SYSC 4001 Operating Systems, Fall 2013
Programming Assignment 1: Patient Monitor
Due: Thursday, Oct. 10th @ 2pm after the lab session

Purpose: The assignment is to build a simplified Patient Monitor using system calls related to process management. The process management aspect of this assignment includes creation of a new process, handling of signals/interrupts/timers, and synchronization via message passing. IPC mechanisms include signals, pipes (FIFOs) and message queues. In addition, the assignment deals with the client/server model.

Some basic information: To solve this assignment, you need to understand basic Unix/Linux processes and IPC concepts. The reference book provides you a solid background and concrete examples. It is strongly recommended that you read related materials in the reference book, particularly chapters 11 (processes and signals), 13 (FIFOs) and 14 (Message Queues), or other relevant references.

IPC describes different ways of message passing between processes that are running on an operating system. IPC is a key feature for any modern OS that allows processes without sharing the same address space to communicate with each other and to synchronize their actions. There are several IPC mechanisms provided by Unix/Linux. This assignment will experiment signals, named pipes (FIFOs), and message queues in addition to process creation using the fork() function.

Assignment description:
This assignment is to simulate a hypothetical Patient Monitor System (PMS). The PMS can monitor the heartbeat and play music with different tempos according to the heartbeat rate. Specially, consider two processes, *Monitor* and *Controller* that are used for the PMS. When the heartbeat rate crosses a pre-defined threshold, the *Monitor* process will send a message to the *Controller* process. There are multiple *Monitor* processes, but only one *Controller* process.

Monitor Process:
The *Monitor* process monitors the heartbeat periodically. The *Monitor* process has both parent and child portions. The parent first sends a *connect* request to the Controller process via a FIFO. The FIFO needs to be created using the Client/Server model as illustrated in the Linux reference book. (We will discuss it in class and you will practice it in the lab.) After the FIFO is created, *Monitor* will start the operation after it receives a *start* message from the *Controller* via a FIFO. When the parent receives a *stop* message from the *Controller*, it sends a signal to its child to terminate the operation.

The *Monitor* process also has a child process which actually monitors the heartbeat periodically and sends the heartbeat rate to the message queue. A heartbeat rate is simply a random number between 60 and 100. The monitoring interval could be 2-3 seconds for this assignment. The main point is to demonstrate the operations and outputs clearly.

Controller Process:
The *Controller* is primarily used to control or coordinate other processes. The *Controller* process also has parent and child. The parent process acts as the server using FIFOs. It receives *connect* requests from other processes. When it receives the first *connect* request message from another process, it replies a *start* message. The parent also keeps track of user commands. For this assignment, there is only one necessary command - <*control-C*> - meaning that the user stops the PMS machine. When it happens, the parent process sends a signal to the child process, and the parent also sends a *stop* message via FIFOs to all other processes and terminates its operation.

The child process receives a message periodically from the *Monitor* process via a message queue. The message mainly contains the sender information (PID and patient name) and the heartbeat rate. The child process of *Controller* only echoes the process information and the heartbeat value it receives. The child process also sends an ACK message back to the sender for each message it receives.

Inter-processor Communications:
Because multiple *Monitor* processes are running concurrently, each *Monitor* process needs a unique name (patient name). Each *Monitor* process establishes a pair of FIFOs with the parent component of the *Controller* process, as explained above, using the Client-Server model (the same model used in the Linux reference book). On the other hand,

the child component of the *Controller* process communicates (for heartbeat rates and ACKs) with all *Monitor* processes using only ONE message queue.

***Miscellaneous**:*
To create a message queue with a pathname, one way is to use the *ftok ()* library function. It generates a key from a pathname.

```
   Example: msgQid = msgget (ftok (argv[1], 1), 0666 | IPC_CREAT);
  // the logic operation sets the flags to the default values, which can be ignored for now
```

Random number generation:
The rand() function in <stdlib.h> returns a pseudo-random integer between 0 and RAND_MAX. A simple way to generate a random number between 0 and 50 is shown below:

```
// 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, e.g., FIFO, message queue, that it has allocated.
Documentation and output: 10%
- A **readme** file is needed to explain how to run your program, including command-line arguments, if any.
- 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 several times and the examples in the reference book. We will discuss it in class. Finally, as we know that customer requirements are constantly changing in practice, I also reserve the right to make changes (only minor changes) to the assignment, but the requirements will be finalized as least one week before.

Some implementation details:

1. Run each process (a Controller and two Monitor processes) on a separate *xterm*. For the Controller process, the parent part is monitoring if <*control-C*> is entered. However, if the user enters <*control-C*>, the child part may stop, since both parent and child are sharing the same *xterm*. To avoid it, the child part needs to ignore <*control-C*>. This can be done using SIG_IGN described on page 483 of the reference book. Specifically, in the child part, add the following statement:

   (void) signal (SIGINT, SIG_IGN);

2. For signal handling, use the robust signals interface, e.g., **sigaction** (); see pages 487-489.