2. Implementation

Three core functions in the simple shell program are:

```
int shell_read_line(char *);
int get_line_args(char *, char **);
int shell_execute(char **, int);
```

Next, we will discuss their implementation one by one.

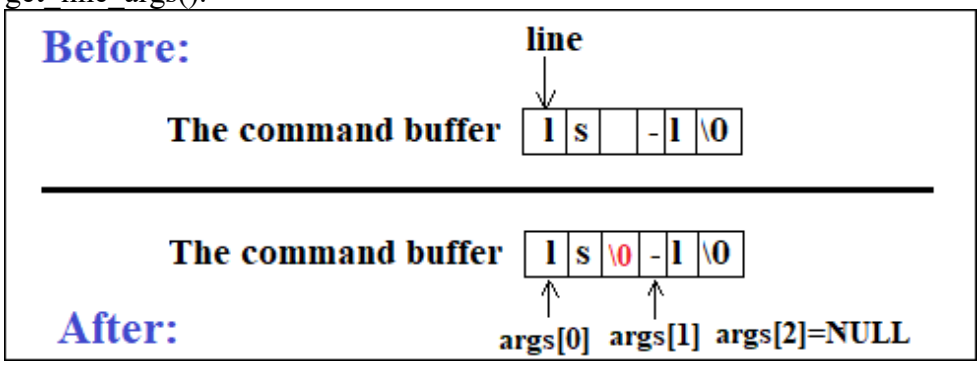
- shell_read_line(): As shown below, in shell_read_line(), we read characters one by one and store each into the command buffer (pointed by cmd_buf) until we found the input character is "\n" (that is the newline character); for "\n", we put "\0" (that is the null character) in the command buffer, then return.

```
int shell_read_line(char * cmd_buf)
{
	int position = 0;
	char c;

	while (1) {
		// Read one character each time
		c = getchar();

		// For newline, put a null character and return.
		if ( c == '\n') {
			cmd_buf[position] =
			'\0'; return position;
		} else { cmd_buf[position]
			= c; position++;


			//if too big, warning and return -1
			if (position >= MAX_LINE_SIZE) {
				printf("The command size is too big\n");
				return -1;
			}
		}
	}
}
```

- get_line_args(): As shown below, in get_line_args(), for the command line stored in the command buffer (pointed by line), we set up args[i] (that is used to record the beginning address of the *i*-th argument) and put "\0" at the end of the argument, by which we can use args[i] to refer a string (starting from args[i] and ending with "\0"). At the end of the command line (containing "\0"), we set up the last argument as NULL and return.

```
int get_line_args(char * line, char ** args)
{
	int start_position = 0;
	int end_position = 0;
	char c;
	int argc = 0;

	while (argc < MAX_ARG_NUM ){
		//Jump to the first non-space/tab char
		while(1){
			c= line[start_position];
			if ( c == ' ' || c == '\t'){
				start_position ++;
			}else{
				break;
			}
		}
		//Check if the end of string - if yes, return the argument as NULL; otherwise, find the argument
		if ( c == '\0'){
			args[argc] = NULL;
			argc++;
			return argc;
		}else{
			end_position = start_position;

			//Move end_position to the end of the argument
			while (1){
				end_position++;
				c= line[end_position];
				if ( c == ' ' || c == '\t' || c == '\0')
					break;
			}

			if( c != '\0'){
				line[end_position] = '\0';
				end_position++;
			}

			args[argc] = & line[start_position];
			argc ++;

			start_position = end_position;
		}

	}

	//Should never go here; Return -1 for error
	return -1;
}
```

An example with the input as “ls -l” is shown below to illustrate the input and output of get_line_args().



- shell_execute(): As shown below, in shell_execute(), we use fork() to generate a child process. In the child process, execvp() is used to execute the command using args as the pointer to all arguments, and the parent process attempts to wait for the child process and then continue for next input.

```
int shell_execute(char ** args, int argc)
{
        int child_pid, wait_return, status;

        if ( strcmp(args[0], "EXIT") == 0 )
                return -1;


        if( (child_pid = fork()) < 0 ){
                printf("fork() error \n");
        }else if (child_pid == 0 ){
                if ( execvp(args[0], args) < 0){
                        printf("execvp error \n");
                        exit(-1);
                }
        }else{
                if ( (wait_return = wait(&status) ) < 0 )
                        printf("wait error \n");
        }

        return 0;
}
```

The description of execvp() is listed below for your reference:

int execvp(const char *file, char *const argv[]);

Description:
The exec() family of functions replaces the current process image with a new process image. The execvp() function provides an array of pointers to null-terminated strings that represent the argument list available to the new program. The first argument, by convention, should point to the filename associated with the file being executed. The array of pointers must be terminated by a NULL pointer.