mpsh

an experimental command interpreter

Introduction

INTRODUCTION TO MPSH (Version 1.2, 2013-12-17)

mpsh is an experimental command interpreter for Unix systems written by Dave Fischer at the Center for Computational Aesthetics and distributed under the 3-clause BSD license.

"mpsh" stands for "multi-processor shell", because one of its core features is the ability to split certain jobs into multiple processes.

The major features are:

Job Handlers
Multiprocess Jobs
Environment Variable Command Aliases
Set Theory Operators

Theory

"The problem posed in Futurist architecture is not one of linear rearrangement. It is not a question of finding new moldings and frames for windows and doors, of replacing columns, pilasters and corbels with caryatids, flies and frogs. Neither has it anything to do with leaving a façade in bare brick, or plastering it, or facing it with stone or in determining formal differences between the new building and the old one. It is a question of tending the healthy growth of the Futurist house, of constructing it with all the resources of technology and science, satisfying magisterially all the demands of our habits and our spirit, trampling down all that is grotesque and antithetical (tradition, style, aesthetics, proportion), determining new forms, new lines, a new harmony of profiles and volumes, an architecture whose reason for existence can be found solely in the unique conditions of modern life, and in its correspondence with the aesthetic values of our sensibilities. This architecture cannot be subjected to any law of historical continuity. It must be new, just as our state of mind is new." **- Antonio Sant`Elia, 1914**

mpsh was written for my own use. I needed it. If anyone else finds a use for it, that's a nice bonus. It's really an experiment to see if a few shell features I had in mind are actually useful in normal day-to-day usage.

There are a number of mpsh features that I think are genuinely good, and should be adopted by other shells. There are a few features which are nice, but not particularly compelling. And then there are the Set Theory features, which are just insane. I have found uses for them, and hopefully there are a few people out there who will find even better uses for them, but most people will justifiably shake their heads and ignore them.

As Lou said:

"Most of you won't like this and I don't blame you at all. It's not meant for you."

The primary feature that I wrote mpsh to experiment with is the job handler.

The general idea behind the job handler feature was inspired directly by VMS's DCL. In DCL, many things that seem to the user to be related, but are completely unrelated in how they run internal to the system, are presented to the user in a consistant fashion. For example, these DCL commands obviously have nothing to do with each other, as they are actually carried out by the system:

```
$ SET HOST GAMMA
$ SET TERMINAL/VT100
$ SET PASSWORD
$ SET DEFAULT SYS$LOGIN:
```

In the Unix world, those commands roughly translate as:

```
$ telnet gamma
$ TERM=vt100
$ passwd
$ cd $HOME
```

Not that I want to change *those* commands to be similar, but I do think it's a bad thing that the following are so different:

```
$ a.out &
$ echo a.out | submit
$ rsh gamma a.out
```

Traditionally in Unix, depending on what you want to do, you execute a command by typing it directly, by using it as an argument to another command (ex: rsh), or by using it as stdin to another command (ex: at). mpsh lets you use the same basic syntax for all of those methods, with an option at the end of the command to specify how the command is to be handled. So you think about the task you want acomplished, and the method is an afterthought.

Depending on how you've configured things, the above three commands could be:

```
mpsh$ a.out &
mpsh$ a.out &s
mpsh$ a.out &g
```

In terms of writing mpsh, my mantra throughout development, whenever I ran into a difficult problem, has been:

A lisp programmer could do this, therefore so can I.

I tried very hard to keep both the design and the implementation clean and simple. Frequently problems have been overcome simply by repeating the mantra, stepping back from the problem, and trying to find a more elegant path forward.

As mpsh is written entirely in K&R C, obviously I consider good software design a matter of attitude and discipline, rather than a question of sophisticated language features.

The primary mpsh development machine is a Sun E3500 running Solaris 8. (In 2013. Really.) Porting is done on an Origin-2000, a Raspberry Pi, and an assortment of laptops.

Job Handlers

Job handlers allow mpsh to send a command to some external program to handle instead of executing it directly. For example: sending jobs to batch queues, execution on remote nodes, delayed execution via /bin/at, run a command in a new x-term window, etc. All job handlers are user configured. The handler is specified by a single letter, which is then the argument after "&". The handler can be given arguments as well. Here are some examples:

Configure job handler "q" to submit job to a batch queue, via the command "batch-submit". When a command is run with the "&q" option, batch-submit will be executed, with the text of that command as stdin. batch-submit can also be given arguments, for example "40" (presumably a batch queue priority level) in this example:

```
mpsh$ setenv handler-q="!batch-submit"
mpsh$ command blah blah &q 40
```

Submit a job to "at" to run at a later time:

```
mpsh$ setenv handler-t="!at"
mpsh$ command blah blah &t 6:00 pm
```

(See more examples in mpshrc_all)

Handing the command off to the job handler is **not** done in the background (unless the job handler has a '&' at the end of the line), and once it's accomplished, it is not listed as a running job.

Multiprocess Jobs

Multiprocess Jobs - split up a command into multiple parallel instances.

In the basic form, this lets you split up a command with many arguments into [n] separate jobs, each with 1/n of the argument list.

So, this command:

```
mpsh$ command arg1 arg2 arg3 arg4 &2
```

will execute two seperate processes:

```
command arg1 arg3
command arg2 arg4
```

Execute a separate process per jpg file:

```
mpsh$ gen-thumbnails *.jpg &*
```

Execute a separate process per file, and submit each to a batch queue:

```
mpsh$ gen-thumbnails *.jpg &q*
```

In the "symmetric" form "&n!", you can generate parallel instances regardless of the argument list, with the entire argument list duplicated for each process:

```
mpsh$ command arg1 arg2 arg3 arg4 &2!
```

will execute two processes, each with the full arguments:

```
command arg1 arg2 arg3 arg4
command arg1 arg2 arg3 arg4
```

("&*!" will still determine the number of instances to execute based on the argument list, but each instance will get all of the arguments.)

When executing multiprocess jobs, the "jobs" command will show how many processes the job was split into, and how many are still running. Also, job PID substitution ("kill -9 %gen") will include the PIDs for all processes that are still running.

If you want to use a multiprocess job in a pipeline, you have to use command grouping:

```
mpsh$ command | [ command arg &n ] | command
```

Environment Variables

Environment variables are set with the **setenv** command:

```
mpsh$ setenv TERM=vt100
```

Displayed with the -s option:

```
mpsh$ setenv -s
```

And deleted with the -d option:

```
mpsh$ setenv -d name
```

(This works for any environment variables: command aliases, mpsh settings, etc.)

Environment Variable Command Aliases

An environment variable can be set to a command alias, which is then executed every time the environment variable is evaluated. This alleviates the need for certain "magic" environment variables, and allows for interesting new experiments.

For example - $CWD will expand to the directory that is current at the time it is expanded, not at the time it was set:

```
mpsh$ setenv CWD="!pwd"
```

Read the X Windows cut buffer (adjust for your computer setup):

```
mpsh$ setenv x="!rsh desktop xcb -p 0 -display $DISPLAY"
```

Display the date in the prompt string:

```
mpsh$ setenv mpsh-prompt="!echo (date '+%l:%M') 'mpsh% '"
```

Command alias variables can be displayed with the -c option:

```
mpsh$ setenv -c
```

Environment Variable Internal Settings

Environment variables are also used to control internal mpsh settings. True/False settings are set to 1 or 0.

The internal settings are:

mpsh-version - version string
mpsh-prompt - command prompt
mpsh-history - Show command history substitution?
mpsh-cdhistory - Show directory history substitution?
mpsh-hist-disp - History display format
mpsh-hist-disp-l - History "-l" display format
mpsh-umask - umask, in octal
mpsh-nice - nice value for "&-[-]"

If you set mpsh-nice=5, then "command &-" will run at nice level 5, "command &---" at nice level 15, etc. The default mpsh-nice value is 10.

mpsh-error-level - error verbosity level 0 - 3

Error messages are formatted:

0: none
1: error
2: error [string]
3: error [string] errno-string

For example, the same error with different error level settings:

```
mpsh$ setenv mpsh-error-level=1
mpsh$ date > /sdfasdf
mpsh: Error redirecting stdout
mpsh$ setenv mpsh-error-level=3
mpsh$ date > /sdfasdf
mpsh: Error redirecting stdout [/sdfasdf] Permission denied
```

Internal setting variables do not get passed to child processes, but will expand on the command line:

```
mpsh$ echo $mpsh-version
```

Show mpsh internal settings:

```
mpsh$ setenv -i
```

Command Substitution

Command substitution is specified by parenthesis:

```
mpsh$ echo (date) (command | sort | etc (etc))
```

This is about the only time mpsh isn't extremely demanding of whitespace.

Command Grouping

Command grouping is accomplished in mpsh with square brackets. The sequence between the brackets is executed in a sub-shell.

```
mpsh$ [ cd src ; tar cf - . ] | [ cd /backup/src ; tar xf - ]
```

Set Theory Operations

mpsh can perform various set theory operations on the output of programs. Seperate lines of the text are parsed as set elements. For instance: the intersection of the output of two commands provides the lines that appear in the output of both commands:

```
mpsh$ ls dir1 |^ ls dir2
```

(Bear in mind that this follows set theory, not simple text filtering, so duplicates and original order are not maintained.)

You can enter sets directly using curly braces (but with spaces instead of commas):

```
mpsh$ { red green blue } |^ { fast red { dog cat gorilla } }

mpsh$ ls |- { *.o }
```

The basic set theory operations are:

Union: |**U**
Intersection: |**^**
Difference: |\
Synonym for difference: |**-**
Symmetric difference: |**o**

You can also execute commands conditionally based on set theory tests:

```
mpsh$ command |[<=>] command ; conditional-command
```

If A is a subset of B: |**<**
If B is a subset of A: |**>**
If A equals B: |**=**

```
mpsh$ { red } |< { red green blue } ; echo red is a subset of rgb.
red is a subset of rgb.
```

Some more examples - take the intersection of *.c with the union of the outputs of command1 and command2, then send that to the sort command:

```
mpsh$ command1 |U command2 |^ { *.c } | sort
```

Open gimp with all the jpg's in the current directory, except *t.jpg:

```
mpsh$ gimp ( { *.jpg } |- { *t.jpg } )
```

The set theory operators are a bit superficial. Sets are deduplicated but subsets aren't, etc. Does not support infinite sets.

IO Redirection

IO redirection is the same for pipes and files. The options are:

e - stderr
b - both (stdout & stderr)
a - append

```
mpsh$ command --help |e more
mpsh$ command >a data.out >e err.out
```

Because of these options, whitespace is essential. The following will NOT be parsed as intended:

```
mpsh$ command >filename
```

The whitespace before the filename is required:

```
mpsh$ command > filename
```

Conditionals

Conditionals are specified as an option after the command seperator:

```
mpsh$ true ;? echo true
mpsh$ false ;! echo false
```

See also the Set Theory section, for set theory conditionals.

Command History

Command history works about the same as in most Unix shells. Repeat previous command, by name, by number, or most recent. "pattern" will match the beginning of a command, while "*pattern" will match anywhere in the string.

```
mpsh$ !command
mpsh$ !*arg
mpsh$ ![n]
mpsh$ !
```

You can also retrieve history information with: **!command.field** where field is one of: text, dir, or num (for the text of the command, the directory it executed in, or the history number). For example:

Go to the directory that you last used vi in:

```
mpsh$ cd (!vi.dir)
```

Look at the detailed history entry for the last invocation of gcc:

```
mpsh$ history (!gcc.num)
```

Subsitute "blue" for "red" in the last "echo" command, and then execute it.

```
mpsh$ (!echo.text | sed s/red/blue/)
```

The **history** command displays command history entries. You can display the short form, long form, user-specified format, or full details for one entry:

```
mpsh$ history
mpsh$ history -l
mpsh$ history format-string
mpsh$ history [n]
```

The fields displayed by the history command are controlled by format strings, one letter for each field. The simple "history" display is controlled by the **mpsh-hist-disp** environment variable, and the long format is controlled by the **mpsh-hist-disp-l** environment variable. You can also specify a new format as the argument to history. For example, to display just the commands, and the directories each one executed in, use:

```
mpsh$ history dc
```

The fields available via the format strings are:

c - Command
C - Command, 20 characters
n - Number
u - CPU user time
s - CPU system time
e - Elapsed time
x - Exit status
d - Directory
D - Directory, 20 characters

(CPU usage is currently only displayed in Solaris & BSD, because Linux's waitid() system call doesn't bother to fill in all the data in the struct it returns.)

Clear history with the -c option:

```
mpsh$ history -c
```

Directory History

Directory history has similar syntax to command history, except a string match is from anywhere in a history entry, not the start of the line.

```
mpsh$ cd !str
mpsh$ cd ![n]
mpsh$ cd !
```

Example: go to the last directory which had "src" somewhere in the path:

```
mpsh$ cd !src
```

Show directory history:

```
mpsh$ cd -s
```

Clear directory history:

```
mpsh$ cd -c
```

You can also change directory to the directory of a command in command history, via command history fields:

```
mpsh$ cd (!vi.dir)
```

See command history section for details.

List of Builtin Commands

All internal mpsh commands:

.

Execute script in the current shell:

```
mpsh$ . file
```

cd

Change directory:

```
mpsh$ cd
mpsh$ cd dir
mpsh$ cd !str
mpsh$ cd !n
mpsh$ cd !
```

Show directory history:

```
mpsh$ cd -s
```

Clear directory history:

```
mpsh$ cd -c
```

exit

```
mpsh$ exit [n]
```

fg

Resume a stopped job:

```
mpsh$ fg [pid]
mpsh$ fg %job
mpsh$ fg %-
```

history

Show history:

```
mpsh$ history
mpsh$ history -l
mpsh$ history n
```

Clear history:

```
mpsh$ history -c
```

jobs

Show jobs:

```
mpsh$ jobs
```

setenv

Set environment variable:

```
mpsh$ setenv name=val
```

Show environment variables:

```
mpsh$ setenv -s
```

Show environment variable command aliases:

```
mpsh$ setenv -c
```

Show mpsh internal settings:

```
mpsh$ setenv -i
```

Delete environment variable:

```
mpsh$ setenv -d [name]
```

wait

```
mpsh$ wait [pid]
```

Quoting

Quoting in mpsh works with alternating single/double quotes, as in literary usage:

```
mpsh$ command "1 1 1 1 '2 2 2 "3 3 3" 2 2 "3 3 '4'" 2 2' 1 1"
```

This doesn't currently do much aside from grouping the quoted text into one argument. It prevents command substitution, pipes, and IO redirection from being parsed, but not environment variable expansion, filename globbing, or job pid substitution.

Tab Completion

Tab-completion currently exists in mpsh in a very primitive form. mpsh uses gnu readline, but doesn't talk to it, so behaviour is for default filename completion, with no substitutions. ie: $HOME/blah<tab> won't work, because $HOME isn't evaluated until the entire line is entered.

Also, command tab-completion doesn't work as one would hope. However, filename globbing for command names is treated specially, inspired by the non-interactive command expansion of Multics, NOS, & VMS. So this:

```
mpsh$ do-some*ng args
```

Will expand to the first match in the directories listed in $PATH.

Startup Files

Startup files are:

$HOME/.mpshrc_all - read by all mpsh instances
$HOME/.mpshrc_login - read by mpsh logins

Differences From sh/bash

Standard or slightly modified features, with slightly different syntax:

The main thing to remember is that since mpsh has options after many special characters (pipe, background, redirect, etc.) it requires whitespace between all arguments. All of the whitespace in the following command is absolutely required:

```
mpsh$ comm arg arg | sort -nr > /dev/null &
```

Job PID substitution (%[n], %[string], %%, %-) is generalized. ie: "echo %vi" works.

Repeat last command / cd to last directory is "!", not "!!".

Command substitution is done with parentheses, not back-quotes.

Curly braces {...} are used for set theory, not command grouping.

Square brackets [...] are used for command grouping.

Quoting is weird, see quoting section.

mpsh automatically collects status data on job exit, which is accessable via "history -l" or "history [n]". CPU time, etc.

Doesn't currently convert ~ to $HOME.

Doesn't do stdin redirects yet.

"pwd" and "cd" follow the real path, not the path following symbolic links. Which is normal at the *nix low level, but is counterintuitive to the user.

Startup File

```
# mpshrc_all
#
# runs for any instance of mpsh


# job handlers:
setenv handler-t="!at"
setenv handler-q="!batch-submit"
setenv handler-w="!xterm -e (cat) &"

# These use `cat` because the remote shell is sh or bash.
setenv handler-a="!rsh a31 cd (pwd) ';' `cat`"
setenv handler-p="!rsh pi0 cd (pwd) ';' `cat`"


# Here is a "misuse" of the handler feature. The "command text"
# doesn't actually have to be a command...
# Evaluate the "command" as a dc calculation:
setenv handler-d="!dc"
# Send the "command" to the X Windows buffer:
setenv handler-b="!rsh desktop xcb -s 0 -display $DISPLAY"

# alias env variables:
# (second versions includes substitutions)
# This gives behaviour similar to bash's "!$" with "$!".
# Which is confusing, admittedly.
#setenv !="!history | tail -1 | sed 's/.* //'"
setenv !="!echo (history | tail -1 | sed 's/.* //')"

# $x reads the X Windows buffer.
setenv x="!rsh desktop xcb -p 0 -display $DISPLAY"

setenv YY="!date +%Y-%m-%d"
setenv CWD="!pwd"
setenv HOST="!hostname"


# Internal mpsh settings:
# Show history & cd history substitutions
setenv mpsh-history=1
setenv mpsh-cdhistory=1
# Max error reporting
setenv mpsh-error-level=3

# Adjust history display formats as desired:
setenv mpsh-hist-disp=nc
setenv mpsh-hist-disp-l=ndxuec

# prompt
setenv mpsh-prompt="!echo (date '+%l:%M') $HOST 'mpsh% '"
```

Login Startup File

```
# mpsh login script


setenv PATH=/bin:/usr/bin:/usr/local/bin:
setenv mpsh-umask=022


echo Welcome to (hostname)
echo
cat /etc/motd
echo
```