OPERATING SYSTEMS - LAB ASSIGNMENT REPORT

Student Name: Jerome Arsany Mansour Farah
Id: 2305093
Course Name: Operating Systems
Github Repository: https://github.com/jeromearsany/Backup-Script-and-Mini-Shell-Assignment.git

==

1. Introduction:

This report documents the successful completion of a two-part lab assignment designed to provide practical experience with fundamental operating systems concepts.

The first part of the assignment involved creating an automated backup utility using a Bash shell script. This task focused on shell scripting, file system management, and process automation.

The second part required the development of a simple command-line interpreter, or "mini-shell," using the C programming language. This task focused on core process management concepts, including process creation with `fork()`, program execution with `execvp()`, and process synchronization with `wait()`.

This document contains the full source code for both parts, along with screenshots demonstrating their correct execution and functionality.

==

2. Part 1: Automated Backup Script:

2.1. Objective:

The primary objective of this part was to write a Bash script that could automatically create, manage, and maintain backups for a specified directory. The script needed to be configurable and capable of cleaning up old backups to save space.

2.2. Implementation Details:

The backup solution was implemented using a Bash script named `` `backup.sh` ``. To make it easy to run, a `` `Makefile` `` was also created.

The script's logic is as follows:

1. It runs in an infinite loop to perform backups periodically.
2. In each cycle, it generates a unique filename for the backup archive using the current date and time.
3. It uses the `` `tar` `` command to create a compressed `` `.tar.gz` `` archive of the source directory.
4. After creating a new backup, it performs a cleanup. It lists all backups by time, identifies the ones that exceed the maximum number to keep, and deletes the oldest ones using `` `ls` ``, `` `tail` ``, and `` `rm` ``.
5. Finally, it waits for a specified interval before starting the next cycle.

2.3. Source Code (backup.sh)

~~~
``  #!/bin/bash

# Check if all required arguments are provided
if [ "$#" -ne 4 ]; then
    echo "Usage: $0 <source_directory> <destination_directory>
<interval_seconds> <backup_count>"
    exit 1
fi

# Assign arguments to variables for clarity
SOURCE_DIR="$1"
DEST_DIR="$2"
INTERVAL="$3"
MAX_BACKUPS="$4"

# Infinite loop to perform periodic backups
while true; do
    echo "Starting backup cycle..."

    # Create a timestamp for the backup file (e.g.,
20231027_153000)
    TIMESTAMP=$(date +%Y%m%d_%H%M%S)
    BACKUP_FILENAME="backup_${TIMESTAMP}.tar.gz"

    # Create the backup using tar
    echo "Backing up ${SOURCE_DIR} to
${DEST_DIR}/${BACKUP_FILENAME}"
    tar -czf "${DEST_DIR}/${BACKUP_FILENAME}" -C "${SOURCE_DIR}" .

    # --- Cleanup old backups ---
    echo "Cleaning up old backups..."
    # List backups in reverse chronological order, skip the newest
ones, and delete the rest
    ls -t "${DEST_DIR}"/backup_*.tar.gz | tail -n +$((MAX_BACKUPS +
1)) | xargs --no-run-if-empty rm

    echo "Cleanup complete. Next backup in ${INTERVAL} seconds."

    # Wait for the specified interval
    sleep ${INTERVAL}
done  ``
~~~

2.4. Execution and Results:

The following screenshot demonstrates the script being executed using the `make run` command. It shows several backup cycles running, followed by the `ls` command, which confirms that only the three most recent backups were kept, successfully demonstrating the cleanup functionality.

```
┌──(kali㉿kali)-[~/Desktop/FinalSubmission]
└─$ make run
bash ./backup.sh ./source_folder ./backups_folder 10 3
Starting backup cycle...
Backing up ./source_folder to ./backups_folder/backup_20251029_043402.tar.gz
Cleaning up old backups...
Cleanup complete. Next backup in 10 seconds.
Starting backup cycle...
Backing up ./source_folder to ./backups_folder/backup_20251029_043412.tar.gz
Cleaning up old backups...
Cleanup complete. Next backup in 10 seconds.
Starting backup cycle...
Backing up ./source_folder to ./backups_folder/backup_20251029_043422.tar.gz
Cleaning up old backups...
Cleanup complete. Next backup in 10 seconds.
Starting backup cycle...
Backing up ./source_folder to ./backups_folder/backup_20251029_043432.tar.gz
Cleaning up old backups...
Cleanup complete. Next backup in 10 seconds.
^Cmake: *** [Makefile:12: run] Interrupt


┌──(kali㉿kali)-[~/Desktop/FinalSubmission]
└─$ ls backups_folder
backup_20251029_043412.tar.gz  backup_20251029_043422.tar.gz  backup_20251029_043432.tar.gz
```

===

3. Part 2: Mini Unix Shell:

3.1. Objective:

The goal of Part 2 was to build a functional command-line interpreter in C. The shell needed to be able to read user commands, execute them in a separate process, and handle a built-in command for changing directories.

3.2. Implementation Details:

The mini-shell was implemented in C in the file `mini_shell.c`. The core of the program is a `while` loop that continuously prompts the user for input.

The program's logic is as follows:

1. It displays a **"mini-shell>"** prompt.
2. It reads a full line of input from the user.
3. The input string is broken down into individual "tokens" (the command and its arguments) using the `strtok` function.
4. It checks for special built-in commands:
    - If the command is `exit`, the loop terminates, and the program ends.
    - If the command is `cd`, it is handled directly by the main (parent) process using the `chdir` function. This is necessary because a child process cannot change the directory of its parent.

5. For all other commands, it uses `` `fork()` `` to create a new child process.
    - The ***child process*** uses `` `execvp()` `` to replace itself with the command the user typed.
    - The ***parent process*** uses `` `wait()` `` to pause and wait for the child process to finish its execution before printing the next prompt.

3.3. Source Code (mini_shell.c):

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

#define MAX_LINE 80

int main(void) {
    char *args[MAX_LINE/2 + 1];
    int should_run = 1;

    while (should_run) {
        printf("mini-shell> ");
        fflush(stdout);

        char input[MAX_LINE];
        fgets(input, MAX_LINE, stdin);

        char *token = strtok(input, " \n\t");
        int i = 0;
        while (token != NULL) {
            args[i] = token;
            i++;
            token = strtok(NULL, " \n\t");
        }
        args[i] = NULL;

        if (args[0] == NULL) {
            continue;
        }

        if (strcmp(args[0], "exit") == 0) {
            should_run = 0;
            continue;
        }

        if (strcmp(args[0], "cd") == 0) {
```

```c
            if (args[1] == NULL) {
                fprintf(stderr, "cd: expected argument\n");
            } else {
                if (chdir(args[1]) != 0) {
                    perror("cd failed");
                }
            }
            continue;
        }

        pid_t pid = fork();

        if (pid == 0) {
            // Child process
            if (execvp(args[0], args) == -1) {
                perror("Command execution failed");
                exit(1);
            }
        } else if (pid > 0) {
            // Parent process
            wait(NULL);
        } else {
            perror("Fork failed");
        }
    }
    return 0;
}  ``
```

3.4. Execution and Results:

The screenshot below shows the C program being compiled with `gcc` and then executed. It demonstrates a full interactive session where several commands (`ls -l`, `pwd`, `cd`) are run successfully, proving that the shell works as required. The session is terminated correctly using the `exit` command.

```
┌──(kali㉿kali)-[~/Desktop/FinalSubmission]
└─$ ./mini_shell
mini-shell> ls -l
total 40
drwxrwxr-x 2 kali kali  4096 Oct 29 04:34 backups_folder
-rwxrwxr-x 1 kali kali  1287 Oct 29 04:05 backup.sh
-rw-rw-r-- 1 kali kali   282 Oct 29 04:06 Makefile
-rwxrwxr-x 1 kali kali 16640 Oct 29 19:12 mini_shell
-rw-rw-r-- 1 kali kali  1434 Oct 29 19:12 mini_shell.c
drwxrwxr-x 2 kali kali  4096 Oct 29 04:05 source_folder
mini-shell> pwd
/home/kali/Desktop/FinalSubmission
mini-shell> cd source_folder
mini-shell> pwd
/home/kali/Desktop/FinalSubmission/source_folder
mini-shell> exit
```

4. Conclusion:

This lab assignment was successful in reinforcing key operating systems concepts through hands-on application.

Part 1 provided valuable experience in shell scripting and task automation, while Part 2 offered deep insight into process creation and management, which is a core function of any modern operating system. All requirements for both parts of the assignment, including the bonus `cd` command, were successfully implemented and tested.

==