CSI 402 – Systems Programming – Handout 14.5
A Simple Shell Program

Note: This handout shows the C source code for a very simple shell. This is simpler than the example discussed in Section 5.9 of the text by Haviland et al. This shell program repeatedly prompts the user for a shell command, reads the command typed by a user, creates a new process to execute the command and waits for the command to be completed.

The source code uses two functions, namely `getline` and `parse_line`, for which sources are not shown. The specifications for the functions are given below. Students should have no difficulty writing the C programs for these functions.

(a) `getline`: The prototype for this function is as follows:

```
int  getline (char line[])
```

This function returns the line typed by the user as a null terminated string in the parameter `line`. The function may assume that the space for the array `line` has already been allocated by the caller. The maximum number of characters on a line is given by the constant `LINE_MAX`. If the user types an empty line or `CTRL-D` (which indicates EOF for `stdin`) or a line that is longer than `LINE_MAX`, the return value of the function must be `-1`; otherwise, the return value of the function should be 0.

(b) `parse_line`: The prototype for this function is as follows:

```
void  parse_line (char line[], char *arg[])
```

The function should parse the command given by the parameter `line`, assuming that successive parameters of the command are separated by one or more spaces. Successive elements of the `arg` array must point to successive parameters of the command. The last element of `arg` must be the `NULL pointer`. The function may assume that the space for the array `arg` has already been allocated by the caller and that the maximum number of parts that a command may have is given by the constant `ARG_MAX`. (The size of `arg` is `ARG_MAX+1` to allow for the `NULL` pointer at the end.)

(c) **Important Note:** The shell shown in this handout does *not* handle the `cd` command correctly. Students are strongly advised to fix this problem.

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

/* A very simple shell. It accepts a command from the user, forks a child to execute */
/* the command and waits until the child exits.                                       */

/* Constants denoting the maximum length of an input line and the */
/* maximum number of arguments for a command.                     */
#define  LINE_MAX   255
#define  ARG_MAX    10
```

```
/* The prompt produced by the simple shell. */
#define   SH_PROMPT   "next> "

/* Prototypes for the functions used. Source code for getline and */
/* parse_line are not shown here.                                 */
int   run_command(char *[]);
int   getline(char *);               /* To be written by the student. */
void  parse_line(char *, char *[]);  /* To be written by the student. */

int main(void) {

  /* The main function reads a command, calls a function to read the input line,  */
  /* another function to parse it and a third function to execute the command.    */

  char line[LINE_MAX+1];  /* To store input line. */
  char *argbuf[ARG_MAX+1];  /* Additional is space needed for the NULL pointer. */
  int  temp;

  do {
    printf(SH_PROMPT); fflush(stdout);   /* Produce prompt. */
    if ((temp = getline(line)) != -1) {
       /* The user typed a command. Parse and execute it. */
       parse_line(line, argbuf);
       run_command(argbuf);
    }
  } while (temp != -1);
  return 0;
} /* End of main. */

int run_command(char *command[]) {

  /* Executes the given command by creating a child process. */

  pid_t  child;    /* Child pid returned by fork. */
  pid_t  c;        /* Pid of child to be returned by wait. */

  if ((child = fork()) == 0) {
     /* Child process. We want it to execute the program given by */
     /* the parameter command.                                    */
     execvp(command[0], command);
     /* If this point is reached, then execvp must have failed. */
     fprintf(stderr, "Child process could not do execvp.\n"); exit(1);
  }
  else { /* Parent process. */
     if (child == (pid_t)(-1)) {
        fprintf(stderr, "Fork failed.\n"); exit(1);
     }
     else {
        c = wait(NULL); /* Wait for child to complete. Ignore the exit status of child. */
        return 0;
     }
  }
} /* End of run_command. */
```