SYSC 4001 Operating Systems, Fall 2022
Programming Assignment 1: Emulation of Internet of Things (IoT) Using Software Processes

Purpose:

The assignment is to emulate a simple IoT system using software processes and system calls for inter-process communications (IPCs)

The process management aspect of this assignment includes process creation, sending/handling of signals, and process communication and synchronization via message passing using the FIFO client/server model.

Necessary information: This assignment is closely related to previous labs and is based on the materials in the Linux reference book, particularly chapters 11 (processes and signals) and 13 (FIFOs).

IPCs are crucial to allow processes without sharing the same address space to communicate with each other and/or to synchronize their actions. There are several IPC mechanisms provided by Unix/Linux. This assignment will experiment signals and named pipes (FIFOs), in addition to process creation using the fork() function and process communication using signals.

Assignment description:
This assignment is to emulate a simple IoT system. The main components for the hypothetical IoT system include *Device*, *Controller*, *Cloud* (*Mobile device)*. For this assignment, the *Cloud* layer and the *Mobile* device are combined as one component. Each of the components is emualted using a software process.

The main components or processes are illustrated as follows:

Device Process:
Generally, there are two main types of IoT devices: sensors and actuators. In short, a sensor monitors situations in an environment, e.g., temperature sensor; while an actuator converts electrical energy into motion and typically triggers an action. Some devices can have both functions. For this assignment, all *Device* processes (both *Sensor* and *Actuator* types of processes) will communicate with the *Controller* process using FIFOs.

The device name and the threshold value will be entered through the command-line arguments. Each *Device* process then establishes the connection with the *Controller* using the client-server model via FIFOs. The first message that a *Device* process (a client) sends to the *Controller* should include the information needed for the *Controller* to distinguish different clients (devices). When a *Device* process receives an acknowledgement (ACK) from the Controller, it starts its normal operations. If the *Controller* sends a "stop" message to a *Device* process, the *Device* process will stop its operation and exit.

A *Sensor* process performs a simple monitoring task periodically, e.g., every second, and sends the sensing result to the *Controller* via a FIFO. The *Controller* checks if the sensing result is over a certain threshold; if so, an alarm will be generated immediately. For the assignment, only a warning needs to be printed for an alarm. If a threshold crossing is observed, the *Controller* needs to take an appropriate action, e.g., turn on/off the AC or a light. See the description for the *Actuator* and *Controller* processes.

For an *Actuator* process, if it receives a message from the *Controller*, it triggers a motion/action, e.g., turning on or off a switch. For this assignment, it is only a print statement.

For the assignment: Two *Sensor* and one *Actuator* processes are needed to test your program. The *Sensor* can be temperature, humidity, smoke sensors; and the *Actuator* can be a switch, bell, etc. You can choose any type of devices. The device type and threshold values used in the program need to be passed through the command-line arguments. They need to be clearly specified and printed on screen. Clear outputs are required for messages sent to and read from another process together with the PID and device name.

Controller Process:
The *Controller* is primarily used to control or coordinate other processes. The *Controller* consists of a parent process and a child process. The child process communicates with various *Device* processes (*Sensor* and *Actuator* processes) via FIFOs, as described earlier.

The child process receives a message periodically from *Device* processes via a FIFO. If this is the first message from a particular *Device* process, it registers the device and all the necessary information, i.e., PID, device name, and any threshold value if available. Each subsequent message from a *Device* process contains the sender information (PID and sensed data). The child process needs to echo the information it receives from any other process with its PID.

If the value from a *Sensor* process exceeds the pre-configured threshold, the child process will trigger an action, e.g., turn on or off the AC or an appliance, by sending a command (message) to an appropriate *Actuator* process. The *Actuator* simply displays the command or message on the terminal for this assignment.

When a threshold crossing occurs and an operation is taken, the child process also *raises a signal* and sends it to its parent process. For this assignment, a simplified communication mechanism for parent and child is done using two signals: SIGUSR1 and SIGUSR2 (on page 481 of the Linux book). SIGUSR1 is used for ON state, i.e., a switch is turned ON, while SIGUSR2 is used for OFF state.

The parent process gets a signal, either SIGUSR1 or SIGUSR2, from its child process, it then communicates with the *Cloud* process using a FIFO. The *Cloud* is the server and the parent process of the *Controller* is a client. For this assignment, the *Cloud* process only prints the current state it receives from the parent process. (In practice, the *Cloud* process will have many client processes.)

Cloud Process:
The *Cloud* process relays the message between a *Controller*. The *Cloud* is the server side of a FIFO that connects to clients. There is only one client process, i.e., the *Controller*, for this assignment. When the *Cloud* process receives an update, it sends it to the *Mobile Device* to notify the user. For this assignment, it is simply a print statement from the *Cloud* process.

Miscellaneous*:*
Random number generation:

You may use the rand() function in <stdlib.h> to return a pseudo-random integer between 0 and RAND_MAX. A simple way to generate a random number between 0 and 50 is shown below. You can tailor it for your initial sensing values, but it is not required.

```
// Example: random int between from 0 to 49, inclusive
int r = rand() % 50;
```

Grading criteria

Correctness (including error checking and final cleaning up): 80%

Note: The processes are responsible for tidying up if they exit, e.g., a process should release resources that it has allocated.

Documentation and output: 10%

- A **readme** file is needed to explain how to run your program, including command-line arguments.
- A "**script**" file is needed that contains the commands to run your programs. Change the permissions of the file and make it executable.
- You need to print **CLEARLY** after each step, e.g., before and after each message or signal is sent and received.

Program structure: 5%

Style and readability: 5%

This is an imperfect world. It is especially true for software, including requirements documents like this one. If you have questions, it may not be your problem. (Of course, it's not mine, either 😊.) That's why *IPC* (Inter-Personal Communications) is important! Read through this description carefully and the examples in the Linux reference book. Start with the high-level design. Finally, as we know that customer requirements are constantly changing in practice, hence, I reserve the right to make changes (only minor changes) to the assignment, but the requirements will be finalized as least one week before.