DNA substring frequency. Pipes and forks

Introduction

The ability to create related processes through the *fork* system call is often used in programs that exploit multiple cooperating processes to solve a single problem. This is especially useful on a multiprocessor where different processes can truly be run in parallel. In the best case, if we have *N* processes running in parallel and each process works on a subset of the problem, then we can solve the problem in 1/*N* the time it takes to solve the problem using 1 processor.

If it were always that easy, we would all be writing and running parallel programs. It is rarely possible to divide up a problem into *N* independent subsets. Somehow the results from each subset need to be combined. There is also a tradeoff between the benefits of parallelism and the cost of starting up parallel processes and collecting results from different processes.

For this assignment, you will write a parallel program using Unix processes (i.e., fork). We hope to gain some benefit from using more than one process even on a uniprocessor since we will be reading data from files, so we might win by letting the scheduler overlap the computation of one process with the file I/O of another process. We would hope to see a performance improvement if the program is run on a multiprocessor.

If we genuinely wanted to write a fast parallel program on a shared-memory system, we would use a thread package rather than multiple Unix processes. Linux/Unix processes are costly to create, and the communication mechanisms between processes are also expensive. However, the ultimate goal of this assignment is to let you practice creating and using multiple processes, and it is also interesting to measure the performance of this kind of program.

Program specifications

Input

Input to the program is a set of several files, containing a small collection (300 MB) of short DNA reads of human DNA obtained by the 1000 genomes project. Each file contains DNA sequences of a single individual.

Output

You will write a C program called `freq5`, which will compute counts for each DNA sub-string of length 5. Counting substrings of length *k* (called *k*-mers) may be of interest to bioinformatics research, because the differences in count profiles may have a discriminative value in classifying human populations.

The valid DNA alphabet consists of 4 “letters” only: {‘A’, ‘C’, ‘G’, ‘T’}. Thus, the total number of different substrings of length 5 which can be generated from this alphabet is 1024. You will count the number of all substrings of length exactly 5 characters in all input files and you will output the histogram of counts to the standard output or to a file. The main requirement is that the computation will be performed using multiple processes, with each process being responsible for computing the substrings of an assigned subset of input files. You do not need to create a chart, just produce the list of counts for each substring in the following format:

aaaaa, 123123
aaaac, 22456544
aaaag, 1222244
… etc.

For every possible combination of 4 characters you output a total number of occurrences in all 15 input files. The substrings are sorted alphabetically.

Note that most of DNA “characters” in this input are upper-case, but occasionally you may encounter lower-case letters which have a special meaning. We will ignore this meaning, and will treat all letters in a case-insensitive manner. The output will be printed using lower-case letters. Each line in the input file is an independent read obtained from the sequencing machine, thus the count of substrings is performed for each line separately, there is no connection between two sequences on two separate lines, except that they came from the same individual.

If you encounter a non-DNA character – that is other than {‘A’,’C’,’G’,’T’}, you will need to skip the substring containing this character. You count the number of valid **uninterrupted** DNA substrings of length 5.

Program arguments and program flow

The program accepts three parameters: one required and two optional. The first parameter is the directory containing input files. Two optional parameters are the number of processes and the output file name.

```
freq5 -d <input file dir> -n <number of processes> -o <output file name>
```

You must use *getopt* to read in the command-line arguments. The command *man 3 getopt* will give you the correct man page which has a nice example that you can use as a template. You can also find an example in the code hints section at the end of this handout.
If parameter *N* = <number of processes> is not specified, your program performs all the counting by sequentially reading all files in the input folder, in its own single process. The valid values of *N*are positive integers. You must check that the value of *N* is valid. If the value is 1, then you still perform the entire computation in a single parent process as before.

If parameter *N* = <number of processes> is specified and is at least 2, then your program will create *N* child processes and divide the input into *N* chunks of (approximately) equal number of files. Each process will read content of the files assigned to it, and compute counts of all substrings.

If parameter *N* = <number of processes> is greater than the number of files to be processed, then you assign a separate process per each file, so the total number of child processes should not exceed the number of files. You do not need to issue any error message in this case, just set the actual number of processes to the minimum between *N* and the number of files.

You can use a simple integer array to record counts for each substring collected so far. Because we know the total number of all different substrings in advance, this array can be of a predefined size. The parent process will set up a pipe between the parent and each of its children. When a child has finished processing its assigned set of files, it will write its counts back to the parent in the predefined order of substrings. The parent will receive the data from each of the children by reading one record at a time from the pipes and updating a total count for the corresponding substring. The parent will write the final frequencies to the standard output or to the output file if the output parameter is specified.

You process all the files in the directory specified by the input parameter *-d*. Before you start running your code on a large repository of files, you probably want to create several (say, 3) small files with up to 10 characters each, to test your program for correctness. You can manually compute the number of substrings and compare to the output of your program. Note that you **do not** need to submit your test input files. This step is only for your own personal benefit.

Please make sure that your final output follows format *substring*,*count* specified above. No spaces or tabs, just a single pair per line. Substrings are sorted alphabetically.

If the optional parameter *-o* is specified, you output the substring counts into a specified output file. Otherwise you write the results to *stdout*. Because *stdout* is reserved for writing output, you must write all your error and debug messages to *stderr*.

The *freq5* program will also print to *stderr* the total time it took to run (see below for details).

Implementation details

First, you will collect all file names in the provided directory into an array of strings. According to the number of processes specified by the optional parameter *-n*, you will assign to each child process a sub-array of file names. It would make a lot of sense to write a function to perform counting in each sub-array of files. Each child will open the assigned files in turn, read each line, count substrings, and update the total counts. After processing all files, each child process will send each count through the pipe connecting the child with the parent.

The parent process will need to implement a function that reads from each of the child pipes for each value of the substring in turn, and updates the corresponding counts. When all counts for a particular substring have been collected, the parent process writes a single total count to the specified output in the same sorted order. An illustration of program with 4 child processes is presented in Figure 2.

Modularize your code into separate source files, and make use of header files as you find fit. The decisions about how do you structure your program are up to you.

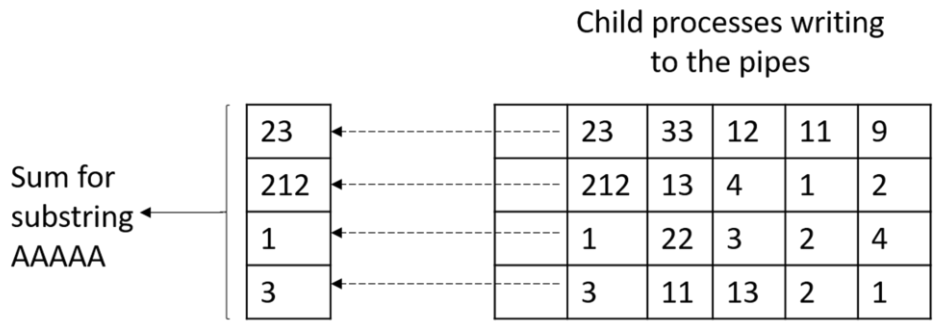

Figure 1. The parent reads the count for current substring from each child process, sums up the counts and writes to the output. Then it reads the next set of counts from the pipes. When all the counts are processed, parent has to close the pipes.

Good system programming style

This is your third C-programming assignment, and you are expected to write a clean, properly formatted and well-documented code. You must perform proper error checking for every section of your program that may cause an error. For example, for *fork()* or *pipe()* calls you have to check the return code for errors, display meaningful messages, and take appropriate actions.

You must practice good system programming skills. Your program should not crash under any circumstance. This means that the return value from all system calls must be checked, files and pipes must be closed when not needed, and all dynamically allocated memory must be freed before your program terminates. You should also be careful to clean up any processes left running. You can get a list of all the processes you have on a machine by running

```
ps aux | grep <user name>
```

The parent process must ensure that all the children have terminated properly and the parent will print out an error message if any of the children have terminated prematurely.

Experimenting with number of processes

The real question is how many processes should we use to get the best performance out of our program? To answer this question, you will need to find out how long your program takes to run. Use `gettimeofday` to measure the time from the beginning of the program until the end, and record this time. Now perform a series of experiments by running your program with different numbers of processes.

Think about what the performance results mean. Would you expect the program to run faster with more than one process? Why or why not? Why does the speedup fade out? Did the performance

results surprise you? If so, how? Write a short report (1-2 paragraphs) with your performance results and explanations and submit it in file *report.pdf*.

Useful C hints

Reading program parameters using *getopt*

```
/*
 * Sample program for parsing command-line parameters using getopt
 */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main(int argc, char *argv[]) {
    char *optarg;
    int ch;
    FILE *infp, *outfp;

    char *infile = NULL, *outfile = NULL;
/* read in arguments */
while ((ch = getopt(argc, argv, "f:o:")) != -1) {
    switch(ch) {
        case 'f':
            infile = optarg;
            break;
        case 'o':
            outfile = optarg;
            break;
        default :
                        fprintf(stderr, "Usage:  test -f <input file
                        name> " "-o <output file name>\n");

          exit(1);
    }
}
...
return 0;
}
```

Getting names of all files within a given directory

```
/*
* This program displays the names of all files
* in the current directory.
* Works only for POSIX-compliant systems (Unix)
*/
#include <dirent.h>
#include <stdio.h>
```

```
    int main(void)
    { DIR *d;
    struct dirent *dir;
d = opendir (".");
if (d) {
    while ((dir = readdir(d)) != NULL) {
        printf ("%s\n", dir->d_name);

    }
    closedir (d);
}

return(0);
}
```

Timing Your Program

Using *gettimeofday*: You should read the man page for *gettimeofday*, but here is an example of how to use it, and how to compute the time between two readings of *gettimeofday*.

```
struct timeval starttime, endtime;
double timediff;

if ((gettimeofday(&starttime, NULL)) == -1) {
    perror("gettimeofday");
    exit(1);

}

// code you want to time

if ((gettimeofday(&endtime, NULL)) == -1) {
    perror("gettimeofday");
    exit(1);

}

    timediff = (endtime.tv_sec - starttime.tv_sec) +
    (endtime.tv_usec - starttime.tv_usec) / 1000000.0;
fprintf(stderr, "%.4f\n", timediff);
```