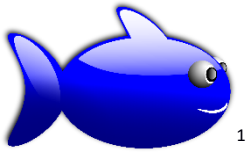
School of Engineering and Technology
University of Washington – Tacoma
Instructor: Wes Lloyd

Assignment 1



Profish: Profile shell

Due Date: Tuesday May 13th, 2025 AOE
Version: 0.10

Objective

The purpose of this assignment is to develop C code using the fork, wait, and exec commands to write a simple Linux shell. This shell, called "profish" will profile the execution of a Linux command. The user will provide a single Linux command with arguments. The profish shell will use fork, exec, and wait to run the command, and then the getrusage() Linux API to profile the command's resources used when executing on Linux.

For this program, implement your "profish" shell using the fork, exec, and wait commands.

The following limitations will apply:

1. The user commands that will be executed including all arguments combined will not exceed 65 characters.

2. Individual command arguments will not contain spaces. For example the command: "tail -n 10" "-n 10" is considered two arguments. The first is "-n", and the second is "10". Spaces are used to delimit arguments.

3. If the command operates on a file, the filename will either be in the local directory, or the user will provide a fully qualified path name which fits within the command length requirements. *The profish shell is not responsible for finding the file.*

4. All commands provided to "profish" will run using the user's original path.
   Type "echo $PATH" to see the current path variable setting.
   The profish shell is not responsible for finding the command file to run.

5. For each command, the maximum number of arguments including the command itself will not exceed 6. This equates to 5 command arguments, plus the command, for a total of 6 inputs.

[1] Creative Commons Licensed image

6. On the event that a user makes a mistake typing a command or its arguments, profish will simply fail to run the command. A simple error should be shown, but only if the exec fails.

7. Profish does not accept command line arguments. Profish will request 1 Linux command to run, plus arguments. Some commands may require a filename as an argument.

8. To profile the command, the Linux getrusage() API should be used.

9. Coding style is important. To encourage source code modularity the int main(void) method should not exceed 20 lines of code. Solutions having a longer int main(void) method will not be accepted and will be treated as a non-submission.

Additional Development Tasks:

10. Capture the output of the command and frame its output before showing the profiling statistics. The output for the command can be captured to a text file and then displayed before the profile results. Use file redirection as shown in the example in class. Capture the output of the command's STDOUT file stream to a separate temporary file on disk. Read the temporary file and display its output in the console output of the profish shell. The output should be framed with a header and footer line as shown in the sample output. Once results are displayed to the screen any temporary files should be deleted.

    File redirection example:
    http://faculty.washington.edu/wlloyd/courses/tcss422/examples/exec2.c

11. *Display 80 character process delimiter line and colored section labels - When profish runs a remote program, an 80-character divider line is printed to separate the command's output. The label should adjust to the length of the command to have the proper number of dashes. In addition, labels for the sections "runtime:", "time components:", "memory metrics:", "I/O operations:", and "interrupts:" should each be shown in blue.*

    ```
    -----CMD: grep -c the googlebig.txt--------------------------------------------
    -----CMD: md5sum googlebig.txt------------------------------------------------
    -----CMD: wc -l googlebig.txt-------------------------------------------------
    ```

    Profish follows the output of all commands with an 80-character FIXED delimiter line at the end of the output.

    ```
    --------------------------------------------------------------------------------
    ```

    As the process finishes running, a line is printed indicating that total elapsed time the process has taken. The timing of the process should be done using two calls to the gettimeofday() Linux API. Once the command has been profiled the profish shell should exit.

12. Display Linux CPU time calculations: On a Linux system, the CPU-user-mode-time plus the CPU-kernel-mode-time equals the elapsed time (runtime) when a process only has 1-thread. This provides a measurement sanity check. For the example below, we see that the elapsed time (runtime) is 1905 milliseconds, and the Linux CPU time accounting value (labeled as "linux cpu

time acct") which represents the total time is 1,904,535 microseconds which is 1,904.535 milliseconds. We can see that this is made up from 1,788,746 microseconds of cpu-user-mode time and 115,789 microseconds of cpu-kernel-mode time. For the example (running md5sum) kernel mode time was about 6% of the total time. Your task is to calculate and display "linux cpu time acct" which is the sum of cpu user time plus cpu kernel time (user-time+kernel-time).

13. When profish can't run an external command, a message indicating failure should be displayed:

    CMD:[SHELL] STATUS CODE=-1

    The message identifies that profish failed to run the command, and provides a status code.

To test profish, a number of commands may be used. Here are some possible commands to test your profish shell:

"wc"	Reports the line count, word count, and character count
"md5sum"	Generates a unique 128-bit md5 (checksum) hash message digest
"grep –c the"	Counts the number of occurrences of a given word, here "the"
"grep –ci the"	Counts the number of occurrences of a given word ignoring case, here "the"
"tail –n 10"	outputs 10 lines from the end of a file
"head –n 10 "	outputs 10 lines from the start of a file
"ls –l"	provides a long directory listing
"sysbench --threads=1 --time=1 cpu run"	Prime number generation CPU benchmark
"sysbench --threads=4 --time=1 cpu run"	Prime number generation CPU benchmark
"sysbench --threads=8 --time=1 cpu run"	Prime number generation CPU benchmark
"sysbench --threads=10 --time=3 mutex run"	Mutex performance test
"sysbench --threads=10 --time=3 threads run"	Thread subsystem performance test
"sysbench --threads=1 --time=10 memory run"	Memory functions speed test
"sysbench --threads=1 fileio prepare"	File I/O test-prepare
"sysbench --threads=1 --time=10 --file-test-mode=rndrw fileio run"	File I/O test-run

Using "top -d .1" it is possible to watch profish run a fork and exec to run a separate program.

A large 1.4 GB text file is available to help test your shell at:
https://faculty.washington.edu/wlloyd/courses/tcss422/assignments/googlebig.txt.gz

Input

There are no command line arguments for profish. The profish shell should be invoked as follows:

```
$./profish
```

Output

Here is a sample output sequence for running profish. The output is annotated with arrows and notes on the right. The notes should not be included in profish output. The notes are to explain the output.

```
$ ./profish
profish>md5sum googlebig.txt                          ← user types command here
-----CMD: md5sum googlebig.txt-------------------------------------------------
0362a7bf9035eba363462ea484bb43a6  googlebig.txt       ← command output
--------------------------------------------------------------------------------
```

```
Total elapsed time:1905ms                          ← command runtime in ms
runtime:                                           ← Linux CPU time calculations
total cpu user time=1788746 microsec
total cpu kernel time=115789 microsec
linux cpu time acct=1904535 microsec
time components:                                   ← time components from getrusage()
cpu user time=1 sec
cpu kernel time=0 sec
cpu user time=788746 microsec
cpu kernel time=115789 microsec
memory metrics:                                    ← memory components from getrusage()
max resident set size=2096
integral shared memory size=0
integral unshared data size=0
integral unshared stack size=0
page reclaims=100
page faults=0
swaps=0
I/O operations:                                    ← I/O metrics from getrusage()
block input ops=0
block output ops=16
IPC msgs sent=0
IPC msgs recv'd=0
interrupts:                                        ← interrupt stats from getrusage()
signals recv'd=0
voluntary context switches=1
involuntary context switches=10
```

Summary of Development Tasks

1. Write code to obtain a user provided commands plus arguments (string).

2. Split individual words from the user provided command to extract the command arguments so they can be provided to exec(). For example, a user may type "grep –ci the". This string will be split into three strings: "grep", "-ci", and "the".

   Here is an example of hard coding these strings to call execlp:

   ```
   execlp("grep","-ci","the",(char *) NULL);
   ```

   This approach is not dynamic.
   A dynamic number of arguments can be accepted using execvp() which accepts a pointer to a NULL terminated array of char pointers (char **). Each char pointer points to a null terminated word.

3. Fork the original parent process and use exec to run the command.

4. Use waitpid() to wait for the child process to exit. This is required by getrusage(). Without waiting, getrusage() will fail.

5. Print out command header lines (**EC**) (80 characters) and the 80-character delimiter lines.

6. Determine how to measure and print out the elapsed time (runtime) of the profiled command using the gettimeofday() API.

7. **(EC)** Store the output of the command into a temporary file. Read the temporary output file and generate clean output for profish with the command's output at the top, and the profiling information below. The command's output should not go to STDOUT when it is run, but it is captured to a file. Delete the temporary output file once the command is finished.

8. Use getrusage() to print out every available output from struct rusage. See and follow the sample output example.

9. **(EC)** process the timing information in struct rusage to calculate the total cpu-user time in microseconds, the total cpu-kernel time in microseconds, and the total cpu-user+kernel time for Linux CPU time accounting. This total time should nicely reflect the elapsed time for single threaded processes.

10. **(EC)** Display section labels for the getrusage output using blue text.

When coding a solution, it is recommended to tackle key design challenges individually (one at a time) to simplify the testing/debugging of the implementation.

Helpful notes:

Time information in struct rusage is represented with struct timeval.

```
struct timeval {
    long tv_sec;                /* seconds */
    long tv_usec;               /* microseconds */
};
```

Struct timeval decomposes time into two pieces (components), the whole seconds (tv_sec), and the additional microseconds (tv_usec), which is the fractional part of the second. For example, if the time is 7.5 seconds, then you will have 7 seconds as a long, and 500,000 microseconds as a long. To calculate the total time, these need to be added. Seconds must be converted to microseconds by multiplying by 1,000,000. Then you can add the microseconds to get the total time.

Here is 'struct rusage'. This structure contains all of the profiling metrics. The metrics are displayed to the screen in the same order they appear in the struct. Note the sections in the output that delineate specific types of metrics. You will need to use the manual page to learn how to use the getrusage() API.

```
struct rusage {
    struct timeval ru_utime; /* user CPU time used */
    struct timeval ru_stime; /* system CPU time used */
    long   ru_maxrss;        /* maximum resident set size */
    long   ru_ixrss;         /* integral shared memory size */
    long   ru_idrss;         /* integral unshared data size */
    long   ru_isrss;         /* integral unshared stack size */
    long   ru_minflt;        /* page reclaims (soft page faults) */
    long   ru_majflt;        /* page faults (hard page faults) */
    long   ru_nswap;         /* swaps */
    long   ru_inblock;       /* block input operations */
    long   ru_oublock;       /* block output operations */
    long   ru_msgsnd;        /* IPC messages sent */
    long   ru_msgrcv;        /* IPC messages received */
```

```
    long   ru_nsignals;      /* signals received */
    long   ru_nvcsw;         /* voluntary context switches */
    long   ru_nivcsw;        /* involuntary context switches */
};
```

Generative-AI policy:

Use of Generative-AI (i.e. LLMs) is permitted on this assignment. If you use generative-AI to support writing the code for this assignment, then you are required to submit two additional files: **#1 (llm.txt)** this file should contain all LLM prompts used to generate C source versions of profish. If LLMs are not used, submit the file with a statement indicating that no LLMs were used in the project. **#2 (llm_bugs.pdf)** this file provides a numbered list of C source code versions and for each version, a numbered list of LLM code generation bugs. Briefly describe the bugs and solutions found.

Grading Rubric

This assignment will be scored out of 110 points. (110/110)=100%
120 points are possible.

Toal:	80 points
10 points	Run a command with at least 1 argument
10 points	Run a command with up to 5 arguments
5 points	Report time component metrics from getrusage()
5 points	Report memory metrics from getrusage()
5 points	Report I/O operations metrics from getrusage()
5 points	Report interrupts metrics from getrusage()
5 points	Shell ends gracefully. The program returns cleanly to the calling shell. (It is not necessary to press <ENTER> to return to the Linux command prompt.
5 points	Display total elapsed time in milliseconds for the command that was run.
5 points	STATUS CODE message shown for a failed command.
5 points	Display 80-character delimiter line after the command's output is displayed
5 points	Frame the output of the command neatly at the top of the output by redirecting the command's output to a temporary file on disk which is deleted after the command finished.
10 points	Display 80-char command header lines which displays the command that was run and include colored BLUE labels to delineate different types of profiling metrics in the getrusage() output
5 points	Display Linux CPU time accounting metrics in the output. This should include the total cpu-user-mode time, the total cpu-kernel-mode time, and the total cpu-time. The total measured CPU-time should closely match the elapsed time for single-threaded programs.

Miscellaneous:	40 points
5 points	Program compiles, and does not crash upon testing
5 points	Coding style, formatting, and comments
5 points	Makefile with valid "all" and "clean" targets provided with submission
5 points	Output format matches the provided example (even if a portion doesn't work!)
10 points	Submission of text file (**llm.txt**) containing all LLM prompts used to generate source code. If LLMs are not use, submit a file with a statement indicating that no LLMs were used.
10 points	Submission of PDF file (**llm_bugs.pdf**) providing a numbered list of all versions of source code generated if using an LLM. For each version, provide a numbered list of LLM code generation bugs and briefly describe solutions if found.

WARNING!

10 points Automatic deduction if main program file is not named "profish.c"

What to Submit

For this assignment, submit a tar gzip archive as a single file upload to Canvas. Package up all of the files into the single tar gzip archive. A makefile with "all" and "clean" targets should be included. Makefile examples can be found online at: https://faculty.washington.edu/wlloyd/courses/tcss422/examples/

Tar archive files can be created by going back one directory from the project source directory with "`cd ..`", then issue the command "`tar czf A1.tar.gz my_dir`". "my_dir" is the directory of where your program is stored. Canvas automatically appends your name to the file upon upload. Here "my_dir" is the directory that contains source code and the makefile. **No other files should be submitted.**

Pair Programming (optional)

O*ptionally*, this programming assignment can be completed using two person teams.

If choosing to work in pairs, ***only one*** person should submit the team's tar gzip archive to Canvas.

Additionally, ***EACH*** member of a pair programming team must provide an **effort report** of team members to quantify team contributions for the overall project. **Effort reports** must be submitted INDEPENDENTLY and in confidence (i.e. not shared) by each team member to capture each person's overall view of the teamwork and outcome of the programming assignment. Effort reports are not used to directly numerically weight assignment grades.

Effort reports should be submitted in confidence to Canvas as a PDF file named: "effort_report.pdf". Google Docs and recent versions of MS Word provide the ability to save or export a document in PDF format. Distribute 100 points for category to reflect each teammate's contribution for: research, design, coding, testing. Effort scores should add up to 100 for each category. Even effort 50%-50% is reported as 50 and 50.

Please do not submit 50-50 scores for all categories.

It is highly unlikely that effort is truly equal for everything. Ratings must reflect an honest confidential assessment of team member contributions. ***50-50 ratings and non-confidential scorings run the risk of an honor code violation.***

Here is an **effort report** for a pair programming team (written from the point of view of Jane Smith):
NOTE: The 'research' task includes both Internet searches and LLM prompting.

1. John Doe

Research	24
Design	33
Coding	71
Testing	29

2. Jane Smith
Research 76
Design 67
Coding 29
Testing 71

FOR SPRING 2025: TCSS 422 effort reports should include a **short description** of how pair programming was conducted. If your group worked virtually, then include a description of the tools used to support the distributed remote pair programming. Example tools include: Google Hangouts, Zoom, GitHub, Slack, and Discord. In addition to describing the use of tools to support teamwork, please describe how tasks were divided or how work was accomplished together in meetings.

Team members may not share their **effort reports**, or collaborate together in writing them. Failure to keep reports confidential is considered an honor code violation. Reports should be submitted independently in Canvas as a PDF file. Failure of one or both members to submit the **effort report** will result in both members receiving NO GRADE on the assignment… (programs are *considered late until both effort reports are submitted*)

Disclaimer regarding pair programming:
The purpose of TCSS 422 is for everyone to gain experience programming in C while working with operating system and parallel coding. Pair programming is provided as an opportunity to harness teamwork to tackle programming challenges. But this does not mean that teams consist of one champion programmer, and a second observer that only passively participates! Tasks and challenges should be shared as equally as possible to maximize learning opportunities.

Change History

Version	Date	Change
0.1	04/29/2025	Original Version