System Programming
(ELEC462)

A Programmable Shell

Dukyun Nam

HPC Lab@KNU

Contents

Introduction

- Ideas and Skills
  - A Unix/Linux 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`

Shell Programming

- A Linux shell
  - Runs programs AND is a programming language
    - Last time, we saw **a shell runs a program**
    - This time, we look at **shell programming**
  - Works as interpreter for a programming language
    - Interprets commands from the keyboard
    - Interprets sequences of commands stored in shell scripts
- General remarks
  - A complex task can be solved by combining several separate programs
  - The shell provides a language to control execution and communication of programs
  - The result is a programming environment

Shell Scripts

- Shell script: “A batch of commands”
  - A file that contains “a batch of commands”
    - “Running a script” means “executing each command in sequence” in that file
  - Can be used to perform several commands with a **single** request
- Example
  - The first two lines are comments
  - The shell executes the commands one by one
    - until end of file or until the shell finds an `exit` command

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

Shell Scripts (cont.)

- Running a shell script
  - 1) Run a shell script by passing its name as an argument to the shell

```
dynam@DESKTOP-Q4IJBP7:~/lab10$ sh script0
Makefile       changeenv.c    phonebook.data  script2    smsh.h   smsh2.c  splitline.c
Makefile.smsh  controlflow.c  process.c       script3    smsh1    smsh3    varlib.c
builtin.c      execute.c      process2.c      showenv    smsh1.c  smsh4    varlib.h
changeenv      execute2.c     script0         showenv.c  smsh2    smsh4.c
the current date/time is
Tue Nov  8 00:52:29 KST 2022
my name is
dynam
```

  - 2) Set the executable attribute of the file

```
dynam@DESKTOP-Q4IJBP7:~/lab10$ chmod +x script0
dynam@DESKTOP-Q4IJBP7:~/lab10$ ./script0
Makefile       changeenv.c    phonebook.data  script2    smsh.h   smsh2.c  splitline.c
Makefile.smsh  controlflow.c  process.c       script3    smsh1    smsh3    varlib.c
builtin.c      execute.c      process2.c      showenv    smsh1.c  smsh4    varlib.h
changeenv      execute2.c     script0         showenv.c  smsh2    smsh4.c
the current date/time is
Tue Nov  8 00:53:27 KST 2022
my name is
dynam
```

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

- Shell scripts are REAL programs!
- Shebang = Sharp (#) + Bang (!)
  - #! *interpreter* [*optional-arg*]
    - *Interpreter* is generally an *absolute path* to an executable program
    - The *optional argument* is a string representing a single argument
  - e.g., #!/bin/sh
    - Execute the file using the Bourne shell

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

#BOOK=$HOME/phonebook.data
BOOK=$PWD/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
```

```
dynam@DESKTOP-Q4IJBP7:~/lab10$ cat $PWD/phonebook.data
ann     222-3456
bob     323-2222
carla   123-4567
dave    432-6546
eloise  567-9876
```

```
dynam@DESKTOP-Q4IJBP7:~/lab10$ ./script2
find what name in phonebook
dave
Entries for dave
dave    432-6546
```

Programming Features of `sh` (cont.)

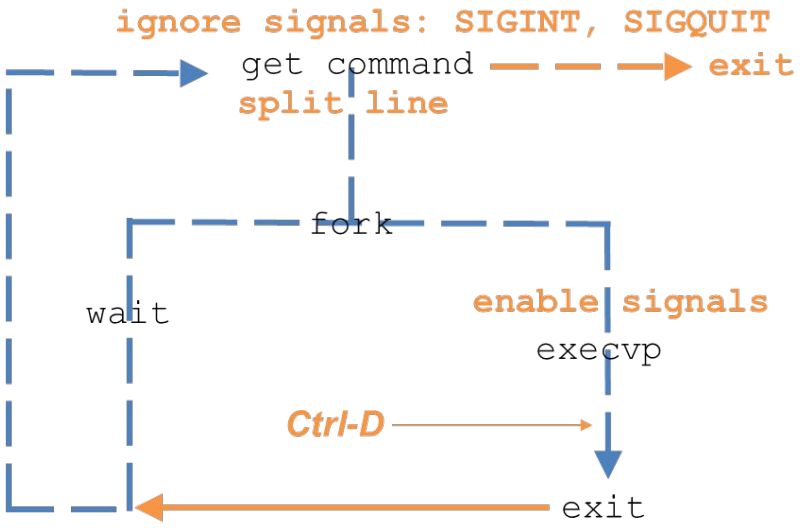
- Variables: e.g., BOOK, NAME
  - Not needed to be uppercase
  - `$`: used for retrieving the value stored in a variable
- User input
  - `read`: a command telling the shell to read strings from standard input
    - Makes scripts interactive (cf. scanf) and also gets values from files or pipes

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

#BOOK=$HOME/phonebook.data
BOOK=$PWD/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
```

- Control
  - Manages program flow: e.g., `if .. then.. else .. fi`, or `while`, `case`, and `for`
- Environment
  - Environment variables: allowing users to record “personalized settings” affecting variable programs
    - `HOME`: contains a path to your home directory
    - `PATH`: contains paths that a user registers for convenience to run user-defined and system programs

Command-Line Parsing: Writing `smsh1`

- Program Logic of `smsh1`



< A shell with signals, exit, and parsing >

  - The modified shell can accept the following:

    ```
    find /home -name core -mtime +3 -print
    ```

    - "Find the files modified 3 days ago and print their filenames if 'core' is contained in the filenames"

[Remind] Parsing Strings

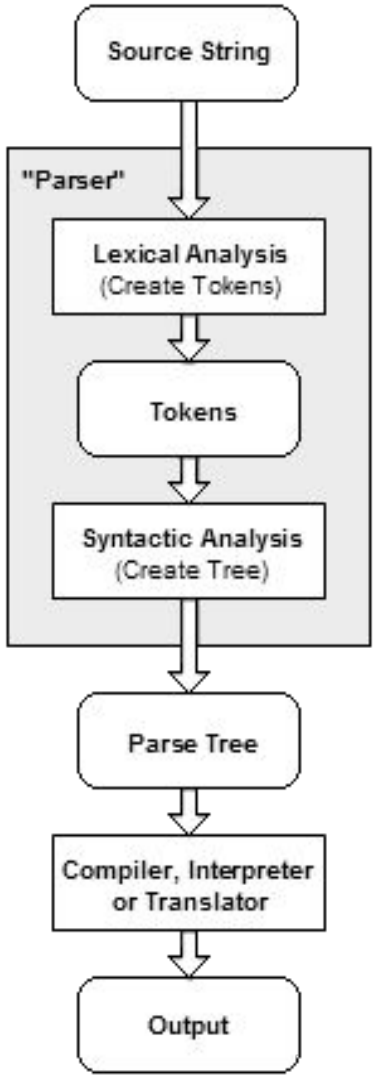
- Parsing
  - Dividing a string into tokens based on the given delimiters
- Token
  - One piece of information, a “word”
- Delimiter (구획 문자)
  - One (or more) characters used to separate tokens

- Example in Java
  - There are seven tokens: the, music, made, it, hard, to, concentrate

```
String phrase = "the music made   it   hard   to
concentrate";
String delims = "[ ]+";
String[] tokens = phrase.split(delims);
```

[Remind] Parsing and Parser

- Parsing
  - The process of analyzing a string of symbols
  - Comes from Latin *pars*, meaning part
- Parser
  - A software component that takes input data and builds a data structure
  - Lexical analysis (어휘 분석)
  - Syntactic analysis (구문 분석)



< Flow of data in a typical parser >

`smsh1`: Command-Line Parsing Support

- Three functions in the main function
  - `next_cmd`
    - Reads the next command from an input stream
    - It calls malloc to accept command lines of any length
  - `splitline`
    - Splits a string into an array of words and returns that array
    - It calls malloc to accept command lines with any number of arguments
  - `execute`
    - Uses fork, execvp, and wiat to run the command
    - This returns the termination status of the command

```
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: Command-Line Parsing Support (cont.)

- `$ cc -o smsh1 smsh1.c splitline.c execute.c`
- `smsh.h`

```
// smsh.h -- declares function profiles for external references
#define YES 1
#define NO  0

/*
 * - Reads the next command from an input stream
 * - Calls malloc to accept command lines of any length
 */
char    *next_cmd();
/*
 * - Splits a string into an array of words
 * - Returns that array
 */
char    **splitline(char*);
// free the list returned by splitline
void    freelist(char**);
// extension of malloc
void    *emalloc(size_t);
// extension of realloc
void    *erealloc(void*, size_t);
// run a program
int     execute(char**);
// report an error
void    fatal(char*, char*, int);
// process
int     process(char**);
```

smsh1: Command-Line Parsing Support (cont.)

- smsh.c

```c
/**  smsh1.c  small-shell version 1
 **            first really useful version after prompting shell
 **            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;
}
```

```c
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);
}
```

smsh1: Command-Line Parsing Support (cont.)

- execute.c

```c
/* 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>
```

```c
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;
}
```

smsh1: Command-Line Parsing Support (cont.)

- splitline.c

```c
/* 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"
```

```c
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           */

        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`: Command-Line Parsing Support (cont.)

- `splitline.c` (cont.)

```c
/**
 **     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;

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

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

`smsh1`: Command-Line Parsing Support (cont.)

- `splitline.c` (cont.)

```c
/*
 * 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);
}
```

```c
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;
}
```

`smsh1`: Command-Line Parsing Support (cont.)

- Execution
  - ‘`ps -f`’ is a child of `smsh1`, which is a child of bash

```
dynam@DESKTOP-Q4IJBP7:~/lab10$ make
cc -o smsh1 smsh1.c splitline.c execute.c
dynam@DESKTOP-Q4IJBP7:~/lab10$ ./smsh1
> ps -f
UID        PID  PPID  C STIME TTY          TIME CMD
dynam       10     9  0 Nov09 pts/0    00:00:00 -bash
dynam      355    10  0 00:07 pts/0    00:00:00 ./smsh1
dynam      356   355  0 00:07 pts/0    00:00:00 ps -f
>
```

Notes on `smsh1` – Need of Additional Conveniences (Not yet supported in `smsh1`)

- Multiple commands on a line
  - The regular shell allows the user to separate commands with semicolons, allowing the user to type several commands on one line
    - `ls demodir; ps -f; date`
- Background processing
  - The regular shell allows a user to run a process in the background (in non-blocking mode) by ending the command with an '&', as in
    - "Running in a process in the background": you start it, the prompt returns at once, and the process continues to run while you use the shell for other commands
    - & (ampersand): the **and** sign
- An exit command
  - The regular shell allows the user to type exit to quit from the shell

Control Flow (`if..then`) in the Shell

- What `if` does
  - The shell provides an `if` control structure
    - "We plan to back up our disk every Friday."

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

    - The `grep` program calls `exit(0)` to indicate success if it would find "Fri"
      - An exit value of 0 is signified for success
    - An `else` block can be added to the script. It's like the `then` block.
      - Check its syntax

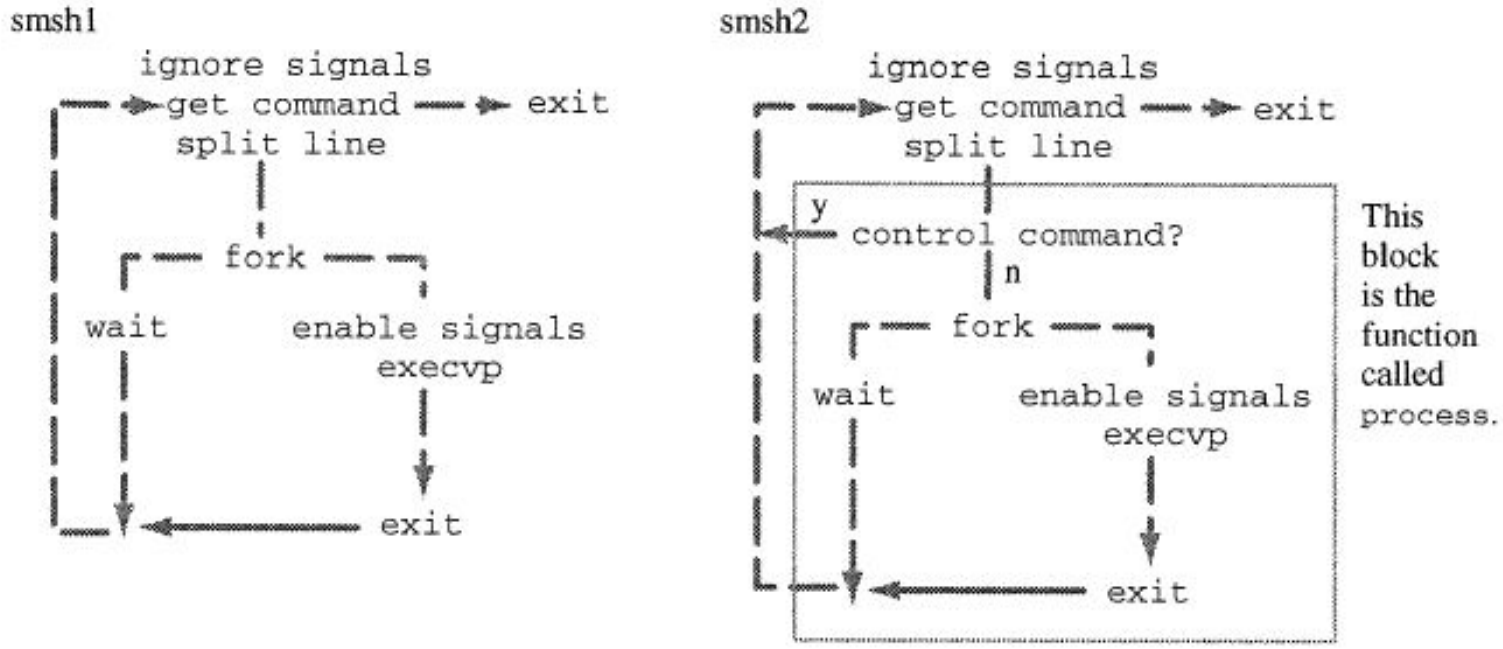
Control Flow (`if..then`) in the Shell (cont.)

```
if diff file1 file1.bak
then
     echo no differences found, removing backup
     rm file1.bak
else
     echo backup differs, making it read-only
     chmod -w file1.bak
fi
```

- How `if` works
  - (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

The Program Logic of `smsh2`

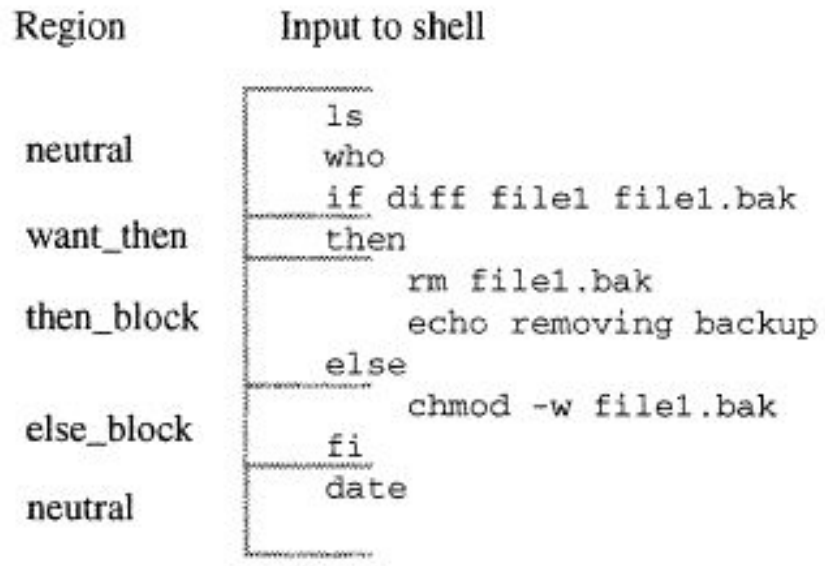
- Adding a new layer, or **`process`**
- Adds the `if` syntax and thus changes the logic of `smsh1`



< Adding flow control commands to `smsh` >

What `process` Does and How It works

- It “manages the control flow” of a script by watching for keywords like `if`, `then`, and `fi`, by calling `fork` and `exec` only when appropriate
- It views the script as a sequence of different regions
  - The `want_then` block
  - The `then` block
  - The `else` block
  - The `neural` block (outside the if structure)



< A script consists of different regions >

smsh2: Adding the Control Structure

- smsh2.c
  - Based on smsh1.c

    has only one change

```c
/** 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;
}
```

```c
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);
}
```

smsh2: Adding the Control Structure (cont.)

- process.c

```c
/* 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();
```

```c
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: Adding the Control Structure (cont.)

- controlflow.c

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

```c
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);
}
```

smsh2: Adding the Control Structure (cont.)

- controlflow.c (cont.)

```c
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;
}
```

```c
int do_control_command(char **args)
/*
 * purpose: Process "if", "then", "fi" - change state or detect error
 * returns: 0 if ok, -1 for syntax error
 *   notes: I would have put returns all over the place, Barry says "no"
 */
{
        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;
}
```

`smsh2`: Adding the Control Structure (cont.)

- Execution

```
dynam@DESKTOP-Q4IJBP7:~/lab10$ make
cc -o smsh2 smsh2.c splitline.c execute.c process.c controlflow.c
dynam@DESKTOP-Q4IJBP7:~/lab10$ ./smsh2
> grep lp /etc/passwd
lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin
cups-pk-helper:x:117:124:user for cups-pk-helper service,,,:/home/cups-pk-helper:/usr/sbin/nologin
> if grep lp /etc/passwd
lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin
cups-pk-helper:x:117:124:user for cups-pk-helper service,,,:/home/cups-pk-helper:/usr/sbin/nologin
> then
> echo ok
ok
> fi
> echo ok
ok
>
```

Shell Variables: Local and Global

- Using Shell 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
$
```

Shell Variables: Local and Global (cont.)

- Two types of variables
  - Local variables: works only for a user and within its current terminal
  - Environment (or global) variables
    - Their values are accessible to “all child processes” of the shell
  - e.g., `set | more`

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`	

Storage System for Variables

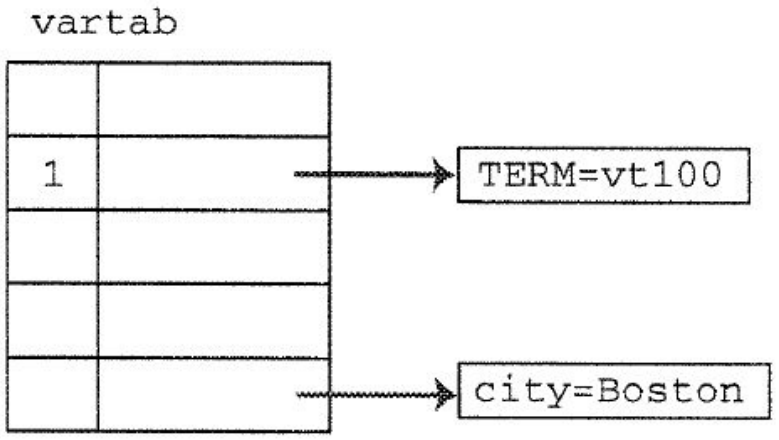
- How can the shell remember these variables?
  - The shell needs a “place” to store these names and values.
  - This storage system must distinguish local variables from global ones
- A possible model for the storage system

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

Storage System for Variables (cont.)

- Interface (function)
  - `VLstore(char *var, char *val)` adds/updates `var=val`
  - `VLlookup(char *var)` retrieves value for var
  - `VList` lists table to stdout
- Data structure for a table
  - Could be a linked list, a hash table, a tree, but an array of structs for now!

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



< A storage system for shell variables >

Adding Variable Commands: Built-ins

- We need to add the *assign*, *list*, and *retrieve commands* to our shell;

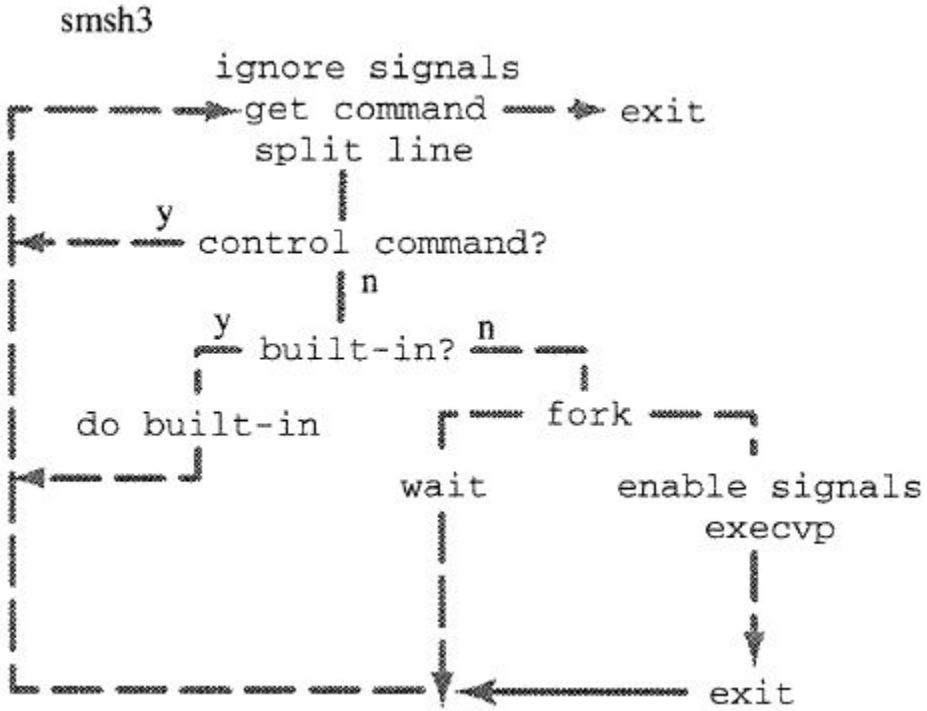
  hopefully, we can do the following: (not yet implemented)

```
dynam@DESKTOP-Q4IJBP7:~/lab10$ ./smsh2
> TERM=xterm
cannot execute command: No such file or directory
> set
cannot execute command: No such file or directory
> echo $TERM
$TERM
```

  - `set`: a "command" to our shell, not a program the shell runs
    - That is, `set` is different than a regular command like '`ls`'
  - To distinguish `set` from commands that the shell runs with `exec`,
    - `set` should be treated as *built-in* commands

Adding Variable Commands: Built-ins (cont.)

- Updated flow: `smsh3`



< Adding **built-in commands** to `smsh` >

smsh3: Adding Built-in Commands

- builtin.c

```c
/* 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 *);
```

```c
int builtin_command(char **args, int *resultp)
/*
 * purpose: run a builtin command
 * returns: 1 if args[0] is builtin, 0 if not
 * details: test args[0] against all known builtins.  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;
}
```

smsh3: Adding Built-in Commands (cont.)

- builtin.c (cont.)

```c
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 retores it to normal
 */
{
        char    *cp;
        int     rv ;

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

```c
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 */
}
```

smsh3: Adding Built-in Commands (cont.)

- varlib.h

```
// *varlib.h*
// declares environment-related functions

// keep val associated with var
int VLstore(char* name, char* val);
// return value of var
char* VLlookup(char* name);
// adds name to list of env vars
int VLexport(char* name);
// performs the shell's set command
void VLlist();
// copy from environ to table
int VLenviron2table(char* env[]);
// copy from table to environ
char**  VLtable2environ();
```

smsh3: Adding Built-in Commands (cont.)

- varlib.c

```
/* varlib.c
 *
 * a simple storage system to store name=value pairs
 * with facility to mark items as part of the environment
 *
 * interface:
 *     VLstore( name, value )    returns 1 for 0k, 0 for no
 *     VLlookup( name )          returns string or NULL if not there
 *     VLlist()                  prints out current table
 *
 * environment-related functions
 *     VLexport( name )          adds name to list of env vars
 *     VLtable2environ()         copy from table to environ
 *     VLenviron2table()         copy from environ to table
 *
 * details:
 *      the table is stored as an array of structs that
 *      contain a flag for `global' and a single string of
 *      the form name=value.  This allows EZ addition to the
 *      environment.  It makes searching pretty easy, as
 *      long as you search for "name="
 *
 */
```

```
#include        <stdio.h>
#include        <stdlib.h>
#include        "varlib.h"
#include        <string.h>

#define MAXVARS 200             /* a linked list would be nicer */

struct var {
                char *str;              /* name=val string      */
                int  global;            /* a boolean            */
        };

static struct var tab[MAXVARS];                 /* the table    */

static char *new_string( char *, char *);       /* private methods      */
static struct var *find_item(char *, int);
```

smsh3: Adding Built-in Commands (cont.)

- varlib.c (cont.)

```
int VLstore( char *name, char *val )
/*
 * traverse list, if found, replace it, else add at end
 * since there is no delete, a blank one is a free one
 * return 1 if trouble, 0 if ok (like a command)
 */
{
        struct var *itemp;
        char    *s;
        int     rv = 1;

        /* find spot to put it              and make new string */
        if ((itemp=find_item(name,1))!=NULL && (s=new_string(name,val))!=NULL)
        {
                if ( itemp->str )               /* has a val?   */
                        free(itemp->str);       /* y: remove it */
                itemp->str = s;
                rv = 0;                         /* ok! */
        }
        return rv;
}
```

```
char * new_string( char *name, char *val )
/*
 * returns new string of form name=value or NULL on error
 */
{
        char    *retval;

        retval = malloc( strlen(name) + strlen(val) + 2 );
        if ( retval != NULL )
                sprintf(retval, "%s=%s", name, val );
        return retval;
}
```

smsh3: Adding Built-in Commands (cont.)

- varlib.c (cont.)

```c
static struct var * find_item( char *name , int first_blank )
/*
 * searches table for an item
 * returns ptr to struct or NULL if not found
 * OR if (first_blank) then ptr to first blank one
 */
{
        int     i;
        int     len = strlen(name);
        char    *s;

        for( i = 0 ; i<MAXVARS && tab[i].str != NULL ; i++ )
        {
                s = tab[i].str;
                if ( strncmp(s,name,len) == 0 && s[len] == '=' ){
                        return &tab[i];
                }
        }
        if ( i < MAXVARS && first_blank )
                return &tab[i];
        return NULL;
}
```

```c
char * VLlookup( char *name )
/*
 * returns value of var or empty string if not there
 */
{
        struct var *itemp;

        if ( (itemp = find_item(name,0)) != NULL )
                return itemp->str + 1 + strlen(name);
        return "";

}

int VLexport( char *name )
/*
 * marks a var for export, adds it if not there
 * returns 1 for no, 0 for ok
 */
{
        struct var *itemp;
        int     rv = 1;

        if ( (itemp = find_item(name,0)) != NULL ){
                itemp->global = 1;
                rv = 0;
        }
        else if ( VLstore(name, "") == 1 )
                rv = VLexport(name);
        return rv;
}
```

smsh3: Adding Built-in Commands (cont.)

- varlib.c (cont.)

```c
void VLlist()
/*
 * performs the shell's  'set'  command
 * Lists the contents of the variable table, marking each
 * exported variable with the symbol  '*'
 */
{
        int     i;
        for(i = 0 ; i<MAXVARS && tab[i].str != NULL ; i++ )
        {
                if ( tab[i].global )
                        printf("  * %s\n", tab[i].str);
                else
                        printf("    %s\n", tab[i].str);
        }
}
```

```c
int VLenviron2table(char *env[])
/*
 * initialize the variable table by loading array of strings
 * return 1 for ok, 0 for not ok
 */
{
        int     i;
        char    *newstring;

        for(i = 0 ; env[i] != NULL ; i++ )
        {
                if ( i == MAXVARS )
                        return 0;
                newstring = malloc(1+strlen(env[i]));
                if ( newstring == NULL )
                        return 0;
                strcpy(newstring, env[i]);
                tab[i].str = newstring;
                tab[i].global = 1;
        }
        while( i < MAXVARS ){           /* I know we don't need this    */
                tab[i].str = NULL ;     /* static globals are nulled    */
                tab[i++].global = 0;    /* by default                   */
        }
        return 1;
}
```

smsh3: Adding Built-in Commands (cont.)

- varlib.c (cont.)

```
char ** VLtable2environ()
/*
 * build an array of pointers suitable for making a new environment
 * note, you need to free() this when done to avoid memory leaks
 */
{
        int     i,                      /* index                        */
                j,                      /* another index                */
                n = 0;                  /* counter                      */
        char    **envtab;               /* array of pointers            */

        /*
         * first, count the number of global variables
         */

        for( i = 0 ; i<MAXVARS && tab[i].str != NULL ; i++ )
                if ( tab[i].global == 1 )
                        n++;

        /* then, allocate space for that many variables */
        envtab = (char **) malloc( (n+1) * sizeof(char *) );
        if ( envtab == NULL )
                return NULL;

        /* then, load the array with pointers           */
        for(i = 0, j = 0 ; i<MAXVARS && tab[i].str != NULL ; i++ )
                if ( tab[i].global == 1 )
                        envtab[j++] = tab[i].str;
        envtab[j] = NULL;
        return envtab;
}
```

smsh3: Adding Built-in Commands (cont.)

- process2.c

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

/* process2.c   - version 2 - supports builtins
 * 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)
 */

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

```c
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() )
                if ( !builtin_command(args,&rv) )
                        rv = execute(args);
        return rv;
}
```

`smsh3`: Adding Built-in Commands (cont.)

- Execution
  - Not interpreted: variable substitution needed

```
dynam@DESKTOP-Q4IJBP7:~/lab10$ ./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
Thu Nov 10 02:20:46 KST 2022
> echo $temp, $day
$temp, $day
>
```

Environment Variables

- Unix/Linux lets users store preferences in a set of variables called the ***environment***
  - Each user has a unique home directory, username, file for incoming mail, the terminal type (`pterm`, `xterm`…), and favorite editor.
    - `pterm`, `xterm`: a type of terminal emulator for X window system
      - The X Window System (X11, or simply X) is a windowing system for bitmap displays, common on Unix-like operating systems. (source: Wiki)
      - X provides the basic framework for a GUI environment: drawing and moving windows on the display device and interacting with a mouse and keyboard. (source: Wiki)
  - Many customized settings are kept in “environment variables”.
    - The settings can be referenced when scripts utilizing the settings are run for convenience.

1) Using the Environment

- Listing your environment
  - Shows all the settings in the environment `env`
- Updating the environment
  - *`var=value`*
  - `export`
    - a built-in command to add a new variable
  - These can be combined:

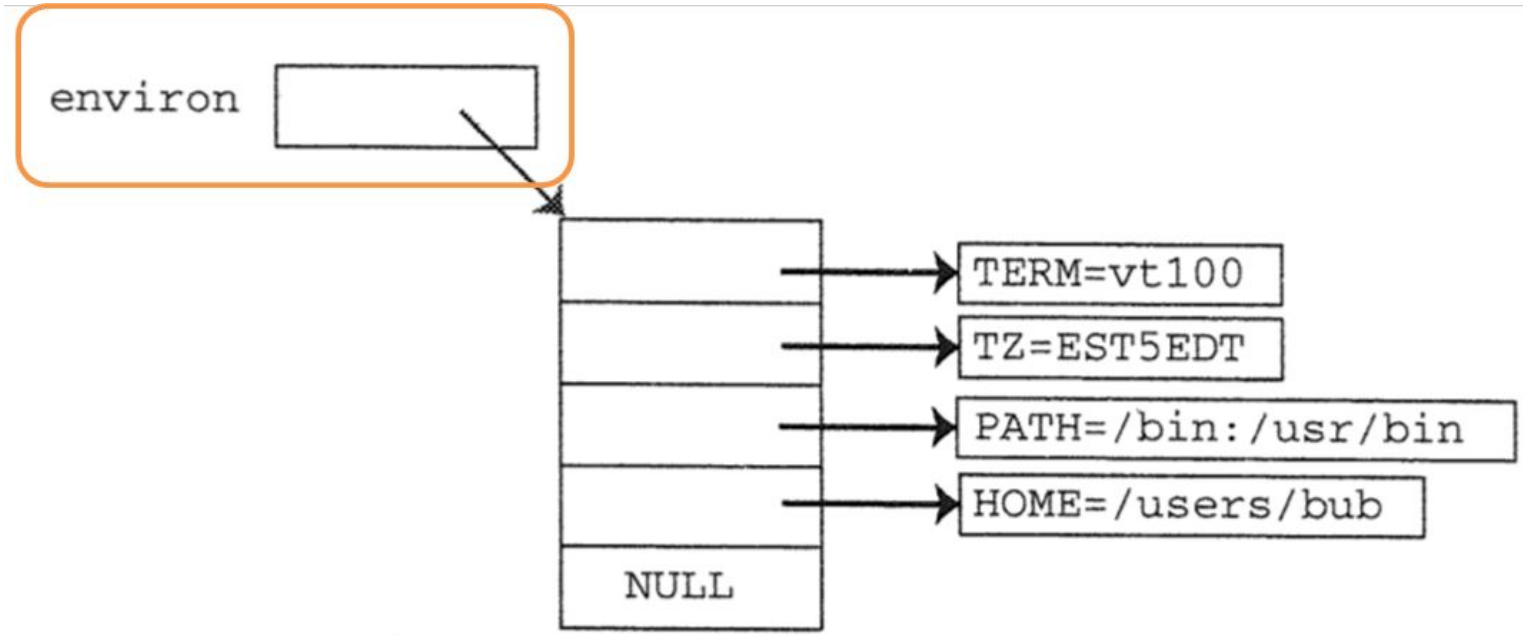
    `export` *`var=value`*
- Reading the environment
  - `getenv`: C library function

```
$ 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
```

```c
// readenv.c
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
int main(){
  char* cp = getenv("LANG");
  if(cp!=NULL && strcmp(cp,"ko_KR.UTF-8") == 0)
        printf("안녕하세요\n");
  else    printf("Hello\n");
}
```

1) Using the Environment (cont.)

- What is the environment? How it works?
  - The environment (`environ`) is an array of pointers to strings.



< The environment is an array of pointers to strings >

2) Showing the Environment

- `showenv.c`

```c
/*  showenv.c - shows how to read and print the environment
 */

extern char     **environ;    /* points to the array of strings */

main()
{
        int     i;

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

2) Showing the Environment

- Execution

```
dynam@DESKTOP-Q4IJBP7:~/lab10$ ./showenv
SHELL=/bin/bash
WSL_DISTRO_NAME=Ubuntu
WT_SESSION=37dfcce5-ff0d-4009-bb93-1fce1c3121e5
NAME=DESKTOP-Q4IJBP7
PWD=/home/dynam/lab10
LOGNAME=dynam
MOTD_SHOWN=update-motd
HOME=/home/dynam
LANG=C.UTF-8
WSL_INTEROP=/run/WSL/9_interop
LS_COLORS=rs=0:di=01;34:ln=01;36:mh=00:pi=40;33:so=01;35:do=01;35:bd=40;33;01:cd=40;33;01:or=40;31;01:mi=00:su=37
;41:sg=30;43:ca=30;41:tw=30;42:ow=34;42:st=37;44:ex=01;32:*.tar=01;31:*.tgz=01;31:*.arc=01;31:*.arj=01;31:*.taz=0
1;31:*.lha=01;31:*.lz4=01;31:*.lzh=01;31:*.lzma=01;31:*.tlz=01;31:*.txz=01;31:*.tzo=01;31:*.t7z=01;31:*.zip=01;31
:*.z=01;31:*.dz=01;31:*.gz=01;31:*.lrz=01;31:*.lz=01;31:*.lzo=01;31:*.xz=01;31:*.zst=01;31:*.tzst=01;31:*.bz2=01;
31:*.bz=01;31:*.tbz=01;31:*.tbz2=01;31:*.tz=01;31:*.deb=01;31:*.rpm=01;31:*.jar=01;31:*.war=01;31:*.ear=01;31:*.s
ar=01;31:*.rar=01;31:*.alz=01;31:*.ace=01;31:*.zoo=01;31:*.cpio=01;31:*.7z=01;31:*.rz=01;31:*.cab=01;31:*.wim=01;
31:*.swm=01;31:*.dwm=01;31:*.esd=01;31:*.jpg=01;35:*.jpeg=01;35:*.mjpg=01;35:*.mjpeg=01;35:*.gif=01;35:*.bmp=01;3
5:*.pbm=01;35:*.pgm=01;35:*.ppm=01;35:*.tga=01;35:*.xbm=01;35:*.xpm=01;35:*.tif=01;35:*.tiff=01;35:*.png=01;35:*.
svg=01;35:*.svgz=01;35:*.mng=01;35:*.pcx=01;35:*.mov=01;35:*.mpg=01;35:*.mpeg=01;35:*.m2v=01;35:*.mkv=01;35:*.web
m=01;35:*.ogm=01;35:*.mp4=01;35:*.m4v=01;35:*.mp4v=01;35:*.vob=01;35:*.qt=01;35:*.nuv=01;35:*.wmv=01;35:*.asf=01;
35:*.rm=01;35:*.rmvb=01;35:*.flc=01;35:*.avi=01;35:*.fli=01;35:*.flv=01;35:*.gl=01;35:*.dl=01;35:*.xcf=01;35:*.xw
d=01;35:*.yuv=01;35:*.cgm=01;35:*.emf=01;35:*.ogv=01;35:*.ogx=01;35:*.aac=00;36:*.au=00;36:*.flac=00;36:*.m4a=00;
36:*.mid=00;36:*.midi=00;36:*.mka=00;36:*.mp3=00;36:*.mpc=00;36:*.ogg=00;36:*.ra=00;36:*.wav=00;36:*.oga=00;36:*.
opus=00;36:*.spx=00;36:*.xspf=00;36:
LESSCLOSE=/usr/bin/lesspipe %s %s
TERM=xterm-256color
LESSOPEN=| /usr/bin/lesspipe %s
USER=dynam
DISPLAY=localhost:0.0
SHLVL=1
WSLENV=WT_SESSION::WT_PROFILE_ID
XDG_DATA_DIRS=/usr/local/share:/usr/share:/var/lib/snapd/desktop
PATH=/home/dynam/.local/bin:/home/dynam/.cargo/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/
usr/games:/usr/local/games:/usr/lib/wsl/lib:/mnt/c/Program Files/WindowsApps/Microsoft.WindowsTerminal_1.15.2874.
0_x64__8wekyb3d8bbwe:/mnt/c/Windows/system32:/mnt/c/Windows:/mnt/c/Windows/System32/Wbem:/mnt/c/Windows/System32/
WindowsPowerShell/v1.0/:/mnt/c/Windows/System32/OpenSSH/:/mnt/c/Program Files (x86)/NVIDIA Corporation/PhysX/Comm
on:/mnt/c/Users/user/AppData/Local/Microsoft/WindowsApps:/mnt/c/Program Files/Bandizip/:/mnt/c/Users/user/AppData
/Local/Programs/Microsoft VS Code/bin:/mnt/c/Program Files (x86)/Vim/vim90:/snap/bin
HOSTTYPE=x86_64
WT_PROFILE_ID={61c54bbd-c2c6-5271-96e7-009a87ff44bf}
_=./showenv
OLDPWD=/home/dynam
```

```
dynam@DESKTOP-Q4IJBP7:~/lab10$ env
SHELL=/bin/bash
WSL_DISTRO_NAME=Ubuntu
WT_SESSION=37dfcce5-ff0d-4009-bb93-1fce1c3121e5
NAME=DESKTOP-Q4IJBP7
PWD=/home/dynam/lab10
LOGNAME=dynam
MOTD_SHOWN=update-motd
HOME=/home/dynam
LANG=C.UTF-8
WSL_INTEROP=/run/WSL/9_interop
LS_COLORS=rs=0:di=01;34:ln=01;36:mh=00:pi=40;33:so=01;35:do=01;35:bd=40;33;01:cd=40;33;01:or=40;31;01:mi=00:su=37
;41:sg=30;43:ca=30;41:tw=30;42:ow=34;42:st=37;44:ex=01;32:*.tar=01;31:*.tgz=01;31:*.arc=01;31:*.arj=01;31:*.taz=0
1;31:*.lha=01;31:*.lz4=01;31:*.lzh=01;31:*.lzma=01;31:*.tlz=01;31:*.txz=01;31:*.tzo=01;31:*.t7z=01;31:*.zip=01;31
:*.z=01;31:*.dz=01;31:*.gz=01;31:*.lrz=01;31:*.lz=01;31:*.lzo=01;31:*.xz=01;31:*.zst=01;31:*.tzst=01;31:*.bz2=01;
31:*.bz=01;31:*.tbz=01;31:*.tbz2=01;31:*.tz=01;31:*.deb=01;31:*.rpm=01;31:*.jar=01;31:*.war=01;31:*.ear=01;31:*.s
ar=01;31:*.rar=01;31:*.alz=01;31:*.ace=01;31:*.zoo=01;31:*.cpio=01;31:*.7z=01;31:*.rz=01;31:*.cab=01;31:*.wim=01;
31:*.swm=01;31:*.dwm=01;31:*.esd=01;31:*.jpg=01;35:*.jpeg=01;35:*.mjpg=01;35:*.mjpeg=01;35:*.gif=01;35:*.bmp=01;3
5:*.pbm=01;35:*.pgm=01;35:*.ppm=01;35:*.tga=01;35:*.xbm=01;35:*.xpm=01;35:*.tif=01;35:*.tiff=01;35:*.png=01;35:*.
svg=01;35:*.svgz=01;35:*.mng=01;35:*.pcx=01;35:*.mov=01;35:*.mpg=01;35:*.mpeg=01;35:*.m2v=01;35:*.mkv=01;35:*.web
m=01;35:*.ogm=01;35:*.mp4=01;35:*.m4v=01;35:*.mp4v=01;35:*.vob=01;35:*.qt=01;35:*.nuv=01;35:*.wmv=01;35:*.asf=01;
35:*.rm=01;35:*.rmvb=01;35:*.flc=01;35:*.avi=01;35:*.fli=01;35:*.flv=01;35:*.gl=01;35:*.dl=01;35:*.xcf=01;35:*.xw
d=01;35:*.yuv=01;35:*.cgm=01;35:*.emf=01;35:*.ogv=01;35:*.ogx=01;35:*.aac=00;36:*.au=00;36:*.flac=00;36:*.m4a=00;
36:*.mid=00;36:*.midi=00;36:*.mka=00;36:*.mp3=00;36:*.mpc=00;36:*.ogg=00;36:*.ra=00;36:*.wav=00;36:*.oga=00;36:*.
opus=00;36:*.spx=00;36:*.xspf=00;36:
LESSCLOSE=/usr/bin/lesspipe %s %s
TERM=xterm-256color
LESSOPEN=| /usr/bin/lesspipe %s
USER=dynam
DISPLAY=localhost:0.0
SHLVL=1
WSLENV=WT_SESSION::WT_PROFILE_ID
XDG_DATA_DIRS=/usr/local/share:/usr/share:/var/lib/snapd/desktop
PATH=/home/dynam/.local/bin:/home/dynam/.cargo/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/
usr/games:/usr/local/games:/usr/lib/wsl/lib:/mnt/c/Program Files/WindowsApps/Microsoft.WindowsTerminal_1.15.2874.
0_x64__8wekyb3d8bbwe:/mnt/c/Windows/system32:/mnt/c/Windows:/mnt/c/Windows/System32/Wbem:/mnt/c/Windows/System32/
WindowsPowerShell/v1.0/:/mnt/c/Windows/System32/OpenSSH/:/mnt/c/Program Files (x86)/NVIDIA Corporation/PhysX/Comm
on:/mnt/c/Users/user/AppData/Local/Microsoft/WindowsApps:/mnt/c/Program Files/Bandizip/:/mnt/c/Users/user/AppData
/Local/Programs/Microsoft VS Code/bin:/mnt/c/Program Files (x86)/Vim/vim90:/snap/bin
HOSTTYPE=x86_64
WT_PROFILE_ID={61c54bbd-c2c6-5271-96e7-009a87ff44bf}
_=/usr/bin/env
OLDPWD=/home/dynam
```

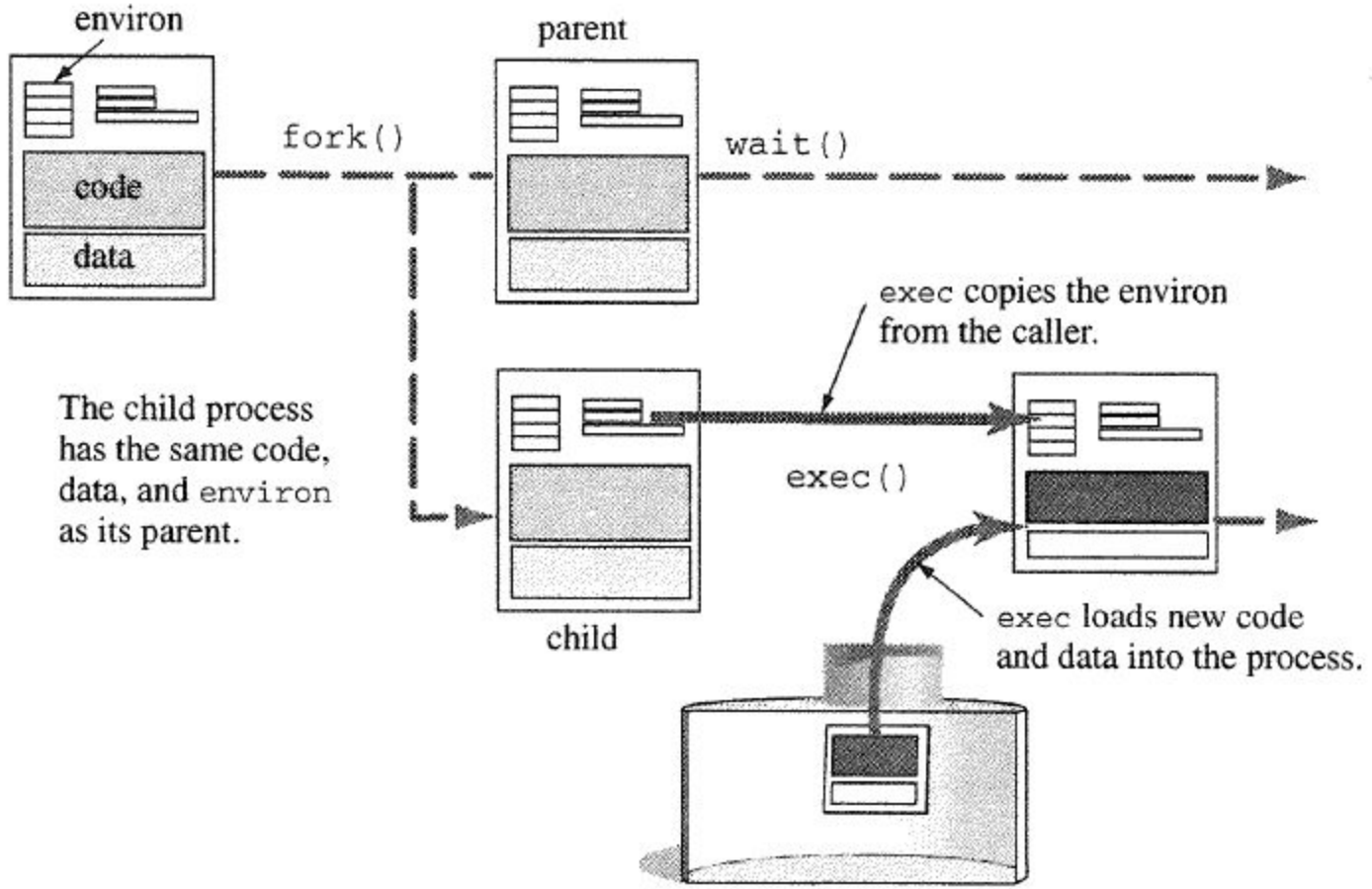
3) Changing the Environment

- `changeenv.c`: changes the environment and then runs env

```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;

int 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       */

        return 0;
}
```

```
dynam@DESKTOP-Q4IJBP7:~/lab10$ make
cc -o changeenv changeenv.c
dynam@DESKTOP-Q4IJBP7:~/lab10$ ./changeenv
TERM=vt100
HOME=/on/the/range
```

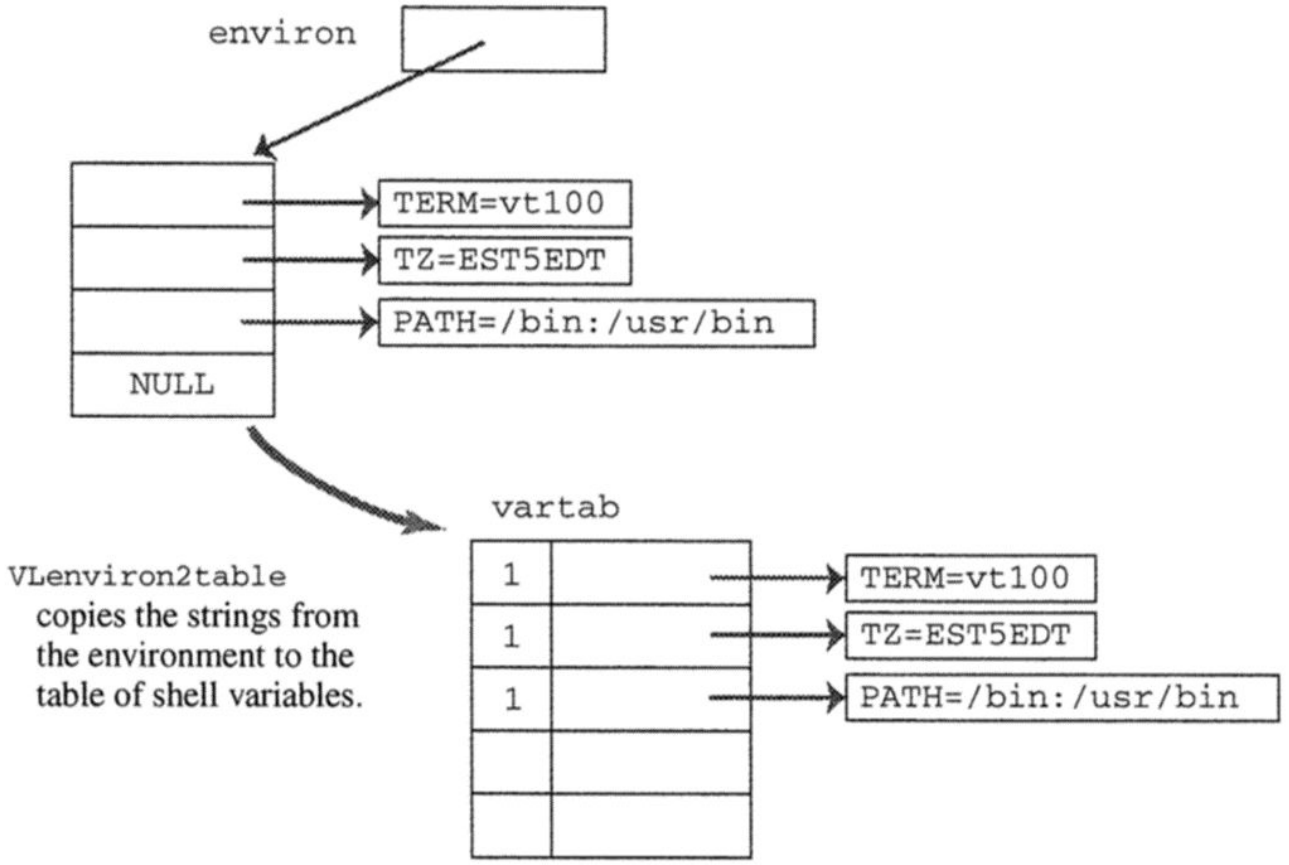
`exec` Wipes Out All Data! – Copied Environment



< Strings in `environ` are copied by `exec()` >

smsh4: Adding Environment Handling to smsh

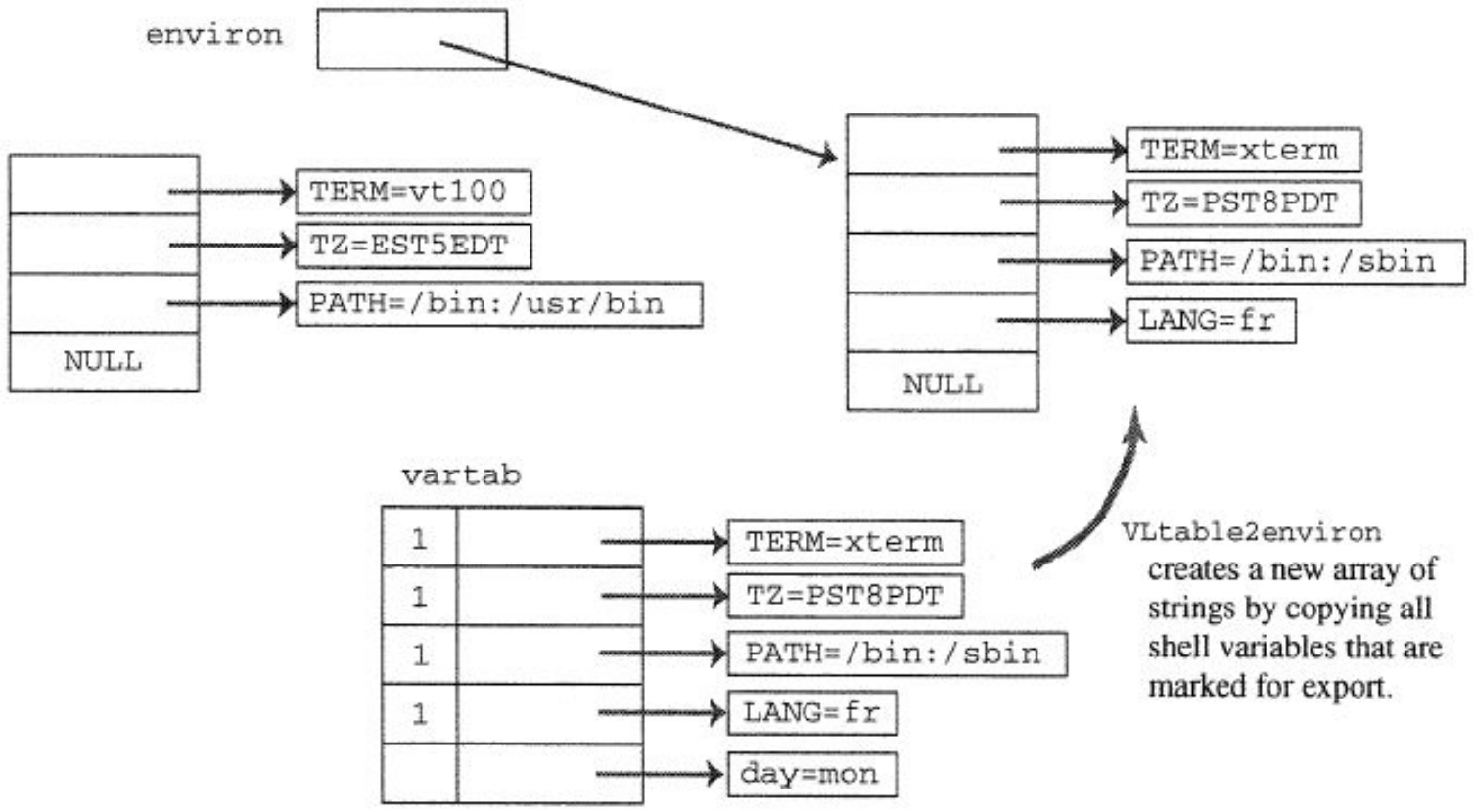
- Access to environment variables



< Copying values from the environment to vartab >

smsh4: Adding Environment Handling to smsh (cont.)

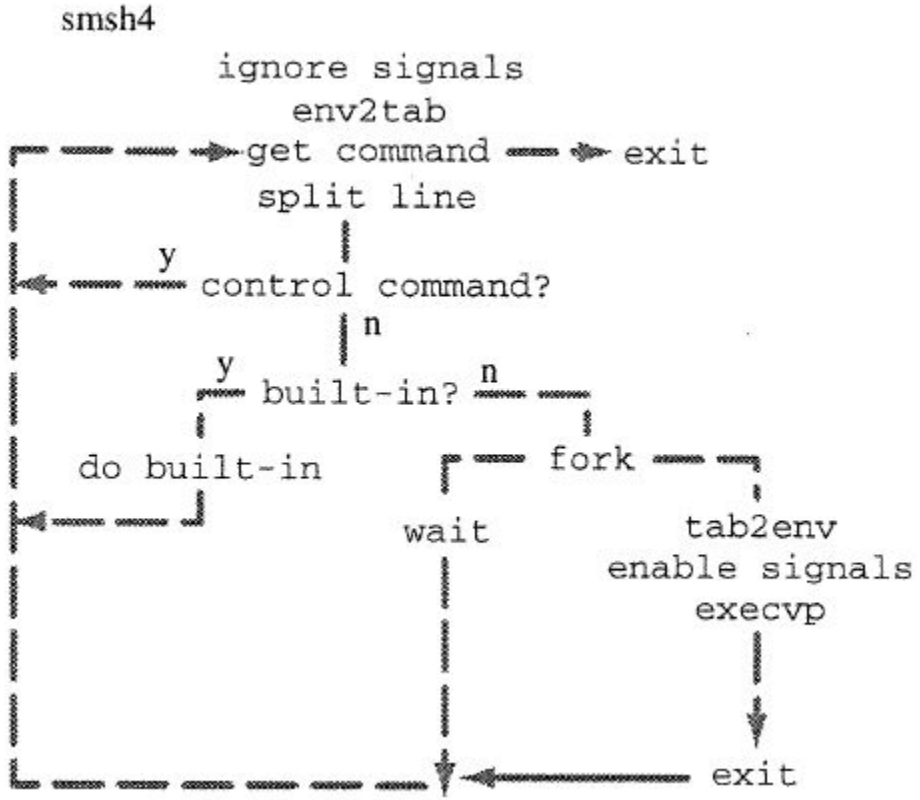
- Changing the environment



< Copying values from vartab to a new environment >

smsh4: Adding Environment Handling to smsh (cont.)

- Updated flow to smsh4



<Adding environment handling to smsh >

smsh4: Adding Environment Handling to smsh (cont.)

- Changes to smsh
  - setup() in smsh4.c (based on smsh2.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);
}
```

smsh4: Adding Environment Handling to smsh (cont.)

- Changes to smsh (cont.)
  - execute2.c based on execute.c

```c
#include        <stdio.h>
#include        <stdlib.h>
#include        <unistd.h>
#include        <signal.h>
#include        <sys/wait.h>
#include        "varlib.h"
/* execute2.c - includes environmment handling */

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
 */
{
        extern char **environ;
        int     pid ;
        int     child_info = -1;

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

        if ( (pid = fork())  == -1 )
                perror("fork");
        else if ( pid == 0 ){
                environ = VLtable2environ();
                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;
}
```

smsh4: Adding Environment Handling to smsh (cont.)

- Execution
  - “Update time zone (TZ) to Pacific Standard Time (PST)”

```
dynam@DESKTOP-Q4IJBP7:~/lab10$ make
cc -o smsh4 smsh4.c splitline.c execute2.c process2.c controlflow.c \
        builtin.c varlib.c
dynam@DESKTOP-Q4IJBP7:~/lab10$ ./smsh4
> date
Thu Nov 10 02:58:20 KST 2022
> TZ=PST
> export TZ
> date
Wed Nov  9 17:58:30 PST 2022
>
```

Summary

- A shell runs programs, called ***shell scripts***, which can run programs, accept user input, use variables, and follow complex logic
- The `if..then` logic in the shell depends on the convention that a program returns an exit value of 0 to indicate *success*
  - The shell uses `wait` to obtain the exit status from a program.
- The shell programming language includes variables
  - These variables store strings
  - These variables may be used in any command
  - Shell variables are “local” to the script

Summary (cont.)

- Every program inherits “a list of strings,” called ***environment***, from its calling process
  - The environment is used to
    - Define “global settings” for the session, and
    - Set “parameters” for specific programs
  - The shell allows users to “view and modify” the environment