Chapter 9.
A Programmable Shell:
Shell Variables and the Environment

Eun-Kyung Ryu
KNU CSE / SWEDU

Objectives

Ideas and Skills

- A Unix shell is a programming language
- What is a shell script? How does a shell process a script?
- How do shell control structures work? `exit(0)` = success
- Shell variables: why and how
- What is the environment? How does it work?

System Calls and Functions

- `exit`
- `getenv`

Commands

- `env`

Contents

Shell Programming

- **A Unix shell** runs programs ***and*** **is itself a programming language.**

- ***Shell programs*****, called *shell scripts*, are an essential part of Unix**

Contents

- **A Unix shell is an *interpreter*** for a programming language.
  - This interpreter interprets commands from the *keyboard*
  - It also interprets sequences of commands stored in *shell scripts*.

9.2.1 A Shell Script Is a Batch of Commands

- **A shell script is a file** that contains a batch of commands.

- **Running a script** means executing each command in the file.

```
# this is called script0
# it runs some commands
ls
echo the current date/time is
date
echo my name is
whoami
```

comments

commands

Running a Shell Script : Two Ways

```
$ sh script0
script0  script1  script2  script3
the current date/time is
Sun Jul 29 23:29:49 EDT 2001
my name is
bruce
$
```

```
$ chmod +x script0
$./script0
script0  script1  script2  script3
the current date/time is
Sun Jul 29 23:31:23 EDT 2001
my name is
bruce
$
```

※ Marking a script as executable makes the script a command.

- **Which Shell Are We Using?**
  - We use the syntax of the original Unix shell, **`sh`** (called the **Bourne Shell**)
  - The tiny subset of syntax we shall study is common to several shells, including `sh`, `bash`, and `ksh`.

Programming Features of `sh`: Variables, I/O, and If..Then

```
#!/bin/sh
# script2: a real program with variables, input,
#          and control flow

BOOK=$HOME/phonebook.data
echo find what name in phonebook
read NAME
if grep $NAME $BOOK > /tmp/pb.tmp
then
        echo Entries for $NAME
        cat /tmp/pb.tmp
else
        echo No entries for $NAME
fi
rm /tmp/pb.tmp
```

```
$ ./script2
find what name in phonebook
dave
Entries for dave
dave    432-6546
```

```
$ ./script2
find what name in phonebook
fran
No entries for fran
```

```
$ cat $HOME/phonebook.data
ann     222-3456
bob     323-2222
carla   123-4567
dave    432-6546
eloise  567-9876
$
```

- **Improving Our Shell (psh2 in Chapter 08)**
  - Add command-line parsing (9.3)
  - Add an *if..then* structure (9.4)
  - Add local and environment variables (9.5)

Contents

- **The program logic of `smsh1.c` (Improvement from `psh2.c`)**

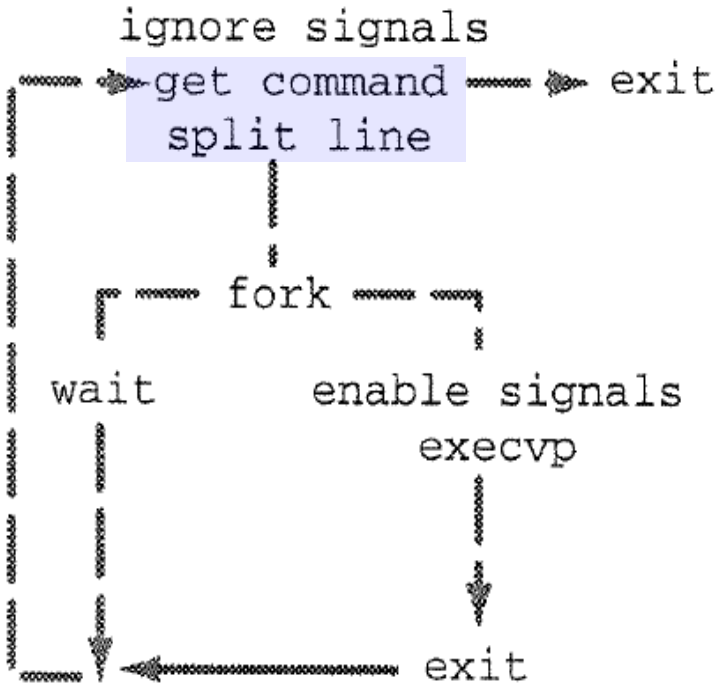


FIGURE 9.1

A shell with signals, exit, and parsing.

```
$ ./smsh1
> ps -f
...

> ls -l
...
>
```

- **Compile and run**

```
$ cc smsh1.c splitline.c execute.c -o smsh1
$ ./smsh1
> ps -f
UID        PID  PPID  C STIME TTY          TIME CMD
bruce    23203 23199  0 Jul29 pts/4    00:00:00 bash
bruce    25383 23203  0 08:23 pts/4    00:00:00 ./smsh1
bruce    25385 25383  0 08:23 pts/4    00:00:00 ps -f
> press Ctrl-D here
$
```

```
/**  smsh1.c  small-shell version 1
 **              first really useful version after prompting shel
 **              this one parses the command line into strings
 **              uses fork, exec, wait, and ignores signals
 **/

#include        <stdio.h>
#include        <stdlib.h>
#include        <unistd.h>
#include        <signal.h>
#include        "smsh.h"

#define DFL_PROMPT      "> "

int main()
{
        char    *cmdline, *prompt, **arglist;
        int     result;
        void    setup();

        prompt = DFL_PROMPT ;
        setup();

        while ( (cmdline = next_cmd(prompt, stdin)) != NULL ){
                if ( (arglist = splitline(cmdline)) != NULL  ){
                        result = execute(arglist);
                        freelist(arglist);
                }
                free(cmdline);
        }
        return 0;
}
```

```
$ ./smsh1
> ps -f
...

> ls -l
...
>
```

```
void setup()
/*
 * purpose: initialize shell
 * returns: nothing. calls fatal() if trouble
 */
{
        signal(SIGINT,  SIG_IGN);
        signal(SIGQUIT, SIG_IGN);
}

void fatal(char *s1, char *s2, int n)
{
        fprintf(stderr,"Error: %s,%s\n", s1, s2);
        exit(n);
}
```

```
/* execute.c - code used by small shell to execute commands */

#include        <stdio.h>
#include        <stdlib.h>
#include        <unistd.h>
#include        <signal.h>
#include        <sys/wait.h>

int execute(char *argv[])
/*
 * purpose: run a program passing it arguments
 * returns: status returned via wait, or -1 on error
 *  errors: -1 on fork() or wait() errors
 */
{
        int     pid ;
        int     child_info = -1;

        if ( argv[0] == NULL )          /* nothing succeeds     */
                return 0;

        if ( (pid = fork())  == -1 )
                perror("fork");
        else if ( pid == 0 ){
                signal(SIGINT, SIG_DFL);
                signal(SIGQUIT, SIG_DFL);
                execvp(argv[0], argv);
                perror("cannot execute command");
                exit(1);
        }
        else {
                if ( wait(&child_info) == -1 )
                        perror("wait");
        }
        return child_info;
}
```

```
/* splitline.c - commmand reading and parsing functions for smsh
 *
 *    char *next_cmd(char *prompt, FILE *fp) - get next command
 *    char **splitline(char *str);           - parse a string
 */

#include        <stdio.h>
#include        <stdlib.h>
#include        <string.h>
#include        "smsh.h"

char * next_cmd(char *prompt, FILE *fp)
/*
 * purpose: read next command line from fp
 * returns: dynamically allocated string holding command line
 *  errors: NULL at EOF (not really an error)
 *          calls fatal from emalloc()
 *   notes: allocates space in BUFSIZ chunks.
 */
{
        char    *buf ;                  /* the buffer           */
        int     bufspace = 0;           /* total size           */
        int     pos = 0;                /* current position     */
        int     c;                      /* input char           */
```

```
$ ./smsh1
> ps -f
...

> ls -l
...
>
```

```
    printf("%s", prompt);                          /* prompt user  */
    while( ( c = getc(fp)) != EOF ) {

            /* need space? */
           if( pos+1 >= bufspace ){                /* 1 for \0     */
                   if ( bufspace == 0 )            /* y: 1st time  */
                           buf = emalloc(BUFSIZ);
                   else                            /* or expand    */
                           buf = erealloc(buf,bufspace+BUFSIZ);
                   bufspace += BUFSIZ;             /* update size  */
            }

            /* end of command? */
            if ( c == '\n' )
                    break;

            /* no, add to buffer */
            buf[pos++] = c;
    }
    if ( c == EOF && pos == 0 )             /* EOF and no input     */
            return NULL;                    /* say so               */
    buf[pos] = '\0';
    return buf;
}
```

```
$ ./smsh1
> ps -f
...

>
```

```
/**
 **      splitline ( parse a line into an array of strings )
 **/
#define is_delim(x) ((x)==' '||(x)=='\t')
char ** splitline(char *line)
/*
 * purpose: split a line into array of white-space separated tokens
 * returns: a NULL-terminated array of pointers to copies of the tokens
 *          or NULL if line if no tokens on the line
 *  action: traverse the array, locate strings, make copies
 *    note: strtok() could work, but we may want to add quotes later
 */
{
        char    *newstr();
        char    **args ;
        int     spots = 0;              /* spots in table       */
        int     bufspace = 0;           /* bytes in table       */
        int     argnum = 0;             /* slots used           */
        char    *cp = line;             /* pos in string        */
        char    *start;
        int     len;
```

```
$ ./smsh1
> ps -f
...

>
```

```
    if ( line == NULL )                     /* handle special case  */
            return NULL;

    args     = emalloc(BUFSIZ);             /* initialize array     */
    bufspace = BUFSIZ;
    spots    = BUFSIZ/sizeof(char *);

    while( *cp != '\0' )

    {
            while ( is_delim(*cp) )         /* skip leading spaces  */
                    cp++;
            if ( *cp == '\0' )              /* quit at end-o-string */
                    break;

            /* make sure the array has room (+1 for NULL) */
            if ( argnum+1 >= spots ){
                    args = erealloc(args,bufspace+BUFSIZ);
                    bufspace += BUFSIZ;
                    spots += (BUFSIZ/sizeof(char *));
            }

            /* mark start, then find end of word */
            start = cp;
            len   = 1;
            while (*++cp != '\0' && !(is_delim(*cp)) )
                    len++;
            args[argnum++] = newstr(start, len);
    }
    args[argnum] = NULL;
    return args;
}
```

```
/*
 * purpose: constructor for strings
 * returns: a string, never NULL
 */
char *newstr(char *s, int l)
{
        char *rv = emalloc(l+1);

        rv[l] = '\0';
        strncpy(rv, s, l);
        return rv;
}

void
freelist(char **list)
/*
 * purpose: free the list returned by splitline
 * returns: nothing
 *  action: free all strings in list and then free the list
 */
{
        char    **cp = list;
        while( *cp )
                free(*cp++);
        free(list);
}
```

```
void * emalloc(size_t n)
{
        void *rv ;
        if ( (rv = malloc(n)) == NULL )
                fatal("out of memory","",1);
        return rv;
}
void * erealloc(void *p, size_t n)
{
        void *rv;
        if ( (rv = realloc(p,n)) == NULL )
                fatal("realloc() failed","",1);
        return rv;
}
```

```
/* smsh.h */
#define YES     1
#define NO      0

char    *next_cmd();
char    **splitline(char *);
void    freelist(char **);
void    *emalloc(size_t);
void    *erealloc(void *, size_t);
int     execute(char **);
void    fatal(char *, char *, int );
```

Contents

Compile and Run

```
$ cc -o smsh2 smsh2.c splitline.c execute.c process.c controlflow.c
$ ./smsh2
> grep lp /etc/passwd
lp:x:4:7:lp:/var/spool/lpd:
> if grep lp /etc/passwd
lp:x:4:7:lp:/var/spool/lpd:
> then
>   echo ok
ok
> fi
> if grep pati /etc/passwd
> then
>   echo ok
> fi
> echo ok
ok
> then
syntax error: then unexpected
```

9.4.1 What `if` Does

- **`if.. Then`**
  - In the shell, the *condition* is a command, and a ***positive result*** means the command succeeded.

if structure

```
if command
then
commands
else
commands
fi
```

```
if date | grep Fri
then
   echo time for backup.  Insert tape and press enter
   read x
   tar cvf /dev/tape /home
fi
```

- **How can a program indicate success?**

- **`exit(0)` for Success**
  - `ex) grep, diff, mv, cp, rm, ...`
    - All Unix programs follow the convention that an exit value of 0 signifies success.

- ***if… then… else…***
  - The …block may contain **any number of commands**, including other *if*..*then* control structures.
- ***If…* accepts a block of commands**
  - The `exit` value from ***the last command*** in the block determines the success of the condition.

if structure

```
if command
then
commands
else
commands
fi
```

9.4.2 How *if* Works

- **The *if* control structure works as follows:**

(a) The shell runs the command that follows the word *if*.
(b) The shell checks the `exit` status of the command.
(c) An `exit` status of 0 means *success*, nonzero means *failure*.
(d) The shell executes commands after the *then* line if success.
(e) The shell executes commands after the *else* line if failure.
(f) The keyword *fi* marks the end of the *if* block.

```
if date | grep Fri
then
   echo time for backup.  Insert tape and press enter
   read x
   tar cvf /dev/tape /home
fi
```

9.4.3 Adding *if* to `smsh`

- **Adding a New Layer: `process`**

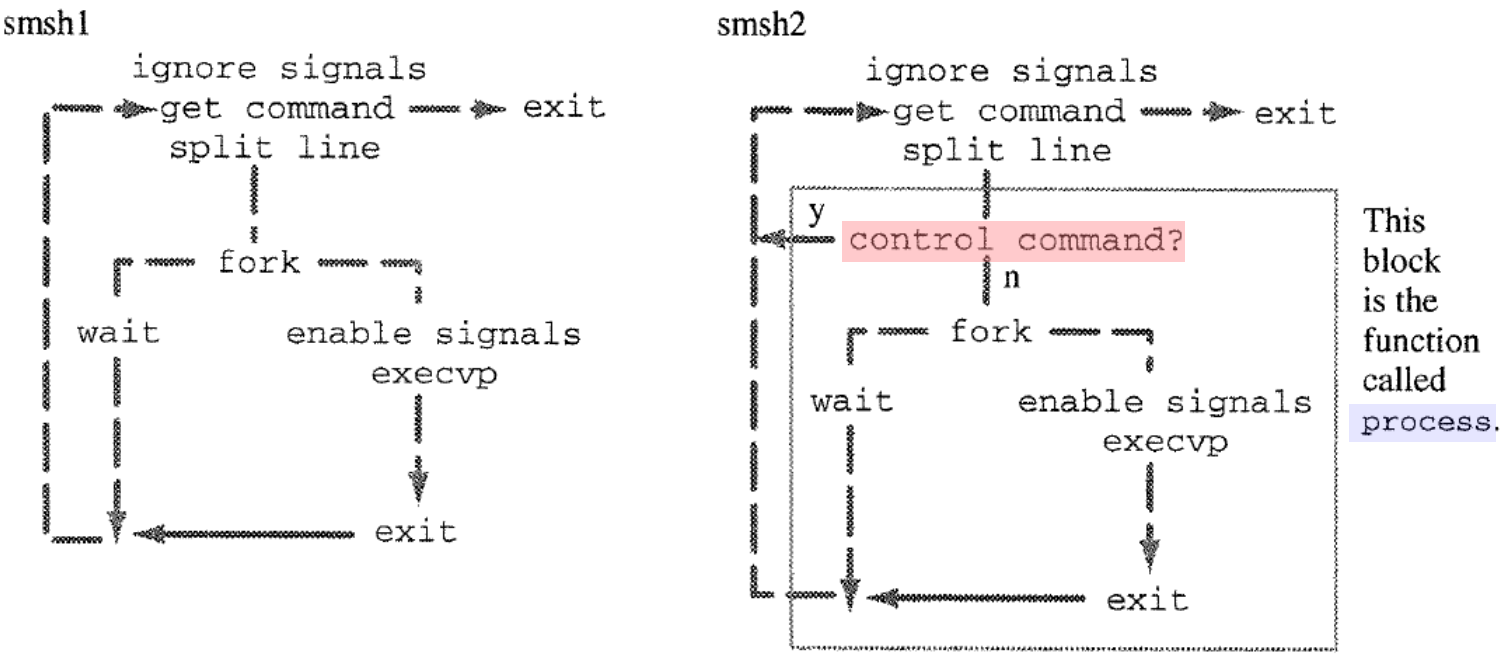
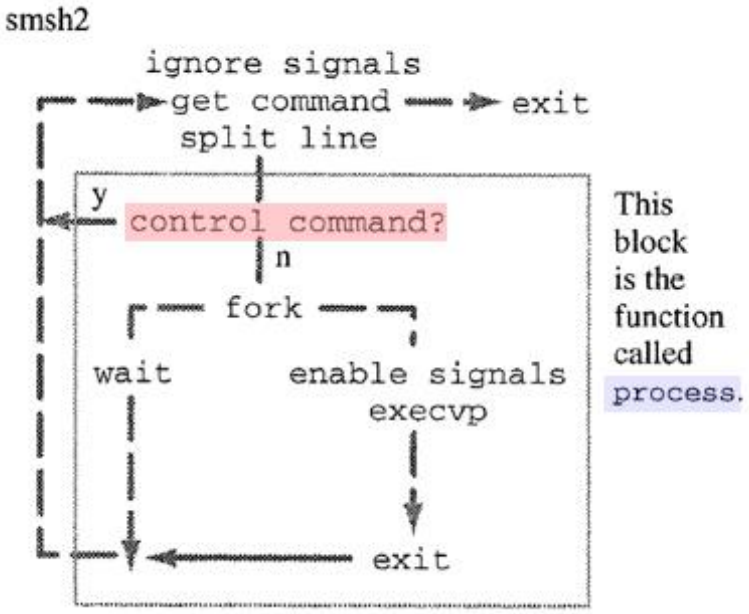


FIGURE 9.2

Adding flow control commands to `smsh`.

※ control commands
if, then, else, fi, …

- **What `process` Does**
  - It manages the control flow of a script by watching for keywords like ***if***, ***then***, and ***fi***
  - It also calls ***fork*** and ***exec*** only when appropriate.
  - It must record the result of condition commands so it can decide how to handle ***then*** and ***else*** blocks.



- **How Does `process` Work?**
  - The shell treats *commands in different regions* in *different ways*.

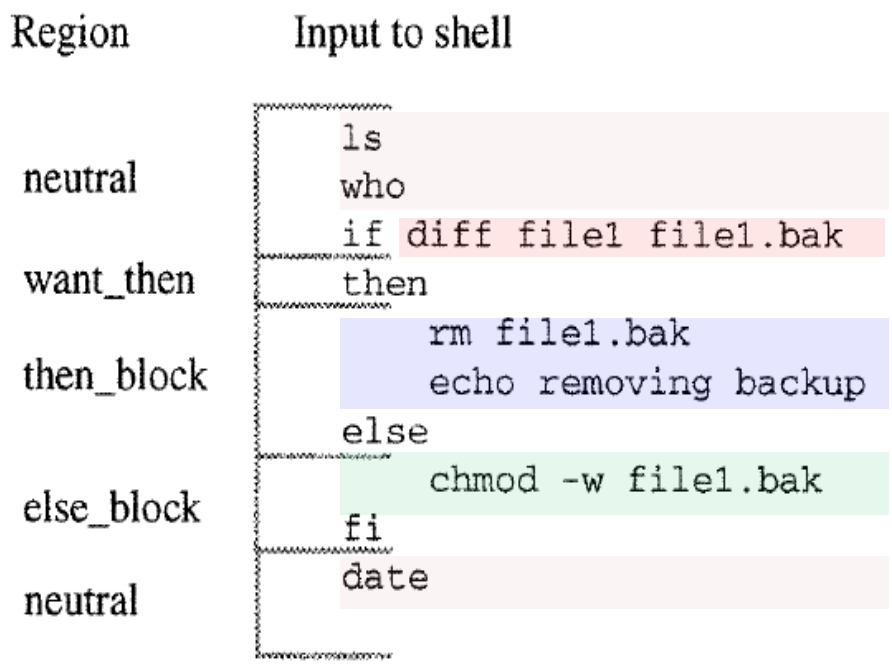
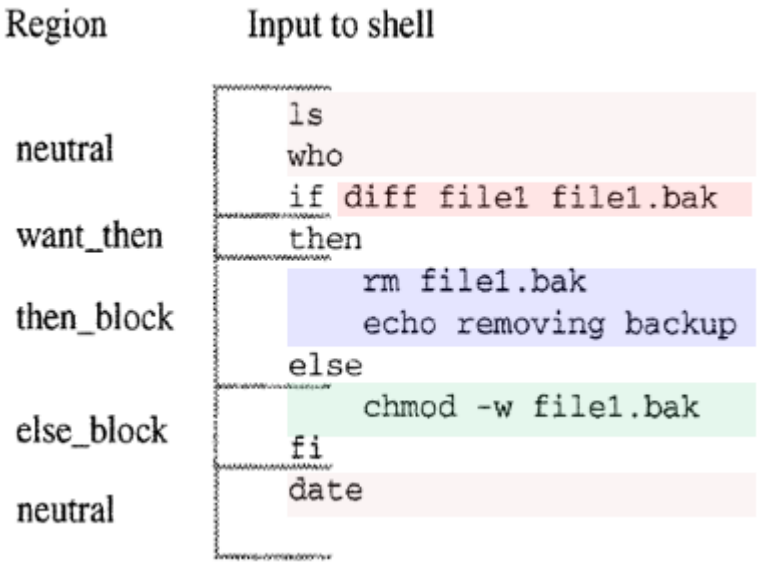


FIGURE 9.3

A script consists of different regions.

- **The shell must *keep track of***
  - the current region and
  - the success or failure of the command executed when it shifted into the `WANT_THEN` region.
- **Different regions** require **different types of processing.**



- **`process` calls three functions:**
  - **`is_control_command`**
    - **tells process *if*** *the command is part of the shell programming language* **or *if*** *the command is something to execute*.
  - **`do_control_command`**
    - handles the keywords ***if*, *then*, and *fi*.**
    - ***updates*** *the state variable* and *performs any appropriate operations*.
  - **`ok_to_execute`**
    - checks *the current state* and *the result of the condition command*.
    - returns a boolean value to indicate ***if the current command should be executed*.**

9.4.4 smsh2.c

```
$ cc -o smsh2 smsh2.c splitline.c execute.c process.c controlflow.c
$ ./smsh2
> grep lp /etc/passwd
lp:x:4:7:lp:/var/spool/lpd:
> if grep lp /etc/passwd
lp:x:4:7:lp:/var/spool/lpd:
> then
>   echo ok
ok
> fi
> if grep pati /etc/passwd
> then
>   echo ok
> fi
> echo ok
ok
> then
syntax error: then unexpected
```

```
/** smsh2.c - small-shell version 2
 **             small shell that supports command line parsing
 **             and if..then..else.fi logic (by calling process())
 **/
#include        <stdio.h>
#include        <stdlib.h>
#include        <unistd.h>
#include        <signal.h>
#include        <sys/wait.h>
#include        "smsh.h"

#define DFL_PROMPT      "> "

int main()
{
        char    *cmdline, *prompt, **arglist;
        int     result, process(char **);
        void    setup();

        prompt = DFL_PROMPT ;
        setup();

        while ( (cmdline = next_cmd(prompt, stdin)) != NULL ){
                if ( (arglist = splitline(cmdline)) != NULL  ){
                        result = process(arglist);
                        freelist(arglist);
                }
                free(cmdline);
        }
        return 0;
}
```

```
$ ./smsh2
> grep lp /etc/passwd
lp:x:4:7:lp:/var/spool/lpd:
> if grep lp /etc/passwd
lp:x:4:7:lp:/var/spool/lpd:
> then
>   echo ok
ok
> fi
```

```
void setup()
/*
 * purpose: initialize shell
 * returns: nothing. calls fatal() if trouble
 */
{
        signal(SIGINT,  SIG_IGN);
        signal(SIGQUIT, SIG_IGN);
}

void fatal(char *s1, char *s2, int n)
{
        fprintf(stderr,"Error: %s,%s\n", s1, s2);
        exit(n);
}
```

```
/* process.c
 * command processing layer
 *
 * The process(char **arglist) function is called by the main loop
 * It sits in front of the execute() function.  This layer handles
 * two main classes of processing:
 *      a) built-in functions (e.g. exit(), set, =, read, .. )
 *      b) control structures (e.g. if, while, for)
 */

#include        <stdio.h>
#include        "smsh.h"

int is_control_command(char *);
int do_control_command(char **);
int ok_to_execute();
```

※ control commands

```
int process(char **args)
/*
 * purpose: process user command
 * returns: result of processing command
 * details: if a built-in then call appropriate function, if not execute()
 *  errors: arise from subroutines, handled there
 */
{
        int             rv = 0;

        if ( args[0] == NULL )
                rv = 0;
        else if ( is_control_command(args[0]) )
                rv = do_control_command(args);
        else if ( ok_to_execute() )
                rv = execute(args);
        return rv;
}
```

```
$ ./smsh2
> grep lp /etc/passwd
lp:x:4:7:lp:/var/spool/lpd:
> if grep lp /etc/passwd
lp:x:4:7:lp:/var/spool/lpd:
> then
>   echo ok
ok
> fi
```

```
/* controlflow.c
 *
 * "if" processing is done with two state variables
 *    if_state and if_result
 */
#include        <stdio.h>
#include        "smsh.h"

enum states   { NEUTRAL, WANT_THEN, THEN_BLOCK };
enum results  { SUCCESS, FAIL };

static int if_state  = NEUTRAL;
static int if_result = SUCCESS;
static int last_stat = 0;

int     syn_err(char *);
```

```
int ok_to_execute()
/*
 * purpose: determine the shell should execute a command
 * returns: 1 for yes, 0 for no
 * details: if in THEN_BLOCK and if_result was SUCCESS then yes
 *          if in THEN_BLOCK and if_result was FAIL    then no
 *          if in WANT_THEN  then syntax error (sh is different)
 */
{
        int     rv = 1;         /* default is positive */

        if ( if_state == WANT_THEN ){
                syn_err("then expected");
                rv = 0;
        }
        else if ( if_state == THEN_BLOCK && if_result == SUCCESS )
                rv = 1;
        else if ( if_state == THEN_BLOCK && if_result == FAIL )
                rv = 0;
        return rv;
}
```

```
int is_control_command(char *s)
/*
 * purpose: boolean to report if the command is a shell control command
 * returns: 0 or 1
 */
{
    return (strcmp(s,"if")==0 || strcmp(s,"then")==0 || strcmp(s,"fi")==0);
}
```

```
int do_control_command(char **args)
/*
 * purpose: Process "if", "then", "fi" - change state or detect error
 * returns: 0 if ok, -1 for syntax error
 */
{
        char    *cmd = args[0];
        int     rv = -1;

        if( strcmp(cmd,"if")==0 ){
                if ( if_state != NEUTRAL )
                        rv = syn_err("if unexpected");
                else {
                        last_stat = process(args+1);
                        if_result = (last_stat == 0 ? SUCCESS : FAIL );
                        if_state = WANT_THEN;
                        rv = 0;
                }
        }
```

```
        else if ( strcmp(cmd,"then")==0 ){
                if ( if_state != WANT_THEN )
                        rv = syn_err("then unexpected");
                else {
                        if_state = THEN_BLOCK;
                        rv = 0;
                }
        }
        else if ( strcmp(cmd,"fi")==0 ){
                if ( if_state != THEN_BLOCK )
                        rv = syn_err("fi unexpected");
                else {
                        if_state = NEUTRAL;
                        rv = 0;
                }
        }
        else
                fatal("internal error processing:", cmd, 2);
        return rv;
}
```

```
int syn_err(char *msg)
/* purpose: handles syntax errors in control structures
 * details: resets state to NEUTRAL
 * returns: -1 in interactive mode. Should call fatal in scripts
 */
{
        if_state = NEUTRAL;
        fprintf(stderr,"syntax error: %s\n", msg);
        return -1;
}
```

- **How'd We Do?**
  - Ours looks OK; The *standard shell* handles the *if* structure in a different way from ours.

```
$ cc -o smsh2 smsh2.c splitline.c execute.c process.c controlflow.c
$ ./smsh2
> grep lp /etc/passwd
lp:x:4:7:lp:/var/spool/lpd:
> if grep lp /etc/passwd
lp:x:4:7:lp:/var/spool/lpd:
> then
>   echo ok
ok
> fi
> if grep pati /etc/passwd
> then
>   echo ok
> fi
> echo ok
ok
> then
syntax error: then unexpected
```

※ standard shell

```
$ if grep lp /etc/passwd
> then
>   echo ok
> fi
lp:x:4:7:lp:/var/spool/lpd:
ok
$
```

Contents

- **Like any programming language, a Unix shell has *variables.***

```
$ age=7                                  # assigning a value
$ echo $age                              # retrieving a value
7
$ echo age                               # the $ is required
age
$ echo $age+$age                         # purely string operations
7+7
$ read name                              # input from stdin
fido
$ echo hello, $name, how are you         # can be interpolated
hello, fido, how are you
$ ls > $name.$age                        # used as part of a command
$ food = muffins                         # no spaces in assignment
food: not found
$
```

※ filename: fido.7

- **The *shell* includes two types of variables: local variables and environment variables.**
  - These **environment** variables behave somewhat **like global variables**;
  - their values are **accessible to all *child* process** of the shell. (ex. $HOME)

9.5.1 Using Shell Variables

- **Operations with variables**

Operation	Syntax	Notes
assignment	`var=value`	no spaces
reference	`$var`	
delete	`unset var`	
stdin input	`read var`	also, `read var1 var2 ..`
list vars	`set`	
make global	`export var`	

- **Variable names** are **combinations of the characters A-Z, a-z, 0-9, and_.**
  - The first character may not be a digit.
  - The names are case sensitive.

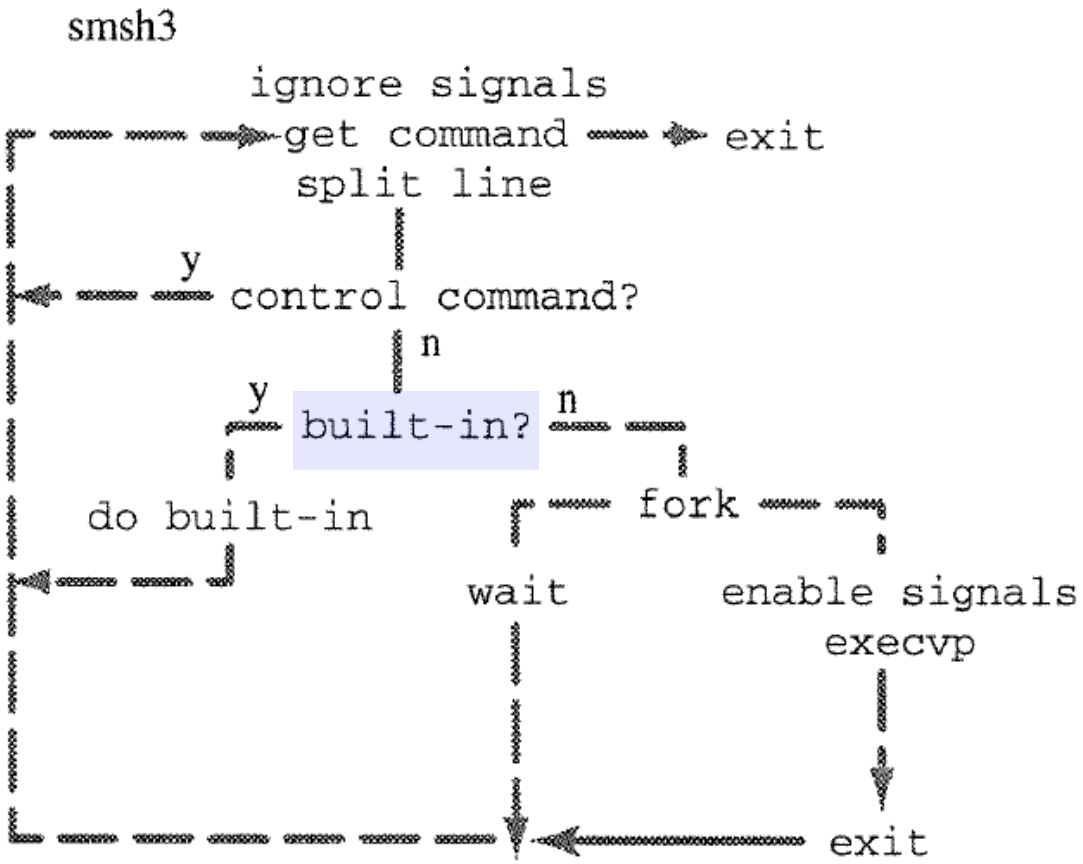
- **Variable values** are ***strings*.**
  - There are no numerical values.
  - All variable operations are ***string operations***.

- **Listing variables** involves **the `set` command,**

```
$ set
BASH=/bin/bash
BASH_VERSION=1.14.7(1)
DISPLAY=:0.0
EUID=500
HOME=/home2/bruce
HOSTTYPE=i386
IFS=

LANG=en
LANGUAGE=en
LD_LIBRARY_PATH=/usr/lib:/usr/local/lib
LOGNAME=bruce
OPTERR=1
OPTIND=1
OSTYPE=Linux
PATH=/bin:/usr/bin:/usr/X11R6/bin:/usr/local/bin:/home2/bruce/bin
PPID=30928
PS4=+
PWD=/home2/bruce/projs/ubook/src/ch09
SHELL=/bin/bash
SHLVL=2
TERM=xterm-color
UID=500
USER=bruce
_=/bin/vi
age=7
name=fido
```

※ This list includes many environment variables that were set when I logged in, plus the two local variables I added later.

- **smsh3: Adding built-ins to our shell:**



Built-ins:

```
$ ./smsh3
> set
> day=monday
> temp=75
> TZ=CST6CDT
```

FIGURE 9.5

Adding built-in commands to smsh.

9.5.2 A Storage System for Variables

- **To add variables to our shell,** we need a place to store these names and values.

variable	*value*	*global*?
`data`	"phonebook.dat"	n
`HOME`	"/home2/fido"	y
`TERM`	"t1061"	y

Built-ins:

```
$ ./smsh3
> set
> day=monday
> temp=75
> TZ=CST6CDT
```

- **Interface (Partial)**

`VLstore(char *var, char *val)` adds/updates *var=val*

`VLlookup(char *var)` retrieves value for *var*

`VLlist` lists table to `stdout`

- **Implementation**

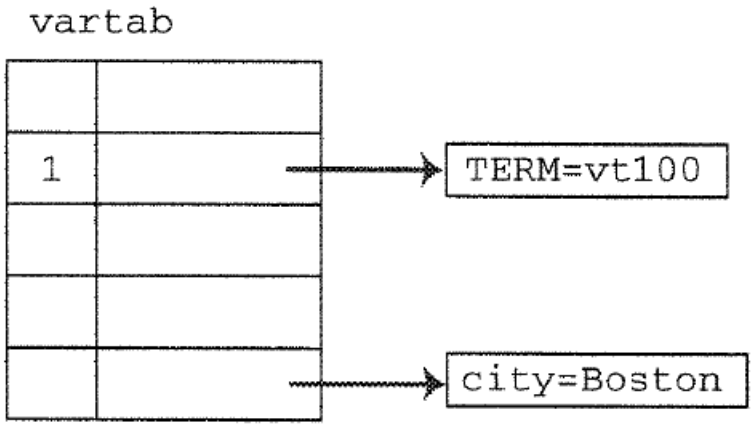


FIGURE 9.4

A storage system for shell variables.

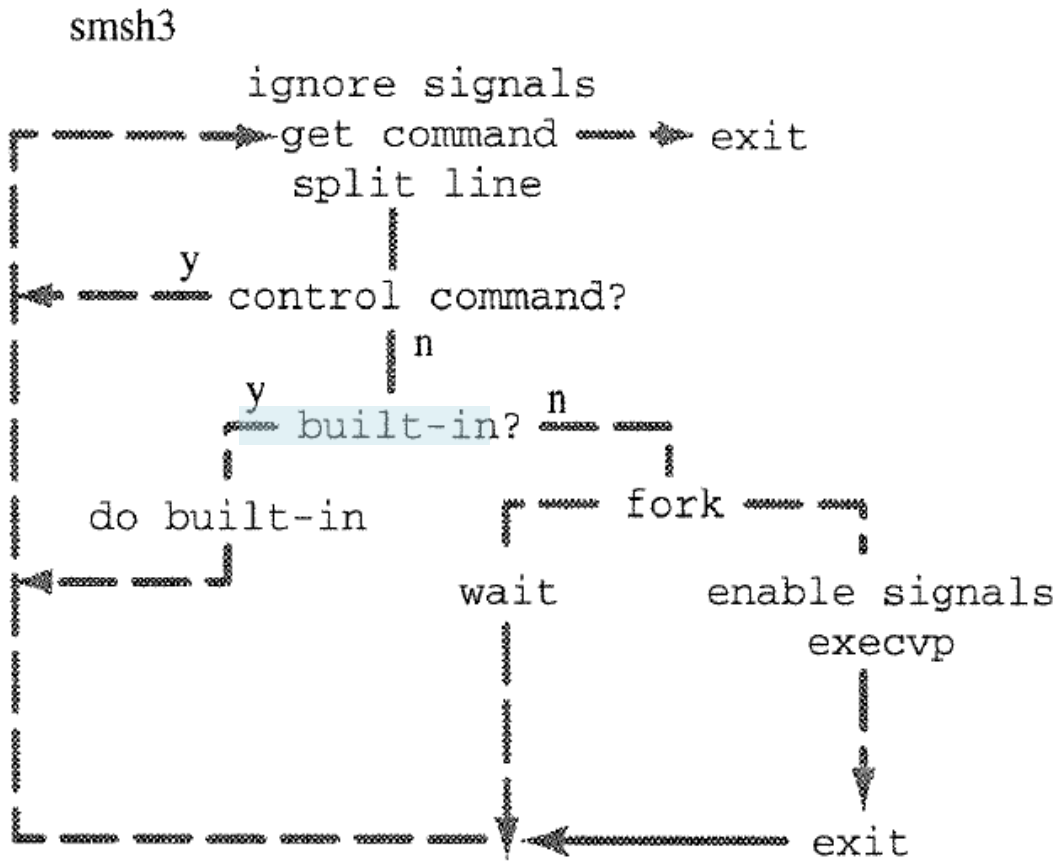
```
struct var {
        char *str;      /* name=val string      */
        int  global;    /* a boolean            */
};
static struct var  vartab[MAXVARS];
```

9.5.3 Adding Variable Commands: Built-Ins

- **Now, we need to *add* the assign, *list*, and *retrieve commands* to our shell.**

```
> TERM=xterm
> set
> echo $TERM
```

- **smsh3: Adding built-ins to our shell:**



Built-ins:

```
$ ./smsh3
> set
> day=monday
> temp=75
> TZ=CST6CDT
```

FIGURE 9.5

Adding built-in commands to smsh.

- **Function: `process2.c`**
  - **Check for built-ins** before calling `fork`/`exec`:

```
if ( args[0] == NULL )
        rv = 0;
else if ( is_control_command(args[0]) )
        rv = do_control_command(args);
else if ( ok_to_execute() ){
    if( !builtin_command(args, &rv) )
            rv = execute(args);
}
```

```
/* builtin.c
 * contains the switch and the functions for builtin commands
 */

#include        <stdio.h>
#include        <string.h>
#include        <ctype.h>
#include        "smsh.h"
#include        "varlib.h"

int assign(char *);
int okname(char *);
```

```
int builtin_command(char **args, int *resultp)
/*
 * purpose: run a builtin command
 * returns: 1 if args[0] is built-in, 0 if not
 * details: test args[0] against all known built-ins.  Call functions
 */
{
        int rv = 0;

        if ( strcmp(args[0],"set") == 0 ){          /* 'set' command? */
                VLlist();
                *resultp = 0;
                rv = 1;
        }
        else if ( strchr(args[0], '=') != NULL ){   /* assignment cmd */
                *resultp = assign(args[0]);
                if ( *resultp != -1 )               /* x-y=123 not ok */
                        rv = 1;
        }
        else if ( strcmp(args[0], "export") == 0 ){
                if ( args[1] != NULL && okname(args[1]) )
                        *resultp = VLexport(args[1]);
                else
                        *resultp = 1;
                rv = 1;
        }
        return rv;
}
```

```
int assign(char *str)
/*
 * purpose: execute name=val AND ensure that name is legal
 * returns: -1 for illegal lval, or result of VLstore
 * warning: modifies the string, but restore    it to normal
 */
{
        char    *cp;
        int     rv ;

        cp = strchr(str,'=');
        *cp = '\0';
        rv = ( okname(str) ? VLstore(str,cp+1) : -1 );
        *cp = '=';
        return rv;
}
```

```
int okname(char *str)
/*
 * purpose: determines if a string is a legal variable name
 * returns: 0 for no, 1 for yes
 */
{
        char    *cp;

        for(cp = str; *cp; cp++ ){
                if ( (isdigit(*cp) && cp==str) || !(isalnum(*cp) || *cp=='_' ))
                        return 0;
        }
        return ( cp != str );   /* no empty strings, either */
}
```

9.5.4 How'd We Do?

```
$ cc -o smsh3 smsh2.c splitline.c execute.c process2.c \
controlflow.c builtin.c varlib.c
$ ./smsh3
> set
> day=monday
> temp=75
> TZ=CST6CDT
> x.y=z
cannot execute command: No such file or directory
> set
    day=monday
    temp=75
    TZ=CST6CDT
> date
Tue Jul 31 11:56:59 EDT 2001
> echo $temp, $day
$temp, $day
```

※ table of variables : `varlib.c, varlib.h` (see smsh4)

U.S. central time

U.S. eastern time

※ date, echo : external programs the shell runs
※ temp, day : local variables
※ TZ : environment variable

Contents

- **Many customized settings are recorded in *environment variables*.**
  - Some programs base their behavior on these settings.

```
#!/bin/sh
# script3 - shows how an environment variable is passed to commands
# TZ is time zone, affect things like date, and ls -l
#
echo "The time in Boston is"
        TZ=EST5EDT
        export TZ               # add TZ to the environment
        date                    # date uses the value in TZ
echo "The time in Chicago is"
        TZ=CST6CDT
        date
echo "The time in LA is"
        TZ=PST8PDT
        date
```

9.6.1 Using the Environment

- **Listing Your Environment**

```
$ env
LOGNAME=bruce
LD_LIBRARY_PATH=/usr/lib:/usr/local/lib
TERM=xterm-color
HOSTTYPE=i386
PATH=/bin:/usr/bin:/usr/X11R6/bin:/usr/local/bin:/home2/bruce/bin
HOME=/home2/bruce
SHELL=/bin/bash
USER=bruce
LANGUAGE=en
DISPLAY=:0.0
LANG=en
_=/usr/bin/env
SHLVL=2
```

- **Updating the Environment :**
  - **To revise** a setting in your environment :
    `var=value`
    `ex) $ LANG=fr`

  - **To add** a new variable to the environment:
    `export var`
    `ex) $ export LANG`

```
root@DESKTOP-K4MA2V5:~# set | grep LANG
LANG=fr
root@DESKTOP-K4MA2V5:~#
```

- **Reading the Environment in C Programs**

```
#include <stdlib.h>
main()
{
        char *cp = getenv("LANG");

        if ( cp != NULL && strcmp(cp, "fr") == 0 )
            printf("Bonjour\n");
        else
            printf("Hello\n");
}
```

9.6.2 What is the Environment?

- The *environment* is simply ***an array of strings*** **available to every program.**

※ global variable

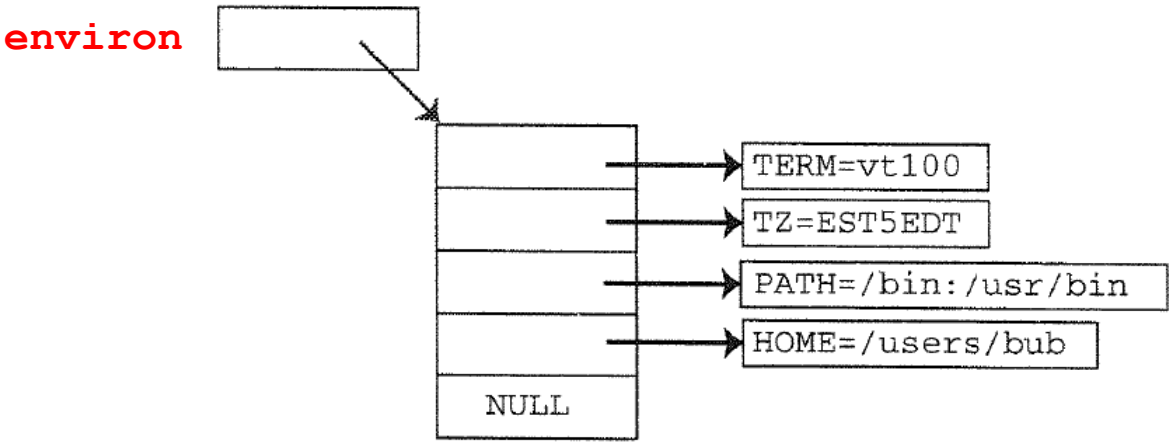
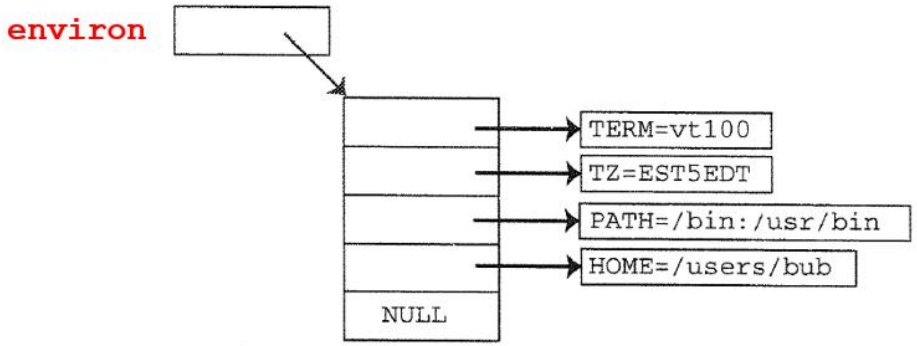


FIGURE 9.6

The environment is an array of pointers to strings.

```c
/*  showenv.c - shows how to read and print the environment
 */
#include        <stdio.h>
extern char     **environ;      /* points to the array of strings */

main()
{
        int     i;

        for( i = 0 ; environ[i] ; i++ )
                printf("%s\n", environ[i] );
}
```



$ man 7 environ

```
execlp(file,argv0,argv1, ..., NULL)
```

execlp does not use an array of arguments as does execvp.

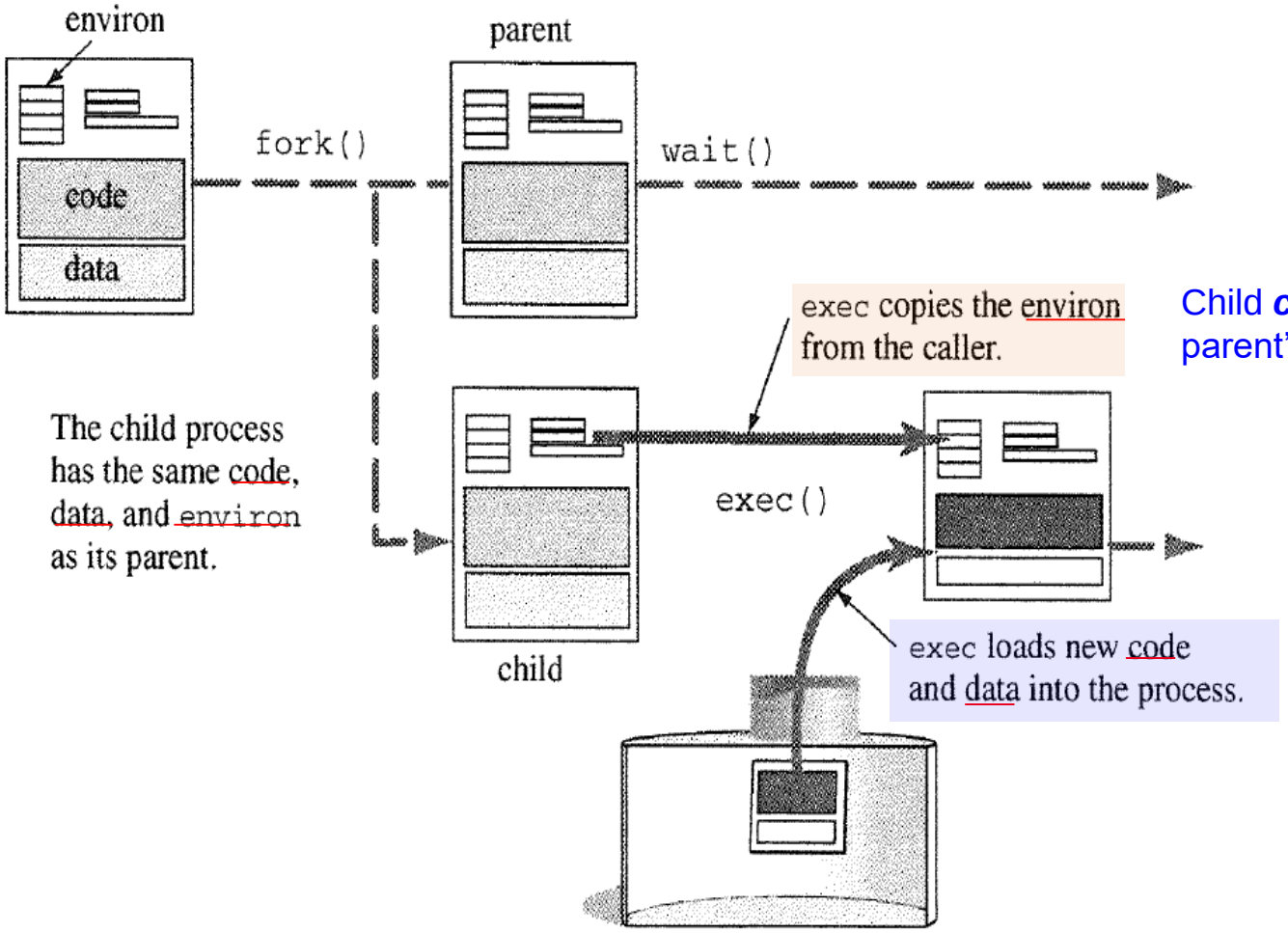
```
execlp("ls", "ls", "-a", "demodir", NULL);
```

```c
/* changeenv.c  - shows how to change the environment
 *                note: calls "env" to display its new settings
 */
#include        <stdio.h>
#include       <unistd.h>
extern char ** environ;

main()
{
        char *table[3];

        table[0] = "TERM=vt100";                    /* fill the table */
        table[1] = "HOME=/on/the/range";
        table[2] = 0;

        environ = table;                            /* point to that table */

        execlp("env", "env", NULL);                 /* exec a program      */
}
```

```
$ ./changeenv
TERM=vt100
HOME=/on/the/range
$
```

Look carefully ! :
The program `env` is able to read the table of strings (`table`)

But exec Wipes Out All Data!



Child ***cannot modify*** parent's environ

FIGURE 9.7

Strings in environ are copied by exec().

9.6.3 Adding Environment Handling to `smsh`

- **Access to Environment Variables:**
  - Ex) set, =,

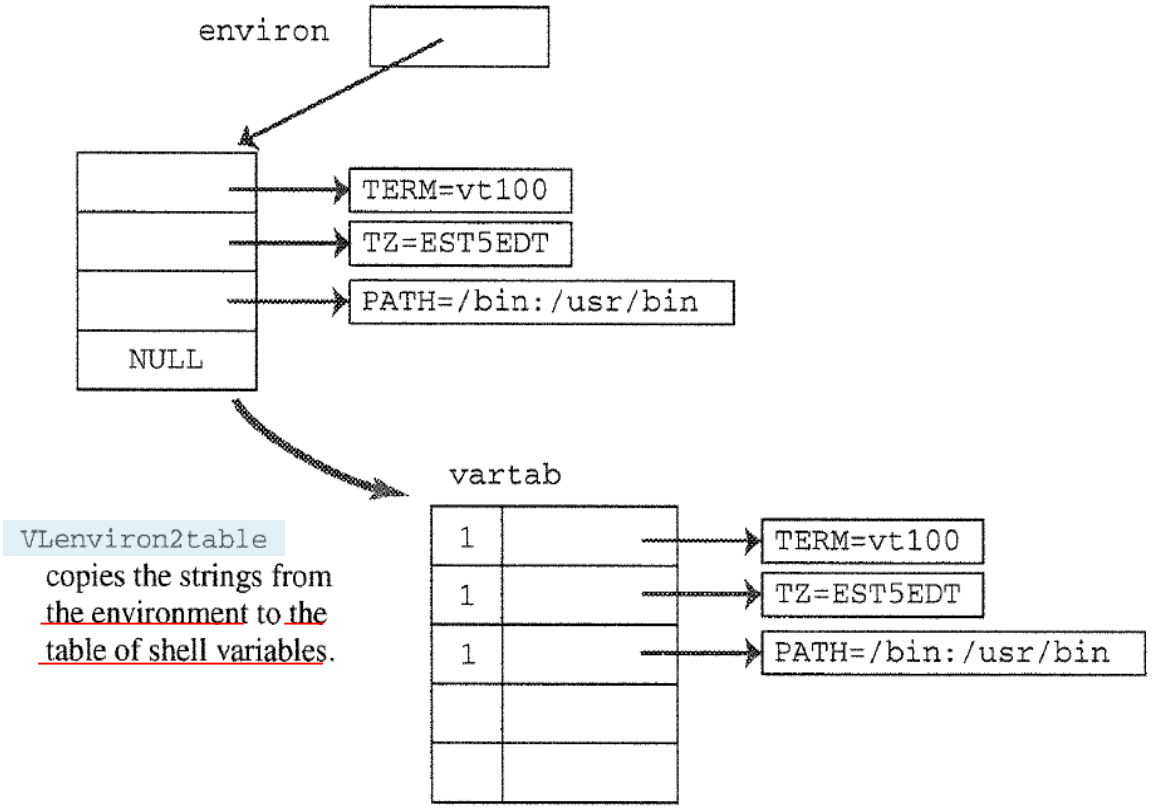


FIGURE 9.8

Copying values from the environment to vartab.

Changing the Environment

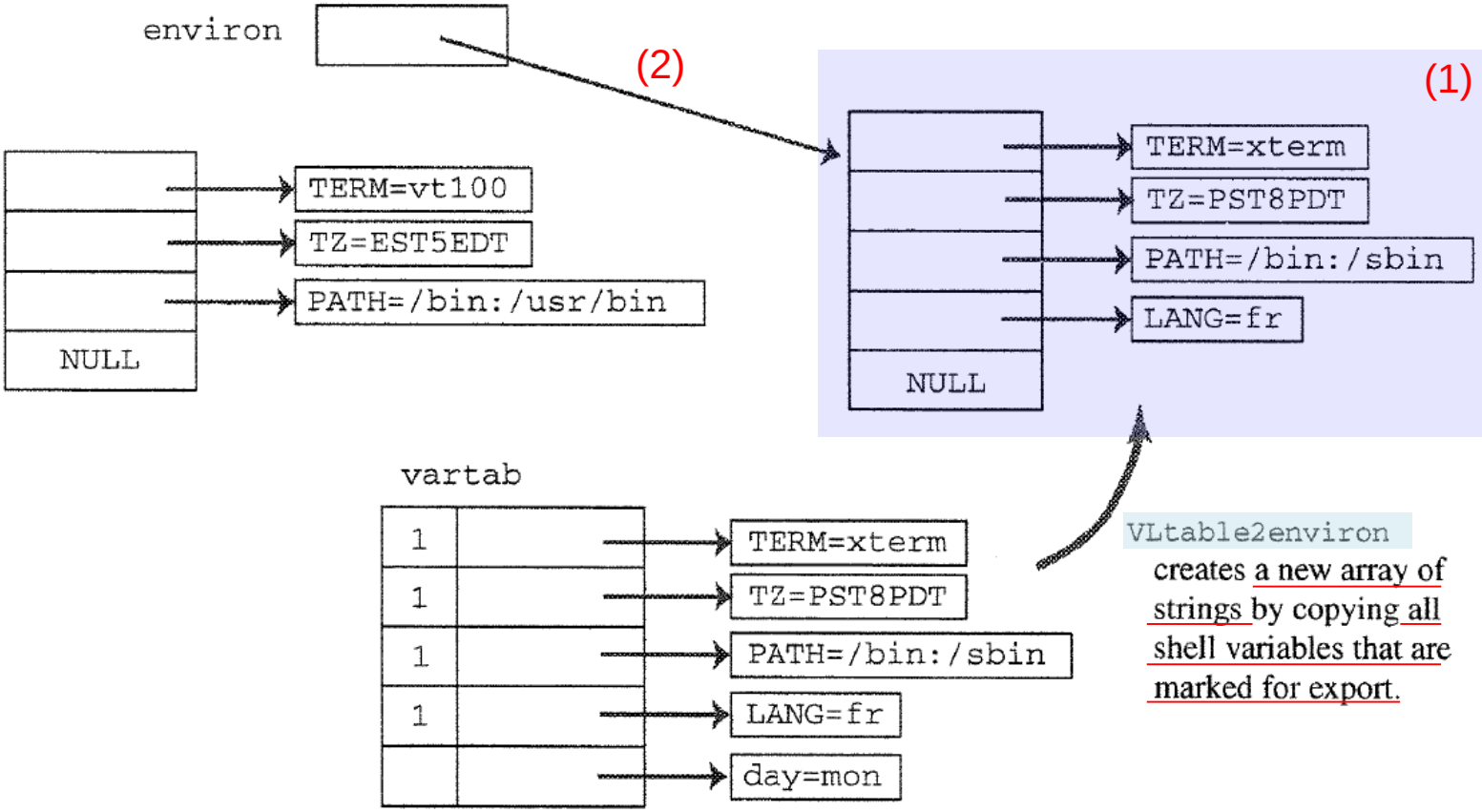


FIGURE 9.9

Copying values from `vartab` to a new environment.

- **Changes to `smsh`**

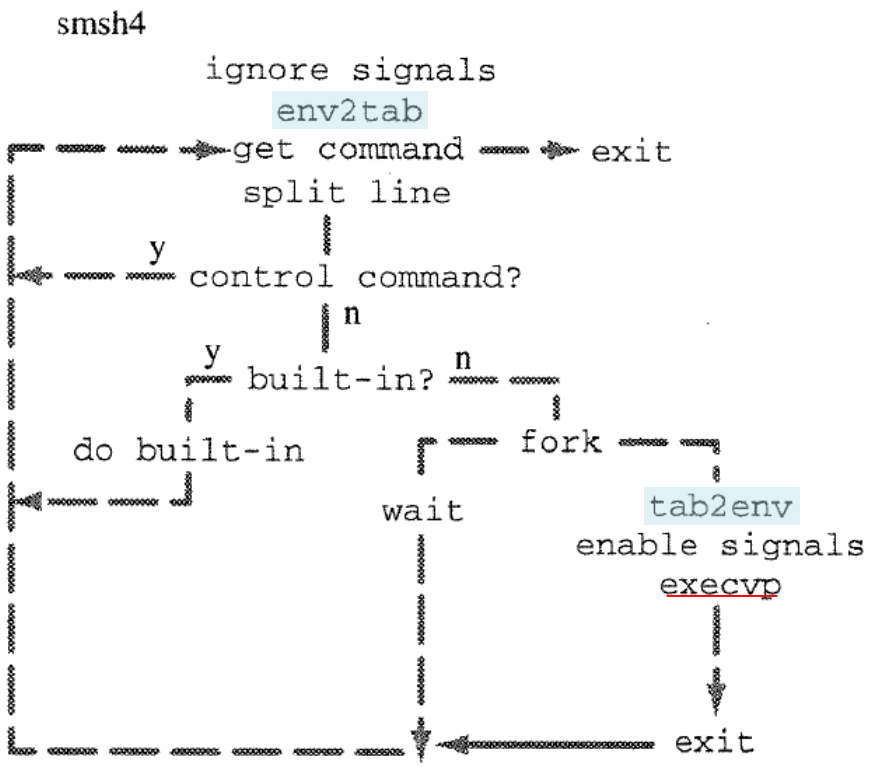


FIGURE 9.10

Adding environment handling to `smsh`.

setup in smsh4.c

```
void setup()
/*
 * purpose: initialize shell
 * returns: nothing. calls fatal() if trouble
 */
{
        extern char **environ;
        VLenviron2table(environ);
        signal(SIGINT,  SIG_IGN);
        signal(SIGQUIT, SIG_IGN);
}
```

execute in execute2.c

```
        if ( (pid = fork())  == -1 )
                perror("fork");
        else if ( pid == 0 ){
                environ = VLtable2environ();  /* new line */
                signal(SIGINT, SIG_DFL);
                signal(SIGQUIT, SIG_DFL);
                execvp(argv[0], argv);
                perror("cannot execute command");
                exit(1);
```

- **Test the Changes :**

```
$ make smsh4
cc -o smsh4 smsh4.c splitline.c execute2.c process2.c controlflow.c \
        builtin.c varlib.c
$ ./smsh4
> date
Tue Jul 31 09:51:03 EDT 2001
> TZ=PST8PDT
> export TZ
> date
Tue Jul 31 06:51:30 PDT 2001
>
```

Contents

- We studied the **Unix shell as a programming language and added three essential features:**
  - command line parsing
  - *if*..*then* logic
  - variables

feature	supports	needs
commands	runs programs	
variables	=, set	read, $var substitution
if	if,then	else
environ	all	
exit		exit
cd		cd
>,<,|	none	all

Contents