

Nakul Patel

Bhargav Sonani

Akarsh Sardana

Alex Chan

Introduction to Computer Systems:

Project 1: InterProcess Communication Report

Note to Grader

For this project, we will be submitting a tar file. When we untar (or uncompress this tar file), you will see that it contains a readme pdf which is this report, as well as 7 folders, four folders for Problems 1 - Problems 4, and three folders for Problem 0, one for each part (a, b, and c). Each respective folder will contain the .c files the .h header files, the makefile, the input txt files, and the output txt files if applicable. Please refer to the report below to see how to execute and run each problem (certain problems have certain notes that the grader should be aware of). We have also pasted the contents of the makefiles below if there are any issues.

Problem 0 Makefiles

A:

```
Project0A: Project0A.c
    gcc Project0A.c -g -o Project0A
```

```
clean:
    rm Project0A
```

B:

```
Project0B: Project0B.c
    gcc Project0B.c -g -o Project0B
```

```
clean:
    rm Project0B
```

C:

```
Project0C: Project0C.c
    gcc Project0C.c -g -o Project0C
```

```
clean:
    rm Project0C
```

Problem 1 Makefile

```
Project1: Project1.c
    gcc Project1.c -g -o Project1
```

```
clean:
    rm Project1
```

Problem 2 Makefile

Prob2: Num2.c

```
gcc Num2.c -g -o num2
```

clean:

```
rm num2
```

Problem 3 Makefile

Prob3: num3.c

```
gcc num3.c -g -o num3
```

clean:

```
rm num3
```

Problem 4 Makefile

Project4: Project4.c

```
gcc Project4.c -g -o Project4
```

Problem 0

Input File:

Input.txt

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
```

*See provided input file for full input file

InputFile.py - Used vary content of input.txt

First parameter of range is start value

Second parameter of range is value right after end value

```
f = open('input.txt', 'w')
modulo = 1
l = range(1,100001)
for i in l:
    if i%modulo == 0:
        f.write(str(i)+"\n")
```

Output File: for given input.txt (range of values = 1-100,000)

Problem 0a:

```
Max = 100000
Min = 0
Sum = 705082704
```

Problem 0b:

```
Hi I'm process 1101 and my parent is 1100.  
Hi I'm process 1102 and my parent is 1101.  
Hi I'm process 1103 and my parent is 1102.  
Max = 100000  
Min = 0  
Sum = 705082704
```

Problem 0c:

```
1 Hi I'm process 10859 and my parent is 10858.  
2 Hi I'm process 10860 and my parent is 10858.  
3 Hi I'm process 10861 and my parent is 10858.  
4 Max = 100000  
5 Min = 1  
6 Sum = 705082704
```

Summary:

In problem 0, we were instructed to write a program that takes in an input text file consisting of a list of integers of varying sizes and to find the minimum, maximum, and sum of the integers. This problem must be done in three ways...

Problem 0a: ...with one process

Problem 0b: ...with multiple processes, where each process spawns at most one process

Problem 0c: ...with multiple processes, where each process spawns multiple processes

Explanation of Program:

Problem 0a:

The design decisions for this program structure include:

1. Program Structure: Read integers in file into an array and perform three functions
 - a. Open the file, read the input text file for integers and store them in an input text array called `input[]`, and then close the file. Repeat this until you reach the end of the input text file. If there are not any integers then close the file. Three separate functions are created (`find_Max`, `find_Min`, `find_Sum`), which all return an `int` and have two parameters, the input text array `input[]` and the number of integers (obtained from `lineCount`). See Design Decisions below for further explanations.
2. Two parameters for each mathematical function
 - a. `input`: Once in the main file, the input text file with the necessary integers is opened and the integers are then read. If the opened and read text file, `in_file`, is empty, then the file is closed. If the file isn't empty then the integers are scanned from top to bottom, reading the integer type data into the `&num` address, before putting the numbers into `input`.
 - b. `lineCount`: Similarly, `line_Count` takes in the input file with the integers as an argument. The `line_Count` function then opens the text file and reads the file until

it reaches the end of the file, with the feof command. The lineCount variable is iterated after every line number.

3. Creation of find_Max

- a. This function is responsible for finding the greatest integer in the input text file. An outer while loop is used to check that all the integers are evaluated, for the variable index is set to 0 and then compared to see if it is less than the size or lineCount of the text file (1 integer per line). After entering the while loop, the actual integer is compared to max, which is instantiated to be 0. If the integer is greater than 0, then that integer is set to be the new max. The index is then incremented to get to the next integer value and the while loop continues until index exceeds the number of integers, size.

4. Creation of find_Min

- a. This function is responsible for finding the smallest integer in the input text file, and has the opposite code structure of find_Max with the minimum value initially being set to be the first integer and then comparing all the integers one by one to this value, the smaller of the two integers taking on the min variable name.

5. Creation of find_Sum

- a. This function is responsible for finding the sum of all the integers in the input text file. Likewise to the previous two while loop statement conditions, the outer while loop checks that the index (set to 0) is less than the number of integers before adding the integer to itself. This continues until all the integers are taken care of.

*Important to note that all of these functions occur under one parent process

Problem 0b:

This problem is very similar to 0a, for line_Count, find_Max, find_Min, and find_Sum all are the same functions as before; however, the main function has additional lines of code to it, which divide the data into smaller subsections for multiple processes to operate on. The instructions state that "You may use any form of inter-process communication (IPC) to partition the file/array into chunks and distribute work to more than one processes." We designed our program to distribute the work to multiple processes by partitioning the file/array into small sections of data rather than by function (minimum, maximum, sum).

Once inside the main function, the input file is opened, read, scanned, and closed the same as before. The difference is that lineCount is divided by 3 (the number of integers divided by 3) and each max, min, and sum are now arrays of 3. The main function contains three forks which separate into parent and child processes...

Arbitrary process numbers assigned by appearance to show connection (see numbers below):

Child Process(3) linked to Parent Process(1)

ChildProcess(2) linked to Parent Process(2)

ChildProcess(1) linked to Parent Process(3)

| Program Structure (focusing on relation between processes/pipes) | | | | | |
|--|------------------|------------------|-------------------------------|-------------------------------|-------------------|
| ChildProcess (1) | ChildProcess (2) | Child Process(3) | Parent Process(1) | Parent Process(2) | Parent Process(3) |
| | | write(fd3[1]) | read(fd3[0]) write(fd2[1]) | read(fd2[0]) write(fd1[1]) | read(fd1[0]) |

3 Child Processes:

Describing forwards (outermost forked child process to innermost forked child process)

1. The first, outermost forked child process contains the buffMax, buffMin, and buffSum arrays of size int 3, which will be used later to calculate the overall max, min, and sum. This outer forked child will contain the next two forked child processes.
2. The second forked child process, which is nested within the outer forked child process, contains the integer variables bufMax, bufMin, and bufSum
3. The third forked child process, which is nested within both forked process calculates the max, min, and sum of the input.txt of integers of the last $\frac{1}{3}$ of data ($\frac{2}{3}$ to 1)

In all the three child processes, the file is opened and a PID statement is printed and appended to the one output file before being closed.

Now that the child processes have been accounted for (fork() == 0), we must address the respective parent processes ...

3 Parent Processes:

Describing backwards (innermost forked parent process to outermost forked parent process)

1. The parent process of the 3rd child process also calculates the max, min, and sum of the input.txt of integers of the middle $\frac{1}{3}$ of data ($\frac{1}{3}$ to $\frac{2}{3}$). Child process(3) and Parent Process(1) are connected via pipes so that child process(3) writes to parent process(1) to have it's subsection of max, min, and sum to be read. After reading in the memory of max, min, and sum of child process(3), parent process(1) calculates its own subsection of max, min, and sum, and writes those as well as the max, min, and sum, calculated in child process(3) to parent process(2) (the following else statement).
2. The parent process of the 2nd child process reads in the two maxes, two mins, and two sums, stores them in the memory of buffMax, buffMin, and buffSum respectively and then finds the max, min, and sum of the first $\frac{1}{3}$ of data (0 to $\frac{1}{3}$). Parent process(2) then writes the max, min, and sum it calculated from the two combined sets to no other than parent process(3).
3. Parent process(1) reads in the max, min, and sum from parent process(2) before calling find_Max, find_Min, and find_Sum one final time to calculate the overall max, min, and sum.

Problem 0c:

This program also calculates the max, min, and sum, but with multiple processes, where each process spawns multiple processes. In order to accomplish this, a shared_memory was used. In addition the buffer was shared so that other processes could access it, but also made to be anonymous so that 3rd party process could not obtain an address for it. Similar to part a and b of this problem, the opening, reading, and scanning of the input file was kept the same. Furthermore, the counting of the number of lines in the input file, and calculations of max, min, and sum of the set of values used the same methodology as described above.

Shared variable was used to take the min, max, and sum calculated for each third (each process) and store it in the variable. This is much more efficient because each of the processes can edit and view the edited value by each other.

Remark Response:

What we observed from testing the times of our 3 programs (0a, 0b, and 0c) was that for smaller inputs (< 1 million), the running times remained very pretty close to being equal because using the forked system call takes up time on its own and the operations themselves (finding min max and sum) are not complicated and don't take up too much time. When input sizes got larger, part c produced faster times than a and b because part c used a parallel division of work that made a larger difference.

What was Learned:

One thing learned from this assignment was how to use the shared variable in linux that all children processes have access to. This was extremely helpful in coding Problem 0c, which required the variable so that the processes can edit and view the min, max, and sum values stored by each other.

Problem 1

NOTE: To run the code, type in the command line:

"make"

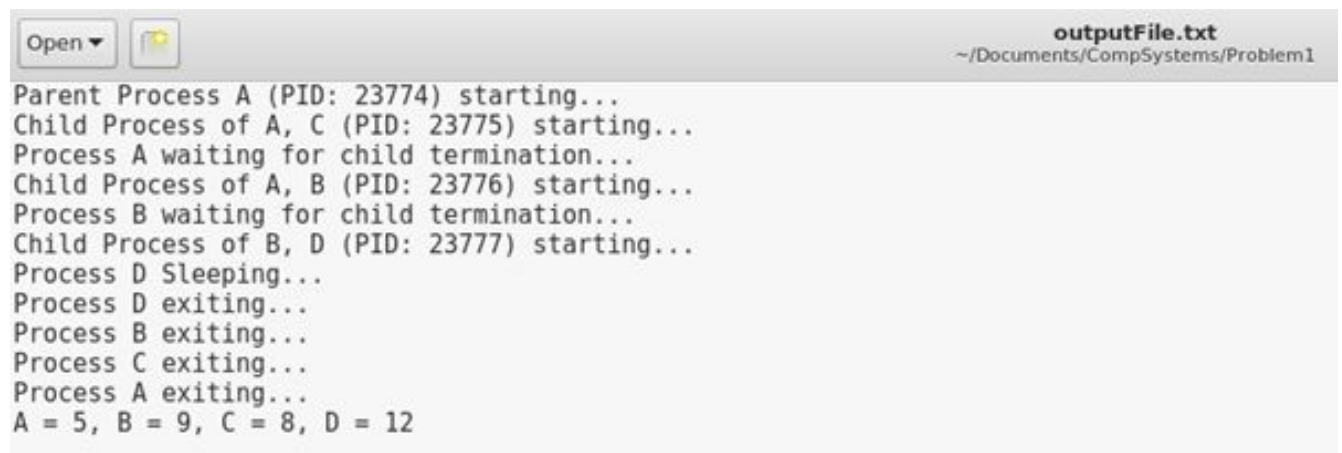
Followed by running the executable:

"./Project1"

NOTE: When run, this program will output to the command line. We have copy-pasted the results from the command line into outputFile.txt so that the graders can see our results.

Input File: None

Output File:

A screenshot of a terminal window with a title bar. The title bar contains an 'Open' button with a dropdown arrow, a file icon, and the filename 'outputFile.txt' with its path '~/.Documents/CompSystems/Problem1'. The terminal text shows a sequence of process starts and exits: Parent Process A (PID: 23774) starting... Child Process of A, C (PID: 23775) starting... Process A waiting for child termination... Child Process of A, B (PID: 23776) starting... Process B waiting for child termination... Child Process of B, D (PID: 23777) starting... Process D Sleeping... Process D exiting... Process B exiting... Process C exiting... Process A exiting... Finally, the values A = 5, B = 9, C = 8, D = 12 are printed.

```
Parent Process A (PID: 23774) starting...
Child Process of A, C (PID: 23775) starting...
Process A waiting for child termination...
Child Process of A, B (PID: 23776) starting...
Process B waiting for child termination...
Child Process of B, D (PID: 23777) starting...
Process D Sleeping...
Process D exiting...
Process B exiting...
Process C exiting...
Process A exiting...
A = 5, B = 9, C = 8, D = 12
```

Questions:

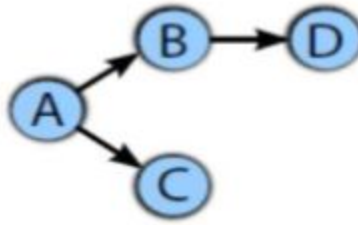
1. If the root process A is terminated prematurely, the children processes will become orphans, as their parent terminated before they finished executing. Eventually, they will be adopted and reaped by init.
2. If getpid() is used the process tree structure will change. The init process will appear in the tree.
3. The maximum random tree that you can create is limited by the total number of processes that a system can handle. While this varies from system to system, we can get a sense of how large this number is by putting in the following command in the command line:

```
/proc/sys/kernel/pid_max
```

When I run this on my system, I get the value 32768. That is around the total number of processes that a random tree can have at the maximum.

Summary:

In problem 1, we were instructed to write a program that generates the following process tree. Each process was designed to print a corresponding message every time it transitions to another phase, so that the validation of the correct program operation is feasible, and each process terminates with a different exit return code.



Explanation of Program:

The design decisions for this program structure include:

1. Program structure: outer for loop with three nested if/else statements
 - a. The outer for loop addresses the creation of child process B and C from parent process A. The first if statement addresses the creation of child processes in general. The second if/elseif statement differentiates child process C from child process B and vice versa. Lastly, the third if statement addresses the creation of grandchild process D within child process B. See Design Decisions below for further explanations.
2. Using forks as conditions in if/else statements
 - a. Three forks were used to derive the process tree of two children (B & C) and one grandchild (D); however, we use forks as conditions within the if/else statement in order to easily fork and then use the pid value of the forked process to discern whether the process is a parent or child. For instance, in line 26, `if((pids[i] = fork()) == 0)`, forks from parent process A, creating child process B. The pids are stored in a 2 index array `pids[2]`, with pid values either being 1 (for parent) or 0 (for child). Once a process is forked, the pid of the child process is 0, thus `pids[0] = 0` is in fact `== 0`. Therefore, we have successfully created Child C. See Design Decision 3 for how Child B is created.
3. Using a for loop to iterate twice in parent process A to create two child processes, C and B respectively
 - a. A for loop that iterates twice is used in line 25 of the program to create child process B and child process C. The creation of child process C is described in Design Decision 2, when `i = 0`. Once child process C is created, the for loop iterates again (and terminates) to `i=1`. When `i=1`, the first if statement is checked...`if((pids[i] = fork()) == 0)`. For the second iteration, this translates to `if((pids[1] == fork()) == 0)`. Since as mentioned above, once a process is forked the pid of the child process is 0, this if statement is true. The next nested if/else statement checks if `i == 0` or `i == 1`. When `i == 0` child process C is created. When `i == 1`, the block of code (under the else if condition) runs and child process B is successfully created. It is important to note that the for loop, which iterates from `i=0` to `i<2` is responsible for differentiating between the two child processes since the first if statement just addresses the creation of processes.
4. After creating child process B, fork again within child process B to create grandchild process D

- a. At this point, child process C and child process B have been created. In addition, it is important to note that child process C has an instruction of `sleep(10)`, while child process B has an instruction of `sleep(5)`. When the leaf processes execute a call to `sleep()`, the processes are suspended from program execution for a specified time. Thus, the if statement inside child process B checks if `fork() == 0`, which it does because the child process has an initial pid value of 0, thereby creating grandchild process D and giving process D time to execute.
- 5. Using the `wait()` and `WEXITSTATUS()` command to obtain the process' exit code and then pipe that code to parent process A
 - a. A key factor in communication between processes within our program is the use of pipes. A pipe is a system call that creates a communication link between two file descriptors in one direction. In general, the children write from `pipe(fd[1])` and the parents read from `pipe(fd[0])`. By using pipes, the children (leaf processes) are able to write back their "different return codes" to the parent (root process) so that they can all be printed together at the end (see line 65 of program).

What was Learned:

One thing learned from this assignment was how to insert forks efficiently and strategically in a program in order to create the correct process tree structure. Initially, we were unsure of the program structure, inserting forks in sequence right below the other in order to create two children; however, forks create copies of the parent process, which is everything below the instruction. Hence, the number of child processes created was incorrect. Afterwards, we used if/else statement to separate the processes. Normally when you `fork()` twice in a row you get 4 processes total, but if you use an if/else statement and `fork` inside one of the conditions, then the other process fulfilling the other condition is not affected.

Problem 2

NOTE: To run the code, type in the command line:

`"make clean"`

Then:

`"make"`

Followed by running the executable:

`./num2 test5.txt` ("`test5.txt`" is just a sample name of input file. Can change name)

NOTE: When run, this program will output to the command line. We have copy-pasted the results from the command line into `outputFile.txt` so that the graders can see our results.

Sample Input File:

```
1 A 6 B C D E F G
2 B 0
3 C 0
4 D 1 H
5 H 0
6 E 2 I J
7 J 2 P Q
8 I 0
9 P 0
10 Q 0
11 F 3 K L M
12 K 0
13 L 0
14 M 0
15 G 1 N
16 N 0
```

Corresponding Output File:

```
Process 6635 (A): My parent is 27758
Process 6636 (B): My parent is 6635. I have no children.
Process 6637 (C): My parent is 6635. I have no children.
Process 6638 (D): My parent is 6635
Process 6639 (H): My parent is 6638. I have no children.
Process 6640 (E): My parent is 6635
Process 6641 (I): My parent is 6640. I have no children.
Process 6642 (J): My parent is 6640
Process 6643 (P): My parent is 6642. I have no children.
Process 6644 (Q): My parent is 6642. I have no children.
Process 6645 (F): My parent is 6635
Process 6646 (K): My parent is 6645. I have no children.
Process 6647 (L): My parent is 6645. I have no children.
Process 6648 (M): My parent is 6645. I have no children.
Process 6649 (G): My parent is 6635
Process 6650 (N): My parent is 6649. I have no children.
```

Summary:

In this Problem, we are tasked with building a tree structure using tree node structs as well as a process tree based on a given input file. The input file will be of the following example format:

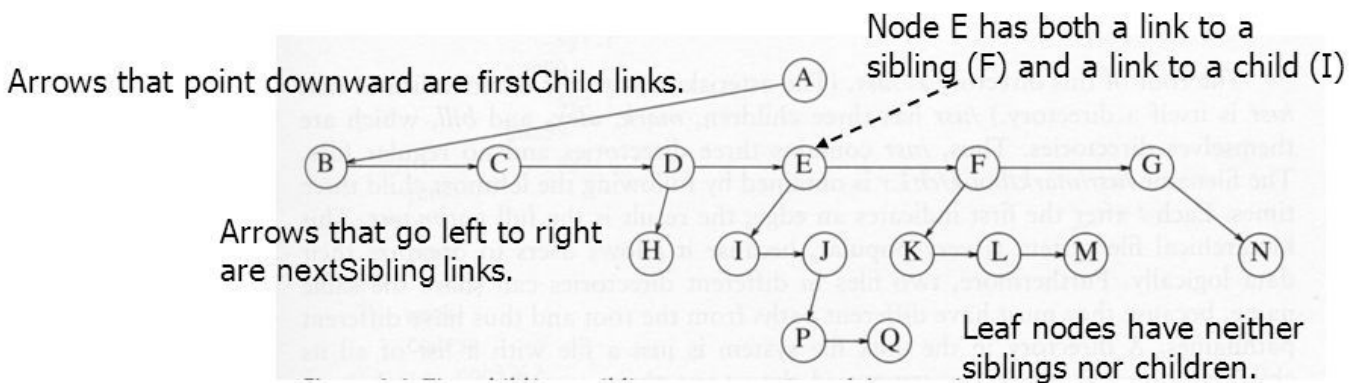
```
A 2 B C
B 1 D
D 0
C 0
```

Where the first line will be for the root of the tree. In each line, the first character is the letter name of the node/process. The second character is an integer, defining the number of children that this node has. If this integer is not 0, then the following character is the letter names of those children.

Specifically, we first build the tree structure containing the tree nodes. Then we use that tree structure to recursively build the process tree. In the process tree, we have parent processes wait on their children and we set up IPC pipes between them. The use of these pipes is so that after the process tree has been created, we can print out the tree structure/process tree and output it to the output file. We chose to print the tree by defining all the relationships of the tree: each process will state its pid, the name (letter) of the node, whether it is a lead node (has no children), and the pid of its parent.

Explanation of Program:

We start by first defining the struct node, which contains a char n (holding the name of the node), int child_no (holding the number of children for the node), struct node *kids (pointing to the notes first child), and struct node *sibling (pointing to that first child's subsequent siblings). We then use the input file to recursive build the tree structure containing this nodes. To give a better idea of how this tree structured will be formed, we have provided the following illustration below:



This sample diagram above is also defined in the input file. Furthermore, we can compare the output file that shows the printed process tree to the above diagram to confirm that we are correctly building the process tree. We can understand how the tree structure is built and formatted by looking at the first line of the input file as an example:

A 6 B C D E F G

Here, A is the root of the tree and it has 6 children. Therefore, if we call this node root, then `root->n = 'A'` and `root->child_no = 6`. The way this is linked is that A has a “kids pointer” to its first child B, and the rest of A’s children are linked via the sibling pointer. So:

`A->kids = B`

`B->sibling=C; C->sibling=D; D->sibling=E; E->sibling=F; F->sibling=G`

We then continue building the rest of the tree in this fashion by recursively calling this function of each of root A’s 6 children.

After the tree structure, we can build the process tree with relatively ease. The way we do this is by forking in the parent for each children the parent process has, and then calling the recursive function, called `buildProcessTree(root)` on the subtree when we are inside a child process. While forking we also set up pipes between the children to pass the name of the nodes, and we have parent processes wait on their children after they have forked for all of their children. Finally, after the process tree has been created, we are able to print out the process tree

What was Learned/Answering Questions:

In this program, we learned how to recursively build a process tree from a given input. Each input file can be of variable length and structure, but we were able to design a recursive method that works for all cases. Furthermore, we learned how to create pipes for these process so that we can pass important information and data between parent and child processes. The way that we ensure that we do not generate more processes than our system can handle is because we assume our input is limited/restricted by the letters of the alphabet. Because each node’s name must be a distinct letter, and there are 26 letters in the alphabet, we can only have a total of 26 processes, which is much lower than the total amount of processes our system can handle. It should be noted that we have assumed that the input file is of the correct format. The order of appearance of the start and termination messages (of our processes being printed) follows the order of Depth First Search (DFS). This is because, as we traverse the process tree, we do so by going all the way down (as depth as we can) until we hit a leaf node/process, at which point we go back up. This is clearly shown in the sample output above.

Problem 3

NOTE: To run the code, type in the command line:

“make clean”

Then:

“make

Followed by running the executable:

“./num3 test4.txt” (“test4.txt” is just a sample name of input file. Can change name)

NOTE: When run, this program will output to the command line. We have copy-pasted the results from the command line into outputFile.txt so that the graders can see our results.

NOTE: When run, this program will output to the command line. We have copy-pasted the results from the command line into output text files so that the graders can see our results.

Input File:

```
1 A 4 B C D E
2 B 3 F G H
3 C 2 I J
4 D 1 K
5 E 0
6 F 2 L M
7 G 0
8 H 1 N
9 I 1 0
10 J 0
11 K 1 P
12 L 0
13 M 1 Q
14 N 0
15 0 0
16 P 1 R
17 Q 0
18 R 1 S
19 S 0
```

Output File:

```
Process 27467 (A): My parent is 13391
Process 27468 (B): My parent is 27467
Process 27469 (C): My parent is 27467
Process 27470 (D): My parent is 27467
Process 27471 (E): My parent is 27467
Process 27471 (E): RAISED SIGSTOP!!
Process 27472 (F): My parent is 27468
Process 27473 (I): My parent is 27469
Process 27474 (G): My parent is 27468
Process 27478 (H): My parent is 27468
Process 27477 (L): My parent is 27472
Process 27475 (K): My parent is 27470
Process 27479 (O): My parent is 27473
Process 27476 (J): My parent is 27469
Process 27479 (O): RAISED SIGSTOP!!
Process 27474 (G): RAISED SIGSTOP!!
Process 27480 (M): My parent is 27472
Process 27476 (J): RAISED SIGSTOP!!
Process 27477 (L): RAISED SIGSTOP!!
Process 27481 (N): My parent is 27478
Process 27481 (N): RAISED SIGSTOP!!
Process 27473 (I): RAISED SIGSTOP!!!!
Process 27478 (H): RAISED SIGSTOP!!!!
Process 27469 (C): RAISED SIGSTOP!!!!
Process 27482 (P): My parent is 27475
Process 27483 (Q): My parent is 27480
Process 27483 (Q): RAISED SIGSTOP!!
Process 27480 (M): RAISED SIGSTOP!!!!
Process 27472 (F): RAISED SIGSTOP!!!!
```



```
Process 27468 (B): RAISED SIGSTOP!!!!
Process 27484 (R): My parent is 27482
Process 27485 (S): My parent is 27484
Process 27485 (S): RAISED SIGSTOP!!
Process 27484 (R): RAISED SIGSTOP!!!!
Process 27482 (P): RAISED SIGSTOP!!!!
Process 27475 (K): RAISED SIGSTOP!!!!
Process 27470 (D): RAISED SIGSTOP!!!!
----WAITING TO RECEIVE SIGCONT SIGNAL----
Process 27467 (A): ACTIVATED A
Process 27468 (B): \\ACTIVATED!!!!
Process 27468 (B): Waiting for children.....
Process 27472 (F): \\ACTIVATED!!!!
Process 27472 (F): Waiting for children.....
Process 27477 (L): \\ACTIVATED!!!!
Process 27477 (L): TERMINATED
Process 27480 (M): \\ACTIVATED!!!!
Process 27480 (M): Waiting for children.....
Process 27483 (Q): \\ACTIVATED!!!!
Process 27483 (Q): TERMINATED
Process 27480 (M): TERMINATED
Process 27472 (F): TERMINATED
Process 27474 (G): \\ACTIVATED!!!!
Process 27474 (G): TERMINATED
Process 27478 (H): \\ACTIVATED!!!!
Process 27478 (H): Waiting for children.....
Process 27481 (N): \\ACTIVATED!!!!
Process 27481 (N): TERMINATED
Process 27478 (H): TERMINATED
Process 27468 (B): TERMINATED
```

```

Process 27469 (C): \\ACTIVATED!!!!
Process 27469 (C): Waiting for children.....
Process 27473 (I): \\ACTIVATED!!!!
Process 27473 (I): Waiting for children.....
Process 27479 (O): \\ACTIVATED!!!!
Process 27479 (O): TERMINATED
Process 27473 (I): TERMINATED
Process 27476 (J): \\ACTIVATED!!!!
Process 27476 (J): TERMINATED
Process 27469 (C): TERMINATED
Process 27470 (D): \\ACTIVATED!!!!
Process 27470 (D): Waiting for children.....
Process 27475 (K): \\ACTIVATED!!!!
Process 27475 (K): Waiting for children.....
Process 27482 (P): \\ACTIVATED!!!!
Process 27482 (P): Waiting for children.....
Process 27484 (R): \\ACTIVATED!!!!
Process 27484 (R): Waiting for children.....
Process 27485 (S): \\ACTIVATED!!!!
Process 27485 (S): TERMINATED
Process 27484 (R): TERMINATED
Process 27482 (P): TERMINATED
Process 27475 (K): TERMINATED
Process 27470 (D): TERMINATED
Process 27471 (E): \\ACTIVATED!!!!
Process 27471 (E): TERMINATED
Process 27467(A): TERMINATED~~~~~!!!!
Order of Activation:
(0)A, (1)B, (2)F, (3)L, (4)M, (5)Q, (6)G, (7)H, (8)N, (9)C, (10)I, (11)O, (12)J, (13)D, (14)K, (15)P, (16)R, (17)S, (18)E,

```

Questions:

1. Rather than `sleep()`, in this program we used signals to synchronize this entire process tree. This is advantageous because signals provide feedback on the status of the process. For example, a parent process is able to check the status of its child using `wait()` or `waitpid()`. If the status of the child process changes, such as it stops or terminates, then the parent process could respond accordingly. However, when using `sleep()`, the parent process is not aware that the child is “sleeping”. This is because it still causes the child to run. Another advantage is that synchronization becomes accurate and scalable with the use of signals. For example, suppose I want to perform two tasks in different processes, task A and task B, and I want to ensure that task B happens after task A. When using `sleep()`, I have to predict how many seconds task A will take, and then only perform task B. However, with signals, as soon as task A is done executing, I can send a signal for task B to start. In this way, the program becomes more efficient as time is not unnecessarily waited and provides security for the order of the program. Because suppose that we added more to what task A had to accomplish, then it might take longer to execute, and therefore, it might cause task B to start even before task A is finished. This is why signals are advantageous as they allow for flexibility, accuracy, and efficiency.
2. In this program, the role of `wait_for_children()` would be to make sure that all children processes are stopped before stopping a parent process. In this code, the function is not explicitly defined, but each process does still check for this. This provides the benefit of ensuring that there will be no orphan processes in this process tree. If this was not checked for and omitted, then there could be a possibility of the entire process tree not being completed and a orphan process. For example, suppose A had two children, process B and process C. Then if process A just raised it's `SIGSTOP` signal after creating process B, then process C could never be created since it's process A's child. Therefore, ensuring that all children processes are stopped is crucial to this problem.

Summary:

In problem 3, we are tasked with expanding on Problem 2, but handling the process with signals, specifically SIGSTOP and SIGCONT, in a recursive manner. The input will still be in the same format as in number 2. This time, however, all the processes are stopped with the SIGSTOP signal. But before a process calls SIGSTOP, it has to make sure all child processes have stopped. Then after all processes stop, the root will first receive a SIGCONT signal and activate all child processes in a recursive dfs manner. When the process receives SIGCONT, a corresponding message should be printed, activating child process one after another.

Explanation of Program:

This program is very similar to that of Problem 2. The biggest change that was made is how the process tree is made after the tree data structure in memory is built. In the previous part, we used wait() to be make sure that the process tree is built in a dfs fashion. However, in this problem, we just have to make sure that all the processes are stopped. Hence, creating the process tree using dfs will not quite work because of how the recursive function is written. In particular, the recursive function to build all the processes first is called with the root node. Then it forks() to create a child. The child process, recursively calls the buildProcessTree function on it's child while the parent recursively calls it with all the siblings. After this, each process looks to raise the SIGSTOP signal, but before it stops, it waits until all children processes are stopped. It does this by using the waitpid() function. So, eventually, the function reaches a leaf node, which is a node that has no children. Since it consequently has no children, we don't have to check if children are stopped, and therefore, it raises SIGSTOP. This notifies the leaf node's parent that a child process has been stopped. However, we have to make sure that all children processes are stopped, not just one. So the parent continues to check if there are any other children that have yet not been stopped. Since this recursively occurs, eventually all processes are able to stop and it is illustrated by the order of which processes stop in the output file.

As it is confirmed that all processes are stopped, the root process then raises the SIGCONT signal. So the order of sequence should be that root receives SIGCONT, and the root activates all its children in dfs order. We use the kill() command to specifically raise SIGCONT on a specific process. Then, as each process is activated, it writes to the output file. After activation, the process waits for all its children to terminate before terminating itself. So another wait() command is used for that. We are able to see in the output file that order of activation is in fact dfs. But in order to make this clearly visible and convenient, the activation order is written at the end of the output file after all the messages and diagnostics are outputted.

however, there were several manipulations to the buildProcessTree function in recursively calling SIGSTOP and SIGCONT. Placing signal SIGSTOP to stop all the processes in one group was the first part of the problem tackled. By placing SIGSTOP within the parent process, we were able to stop all child processes in a recursive manner, while having the parent process still running. The parent process needed to be kept running, because if the parent process also received SIGSTOP, then the program would not be able to receive signal SIGCONT to be resumed. The strategy of placing raise(SIGSTOP) in the recursive function to stop the child

processes and not the root turned out to be a successful method; however, using SIGCONT to resume the processes was less intuitive.

What was Learned:

One thing that was learned that when recursively calling signal SIGSTOP, you must be very specific where to place the signal, because once SIGSTOP is called in the parent process, the entire process stops and can not be resumed from the code itself. Thus, all the child process should receive a signal from SIGSTOP, but the sole parent process, should remain active if you plan on sending additional signals to the program within the code itself (not the command line).

Problem 4

*****NOTE:** I wrote the code for number 4 on my MAC, so the code needs to be run on a MAC OS as well or else there might be errors due to the lack of portability between code on MAC OS and other operating systems.

NOTE: To run the code, type in the command line:

“make”

Followed by running the executable with input text file as single argument:

“./Project4 (name of input text file here)”

NOTE: When run, this program will output to the command line. We have copy-pasted the results from the command line into output text files so that the graders can see our results.

Summary:

In Problem 4, we are tasked with building a process tree that represents a given expression, and then using that process tree to evaluate the expression. It should be noted that the only operations that the expression can handle/contain are addition (“+”) and multiplication (“*”), as stated per the instructions of the assignment. When building the process tree, all of the “leaf nodes” will be the integer numbers, whereas the internal nodes and the root of the tree will be an operation. The point of building a process tree is so that we can calculate segments of the expression in parallel, where each parent process performs its operation on the value of its two children. Each parent process will have two children processes; therefore, we are working with a binary tree. Furthermore, the project recommends that we have intermediate printouts as well. Therefore, we will print out the relationships of the expression tree as we traverse through it. After building the whole process tree, we will send the value of the leaf nodes back to their parents using IPC pipes.

Explanation of Program:

The way that we decided to solve this problem is by first translating the given input expression, which will be in infix notation, into postfix notation by using a well known algorithm.

This algorithm is implemented in `char* infixToPostFix(char *e){}` which takes the input expression in infix notation and returns the same expression in postfix notation. One challenge that we had to deal with was working with multi-digit integers (i.e. being able to handle integers that are not just single digit). This is a challenge because our algorithm/function converts from infix notation to postfix by iterating through the input expression, one character at a time, so we have to be sure that it doesn't split up a number that is more than one digit long (for example, we have to ensure that the number 45 is not broken into two numbers, 4 and 5). The method then returns the postfix notation, where there is a space between every different number and operation.

Next, we use this postfix notation to actually build the expression tree structure using nodes. The postfix notation is broken up into the various nodes, based on the placement of the spaces; we use a string tokenizer, where the spaces are the delimiters and the resulting tokens are the nodes. Each node struct stores the data of that node (an operation or an integer value) and a pointer to a left and right node/subtree. We use a stack that stores these node structs to build the expression tree. As we iterate through the nodes in order, we push a node to the stack if it contains an integer value; when we reach a node that refers to an operation, we pop two integer nodes from the stack and make them children of the integer node, and then we push that subtree back into the stack. In the end, only one node should remain in the stack, which is the root of the entire expression tree.

We can pop the root node of the tree structure and recursively build the process expression tree exactly the same way we recursively build the expression tree in Problem 2. The function that builds the process tree will fork twice for each root it's given, once for the left subtree and once for the right subtree; we will then recursively call this function on the root of each subtree and continue this until we reach our base case, which is that the root is a leaf node and contains no subtree.

Once we have the process tree built, we can easily find the value of the expression by starting at the bottom of the process tree at the "leaf nodes" and working our way up. Each child process will send a value to its parent process through the use of pipes. Leaf node processes will simply send the integer value to the parents. Each non-leaf node process will take the integer values of its two children and perform its operation on the integer values; these processes will also then send the resulting value to its parent process. We will continue this until we reach the initial parent process (the "root"), and its value will be the total value of the expression.

Assumptions Made:

- Operations will only be * or +
- Operands will only be positive integers, as negative integers are not expected to be handled (if it were expected to be handled, we would just have a minus operation as well)
- Input expression can only contain positive integers, "+", "*", "(", and ")"
- Input expression must be properly formatted: same number of "(" and ")"

Test Cases (input via text files):

1. $10 * (5+6)$ - Expected Answer = 110

```
1 Input Expression: 10 * (5+6)
2 Postfix Expression: 10 5 6 + *
3 Tree Build Successful!
4 I am parent process 10379. My value is *
5 I am child process 10380. My value is 10. My parent process is 10379
6 I have no children. I am a leaf node process 10380. My value is 10. My parent process is 10379
7 I am child process 10381. My value is +. My parent process is 10379
8 I am parent process 10381. My value is +
9 I am child process 10382. My value is 5. My parent process is 10381
10 I have no children. I am a leaf node process 10382. My value is 5. My parent process is 10381
11 I am child process 10383. My value is 6. My parent process is 10381
12 I have no children. I am a leaf node process 10383. My value is 6. My parent process is 10381
13 Evaluated Expression = 110
```

2. $10 * (7*(5+6))$ - Expected Answer = 770

```
1 Input Expression: 10 * (7*(5+6))
2 Postfix Expression: 10 7 5 6 + * *
3 Tree Build Successful!
4 I am parent process 10397. My value is *
5 I am child process 10398. My value is 10. My parent process is 10397
6 I have no children. I am a leaf node process 10398. My value is 10. My parent process is 10397
7 I am child process 10399. My value is *. My parent process is 10397
8 I am parent process 10399. My value is *
9 I am child process 10400. My value is 7. My parent process is 10399
10 I have no children. I am a leaf node process 10400. My value is 7. My parent process is 10399
11 I am child process 10401. My value is +. My parent process is 10399
12 I am parent process 10401. My value is +
13 I am child process 10402. My value is 5. My parent process is 10401
14 I have no children. I am a leaf node process 10402. My value is 5. My parent process is 10401
15 I am child process 10403. My value is 6. My parent process is 10401
16 I have no children. I am a leaf node process 10403. My value is 6. My parent process is 10401
17 Evaluated Expression = 770
```

3. $(10+1) * (7*(5+6))$ - Expected Answer = 847

```
1 Input Expression: (10+1) * (7*(5+6))
2 Postfix Expression: 10 1 + 7 5 6 + * *
3 Tree Build Successful!
4 I am parent process 10425. My value is *
5 I am child process 10426. My value is +. My parent process is 10425
6 I am child process 10427. My value is *. My parent process is 10425
7 I am parent process 10426. My value is +
8 I am child process 10428. My value is 10. My parent process is 10426
9 I have no children. I am a leaf node process 10428. My value is 10. My parent process is 10426
10 I am child process 10429. My value is 7. My parent process is 10427
11 I have no children. I am a leaf node process 10429. My value is 7. My parent process is 10427
12 I am parent process 10427. My value is *
13 I am child process 10430. My value is 1. My parent process is 10426
14 I have no children. I am a leaf node process 10430. My value is 1. My parent process is 10426
15 I am child process 10431. My value is +. My parent process is 10427
16 Evaluated Expression = 847
17 I am parent process 10431. My value is +
18 I am child process 10432. My value is 5. My parent process is 10431
19 I have no children. I am a leaf node process 10432. My value is 5. My parent process is 10431
20 I am child process 10433. My value is 6. My parent process is 10431
21 I have no children. I am a leaf node process 10433. My value is 6. My parent process is 10431
22 Evaluated Expression = 847
```


What was Learned/Answering Questions:

Because our process tree has a binary tree structure, each process will either have 1, 2, or 3 pipes. Child processes that are leaf nodes will only have one pipe, for sending its corresponding integer value to its parent. The initial parent process, the root, will have two pipes, each pipe for receiving the integer value from each of its two children processes. Finally, the internal nodes/processes will have three pipes, two for receiving the value from each of its two children processes, and a third for sending the resulting value (after performing its operation on the two values) to its parent process. According to our knowledge, this is the only way to set up the pipes and still allow for parallel execution of the processes. That being said, as recommended by the additional remarks/clarifications, we did try other forms of IPC as well in addition to pipes. For example, we tried using exit statuses as well to return the values of subexpressions, and we saw that this method worked too.

Table of Contributions:

| Question | Alex | Akarsh | Bhargav | Nakul |
|----------|------|--------|---------|-------|
| 0 | X | | | X |
| 1 | X | | | X |
| 2 | | X | X | |
| 3 | X | X | X | |
| 4 | | X | X | X |