Machine Problem 3: UNIX System Programming

100 points + 10 bonus points
Due date: To Be Announced

1 Introduction

As opposed to the previous projects in this course, MP3 focuses on *system programming* as opposed to *OS design*. You will have the opportunity to experiment with some non-trivial system programming. In particular, we will be looking at process forking and termination, Unix signal processing, shared memory, pipes (named and unnamed), and – for the more ambitious among you – semaphores.
The application we will be using is the control back-end for a Tic-Tac-Toe game.
While most of the primitives used in the MP can be found in most OS's, we will be using them in their Unix System V flavor. In other words, we will be implementing the code for this MP to run on our Sun/Solaris machines.

2 System Primitives used in this MP

This machine problem will exercise primitives for process control, signal management, pipes, and shared memory. At advanced level, you are also asked to use shared-memory primitives. We list the primitives in detail below:

- Process Control

  `fork`:

  `exec`:

- Signal Management

  `sigset`:

  `kill`:

- Pipes and Files

  `pipe`:

  `mknod`:

  `unlink`:

  `read/write`:

- Shared Memory

  `shmget`:

  `shmat`:

- Semaphores (System V style)

  `semget`:

`semop`:

You will also find the following commands useful:

`ipcs`: This command reports on the status of inter-process communication facilities. Allows you to monitor the correct use and operation of the interprocess communication primitives in your program.

`ipcrm`: Remove a message queue, semaphore set, or shared memory ID. This comes in handy if your program does not clean up things correctly.

`kill`: Terminate or signal a process. Comes in handy to clean up your processes.

`ps`: Report on process status. This at least lists all your processes.

For details about the system calls and the commands, use the `man` command on the UNIX command line. This gives you access to the appropriate man page for the command/primitive. If you don't know how to use man pages, use the `man` command on itself, in the following way:

```
% man man
```

For searching a command related to a keyword, use the `-k` option:

```
% man -k keyword
```

3 Test Application: Tic-Tac-Toe

In order to exercise the primitives described above, you will be writing a back-end for a simple, text based, **Tic-Tac-Toe** game. This game will be run on a single machine, but will using multiple Unix processes. These processes communicate through pipes and shared memory. The goal of the project is to learn, understand and use a number of system primitives, not the implementation of a time-efficient or interesting game. Thus we will implement the game as outlined below.

3.1 System Overview

Your system will have the following four processes.

Main Process: Creates and monitors the other three processes. It also creates the Unix pipes for inter-process communication (IPC).

Manager Process: This process coordinates the players and determines the outcome of the game. It also creates the *shared memory segment* for the communication between the manager process and the player processes.

Two Player Processes: Interacts with the human players (gets the input from the screen) and interacts with the manager process for the proper coordination (with the other player).

Figure 1 shows the arrangement of the processes.

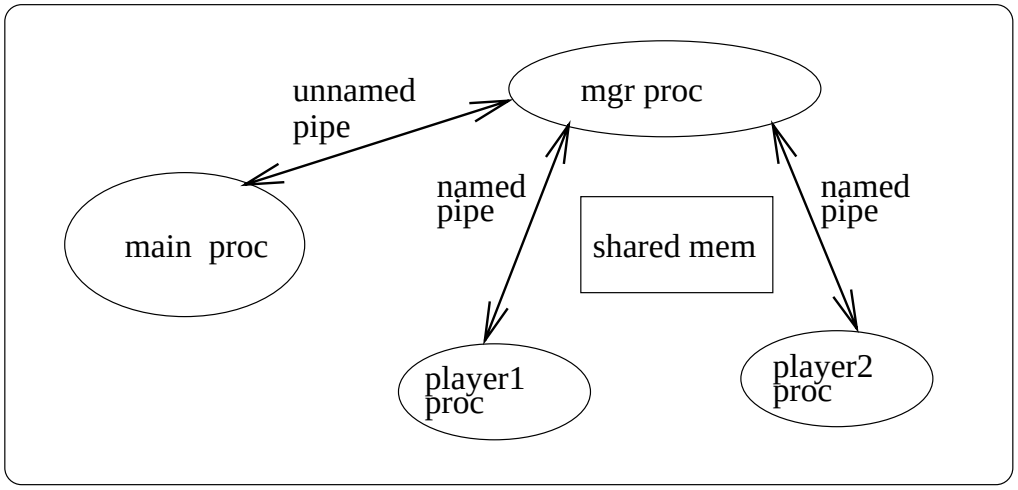

Figure 1: Processes involved in the tic-tac-toe game.

3.2 Implementation Requirements

You have to write three programs : (1) main program, (2) manager program, and (3) player program. (Note: these are three independent programs, i.e., with three *main()* functions; not three subroutines with one *main()* function.) You should use the same player program for both *player1* and *player2*.

The main process and the manager processes will use **unnamed pipe(s)** for IPC. The manager process and the player processes will use **named pipes** for IPC (synchronization) and **shared memory** for data sharing/communication. The player process will use simple printf/scanf to get the input from the user (players).

The project should work on our Solaris machines. Note that your program need not implement the game strategy (only two users will play the game, not the machine versus user). However, the program should able to find out the status of the game at any time.
Following is the high-level logic for the three programs:

```
main-program:
{
   initialize();
   create-pipes();
   fork-and-exec-mgr-proc();
   when (mgr-proc-is-ready)
      fork-and-exec-player-procs();

   wait for ‘‘GAME-OVER’’ message from mgr-proc;
   cleanup-and-terminate();
}

manager-program:
{
```

```
   initialize();
   create-shared-mem();
   establish-connection-to-pipes();
   send ‘‘READY’’ msg to main-proc;

   wait for ‘‘READY’’ msgs from player procs;

   coordinate the players for the play
              and determine the result;

   when game is over
        send ‘‘GAME-OVER’’ msg to main-proc;
}

player-program:
{
   initialize();
   establish-connection-to-shared-mem-and-pipes();
   send ‘‘READY’’ msg to mgr-proc;

   wait for my-turn:
       get input from the user (screen);
       update shared memory;
       send ‘‘DONE’’ to mgr proc;
}
```

3.3 Details

3.3.1 Input

The game is run/played by typing the main program name without any arguments.

3.3.2 Output

You need not submit any outputs. Instead, the grader should be able to play the game as described in the sample sessions. A sample session is given at the end of this handout.
Notes about the output:

- For any string printed on the screen (using `printf`), the string should be prefixed by the process-id of the process that prints that string.
- Have your programs print **all the information** printed on the sample output.
- Try to keep the output as similar as possible to the sample output. This makes the grader's live significantly simpler.
- In the main program you should catch and handle the `^C` properly. You (and the grader) should not able to kill the main program using `^C` when the game is in progress (see sample output 3).

4 Suggestions

1. If you are familiar (or good) with Unix system, this project should be fairly easy. Then you may want to try to go for the advanced level, where you will be implementing the player/manager communication through an explicit shared buffer.

2. For those who are less familiar with Unix, this project should provide ample opportunity to learn. It will help you to *find your way around* in the *systems* world. So do the following:

   - **Use man pages extensively.** Sun also makes information available on Solaris at `http://docs.sun.com/`. Lots of information is available on the web simply by googling the appropriate primitive or command. (Be careful, however, that our environment is UNIX System V / Solaris. Signals and semaphores are used differently in a POSIX environment, for example.)
   - When in doubt, always refer to man pages.
   - Any book on Unix systems programming will give you more details and examples. But a book is not necessary to do this project.

3. You should use your *user-id* as the key for the shared memory.

5 What to Hand In

- Hand in a file called `design.pdf` that contains the design of your project.
- Turn in a gzipped version of your source files. You should typically have three files, one each for the main program, manager program, and player program. You should also have a `makefile` as part of the set of source files.
- You should have a *makefile* to make the executables. Given the source files, the grader should only type `make` to create the executables.

6 Grading Info

The graders will assign points according to the following guidelines:

- Design: 10 pts
- Three programs (compiled and working makefile) : 20 pts
- Code readability (style and documentation): 20 pts.
- Four tests as given in the sample output: 10 * 4 = 40 pts
- Clean-up: 10 pts.

  When the main program/process exits, there should not be any processes, pipes or shared memory hanging in the system. If there are after your program exits, you will loose 10 pts.

7 Bonus Assignment

For bonus, you are to replace the named pipes through what we will call `ProdConQueue`'s. Each ProdConQueue implements a bounded-buffer producer-consumer queue. Our implementation will be straightforward, with a shared memory segment as buffer, and two semaphores to synchronize access to the buffer (and possibly a third semaphore to ensure mutual exclusion).
The interface (part of file `pcq.h` provided for ProdConQueues should look as follows. This is the C version. Modify accordingly for C++ version if that's what you use to implement this machine problem.

```
/* -- INITIALIZATION AND STUFF */

extern int pcq_init();
/* Set up any infrastructure that you may need to manage PCQ's.
   Depending on how you go about implementing PCQ's, this function
   may not be needed. */

/* -- CREATE/DELETE A PCQ */

extern int pcq_create(int global_id, int buf_size);
/* Create a PCQ with given global identifier and given buffer size
   (in bytes). Return 0 if successfully created. */

extern int pcq_delete(int global_id);
/* Delete and clean up PCQ with given global identifier. Returns
   0 if successfully deleted. */

/* -- READ/WRITE OPERATIONS */

extern int write(int global_id, char * buf, int size);
/* Write 'size' characters from character string 'buf'
   into the PCQ with given global identifier. Return number
   of written characters. Block until sufficient space is
   available in the PDQ. */

extern int read(int global_id, char * buf, int size);
/* Read at most 'size" characters from PCQ with given global
   identifier. Return the number of characters read. If buffer is
   empty, block until buffer contains characters.
*/
```

8 Example Output

```
~/cs410/prj2> tmain
[pid:14823] This is TTT main program
[pid:14824] This is TTT mgr program, ppid[14823]

                 0   1   2
                --- --- ---
               |   |   |   | 0
                --- --- ---
               |   |   |   | 1
                --- --- ---
               |   |   |   | 2
                --- --- ---
[pid:14823] The tttmgr [mgr_pid:14824]is ready, Creating the players
[pid:14825] This is TTT player pgm (1), ppid [14823]
[pid:14826] This is TTT player pgm (2), ppid [14823]
[pid:14825] Please enter your tick [Player 1]; ( -1 -1 ) for QUIT ==> 1 1

                 0   1   2
                --- --- ---
               |   |   |   | 0
                --- --- ---
               |   | o |   | 1
                --- --- ---
               |   |   |   | 2
                --- --- ---
[pid:14826] Please enter your tick [Player 2]; ( -1 -1 ) for QUIT ==> 0 0

                 0   1   2
                --- --- ---
               | x |   |   | 0
                --- --- ---
               |   | o |   | 1
                --- --- ---
               |   |   |   | 2
                --- --- ---
/.... and so on .... /
[pid:14825] Please enter your tick [Player 1]; ( -1 -1 ) for QUIT ==> 1 0

                 0   1   2
                --- --- ---
               | x |   |   | 0
                --- --- ---
               | o | o | o | 1
                --- --- ---
               |   | x |   | 2
                --- --- ---
[pid:14824] Player 1 has won this game!!
[pid:14823] Game_OVER, cleaning UP
[pid:14823] Child [pid = 14824] Terminated
[pid:14823] Child [pid = 14825] Terminated
[pid:14823] Child [pid = 14826] Terminated
[pid:14823] Main program terminated
```

Note: The task for the manager to detect that a player has won is optional.