

RJBS

Antediluvian Unix

a guide to unix fundamentals

Why is UNIX so hard?

- UNIX is made of tiny little pieces, all alike.
- They can be put together in many ways.
- The pieces have manuals, but UNIX doesn't.
- The Mysteries of UNIX were handed down in an oral tradition.
- Then the GNU/Linux flood happened.

The Deluge

- Users adopted Linux as their first unix, learning everything from HOWTOs.
- The GNU system combined toolkit pieces.
- Distributions with X11 out of the box further obfuscated the shell.

We're going to talk about antediluvian UNIX.

...but we're going to use modern tools.

Because being a fundamentalist does not mean being a relic.

UNIX at Rest

the unix filesystem

The Time Before UNIX

- Every file had a structure: a list of similar records.

The Age of UNIX

- Forcing a record structure on every file was a hassle.
- Plain text was easier to read and edit without special tools.
- So records were stored in the natural plain text way: columns and lines.
- Every file is just a bytestream.

The Flat File

```
127.0.0.1      localhost
66.92.232.136 cheshirecat
66.92.232.135 redking
192.168.200.1  airport
```

FS = space

RS = newline

The Flat File

```
root:x:0:0::/root:/bin/sh
rjbs:x:101:100:RJBS:/home/rjbs:/bin/zsh
```

FS = colon

RS = newline

Special Files

- Files that contain data are plain files.
- Other (special) files perform system magic.
 - directories
 - symbolic links
 - named pipes
 - sockets
 - devices

Permissions

- Every file has associated permissions.
- Permissions govern reading, writing, and execution.
- And then there are the special permissions: `suid` and `sticky`.



Magic Numbers

- UNIX supports many executable file types.
- Roughly 65,535, at the minimum.
- Magic numbers identify the file type.
- They're the first two bytes of the file.

The Magic Magic Number

- One magic number, 0x2321, is common in modern unices.
- In ASCII, it's #!
- In English, it's “shebang!”
- The kernel will run the program named after #! with the given args, plus the filename.

#! in Action

~/bin/hello

```
#!/bin/sh  
echo Good morning, $USER!
```

knave!rjbs% hello

```
execve(  
    "/bin/sh",  
    "/home/rjbs/hello"  
);
```

#! in Action

~/bin/hi_all

```
#!/usr/bin/awk -f
BEGIN { FS=":"; } {}
($7 != "") && ($5 != "") {
{print "Hello, " $1 "!";}
END { print "Buh-bye!" }
```

#! in Action

~/bin/hi_all

```
#!/usr/bin/awk -f
BEGIN { FS=":"; } {}
($7 != "") && ($5 != "") {
{print "Hello, " $1 "!";}
END { print "Buh-bye!" }
```

#! in Action

~/bin/hi_all

```
#!/usr/bin/awk -f
BEGIN { FS=":"; } {}
($7 != "") && ($5 != "") {
{print "Hello, " $1 "!";}
END { print "Buh-bye!" }
```

knave!rjbs% hi_all /etc/passwd

Hello, sync!
Hello, shutdown!
Hello, halt!
Hello, operator!
Hello, gdm!
Hello, samael!
Hello, rjbs!
Hello, calliope!
Hello, solios!
Hello, photon!
Buh-bye!

UNIX in Motion

unix program execution

The Process Tree

- Every running process is the child of the process that ran it (except init).
- A child can't live without its parent.
- A child starts life with a copy of its parent's environment.

Environment

- An environment is a set of data communicated from parent to child.
- Parents teach children, but children can never teach parents.
- This is important.
- Really important.

```
init -+  
      |-zsh -irssi
```

```
MAIL=/var/spool/mail/rjbs  
MAILCHECK=60  
IRCSERVER=us.slashnet.org  
IRCNICK=r$ðñnes  
LESS=-M
```

```
knave!rjbs% IRCNICK=rsignes irssi
```

Signals

- Processes mind their own business and try to run to completion.
- But they can be sent signals to make them do unusual things.
- Mostly, “stop running.”

Some Common Signals

| | | | |
|---------|---------|---------|---------|
| SIGHUP | SIGINT | SIGQUIT | SIGILL |
| SIGTRAP | SIGABRT | SIGBUS | SIGKILL |
| SIGSEGV | SIGTERM | SIGSTOP | SIGPWR |

The Time Before UNIX

- Your computer ran one program.
- When that program was done, it ran the next.
- Lather, rinse, repeat.
- Batch processing.

The Age of UNIX

- Many programs run at once.
- Interactive programs became possible.
- ...but batch processing is not dead.

Job Control

- A system for handling batch processing in UNIX.
- Jobs can be
 - paused
 - run in the background
 - brought to the foreground

Job Control

```
knave!rjbs%
```

Job Control

```
knave!rjbs% jobs
```

```
[1]+  Running
```

```
make bzImage&
```

```
[2]   Running
```

```
backup&
```

```
[3]-  Suspended
```

```
wget -rnp http...
```

```
[4]   Running
```

```
pdflatex thesis.tex
```

```
knave!rjbs%
```

Job Control

```
knave!rjbs% jobs
```

```
[1]+  Running
```

```
make bzImage&
```

```
[2]   Running
```

```
backup&
```

```
[3]-  Suspended
```

```
wget -rnp http...
```

```
[4]   Running
```

```
pdflatex thesis.tex
```

```
knave!rjbs% mutt
```

Job Control

```
knave!rjbs% jobs
```

| | | |
|------|-----------|---------------------|
| [1]+ | Running | make bzImage& |
| [2] | Running | backup& |
| [3]- | Suspended | wget -rnp http... |
| [4] | Running | pdflatex thesis.tex |

```
knave!rjbs% mutt
```

| | | |
|-----|------|---------------------|
| [2] | Done | backup& |
| [4] | Done | pdflatex thesis.tex |

Job Control in Action

~/bin/waiter

```
#!/bin/sh
for i in 1 2 3 4 5 6 7 8 9 10
do sleep $i
done
```

How long does this take to run?

Job Control in Action

~/bin/waiter

```
#!/bin/sh
for i in 1 2 3 4 5 6 7 8 9 10
do (sleep $i)&
done
wait
```

How long does this take to run?

Job Control

- &
- CTRL-Z
- jobs
- bg and fg
- wait

Bytestreams

- Files, you remember, are just bytestreams.
- A bytearray is list of bytes, in order.
- But it's a stream: it flows
 - in and out of programs
 - through pipes

Bytestreams

- Programs read and write from bytestreams all the time.
- But a UNIX program's favorite bytestreams are the standard IO streams.
 - stdin - standard input
 - stdout - standard output (1)
 - stderr - standard err (2)

Standard I/O

- The standard I/O streams are usually connected to the terminal.
- But you can redirect them.
- Understanding I/O redirection is fundamental to using UNIX effectively.

Redirection in Action

~/bin/savalias

```
#!/bin/sh
alias > .bash_tmp_1
cat .bash_alias .bash_tmp1 \
    > .bash_tmp_2
sort < .bash_tmp_2 > .bash_tmp_1
uniq < .bash_tmp_1 > .bash_tmp_2
mv .bash_tmp_2 .bash_alias
rm .bash_tmp_1
```

Redirection in Action

~/bin/savalias

```
#!/bin/sh
alias \
| cat - .bash_alias \
| sort
| uniq > .bash_alias
```

Redirection in Action

~/bin/connected

```
#!/bin/sh
netstat -n \
| grep ESTABLISHED \
| cut -c 45-65 \
| awk 'BEGIN{FS=":"}{print $1}' \
| sort -n \
| uniq
```

Redirection

- >

- >>

- <

- |

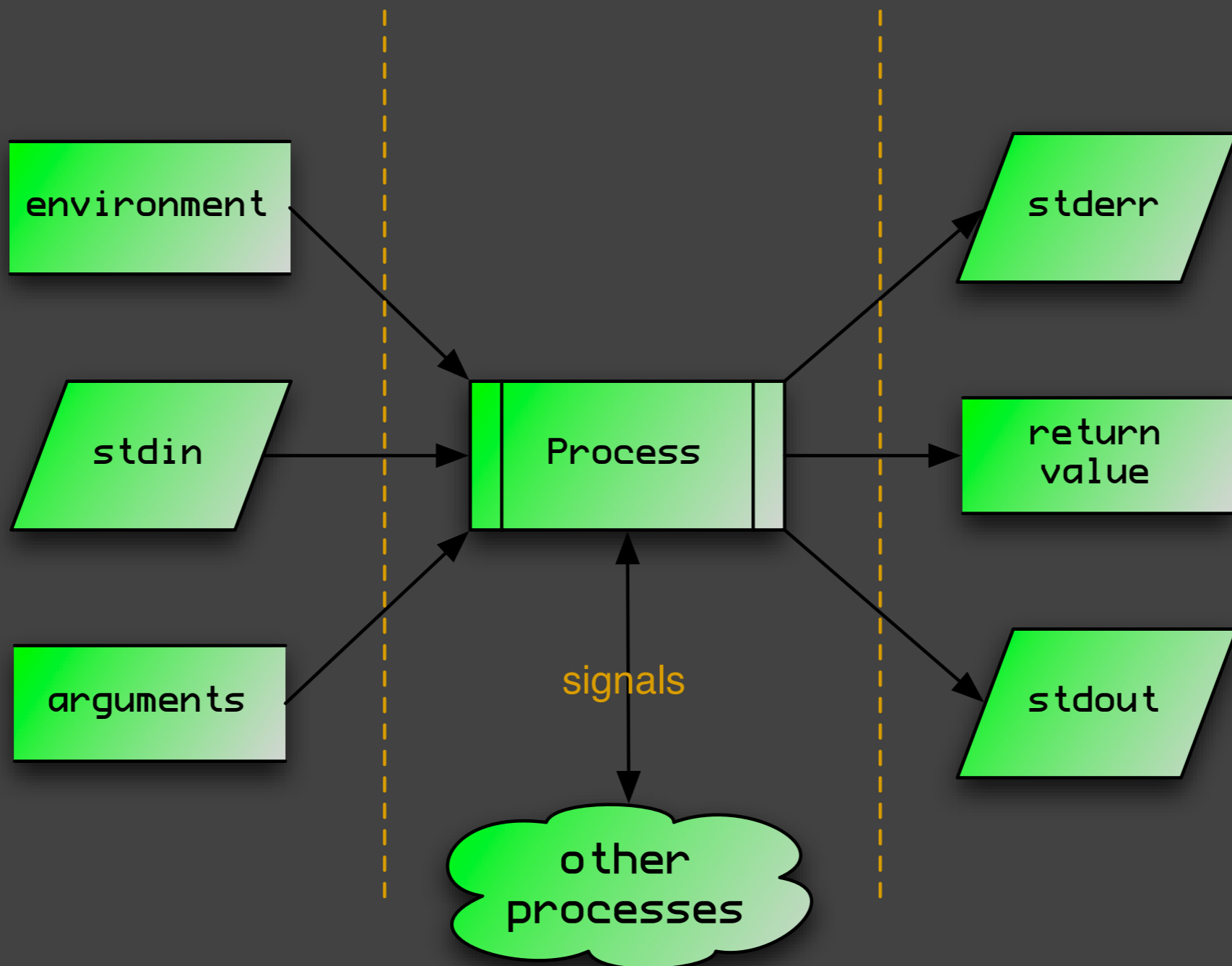
- \ or \$()

Arguments

- Programs are passed a list of arguments, given on the command line.
- These are usually switches that change how the program will work.
- In lieu of stdio, arguments may name files on which to operate.

Return Values

- When a job finishes, its status is reported by a single integer.
- This “return value” indicates success, failure, or other information.



The UNIX Toolkit

elementary programs

The Toolkit

- Little utilities.
- Found almost everywhere.
- Work the same almost everywhere.

Some Toolkit Programs

cd

ls

cp

mv

rm

mkdir

chmod

chown

sort

uniq

cat

test

basename

cut

tr

grep

seq

yes

true

false

The Toolkit in Action

/sbin/armageddon

```
#!/bin/sh  
cat /etc/passwd
```

The Toolkit in Action

/sbin/armageddon

```
#!/bin/sh  
cat /etc/passwd \  
| grep :100:
```

The Toolkit in Action

/sbin/armageddon

```
#!/bin/sh
cat /etc/passwd \
| grep :100: \
| cut -d: -f6
```

The Toolkit in Action

/sbin/armageddon

```
#!/bin/sh
rm -R $(cat /etc/passwd \
| grep :100: \
| cut -d: -f6)
```

The Toolkit in Action

/sbin/armageddon

```
#!/bin/sh
yes | rm -R $(cat /etc/passwd \
| grep :100: \
| cut -d: -f6) \

echo Users obliterated!
```

The Shell

- Just another piece of the toolkit.
- But the first among equals.
- Provides flow control for processes.
- Also, built-in utilities.
 - ...but most of these are extraneous.

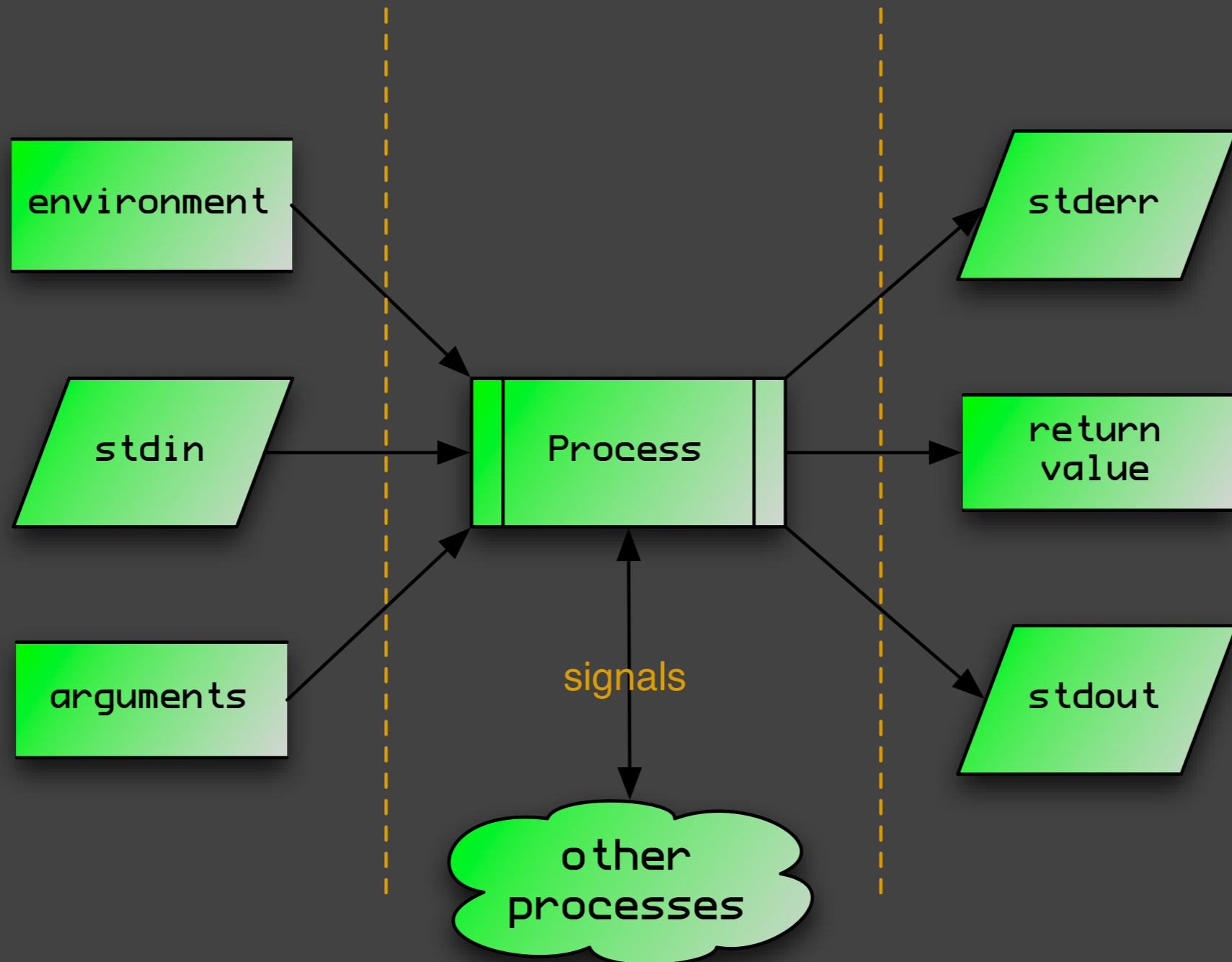
Flow Control

- Controls:
 - whether a process will run
 - when a process will run
 - process iteration over a list of data

Flow Control

```
#!/bin/sh
if [ $UID = 0 ]; then
    echo You're root!
else
    echo You're a luser!
fi
```

So, how does *sh* compare *\$UID* and zero?



if

```
#!/bin/sh
if test $UID = 0; then
    echo You're root!
else
    echo You're a luser!
fi
```

if

```
#!/bin/sh
if cp @$ $HOME; then
    echo Everything's fine!
else
    echo WE'RE ALL GONNA DIE!
fi
```

for

~/bin/makeall

```
#!/bin/sh
for dir in ~/code/*;
do cd $dir; make
done
```

for and if

~/bin/makeall

```
#!/bin/sh
for dir in ~/code/*;
do \
  if [ -d $dir -a -r $dir/Makefile ]; then
    cd $dir; make
  fi
done
```

for, if, and job control

~/bin/makeall

```
#!/bin/sh
for dir in ~/code/*;
do \
  if [ -d $dir -a -r $dir/Makefile ]; then
    (cd $dir; make)&
  fi
done
wait
```

while

~/bin/unique

```
#!/bin/sh
echo $$ > ~/unique_pid
while [ $$ = $(cat ~/unique_pid) ]
do sleep 1
done
echo Replaced by $(cat ~/unique_pid)
```

until

~/bin/unique

```
#!/bin/sh
echo $$ > ~/unique_pid
until [ $$ != $(cat ~/unique_pid) ];
do sleep 1;
done;
echo Replaced by $(cat ~/unique_pid)
```

while and if

~/bin/rmcont

```
#!/bin/sh
if [ -r ~/unique_pid ]; then
    kill $(cat ~/unique_pid)
    rm ~/unique_pid
fi
echo $$ > ~/unique_pid
while true
do rm -R $1/* > /dev/null
done
```

select

~/bin/chat

```
#!/bin/sh
select user in $(who|cut -f1 -d" "|uniq)
do
    if [ $user ]; then talk $user; fi
    break
done
```

```
1) rjbs
2) calliope
#?
```

case

~/bin/unarc

```
#!/bin/sh
case "$1" in
    *.tgz|*.tar.gz) tar zxvf $1;;
    *.tar.bz2)      tar jxvf $1;;
    *.gz)           gunzip $1;;
    *.tar)          tar xvf $1;;
    *.zip)          unzip $1;;
    *.shar.gz)      cat $1 | gunzip | unshar > \
                    $(basename $1 .shar.gz) ;;
    *) echo Unknown archive type!
esac
```

Redefining Your UNIX

customizing the interactive experience

Why script?

- Any shell command can be entered directly at the command line.
- So why script?
- Scripting lets you remember useful recipes.
- And abstract them.

My Favorite Scripts

- ...are incredibly simple.
- They just eliminate some keystrokes.

~/bin/sx

```
#!/bin/sh  
screen -x $@
```

~/bin/n2pm

```
#!/bin/sh  
module-starter --author="Ricardo Signes \  
--email="rjbs@cpan.org" $@
```

Functions

- These don't need to be scripts.
- You can do this with functions.

```
knave!rjbs% sx() { screen -x $@ }
```

```
knave!rjbs% cs() { cp $2 $2.bak; cp $1 $2 }
```

```
knave!rjbs% ch() {  
    awk 'FNR==1{for(i=1;i<=NF;i++)print $i}{next}' $@  
}
```

Aliases

- There are three reasons to bother:
 - better recursion protection
 - non-command aliases (the space hack)
 - higher precedence
- I still don't bother.

Aliases

- Aliases can do much of what functions can.
- Quoting makes them hard to write.
- Single-lining makes them harder.
- Those can be worked around, but why bother?

Functions

- More reasons to use functions:
 - they're already in memory
 - they run in the current process
 - they produce scope

Scope

- Scope is the place where something is visible.
- Using scope wisely makes life easy.
- The file hierarchy is a kind of scope.
- So is the process environment.

Scope

Functions can alter the running shell. (It's in scope.)

```
cvsqrt() {  
    if [ "$1" != "" ]; then  
        if [ -f $CVSROOT_DIR/$1 ]; then  
            CVSROOT=$(cat $CVSROOT_DIR/$1)  
        else  
            echo cvsroot: $CVSROOT_DIR/$1 is invalid  
        fi  
    else  
        echo cvsroot: currently $CVSROOT  
    fi  
}
```

Scope

Function variables can be made local (scope-limited).

```
mown() {  
    local filename = $1  
    local owner = $(ls -l $1 | cut -c21-29)  
    mail $owner  
}
```

Scope

Nested functions are scoped to the enclosing function.

```
#!/bin/sh
hypotenuses() {
    pythagoras() {
        echo $(dc -e "$1 2 ^ $2 2 ^ v f")
    }
    for $side in $*; do pythagoras $side; done
}
```

The `pythagoras()` function is not visible outside of `hypotenuses()`.

Scope

Functions are scoped to their enclosing script.

```
#!/bin/sh

hypotenuse() {
    echo $(dc -e "$1 2 ^ $2 2 ^ v f")
}

for side do hypotenuse $(dc -e "$side 2 / f")
```

The special-purpose `hypotenuse()` function will go away when the script is done running.

Scope

This is how I load my functions.

```
for func in ~/.bash/functions/*; do  
    . $i  
done
```

(it's in my profile)

Scripts

- When are scripts useful?
 - programs not run under a login
 - programs too large to leave in memory
 - rarely-used programs
- In almost no case do I write *sh* scripts.

Scripts

- ...but I write plenty of other scripts.
- Shell scripts lose utility because I'm always in the shell.
- But I'm not limited to shell scripts! I have the shebang!

`#!/usr/bin/whatever`

- Almost any interpreted data file can be run.
- It needs a shebang and +x permissions.
- The key is knowing the right tool.

#! in Action

~/bin/hi_all

```
#!/usr/bin/awk -f
BEGIN { FS=":"; } {}
($7 != "") && ($5 != "") {
{print "Hello, " $1 "!";}
END { print "Buh-bye!" }
```

`#!/usr/bin/whatever`

`~/bin/hi_all`

```
#!/usr/bin/awk -f
BEGIN { FS=":" }

($7 != "") && ($5 != "") {print "Hello, " $1 "!";}

{next}

END { print "Buh-bye!" }
```

#!/usr/bin/whatever

~/bin/backup

```
#!/usr/bin/make -f
RSYNC_SW="--delete -ave ssh"

default: web gnupg logs
mirrors: ifarchive minicpan
all:      default mirrors

web:
    rsync ${RSYNC_SW} ${HOME}/public_html/ cheshire:~/public_html/
gnupg:
    rsync ${RSYNC_SW} cheshire:~/.gnupg/ ${HOME}/backup/gnupg/
logs:
    rsync ${RSYNC_SW} cheshire:/var/www/ ${HOME}/backup/www/
    rsync ${RSYNC_SW} cheshire:~/cvs/    ${HOME}/backup/cvs/
    rsync ${RSYNC_SW} cheshire:~/log/    ${HOME}/backup/log/
ifarchive:
    rsync ${RSYNC_SW} cheshire:~/ifarchive/ ${HOME}/mirrors/ifarchive/
```

`#!/usr/bin/whatever`

`~/bin/bestsongs`

```
#!/usr/bin/mpg123 -z -@  
/usr/bin/music/outkast/ms.jackson.mp3  
/usr/bin/music/devo/shout.mp3  
/usr/bin/music/talking_heads/television_man.mp3  
...
```

```
addsong() {  
    if [ "$1" != "" -a -r "$1" ]; then  
        echo $1 >> ~/bin/bestsongs;  
    else echo can't add file  
    }
```

More Slides

things that deserve more time

Globber

- * and ?
- [ABC]
- {xyz}
- *zsh* globbing

Environment Variables

- shell vars v. environment vars
- `$x` and `${x}`
- `${x:-default}`
- `${x:=default}`
- `${x:?epitaph}`
- `${x:+iftrue}`
- `${x:offset:length}`

STDIO

- `` and \$()
- >&
- << and <<-
- command blocks { }
- the read builtin

Magic Nonsense

- (job&) # job into void
- \$((2+2))
- “” versus “
 - ugh! ‘abc’\”def’
- eval
- source

Precedence Hackery

- command
- builtin
- enable