Assignment 3: Processes and Pipes

Due Mar 18 by 4pm **Points** 10

Introduction

The fork system call is often used to create 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 one 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. Usually, the results of each subset need to be combined in some way. There is also a difficult tradeoff to make 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 sorting program using Unix processes (i.e., `fork`). We can hope to gain some benefit from using more than one process even on a uniprocessor since we will be reading data from a file, 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 sort 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, a large part of the purpose of the assignment is to give you practice creating and using multiple processes, and it is also interesting to measure the performance of this kind of program.

Specifications

The data to be sorted is a list of records where a record contains a word and its frequency measure (see `struct rec` in `helper.h`). The records are written to the file in sequence in binary format. This means that you can use `fread` to read one record at a time or multiple records in one `fread` call.

You will write a C program called `psort` that takes 3 arguments:

- the number of processes to create,
- the name of the input file to sort, and
- the name of the file to write the output to.

It will be called as shown below. 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.

```
psort -n <number of processes> -f <inputfile> -o <outputfile>
```

If the correct number of command-line arguments and the correct options (`-n`, `-f`, and `-o`) are provided, you may assume that the values corresponding with the options are valid. If the incorrect number of

command-line arguments or incorrect options are provided, you must report that using the following message and exit the program with an exit code of `1` (copy and paste this line of code to your program):

```
fprintf(stderr, "Usage: psort -n <number of processes> -f <inputfile> -o <outputfile>\n");
```

If `<number of processes>` is `N`, then your program will create `N` processes and divide up the file to be sorted into `N` chunks. Each process will read `1/N` of the file and sort its chunk of the file according to frequency (from smallest to largest) in memory using the `qsort` library function. You should divide up the words as evenly as possible. So if there are 3 processes and 5 records, the first two processes get 2 records each and the third gets 1 record.

The parent process will also set up a pipe between the parent and each of its children. When a child has finished sorting its chunk, it will write each record in sorted order to the pipe. The parent will merge the data from each of the children by reading one record at a time from the pipes. The parent will write the final sorted list to the output file. The output file will be in the same binary format as the input.

The `psort` program may print to `stdout` the time it took to run (see below for details). If it runs correctly, this will be the only output to `stdout.`

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

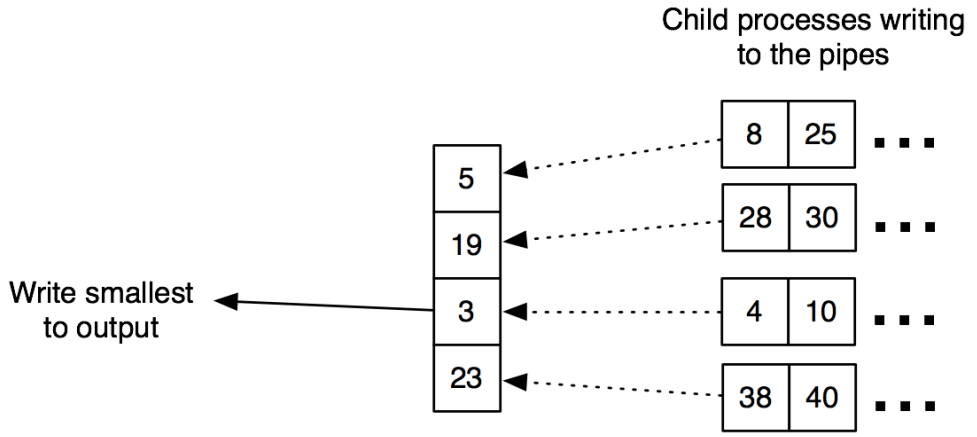
Details

The first thing you should do is to write a small program that reads a binary file in the given format and prints its contents to standard output. This will not only help you test your program, but your `psort` program will also need to be able to read the binary file, so it gives you the opportunity to make sure you can do that part correctly. We will not be grading your program that displays a binary file.

The `qsort` function sorts an array in place. The final argument to `qsort` is a pointer to a comparison function. The comparison function will be called by `qsort` with arguments that are elements of the array to be sorted. The comparison function is given in `helper.c`. Here is **a tutorial on using `qsort`** **(http://www.anyexample.com/programming/c/qsort__sorting_array_of_strings__integers_and_structs.xml)**. The man page for `qsort` also has an example.

Since you can compute that each child will sort a portion of the file from smallest to largest, it would make a lot of sense to write a function to perform this task. Each child will open the file, use `fseek` to get to the correct location in the file, and begin reading at that point. The child can read the data in one system call, use `qsort` to sort the data, and then will write one record at a time to the pipe connecting the child to the parent.

The parent process will need to implement a `merge` function that compares the current smallest value read from each child pipe, and writes the record with the smallest frequency to the output file each time. The following diagram shows how merge might work with 4 child processes.



First, the parent reads one value from each of the child pipes. Since the smallest value is 3, the parent will write 3 to the output file, and will then read from that child's pipe, loading 4 into the array. Now, 4 is the smallest value, so the parent will write 4 to the output file, and read from that child's pipe, loading 10 into the array. Now, 5 is the smallest value, so the parent will write 5 to the output file, and read from that child's pipe, loading 8 into the array. And so on...

The parent must use `wait` (rather than `waitpid`). If a child does not terminate normally for any reason, the following error message should be printed, but this should not cause the program to exit. (Copy and past the line below into your program.)

```
fprintf(stderr, "Child terminated abnormally\n");
```

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.

When working on the assignment, you should also be careful to clean up any processes left running. You can get a list of all of the processes you have on a machine by running:

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

Do not log out without checking for and cleaning up any processes left running.

The starter code in your repository contains a sample input file: `8-words.b`, and the `8-words.txt` file used to generate it. We have also given you a program called `mkwords.c` that can be used to generate more input files. Read the comments in the code to see how to run it. Note that you will need to compile it as below to link in the math library:

```
      gcc -Wall -g -std=gnu99 -o mkwords mkwords.c -lm
```

You will want to ensure that your program works with a relatively small data set before you begin experimenting with the large ones. You might want to use parts of the `dictionary.txt` file from a2 as source of words. Remember to create the input files on teach.cs machines so that you don't need to worry about line endings and other differences between files on different systems.

The starter code also includes `helper.h` and `helper.c` which define the `struct rec` and include two functions that you will find useful.

Makefile

Create and submit a Makefile that compiles the program to produced the executable named `psort`. To build the executable, it may depend on `helper.h`, `helper.c`, and `psort.c`. You may edit and add to those files, but do not add any additional files or change `struct rec`.

Marking

We will use testing scripts to evaluate correctness. As a result, it's very important that your output matches the expected output precisely. Also, it is not sufficient to produce the correct output -- following the algorithm specified is required.

For this and future assignments, your code may also be evaluated on:

- **Code style and design**: At this point in your programming career you should be able to make good choices about code structure, use of helper functions, variable names, comments, formatting, etc.
- **Memory management**: your programs should not exhibit any memory leaks. This means freeing any memory allocated with malloc and realloc. This is a very important marking criterion. Use valgrind to check your code.
- **Error-checking**: library and system call functions (even malloc!) can fail. Be sure to check the return values of such functions, and terminate your program if anything bad happens.
- **Warnings**: your programs should not cause any warnings to be generated by gcc -Wall.

Reminder

Your program must compile on `teach.cs` using `gcc` with the `-Wall` option and should not produce any error messages. Programs that do not compile, will get 0. You can still get part marks by submitting something that doesn't completely work but does some of the job -- but it **must at least compile** to get any marks at all. Also check that your output messages are **exactly** as specified in this handout.

Submission

We will be looking for the following files in the a3 directory of your repository:

- `helper.c`
- `helper.h`
- `psort.c`
- `Makefile`

Do not commit `.o` files or executables to your repository.

Timing Your Program (Optional)

This part is optional and not for any marks, so complete it only to satisfy your curiosity!

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`

```
#include <sys/time.h>

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(stdout, "%.4f\n", timediff);
```

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 print this time to standard output. Now try running your program on different numbers of processes on different sized datasets.

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 eventually decrease below 1? How does the speedup differ between data sets? Why? Did the performance results surprise you? If so, how?