MYY106 - Introduction to Computer Science
7th Lab

Introduction

In today's lab, we will use the C and Python programs from Lab 6 and measure their execution time using shell scripts. As we noticed from the measurements performed in the previous lab, the execution times can vary significantly for different runs of the same program. To obtain reliable results, we should run the same program (process) multiple times and calculate the average of the times measured. The average of a large number of measurements is much more reliable than a single measurement.

It is common to run the same program 10, 30, or even 100 times, measuring each execution separately, and then calculate the average. This process is simple for the computer but extremely tedious for the user if done manually. With shell scripts, we can automate the entire procedure so that the script performs the measurements for the desired program and computes the average automatically.

In today's lab, you do not need to record anything in an answer file. The shell script you create while answering the questions will serve as the file to submit.

Create the directory `lab7` inside the directory `myy106` in your personal folder. Download the files `sum.c` and `sum.py` you used in Lab 6 using `wget` from the address `https://www.cse.uoi.gr/~myy106/files/`. Compile the program `sum.c` using `gcc sum.c -o newsum` to create the executable `newsum`.

1. Using `vi`, create a file `experiments.sh` to write the shell script. Give execution permissions to `experiments.sh`.
2. The first line in a shell script should specify the absolute path of the shell that will execute it. Add the appropriate line so the script runs under `bash`.
3. Define a variable named `ITERATIONS` corresponding to the number of times the script will repeat the timing. Initialize it to `10`.
4. Define a variable named `PROG` corresponding to the program to execute. Initialize it with the C program name `newsum`, including its relative or absolute path.
5. Immediately after defining the variables, use `echo` to display the message `"Running N experiments for the program: PROG"` where `PROG` shows the program name (the value of the variable `PROG`) and `N` is the number of iterations (the value of the variable `ITERATIONS`). Execute the script `experiments.sh`.
6. Use a `while` loop with the appropriate condition to run for `ITERATIONS` repetitions.
7. In each iteration, display the message `"Running iteration n out of N"`, where `n` is the current iteration number from the `while` loop variable, and `N` is the total number of iterations (`ITERATIONS`). Execute `experiments.sh`.
8. Immediately after the message, inside the `while`, add `/usr/bin/time $PROG` to measure the program execution time. Execute the script.
9. For each execution of `PROG`, we notice two issues:
   - α′. We are not interested in the program output (the sum display in this case). We should avoid showing this unnecessary information on the screen. We do care about the output of `/usr/bin/time`.
   - β′. Execution time appears on the screen. To automate the process, we need to filter the `/usr/bin/time` output and retain only the actual execution time (the `real` value).
10. To ignore the program's output, redirect both stdout and stderr to `/dev/null` by adding `2>&1 >/dev/null` at the end of the program execution command. Execute the script to confirm that the program output is hidden, but `/usr/bin/time` output still appears.

11. To simplify the `/usr/bin/time` output, add the `-p` option. Execute the script again to see the difference.
12. Store the timing information in a variable `timing_output` instead of displaying it.
    Hint: use `timing_output=$(...)`.
13. Display `timing_output` to verify.
14. To extract only the decimal number after `real`, use `real_time=$(echo "$timing_output" | awk '/^real/ print $2')`. Display `real_time` and run the script to verify.
15. Once verified, remove the commands that display these variables.
16. Initialize `total_real_time` to `0` before the `while` loop to store the cumulative execution time. Inside the loop, update it by adding `real_time`. Run the script. Why does an error occur?
17. Use `bc` to handle decimal arithmetic: `total_real_time=$(echo "$total_real_time + $real_time" | bc)`.
18. After the loop, display `"Total real time: N seconds"` using `total_real_time`.
19. To calculate the average, divide `total_real_time` by `ITERATIONS` using `bc`: `average=$(echo "scale=6; $total_real_time / $ITERATIONS" | bc)`. Display `average`.
20. To avoid creating a new line for each iteration, use the carriage return character: `echo -en "\rRunning iteration $((i+1)) out of $ITERATIONS"`.
21. Improve the message to include progress percentage: `echo -en "\rRunning iteration $((i+1)) out of $ITERATIONS, Progress $(( i * 100 / ITERATIONS ))%"`.
22. Add a diagnostic message and newline for neat output.
23. Change `ITERATIONS` from 10 to 17 and execute the script.
24. Change `PROG` to `sum.py` (Python) and reduce `ITERATIONS` to 4. Execute the script.
25. With this shell script, we can easily measure different programs by changing `PROG`, and adjust repetitions via `ITERATIONS`. This demonstrates the usefulness of shell scripts. The script could be extended to compute standard deviation or accept `ITERATIONS` and `PROG` as arguments.

In later years, when implementing large projects or your thesis, you may need to measure execution times of programs and compare them with other implementations. Remember that while in your first year as students, you learned an automated way of timing processes using shell scripts.

Submit your answers: Paste the **contents** of the file `experiments.sh` in the form
https://forms.office.com/e/HNRXQJmv0n