COMP 7500/7506 Advanced Operating Systems
Project 2: pWordCount: A Pipe-based WordCount Tool

Version 2.0
Revised: Feb. 18, 2022

Points Possible: **100**
Submission via **Canvas**

This is an individual assignment; no collaboration among students. Students shouldn't share any project code with any other student. Collaborations among students in any form will be treated as a serious violation of the University's academic integrity code.

Learning Objectives:

- To learn Unix Pipe, which is one of the first IPC mechanisms
- To get familiar with file I/Os in C
- To design a multi-process program in which there are two cooperating processes
- To Implement a C program using ordinary pipes
- Use the GDB tool to debug your C program in Linux

1. Overview

This project aims at providing you with an opportunity to design and implement a C program, where two processes are cooperating through Unix Pipes. Pipes is one of the first interprocess communication mechanisms or IPCs in early Unix systems. Unix pipes offer simple yet efficient ways of communicating among collaborating processes. In this project, we focus on ordinary pipes that enable two processes to communicate in a producer-consumer fashion. In other words, a producer process writes to one end of a pipe (i.e., write-end) and a consumer process reads from the other end (i.e., read-end).

2. A Pipe-based WordCount Tool

2.1 Counting Words in a Text File

Your program must (1) read a file containing a list of words, (2) count the total number of words in the file, and (3) print out the number of words.

Your program should create two cooperating processes, which communicate through Unix ordinary pipes. After the first process opens and reads a text file, the process sends the file content to the second process. The second process is in charge of counting the number of words and reporting the result back to the first process.

2.2 Sample Usage

The name of your tool should be `pwordcount`, which takes a file name as a user input.

Let us consider a file `"input.txt"`, which contains the following text – `"This is the second project of COMP7500 class."`

Here is a sample dialog (where the user input is depicted as **Bold**, but you do not need to display user input in bold.):

```
$./pwordcount input.txt
Process 1 is reading file "input.txt" now ...
Process 1 starts sending data to Process 2 ...
Process 2 finishes receiving data from Process 1 ...
Process 2 is counting words now ...
Process 2 is sending the result back to Process 1 ...
Process 1: The total number of words is 8.
```

In case the user doesn't provide an input file name, your program must offer an instruction on how to use your program. A sample dialog for this case is given below.

```
$./pwordcount
Please enter a file name.
Usage: ./pwordcount <file_name>
```

Important! Your program's output should match the style of the sample output. You will lose points if you do not use the specific program file name, or do not have a comment block on EVERY program you hand in.

2.3 Two Unix Pipes

In this project, you must create two pipes. The first pipe is used to deliver the input file's content from the process 1 to process 2. The second pipe aims to send a wordcount result from process 2 to process 1. You are required to implement your program using Unix pipes.

2.4 Two Cooperating Processes

Your mission is to create the following two processes:

- Process 1: Read a file containing a list of words.
- Process 2: Count the number of words and send the result back to Process 1.
- Process 1: Print out the number of words.

3. How to embark on this project?

3.1 Step 1: Creating A Child Process in A Parent Process

The first step is understanding how to create two cooperating processes (i.e., a parent process and a child process) using the `fork()` system call. Please refer to the webpage below for the following sample code.
timmurphy.org/2014/04/26/using-fork-in-cc-a-minimum-working-example/

```
#include <stdio.h>
#include <unistd.h>
int main(int argc, char **argv)
{
    printf("--beginning of program\n");
    int counter = 0;
    pid_t pid = fork();
    if (pid == 0)
    {
        // child process
        int i = 0;
        for (; i < 5; ++i)
        {
            printf("child process: counter=%d\n", ++counter);
        }
    }
    else if (pid > 0)
    {
        // parent process
        int j = 0;
        for (; j < 5; ++j)
        {
            printf("parent process: counter=%d\n", ++counter);
        }
    }
    else
    {
        // fork failed
        printf("fork() failed!\n");
        return 1;
    }
    printf("--end of program--\n");
    return 0;
}
```

3.2 Step 2: Create Unix Pipes

You should learn how to build two pipes using the `pipe(int fd[])` function, which creates a pipe that is accessed through the `fd[]` file descriptor. Please note that `fd[0]` is the read-end of the pipe, whereas `fd[1]` is the write-end of the pipe. The created pipe can be accessed through the `read()` and `write()` system calls. Like files, the write-end and read-end of pipes must be closed if they are no longer in use. A sample code illustrating how to create pipes is given below.

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <string.h>

#define BUFFER_SIZE 25
#define READ_END       0
#define WRITE_END      1

int main(void)
{
        char write_msg[BUFFER_SIZE] = "Greetings";
        char read_msg[BUFFER_SIZE];
        pid_t pid;
        int fd[2];

        /* create the pipe */
        if (pipe(fd) == -1) {
                fprintf(stderr,"Pipe failed");
                return 1;
        }

        /* now fork a child process */
        pid = fork();

        if (pid < 0) {
                fprintf(stderr, "Fork failed");
                return 1;
        }

        if (pid > 0) {  /* parent process */
                /* close the unused end of the pipe */
                close(fd[READ_END]);

                /* write to the pipe */
                write(fd[WRITE_END], write_msg, strlen(write_msg)+1);

                /* close the write end of the pipe */
                close(fd[WRITE_END]);
        }
        else { /* child process */
                /* close the unused end of the pipe */
                close(fd[WRITE_END]);

                /* read from the pipe */
                read(fd[READ_END], read_msg, BUFFER_SIZE);
                printf("child read %s\n",read_msg);

                /* close the write end of the pipe */
                close(fd[READ_END]);
        }

        return 0;
}
```

3.3 Step 3: Loading an Input File

The responsibility of the parent process is to read an input file and send data to the child process. In step 3, please implement the parent process that can successfully load the file and deliver data to process 2 (i.e., child process).

3.4 Step 4: Counting Words

Now you are in a position to implement a wordcount function. You must conduct a unit test of this function prior to carrying out the next step.

3.5 Step 5: Integrating the wordcount Function

In this step, you focus on integrating your wordcount function with the two pipes. Thus, the first pipe offers the input of the wordcount function, the output of the wordcount function feeds the second pipe.

3.6 Step 6: Testing

After integrating all the components into your program, you conduct a final test using various input files (e.g., a small file and a large file).

4. Programming Requirements

4.1 Programming Environment

You must implement your `pwordcount` in C. Please compile and run your system using the `gcc` compiler on a Linux box (either in Tux machines, computer labs in Shelby, your home Linux machine, a Linux box on a virtual machine, or using an emulator like Cygwin).

4.2 Function-Oriented Approach

You are *strongly suggested* to use a structure-oriented (a.k.a., function-oriented) approach for this project. In other words, you will need to write function definitions and use those functions; you can't just throw everything in the `main()` function. A well-done implementation will produce a number of robust functions, many of which may be useful for future programs in this project and beyond.

Remember good design practices include:

- A function should do one thing, and do it well
- Functions should NOT be highly coupled

4.3 File Names

You will lose points if you do not use the specific program file name, or do not have a comment block on **EVERY** program you hand in.

4.4 Usability Concerns and Error-Checking

You should appropriately prompt your user and assume that they only have basic knowledge of the tool. You should provide enough error-checking that a moderately informed user will not crash your system. This should be discovered through your unit-testing. Your prompts should still inform the user of what is expected of them, even if you have error-checking in place (see an example in Section 2.2).

5. Separate Compilation (Optional Requirement)

You don't need to use separate compilation. If you are unfamiliar with separate compilation, you are encouraged to following the instructions in this Section to practice separate compilation and create a makefile for this project.

What is Make?

Make is a program that looks for a file called "makefile" or "Makefile", within the makefile are variables and things called dependencies. There are many things you can do with makefiles, if all you've ever done with makefiles is compile C or C++ then you are missing out. Pretty much anything that needs to be compiled (postscript, java, Fortran), can utilize makefiles.

Format of Makefiles -- Variables

First, let's talk about the simpler of the two ideas in makefiles, variables. Variable definitions are in the following format:

```
VARNAME = Value
```

So, let's say I want to use a variable to set what compiler I'm going to use. This is helpful b/c you may want to switch from cc to gcc or to g++. We would have the following line in our makefile

```
CC = gcc
```

This assigns the variable CC to the string "gcc". To expand variables, use the following form:

```
${VARNAME}
```

So, to expand our CC variable we would say:

```
${CC}
```

Format of Makefiles -- Dependencies

Dependencies are the heart of makefiles. Without them nothing would work. Dependencies have the following form:

```
dependecy1: dependencyA dependencyB ... dependencyN
    command for dependency1
```

Check out the following links for more information on makefiles:
http://oucsace.cs.ohiou.edu/~bhumphre/makefile.html

6. Project Report

Write a project report that explains how you design and implement your wordcount tool. Your report should include data flow diagram, function prototypes, data structures, and discussions on design issues. Your report is worth 20 points.

Important! Your project report should provide sample input and output from your implemented program.

7. Deliverables

7.1 Final Submission

Your final submission should include:

- Your project report (see also Section 6).
- A copy of the complete source code of your wordcount tool.

Important! You must submit a single compressed file (see Section 7.2) as a .tar.gz file, which includes both a project report and source code.

7.2 A Single Compressed File

Now, submit your tarred and compressed file named `project2.tgz` through Canvas. You must submit your single compressed file through Canvas (no e-mail submission is accepted.

7.3 What happens if you can't complete the project?

If you are unable to complete this project for any reason, please describe in your final design document the work that remains to be finished. It is important to present an honest assessment of any incomplete components.

8. Grading Criteria

The approximate marks allocation will be:

1) Project Report: 20%
2) Implementation (i.e., Source Code): 70%
3) Clarity and attention to details: 10%

9. Late Submission Penalty

- Ten percent (10%) penalty per day for late submission. For example, an assignment submitted after the deadline but up to 1 day (24 hours) late can achieve a maximum of 90% of points allocated for the assignment. An assignment submitted after the deadline but up to 2 days (48 hours) late can achieve a maximum of 80% of points allocated for the assignment.
- Assignment submitted more than 3 days (72 hours) after the deadline will not be graded.

10. Rebuttal period

- You will be given a period a week (7 days) to read and respond to the comments and grades of your homework or project assignment. The TA may use this opportunity to address any concern and question you have. The TA also may ask for additional information from you regarding your homework or project.