CS 3013 Operating Systems — WPI, A Term 2007
Craig E. Wills — Project 2 (30 pts)
Assigned: Friday, August 31, 2007 — Due: Monday, September 17, 2007

Introduction

This assignment is intended to help you learn about different architectures for building a server. In particular, you will build a program that can works using a single process, using multiple processes, or using multiple threads within a single process. You will also learn about file attributes.

Problem

The basic idea of this assignment is to handle "requests" to a server. Each request will be a file name. The server needs to determine the type and possibly the size for the given file. Requests will be read in from stdin (using *cin.getline()* in C++ or *gets()* or *fgets()* in C) with one file name given on each line of input. Your server should use one of three architectures to handle requests. The default architecture is a serial architecture, which is a single process with no threads.

Serial Architecture

In the serial architecture version of your program (the default), your program will read one file name from input, process it and then continue until all input file names are processed. Your program should continue to read file names until an EOF is detected on input. The serial version of your program will allow you to get the file statistic pieces working. The output of the all architectures should be identical to this architecture.

Your server will be using the Unix system call *stat()* to determine information about each file. This system call takes a file name and a statistic buffer of type `struct stat` (defined in `/usr/include/bits/stat.h`) and fills information about the file into the buffer. There is much information returned about a file, but the fields you will need to use are `st_mode` (the file mode) and `st_size` (the file size in bytes). More information about this system call can be found by looking at its man page.

The output of your program will be to print out the following information:

- total number of "bad files." These are file names causing *stat()* to return an error.
- total number of directories. You can test if a file is a directory by using the `S_ISDIR` macro on the `st_mode` field.
- total number of "regular" files. These are non-directory and non-special files. Most files are of this type. You can test for a regular file by using the `S_ISREG` macro on the `st_mode` field.

- total number of "special" files. There are three types—block, character and fifo. If a file is not a directory or a regular file then it is a special file. For example files in the directory `/dev/` are often special types of files as they refer to devices.
- total number of bytes used by regular files. Accumulate the sizes of all regular files.
- total number of regular files that contain all text (see more details below).
- total number of bytes used by text files. Accumulate the sizes of all regular text files.

Text Files

The *stat()* system call cannot be used to determine if a file contains text. Rather your program will need to read the contents of the file to determine if it is text. Because the contents of a file are unknown you cannot use input routines in C/C++ that only work for text input. Rather you need to use the *read()* system call to read in the *bytes* of a file and then determine if each byte is actually a printable character. The following piece of code shows how to open a file (stored in variable `filename`) with the *open()* system call and read its content into a buffer. The *open()* call returns a negative number on failure and *read()* returns the number of characters actually read. See the Section 2 of the man pages for more details on the use of these calls.

```
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>

    int cnt;  // count of chars read
    int fd;   // file descriptor
    char buf[128];  // buffer for reading

    if ((fd = open(filename, O_RDONLY)) >= 0) {
        while ((cnt = read(fd, buf, 128)) > 0) {
            // further processing
        }
        close(fd);
    }
```

If a file cannot be successfully opened then it should not be counted as a text file. As the file contents are processed, your program needs to check if each byte in the buffer is a printable character. To do so you should use two routines: *isprint()*, which determines if a byte value is a printable character; and *isspace()*, which determines if a byte is a space, newline, tab, etc. Check the man pages of these routines for details and the needed include file. You should classify a file as a text file if *all* bytes of the file are a printable or a space character. If any byte in the file fails both tests then the file is not a text file. Be sure and close the file when done processing as there is a limit on the number of open file descriptors a process may have.

Testing

You can test your code by manually entering file names or you can put a list of files into a text file and redirect it as stdin to your code. Another way to test a set of files is to combine your program with the *ls* command using a pipe. If your code is compiled as *proj2* then the following command line will check all files in the current directory.

```
% ls -1d * | proj2
```

The options to *ls* are "*1*" (the number one), which causes the files to be listed one per line and "*d*", which causes just directory names and not directory contents to be displayed.

Other directories can be specified with *ls*. For example, here is sample output obtained from running the code on the `/dev/` directory of a Computer Science Department machine.

```
% ls -1d /dev/* | proj2
Bad Files: 0
Directories: 3
Regular Files: 3
Special Files: 140
Regular File Bytes: 32881
Text Files: 2
Text File Bytes: 16541
```

Multi-Process Architecture

Once your server can handle each request in a serial manner, it should be modified to also support a multi-process architecture. You should use this architecture when the argument "`fork`" is specified on the command line. In this architecture you should create a worker child process (using the *fork()* system call) for each new file name. The worker process should use the *stat()* and *open()/read()* calls to obtain statistics about the file. It should update the appropriate type count and total byte variables as appropriate. Note that because multiple processes will now be accessing these variables and because variables are not shared amongst processes, you will need to allocate storage in shared memory (using the routine *shmcreate()*) for these variables.

You will also need to prevent concurrent access to these variables by using mutual exclusion for a critical region of code. You can use semaphores to implement mutual exclusion. Any semaphores for coordination between processes should be created using the routine *semcreate()*, as discussed in class. All semaphores and shared memory must be created *before* any worker processes are created.

Once a worker process has updated the appropriate variables for its file in shared memory the process terminates. This approach of creating a worker process for each new request allows these requests to be processed in parallel. However to avoid too much parallelism, the maximum number of worker processes that can be executing at one time is also specified on the command line. This limit is an integer between one and the maximum of 15.

Your parent process should work by forking a new process for each new file request until the limit is reached. At this point, your parent process should not create another process until it waits for one to complete. Once the number of worker processes is no longer at the limit your parent process can create another worker process. Your parent process should continue in this way until all files have been read and worker processes created. Once all file requests have been read, the parent process needs to wait for all remaining worker processes to complete.

When all processing is done your main process should print out results, which should be the same result as for the serial architecture. The final action of the parent process is to cleanup the shared memory and semaphores. You should use *shmdelete()* and *semdelete()* to cleanup resources.

The object file containing semaphore and shared memory primitives is `/cs/cs3013/public/lib/sem.o`. You will also need to include the header file /cs/cs3013/public/lib/sem.o`/cs/cs3013/public/lib/sem.h`. You can compile `proj2` and then run your program with the same input as before with at most 10 worker processes using the following:

```
% g++ -o proj2 proj2.C /cs/cs3013/public/lib/sem.o
% ls -1d * | proj2 fork 10
```

Multi-Threaded Architecture

Once your server can support the "`fork`" option, you can also add support for the "`thread`" option. With this option your program should use only one process to handle all requests, but each request should be handled by a worker *thread*. You will need to create a thread to handle each thread using the routine *pthread_create()*. Each thread should determine the stats for its file and update the appropriate type count and total byte variables as appropriate. You can store these variables as globals because all threads can access global variables, but because multiple threads will be accessing these variables you will need to prevent concurrent access by using thread routines to implement mutual exclusion for a critical region of code. Note: you should use primitives available with the pthread library for implementing mutual exclusion of threads, not the primitives used for mutual exclusion amongst processes.

Similar to the multi-process architecture you should impose a limit on the number of worker threads. When this limit is reached, your main thread must wait (using *pthread_join()*) for a worker thread to complete. However unlike waiting for a process, your main thread must wait for a thread with a specific identifier. Your program will need to remember the thread ids that it creates and wait for them in the order of creation. After your main thread (the one reading in file names) has read all files by detecting EOF on input, it should wait to make sure all threads have completed and then print out the output statistics.

Watch out! Threads are nice, but they can cause programming problems due to all threads running in the same address space. Specifically, be wary of passing the file name to a newly created thread. If you use the same static character array for passing file names to each thread then each time your main thread reads a new file name it will write over the previous file name. Even worse, all threads will be pointing to this same buffer and hence

the file name could change after the thread is created. Moral of the story: allocate a unique buffer for the file name passed to each thread.

The following shows how to compile your program using both the semaphore library for the multi-process architecture and the pthread library for the multi-threaded architecture. It also shows invoking your program with the same example using the multi-threaded architecture with at most 10 threads.

```
% g++ -o proj2 proj2.C /cs/cs3013/public/lib/sem.o -lpthread
% ls -1d * | proj2 thread 10
```

Additional Work

Satisfactory completion of a serial architecture version of the basic objective of this assignment is worth 12 of the 30 points. The multi-process and multi-threaded architectures are each worth an additional 8 points. For two additional points, you need to also add code to calculate the total wall-clock time, system and user time used by your program (includes both parent and child processes for multi-process architecture). You should test your code on different directories with all three architectures varying the thread and process limits. Plot separate graphs of program performance versus the number of maximum number of threads and processes. Also include results obtained for the serial architecture. Make sure your graph is correctly labeled. You should submit a *hard copy* plot of the results along with your explanation of the results to the instructor.

Resource Cleanup

For the multi-process architecture, there is a fixed limit on the number of semaphores and shared memory segments that can be allocated on the system. These resources DO NOT automatically get cleaned up when a process exits abnormally. It is expected that your program will cleanup semaphore and shared memory segments that your program has allocated before it exits. To determine if your program terminates without cleaning up resources use the command `ipcs -m -s`. This command will list all shared memory, and semaphore resources that have been allocated. To cleanup your resources use the command `ipcrm -m memid -s semid ...`. The following example illustrates use of these commands.

```
% ipcs -s -m

------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     status
0x0efd0000 327680     cew       660        1024       0

------ Semaphore Arrays --------
key        semid      owner      perms      nsems
0x0efd0000 65536      cew       660        15

% ipcrm -m 327680 -s 65536

% ipcs -s -m

------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     status

------ Semaphore Arrays --------
key        semid      owner      perms      nsems
```

Submission

Use the *turnin* command to submit your project with the project name of *proj2*.