp.html) for information about these two variables.

Prevent propagation of TSO/E or ISPF STEPLIB data sets

You can add a statement in /etc/profile to improve the shell's performance for users who enter the OMVS command from ISPF or with STEPLIB data sets allocated. This prevents excessive searching of STEPLIB data sets and the propagation of STEPLIB data sets from the shell process to the shell command processes on exec. The change that you need to make to /etc/profile is included in the tuning recommendations for each OS/390 release (above).

The next step

The steps discussed above will deliver a noticeable difference in performance. To complete the job, there's more you can do:

- Tune OS/390 UNIX limits in parmlib
- Organize the file system for improved performance

These topics are discussed *z/OS UNIX System Services Planning*.

You can also refer to *z/OS MVS Initialization and Tuning Guide* and *z/OS MVS Initialization and Tuning Reference* for information about the MVS element of z/OS.

Two hot shell environment variables

You'll see the shell and utilities perform better when you set these two environment variables:

- "_BPX_SHAREAS"
- "_BPX_SPAWN_SCRIPT" on page 110

_BPX_SHAREAS

To enable shared address space for the shell, issue the command

```
export _BPX_SHAREAS=YES
```

or

```
export _BPX_SHAREAS=REUSE
```

interactively or place it in your **$HOME/.profile**.

The benefits of _BPX_SHAREAS=YES are:

- The spawn runs faster
- The child address space consumes fewer system resources.
- The system can support more resources.

The side effects are:

- When multiple processes are running with BPX_SHAREAS=YES, the processes cannot change identity information. For example, setuid and setgid will fail.
- You cannot run a setuid or setgid program in the same address space as another product.
- When the parent terminates, the child will terminate because it is a subtask.

With BPX_SHAREAS=REUSE, the child process is created on a subtask in the parent's address space and when the process terminates, system structures for the child process are left in place and reused when the parent spawns another process with _BPX_SHAREAS=REUSE.

With this variable set to YES or REUSE, all simple commands (any command run in the foreground and that is not in a pipeline) will run in processes nested in the shell's address space. If this variable is not set or is set to NO, the shell creates all processes in separate address spaces. No matter how the shell is started (with or without shared address space enabled), you must set _BPX_SHAREAS to YES or REUSE if processes started by the shell itself are to run in processes nested in the shell's address space.

For the OMVS command, use the SHAREAS keyword to enable shared address space. When the SHAREAS keyword is used, the login shell process is nested in the user's TSO address space. Any other login shells started with the OMVS OPEN subcommand are also nested in the user's TSO address space. (With NOSHAREAS, other login shells started with the OMVS OPEN subcommand will each consume another address space.)

User applications can use shared address spaces as well. For details, see the description of the spawn() function and the BPX1SPN and BPX1ATX callable services in *z/OS UNIX Programming: Assembler Callable Services Reference*.

Some processes cannot execute correctly in a shared address space. For example, if a process needs to reserve MVS system resources that are common to all processes in an MVS address space, it must run by itself. If two processes using the same MVS resource attempted to execute concurrently in the same address space, they would compete for these resources thus causing at least one of them to fail. When a potential storage shortage is detected, the new processes are created in their own address spaces, even if _BPX_SHAREAS=YES is present in the invoker's environment. For more details about these restrictions, see the descriptions of the spawn() function and BPX1SPN callable service in *z/OS UNIX Programming: Assembler Callable Services Reference*.

_BPX_SPAWN_SCRIPT

To improve performance when running shell scripts, set the _BPX_SPAWN_SCRIPT environment variable to a value of YES.

This causes the spawn callable service to run files that are not in the correct format to be either an HFS executable or a REXX exec as shell scripts directly from the spawn callable service. The setting of this variable to YES eliminates the additional overhead that occurs when the shell invokes fork after receiving ENOEXEC for an input shell script.

To provide this performance benefit to all shell users, it is recommended that **/etc/profile** or **$HOME/.profile** set environment variable _BPX_SPAWN_SCRIPT=YES.

spawn: After a spawn(), the child process runs the new program specified on the spawn(). The spawn() function is the logical combination of fork and exec; its purpose is to avoid the system overhead incurred with fork.

After a fork(), the child process receives a copy of the parent's storage and inherits open files. Execution in the child continues at the instruction following the fork(). Forking is similar to creating an address space and attaching.

z/OS UNIX Setup Verification

The Setup Verification Program (SVP) lets you check for troublesome setup errors before they trip you up. After you have followed the instructions in *z/OS UNIX Planning*, and completed your setup and customization (including the shell and utilities), you can run the SVP.

Using the SVP, you can:

- Verify that each user has a UID and OMVS segment defined, and each group has a GID.
- Check for duplicate assignment of UIDs and GIDs.

- Verify that each user has access to and owns a home directory and has read, write, and search access to it.
- Check the permissions for several directories usually set up at installation.
- Check that files in the /dev directory are defined correctly. Reconcile the number of pseudo-ttys and file descriptor files with the BPXPRMxx definitions.
- Verify that the shell will run.
- Verify that the OMVS command will run.
- Check customization for utilities. The program checks:
  - files that have been copied from /samples to /etc
  - terminfo files
  - settings for some environment variables
  - ability to compile and run a program

  and performs various other checks.

If it detects a problem, the SVP warns you about it and, if you request, corrects the problem. We estimate the SVP can take up to one-half hour to complete; but the exact amount of time depends on your system.

To use the SVP,
- You must be a superuser (UID=0) with RACF SPECIAL authority, or the equivalent.
- Your system must be at MVS release 5.2.2 or higher, or any release of OS/390 or z/OS.
- Your system must be at ISPF version 4.1 or higher.
- You can use any security product; RACF is not required.

Downloading and Running the Program

You can download the Setup Verification Program. Follow the instructions in the README file.

We have instructions on downloading through your browser and anonymous FTP.

1. Download the file to your workstation.
2. Upload it into an FB 80 MVS data set.
3. Run the program. For example, from ISPF, run the TSO command
   ```
   EXEC 'prefix.OESVP.EXEC'
   ```
   where *prefix* is your userid.

Feedback

We welcome your feedback on this tool. We want it to be useful to you. Let us know if there are other setup steps that you would like the tool to check. Also, is downloading the tool via ftp acceptable to you? Send us your feedback.

Porting with pthreads

This list of ″differences″ encountered with pthreads and mutexes on z/OS UNIX System Services was originally created by customers who subscribe to the mvs-oe mailing list: Dwayne Blumenberg, Chuck Gehr, Thomas Vogler, and Stephen Wild.

Most of these differences exist because z/OS UNIX implemented the POSIX.4a draft 6 standard rather than the final version, POSIX.1c draft 10. The book *Pthreads Programming* by Nichols, Buttlar, and Farrell (ISBN 1-56592-115-1) has a chapter on these differences.

- **alarm() and malarm() functions**

  The functions alarm() and malarm() will send the SIGALRM signal only to the thread that called alarm() and malarm(). On some other platforms, SIGALRM can be sent to any thread in the process.
- **Thread-safe variants of POSIX routines**

  The pthreads standard defines thread-safe variants of existing POSIX functions (for example, strtok_r instead of strtok); however, these are not available under Open Edition. IBM's response is that under z/OS UNIX the normal versions are thread-safe so you can use them directly. Because the two variants have differing prototypes, this represents a problem if you are porting code which contains the thread-safe variants. You have to either change the code or provide your own versions of the _r routines which map onto the "normal" ones. Here is an example of how to use a macro to create your own version:

  ```
  #define strtok_r(s,sep,lasts) strtok(s,sep)
  ```
- **Static initialization of mutexes**

  z/OS UNIX doesn't allow you to statically initialize mutexes with pthread_mutex_initialiser(). To initialize mutexes, use pthread_attr_init() and pthread_mutexattr_init(). In addition to those two, you may want to use pthread_mutexattr_setkind_np().
- **pthread_delete_key()**

  z/OS UNIX doesn't provide the pthread_delete_key() function.
- **pthread_attr_setdetachstate**

  z/OS UNIX doesn't define pthread_create_detached(), and the call to pthread_attr_setdetachstate() is slightly different, so look at the interface. You can set a variable to __DETATCHED and use this on the call instead.
- **Defaults for DETACHSTATE**

  For pthread_attr_setdetachstate(), z/OS UNIX and other platforms vary in their defaults for DETACHSTATE.
- **Process-shared attribute for mutexes**

  z/OS UNIX does not support mutexes shared across processes. You can use semaphores instead.
- **pthread_getspecific**

  pthread_getspecific() has a slightly different prototype under z/OS UNIX than that specified in the standard.

  z/OS UNIX provides two forms of pthread_getspecific:
  - pthread_getspecific()
  - pthread_getspecific_d8_np()
- **Value returned on error**

  When z/OS UNIX pthread functions encounter an error they return -1 and set errno. Other platforms return the error number as the function value. For example, pthread_mutex_trylock() returns -1 and errno contains EBUSY when the lock is occupied, instead of the POSIX-specified behaviour to return a value of EBUSY.
- **Using a C++ function pointer**

  If you are using C++, you must use specific declarations for functions that will be passed as function pointers to calls such as pthread_create() and

pthread_cleanup_push(), so that they have C linkage. This is also a problem for qsort(), atexit(), bsd_signal(), and for defining signal catchers, signal() and sigaction(). This is not specifically a pthread issue — this limitation appplies to all z/OS C functions. Compilers on some other platforms do not discourage mixing C++ and C functions, which allows undesirable programming practices such as trying to use a C++ function when invoking pthread_create().

Here are some solutions for a situation where C++ code needs the compiler to use C linkage:

- Declare the functions extern ″C″. Using an extern ″C″ wrapper around the function declarations is more portable. If you are writing code for multiple platforms, use this approach.

  Make the function pointer a typedef with extern ″C″ wrapped around it. Then use the typedef in the structure.
- Force C linkage to these procedures by using the **__cdecl** modifier in the function declarations and function definition. This approach is more convenient because it requires fewer code changes, but it is less portable.
- If the funtion is a member of a class, declare the function as static __cdecl. The static declaration should be required on most platforms to keep the class instance's ″this″ from being passed as an argument to the function. __ cdecl is required on z/OS to tell the compiler to generate a function that uses standard C argument and stack manipulation conventions.

• **sigwait**

z/OS UNIX implements sigwait as:

```
int sigwait(sigset_t *set);
```

In the standard it is:

```
int sigwait(sigset_t *set, int *sig);
```

z/OS UNIX returns the signal that interrupts the sigwait function as a ReturnValue; the standard has it being returned in *sig.

So sigwait has a different prototype under z/OS UNIX than in the standard. The confusion over which prototype to use extends to other platforms.

Chapter 17. CHARMAP source for IBM-1047

—>

This is the charmap source file for code page IBM-1047, showing the hexadecimal values for characters. The charmap files are shipped in /usr/lib/nls/charmap. The charmap file names are identical to code page names, for example, IBM-1047. The symbol % is used for a comment character.

```
<NUL>                  /x00
<SOH>                  /x01
<STX>                  /x02
<ETX>                  /x03
<SEL>                  /x04
<tab>                  /x05
<HT>                   /x05
<RNL>                  /x06
<DEL>                  /x07
<GE>                   /x08
<SPS>                  /x09
<RPT>                  /x0a
<vertical-tab>         /x0b
<VT>                   /x0b
<form-feed>            /x0c
<FF>                   /x0c
<carriage-return>      /x0d
<CR>                   /x0d
<SO>                   /x0e
<SI>                   /x0f
<DLE>                  /x10
<DC1>                  /x11
<DC2>                  /x12
<DC3>                  /x13
<RES>                  /x14
<newline>              /x15
<backspace>            /x16
<BS>                   /x16
<POC>                  /x17
<CAN>                  /x18
<EM>                   /x19
<UBS>                  /x1a
<CU1>                  /x1b
<IFS>                  /x1c    % file separator
<IS4>                  /x1c
<FS>                   /x1c
<IGS>                  /x1d    % group separator
<IS3>                  /x1d
<GS>                   /x1d
<IRS>                  /x1e    % record separator
<IS2>                  /x1e
<RS>                   /x1e
<IUS>                  /x1f    % unit separator
<IS1>                  /x1f
<US>                   /x1f
<ITB>                  /x1f
<DS>                   /x20
<SOS>                  /x21
<fs>                   /x22    % field separator
<WUS>                  /x23
<BYP>                  /x24
<LF>                   /x25
<ETB>                  /x26
```

© Copyright IBM Corp. 1998, 2001

```
<ESC>                 /x27
<SA>                  /x28
<SFE>                 /x29
<SM>                  /x2a
<CSP>                 /x2b
<MFA>                 /x2c
<ENQ>                 /x2d
<ACK>                 /x2e
<alert>               /x2f
<BEL>                 /x2f
<SYN>                 /x32
<IR>                  /x33
<PP>                  /x34
<TRN>                 /x35
<NBS>                 /x36
<EOT>                 /x37
<SBS>                 /x38
<IT>                  /x39
<RFF>                 /x3a
<CU3>                 /x3b
<DC4>                 /x3c
<NAK>                 /x3d
<SUB>                 /x3f
<space>               /x40
<SP01>                /x40
<nobreakspace>        /x41        % required space
<RSP>                 /x41
<SP30>                /x41
<a-circumflex>        /x42
<LA15>                /x42
<a-diaeresis>         /x43
<a-diaresis>          /x43        %
<LA17>                /x43
<a-grave>             /x44
<LA13>                /x44
<a-acute>             /x45
<LA11>                /x45
<a-tilde>             /x46
<LA19>                /x46
<a-ring>              /x47
<LA27>                /x47
<c-cedilla>           /x48
<LC41>                /x48
<n-tilde>             /x49
<LN19>                /x49
<cent>                /x4a
<SC04>                /x4a
<period>              /x4b
<full-stop>           /x4b       %
<SP11>                /x4b
<less-than-sign>      /x4c
<SA03>                /x4c
<left-parenthesis>    /x4d
<SP06>                /x4d
<plus-sign>           /x4e
<SA01>                /x4e
<vertical-line>       /x4f
<SM13>                /x4f
<ampersand>           /x50
<SM03>                /x50
<e-acute>             /x51
<LE11>                /x51
<e-circumflex>        /x52
<LE15>                /x52
<e-diaeresis>         /x53
<e-diaresis>          /x53           %
<LE17>                /x53
```

```
<e-grave>              /x54
<LE13>                 /x54
<i-acute>              /x55
<LI11>                 /x55
<i-circumflex>         /x56
<LI15>                 /x56
<i-diaeresis>          /x57
<i-diaresis>           /x57           %
<LI17>                 /x57
<i-grave>              /x58
<LI13>                 /x58
<s-sharp>              /x59
<LS61>                 /x59
<exclamation-mark>     /x5a
<SP02>                 /x5a
<dollar-sign>          /x5b
<SC03>                 /x5b
<asterisk>             /x5c
<SM04>                 /x5c
<right-parenthesis>    /x5d
<SP07>                 /x5d
<semicolon>            /x5e
<SP14>                 /x5e
<circumflex>           /x5f
<circumflex-accent>    /x5f
<SD15>                 /x5f
<hyphen>               /x60
<hyphen-minus>         /x60
<SP10>                 /x60
<slash>                /x61
<solidus>              /x61       %
<SP12>                 /x61
<A-circumflex>         /x62
<LA16>                 /x62
<A-diaeresis>          /x63
<A-diaresis>           /x63       %
<LA18>                 /x63
<A-grave>              /x64
<LA14>                 /x64
<A-acute>              /x65
<LA12>                 /x65
<A-tilde>              /x66
<LA20>                 /x66
<A-ring>               /x67
<LA28>                 /x67
<C-cedilla>            /x68
<LC42>                 /x68
<N-tilde>              /x69
<LN20>                 /x69
<broken-bar>           /x6a
<SM65>                 /x6a
<comma>                /x6b
<SP08>                 /x6b
<percent-sign>         /x6c
<SM02>                 /x6c
<underscore>           /x6d
<low-line>             /x6d     %
<SP09>                 /x6d
<greater-than-sign>    /x6e
<SA05>                 /x6e
<question-mark>        /x6f
<SP15>                 /x6f
<o-slash>              /x70
<LO61>                 /x70
<E-acute>              /x71
<LE12>                 /x71
<E-circumflex>         /x72
```

```
<LE16>                 /x72
<E-diaeresis>          /x73
<E-diaresis>           /x73         %
<LE18>                 /x73
<E-grave>              /x74
<LE14>                 /x74
<I-acute>              /x75
<LI12>                 /x75
<I-circumflex>         /x76
<LI16>                 /x76
<I-diaeresis>          /x77
<I-diaresis>           /x77         %
<LI18>                 /x77
<I-grave>              /x78
<LI14>                 /x78
<grave-accent>         /x79
<SD13>                 /x79
<colon>                /x7a
<SP13>                 /x7a
<number-sign>          /x7b
<SM01>                 /x7b
<commercial-at>        /x7c
<SM05>                 /x7c
<apostrophe>           /x7d
<SP05>                 /x7d
<equals-sign>          /x7e
<SA04>                 /x7e
<quotation-mark>       /x7f
<SP04>                 /x7f
<O-slash>              /x80
<LO62>                 /x80
<a>                    /x81
<LA01>                 /x81
<b>                    /x82
<LB01>                 /x82
<c>                    /x83
<LC01>                 /x83
<d>                    /x84
<LD01>                 /x84
<e>                    /x85
<LE01>                 /x85
<f>                    /x86
<LF01>                 /x86
<g>                    /x87
<LG01>                 /x87
<h>                    /x88
<LH01>                 /x88
<i>                    /x89
<LI01>                 /x89
<left-angle-quotes>    /x8a
<guillemot-left>       /x8a
<SP17>                 /x8a
<right-angle-quotes>   /x8b
<guillemot-right>      /x8b
<SP18>                 /x8b
<eth>                  /x8c
<LD63>                 /x8c
<y-acute>              /x8d
<LY11>                 /x8d
<thorn>                /x8e
<LT63>                 /x8e
<plus-minus>           /x8f
<SA02>                 /x8f
<degree>               /x90
<SM19>                 /x90
<j>                    /x91
<LJ01>                 /x91
```

```
<k>                    /x92
<LK01>                 /x92
<l>                    /x93
<LL01>                 /x93
<m>                    /x94
<LM01>                 /x94
<n>                    /x95
<LN01>                 /x95
<o>                    /x96
<LO01>                 /x96
<p>                    /x97
<LP01>                 /x97
<q>                    /x98
<LQ01>                 /x98
<r>                    /x99
<LR01>                 /x99
<feminine>             /x9a
<SM21>                 /x9a
<masculine>            /x9b
<SM20>                 /x9b
<ae>                   /x9c
<LA51>                 /x9c
<cedilla>              /x9d
<SD41>                 /x9d
<AE>                   /x9e
<LA52>                 /x9e
<currency>             /x9f
<SC01>                 /x9f
<mu>                   /xa0
<SM17>                 /xa0
<tilde>                /xa1
<SD19>                 /xa1
<s>                    /xa2
<LS01>                 /xa2
<t>                    /xa3
<LT01>                 /xa3
<u>                    /xa4
<LU01>                 /xa4
<v>                    /xa5
<LV01>                 /xa5
<w>                    /xa6
<LW01>                 /xa6
<x>                    /xa7
<LX01>                 /xa7
<y>                    /xa8
<LY01>                 /xa8
<z>                    /xa9
<LZ01>                 /xa9
<exclamation-down>     /xaa
<SP03>                 /xaa
<question-down>        /xab
<SP16>                 /xab
<Eth>                  /xac
<LD62>                 /xac
<left-square-bracket>  /xad
<SM06>                 /xad
<Thorn>                /xae
<LT64>                 /xae
<registered>           /xaf
<SM53>                 /xaf
<not>                  /xb0
<SM66>                 /xb0
<sterling>             /xb1
<SC02>                 /xb1
<yen>                  /xb2
<SC05>                 /xb2
<dot>                  /xb3
```

```
<SD63>                  /xb3
<copyright>             /xb4
<SM52>                  /xb4
<section>               /xb5
<SM24>                  /xb5
<paragraph>             /xb6
<SM25>                  /xb6
<one-quarter>           /xb7
<NF04>                  /xb7
<one-half>              /xb8
<NF01>                  /xb8
<three-quarters>        /xb9
<NF05>                  /xb9
<Y-acute>               /xba
<LY12>                  /xba
<diaeresis>             /xbb
<diaresis>              /xbb        %
<SD17>                  /xbb
<macron>                /xbc
<SM15>                  /xbc
<right-square-bracket>  /xbd
<SM08>                  /xbd
<acute>                 /xbe
<SD11>                  /xbe
<multiply>              /xbf
<SA07>                  /xbf
<left-brace>            /xc0
<left-curly-bracket>    /xc0
<SM11>                  /xc0
<A>                     /xc1
<LA02>                  /xc1
<B>                     /xc2
<LB02>                  /xc2
<C>                     /xc3
<LC02>                  /xc3
<D>                     /xc4
<LD02>                  /xc4
<E>                     /xc5
<LE02>                  /xc5
<F>                     /xc6
<LF02>                  /xc6
<G>                     /xc7
<LG02>                  /xc7
<H>                     /xc8
<LH02>                  /xc8
<I>                     /xc9
<LI02>                  /xc9
<syllable-hyphen>       /xca
<dash>                  /xca         %
<SP32>                  /xca
<o-circumflex>          /xcb
<LO15>                  /xcb
<o-diaeresis>           /xcc
<o-diaresis>            /xcc         %
<LO17>                  /xcc
<o-grave>               /xcd
<LO13>                  /xcd
<o-acute>               /xce
<LO11>                  /xce
<o-tilde>               /xcf
<LO19>                  /xcf
<right-brace>           /xd0
<right-curly-bracket>   /xd0
<SM14>                  /xd0
<J>                     /xd1
<LJ02>                  /xd1
<K>                     /xd2
```

```
<LK02>                  /xd2
<L>                     /xd3
<LL02>                  /xd3
<M>                     /xd4
<LM02>                  /xd4
<N>                     /xd5
<LN02>                  /xd5
<O>                     /xd6
<LO02>                  /xd6
<P>                     /xd7
<LP02>                  /xd7
<Q>                     /xd8
<LQ02>                  /xd8
<R>                     /xd9
<LR02>                  /xd9
<one-superior>          /xda
<ND011>                 /xda
<u-circumflex>          /xdb
<LU15>                  /xdb
<u-diaeresis>           /xdc
<u-diaresis>            /xdc            %
<LU17>                  /xdc
<u-grave>               /xdd
<LU13>                  /xdd
<u-acute>               /xde
<LU11>                  /xde
<y-diaeresis>           /xdf
<y-diaresis>            /xdf            %
<LY17>                  /xdf
<backslash>             /xe0
<reverse-solidus>       /xe0
<SM07>                  /xe0
<divide>                /xe1
<division>              /xe1
<SA06>                  /xe1
<S>                     /xe2
<LS02>                  /xe2
<T>                     /xe3
<LT02>                  /xe3
<U>                     /xe4
<LU02>                  /xe4
<V>                     /xe5
<LV02>                  /xe5
<W>                     /xe6
<LW02>                  /xe6
<X>                     /xe7
<LX02>                  /xe7
<Y>                     /xe8
<LY02>                  /xe8
<Z>                     /xe9
<LZ02>                  /xe9
<two-superior>          /xea
<ND021>                 /xea
<O-circumflex>          /xeb
<LO16>                  /xeb
<O-diaeresis>           /xec
<O-diaresis>            /xec             %
<LO18>                  /xec
<O-grave>               /xed
<LO14>                  /xed
<O-acute>               /xee
<LO12>                  /xee
<O-tilde>               /xef
<LO20>                  /xef
<zero>                  /xf0
<ND10>                  /xf0
<one>                   /xf1
```

```
<ND01>                  /xf1
<two>                   /xf2
<ND02>                  /xf2
<three>                 /xf3
<ND03>                  /xf3
<four>                  /xf4
<ND04>                  /xf4
<five>                  /xf5
<ND05>                  /xf5
<six>                   /xf6
<ND06>                  /xf6
<seven>                 /xf7
<ND07>                  /xf7
<eight>                 /xf8
<ND08>                  /xf8
<nine>                  /xf9
<ND09>                  /xf9
<three-superior>        /xfa
<ND031>                 /xfa
<U-circumflex>          /xfb
<LU16>                  /xfb
<U-diaeresis>           /xfc
<U-diaresis>            /xfc         %
<LU18>                  /xfc
<U-grave>               /xfd
<LU14>                  /xfd
<U-acute>               /xfe
<LU12>                  /xfe
<eo>                    /xff
```

Notices

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates.

Any reference to an IBM product, program, or service is not intended to state or imply that only IBM's product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any of IBM's intellectual property rights may be used instead of the IBM product, program, or service. Evaluation and verification of operation in conjunction with other products, programs, or services except those expressly designated by IBM, are the responsibility of the user.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
USA

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
Mail Station P300
522 South Road
Poughkeepsie, NY 12601-5400
USA
Attention: Information Request

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

Any pointers in this publication to non-IBM Web sites are provided for convenience only, and do not in any manner serve as an endorsement of these web sites. IBM accepts no responsibility for the content or use of non-IBM Web sites specifically mentioned in this publication or accessed through an IBM Web site that is mentioned in this publication.

Trademarks and Service Marks

The following terms are trademarks of the International Business Machines Corporation in the United States, or other countries, or both:

AIX
AD/Cycle
ADSTAR
AIX

AnyNet
BookManager
C/370
CICS
DB2
DFSMS
DFSMShsm
IBM
IBMLink
Language Environment
OS/390
RACF
Resource Measurement Facility
RMF
VTAM

Lotus, Domino, and Lotus Go Webserver are trademarks of the Lotus Development Corporation.

Java and all Java-based trademarks and logos are trademarks of Sun Microsystems, Inc. in the United States and/or other countries.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States and/or other countries.

Oracle and Oracle8 are registered trademarks of the Oracle Corporation in the United States and/or other countries.

UNIX is a registered trademark in the United States and/or other countries, licensed exclusively through X/Open Company Limited.

Other company, product, and service names may be trademarks or service marks of others.

Index

Special Characters

[ ] (square brackets) 18
$HOME/.profile 18

Numerics

3270 emulation 4

A

absolute pathnames 51
accessing MVS data sets from z/OS UNIX 55
Adabase 4
ADSTAR Distributed Storage Manager (ADSM) 10
advisory locking 55
ANSI C 3
AnyNet 71
APF-authorized program 33
application tuning 5
archive file transfer 14
arithmetic expressions 21
ASCII characters and strings, debugging 43
ASCII-EBCDIC
  issues 12
  translation table 13
ASCII-to-EBCDIC 7, 52, 82, 95, 102
Assembler 23
authority checking in the HFS 34
authorization protocols 15
authorized programs 33, 44

B

Berkeley sockets 72

C

C++
  considerations 4
  X-Windows 4
C/C++ 23
  compiler options 39
  error handling 26
  function pointers for X11 callbacks 25
  header files 24
  portability 24
  resolver configuration data 78
c89/cc/c++ utility 18
c89 utility
  default settings 37
  options 41
  socket header files 41
caching 5
CEEDUMP 44
char in C/C++ 24
character I/O 95
character set conversion 7, 52, 82, 95, 102
character string testing 20
charmap, IBM-1047 115
checking your setup 21
CHNGDUMP 44
choosing a UNIX application 3
class libraries 4
COBOL 4, 23
code checker 7
Code Integrity 7
code page, IBM-1047 115
code page default 52
code pages 12
collating sequence 13
Common INET PFS 76
compiler
  conditional compilation 41
  default settings 37
  optimization 93
  options 39, 40
  ordering options 39
  setup checker 37
  socket header files 41
concurrent server programs 86
conditional compilation 41
const qualifier in C/C++ 24
CONVERT keyword 17
CONVERT option on OMVS command 19
converting character sets 52, 95
copying data 17
Cscope 21
customizing the shell 17

D

daemon program setup 34
data conversion 7, 52, 82, 95, 102
data exchange
  binary data 16
  text data 16
data sets, opening 55
database 3
  migration 91
db2
  migration 91
DB2 4
dbx 34, 43
debugging 43
DFSMShsm 10
disk space allocation 10
distributed file system 50
dumps 44
dynamic link library (DLL) 26
  for C 27
  for C++ 28

E

ed editor 11
editing
  ed 11
  vi 11
Emacs editor 11
end-of-line delimiters 16
environment variables 18
environment variables and the C/C++ resolver 83
ESQA 96
executable modules in HFS 51
exporting functions and variables 39
extended attribute 33, 51
external links 54
external time reference 29

F

file
  I/O 95
  locking 54
  sharing 53
  system security 53
  types 50
fork() 58, 93
freeware 5
fsck utility 53
FTP client, ncftp 15
function exporting 39
function libraries 38

G

gethostid call 82
gethostname call 82
gnu utilities 29

H

header files 7, 24
header libraries 38
heap storage functions 98
HEAPPOOLS runtime option 98
help online 22
hierarchical file system (HFS) 10, 47
  introduction 47
  security 53
HLLAPI 4
host name resolution 81

I

I/O
  character 95
  file 95
IBM-1047 code page 115
IBM porting centers 8
iconv() 96

© Copyright IBM Corp. 1998, 2001

IDMS 4
InetD generic listener 29, 88
Informix migration 91
Ingres migration 91
integrated sockets PFS 75
interprocess communications (IPC) 64
ISV 3
Iterative server programs 85

J

job control 62

K

keyboard mapping 18
KornShell 9
ksh93 17

L

LAN server 54
language differences, C/C++ 24
language support 23
LANRES 54
large load modules 97
libascii 13
libraries for functions and headers 38
listener program 86
locking a file 55
LPA 97

M

magic value 19
make 38
malloc 98
man pages 22, 28
memory, shared 96
memory mapped files 51
memory mapping 67
message queues 66
mixing options and operands 39
mountable file system 50
mountpoints 15
multithreaded application 93
MVS DASD space 10
MVS data sets from z/OS UNIX 55
myslogin authorization protocol 15

N

ncftp FTP client 15
nedit editor 11
network file system 50
Network File System 9
Network Time Protocol 29
networking 71
NFS 9
NFS Windows client 15
non-standard interfaces 7
nroff 28

O

OCOPY TSO/E copy command 17
OGET TSO/E copy command 17
OGETX TSO/E copy command 17
OHELP command 22
OMVS command 11
OMVS command CONVERT option 19
OMVSDATA IPCS subcommand 44
online help 22
OpenEdition sockets 72
optimization options 97
OPUT TSO/E copy command 17
OPUTX TSO/E copy command 17
Oracle 4
Oracle migration 91
owning the code 3

P

passwords 31
pathnames 51
PCNFS authorization protocol 15
performance
  optimization 5, 93
  tools 93
permission 700 10
Personal Communications/3270 19
pipes 94
port sizing 7
porting centers 8
POSIX 7
POSIX threading services 63
pound bang 19
power failures 53
process
  authorization 61
  control 58
  defined 58
  forking 59
  groups 62
  management 57
  priorities 62
  spawning 60
program size 3
protocol configuration data 81
pthread_yield() calls 98
pthreads, porting 64

R

replacing the program in a process 61
resolver 78
rhosts.data file 90
rlogin command 11
RogueWave 4
root directory 47
root file system 50
runtime environment 43
runtime library 38

S

SDUMP 44
security 11, 31
  for server programs 90
security 11, 31 *(continued)*
  in the HFS 33, 53
  thread-level 31
sed editor 11
semaphores 66
serialization 97
server
  models 85
  security 90
  thread-level security 31
service configuration data 81
setup checker 37
shared memory 65, 96
sharing HFS files 53, 54
shbang 19
shell access
  OMVS command 11
  rlogin command 11
  telnet command 11
signals 67
  with POSIX(OFF) 68
  with POSIX(ON) 69
SLIP command 44
sockets in z/OS UNIX 72
sorting sequence 13
spawn() 58, 93
square brackets 18
starting listener programs 89
stopping threads 64
Sybase migration 91
symbolic links 15
SYSABEND 44
SYSMDUMP 44
sysplex timer 29
system calls 7
system maintenance 11
system tuning 9
SYSUDUMP 44

T

TCP/IP 71
  ASCII-to-EBCDIC 82
  environment variables 83
  hosts file 81
  protocol configuration data 81
  resolver configuration data 80, 82
  service configuration data 81
TCP/IP FTP 14
telnet command 11
template support 24
temporary file system 50
testing character strings 20
thread-level security 35
threaded model 95
threads
  limit 63
  security 31
  stopping 64
  types 63
time management 29
tools 21
  code checker 7
toys 21
TSO/E copy commands 17
tuning target guidelines 37
tuning the system 9

U

UID/GID assignment 61
unauthorized libraries 33
UNIX System V shell 9
unpacking data 17
user ID 11
user security 31
using the shell 17
utilities 21

V

variable exporting 39
vendor APIs 3
vi UNIX editor 11
viascii (ASCII vi) editor 11
Visual SlickEdit 11
VLF 97

X

X/OPEN 5, 7
X/Open sockets 72
X-Windows support 28
XPG4-compliance 7
XPG4 standard 17

Z

z/OS differences 23
z/OS shell 9

Printed in the United States of America