Assignment 1
CS-351
Fall 2023

Due Date: 10/12/2023 at 11:59 pm
You may work in groups of 1-5 (only one person from the group should submit)
You may submit as many times as you like
Canvas will save only the last attempt

Outcomes:

1. **Implement** applications using `fork()`, `exec()`, and `wait()` system calls.
2. **Design and implement** multi-process applications.
3. **Describe** fault-tolerance benefits of multi-process applications.
4. Empirically **evaluate** the performance of serial vs parallel applications.
5. **Implement** basic process management functions within applications.
6. **Explain** how the permission inheritance works between parent and child processes.

Knowledge Gained:

1. Knowledge of operating system structures and internals (e.g., process management, directory structure, installed applications).
2. Knowledge of concepts for operating systems (e.g., Linux, Unix.)
3. Knowledge of parallel and distributed computing concepts.

The knowledge and skills gained in this assignment is useful in the following job roles:

- Research & Development Specialist
- Systems Requirements Planner
- Systems Developer
- Enterprise Architect
- Security Architect
- Systems Security Analyst
- Information Systems Security Developer

Part I Overview

Write a simple shell program called `shell`. When ran, the parent process will print out the prompt which looks like:

```
cmd>
```

and will then wait for the user to enter a command such as `ls`, `ps`, or any other command. For example:

```
cmd> ls
```

The parent process will then use `fork()` in order to create a child and then call `wait()` in order to wait for the child to terminate. The child process will use `execlp()` in order to replace its program with the program entered at the command line. After the child produces the output, the parent will prompt the user for another command. For example:

```
cmd>ls
file.txt fork fork.c mystery1 mystery1.cpp mystery2 shell shell.cpp
```

The parent will repeat the above sequence until the user types the `exit` command, which will cause the parent process to exit.

Sample/Skeleton Codes

A basic forking example (`fork.cpp`) had been provided in the class notes as well as in the zip bundle of sample files/skeletons accompanying this assignment called `samples.zip`. The bundle also includes a skeleton file for the shell with parts to complete marked with `TODO`. You are **not required** to use these files, but may find them helpful.

NOTE: ***Please make sure to error-check all system calls in this assignment***. This is very important in practice and can also save you hours of debugging frustration. `fork()`, `execlp()`, and `wait()` will return -1 on error. Hence, you need to always check their return values and terminate your program if the return value is -1. For example:

```
pid_t pid = fork()

if(pid < 0)
{
        perror(''fork'');
        exit(-1);
}
```

The `perror()` function above will print out `fork` followed by the explanation of the error.

Part II Overview

In this part you will design a two programs that make use of `fork()`, `execlp()`, and `wait()` functions to fetch weather information for a bunch of locations from a web-based weather service API. The basic logic of each program will be as follows:

1. The program reads the latitude and longitude location from the specified file.
2. The program creates a child that uses the `curl` program to contact the weather service, fetch the weather information (in JSON format), and save it to the file.
3. Repeat the process for the next location specified in the file. If there are no more locations left, exit

The difference between the two programs is the method they use for fetching weather for multiple locations:

1. **a serial fetcher:** which will fetch the city information one city at a time. That is, the parent must wait for each child to finish fetching the weather for its location before creating another child.
2. **a parallel fetching:** the parent will create a child for each location (i.e., without waiting for a child to complete) and only once all children have been created, the parent will wait for all of them to terminate.

You will then compare the performance of the two types of fetchers using the `time` utility. Both fetchers will use the Linux `curl` program in order to communicate with the weather API and fetch the weather information that is formatted according to the JavaScript Object Notation format. You can read more about the https://www.tutorialspoint.com/json/json_quick_guide.htm.

The command line to fetch the JSON weather data for location at latitude location 52.52 and longitude location 13.41 and then save it in file file1.json is as follows:

```
/usr/bin/curl -o file1.json "https://api.open-meteo.com/v1/forecast?latitude=52.520000&longitude=13.410000&current_weather=True
```

Where

- `/usr/bin/curl`: The name of the curl program
- `-o`: Specifies that we want to save the data fetched to the file.
- `file1.json`: The name of the file where we want to save the fetched data.
- `https://api.open-meteo.com/v1/forecast?latitude=`**52.520000**`&longitude=`**13.410000**`&current_weather=True`: is the URL of the web API from which to fetch the data. The values in **bold blue** represent latitude and longitude of locations from which to fetch the weather.

You can practice the command in the shell before continuing.

Functional Requirements:

In your program, the parent process shall first read the file, `locations.txt`, containing the latitude and longitude locations (as floating point numbers) one per line. The basic format is as follows:

```
latitude1 longitude1 latitude2 longitude2 .
.
.
latitudeN longitudeN
```

For example:

```
52.52 13.41
48.8567 2.3510
40.4167 -3.7033
41.8955 12.4823
50.4422 30.5367
```

You can get the latitude and longitude locations for e.g., a city from https://open-meteo.com/en/docs to create your own input files to test your programs.
Next, the parent process shall fork the child processes. Each created child process shall use the `execlp()` system call to replace its executable image with that of the `curl` program with proper parameters being passed to `curl` to ensure that curl fetches the weather from the latitude and longitude locations and saves it to the file with the proper name. The file for the first location should be named `file1.json`, `file2.json` for the second location, etc. The requirements for how the parent proceeds next depends on whether this program is a serial or a parallel fetcher. These are described below.

The two types fetchers are described in detail below.

The two fetchers shall be implemented as **separate programs**. The serial fetcher program shall be called `serial.c` (or `.cpp` extension if you use C++). The parallel fetcher program shall be called `parallel.c` (or `.cpp` extension if you use C++).

Serial Fetcher Requirements

The serial fetcher shall be invoked as `./serial` and fetch locations one by one. After the parent process has read and parsed an entry from `input.txt` file, it shall proceed as follows:

1. The parent process forks off a child process.
2. The child will print it's process ID (you can use `getpid()` system call to retrieve process's ID) and the process ID of it's parent (you can use `getppid()` system call to do this). The child and parent should also print their `UID` using the `getuid()` and `GID` using the `getgid()` system calls. Each process is assigned a UID and GID when it is launched. Typically, these are the same as the UID of the user who has launched the process and the GID of that user's default group. Typically, these are the same user ids and group ids that you have learned about in the permissions quiz (unless the program had an s bit set to run under another user/group). When the process accesses a file/directory, the OS checks these process UID/GID values against the permissions of the file/directory to enforce permissions.

Based on this, do children inherit the user and group from the parent?

3. The child uses `execlp("/usr/bin/curl", "curl",... other arguments NULL)` system call in order to replace its program with `curl` program that will fetch the weather information for the latitude/longitude location and save it in `fileX.json` where X represents the location number. The first location will be saved in file named `file1.json`, second in `file2.json` etc. You can check the file called `numberedfilegen.cpp` to see how to dynamically generate names of files.
4. The parent executes a `wait()` system call until the child exits.
5. The parent forks off another child process which downloads the next location specified in `input.txt`.
6. Repeat the above steps until all files are downloaded.

Parallel Fetcher

The parallel fetcher shall be invoked as `./parallel` and

1. The parent will fork off n children, where n is the number of locations in `input.txt`.
2. Each child executes `execlp("/usr/bin/curl", "curl",... NULL)` so that the child fetches the weather information for a latitude/longitude location and saves it to the file (same as for the serial fetcher).
3. The parent calls `wait()` (n times in a row) and waits for all children to terminate.
4. The parent exits.

Please note:

- While the parallel fetcher executes, the outputs from different children may intermingle. This is acceptable.

- `fork.c` file posted on Canvas provides an example of using `fork()`, `execlp()`, and `wait()` system calls. Please feel free to modify it in order to complete the above tasks.

Performance Comparison

Use the `time` program to measure the execution time for the two fetchers. For example:

```
time ./serial
real 0m10.009s
user 0m0.008s
sys 0m0.000s
```

The column titled `real` gives the execution time in seconds. Please get the execution times for both fetchers using the following `input.txt` file:

```
52.52 13.41
48.8567 2.3510
40.4167 -3.7033
41.8955 12.4823
50.4422 30.5367
```

Your execution times should be submitted along with your code (see the section titled "Submission Guidelines".

In your submission, please include the answers to the following questions (you may need to do some research):

1. In the output of `time`, what is the difference between `real`, `user`, and `sys` times?
2. Which is longer: `user` time or `sys` time? Use your knowledge to explain why.
3. When fetching all of the locations in the file, which fetcher finishes faster? Why do you think that is?
4. Suppose one of the fetcher children crashes. Will this affect other children? Explain. What is your conclusion regarding the benefits or disadvantages of isolating each fetch task within a different process?

Sample/Skeleton Codes

The following files are provided for your benefit. They illustrate how to perform various functions you may find useful in this assignment.

- `fork.cpp`: A sample program showing the basic use of `fork()/execlp()/wait()`.
- `urlgenerator.cpp`: a basic program giving an example of how to generate the URL based on the latitude and longitude location.
- `numberedfilegen.cpp`: a basic program showing how to generate a bunch of files with names such as e.g., `file1.txt`, `file2.txt`, etc.
- `args.cpp` Shows the basic program showing how the argument passing works.
- `json.hpp` and `viewjson.cpp`: the `.cpp` file contains a self-standing program that can parse the downloaded JSON file and prints out the information. You can use it to verify that the files were properly downloaded. The basic syntax is `./viewjson <JSON FILE NAME>` e.g., `./viewjson file1.json`.
- `input.txt`: a sample file containing locations. Your program should be able to support any number or types of locations.

BONUS

Implement a multi-process linear search.

The parent process shall load a file of strings into an array (or a vector), split the array into n sections, and then fork off n children. Each child process shall search one of the n sections of the array, for the specified string. If the child does not find the string, it terminates with the exit code of 1. Otherwise, it terminates with the exit code of 0.

The parent, meanwhile, continuously executes `wait()`. Whenever, one of the child processes terminates, the `wait()` will unblock, and the parent process shall check the exit code of the child. The exit status of the child can be checked using the `WEXITSTATUS` macro as follows:

```
//The variable to hold the exit status int exit_status;

//Other code ....

if(wait(&exit_status) < 1) { perror("wait"); exit(-1); }
if(WEXITSTATUS(exit_status) == 0) { ...  } ...
```

If all children terminate with the code of 1, then the parent prints `"No string found"` to the terminal. Otherwise, the parent terminates all child processes and exits. A child process can be terminated using the `kill()` system call. For example, `kill(1234,SIGKILL)` will terminate the process with process id `1234` (be sure to include the header files `#include <sys/types.h>` and `#include <signal.h>`; otherwise your program may not compile).

Technical Details

The program shall be ran using the following command line:

```
./multi-search <FILE NAME> <KEY> <NUMBER OF PROCESSES>
```

Where `<FILE NAME>` is the name of the file containing the strings, `<NUMBER OF PROCESSES>` is the number of child processes, and `<KEY>` is the string to search for. For example, `./multi-search strings.txt abcd 10` tells the program to split the task of searching for string `abcd` in file `string.txt` among 10 child processes.

SUBMISSION GUIDELINES:

- ***This assignment MUST be completed using C or C++ on Linux.*** item You may work in groups of 5.
- ***Your assignment must compile and run on Tuffix.***
- Please hand in your source code electronically (do not submit .o or executable code) through **CANVAS**.
- You must make sure that the code compiles and runs correctly.
- Write a README file (text file, do not submit a .doc file) which contains
  - The names and email addresses of all group members.

  - Description of contributions of each team member.
  - A **truthful** statement asserting that each team member is familiar with all functionality in the assignment.
  - The programming language you used (i.e., C or C++).
  - How to execute your program.
  - The execution times for both fetchers.
  - The answers to all questions above.
  - Whether you implemented the extra credit.
  - Anything special about your submission that we should take note of.

- Place all your files under one directory with a unique name (such as `p1-[userid]` for assignment 1, e.g. `p1-mgofman1`).
- Tar the contents of this directory using the following command. `tar cvf [directory_name].tar [directory_name]` E.g. `tar -cvf p1-mgofman1.tar p1-mgofman1/`
- Use CANVAS to upload the tared file you created above.

Grading guideline:

- All programs compile: 5'
- Correctly shell: 25'
- Correct serial fetcher: 30'
- Correct parallel fetcher: 30'
- Execution times for both fetchers: 5'
- README file: 5'
- Bonus: 15'
- Late submissions shall be penalized 10%. No assignments shall be accepted after 24 hours.

Academic Honesty:

Academic Honesty: All forms of cheating shall be treated with utmost seriousness. You may discuss the problems with other students, however, you must write your **OWN codes and solutions**. Discussing solutions to the problem is **NOT** acceptable (unless specified otherwise). Copying an assignment from another student or allowing another student to copy your work **may lead to an automatic F for this course**. Moss shall be used to detect plagiarism in programming assignments. If you have any questions about whether an act of collaboration may be treated as academic dishonesty, please consult the instructor before you collaborate. Details posted at http://www.fullerton.edu/senate/documents/PDF/300/UPS300-021.pdf.