```
:::::::::::::: script1 ::::::::::::::
#
# a shell script a a batch of commands
#
	echo "Status of system now. Includes users, files, processes"
	date
	who
	ls
	ps

:::::::::::::: script2 ::::::::::::::
#!/bin/sh
# shows a real program in the shell
#
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

:::::::::::::: script3 ::::::::::::::
#!/bin/sh
# a script to show how an environment variable is passed to commands
# TZ is timezone, affect things like date, and ls -l
#
echo "The time in Boston is"
	TZ=EST5EDT
	export TZ
	date
echo "The time in Chicago is"
	TZ=CST6CDT
	date
echo "The time in LA is"
	TZ=PST8PDT
	date

:::::::::::::: forkquiz1.c ::::::::::::::
#include	<stdio.h>
#include	<signal.h>
/*
 *	forkquiz1.c
 *		what does this program do in the following cases:
 *		a) user input, b) Ctrl-C, c) timeout
 */
main()
{
	int	child_status, retval;

	switch( fork() ) {
		case -1:
			perror("fork");
			exit(1);
		case  0:
			alarm(10);	/* what if this were before fork? */
			printf("Child here.  Exit with what value? ");
			scanf("%d", &retval);
			exit(retval);
		default:
			signal(SIGINT, SIG_IGN);
			signal(SIGQUIT, SIG_IGN);
			printf("parent waiting for child\n");
			wait( &child_status );
			printf("Status from child is:  %4d\n",  child_status);
			printf("              in hex:  %04x\n", child_status);
			printf("     as three fields: [xxxxxxxxcsssssss]\n");
			printf("                       %8d%d%7d\n",
						      child_status>>8,
						      (child_status>>7) & 1,
						      child_status & 0x7F
				);
			printf(" x: value from eXit(), ");
			printf("C: coredump flag, s: signal number\n");
	}
}
```

```
::::::::::::: smsh1.c :::::::::::::
#include       <stdio.h>
#include       <signal.h>
#include       "smsh.h"

/**
 **    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
 **/

#define MAXARG         20
#define MAXCMDLEN      512
#define DFL_PROMPT     "> "

main()
{
        char    *argv[MAXARG + 1], *prompt;
                cmdline[MAXCMDLEN];
        int     argc;

        prompt = DFL_PROMPT ;
        signal(SIGINT,  SIG_IGN);
        signal(SIGQUIT, SIG_IGN);

        while ( get_next_command( prompt, cmdline, stdin ) == TRUE ) {
                if ( splitline(cmdline, &argc, argv, MAXARG) == FALSE )
                        continue;
                execute( argc, argv );
        }
}
get_next_command( prompt, buffer, input_stream )
char *prompt, *buffer;
FILE *input_stream;
/*
 *  read next line from input_stream.  Return FALSE on EOF
 */
{
        printf("%s", prompt);                           /* prompt user */
        if ( fgets(buffer, MAXCMDLEN, input_stream) ){       /* get line    */
                buffer[strlen(buffer) - 1] = '\0';      /* remove nl   */
                return TRUE;                            /* say ok      */
        }
        return FALSE;                                   /* no more     */
}

execute(argc, argv)
char *argv[];
/*
 *      argv is all set to pass to execvp, and argc is nice to know about
 *      but execvp uses the terminating NULL in argv.
 */
{
        int     pid = fork(), child_info;

        if ( pid == -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");
        }
}
```

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

/**     splitline ( parse a line into an array of strings )  **/

splitline(cmdline, argcp, argv, max)
char    *cmdline;
int     *argcp;
char    *argv[];
/*
 *      cmdline has a string of white-space separated tokens
 *      put the addresses of the tokens in the array argv[]
 *      put their number in *argcp and do not put more than max
 *      in argv or suffer dire consequences!
 *      NOTE: this modifies cmdline
 *      returns FALSE on too many args or zero args.  TRUE for ok stuff
 */
{
        int i = 0, retval = FALSE ;
        char *cmdp = cmdline ;

        while ( i<=max ){                     /* still room?                 */
                while ( isspace( *cmdp ) )    /* skip leading space  */
                        cmdp++;
                if ( *cmdp == '\0' )          /* at end of string?   */
                        break;
                                              /* record string       */
                argv[i++] = cmdp ;            /* and bump counter    */
                                              /* move to end of word */
                while ( *++cmdp && *cmdp != ' ' && *cmdp != '\t' )
                        ;
                if ( *cmdp != '\0' )          /* past end of word    */
                        *cmdp++ = '\0';               /* terminate string    */
        }
        if ( i > max )
                printf("Too many args\n");
        else if ( i > 0 ){
                argv[i] = NULL ;              /* mark end of array   */
                *argcp = i;                   /* and store argc      */
                retval = TRUE ;                       /* say ok              */
        }
        return retval ;
}
```

```
:::::::::::::: smsh2.c ::::::::::::::
#include       <stdio.h>
#include       <signal.h>
#include       "smsh.h"

/**
 **    small-shell version 2
 **            a small step up from smsh1, it displays result from wait
 **            (the report) function is the new item here
 **/

#define MAXARG         20
#define MAXCMDLEN      512
#define DFL_PROMPT     "> "

main()
{
        char    *argv[MAXARG + 1], *prompt, cmdline[MAXCMDLEN];
        int     argc;

        prompt = DFL_PROMPT ;
        signal(SIGINT,  SIG_IGN);
        signal(SIGQUIT, SIG_IGN);

        while ( get_next_command( prompt, cmdline, stdin ) == TRUE ) {
                if ( splitline(cmdline, &argc, argv, MAXARG) == FALSE )
                        continue;
                execute( argc, argv );
        }
}

get_next_command( prompt, buffer, input_stream )
{
        /* same as in smsh1.c */
}

execute(argc, argv)
char *argv[];
/*
 *      argv is all set to pass to execvp, and argc is nice to know about
 *      but execvp uses the terminating NULL in argv.
 */
{
        int     pid = fork(), child_info;

        if ( pid == -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");
                else
                        report(child_info);
        }
}
report( int info )
/*
 * prints out result of child process
 */
{
        int     from_exit, core_flag, signal_num;

        signal_num = info & 0x7F ;
        core_flag  = (info >> 7 ) & 1;
        from_exit  = (info >> 8) ;

        if ( signal_num != 0 ){
                printf("\n[child died from signal %d]", signal_num);
                if ( core_flag ) printf(" (core dumped)" );
        }
        else
                printf("\n[child exited with code %d]", from_exit);
        putchar('\n');
}
```

```
:::::::::::::: smsh3.c ::::::::::::::
#include       <stdio.h>
#include       <signal.h>
#include       "smsh.h"
#include       "varlib.h"

/**
 **    small-shell version 3
 **            the first version with local variables (uses varlib.c)
 **            includes the = operator and the set command
 **            no ability to use these in commands, yet
 **/

#define MAXARG         20
#define MAXCMDLEN      512
#define DFL_PROMPT     "> "

main()
{
        char    *argv[MAXARG + 1], *prompt, cmdline[MAXCMDLEN];
        int     argc;

        prompt = DFL_PROMPT ;
        signal(SIGINT,  SIG_IGN);
        signal(SIGQUIT, SIG_IGN);

        while ( get_next_command( prompt, cmdline, stdin ) == TRUE ) {
                if ( splitline(cmdline, &argc, argv, MAXARG) == FALSE )
                        continue;
                if ( built_in_command( argc, argv ) == FALSE )
                        execute( argc, argv );
        }
}

built_in_command(int argc, char *argv[])
/*
 *      if a built-in, do it, else return FALSE
 */
{
        char    *cp;

        if ( (cp=strchr(argv[0], '=')) != NULL ){    /* var=val ?   */
                *cp = '\0';                           /* yes         */
                VLstore(argv[0], cp+1);                       /* add to vartab*/
                return TRUE;
        }
        if ( strcmp(argv[0], "set") == 0 ){           /* set command */
                VLset();                              /* y: do it    */
                return TRUE;
        }
        return FALSE;
}
get_next_command( prompt, buffer, input_stream )
{
        /* same as smsh1.c */
}
execute(argc, argv)
char *argv[];
/*
 *      argv is all set to pass to execvp, and argc is nice to know about
 *      but execvp uses the terminating NULL in argv.
 */
{
        /* same as smsh2.c */
}
report( int info )
/*
 * prints out result of child process
 */
{
        /* same as smsh2.c */
}
```

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

/*
 * 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
 *     VLset()                 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="
 *
 */

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

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

/** local vars: the table **/

static struct var tab[MAXVARS];

/** local functions **/

static char *new_string( char *, char *);
static struct var *find_item(char *, int);

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 0 if trouble, 1 if ok
 */
{
       struct var *itemp;
       char    *s;

       /* find spot to put it */
       if ( (itemp = find_item(name,1)) == NULL )
               return 0;

       /* if already there, then chuck old old value */
       if ( itemp->str )
               free(itemp->str);

       if ( ( s = new_string( name, val )) == NULL )        /* new name=val */
               return 0;                               /* no memory   */
       itemp->str = s;                                      /* store it    */
       return 1;
}
```

```
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;
}

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 0 for no, 1 for ok
 */
{
        struct var *itemp;

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

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;
}
```

```
void
VLset()
{
        int     i;
        for(i = 0 ; i<MAXVARS && tab[i].str != NULL ; i++ )
        {
                if ( tab[i].global )
                        printf("[E] %s\n", tab[i].str);
                else
                        printf("    %s\n", tab[i].str);
        }
}

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;
}

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          */

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

        /* go get an array for n+1 pointers          */
        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;
```

```
:::::::::::::: showenv.c ::::::::::::::

/*
 *      showenv.c
 *
 *              demonstrates how to read the entire list
 *              of environment variables
 */


        extern char     **environ;

main()
{
        int     i;

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

:::::::::::::: envchange.c ::::::::::::::

#include <stdio.h>
#include <malloc.h>

/*
 * envchange.c
 *         what: shows that the environment is SIMPLY the array pointed
 *                to by the global variable called environ
 *       method: change it to something else and then use getenv()
 */

char **environ;

main()
{
        char    *newenv[5];

        printf("The current environment is..\n");
        system( "showenv|more" );

        printf("\nPress return to continue..."); getchar();
        printf("***** Replacing entry 0 with MONTH=APRIL..\n"); getchar();
        environ[0] = "MONTH=April";
        system( "showenv|more" );

        printf("\nPress return to continue..."); getchar();
        printf("***** Now pointing environ to a new table..\n"); getchar();
        newenv[0] = "HOME=/on/the/range";
        newenv[1] = "LOGNAME=nobody";
        newenv[2] = "PATH=.:/bin:/usr/bin";
        newenv[3] = "DAY=Wednesday";
        newenv[4] = NULL ;
        environ = &newenv[0];          /* or environ = newenv */
        system( "showenv" );
}
```

```
:::::::::::::: smsh4.c ::::::::::::::
#include        <stdio.h>
#include        <signal.h>
#include        "smsh.h"
#include        "varlib.h"
/**
 **     small-shell version 4
 **             this one supports environment variables
 **             loads vars from environ, handles export, reloads environ before exec
 **/
#define MAXARG          20
#define MAXCMDLEN       512
#define DFL_PROMPT      "> "
extern char **environ;

main()
{
        char    *argv[MAXARG + 1], *prompt, cmdline[MAXCMDLEN];
        int     argc;

        signal(SIGINT,  SIG_IGN); signal(SIGQUIT, SIG_IGN);
        if ( VLenviron2table(environ) == 0 )
                exit(1);
        prompt = DFL_PROMPT ;

        while ( get_next_command( prompt, cmdline, stdin ) == TRUE )
                if ( splitline(cmdline, &argc, argv, MAXARG) == TRUE )
                        if ( built_in_command( argc, argv ) != TRUE )
                                execute( argc, argv );
}

built_in_command(int argc, char *argv[])
/*
 *      if a built-in, do it, else return FALSE
 */
{
        char    *cp;

        if ( (cp=strchr(argv[0], '=')) != NULL ){     /* var=val ?   */
                *cp = '\0';                           /* yes         */
                VLstore(argv[0], cp+1);                       /* add to vartab*/
                return TRUE;
        }
        if ( strcmp(argv[0], "set") == 0 ){           /* set command */
                VLset();                              /* y: do it    */
                return TRUE;
        }
        if ( strcmp(argv[0], "export") == 0 ){                /* export cmd? */
                VLexport(argv[1]);                    /* need argc   */
                return TRUE;                          /* check here  */
        }
        return FALSE;
}
get_next_command( prompt, buffer, input_stream )
{
        /* same as smsh1.c */
}
execute(argc, argv)
char *argv[];
/*
 *      fork(), then execvp(), using argv[], waits for child process
 */
{
        int     pid = fork(), child_info;

        if ( pid == -1 )
                perror("fork");
        else if ( pid == 0 ){
                signal(SIGINT, SIG_DFL); signal(SIGQUIT, SIG_DFL);
                environ = VLtable2environ();
                execvp(argv[0], argv);
                perror("cannot execute command");
                exit(1);
        }
        else {
                if ( wait( &child_info) == -1 )
                        perror("wait");
        }
}
```