Machine Problem 3: Working with Processes (Due 10/22/15)

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 Process 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.
THIS FUNCTION IS LOW-LEVEL AND IS TYPICALLY NOT USED BY THE CLIENT. */

int cwrite(string msg);
/* Write the data to the channel. The function returns the number of
characters written to the channel.
THIS FUNCTION IS LOW-LEVEL AND IS TYPICALLY NOT USED BY THE CLIENT. */
```

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 filesystem is a psuedo filesystem mounted at /proc 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.

Assignment: Using the files stored at /proc make a program/script to find information about a specific process using a pid given by the user. 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 finding can be found in a process's proc directory. After finding the information, answer a few questions about some of the information you have found.

Questions:

1. For a process run by a user other than yourself get the following items from the next page [item# 1, 2, 3, 4, 5, 7, 9]
2. For a process that you have created get the following items from the next page [item #s 1-9]
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?

Category	Required Variables/ Items	Description/Explanation
1) Identifiers	PID, PPID	Process ID of the current process and its parent
	EUID, EGID	Effective user and group ID
	RUID, RGID	Real user and group ID
	FSUID, FSGID	File System IDs of the user and the group
2) State	R, S, D, T, Z, X	One of Running, Sleeping, Disk sleeping, sTtopped, Zombie, dead(X), respectively
3) Thread Information	Thread_info	Show thread numbers (a number that corresponds to the thread name) in each process
4) Priority	Priority Number	Range [1,99]
	Nice Value	Range [-20,19]
5) Time Information	stime, utime	Amount of time the process has been scheduler in kernel/user mode
	cstime, cutime	Time the process has waited on children being run in kernel/user mode
6) Address Space	Startcode, Endcode	The start and end address of a process in memory
	ESP, EIP	
7) Resources	File handles, Context Switches	Number of file descriptors used, number of voluntary and involuntary context switches,
8) Processors	Allowed processors and the last used once	Which cores the process is allowed to run, and on which one the process was last executed
9) Memory map	address range, permissions, offset, dev, inode, and path name	Output a file containing the process's currently mapped **memory regions**. Include the indicated fields for each region. Also include **memory consumption** for each of these pathnames

What to Hand In

- **Code:** You are to hand in one file ZIP file, called `Submission.zip`, which contains two directories under the root: `part1` and `part2`.
    - **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** Include your script named proctest.cpp in the directory `part2`. If you have more than 1 files, include those, and provide makefile

- **Report:** Submit only one report, called `report.pdf`, which you include as part of the root directory of your submission.
    - **Part 1** You present a brief performance evaluation of your implementation of the client/server communication. Measure the invocation delay (i.e. the time between the invocations of a request until the response comes back) of at least **10,000 requests** by providing the **average** and **standard** deviation of the observed delays. Compare these 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). The report should compare the two.
    - **Part 2** List your finding as part of the investigation done in the second part with /proc.