COMP3230A/CSIS0230A Principles of Operating Systems

Programming Project – timekeeper – A new system utility program

Total 14 points

(version: 1.0)

Objectives

1. An assessment task related to ILO4 [Practicability] – "demonstrate knowledge in applying system software and tools available in modern operating system for software development".
2. A learning activity related to ILO 2a.
3. The goals of this programming project are:
   - to have hand-on practice on designing and developing a system utility program, which involves execution of multiple processes and collection of process statistics;
   - to learn how to use various important Unix system functions
     - to perform process creation and program execution;
     - to interact between processes by using signals and pipes; and
     - to get process's running statistics.

Task

For this project, you are going to write a new system program - ***timekeeper***, which is being used by end-users to obtain the execution statistics of a program or sequence of programs connected by pipes. This utility program will report the wall clock (real) time, user time, system time, number of context switches, and termination status of each program. For example, to obtain the execution statistics of the program *firefox*, we enter the following command under the command prompt:

```
./timekeeper firefox
```

After the *firefox* program terminated, timekeeper will display firefox process's running statistics:

```
The command "firefox" terminated with returned status code = 0
real: 37.21 s, user: 4.13 s, system: 0.44 s, context switch: 20292
```

The project consists of four parts:

- Lab 1 - Create the 1st version of the timekeeper program that (i) parses input arguments to extract the command (and its arguments), (ii) creates a child process to execute the command, and (iii) waits for child process to terminate and then prints out child process's termination status.
- Lab 2 - Modify Lab 1 exercise program to make the timekeeper process (i) not being affected by SIGINT signal, (ii) uses SIGUSR1 signal to inform its child process to start execution of the command, and (iii) checks whether the child process was terminated because of interrupted by signal.
- Lab 3 - Modify Lab 2 exercise program to make the timekeeper process to print out the running statistics of the terminated child process.

- Lab 4 - Modify Lab 3 exercise program to allow the timekeeper program to (i) accept multiple commands (with arguments) that are interconnected by the special symbol "**!**", which acts as the *pipe* operator, and (ii) print out the termination status and running statistics of each terminated child process.

In summary, you are going to develop the timekeeper program step-by-step.

Specification

Basic features of the *timekeeper* program

1. When invoking the *timekeeper* program without arguments, it will terminate successfully without any output.

2. When invoking the *timekeeper* program with input arguments, it will interpret the input and will start the corresponding program(s) with necessary arguments. The *timekeeper* program should be able to execute any normal program (i.e. programs that are in binary format) that can be found under

   – absolute path (starting with /) specified in the command line, e.g.

   ./timekeeper /home/tmchan/a.out

   – relative path (starting with .) specified in the command line, e.g.

   ./timekeeper ./a.out abc 1000

   – directories listed in environment variable $PATH, e.g.

   ./timekeeper gedit readme.txt

   where *gedit* is in the directory /usr/bin, and *timekeeper* should locate the target program by searching through all paths listed in the PATH environment variable, e.g., $PATH=/bin:/usr/bin:/usr/local/bin

   After a new child process is created, *timekeeper* will print out a message that shows the child process id. For example:

   Process with id: 11463 created for the command: cat

   If the target program cannot be started/executed, *timekeeper* will print out a message to indicate the problem. For example:

   timekeeper experienced an error in starting the command: xxxxxx

   To reduce the complexity of this project, other special characters (e.g. ‘, “, >, >>, <, <<, |) will not be appeared in the input arguments.

3. When the invoked program terminated, *timekeeper* will print out the wall clock (real) time, user time, and system time used by that process as well as the number of context switches experienced by that process. Here are the definition of those timings:

   – Real time: the elapsed wall clock time (in seconds) between start and end of the invoked program.

   – User time: the total number of CPU-seconds that the process spent in user mode.

- System time: the total number of CPU-seconds that the process spent in kernel mode.

All timings are displayed in units of second and have the precision of 2 decimal places. For example,

real: 37.21 s, user: 4.13 s, system: 0.44 s, context switch: 20292

4. When the invoked program terminated successfully, *timekeeper* will print out that process's termination status in addition to the running statistics as defined in item 3. For example,

The command "firefox" terminated with returned status code = 0
real: 37.21 s, user: 4.13 s, system: 0.44 s, context switch: 20292

5. The *timekeeper* process and its child process(es) are required to response to the SIGINT signal (generated by pressing Ctrl-c or by the *kill* system command) as according to the following guideline:

- Upon receiving the SIGINT signal, the child process(es) will response to the signal as according to the predefined behavior of the program in response to the SIGINT signal; while the *timekeeper* process should not be affected by SIGINT and will continue until it detects that its child processes have terminated.

- Instead of printing out a process's termination status, *timekeeper* will output a message to indicate that the corresponding process is terminated by the SIGINT signal. For example,

The command "add" is interrupted by the signal number = 2 (SIGINT)
real: 1.38 s, user: 1.26 s, system: 0.10s, context switch: 33

6. Similar to item 5, when the invoked program terminated involuntary because of receiving signal (e.g. SIGSEGV), *timekeeper* will output a message to indicate that the corresponding process is terminated by a specific signal. For example,

The command "segfault" is interrupted by the signal number = 11 (SIGSEGV)
real: 0.45 s, user: 0.29 s, system: 0.02s, context switch: 27

7. To have a better coordination between the *timekeeper* process and its child process(es), all child process(es) created by *timekeeper* has/have to wait for the SIGUSR1 signal before the start of execution of the target program. Once *timekeeper* has created all child process(es), it will send the SIGUSR1 signal to child process(es) to trigger the execution.

8. The timekeeper program uses a special symbol "**!**" to act as the ***pipe*** operator. Please note that we are not using the tradition pipe symbol "|" as we don't want the shell process to be confused if "|" is used in this project. When the *timekeeper* process detects that its input arguments contain the special symbol "**!**", it will identify the sequence of commands and will start the corresponding programs with necessary arguments. In addition, these programs will be linked up as according to the logical pipe defined by the input arguments. For example,

./*timekeeper* cat c3230a.txt ! grep kernel ! wc -w

In this example, the *timekeeper* program will create three child processes to execute the *cat*, *grep*, and *wc* programs. The standard output of the *cat* process is connected to the standard input of the *grep*

process, and the standard output of the *grep* process is connected to the standard input of the *wc* process.

When detecting a child process has terminated, *timekeeper* will print out the termination status (or signal information) and running statistics of that terminated child process. Upon detecting all child processes have terminated, *timekeeper* will terminate too.

System Calls

You need the following system calls to build the *timekeeper* program.

- *fork()* – The fork function is the primitive for creating a child process.
- *exec* family of functions – Use one of these functions to make a child process execute a new program.
- *waitpid()* and *waitid()* – Wait for a child process to terminate or stop, and determine its status.
- *signal()* and *sigaction()* – Specify how a signal should be handled by the process.
- *kill()* – A process uses this system function to send a signal to another process.
- *pipe()* – This creates both the reading and writing ends of the pipe for interprocess communication. The returned reading and writing ends are represented as file descriptors.
- *dup2()* – This duplicates an open file descriptor onto another file descriptor; in other words, it allocates another file descriptor that refers to the same open file as the original.

Note: You are not allowed to use the *system()* or *popen()* functions to execute the target program.

Documentation

1. At the head of the submitted source code, state clearly the
   - Student's name
   - Student Number
   - Development platform
   - Compilation – describe how to compile your program
2. Inline comments (try to be detailed so that your code could be understood by others easily)

Computer platform to use

For this assignment, you are expected to work on any Linux systems (preferred on the latest distribution of Fedora or Ubuntu). Your programs are written in C (or C++) and successfully compiled with gcc or g++.

Grading Criteria

1. The tutor will first test your *final submission* after the project deadline. If your program fully complies with the project specification, you'll get all the marks for all four stages automatically. Otherwise, tutor will test your *Lab3 submission* (if any). If it passes all test cases, you'll get all the marks for the first three stages. Otherwise, tutor will test your *Lab2 submission* (if any). If it passes all test cases, you'll get all the marks for the first two stages. Otherwise, tutor will test your *Lab1 submission* (if any) to give marks.
2. If you want to get those stage marks *earlier*, you are allowed to demonstrate your program to the tutor, during the consultation hours or by appointment. This is recommended for those who have difficulties in finishing all stages (e.g. when you are in a rush for multiple deadlines).

3. Your submission(s) will be primarily tested under Ubuntu 14 (installed in HW311/312). Make sure that your program can be compiled *without any error or warning*. Otherwise, we have no way to test your submission(s) and thus you have to lose *all* the marks.
4. As the tutor will check your source code, please write your program with good readability (i.e., with good code convention and sufficient comments) so that you will not lose marks due to possible confusions.

Documentation (1 point)	High Quality (0.5 point)	• Include necessary documentation to clearly indicate the logic of the program
	Standard Quality (0.5 point)	• Include required student's info at the beginning of the program • Include minimal inline comments
Correctness of the program (13 points)	Lab 1 Exercise (2 points) – Process creation	• Given the filename of a standard Unix utility program, the system can locate and execute the corresponding program • Can work with full path • Can work with relative path • Can work with executing a program with any number of arguments • Can handle error situation correctly, e.g., incorrect filename, incorrect path, not a binary file, etc. • display a message to report the termination status of the child process
	Lab 2 Exercise (2 points) – Signal handling	• Should cover all test cases in Lab1 • timekeeper process should not be terminated by SIGINT signal • child process can be terminated by SIGINT signal • display a message to indicate that the child process is terminated by signal
	Lab 3 Exercise (2 points) – Report timing	• Should cover all test cases in Lab1, Lab2 • print out the wall clock time, user time, system time and the number of context switches experienced by the child process in the correct format
	Final program (7 points)	• Should cover all test cases in Lab1, Lab2, Lab3 • Can execute two programs which are connected by the logical pipe symbol "!" • Can execute two programs with any number of arguments and are connected by the logical pipe • Can execute any number of programs with any number of arguments and are connected by a sequence of pipes • For each child process, display the running statistics and termination status (or signal information) of the process.

Plagiarism

Plagiarism is a very serious offence. Students should understand what constitutes plagiarism, the consequences of committing an offence of plagiarism, and how to avoid it. **Please note that we may request you to explain to us how your program is functioning as well as we may also make use software tools to detect software plagiarism.**