

Project Six: Multiprocessing

CS314 Operating Systems

Fall 2006

Date Assigned: November 15, 2006

Date Due: November 29, 2006

GOALS:

- Develop a distributed search engine.
- Use Remote Procedure Call (RPC) best practices.

You are going to use socket programming and your new found multiprocessing skills to develop a distributed search engine.

CLIENT:

The CLIENT, does the following:

```
char **findText(char *text, char *directory)
```

Precondition:

- text is a C-String containing a text string of any size to search for
- directory is the absolute location directory on remote machines containing the files to search

Postcondition:

- Returns an array of C-Strings containing the line number and file name for each occurrence of the text string. Each C-String has the following format:

```
Found on line <lineno> in <filename>
```

where <lineno> is the line number the text string was found and <filename> is the relative filename where the text string was found.

NOTE: It's important that you adhere to the RPC best practices by implementing findText() as a STUB. That means that the findText() signature appears to the client programmer as a local function call. When you implement findText() you can add code to the stub which network enables the function.

The user interface for the client will look something like this:

```
<Welcome to Bob's Search Engine>

Enter search text> Romeo
Found on line xxx in RomeoAndJuliet.txt
Found on line xxx in RomeoAndJuliet.txt
...
Found on line xxx of RomeoAndJuliet.txt
```

SERVER:

The SERVER responds, in a *multithreaded* manner to findText() requests.

```
char **findText(char *text, char *directory, int serverNumber, int serverTotal)
```

Precondition:

- text is a C-String containing a text string of any size to search for
- directory is the absolute location directory containing the files to search
- serverNumber is the server number assigned to this server by the client
- serverTotal is the total number of servers the client is using

Postcondition:

- Returns an array of C-Strings containing the line number and file name for each occurrence of the text string. Each C-String has the following format:

```
Found on line <lineno> in <filename>
```

where <lineno> is the line number the text string was found and <filename> is the relative filename where the text string was found.

To support multiprocessing, the servers will divide up the file search work. Each server has exactly the same files in the directory specified by the directory parameter. To make this exercise more realistic, I want you to search a lot of big files. I'm providing a tar file which contains 500MB worth of public domain books that you should place in your search directory. This tar file is HUGE, and we have limited disk space on the linux machines, so I want you to do this:

- 1) Untar files into directory /tmp/search
- 2) Make sure that all of the book files are in the directory /tmp/search, no subdirectories
- 3) Type the following commands to allow all users to access this directory:

```
chmod a+rwx /tmp/search
chmod a+rwx /tmp/search/*
```

This procedure will allow other students to run a server on your machine without having to install the tar file.

The function ONLY searches a subset of the files in directory. The subset is determined creating an ordered set of the files in the directory, and using a combination of serverNumber and serverTotal to extract a subset.

For example, if there are three files in the directory: (file1.txt, file2.txt, file3.txt), and there is only ONE server then there will only be one call to findText() with serverNumber == 1 and serverTotal == 1. In this situation the single server will search all files in the directory. If there are TWO servers, then the client will call findText() twice. First with serverNumber == 1 and serverTotal == 2. The second will call findText() with serverNumber == 2 and serverTotal == 2.. The first call will search file1.txt, the second call will search file2.txt and file3.txt. If there are THREE servers, then each server will search one file.

TIMING:

Is a multiprocessing superior to a uniprocessing solution? You can answer this question by timing the performance of a client call to findText() when you have one, two, and three server machines. Use this code fragment to time the performance of any C++ function or method call:

```
#include <iostream>
#include <ctime>
#include <cstdlib>

using namespace std;

...
    int start = time(NULL);
    findText("Romeo", "/tmp/search");
    int stop = time(NULL);

    cout << "time taken is: " << stop - start << " seconds" << endl;
...

```

How will your multiprocessing solution scale as you add more servers? You can answer this question by estimating the performance of the client call to findText() with 5, 10, 25, 50, and 100 servers using the timing data you've collected.

DELIVERABLES:

Emailed to bdugan@stonehill.edu the following:

- All the files necessary to compile, link, and run your programs.
- An ELECTRONIC document (call this document README.TXT) describing:
 - o How to run your client and server programs
 - o Timing results and scaling estimates for 5, 10, 25, 50, and 100 servers
- These files should be placed in a directory called "<username>project6" .

- Use the tar command to place all the files in a single file called "`<username>project6.tar`". See the first project for instructions on how to do this.
- Email the `<username>project6.tar.gz` as an attachment to bdugan@stonehill.edu.