Parallel Keyword Counters

DUE: Sunday, December 1, 2013, by 10pm. No late submission accepted.

IMPORTANT: Please read the entire handout before you start coding.

Introduction

A keyword counter scans a given text and counts how many times a keyword appears in the text. In this project, you will write two keyword counters using multiprocessing and various ways of inter-process communications (IPC) for efficient counting.

Project at a Glance

This project comprises of two parts. In both cases several processes will be created to search for the keyword concurrently.

- *Part One:* counts the occurrences of only one keyword provided in the command line and uses pipes to allow each child process to inform its parent about its count.
- *Part Two:* counts the occurrences of multiple keywords provided as a list in a file and uses shared memory for inter-process communication.

Defining a Keyword

A word will be counted as a keyword if and only if it meets all of the following rules:

1. Keywords are case-sensitive. E.g., "`hello`" matches only to "`hello`" and not to "`Hello`" or "`hELLo`", etc.
2. The match is a complete string match. E.g., "`hello`" is matched to only "`hello`" and not to "`Hihello`" or "`hello,`", etc.
3. Keywords are separated by delimiters: space, tab, and newLine. A keyword always follows a delimiter unless it is at the beginning of the file, and is always followed by a delimiter unless it is at the end of the file.

In addition, the keyword in Part One is always alphanumeric (i.e., no special character), while keywords for Part Two are not (see Keyword List section of Part Two).

Part One: pipe_counter.c

The first part involves writing a program that counts the number of occurrences of a keyword in a text file and prints the number in the output file. To be more efficient, your program will use parallel processing – using the `fork()` function to create a number of processes, each of which scans part of the input text, and the `pipe()` function to build pipes to propagate the count of the keyword from the child process to its parent process.

Create an empty file named `pipe_counter.c` and write all your code in it. You can include only the following headers: `stdio.h`, `stdlib.h`, `string.h`, `sys/types.h`, and `unistd.h`.

Command-Line Arguments

Your program will run given four flags:

```
./pipe_counter -i <inFile> -o <outFile> -k <keyword> -b <bufferSize>
```

in which `<inFile>` and `<outFile>` are the names of input and output files, respectively, `<keyword>` represents the keyword of which to count the occurrence, and `<bufferSize>` specifies the size, in bytes, of the buffer each process works on; its meaning will be explained in detail in the Multiprocessing Mechanism section. The order of the flags is random.

For instance, the following command

```
./pipe_counter -k project –b 256 -i in.txt -o result.txt
```

means counting the occurrences of the word "`project`" in the file `in.txt` and writing the result in a file named `result.txt`. Each process will handle a 256-byte chunk of `in.txt`.

Input and Output

The input can be any ASCII text file such as a HTML document or a plain text file. After running the program, the output file specified in the command line should contain the number of the keyword's occurrences, followed by a newline.

For example, if the example command above is run with the file `in.txt` whose content is:

```
This is a forking projecT of cs240. This project is about fork() and
pipe(). We have 2 weeks to finish this project.
```

Then the output file `result.txt` should contain (note that the number is followed by a "`\n`"):

```
1
```

because the keyword "`project`", already marked by red color, appears only once.

Multiprocessing Mechanism

To make testing more consistent, here are some rules that you need to follow carefully:

The argument `bufferSize` determines how work is distributed among processes. That is,

- The total number of times to call `fork()` = input file size *div* `bufferSize`
- The number of bytes the `main` process works on = input file size *mod* `bufferSize`

For example, if the input file is of size 1000 bytes, and `bufferSize` equals 256, then a total of *three* child processes should be forked. While each of them gets a 256-byte chunk of the input to do the searching, the `main` process searches the remaining 232-byte chunk for the keyword. However, please note that according to the equations above, when `bufferSize` can divide file size without remainder, or when `bufferSize` is greater than file size, the `main` process should not do any searching.

Besides, your program must call the function `pipe()` to create pipes through which a child process reports its count to the parent process when its search is done.

There are many possible parent-children hierarchies of processes, the simplest one of which is to let the `main` process fork all the child processes. In this hierarchy, one pair of pipes is sufficient. In addition, please note that while you can create as many pipes as necessary (but at least one pair), the function `fork()` must be called the exact times defined as above, whatever hierarchy your program adopts. In the end, it is the `main` process that collects and adds all the counts to make the final output.

With pipes, you need NOT and should NOT use any wait functions like `wait()` and `waitpid()`.

Compiling & Testing

As usual, compile your code and do unit test and system test before submission.

Part Two: `shm_counter.c`

In Part Two, the counter searches for a number of keywords, which requires a more complex way of IPC – every child process needs to write its results to the shared memory created by the `main` process. You will use semaphores to control the access to the shared memory and the progress of `main` process if needed. In the end, the `main` process reads those results, adds the numbers of occurrences for each keyword, and writes them to a specified output file.

Three files are given to help you with this part (no other files needed):

- `interface.h`: defines all the necessary global variables and data structures to ensure your program's compatibility with the Autograder. No need to modify or submit it.
- `shm_counter.h`: contains the common definitions and declarations. You can add your own definitions there.
- `shm_counter.c`: the source file you need to complete.

Command-Line Arguments

Your program will run given the following command-line arguments:

```
./shm_counter -i <inFile> -o <outFile> -k <keywordsFile>
```

where `<inFile>`, `<outFile>`, and `<keywordsFile>` represent the names of the input file, the output file, and the keyword list file, respectively. Still, the order of those flags is random and your program should handle any possible order, such as the example below:

```
./shm_counter -i in.txt -k kwList.txt -o out.txt
```

which requires the program to read a list of keywords from the file `kwList.txt`, scan within the file `in.txt` to count the occurrences of those keywords, and print the output, in the required format, to the file `out.txt`.

Input and Output

Keyword List

In the keyword list file, each valid line contains a keyword. Invalid lines, i.e., empty lines, or lines that contain any tabs or spaces, should be ignored. Besides, in Part Two, a keyword can contain punctuation marks comma "`,`", period "`.`", exclamation "`!`", and semicolon "`;`". Therefore, "`hello,`" and "`O.K.!`" are valid keywords, while "`hello, world`" is not.

For instance, here is a keyword list file named `kwList.txt` (line numbers are shown only for demonstration's sake; they are never part of the content of related files):

```
1 project.
2 fork
3 is about
4 This
5
```

In the above file, Line 3 is an invalid line because it contains a space, and Line 5 is also invalid because it is empty. Those two lines should be skipped by your program when processing the list.

Hint: create a keyword list from an empty file in the virtual machine `CS240_VM` to avoid potential problems caused by different newLine representations in different operating systems.

Each valid keyword is represented in the shared memory by a `key_id`, starting from `0`. The `key_id`s for the keywords in `kwList.txt` are shown below:

Ln	key_id	Content of Keyword List File
1	0	project.
2	1	fork
3	(INVALID)	is about
4	2	This
5	(INVALID)	

Input

The specifications for the input file remain the same as in Part One. Here is the `in.txt` used in Part One:

```
This is a forking projecT of cs240. This project is about fork() and
pipe(). We have 2 weeks to finish this project.
```

Output

In the output file, each line contains two items – keyword and count, separated by a colon followed by a space. If a keyword does not appear in the input text at all, report its number of occurrences as `0`, as indicated by the following example.

Given the `in.txt` and `kwList.txt` above, the final output is supposed to be (w/o line numbers):

```
1 project.: 1
2 fork: 0
3 This: 2
```

Note that the order of those keywords should match the order in the keyword list file.

More Instructions on Multiprocessing and Shared Memory

Wrapper Functions

For auto grading's purpose, in this part you are NOT allowed to call the system functions `fork()`, `semget()`, and `shmget()` directly; instead, you should use their corresponding wrapper functions, defined in `interface.h`: `FORK()`, `SEMGET()`, and `SHMGET()`. To do this, simply write the function name in uppercase. For example, type "`FORK`" whenever you need to type "`fork`", and so on, as everything other than the name remains unchanged.

By using those wrapper functions, the words "`fork`", "`semget`", and "`shmget`" with the exact spellings should not appear in your source code.

Multiple Processes

The requirements for multiprocessing are as follow:

- The `main` process initializes all needed variables, parses input and keyword list, creates and initializes shared memory and the semaphore(s), forks ALL child processes and waits for them to finish searching; the `main` process itself, however, does not do any searching. When all child processes are done, the `main` process reads the results from shared memory, and generates the output. (You can use more than one semaphore if you need.)
- Number of child processes to create: defined by a macro called `MAX_PROCS`; does not depend on the sizes of the input file or the buffer.
- Buffer size for each child process: defined by a macro called `MAX_BUFFER_SIZE`. It is the size of a chunk, and each child process searches only in its corresponding chunk. i.e., the first child process will search only in the first chunk, the second child process in second chunk and so on.
- Still, do not use any wait function (`wait()`, `waitpid()`, etc.) in your code. Use semaphores to control the progress of the `main` process if needed.

The macros `MAX_PROCS` and `MAX_BUFFER_SIZE` are defined in the header `interface.h`. To change their values for testing, you may either modify the header, or use the macro definition flag `-Dname=value` during compilation. Note that your program will not be tested against unreasonable configurations like {`MAX_BUFFER_SIZE=2` while some keyword is longer than 2 bytes} or {input file size=100 bytes, `MAX_PROCS=4`, and `MAX_BUFFER_SIZE=20`}.

More about Shared Memory

In UNIX, shared memory is memory space that can be simultaneously accessed by multiple processes. Such simultaneous accessibility requires mutual exclusion synchronization. One method to do so is to use semaphores. If you use POSIX semaphores in this part, be sure to

- include the header `semaphore.h` to enable POSIX semaphore system calls;
- use options `-lrt` and `-pthread` to compile your code with `gcc`.

Instructions about using shared memory:

- The `main` process creates and initializes a shared memory space using `SHMGET()` and other related functions/operations.

- The global variable `shared_mem` should point to the beginning address of the shared memory space that stores the results.
- Every child process accesses the shared memory and writes its searching results. The values written to the shared memory should be of type `struct proc_key_count`, as defined in `interface.h`.
- After all child processes complete, the `main` process needs to clear the semaphore(s) and detach the shared memory.
- You need to complete the function `destroy_sharedmem()` to release the shared memory you create, and free all the dynamic memory you allocate, if any.

The semaphore may be stored in a shared memory space to be effective for all processes, depending on your implementation. To know more about semaphores and shared memory, visit the Linux man website.

A Corner-Case Analysis

To clarify the instructions stated previously, here is an example involving a corner case. Notice that we deliberately make the value of `MAX_BUFFER_SIZE` smaller than the length of a keyword to demonstrate how this situation should be handled if it really happens.

Suppose there is a 14-byte input file whose content is

```
abc defgh ijkl
```

and let `MAX_BUFFER_SIZE=4`, `MAX_PROCS=5`, and keywords be "`defgh`" and "`ijkl`".

Under this configuration, the input text is separated into four chunks, as shown below (the symbol "`|`" denotes the border between chunks):

```
abc |defg|h ij|kl
```

As you can see, the keyword "`defgh`" is split between chunks since it is longer than a buffer. The keyword "`ijkl`" is also split because of its particular position.

Doing the Search

To deal with this situation, the following rule is applied (you can also apply this to Part One):

If the buffer splits a word and the substring concatenated with whatever sequence of characters, up to a delimiter, in the following buffer will form a keyword, then the occurrence of that keyword should be tagged by the process owning the buffer in which the first character is positioned.

With this rule, what each process does can be clearly described. After taking the input and keyword list, the `main` process forks *five* child processes as required by the value of `MAX_PROCS`. Each one of the first four child processes gets a chunk and searches for keywords within it, while the fifth child has no work to do at all.

Number the five child processes from 0 and 4, then here is what they would do:

- *Child-0* and *Child-3*: no keyword in their corresponding chunks. So, each of them writes two 0-occurrence entries to shared memory and then exits.
- *Child-1*: it has a substring of one of the keyword, "`defg`". When it is concatenated with "`h`" from following buffer it forms a keyword. Therefore, it should report one hit of the keyword "`defgh`".
- *Child-2*: same as child-1 but it will find the keyword "`ijkl`".
- *Child-4*: since there is nothing to read for child-4, all it should do is to update its entries in the `proc_key_count` table saying that it has zero occurrences for each keyword. Then it exits.

Details in Shared Memory

In `struct proc_key_count`, a variable named `key_id` of type `int` is used to denote each keyword. How the `key_id` for a keyword is determined is described in the Keyword List section. For the example we are using, suppose the keyword list file has the following content:

key_id	Ln	Content of Keyword List File
0	1	defgh
1	2	ijkl
	3	

In the shared memory, the number of records equals `MAX_PROCS` times the number of keywords. Let `childPidX` denote the `PID` of the child process whose index is `X`, and the form `{pid, key_id, freq}` symbolize `struct proc_key_count`, then the shared memory to record results should store data like the following when all child processes complete their tasks:

```
{childPid0, 0, 0}, {childPid0, 1, 0},
{childPid1, 0, 1}, {childPid1, 1, 0},
{childPid2, 0, 0}, {childPid2, 1, 1},
{childPid3, 0, 0}, {childPid3, 1, 0},
 {childPid4, 0, 0}, {childPid4, 1, 0}
```

When the shared memory is all set, the `main` process adds those hits for each keyword, and generates the output.

Compiling & Testing

A sample command for compilation:

```
gcc -lpthread -pthread -lrt -DMAX_BUFFER_SIZE=64 -o shm_counter shm_counter.c
```

Notes:

- The tag "`-DMAX_BUFFER_SIZE=64`" defines the macro `MAX_BUFFER_SIZE` to be `64` at compilation time. This is added for illustration's purpose.
- Add the flag `-m32` if you compile on a 64-bit machines to avoid behavior inconsistency.
- The library `libpthread` may be needed for compilation, depending on the OS.

Grading Criteria

Coding Standard (15%)

Pipe Counter (30%)

- The source, `pipe_counter.c`, compiles and runs without error
- The program always correctly reads data from the given input file
- Program uses `fork()` and `pipe()` as required, and distributes work among processes
- The program writes the correct result to the output file

Shared Memory Counter (55%)

- The source files, namely, `shm_counter.c` and `shm_counter.h`, compile correctly
- Program can correctly load input text and parse keyword list from specified files, and write the correct result to the specified output file
- Program implements multiprocessing and shared memory as instructed; each process works on the correct chunk and writes correct data to the shared memory pointed to by `shared_mem`.
- Your program MUST use at least one semaphore to control the access to shared memory or the progress of `main` process or both, whichever is needed by your implementation.
- The main process creates and clears the shared memory and semaphores properly
- In Part Two, piping is allowed ONLY to control the progress of `main` process, but any use of piping will incur a penalty of 10% deduction from your TOTAL grade of this project. Further, if neither piping nor semaphore is used by your `shm_counter`, an additional 10% of your total score will be deducted.

Still, your code will be inspected by the instructor and TAs to determine your final grade.

Turning in

To submit for Part One, use `tar` to compress your `pipe_counter.c` to `pipe_counter.tar.gz` and submit the latter to "Project 4 Pipe Counter" page of the Autograder.

To submit for Part Two, pack the two source code files as directed, and submit the `shm_counter.tar.gz` to "Project 4 Shared Memory Counter" page:

```
tar zcf shm_counter.tar.gz shm_counter.c shm_counter.h
```