Instructor: Tuna Tuğcu
TAs: Mısra Yavuz & Selen Parlar
Contacts: misra.yavuz@boun.edu.tr & selen.parlar@boun.edu.tr

Due: **December 30, 2020, Wednesday, 23.59**

1 Introduction

In your professional lives as computer engineers, you will often develop grand projects composed of several processes interacting to achieve a complex task in a collaborative manner. However, some of these processes might fail in production (i.e. when deployed in the real-life environment). Therefore, such grand software projects typically also embody an additional *watchdog* process that keeps track of the "well being" of all other processes. The watchdog process detects the crashing of the project components, logs the error, and then restarts that process (or all processes of the project, if necessary). In this manner, it is possible to deploy an important project *unattended*, i.e. without a human administrator keeping track of the proper execution of the software 7x24. (Typically, we also set up a connected warning system, like Nagios, to inform the human administrator about the failure so that (s)he can analyze the real cause, but that is beyond the scope of this project.)

In this project, you are expected to create a simple program in $C/C++$ that implements a watchdog system. The project consists of two parts; process definition and process control. The process definition part specifies the process-specific information such as process id, process name, and signals to be handled by each process. The process control part creates processes and tries to keep these processes alive so that the system continues to run smoothly.

The project will be evaluated automatically in the Linux environment (Ubuntu Version 20.04.1) with a gcc/g++ compiler (Version 9.3.0). Please follow all the requirements specified below. Your submissions will be compiled and tested via automatic scripts. Therefore, it is crucial that you follow the protocol (i.e. the rules and the specifications) defined in the project document. Failure in the compilation and/or execution is your responsibility. You should use the file names, parameters, etc. as mentioned in the project specifications.

2 Project Description

The project is to be designed in a flexible manner. That means, you will have N processes and a watchdog, but those N processes will be created by running the same executable file, process.c/process.cpp. During the evaluation, we may take N to be 3 or maybe even 120. So, do not develop your code depending on a specific value of N.

You are expected to have three C/C++ programs; *process*, *watchdog*, and *executor* with the properties specified below. Note that, *executor* program is provided to you, and you should not alter anything on the original program.

2.1 executor.cpp

1. File Properties:
   - The name of the file is *executor.cpp*. (This file is already provided to you. Do not modify.)
2. Operations:
   - Program reads instructions from *instructions.txt*.
   - Creates a named pipe to be shared with *watchdog* program.
   - Reads *"# <PID>"* tuples from the read end of the pipe (P1 3564, P2 3565 etc.).
   - Detects whether it is a *signal* or *wait* command.
   - Sends the intended signal to the given process.
   - When instructions are completed, first it kills the watchdog process, then all of the other running processes by sending the SIGTERM signal.

2.2 process.c/process.cpp

1. File Properties:
   - The name of the file must be *process.c/process.cpp*.
2. Process definition:
   - Assume you are asked to create a system with N processes. Name the processes as $P1, P2, P3, P4,$ ... $, PN$.
   - All processes should work concurrently.
   - Process $P1$ should be considered as the *head* process (not parent).
   - For the sake of the scenario, we assume the *head* is such a critical process that, if it fails, all other processes should be terminated, $P1$ be re-created, and then all remaining processes $P2, P3$, ..., and PN should be created following $P1$.
3. Operations:
   - Each process is forked from the *watchdog* process (which is described in the following sections).
   - Each process should display a message (as explained below) indicating that it has started and then go to *sleep* until it receives a signal.
   - Each process must handle the signals given in Table 1.
   - Processes handle signals by only printing the value of the corresponding signal.
   - Specific to SIGTERM signal, the process should terminate after printing the required message.
   - The instructions file has the signals and wait commands listed line by line. A sample instruction file is shown in Table 2. For instance, if the instruction is "SIGHUP P3", P3 receives a SIGHUP signal. If the instruction is "wait 1", all the processes should sleep for 1 second.

Signal	Value	Description
SIGHUP	1	Hangup (POSIX) Report that the user's terminal is disconnected.
SIGINT	2	Interrupt (ANSI) Program interrupt. (ctrl-c)
SIGILL	4	Illegal Instruction (ANSI) Generally indicates that the executable file is corrupted.
SIGTRAP	5	Trace trap (POSIX)
SIGFPE	8	Floating-Point arithmetic Exception (ANSI) This includes division by zero and overflow.
SIGSEGV	11	Segmentation Violation (ANSI) Indicates an invalid access to valid memory.
SIGTERM	15	Termination (ANSI)
SIGXCPU	24	CPU limit exceeded.

Table 1: Signals to be handled

2.3 watchdog.c/watchdog.cpp

1. File properties:
   - The name of the file must be *watchdog.c/watchdog.cpp*.
   - Main method should be in the watchdog file.
2. Operations:
   - The watchdog program opens the named pipe.
   - The watchdog process writes its ID to the named pipe as *P0 <PID>*.
   - The watchdog should be the parent process, so each process should be created using *fork()* from the watchdog.
   - The watchdog process writes the ID of each newly forked processes to the named pipe as *P# <PID>*.
   - The watchdog process should initiate each process from the *process* executable using *exec()*.
   - If $num\_of\_process = 10$, your program should create 10 distinct processes (in addition to the watchdog)..
   - The number of processes in the program should be preserved, and be equal to the $num\_of\_process$ all the time.
   - Watchdog sleeps until a process terminates.
   - If a process terminates, watchdog should detect and restart it. Do not forget to send "*P# <PID>*" tuple for the restarted process since the PID value has changed.
   - If the head process receives SIGTERM signal, all of the other processes should also be killed and restarted by the **watchdog**. The processes should be terminated in increasing order of the process numbers and this order should be visible in the output.

- Watchdog kills processes via the SIGTERM signal.
- The whole project should be terminated in a safe state. That is, *executor* kills all the child processes and the watchdog when it reaches the end of *instructions.txt* file.
- **Bonus Case:** What happens if the watchdog dies? (You do not have to handle this case.)

SIGHUP P3
wait 1
SIGINT P2
wait 2
SIGTERM P1
SIGTRAP P2
wait 1
SIGTERM P3
wait 1
SIGILL P2

Table 2: Sample *instructions.txt* file

P1 is waiting for a signal
P2 is waiting for a signal
P3 is waiting for a signal
P3 received signal 1
P2 received signal 2
P1 received signal 15, terminating gracefully
P2 received signal 15, terminating gracefully
P3 received signal 15, terminating gracefully
P1 is waiting for a signal
P2 is waiting for a signal
P3 is waiting for a signal
P2 received signal 5
P3 received signal 15, terminating gracefully
P3 is waiting for a signal
P2 received signal 4
P1 received signal 15, terminating gracefully
P2 received signal 15, terminating gracefully
P3 received signal 15, terminating gracefully

Table 3: Output of the *process* program for the sample instructions

P1 is started and it has a pid of 4105
P2 is started and it has a pid of 4106
P3 is started and it has a pid of 4107
P1 is killed, all processes must be killed
Restarting all processes
P1 is started and it has a pid of 4108
P2 is started and it has a pid of 4109
P3 is started and it has a pid of 4110
P3 is killed
Restarting P3
P3 is started and it has a pid of 4111
Watchdog is terminating gracefully

Table 4: Output of the *watchdog* program for sample instructions

3 Input & Output

Your code must read the number of processes, the path of the input and output files from the command line. We will run your code as follows (**Make sure that your final submission compiles and runs with these commands**):

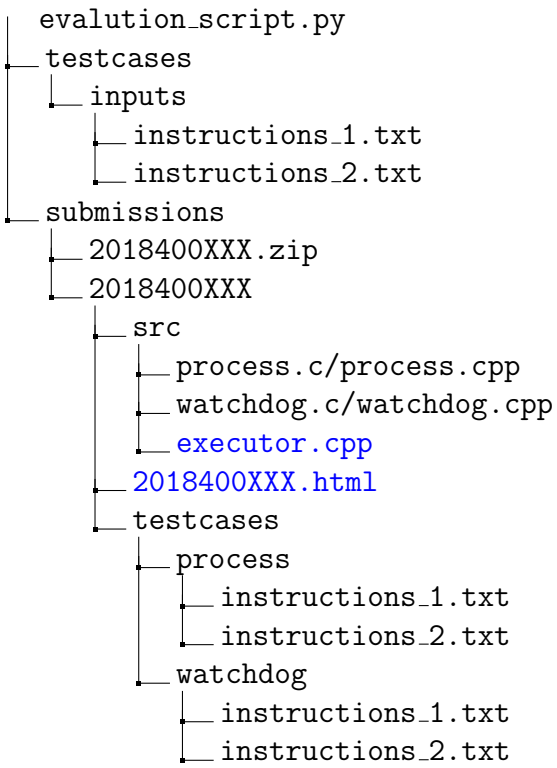
```
// Run the executor.cpp in background
./executor 5 instruction_path &

// For C++:
g++ process.cpp -std=c++14 -o process
g++ watchdog.cpp -std=c++14 -o watchdog
./watchdog 5 process_output watchdog_output

// For C:
gcc process.c -o process
gcc watchdog.c -o watchdog
./watchdog 10 process_output watchdog_output
```

3.1 Input

- The file structure for the evaluation is provided below. Please design your code and zipped file accordingly. We'll provide you absolute paths, so you do not need to create directories except for the *src* file.



- Main method should be in the watchdog file and and should expect 3 command line arguments:
  1. *num_of_process*: an integer value to specify the number of processes in the system
  2. *process_output_path*: absolute path of the *process* program's output file
  3. *watchdog_output_path*: absolute path of the *watchdog* program's output file

- The *executor* program takes 2 command line arguments:
  1. *num_of_process*: an integer value to specify the number of processes in the system
  2. *instruction_path*: absolute path of the instructions file

3.2 Output

- All operations performed by the process file should be logged to the file specified by the command line argument *process_output_path*.
  - When a process is ready to execute, it prints: ***P<ID> is waiting for a signal***
  - When a process receives a signal, it prints: ***P<ID> received signal <VALUE>***
  - When a process receives SIGTERM signal, it prints: ***P<ID> is received signal*** 15***, terminating gracefully***
  - Table 3 shows the expected outputs of process file for the sample instructions given in Table 2 with $num\_of\_process = 3$.
- All operations performed by the watchdog should be logged to the file specified by the command line argument *watchdog_output_path*.
  - When watchdog creates a process, it prints: ***P<ID> is started and it has a pid of <PID VALUE>***.
  - When the head process is killed, watchdog prints: $P1$ ***is killed, all processes must be killed***.
  - When restarting all processes, watchdog prints: ***Restarting all processes***.
  - When a process (except P1) is killed, watchdog prints: ***P<ID>*** **is killed**. And when a process is being restarted/recreated, watchdog prints: ***Restarting P<ID>***.
  - When the watchdog terminates normally (i.e., all the instructions are completed), it prints: ***Watchdog is terminating gracefully***
  - Table 4 shows the expected outputs of the watchdog file for the sample instructions given in Table 2 with $num\_of\_process = 3$.

4 Report & Grading

You are expected to submit a well-documented code. For this purpose, you are required to use Doxygen, which is the de-facto standard tool for generating documentation from annotated C/C++ sources. You need to explain the parameters, variables, and functions used in the code. You also need to provide author information, the main idea of the project, and a conclusion. Please check Doxygen before starting the project. Corresponding points are **tentatively** specified in parentheses.

1. **Documentation:** Explain your code in your own sentences and provide additional information about you and the project using Doxygen. *(10 pts)*
2. **process.c/process.cpp:** File that has process definition and handles signals. *(15 pts)*
3. **watchdog.c/watchdog.cpp:** File that creates and manages processes. *(25 pts)*
4. **Test Cases:** Your program will be tested using several test cases. *(40 pts)*
5. **Auto-runnable submission:** Submit your code that runs smoothly with the specified commands on the Linux environment. Each re-submission deduces 5 points. *(10 pts)*

5 Submission

Submissions will be through Moodle. Submit a single .zip file named with your student ID (e.g. 2018400XXX.zip) that matches the specified structure. Your zipped project will be placed and extracted to the *submissions* folder. The zipped project should contain *src* directory that has *process.c/.cpp* and *watchdog.c/.cpp* files and the documentation in HTML format (2018400XXX.html). You do not need to send the *executor* file. Please pay attention to the file naming. Note that your project will be inspected for plagiarism with previous years' materials as well as this year's. Any sign of **plagiarism** will be **penalized**.