CSci 4061: Introduction to Operating Systems
Fall 2020
Project #2: IPC-based Map Reduce

Instructor: Jon Weissman

Due: 5 pm, November 4, 2020

1 Purpose

In project 1, we built a simple version of mapreduce using operating system primitives such as `fork`, `exec` and `wait`. While doing so, several utility functions were provided which helped you implement the map and reduce tasks. In this project, you will be required to implement these utility functions using inter process communiation (IPC) based system calls such as `msgget`, `msgsend`, `msgrecv`, `msgctl` etc. You should work in groups[1] of 3 as in Project 1. `Please adhere to the output formats provided in each section.`

2 Problem Statement

In this project, we will revisit the single machine map-reduce designed for the word count application[2] in Project 1. There are four phases: `Master, Map, Shuffle` and `Reduce`. In `Master` phase, the input text file is taken as input from the command line. The master will split the input file in chunks of size 1024 bytes and distribute it uniformly with all the mapper processes. In the `Map` phase, each mapper will tokenize the text chunk received from the master and writes the `<word 1 1 1...>` information to `word.txt` files. Once the mappers complete, the master will call the `Shuffle` phase to partition the word.txt files for the reducers. The files are partitioned across different reducers based on a `hash function`. Partitioning essentially allocates specific non-overlapping key ranges (i.e. words in our case) to specific reducers to share the load. Once the partitioning is complete, the `word.txt` file paths are shared with the `Reduce` phase. Then the main program will spawn the reducer processes to carry out the final word count in the `Reduce` phase.

Objective: You will have to design and implement portions of `Master`, `Map`, `Shuffle` and `Reduce` phases. A code template will be provided to you. You are also free to use portions of your implemented code from Project 1. You can also just start from your Project 1 solution.

3 Functions to implement

In this section, we will discuss the details of the functions which you are supposed to implement. Please refer to Project 1 for detailed description of each of the four phases in MapReduce. We use END and ACK (acknowledge) messages to mark the end of any phase so that the involved processes can move on to their next phases.

[1]Group formation information has been shared separately on canvas.

[2]Refer to Project 1 description to refresh your understanding of word count application.

3.1 sendChunkData

The `Master` phase uses the `sendChunkData` function to distribute chunks of the input file to the mappers in a round robin fashion. Refer to Algorithm 1 for details.

File: src/utils.c

Algorithm 1: `sendChunkData()`

```
Input: ($String\ inputFile, Integer\ nMappers$), inputFile: text file to be sent, nMappers: number
       of mappers
// open message queue
$messageQueue \leftarrow openMessageQueue()$;
// Construct chunks of at most 1024 bytes each and send each chunk to a mapper in a round
  robin fashion.
while inputFile has remaining text do
    $chunk \leftarrow getNextChunk(inputFile)$;
    $messageSend(messageQueue, chunk, mapperID)$;
end
//send END message to mappers
for each mapperId do
    $messageSend(messageQueue, EndMessage, mapperId)$;
end
// wait for ACK from the mappers for END notification
for each mapper do
    $wait(messageQueue)$;
end
// close the message queue
$close(messageQueue)$;
```

Notice: The code for `bookeepingCode()`, `spawnMappers()`, `spawnReducers()`, `waitForAll()` used by the `master` are already provided in code template.

Tip: While constructing the 1024 bytes chunk, if the 1024th byte is somwhere in middle of a word, constructing the 1024 byte chunk will result in that word being split across multiple chunks. Therefore, just construct the chunk upto the previous word so that no word gets split.

To-do: You are supposed to implement the `sendChunkData()` function.

3.2 getChunkData

Each `mapper` in the `Map` phase calls the `getChunkData` function to receive the text chunks from the `master` process. Refer to Algorithm 2 for details.

File: src/utils.c

Algorithm 2: `getChunkData()`

```
Input: $(Integer\ mapperID)$, mapperID: mapper's id assigned by master ∈ {1, 2, ..., nMappers}
Result: $chunkdata$, chunk data received from master

// open message queue
$messageQueue \leftarrow openMessageQueue()$;

// receive chunk from the master
$chunkData \leftarrow messageReceive(messageQueue, mapperID)$;

// check for END message and send ACK to master
if chunkData == EndMessage then
    $messageSend(messageQueue, ACK, master)$;
end
```

Notice: The code for `createMapDir()`, `map()`, `writeIntermediateDS()` functions used by the `mapper` are already provided in code template.

To-do: You are supposed to implement the `getChunkData()` function.

3.3 shuffle

Once all the mapper processes complete and terminate, the master process will call the `shuffle()`. The shuffle function will divide the `word.txt` files in `output/MapOut/Map_mapperID` folders across nReducers and send the file paths to each reducer based on a hash function.

The flow of control in shuffle is given in algorithm 3.

File: src/utils.c

Algorithm 3: `shuffle()`

```
Input: $(Integer\ nMappers, Integer\ nReducers)$, nMappers: #mappers, nReducers: #reducers

// open message queue
$messageQueue \leftarrow openMessageQueue()$;

// traverse the directory of each Mapper and send the word filepath to the reducers
for each mapper do
    for each wordFileName in mapOutDir do
        // select the reducer using a hash function
        $reducerId = hashFunction(wordFileName, nReducers)*$;
        // send word filepath to reducer
         $messageSend(messageQueue, wordFilePath, reducerId)$;
    end
end
//send END message to reducers
for each reducerId do
    $messageSend(messageQueue, EndMessage, reducerId)$;
end
// wait for ACK from the reducers for END notification
for each reducer do
    $wait(messageQueue)$;
end
// close the message queue
$close(messageQueue)$;
```

Notice: The code for `hashFunction()` function is already provided in code template.

To-do: You are supposed to implement the rest of the `shuffle()` function.

3.4 getInterData

Each reducer uses the `getInterData` function to retrieve the file path for words for which it has to perform the `reduce` operation and compute the total count. Refer to Algorithm 4 for details.

File: src/utils.c

Algorithm 4: `getInterData()`

```
Input: ($String\ wordFileName, Integer\ reducerID$), wordFileName: placeholder for storing
       the word file path received from master, reducerID: reducer's id assigned by master $\in$ {1,
       2, ..., nReducers}
Result: $wordFileName$ has the word file path received from master

// open message queue
$messageQueue \leftarrow openMessageQueue()$;

// receive data from the master
$wordFileName \leftarrow messageReceive(messageQueue, reducerID)$;

// check for END message and send ACK to master
if chunkData == EndMessage then
    $messageSend(messageQueue, ACK, master)$;
end
```

Notice: * The code for `reduce()`, `writeFinalDS()` functions used by `reducer` are already provided in code template.

To-do: You are supposed to implement the `getInterData` function.

Note:

- The `master` process sends an END message to each `mapper` to inform it of the completion of transfer of chunks (in `sendChunkData()` function). Each `mapper`, in turn, sends an ACK message to the `master` for acknowledging the receipt of all chunks (in `getChunkData() function`). Once the `master` and `mapper` processes have exchanged END and ACK messages, they move on to next phase.
- The `master` process also sends an END message to each `reducer` to inform it of the completion of sending of intermediate word file paths (in `shuffle()` function). Each `reducer`, in turn, sends an ACK message to the `master` for acknowledging the receipt of all file paths (in `getInterData() function`). Once the `master` and `reducer` processes have exchanged END and ACK messages, they move on to next phase.

4 Compile and Execute

Please refer to Project 1.

5 Expected Output

Please refer to Project 1.

6 Testing

Please refer to Project 1.

7 Assumptions / Points to Note

The following points should be kept in mind when you design and code:

- The input file sizes can vary, there is no limit.
- Number of mappers will be greater than or equal to number of reducers, other cases should error out.
- We recommend using message queues related system calls for this project i.e. msgsend, msgrecv, msgget, msgctl etc. You are free to use the pipe, read, write system calls if you want to use pipes instead of message queues for inter-process communication but we strongly recommend to use the message queues. TAs will be able to better help you out with message queues instead of pipes.
- Add error handling checks for all the system calls you use.
- Do not use the system call "system” to execute any command line executables.
- You can assume the maximum size of a file path to be 50 bytes.
- Follow the expected output information provided in the previous section.
- The chunk size will be atmost 1024 bytes as there is a chance that some of the 1024th byte in inputFile is the middle of a word.
- If you are using dynamic memory allocation in your code, ensure to free the memory after usage.

8 Deliverables

One student from each group should upload to Canvas , a zip file containing the source code, Makefile and a README that includes the following details:

- The purpose of your program
- How to compile the program
- What exactly your program does
- Any assumptions outside this document
- , Project group name, Team member names, x500
- Contribution by each member of the team

The README file does not have to be long, but must properly describe the above points. The code should be well commented, it doesn’t mean each and every line. When a TA looks at your code he/she/they should be able to understand the jist. You might want to focus on the “why” part, rather than the “how”, when you add comments. At the top of the README file, please include the following:

```
README.md

test machine: CSELAB_machine_name
date: mm/dd/yy
name: full_name_1, [full_name_2, ...]
x500: id_first_name, [id_second_name, ...]
```

9 Rubric: Subject to change

- 5% README
- 20% Documentation within code, coding, and style: indentations, readability of code, use of defined constants rather than numbers
- 75% Test cases: correctness, error handling, meeting the specifications
- Please make sure to pay attention to documentation and coding style. A perfectly working program will not receive full credit if it is undocumented and very difficult to read.
- A sample test case is provide to you upfront. You may change the value of #mappers and #reducers to test out your code. Think about other corner cases that may occur in the code, for example, an empty input file. Your code should be able to handle such cases. Please make sure that you read the specifications very carefully. If there is anything that is not clear to you, you should ask for a clarification.
- We will use the GCC version installed on the CSELabs machines(i.e. 9.3.0) to compile your code. Make sure your code compiles and run on CSELabs.
- **Please make sure that your program works on the CSELabs machines** e.g., KH 4-250 (csel-kh4250-xx.cselabs.umn.edu). You will be graded on one of these machines.