Shell Script Programming in z/OS

Shell Script Programming in z/OS UNIX - Course Objectives

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

1. Use regular expressions in UNIX shell commands, where supported

2. Use the 'ed' line editor as well as the oedit editor to create and maintain shell scripts

3. Work with directories, files, and variables using commands such as grep, find, and typeset

4. Work with shell variables using let and expr commands

5. Create and run shell scripts, including the use of these commands and features:
   a. Prompting the script user for input using echo and read
   b. Conditional logic and looping using if, test, until, while, break, continue
   c. Supporting options and parameters using for, select, case, and getopts

6. Create and use user-defined shell functions, both inside and outside of scripts

7. Use the powerful facilities of sed, the stream editor, including using sed scripts as a tool to convert text files and flat files into HTML files for viewing on the Internet or corporate intranet

8. Use classic and topological sorts of files, and various tools to compare files and directories

9. Run shell scripts and programs in batch using the BPXBATCH facility.

Although not part of the formal lecture materials, the appendices include comprehensive coverage of the vi / ex editor as well as the shell command line editor and its vi, emacs, and gmacs modes, and the bc programming language.

Shell Script Programming in z/OS UNIX - Topical Outline

Shell Script Programming in z/OS UNIX - Topical Outline, p. 2.

Day Two

Shell Script Programming in z/OS UNIX - Topical Outline, p.3.

Day Three

UNIX Standards

There have been a number of UNIX standards that have become popular over the years, so many that "open source" is almost "open chaos". In an attempt to re-establish common ground, two major organizations, The Open Group, an organization sponsored by many of the major vendors, and the IEEE (Institute of Electrical and Electronics Engineers) a well-respected technical, non-profit organization, have combined to establish the Single UNIX Specification (SUS), a document that both organizations have pledged to adhere to.

This document merges and extends standards by establishing a common vocabulary and set of APIs (Application Programming Interface's, including commands and utilities) that build on the IEEE's POSIX standard and The Open Group's UNIX 95 (XPG) standard.

Some web pages that are of interest for those who want to explore more details:

http://www.unix-systems.org/	- main page to explore the SUS from; the standard may be downloaded from here in hypertext format or PDF
http://www.ieee.org/index.html	- home page for the IEEE
http://www.opengroup.org/	- home page for The Open Group

IBM's z/OS UNIX System Services conforms to various levels of these standards and includes extensions to the standards. Remember that while any given extension may be nice to have / use, using such a feature may make your work less portable to other UNIX platforms (or even not portable to such platforms).

In 2006, with the advent of z/OS 1.8, some commands had to change the meaning of some of their options and flags in order to conform to version 3 of SUS (SUSv3). A special environment variable was defined, _UNIX03, such that if that variable has a value of YES, then the new behavior takes effect; if this variable is undefined or does not have a value of YES, the prior behavior is in effect. Places where this is a concern are documented throughout these materials.

In 2008, SUSV4 was released. No information on how this is implemented for z/OS as of this course publication date.

This page intentionally left almost blank.

Section Preview

❒ **Introduction to the class**

- ♦ **The uses for shell scripts**
- ♦ **Scripting languages available**
- ♦ **And now for something completely different**
- ♦ **Class set up (Machine exercise)**

Introduction to the Class

❒ **This course builds on our previous course "Introduction to z/OS UNIX", and so assumes you have taken that course**

- ♦ **Or have equivalent experience**

❒ **In this section, we want to accomplish these tasks:**

- ♦ **Establish the reasons for writing shell scripts**
- ♦ **Describe the various scripting languages available**
  - ✗ Although in this course we will only write scripts using the z/OS shell scripting language
- ♦ **Outline the directions we'll go from here**

❒ **We'll end the section with a lab to set up our files for the rest of the class**

Notes

- ♦ **Appendix D has a summary of the content of the prerequisite course and this course, combined, as a handy reference**
- ♦ **You may run shell labs using** OMVS **or** telnet**, except the first lab and the last lab require using TSO/ISPF**

The Uses for Shell Scripts

- ❒ **The primary reasons people code and use shell scripts are...**
  - ♦ **Simplify using commands**
    - ✗ Scripts can prompt the user for information then build and run the appropriate command(s), thus freeing the user from remembering [or even knowing] the syntax and options of the commands
      - ➢ And "user" here can even include the programmer
  - ♦ **Reduce keying efforts**
    - ✗ Storing a commonly used sequence of commands (such as setting environment variables) reduces this repetitive process to running one script
  - ♦ **Help manage batch jobs**
    - ✗ Scripting in UNIX plays the same role as JCL in running z/OS batch jobs
      - ➢ With even greater flexibility
    - ✗ A script can set up the environment for running a program, run the program, and repeat the process for additional programs
    - ✗ Or the same program with different files or parameters

Scripting Languages Available

❒ In the z/OS environment, you may write scripts using the scripting language provided by each shell

- ♦ IBM supplies two shells: z/OS shell, and tsch shell
- ♦ In addition, you can write shell scripts in REXX
- ♦ PERL and AWK are two other UNIX-based scripting languages, primarily used for CGI scripts

❒ In this course, we will be writing scripts using the z/OS shell scripting language, which provides these facilities

- ♦ A large collection of imperative commands
- ♦ Ability to create, set, and manipulate variables, including variable substitution in commands
- ♦ Ability to accept and use parameters specified when a script is invoked
- ♦ Access to three files: stdin, stdout, stderr, automatically; support for file descriptors
- ♦ Ability to invoke programs resident in the HFS or in z/OS PDS or PDSEs
- ♦ Conditional logic and looping capabilties

And Now For Something Completely Different

❒ **To create a script, you need an** editor **- a program that lets you key in new scripts and modify exisiting scripts in the z/OS UNIX file system (since shell scripts have to be run from there)**

❒ **In the prerequisite course, we used** oedit **to build some small scripts as part of the labs (cmds, .profile, and run_set)**

❒ **But we should also learn about the editors more commonly used in UNIX environments**

- ed **- the shell line editor; used from the command line**
- sed **- the shell stream editor; mostly used in scripts**
- **The** vi **editor used under** telnet **and** rlogin **(including** ex**)**
- shell editor **- built-in shell editor for the command line and history file only; not discussed in this cours**
  - ✗ Our editor focus will be on ed and sed; vi and ex are discussed in the appendices to this course handout

❒ **Before we can work with any editors, we need to work on some concepts familiar to every UNIX user**

- ✗ Shells and processes
- ✗ Regular expressions, and the commands grep, egrep, fgrep
- ✗ The shell find command

❒ **But first, we will run a setup lab to prepare for the rest of the course ...**

Computer Exercise: Class Set Up

To set up for the lab requires some work. First, from ISPF option 6 issue this command:

```
===> ex   '__________.train.library(u515strt)'   exec
```

This will invoke a small dialog to create some files we will use for later exercises. The first thing you will see is a prompt for the high level qualifier to use for the data set names; it is set to be your TSO id and this is probably OK. In any case, set the value you want and press <Enter>. At this point the files you need will be created. The file names will begin with your high level qualifier (<hlq>) followed by TR:

<hlq>.TR.CNTL	(PDS for JCL)
<hlq>.TR.INPUTX	(data file)
<hlq>.TR.LIBRARY	(PDS for data and source code)
<hlq>.TR.LOAD	(PDS for executable programs)
<hlq>.TR.ZINPUTX	(data file)

Next you need to create some directories under your UNIX home directory, so get into the shell (this can be omvs or telnet). Follow these steps:

Make sure you have these directories (you may have them from the prerequisite course, in which case, you don't have to create them):

data	
bin	
tmp	
public_html	(your installation may vary on this one)

Finally, copy these files (again, any that are there before may be used as is); you may use TSO commands (**oputx**) or UNIX commands (**cp**) to do this

<hlq>.TR.LOAD(ASTAT)	—>	/u/*yourid*/bin/astat	
<hlq>.TR.LIBRARY	—>	/u/*yourid*/data	(use 'lc' option of oputx)
<hlq>.TR.INPUTX	—>	/u/*yourid*/inputx	(as binary)
<hlq>.TR.ZINPUTX	—>	/u/*yourid*/zinputx	

Section Preview

❒ **UNIX Applications**

- ♦ **Running the UNIX Shells**
- ♦ **Shells and Processes and Such**
- ♦ **Pseudo-Terminals**
- ♦ **Sessions**
- ♦ **UNIX Jobs**
- ♦ **Commands and Processes**
- ♦ **Shell Commands: tty, sleep**
- ♦ **Shell Commands: ps**
- ♦ **Shell Commands: uname**
- ♦ **Shells, Sessions, and Processes (Machine Exercise)**

Running the UNIX Shells

❒ **Whether you enter the OMVS command from either TSO READY or ISPF 6, or you connect using** telnet**, the information in your security package's OMVS segment is used to start a shell**

- ♦ **In this section, we assume you are using IBM's RACF program product; similar remarks apply to other products**

❒ **You can examine your RACF OMVS segment by issuing the TSO command (from READY, ISPF 6, or omvs command line, but not telnet):**

listuser *username* **omvs**

- ♦ **The response to this command is several lines of display that look something like this:**

```
UID= nnnnnnnnnn              (your numeric user id)
HOME= /u/username            (your initial working directory)
PROGRAM= /bin/sh             (your default shell program)
.                            (other parameters)
.                                  "
.                                  "
```

- ♦ **At this time, we just care about the PROGRAM parameter**

- ♦ **This tells the initialization routines which shell program to start up**

  ✗ /bin/sh is the program name for the z/OS shell, the shell we have been working under during the prerequisite course and will work under in this course

Running the UNIX Shells, continued

❒ **IBM supplies two shell programs**

- z/OS shell **(/bin/sh), based on UNIX System V with some features of KornShell included; conforms to POSIX standard 1003.2**

- tcsh shell **(/bin/tsch), based on the Berkeley UNIX C shell (with some additions / enhancements)**

❒ **You can request the tcsh shell be your default by, in the RACF case, issuing this TSO command:**

altuser *username* **omvs(program('/bin/tcsh'))**

✗ [in a similar fashion, you can alter other properties, such as your home directory; in some cases you need authorization]

❒ **You can supply other alternative shell programs if you like, and set the default in a similar fashion**

❒ **We discuss the major differences between the z/OS shell and tcsh shell in a different course**

Running the UNIX Shells, continued

❒ **When z/OS is started, the z/OS UNIX kernel is also started**

- ♦ **As part of that process, the script called** /etc/rc **is run to accomplish such tasks as**
  - ✗ **Start the INET daemon**
  - ✗ **Setup the automount process**
  - ✗ **Establish system-wide environment variables**

❒ **When you issue** omvs **(or when you** telnet **or** rlogin **to a system running z/OS UNIX), the appropriate shell is started as a <u>login shell</u>**

- ♦ **When a login shell starts, three files are searched to find scripts that establish preferences, options, and environment variables, in this order**
  - ✗ /etc/profile **- system-wide profile script, establish default, standard values in environment variables, say**
  - ✗ $HOME/.profile **- user-specific profile script, if one exists**
  - ✗ **Any file named in the ENV variable (must be set from one of the two scripts above)**

Running the UNIX Shells, continued

❒ **An application is considered to be a "UNIX application" if it uses services from the z/OS UNIX kernel**

- ♦ **This can be done directly through calls to BPX1... services from Assembler, COBOL, PL/I, C, C++, and other languages**

- ♦ **This can be done indirectly by using a shell**

A shell provides:

- ✗ Setup for stdin, stdout, and stderr - assigning these to file descriptors 0, 1, and 2, respectively, and managing these files automatically (*i.e.*: doing open on startup and close when the shell is exited)
- ✗ Facilities to work with local and environmental variables
- ✗ A large number of commands and utilities that make it easy to use kernel services
- ✗ Facilities for passing arguments, parsing statements, specifying patterns ("regular expressions"), and evaluating arithmetic, string, and logical expressions
- ✗ Facilities for running user-written programs / commands / utilities and scripts
- ✗ Conditional logic and looping capabilities
- ✗ Organization / coordination of terminals, processes, sessions, and groups

Running the UNIX Shells, continued

- ❒ **So "running under the shell" means**
  - ♦ **Starting a shell and requesting that the shell run an application under its environment**
    - ✗ Note: you can run a shell in traditional z/OS batch by executing program BPXBATCH from JCL (more later)
- ❒ **z/OS applications that use UNIX kernel services may be able to run without a shell, but that is discussed in a separate course**
- ❒ **To do more advanced work with shells, we need to understand a number of terms and concepts, such as "process", "session", and so on**
  - ♦ **We address that next**

Shells and Processes and Such

- ❒ **The basic unit of work in a UNIX environment is a <u>process</u>**

    - ♦ **A process is, essentially, an address space where a program is running**

    - ♦ **A <u>shell</u> is an executable program, running in an address space, that** <u>creates an environment</u> **for processing commands**

        - ✗ The environment consists of functions, virtual storage, environment variables, settings, and files

        - ✗ When run as a <u>login shell</u>, the login work establishes the initial environment (based on profile scripts discussed earlier) then the shell work consists of reading from stdin and responding to what is found there

    - ♦ **The shell itself, we say,** <u>is a process</u> **or** <u>runs in a process</u>

- ❒ **Each process has an identifier, the** <u>process ID</u> **(**<u>PID</u>**) - an integer assigned by the kernel**

- ❒ **A shell can be a <u>login shell</u> (profile scripts are run), or not (profile scripts are not run)**

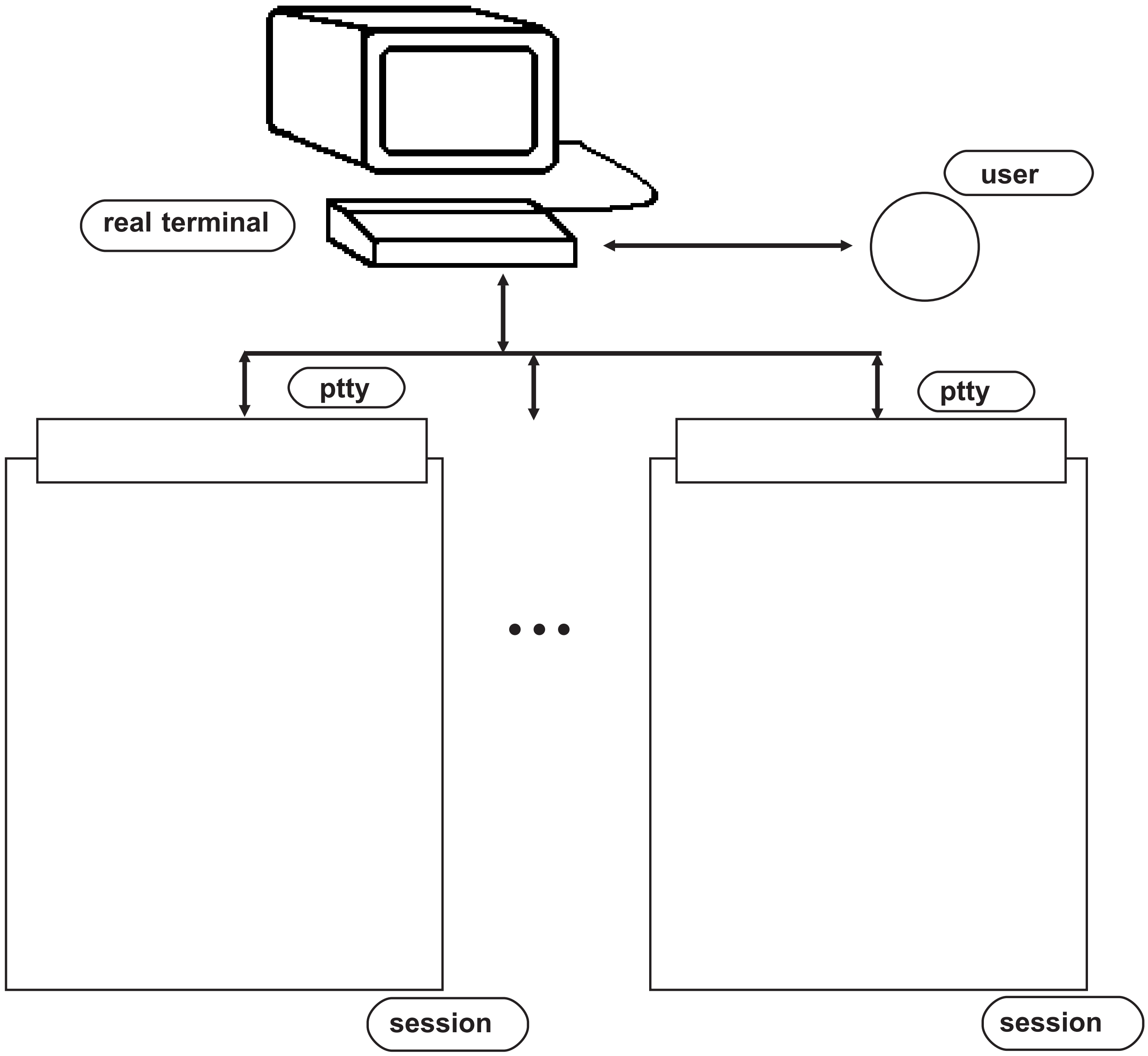
- ❒ **A shell can be an <u>interactive shell</u> (where** stdin **must be a terminal), or not (**stdin **can be a file)**

    - ♦ **A login shell, however, is an interactive shell**

Pseudo-Terminals

❒ **When a user logs into UNIX, a** pseudo-terminal **(**ptty**) is created**

- **The ptty (a sort of logical terminal or conceptual terminal) is used since a real terminal might be running multiple sessions, so each session can be mapped to a separate ptty**



Sessions

❒ **You always start with two sessions: one for the terminal controller (**omvs **or** telnet**) and one for the shell process itself**

♦ **We pretty much ignore the controller session: when the last shell process ends, the controller session is terminated too**

♦ **An** OMVS **user can create multiple shell sessions ...**

✗ As part of the OMVS command (specify the **sessions(***number_of_sessions***)** parameter)

✗ Or by issuing the OMVS subcommand OPEN

✗ This is multiple sessions using a single login

♦ **A** telnet **or** rlogin **user can login multiple times from the same or different terminals**

✗ This is multiple logins with a single shell session each

❒ **In both cases, there is a ptty for each separate shell session**

❒ **So, then: a shell reads from a ptty, looking for commands to process in** stdin**, writing results to** stdout **and / or** stderr

Sessions and Process Groups

- ❒ **A** session **is composed of one or more** process groups

- ❒ **Every process belongs to a process group, and each process group has an identifier, the** process group ID **(**PGID**), which is the same as the PID of the first process in the process group**

  - ♦ **This first process in a process group is called the** process group leader

- ❒ **Generally, each command or script will run in its own process**

  - ♦ **This is called a** child process**, and the process that creates the child is called a** parent process

    - ✗ Every process has a Parent Process ID (PPID) which identifies the process that is its parent

  - ♦ **Each child process may belong to its own process group**

  - ♦ **Note that sometimes a command can run in the same process as its parent process (usually for performance reasons)**

- ❒ **When a script is started, its commands run as separate processes in a single process group**

- ❒ **A new process group can also be created through program calls to kernel services, not discussed in this course**

More Sessions

❒ **A session, then, is a collection of process groups that share a ptty for input and output (**stdin**,** stdout**,** stderr**)**

- **Every process group in the session has the ptty as its** controlling terminal

- **The process that creates the session (usually the login shell) is called the** session leader

- **Once the session leader process ends, no other process in the session can communicate with the ptty (although they may continue to run)**

❒ **Every session has an identifier, the** session ID **(**SID**)**

- **Initially, there is a single session containing a single process group containing a single process**

  ✗ Or, if omvs was invoked with sessions(*n*), there are *n* sessions, each containing a single process group that contains a single process

- **An existing process can invoke** setsid() **(C,C++) or call** BPX1SSI **(other languages) and that process becomes the session leader in a new session**

  ✗ This is what the OMVS OPEN subcommand does

UNIX Jobs

❒ **A UNIX job is a set of processes in the same process group**

- ♦ **A job may run in the** foreground **(interacting with the ptty) or the** background **(running independently of other process groups in your session)**

  ✗ A background job is started by entering a command or script name on the command line followed by whitespace then an ampersand (&)

  ➤ For example: **myscript &**

❒ **When a background job is started, the shell creates a new process in a new process group in the current session and runs the command or script in that process**

- ♦ **Such a process group is called a [UNIX]** job**, and each job is assigned an identifier, the job ID (***job-identifier***)**
- ♦ **This job identifier is displayed for you when you start the job**
- ♦ **The job runs with no intervention, asynchronously to other process groups (you are able to continue working, not needing to wait for a job to finish to enter more commands)**

  ✗ Note that a UNIX job is not the same thing as a z/OS batch job

  ➢ More on UNIX jobs in a later section

Session Hierarchy

❒ **So the organizational hierarchy is**

```
user (uid)
      session   (sid)
            process group       (pgid)
                  process (process group leader, session leader)   (ppid, pid)
                  process                                          (ppid, pid)
                  .
                  .
                  .
            process group       (pgid)
                  process (process group leader)                   (ppid, pid)
                  process                                          (ppid, pid)
                  .
                  .
                  .
      session   (sid)
            process group       (pgid)
                  process (process group leader, session leader)   (ppid, pid)
                  process                                          (ppid, pid)
                  .
                  .
                  .
```

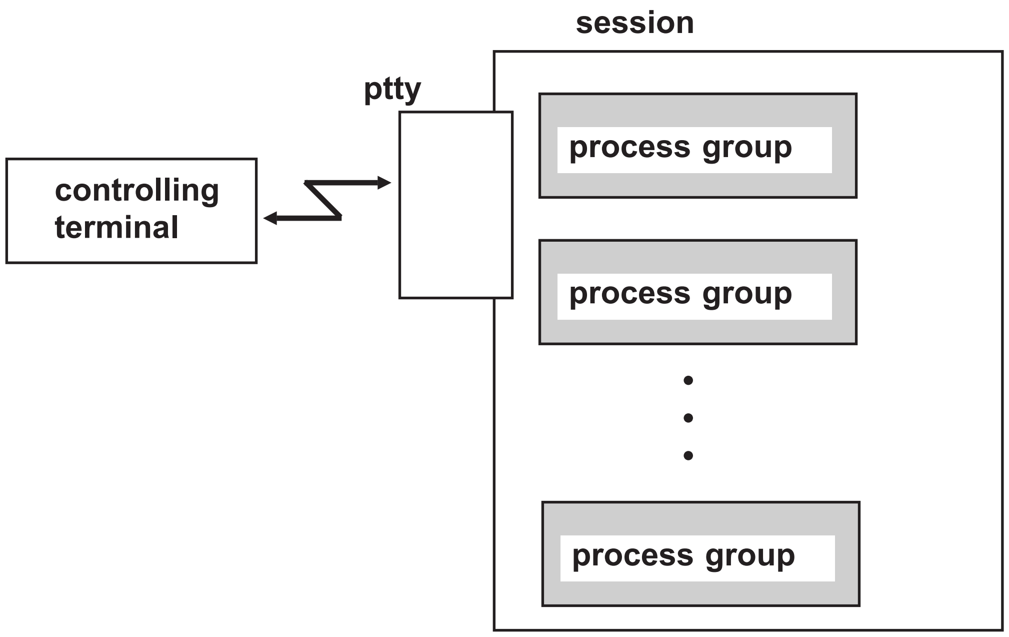
❒ **Finally, there is one more level to be aware of: a** <u>thread</u>

- ♦ **That is, each process is always running one or more threads: the thread is the level of execution control**

- ♦ **Note that in other UNIXes, threads may or may not be present, but in z/OS UNIX each process always has at least one thread**

  - ✗ Of course, each thread has a thread ID (TID)

A Session, pictorially



❒ **One or more of these process groups may be background jobs**

❒ **The significance of all this is one of <u>scoping</u>**

- **For example, the** kill **shell command can signal a job, a process, or all processes in a process group**

  ✗ If a session leader is terminated, no other processes in the session can get to the ptty - but the processes can continue to run

Commands and Processes

- **UNIX uses the term "executable" to mean a command, program, or script**
  - **Traditional z/OS users think of an executable as a load module or program object**
- **When running under a shell, these are the ways to run an executable under that shell:**
  - **Issue a shell** exec **command followed by the name of the executable to run (and optional parameters to pass)**
    - ✗ This causes the executable to replace the shell program, so when the executable completes and exits, the shell and its session ends
  - **Issue an** sh **command or** tsch **command, with flags and options and the name of an executable to run (and optional parms to pass)**
    - ✗ This causes a sub-shell to be created, the executable to be run under the sub-shell, and on exit the sub-shell is removed and the parent shell resumes control
    - ✗ This is unlike a child shell, which runs independent of its parent shell, in a separate process
      - ➢ The sh command is discussed later

Commands and Processes, continued

- **When running under a shell, these are the ways to run an executable under that shell, continued:**

  - **Enter a dot (.) followed by whitespace followed by the name of a script (followed by optional parms)**

    - The script is run under the current shell; this is the only way a script can set environment variables in the current process

      - If you want to run a script in the current directory, and the directory is not in the PATH, you can run using:
        . ./*script_name*

  - **Enter the name of an executable (other than** exec **or the name of a shell program) optionally followed by parameters**

    - If this is a shell built-in command, the command is executed in the current shell environment

    - Otherwise the shell issues a fork request (create a new process that is a copy of the fork-ing shell) followed by an exec request (replace the copy of the shell by the named executable)

      - This is the standard / default processing

      - Note that C programs can issue these kernel requests internally, and other compiled or assembled languages can call BPX1... services to accomplish the same thing, as discussed a few pages ago

    - If the command name contains a slash, the name is assumed to be a pathname, absolute or relative, and the shell searches there; otherwise the search is in the directories specified in the PATH and / or FPATH environment variables (this order can be changed by a shell flag)

Shell Commands: tty

❐ **The** tty **command returns the name of the terminal currently associated with** stdin

Syntax

```
tty
```

♦ **Typically what is returned is this string:**

```
      /dev/ttyp000
or
      ttyp000
```

Shell Commands: sleep

❒ **We introduce the** sleep **command here because it allows us to create a process that lasts long enough for us to examine multiple process and multiple session information**

- ♦ **In production scripts you often need to wait a period of time before [re]trying some work**

❒ **The** sleep **command runs a do-nothing process for some number of seconds**

Syntax

sleep *seconds*

Where *seconds* is expressed as one of

- ♦ **An integer, for example:**
  - ✗ sleep 500
- ♦ **Labeled time, using h for hours, m for minutes, s for seconds, for example (all these are valid):**
  - ✗ sleep 1h10m30s
  - ✗ sleep 20m
  - ✗ sleep 1h30s

Shell Commands: ps

❒ **The** ps **command (process status) provides information on currently running processes and, optionally, threads**

❒ **In the discussion that follows, the underlying assumption is that you will only see information about processes and threads that your UID is authorized to see**

- ♦ **Even when the notes refer to things like "all processes"**

Syntax

ps [-AadefjIm] [-G *idlist*] **[-g** *pid_list*] **[-o** *format_spec*] ...
[-p *pid_list*] **[-s** *sessid_list*] **[-t** *term_list*] **[-{U|u}** *uid_list*]

Where the options work this way

- ✗ **a**, **A**, **e** - display information on processes (**e**: accessible processes; **A**: available processes; **a** - processes associated with terminals); any combination, but **a** overrides **A** and **e**
- ✗ **G**, **g**, **p**, **s**, **t**, **u**, **U** - select processes based on process id(s), session id(s), terminal id(s), and / or session id(s)
- ✗ **f**, **j**, **l**, **o** - specify predefined field lists (**f**, **j**, **l**) to display and user-specified fields to display (**o**); (notice the ellipsis after **-o**; you can specify multiple format items) (**l** is lower-case EL)
- ✗ **d** - display information for all processes except process group leaders

Shell Commands: ps, continued

Syntax, repeated

ps **[-Aadefjlm]** **[-G** *idlist***]** **[-g** *pid_list***]** **[-o** *format_spec***]** ...
[-p *pid_list***]** **[-s** *sessid_list***]** **[-t** *term_list***]** **[-{U|u}** *uid_list***]**

Where, continued

✗ **m** - display thread status information

✗ **G** *idlist* - *idlist* is a list of group IDs; the entries are separated by spaces or commas; the ps command will display information about processes with these real GIDs

example: **ps -G 12, 32**

✗ **g** *pid_list* - *pid_list* is a list of process IDs; the entries are separated by spaces or commas; the ps command will display information about processes with these PIDs

example: **ps -g 92, 134**

✗ **p** *pid_list* - works the same as **g** *pid_list*

✗ **s** *sessid_list* - *sessid_list* is a list of session IDs; the entries are separated by spaces or commas; the ps command will display information about processes with these SIDs

✗ **t** *term_list* - *term_list* is a list of terminal IDs; the entries are separated by spaces or commas; entries are either filenames of the device (*e.g.*: **tty04**) or if the filename begins with **tty**, just the characters after the **tty** (*e.g.:* **04**)

Shell Commands: ps, continued

Syntax, repeated

ps [-Aadefjlm] [-G *idlist***] [-g** *pid_list***] [-o** *format_spec***]** ...
[-p *pid_list***] [-s** *sessid_list***] [-t** *term_list***] [-{U|u}** *uid_list***]**

Where, continued

✗ **U** *uid_list* - *uid_list* is a list of user IDs; the entries are separated by spaces or commas; the entries may be numbers or login names; the ps command will display information about processes with these UIDs

➢ Note: you can specify either case: **U** and **u** are the same

♦ **In z/OS V1.7, a new flag was added: -n** *name***, where** *name* **is the name of the executable file containing the kernel symbol table - then the doc says this feature is not supported(!)**

Shell Commands: ps, continued

Syntax, repeated

ps **[-Aadefjlm]** **[-G** *idlist***]** **[-g** *pid_list***]** **[-o** *format_spec***]** ...
[-p *pid_list***]** **[-s** *sessid_list***]** **[-t** *term_list***]** **[-{U|u}** *uid_list***]**

Where, continued

✗ **o** *format_spec* - which fields to be displayed and the column headers for the fields

Format specifications

♦ **The first line of ps output contains column headings for each status field; each field has a default heading (shown in brackets below)**

♦ **There is a list of available fields to display; simply key in the list of field names you want to see, separated by commas or spaces**

✗ If the entries in the list are separated by spaces, you may need to put the whole string in single quotes

♦ **To override the default column heading for a field, after the fieldname code =***value* **where** *value* **is the string you want to use for the heading**

♦ **The appearance of a user-specified column heading must be the last in any list of fieldnames, and if you want to request additional fields they must be specified with separate -o options**

Shell Commands: ps, continued

- ❐ The fields that can be displayed by the ps command may be grouped into three classes

  - ♦ Process only - values are only meaningful for processes; if the item being displayed is a thread, it will have dashes for a value

  - ♦ Thread only - values are only meaningful for threads; if the item being displayed is a process, it will have dashes for a value

  - ♦ Process and Thread both - the field is meaningful for both processes and threads

- ❐ We discuss each collection of fields separately then give a series of examples

  - ♦ The names are case sensitive

  - ♦ The order you request them is the order the fields will display

  - ♦ Remember the default heading is shown in brackets after the description

Shell Commands: ps, continued

❒ **Fields only meaningful for processes:**

- ✗ **addr** - address of the process; not currently supported; will always have a value of a dash [ADDR]
- ✗ **args** - displays the command that is running, with all its arguments [COMMAND]
- ✗ **atime** - CPU time used by this process since it started; in format: {*days* d *hrs* | *hrs* **h** *min* | *min*:*sec*} [TIME]
- ✗ **attr** - displays process attributes **B** (blocking shudowns), **P** (permanent; survives across shutdowns); **R** (will restart on end; introduced in z/OS 1.8); **T** (tracing is active; z/OS 1.11) [ATTR]
- ✗ **comm** - name of the command running, without its arguments (right padded if necessary) [COMMAND]
- ✗ **etime** - elapsed [wall] time since the process started running in the format [[dd-]hh:]mm:ss [ELAPSED]
- ✗ **gid** - effective group ID of the process [EGID]
- ✗ **group** - effective group ID of the process, as a name, if possible, otherwise as a decimal GID [GROUP]
- ✗ **jobname** - the z/OS jobname [JOBNAME]
- ✗ **nice** - the nice value (priority or urgency) of the process; not currently supported; will always show as a dash [NI]

Shell Commands: ps, continued

❒ **Fields only meaningful for processes, continued:**

- ✗ **pcpu** - percentage of available CPU time this process has taken; not currently supported; will always show as a dash [%CPU]
- ✗ **pgid** - process group id (PGID) in decimal [PGID]
- ✗ **pid** - process id (PID) in decimal [XPID]
- ✗ **ppid** - parent process id (PPID) in decimal [PPID]
- ✗ **pri** - process priority; not currently supported; will always show as a dash [PRI]
- ✗ **rgid** - real group id of the process [GID]
- ✗ **rgroup** - real group id of the process, as a name if possible, otherwise as an integer [RGROUP]
- ✗ **ruid** - real user id of the process [UID]
- ✗ **ruser** - real user id as a name, if possible, otherwise use an integer [RUSER]
- ✗ **sid** - session id of the process [SID]
- ✗ **stime** - start time of the process [STIME]
- ✗ **thdcnt** - total number of threads in the process [THCNT]

Shell Commands: ps, continued

❐ **Fields only meaningful for processes, continued:**

- ✗ **time** - same as etime [TIME]
- ✗ **tty** - name of the controlling terminal, if any [TT]
- ✗ **uid** - effective user id of the process [EUID]
- ✗ **user** - effective user id of the process as a name if possible, otherwise as an integer [USER]
- ✗ **vsz** - amount of virtual storage used by the process as a decimal number of KB [VSZ]
- ✗ **vszlmt64** - maximum virtual storage allowed above the bar [VSZLMT64]
- ✗ **vsz64** - actual virtual storage used above the bar [VSZ64]
- ✗ **wchan** - channel the process is waiting on; not currently supported; will always show as a dash [WCHAN]
- ✗ **xasid** - address space id in hex [ASID]
- ✗ **xpgid** - process group id in hex [XPGID]
- ✗ **xpid** - process id in hex [XPID]
- ✗ **xppid** - parent process id in hex [XPPID]

Shell Commands: ps, continued

❒ **Fields only meaningful for threads:**

- ✗ **lpid** - latch pad waited for [lpid]
- ✗ **lsyscall** - last five syscalls (four character syscalls with no delimiter as a 20 character string) [LASTSYSC]
- ✗ **semnum** - semaphore number thread is waiting on (or dash, if none) [SNUM]
- ✗ **semva**l - semaphore value thread is waiting on (or dash, if none) [SVAL]
- ✗ **sigmask** - signal pending mask in hex [SIGMASK]
- ✗ **syscall** - current syscall [SYSC]
- ✗ **tagdata** - tag assigned to the thread (or dash, if none) [TAGDATA]
- ✗ **wtime** - time thread has been waiting in the format {*days* d *hrs* | *hrs* **h** *min* | *min*:*sec*} [TIME]
- ✗ **xtcbaddr** - tcb address in hex [TCBADDR]
- ✗ **xstid** - low order word of thread id in hex [STID]
- ✗ **xtid** - thread id in hex [TID]

Shell Commands: ps, continued

❒ **Fields meaningful for both processes and threads:**

- ✗ **state** - process / thread state as string of characters [STATE]
- ✗ **flags** - process / thread state as a hex value, where individual bits correspond to specific state values [F]

State value	Bit equivalent (hex)	Meaning
1	00 00 00 10	single task using callable services
A	80 00 00 00	message queue receive wait
B	40 00 00 00	message queue send wait
C	20 00 00 00	communication system kernel wait
D	10 00 00 00	semaphore operation wait
E	08 00 00 00	quiesce frozen
F	04 00 00 00	file system kernel wait
G	02 00 00 00	MVS pause wait
H	01 00 00 00	one or more pthread tasks
J	00 40 00 00	pthread created
K	00 20 00 00	other kernel wait
M	00 08 00 00	multi-thread
N	00 04 00 00	medium weight thread
O	00 02 00 00	asynchronous thread
R	00 00 40 00	running (not kernel wait)
S	00 00 20 00	sleeping
U	00 00 08 00	initial process thread
V	00 00 04 00	thread is detached
W	00 00 02 00	waiting for a child
X	00 00 01 00	creating a new process
Y	00 00 00 80	MVS wait
Z	00 00 00 40	canceled and parent has not performed wait - a Zombie task

Shell Commands: ps, continued

Examples

```
ps   -e   -otty,group,pgid,pid,ppid,sid,comm

ps   -e   -o tty,group,pgid,pid,ppid,sid,comm

ps   -e   -o tty group pgid pid ppid sid comm
```

- Requests the display of a collection of fields for all accessible processes
- Note that all three of these commands are equivalent

```
ps   -e   -otty,group=GroupName   -osid=Session
```

- Requests, for all accessible processes, the name of the controlling terminal, the group id with a column heading of GroupName, and the session id with a column heading of Session
- Note that adding the user header (=GroupName) required that any additional parameters start with a new -o flag

Shell Commands: ps, continued

Special values

- **Earlier (see page 31) we mentioned that flags f, j, and l cause predetermined sets of format options to be used, so here they are:**

  ✗ **-f** is the same as

```
-o ruser=UID -o pid,ppid,pcpu=C -o stime,tty=TTY -o atime,args=CMD
```

  ✗ **-j** is the same as

```
-o pid,sid,pgid=PGRP -o tty=TTY -o atime,args
```

  ✗ **-l** (lower-case EL) is the same as

```
-o flags,state,ruid=UID -o pidppid,pcpu=C -o pri,nice,addr,vsz=SZ
-o wchan,tty=TTY -o atime,comm=CMD
```

  ✗ omitting any format (that is, not specifying any of **o**, **f**, **j**, and **l**) is the same as

```
-o pid,tty=TTY -o atime,comm
```

Shell Commands: uname

❒ **The** uname **command returns information on the current operating system you are running under**

Syntax

```
uname   [-almnrsv]
```

Where

- ✗ **a** - requests all fields
- ✗ **l** (upper case i) - provides system name, release, and version levels relative to z/OS (if omitted this information is provided relative to OS/390)
- ✗ **m** - request machine type
- ✗ **n** - requests node name
- ✗ **r** - requests release number
- ✗ **s** - requests operating system name (default if no operands specified)
- ✗ **v** - requests version number

Sample output from: uname -al:

```
z/OS   S0W1   13.00   01   2094
```

Computer Exercise: Shells, Sessions, and Processes

Get into OMVS - be sure to include sess(3) on the omvs command, to start out with three sessions. Or telnet into the system three times. Or use some combination of OMVS and telnet to get three sessions going.

1. Issue a **tty** command in each session (use the NextSess function key to swap among omvs sessions.); note the differences, if any.

2. Issue a **ps** command; display the address space id in hex, the tty, the process id, the process group id, the parent process id, the session id, the related command (no arguments) being processed, and the jobname, for all accessible processes;

   examine the output; see what processes belong to which session and process groups; note the tty assignments to sessions; Note that you have one session that has two pid's (one for the shell and one for the **ps** command) sharing the ttyp0000, and two other sessions with just the shell running (using ttyp0001 and ttyp0002); also notice that the tty assigned to omvs and telnet displays as a question mark (?);

3. Issue this command string:

   ```
   sleep 45 ; sleep 45 ; sleep 45
   ```

   then switch to a different session and re-issue the **ps** command from 2) above; examine the results, looking for the groupings of sessions, processes, and process groups.

4. Re-issue the sleep command string followed by a space and an ampersand (&); are the results the same when you switch to another session and re-issue the **ps** command from 2)?

5. Issue the **uname** command and find out what version of operating system you're running.

6. As time permits, experiment with other options of the **ps** command, to see real examples of the information available.