COMP3230A/B Principles of Operating Systems

Program Assignment One

Due date: Oct. 18th, 2021, at 23:59 pm

Total 13 points

(version 1.0)

Programming Exercise – Monitor: A new system utility program

Objectives

1. An assessment task related to ILO 4 [Practicability] – "demonstrate knowledge in applying system software and tools available in the modern operating system for software development".
2. A learning activity related to ILO 2.
3. The goals of this programming project:
   - to have hands-on practice in designing and developing a system utility program, which involves the execution of multiple processes and collection of processes' 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 the process's running statistics.

Task

For this assignment, you are going to write a new system program - ***monitor***, 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:

```
./monitor firefox
```

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

```
The command "firefox" terminated with returned status code = 0

real: 211.385 s, user: 24.568 s, system: 7.807 s
no. of page faults: 373360
no. of context switches: 423601
```

Specification

Basic features of the *monitor* program

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

2. When invoking the *monitor* program with input arguments, it will interpret the input and execute the corresponding program(s) with necessary arguments. The *monitor* 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.

     ```
     ./monitor /home/tmchan/a.out
     ```

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

     ```
     ./monitor ./a.out  abc  1000
     ```

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

     ```
     ./monitor gedit readme.txt
     ```

     where *gedit* is in the directory /usr/bin, and *monitor* 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, the *monitor* process will print out a message that shows the child process's id. For example:

   ```
   Process with id: 11463 created for the command: cat
   ```

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

   ```
   Process with id: 12557 created for the command: xxxxx
   exec: : No such file or directory

   monitor experienced an error in starting the command: xxxxx
   ```

   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 is terminated, the *monitor* process will print out the ***wall clock (real) time***, ***user time***, and ***system time*** used by that process as well as the number of ***page faults*** and ***context switches*** experienced by that process. Here are the definitions of these timings:

   - Real time: the elapsed wall clock time (in seconds) between the 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 3 decimal places. For example,

```
real: 211.385 s, user: 24.568 s, system: 7.807 s
```

- Page fault: including both the soft and hard page faults
- Context switch: including both the voluntary and involuntary context switches

```
no. of page faults: 373360
no. of context switches: 423601
```

4. The *monitor* process and its child process(es) respond to the SIGINT signal (generated by pressing Ctrl-c or by the *kill* system command) according to the following guideline:
   - Upon receiving the SIGINT signal, the child process(es) will respond to the signal according to the predefined behavior of the program in response to the SIGINT signal; while the *monitor* process should not be affected by SIGINT and continue until it detects that its child processes have terminated.
   - Instead of printing out a child process's termination status, the *monitor* process 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: 4.415 s, user: 4.411 s, system: 0.004 s
no. of page faults: 54
no. of context switches: 16
```

5. Like item 4, when the invoked program is terminated involuntarily because of receiving a signal (e.g. SIGSEGV), the *monitor* process 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: 0.478 s, user: 0.328 s, system: 0.000 s
no. of page faults: 318
no. of context switches: 4
```

6. The monitor program uses a special symbol "**!**" to act as the ***pipe*** operator. Please note that we are **not using the traditional pipe symbol "|"** as we don't want the built-in shell process to be confused if "|" is used in this assignment. When the *monitor* process detects that its input arguments contain the special symbol "**!**", it will identify the sequence of commands and start the corresponding

programs with necessary arguments. In addition, these programs will be linked up according to the logical pipe defined by the input arguments. For example,

```
./monitor cat c3230a.txt ! grep kernel ! wc -w
```

In this example, the *monitor* 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 any child process has terminated, the *monitor* process will print out the termination status (or signal information) and running statistics of that terminated child process. Upon detecting all child processes have terminated, the *monitor* process will terminate too.

System Calls

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

- *fork()* – The fork function is the main primitive for creating a child process.
- *exec* family of functions – Use one of these functions to make a child process execute a new program.
- *wait4()* – Wait for a child process to terminate or stop, and returns its termination status and running statistics.
- *signal()* or *sigaction()* – Specify how a signal should be handled by the 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.
- *clock_gettime()* or *gettimeofday()* – get the wall-clock time for measuring the process's real time.

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
   - Filename
   - Student's name
   - Student Number
   - Development platform
   - Remark – describe how much you have completed
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 develop and test your programs on the workbench2 Linux platform. Your programs must be written in C and successfully compiled with gcc. It would be nice if you develop and test your program on your own machine (WSL2 or Linux VM) or academy servers. After fully tested locally, upload the program to the workbench2 for the final tests.

Submission

Submit your program to the Programming Assignment One submission page at the course's moodle website. Name your program with this format: monitor_StudentNumber.c (replace StudentNumber with your HKU student number). As the Moodle site may not accept source code submission, please compress your program to the zip or tgz format.

Grading Criteria

1. Your submission will be primarily tested under workbench2. Make sure that your program can be compiled *without any error*. Otherwise, we have no way to test your submission and you will get a zero mark.
2. 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 confusion.

Documentation (1 point)	• Include necessary documentation to clearly indicate the logic of the program • Include required student's info at the beginning of the program	
Correctness of the program (12 points)	Process creation (3 points)	• Given a standard Unix command, the monitor process can locate and execute the corresponding program • Can work with full path and relative path • Can work on executing a program with any number of arguments • Can handle error situations correctly, e.g., incorrect filename, incorrect path, not a binary file, etc. • Display a message after creating the child process
	Signal handling (1 point)	• The monitor process should not be terminated by the SIGINT signal • Child processes may be terminated by SIGINT signal
	Process termination (4 points)	• Display a message to report the termination status of each child process or the signal experienced by the child process • 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

	Support pipeline (4 points)	• Can execute two programs that 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 • Display a message to report the termination status of each child process or the signal experienced by the child 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 of software tools to detect software plagiarism.**