Machine Problem 3: Working with Processes (Due: 3/7 at 11:59pm)

Introduction:

In this machine problem, we set the stage for a high-performance data query and processing system. In a first step you are to implement a client program that first loads a data server program into memory and then requests data from that server. Then, in the second part, you look at how the kernel exposes the details about processes through /proc file system directory.

Part 1 – Creating Processes Using fork();

The client program (to be implemented as program client.cpp) is to fork off a process, which is supposed to load the data server program (the source code is provided in file dataserver.cpp). The client then issues requests to the server through what we will be calling request channels. Request channels are a simple inter-process communication mechanism, and they are provided by the class RequestChannel. Request channels provide the following interface:

```
RequestChannel(const string __name, const Side __side);
/* Creates a "local copy" of the channel specified by the given name. If the
    channel does not exist, the associated IPC mechanisms are created. If the
    channel already exists, this object is associated with the channel. The channel
    has two ends, conveniently called "SERVER_SIDE" and "CLIENT_SIDE". If two
    processes connect through a channel, one has to connect on the server side and the
     other on the client side. Otherwise the results are unpredictable. */

~RequestChannel();
/* Destructor of the local copy of the channel. By default, the Server Side deletes
    any IPC mechanisms associated with the channel. */

string send_request(string _request);
/* Send a string over the channel and wait for a reply. This function returns the
    reply to the caller. */

string cread();
/* Blocking read of data from the channel. Returns a string of characters read from
    the channel. Returns NULL if read failed. */

int cwrite(string msg);
/* Write the data to the channel. The function returns the number of characters
    written to the channel. Both cread and write and low-level and typically not
    used by the client in this implementation. */
```

You will be given the source code of the data server (in file dataserver.cpp) to compile and

then to execute as part of your program (i.e. in a separate process). This program handles three types of incoming requests:

- *hello*: The data server will respond with hello to you too.
- *data* <name of item>: The data server will respond with data about the given item.
- *quit*: The data server will respond with bye and terminate. All IPC mechanisms associated with established channels will be cleaned up.

The Assignment:

You are to write a program (call it client.cpp) that first forks off a process, then loads the provided data server, and finally sends a series of requests to the data server. Before terminating, the client sends a quit request and waits for the bye response from the server before terminating. You are also to write a report that briefly compares the overhead of sending a request to a separate process compared to handling the request in a local function.

Part 2 – Examining Processes Using /proc

The /proc directory is a pseudo filesystem that allows access to kernel data structures while in user space. It allows you to view some of the information the kernel keeps on running processes. To view information about a specific process you just need to view files inside of /proc/[pid]. For more information simply view the manpage with man proc.

The Assignment:

Using the files stored at /proc write a program/script to find information about a specific process using a user provided pid. In the following, you will find a list of the task_struct members for which you are required to find their value. In the task_struct a lot of the data you are finding is not represented as member values but instead pointers to other linux data structures that contain these members. All of the information you will be retrieving can be found in a processs proc directory (/proc/[pid]). Your program must be able to retrieve the following data about any given process:

Table #1: Process Attributes

Category	Required Variables/Items	Description
Identifiers	PID, PPID EUID, EGID RUID, RGID FSUID, FDGID	Process ID of the current process and its parent Effective user and group ID Real user and group ID File system user and group ID
State	R, S, D, T, Z, X	Running, Sleeping, Disk sleeping, Stopped, Zombie, and Dead
Thread Information	Thread_Info	Thread IDs of a process
Priority	Priority Number Niceness Value	Integer value from 1 to 99 Integer value from -20 to 19
Time Information	stime & utime cstime & cutime	Time that a process has been scheduled in kernel/user mode Time that a process has waited on children being run in kernel/user mode
Address Space	Startcode & Endcode ESP & EIP	The start and end of a process in memory
Resources	File Handles & Context Switches	Number of fds used, and number of voluntary/involuntary context switches
Processors	Allowed processors and Last used one	Which cores the process is allowed to run on, and which one was last used
Memory Map	Address range, permissions offset, dev, inode, and path name	Output a file containing the process's currently mappped memory regions

Deliverables:

- **Code**:
  - You are to turn in one file ZIP file, named <Last Name>_<First Name>_MP3.zip, which contains two directories under the root: *part*1 and *part*2.
  - **Part 1**: Contains all needed files for the TA to compile and run your program. This directory should contain the following files: *client.cpp*, *dataserver.cpp*, *reqchannel.cpp*, *reqchannel.h*, and *makefile*.
    (The expectation is that the TA, after typing make, will get a fully operational program called client, which then can be tested.)
  - **Part 2**: Contains your program/script named proctest.cpp/proctest.py/proctest.sh (c++, python, and bash names respectively). If your implementation consists of more than one file, include those and a makefile (if applicable).
- **Report**:

- **Part 1**: Present a brief performance evaluation of your implementation of a client/server communication. Measure the invocation delay (i.e. the time between the invocations of a request and the response back) of at least 10,000 requests by providing the average and standard deviation of observed delays. Compare those statistics with those for the time to submit the same request string to a function that takes a request and returns a reply (as compared to a separate process that does the same thing).
- **Part 2**: Answer the following additional questions
  1. For a process run by a user other than yourself, find the following items from Table #1: [Identifiers, State, Thread Information, Priority, Time Information, Resources, and Memory Map]
  2. For a process that you have created, retrieve all items enumerated in Table #1.
  3. What are the differences between the real and effective IDs, and what is a situation where these will be different?
  4. Why are most of the files in /proc read only?
  5. Why is the task_struct so important to the kernel and what is it used for?