Linux on zSeries: Tips, Tools and Tricks

Malcolm Beattie
IBM EMEA Enterprise Server Group

Introduction

- A mixed bag of tips and tricks for users and system administrators of Linux systems
- Not zSeries-specific...but not specific to non-zSeries
- Shell usage: auto-completion, events and script
- Common filters: cut, awk, sort counts, perl -i
- Cron, At, Batch
- Screen
- Filesystems: bind and loop mounts
- Processes and Performance
- Networking: iproute2, ip, tc
- Debugging simple TCP protocols (if time)

Shell auto-completion of filenames

- Shells can auto-complete commands and filenames
- Start typing a filename:

```
$ lpr lah_
```

- Hit <Tab> key: shell completes name if it's unique:

Shell auto-completion of filenames

- Shells can auto-complete commands and filenames
- Start typing a filename:

```
$ lpr lah_
```

- Hit <Tab> key: shell completes name if it's unique:

```
$ lpr lahulpe-2002-tips.ps _
```

Shell auto-completion of filenames

- Shells can auto-complete commands and filenames
- Start typing a filename:

```
$ lpr lah_
```

- Hit `<Tab>` key: shell completes name if it's unique:

```
$ lpr lahulpe-2002-tips.ps _
```

- If ambiguous, the name is completed as far as possible:

`$ lpr myb_` then `<Tab>` key; line becomes:

Shell auto-completion of filenames

- Shells can auto-complete commands and filenames
- Start typing a filename:

```
$ lpr lah_
```

- Hit `<Tab>` key: shell completes name if it's unique:

```
$ lpr lahulpe-2002-tips.ps _
```

- If ambiguous, the name is completed as far as possible:

`$ lpr myb_` then `<Tab>` key; line becomes:

```
$ lpr mybook-chapt_
```

- and shell beeps to warn of ambiguity

Shell completion of ambiguous names

- When shell beeps and stops at an ambiguity, you can list all possibilities by hitting `<Tab>` again (for bash) or `Ctrl/D` (for tcsh).
- Shell then lists all possibilities and re-prompts with partially completed line:

`$ lpr mybook-chapt_` (hit `<Tab>` again)

Shell completion of ambiguous names

- When shell beeps and stops at an ambiguity, you can list all possibilities by hitting `<Tab>` again (for bash) or `Ctrl/D` (for tcsh).
- Shell then lists all possibilities and re-prompts with partially completed line:

```
$ lpr mybook-chapt_  (hit <Tab> again)
mybook-chapt1.ps mybook-chapt2.ps
mybook-chapt3.ps mybook-chapt4.ps
$ lpr mybook-chapt_
```

Shell Events

- *Event designators* let you refer to previous command lines (or parts) via abbreviations starting "`!`".
- The last word of your previous line is "`!$`"

```
$ munge report-2002jan12.txt
```

- You can now print out the report with

```
$ lpr !$
```

- Repeat the most recent command that started "re":

```
$ !re
regenerate foo bar baz
...
```

- Plenty of others exist, some more useful than others

Shell session transcripts

- You can generate a transcript of a shell session:

```
$ script mytrans
Script started, file is mytrans
$ dosomething long and complex
...
$ exit
exit
Script done, file is mytrans
```

- The session transcript is now in file "mytrans"

Common filters

- `cut`
- `awk '{print $n}'`
- `... | sort | uniq -c | sort -nr`
- `perl -i.bak -ple 's/foo/bar/'`

cut

- `cut` selects columns/fields from fixed-column files
- To output characters 1-8 and 51-52 of each line:

```
$ cut -c1-8,51-52
```

- To output field 2 (fields delimited by ":") of each line

```
$ cut -f2 -d:
```

- Field delimiter defaults to \t (tab) when "`-d`" omitted

awk

- `awk` is a "little language" for line-at-a-time filtering
- fancy functions available but basics are useful too
- Basic filter usage is

```
$ awk '(PATTERN) {ACTION} ...'
```

- `awk` reads through stdin once, and splits each line...
- ...into whitespace-separated fields named $1, $2, $3,...
- If the line matches any `PATTERN` specifier...
- ... `awk` performs the associated `ACTION`

awk examples

- A directory listing from "`ls -l`" looks like this:

```
-rw-r--r--    1 fred     users      173346 Jun  6 12:00 foo
-rw-r--r--    1 fred     users       14277 Jun  6 12:00 bar
-rw-r--r--    1 fred     users       21440 Jun  6 12:00 baz
```

- To print out the sizes (in bytes) of those files:

```
$ ls -l | awk '{print $5}'
173346
14277
21440
```

- Note omitted `PATTERN` means "match every line"

awk examples

- A disk usage listing from "`du`" looks like this:

```
$ du
120     ./foo/conf
524     ./foo/extra
1808    ./foo
240     ./bar
```

- To list subdirectories containing over 1000KB:

```
$ du | awk '($1 > 1000)'
1808    ./foo
```

- Note omitted `ACTION` means "print the whole line"

awk examples

- To add up the size in KB of all subdirectories:

```
$ du */ | awk '{s += $1} END {print s}'
99432
```

- Trailing `/` on glob `*/` restricts matches to directory names only
- awk variables ("s" above) do not need to be declared
- special pattern `END` (no parentheses) matches after end of file

Sorted frequency list

- `last` shows recent logins. Its output looks like:

```
alice    pts/9        rhodium.testlan Tue Jun  4 13:54 - 15:03  (01:09)
bob      pts/12       mercury.testlan Tue Jun  4 13:44 - 15:03  (01:19)
alice    pts/8        rhodium.testlan Tue Jun  4 13:43 - 13:52  (00:09)
```

- Who logged in most frequently recently?

```
$ last | awk '{print $1}' | sort | uniq -c | sort -nr
   8492 bob
   5284 alice
   3102 charlie
```

Sorted frequency list

- A line from an Apache access log looks like:

```
10.1.2.1 - - [14/May/2002:16:52:35 +0100] "GET / HTTP/1.1" 200 1763
```

- What are the peak hours for web connections?

```
$ awk '{print $4}' < access_log | cut -c14-15
  | sort | uniq -c | sort -nr
  11765 10
   9342 11
   8723 14
```

perl -i -pe

- Perl has command-line options for in-place edits

```
$ perl -i.bak -pe 's/OLD/NEW/' foo
```

- Perl reads file foo one line at a time...
- ...and for each one substitutes OLD with NEW...
- ...and writes out the resulting line
- The old data ends up in `foo.bak`
- The new data ends up in `foo`

```
open(foo,O_RDONLY);rename(foo,foo.bak);open(foo,O_WRONLY)
```

Cron, At, Batch

- Cron runs jobs every hour/day/third Monday
- It comes with two less-well-known subsystems
  - at
  - batch
- Not in JES' league but sometimes useful

at

- `at` triggers a one-off job at a chosen time

```
$ at 10pm
at> wget http://busy.example.com/foo
at> ^D
job 20 at 2002-06-04 22:00
```

- `at` captures your current directory, shell and environment at submission for running the job
- stdout/stderr from the job is mailed to you

at now

- `at now` can be useful

```
$ at now
at> make all
at> ^D
job 21 at 2002-06-04 14:24
```

- Do something else; review output later at your leisure
- `atq` and `atrm` allow for job list and removal
- `batch` allows (very) basic queue configuration

Screen

- `screen` allows your login session to persist across disconnections/reconnections

```
$ screen
```

- Initialises a new persistent session; clears screen

```
$ long_complex_task
```

...

- Session disconnects (network error, coffee time, ...)

Screen

- `screen` to the rescue...
- Connect again from anywhere

```
$ screen -r
```

- Restores your session from where it left off
- Even display contents are restored
- Multiple sessions and hot-keys supported
- Terminal-based; GUI equivalent would be vnc

Filesystems: loop and bind mounts

- Linux lets you "loopback" mount a filesystem from a file containing a disk image

```
# mount -o loop -t ext2 foo.img /mnt/foo
```

- The option "`-o loop`" sets up a block device behind the scenes (`/dev/loop`*n*) to fetch blocks from `foo.img`
- Also useful when you have a CD image file

```
# mount -o loop -r -t iso9660 cd.img /mnt/cdrom
```

- Good idea to give filesystem type to "`-t`" explicitly
- and to use "`-r`" to mount read-only where appropriate

Filesystems: bind mounts

- Linux 2.4 introduces a powerful namespace feature
- Take part of the existing filesystem namespace...
- ...and mount it on another part of the namespace...
- ...concurrently and fully coherent in both places

```
# mount --bind /lib /opt/dumbd/lib
```

- A chroot to `/opt/dumbd` will have `/lib` available

```
# mount --bind /guestvol/etc /etc
```

- A guest-specific etc directory overmounts the old etc...
- ...even if the root filesystem is mounted readonly...
- ...and, unlike symlinks, current directory remains right

Filesystem: bind mounts

- New Linux tasks can share parent's namespace...
- ...or choose to have their own independent namespace
- Inspired by Plan9, Linux version written by Al Viro
- Per-instance mount flags (e.g. readonly) for 2.5.x
- Allows powerful ways to separate users or daemons whilst sharing necessary parts of the filesystem

Processes and Performance

- fuser and lsof
- /proc/PID/fd and netstat -e
- /proc/PID/maps
- vmstat, iostat, sar
- strace and ltrace

fuser

- fuser lists which processes are currently using a file, mountpoint or network port
- Who currently has `/etc/foo.conf` open?

```
$ fuser -v /etc/foo.conf
                     USER        PID ACCESS COMMAND
/etc/foo.conf        fred       4705 f....  fooprog
```

- Option `-v` shows verbose ps-like list
- ACCESS type: ordinary (f)ile, (c)urrent directory, (e) xecutable, (r)oot directory or (m)mapped file.

fuser

- Who is keeping mountpoint `/opt/bigapp` busy?

```
$ fuser -m /opt/bigapp
/opt/bigapp: 2544 2544c 2602 2602c
```

- Who is connected to local port 22 (ssh)?

```
$ fuser -n tcp 22
```

- Who is connected to remote host 10.1.2.3?

```
$ fuser -n tcp ,10.1.2.3
```

- Full spec is `local_port,remote_host,remote_port`

fuser and lsof

- `fuser` can send a signal to all the processes it finds

```
$ fuser -k ...
```

- `lsof` has similar functionality to `fuser`

/proc/PID/fd

- The `/proc` "pseudo"filesystem presents live kernel status information in the form of files and directories
- `/proc/PID/fd` looks like a directory containing a symlink for each open file descriptor of process PID

```
$ ls -l /proc/1234/fd
lrwx------    1 fred    users          64 Jun 4 15:18 0 -> /dev/pts/10
lrwx------    1 fred    users          64 Jun 4 15:18 1 -> /dev/pts/10
lrwx------    1 fred    users          64 Jun 4 15:18 2 -> /dev/pts/10
lrwx------    1 fred    users          64 Jun 4 15:18 4 -> /var/spool/mail/fred
```

- stdin/stdout/stderr on pseudoterminal 10, descriptor 4 is fred's mail spool file

/proc/PID/fd

- Open files that have been deleted show their original name followed by " (deleted)"
- Sockets are shown in the form "socket:[81240]"
  - 81240 is the "inode number" of the socket.
  - Match socket inode numbers to the conenctions they represent by using the "-e" option to netstat:

```
$ netstat --inet -e
Active Internet connections (w/o servers)
... Local Address           Foreign Address     ... Inode
... mercury.testlan:smtp    foo.example.com:smtp... 81240
```

/proc/PID/maps

- /proc/PID/maps shows the memory map of the address space of process PID

```
$ cat /proc/1234/maps
08048000-080af000 r-xp 00000000 03:05 197058     /usr/bin/mutt
080af000-080b4000 rw-p 00066000 03:05 197058     /usr/bin/mutt
080b4000-080d3000 rwxp 00000000 00:00 0
40000000-40016000 r-xp 00000000 03:01 96804      /lib/ld-2.2.4.so
40016000-40017000 rw-p 00015000 03:01 96804      /lib/ld-2.2.4.so
...
bfff8000-c0000000 rwxp ffff9000 00:00 0
```

- (This map is from Linux/x86 not Linux on S/390)

/proc/PID/maps

- Let's look more closely at one line:

```
08048000-080af000 r-xp 00000000 03:05 197058  /usr/bin/mutt
```

- Each line shows a linear region (vma) of address space
- The line starts with `[start_address, end_address + 1]`
- Then follow the permissions
  - (r)ead, (w)rite, e(x)ecute, as for a file
  - Though Linux tracks "x", some hw architectures ignore it
  - Then (p)rivate as opposed to (s)hared
- Then the file offset at which the mapping occurs
- Then the (hex) major:minor, inode number and filename of the underlying object (0 for anonymous)

vmstat, iostat, sar

- `vmstat` shows current system activity
  - number of processes running/blocked
  - amount of memory free, idle and used as buffers
  - global count of swap in/out and blocks in/out
  - interrupts/sec, context switches/sec
  - percentages of CPU for user/system/idle
- `iostat` shows current activity by I/O devices
  - relies on kernel code not in every vendor kernel
- `sar` collects and displays system activity history

strace

- `strace` shows system calls performed by a process
  - the process may be started fresh ("`strace someprog`")
  - or it may be an existing process ("`strace -p` *`PID`*")
  - multithreaded processes can't currently be traced reliably
- What config files does oddapp read?

```
$ strace -e trace=file oddapp
...
open("/home/fred/.oddapprc", O_RDONLY) = -1 ENOENT (No such file or directory)
open("/opt/oddapp/etc/oddapp.conf", O_RDONLY) = 3
```

strace

- What network connections does oddapp make?

```
$ strace -e trace=connect oddapp
connect(3, {sin_family=AF_INET, sin_port=htons
  (80), sin_addr=inet_addr("10.1.2.3")}}, 16)
  = 0
```

- Do a hex and ASCII dump of all data written to fd 5:

```
$ strace -o fd5.out -e write=5 oddapp
```

- `strace` can follow fork() to child processes, print timestamps of calls and do other useful things too

ltrace

- `ltrace` traces calls to dynamic libraries
- Similar to `strace` except for the executable/library boundary rather than the userland/kernel boundary

```
$ ltrace date
...
time(0xbfffe25c)                                    = 1023711897
localtime(0xbfffe234)                               = 0x401663c0
...
printf("%s\n", "Mon Jun 10 13:24:57 BST 2002") = 29
```

iproute2: ip and tc

- Linux kernels since 2.2.x include advanced routing and traffic control functionality
- kernel functionality is controlled by utilities `ip` and `tc`
- `ip` is a superset of the "old" `ifconfig` and `route` utilities
- `tc` allows advanced traffic control such as CBQ (Class Based Queuing) and other traffic shaping policies
- A PostScript manual `ip-cref.ps` is the only official documentation for `ip` and web searches are required for the more advanced `tc` functionality.

iproute2: ip examples

- `ip` enables capabilities such as:
  - routing by source address and with multiple tables

```
$ ip rule add from 10.0.2.0/24 table 2
```

  - source address selection for chosen routes

```
$ ip route add ... src 10.0.1.7
```

  - multiple default routes with dead gateway detection

```
$ ip route add default nexthop via
  10.90.1.1 nexthop via 10.90.2.1
```

  - aliasing of entire subnets

```
$ ip addr add 10.22.0.0/16 dev lo
```

iproute2: tc

- Traffic control is configured via the `tc` utility
- The full functionality allows arbitrary trees of classes, filters and queue disciplines
- More basic use such as

restrict subnet 10.123.0.0/16

to a bandwidth of 1 Mbps

with/without a hard cap of 2 Mbps

can be done with `tc` via `shapecfg` and `cbq`

- `shapecfg` and similar may not be shipped with all Linux distributions but examples are around if you look hard

Contact details

- Malcolm Beattie
- Linux Technical Consultant
- IBM EMEA Enterprise Server Group
- beattiem@uk.ibm.com
- Malcolm Beattie/UK/IBM@IBMGB

Debugging simple TCP stream protocols

- Many application level protocols are carried over TCP and use plain line-by-line ASCII text
- This makes basic debugging very easy if you know a few of the relevant commands; you don't need a fancy client app
- All you need is a decent straightforward telnet client (any Unix/Linux one should do fine)
- Standard steps are

```
$ telnet targethost portname
Trying 10.1.2.3...
Connected to targethost.
Escape character is '^]'.
Banner line from target host
```

Debugging SMTP connections

- SMTP (Simple Mail Transfer Protocol) is the protocol which carries electronic mail across the Internet
- It is documented in RFC 2821 (a recent update of RFC 821) and uses port name "smtp" (port number 25)

```
$ telnet foo.example.com smtp
220 foo.example.com ESMTP Exim 3.22 #1 Mon, 10
  Jun 2002 14:32:16 +0100
```

- Banner shows host, MTA software and server time

Debugging SMTP connections

- Following the banner, a mail delivery looks like this:

```
HELO me.testlan.example.com
250 foo.example.com Hello me.testlan.example.com [10.0.1.2]
MAIL FROM:<fred@example.com>
250 <fred@example.com> is syntactically correct
RCPT TO:<bob@example.com>
250 <bob@example.com> is syntactically correct
DATA
354 Enter message, ending with "." on a line by itself
Subject: Hello world

Test if this gets delivered
.
250 OK id=17HPPC-0001k9-00
```

Debugging SMTP Connections

- Now you can do a new "MAIL FROM:<...>" or else type "QUIT" to disconnect from the server
- The descriptive output from servers will vary: the 3-digit numbers starting each reply line are the canonical response codes.
- Servers vary in how much sanity checking they do on the "HELO" host address you supply
- Don't forget the <angle brackets> around addresses
- Don't forget to separate mail headers from the mail body with a blank line

Debugging POP connections

- POP (Post Office Protocol) is a protocol used to download mail from a mailstore server (cf. IMAP)
- It is documented in RFC 1939 and uses portname "pop3" (port number 110).

```
$ telnet foo.example.com pop3
+OK Hello there.
USER fred
+OK Password required.
PASS passwordgoeshere
+OK logged in.
```

Debugging POP Connections

- Now that you are logged in, you can list the messages stored on the server:

```
LIST
+OK 3 messages (2485 octets)
1 610
2 710
3 1165
.
```

- You can delete a message like this:

```
DELE 2
+OK message 2 deleted
```

Debugging POP Connections

- You can retrieve an entire message like this:

```
RETR 1
+OK 610 octets follow.
Subject: Cancelled meeting
[and all the rest of the message]
.
```

- Or you can look at just the top 20 (say) lines like this

```
TOP 1 20
+OK headers follow.
Return-Path: <bob@example.com>
Delivered-To: fred@example.com
[rest of the first 20 lines]
.
```

Debugging POP Connections

- You can undo any deletions carried out in the current session like this:

```
RSET
+OK Resurrected.
```

- and you log out like this (which, in the process, actually carries out any deletions you have marked):

```
QUIT
+OK Bye-bye.
```

Debugging IMAP Connections

- IMAP (Internet Message Access Protocol) is a protocol used to manipulate mail held on a server: multiple mailboxes, server-based searching and client-side cachine are supported.
- IMAP4rev1 is documented in RFC 2060 and uses the portname "imap" (port number 143)

```
$ telnet foo.example.com imap
* OK [CAPABILITY IMAP4 IMAP4REV1 AUTH=LOGIN]
  foo.example.com IMAP4rev1 2000.287rh at Mon,
  10 Jun 2002 15:48:28 +0100 (BST)
```

Debugging IMAP Connections

- IMAP is somewhat more complex than other text-based stream protocols
- Each line from the client must be of the form

```
TAG COMMAND ARGUMENTS
```

- The command streams and response streams may be out of synchronisation: response lines are either
  - prefixed with the TAG of the command they relate to
  - or prefixed with "*" for unsolicited responses
- When debugging a test connection you may as well just use the same tag each time: "a", say.

Debugging IMAP Connections

- Log in (with the LOGIN auth method) like this:

```
a LOGIN fred passwordgoeshere
* CAPABILITY IMAP4 .......
a OK LOGIN completed
```

- Before accessing mail, you must select which mailbox to operate on: the magic name INBOX always exists:

```
a SELECT INBOX
* 3 EXISTS
* 2 RECENT
[More unsolicited lines with extra information]
a OK [READ-WRITE] SELECT completed
```

Debugging IMAP Connections

- The FETCH command gives information about chosen parts (header(s), body, MIME parts, ...) of one or more messages
- Here's a FETCH to list brief info on all messages:

```
a FETCH 1:* FAST
* 1 FETCH (FLAGS \Seen) INTERNALDATE "10-Jun-
  2002 15:31:41 +0100" RFC822.SIZE 1383)
* 2 FETCH (FLAGS \Recent) INTERNALDATE "10-
  Jun-2002 16:22:36 +0100" RFC822.SIZE 2385)
a OK FETCH completed
```

Debugging IMAP Connections

- Here's a FETCH to list the headers of message 2:

```
a FETCH 2 RFC822.HEADER
* 2 FETCH (RFC822.HEADER {370}
Subject: Event on Thursday
...

)
a OK FETCH completed
```

Debugging IMAP Connections

- Here's a FETCH to list the body of message 2:

```
a FETCH 2 BODY[TEXT]
* 2 FETCH (BODY[TEXT] {2876}
The details for the Thursday event
...
)
* 2 FETCH (FLAGS (\Recent \Seen))
a OK FETCH completed
```

Debugging IMAP Connections

- Deletion is done by setting the appropriate flag. To delete messages 10 through 20, do this:

```
a STORE 10:20 +FLAGS (\Deleted)
* 10 FETCH (FLAGS (\Seen \Deleted))
...
a OK STORE completed
```

- and in order to commit the deletions, follow up with:

```
a EXPUNGE
```

- To log out from the server, use

```
a LOGOUT
```

Debugging HTTP Connections

- HTTP (Hypertext Transfer Protocol) is a protocol used for the World Wide Web
- It is documented in RFC 2616 (for version 1.1) and uses portname "http" (port number 80).

```
$ telnet www.example.com http
```

- There is no server banner printed for HTTP
- Most web servers allow three different GET methods:
  - GET url (quick but no headers returned)
  - GET url HTTP/1.0 (can't set virtual server name)
  - GET url HTTP/1.1 (needs some extra client headers)

Debugging HTTP Connections

- If you just want to see the contents of a text file:

```
GET /index.html
<HTML><HEAD>
...
</HTML>
Connection closed by foreign host.
```

- As soon as you type the "GET" line, the file is downloaded and the server closes the connection.

Debugging HTTP Connections

- If you want to see header information from the server, then you need to talk real (but minimal) HTTP:

```
GET /index.html HTTP/1.0

HTTP/1.1 200 OK
Date: Mon, 10 Jun 2002 15:52:53 GMT
Last-Modified: Thu, 01 Nov 2001 20:51:45 GMT
Content-Length: 2890
Content-Type: text/html

<HTML>
...
```

- Note you need to hit <Return> twice after your GET

Debugging HTTP Connections

- That second `<Return>` was to signal to the server that you weren't supplying headers of your own.
- If you want to specify a virtual server name, you need to do so in a header and use HTTP 1.1:

```
GET /index.html HTTP/1.1
Host: www.example.com
```

- Now do a header-ending `<Return>`, as before
- You should really specify the full URL in the GET
- HTTP 1.1 defaults to leaving the connection open: use an explicit "`Connection: close`" header to counter it.