

Contributions:

Ravi: Problem 1, Question 3 Code and WriteUP. Variation Question from Problem 1 Code and Writeup Problem 2 Code and Questions 1, 2, and 3 WriteUp

Rohan: will help out with remaining parts in Problem 1, help with Problem 3

Milan: Problem 1, I will help with question 1 and other remaining questions

Preston: Problem 1 question 1 and 2 code

Parth: problem 1 question 6,7

Anthony: Problem 2, Question 3 and assistance with question 2.

Problem 1

** Assumed that Array Indices start at 0 just like C Programming is designed to be

1. Code on GitHub, Also Attached as C File to Submission Along with MakeFile

What I Learned From Problem:

- Since this problem was very straightforward in asking us to use only 1 process, we learned how to load numbers from a File into an Array, and perform simple computations on that array by looping through it. Furthermore, I also learned how to identify certain numbers in the array(i.e. In this case numbers equal to -1)
- I also learned how to evaluate the execution time of a program using the built in system command clock(). Using this system call, I was able to determine the execution time of the program and compare the performance of this program with other programs throughout my project.

Program Information/Overview:

- The 1st Input(argv[1]) must be equal to the number of positive integers to place in the File
- The 2nd input(argv[2]) must be equal to the number of Hidden Keys(i.e. integers whose value is -1) to place in the File
- The 3rd Input (argv[3]) must be equal to the number of Hidden Keys, out of the total number of Hidden Keys in the File, to identify. Hence, argv[3] <= argv[2]

Design Decisions:

- Very trivial program so we just used a loop to loop through the array and keep a running total of the maximum number observed thus far, the total sum observed thus far, and the number of hidden numbers observed thus far. If the number of

hidden numbers observed exceeded argv[3], we stopped searching for hidden numbers. At the end of the program, we printed the maximum and average of the numbers in the array as the problem description asks us to do

Runtime Analysis:

- I did a runtime analysis because I wanted to compare the performance of this program to the later programs we developed to see if running this program with more processes actually influenced the runtime of the program

To evaluate the performance of this program, I ran this program on my Operating System(Engsoft 20.0.4 Ubuntu). I used the clock() system call to determine the start time of the program and the end time of the program and simply performed a subtraction to determine the runtime of the program. To determine the runtime, since the runtime can certainly vary from execution to execution, I executed the program, for each list size, multiple times and then I calculated a rough average runtime.

I have tabulated the data below in a data table. I used the recommended list sizes on the project document(i.e. 5000, 10000, 100000, 1000000)

| List Size | Runtime |
|-----------|----------|
| 5000 | 0.006422 |
| 10000 | 0.012482 |
| 100000 | 0.064785 |
| 1000000 | 0.490943 |

There is nothing much surprising with these runtimes. We will naturally expect that the program will have a shorter runtime for a list size that is smaller. The reason for this is that a smaller list will have less data for our program to process. Hence, the number of loop iterations will be less.

2. Code on GitHub, Also Attached as C File to Submission Along with MakeFile

What I Learned From Problem:

- How to create a Process Tree and use IPC Pipes to pass information between Nodes in the Process Tree. I had to use multiple pipes in my program to send data between parent process and child process and vice versa. In lecture we learned that IPC pipes are unidirectional and half duplex(i.e. Data can travel one way at a time), I had to make two pipes for each process. One pipe would enable a process

to retrieve data from its child process and the second pipe would enable a process to send data to its parent process.

- How to create processes using the fork() system call
- How to ensure that child processes are terminated accordingly and are not remaining in the system as zombies as I am performing a Depth First Search Traversal of the process tree. To do this I had to use the kill() system call in conjunction with the waitpid() system call. After receiving data from its child process, I had the process send a SIGKILL signal to the child process via kill(). Since SIGKILL cannot be caught or handled, the child process will be killed. Furthermore, I used the waitpid() system call to enable the child process to be reaped(i.e. The process id of the child process is removed from the process table)
- I also learned how to dynamically allocate and free memory in this program, something which was a new concept to me since my previous background in Computer Science was in Java! When I was creating child processes in my program, I had to create data structures for each child to store the results of its computations/finding hidden elements of an array. This required me to use malloc() to create memory dynamically. Furthermore, I had to use free(), when each child process was terminated, to free memory that I had previously allocated.
- I also learned how to create a File using a File pointer, write data to that file, and read data from a file into an Array Data Structure
- I learned how to generate random numbers using the rand() call in the C library

Program Information/Overview:

- The 1st Input(argv[1]) must be equal to the number of positive integers to place in the File
- The 2nd input(argv[2]) must be equal to the number of Hidden Keys(i.e. integers whose value is -1) to place in the File
- The 3rd Input (argv[3]) must be equal to the maximum number of processes that can be in our process tree. Please note that this INCLUDES the main process. So, if you want your process tree corresponding to this program to have a maximum of 10 processes, set argv[3] to 10. This means that a total of 9 processes will be generated via the fork() call. The main process already exists when we execute this program!
- The 4th Input (argv[4]) must be equal to the number of Hidden Keys, out of the total number of Hidden Keys in the File, to identify. Hence, argv[4] <= argv[2]

Design Decisions:

- When I was making this program, I used a structure in C to represent each process. Since we were given information in the problem statement that each

process had a maximum of one child, it was quite easy to generate processes recursively rather than just pre-defining an entire tree structure.

- For the calculations and identifying the hidden keys in the file, I made two separate functions in my program. Why did I do this?

The reason I set it up this way is because performing the computations on the array can easily be done in a parallel manner by the processes we generate. The maximum value in a given array is basically the maximum value among the maximum values in each subarray. To calculate the average, we can first calculate the sums in each of the subarrays and this can clearly be done independently. Then, we can take our results from the subarrays and put them together. We can add all the sums together to get a sum for the original array. Furthermore, we can simultaneously calculate the total number of elements in each subarray. We can accomplish this by having each process calculate the total number of elements in its respective subarray. Then, we can put these results that we obtain from our child processes together. Then, we will get the total amount of numbers in our array. Then, we divide the sum by the total amount of numbers to get the average. However, identifying a certain number of hidden elements in the array cannot be done 100% in a parallel manner. We need one process to wait for the previous process to complete. Why is this the case? Well, if we had all processes running concurrently to identify a certain number of hidden elements in the array, it could well be the case that all these processes identify more than that number.

Consider the following example: [-1, 2, 3, -1, 4, 5, -1, -1, 6]

This array partitioned into equal subarrays whose sizes are 3 elements would be [-1, 2, 3], [-1, 4, 5], and [-1, -1, 6]. Let's say our program goal is to only identify 2 hidden elements. Well, when running each child process concurrently, the 1st child process will work with [-1, 2, 3], the 2nd child process will work with [-1, 4, 5], and the 3rd child will work with [-1, -1, 6]. Since the requirement is to identify 2 hidden elements, the 1st child will identify 1 (i.e. The item at index 0), the 2nd child will identify 1 (i.e. The item at index 0) and the 3rd child will identify 2 (i.e. The items at indices 0 and 1). Across all our subprocesses, we have identified 4 hidden elements when the requirement was to only find 2. The solution to this is to have our child processes run in some predefined order. Hence, if we run Child Process 1 first, it will identify 1 hidden element. Then, we could run Child Process 2 to identify 1 hidden element. When it comes to child process 3, we notice that Child Processes 1 and 2 have already identified 2 hidden elements and we don't need to identify any more hidden elements. This can be easily accomplished with pipes as I will explain later on

How I developed the Calculation Function:

Since we knew that each process had a maximum of one children, I already had an idea of what the tree structure would look like. The tree essentially would have a shape of a linked list where the root node of the tree corresponds to the front of the linked list data structure

With regards to the pipes, each process had 1 read pipe and 1 write pipe. The write pipe was used for a process to send data to its parent process. The read pipe was used for a process to retrieve data from its child process.

If we had already created enough processes, with the threshold given to us by `argv[3]`, I would have the process just perform computations on the subarray it was assigned, and write the values it has obtained from these computations(i.e. The maximum of its subarray, the total sum of the elements in its subarray, the average of the elements in its subarray) to its write pipe.

If we haven't yet reached the threshold of the number of processes we can create, with the threshold given to us by `argv[3]`, our program will execute a `fork()` system call to create a child process. The parent process will execute a `read()` system call to read data from its read pipe. Based on what we have studied about the `read()` system calls in the lectures, we know that the `read()` system call will BLOCK until data has been written to the pipe. For the child process, we recursively call the calculation function and give it a new read pipe. Since the child process needs to write data to the process itself and the process must read data from the child process, The write pipe for the child process will just be the read pipe of the parent process.

Based on this information, we can see that the parent process will be blocked until the child process has finished writing data to its write pipe. To speed up the process, prior to executing the `read()` system call, I had the parent process finish its computations for its assigned subarray. That way, when the child process has written data to the pipe, the parent process will already have finished its computations and we can just combine the data which will take a negligible amount of time.

To ensure that all processes terminate gratefully and to ensure that we have no zombie processes, once the child process finished writing data to the pipe, I had the parent process send a `SIGKILL` signal to the child process and then enter into a `waitpid()` system call. Since the `SIGKILL` signal cannot be caught, the child process will be killed and the parent process will reap the child process(i.e. The pid of the child process will be removed from the process table) in the `waitpid` system call.

How I developed the Hidden Identify Function:

Since we knew that each process had a maximum of two children, I already had an idea of what the tree structure would look like. The tree essentially would have a shape of a linked list where the root node of the tree corresponds to the front of the linked list data structure

With regards to the pipes, each process had 1 read pipe and 1 write pipe. The write pipe was used for a process to send data to its parent process. The read pipe was used for a process to retrieve data from its child process.

If we had already created enough processes, with the threshold given to us by `argv[3]`, I would have the process just find hidden keys on the subarray it was assigned, and write the amount of hidden keys it has found to its write pipe.

If we haven't yet reached the threshold of the number of processes we can create, with the threshold given to us by `argv[3]`, our program will execute a `fork()` system call to create a child process. The parent process will execute a `read()` system call to read data from its read pipe. Based on what we have studied about the `read()` system calls in the lectures, we know that the `read()` system call will BLOCK until data has been written to the pipe. For the child process, we recursively call the calculation function and give it a new read pipe. The write pipe for the child process will just be the read pipe of the parent process!

Based on this information, we can see that the parent process will be blocked until the child process has finished writing data to its write pipe. In the meantime, the parent could not find hidden integers on its own for its own subarray. As I mentioned before, the identity hidden part is not completely parallelizable like the computation part.

To ensure that all processes terminate gracefully and to ensure that we have no zombie processes, once the child process finished writing data to the pipe, I had the parent process send a SIGKILL signal to the child process and then enter into a `waitpid()` system call. Since the SIGKILL signal cannot be caught, the child process will be killed and the parent process will reap the child process (i.e. The pid of the child process will be removed from the process table) in the `waitpid` system call.

How did I partition the Array Among Processes?

- To partition the array among processes, I used a very trivial partitioning scheme. In the beginning, I assigned the entire array to the main process. Since I already knew the maximum number of processes, to make my program as efficient as possible, I had my program generate this maximum number! Hence, my tree structure, since each process has 1 child and 1 parent process, closely resembled a linked list data structure.
- We know that `argv[3]` represents the maximum number of processes we can create, including the main process. Hence, every time I used the `fork()` system call, I assigned the subarray `A[1: argv[3]]` to the main process, `A[argv[3] + 1: 2 * argv[3]]` to the child process of the main process and so on and so forth.

Runtime Analysis:

- Runtime analysis was performed to get an insight into how this program would perform relative to the program that I wrote in Part 3

| List Size | Number of Processes | Runtime |
|-----------|---------------------|----------|
| 5000 | 5 | 0.007988 |
| | 25 | 0.006482 |
| | 50 | 0.007266 |
| | 125 | 0.006626 |
| | 450 | 0.007826 |
| 10000 | 5 | 0.013528 |
| | 25 | 0.013077 |
| | 50 | 0.011480 |
| | 125 | 0.014844 |
| | 450 | 0.013712 |
| 100000 | 5 | 0.063023 |
| | 25 | 0.060658 |
| | 50 | 0.067921 |
| | 125 | 0.064644 |
| | 450 | 0.067086 |
| 1000000 | 5 | 0.499937 |
| | 25 | 0.482171 |
| | 50 | 0.482709 |
| | 125 | 0.482316 |
| | 450 | 0.489390 |

Based on this experimental data, we can conclude the following about our operating system(on the Engsoft Machine(Ubuntu 20.0.4 Operating System)):

Furthermore, based on my experimental data, in regards to increasing the number of processes, the trend on my Operating System seems to be that a small number of processes(i.e. Close to 5 processes) and a large number of processes(i.e. Roughly near 450) tend to have a larger runtime.

When I use around 50-125 processes, my experimental data seems to suggest that the runtime has hit a minimum. My proposed explanation for this is that when we are using a smaller number of processes, each node in the process tree will be assigned a larger portion of the original array. Hence, performing the operations on that portion of the array and finding the hidden keys will obviously take more time. When we are using a larger number of processes, although each node in the process tree is assigned a relatively smaller portion of the original array, there will be a larger overhead due the fact that we will require more pipes to communicate between the processes, we will need to use the wait/waitpid system call more often for the parent processes to reap the child processes, and the Operating System will have to perform context switching more often. Performing context switching, depending on the hardware on that particular system, will certainly incur some overhead cost and this will increase the runtime. This trend seems to be visible for all of the list sizes.

Additionally, when we increase the list size, the runtime of the program increases, but this is completely expected since we are given more data to process.

3. Code on Github, Also Attached as C File to Submission Along with MakeFile

What I Learned From Problem:

- How to create a Process Tree and use IPC Pipes to pass information between Nodes in the Process Tree. I had to use multiple pipes in my program to send data between parent process and child process and vice versa. In lecture we learned that IPC pipes are unidirectional and half duplex(i.e. Data can travel one way at a time), I had to make two pipes for each process. One pipe would enable a process to retrieve data from its child process and the second pipe would enable a process to send data to its parent process.
- How to ensure that child processes are terminated accordingly and are not remaining in the system as zombies as I am performing a Depth First Search Traversal of the process tree. To do this I had to use the kill() system call in conjunction with the waitpid() system call. After receiving data from its child process, I had the process send a SIGKILL signal to the child process via kill(). Since SIGKILL cannot be caught or handled, the child process will be killed. Furthermore, I used the waitpid() system call to enable the child process to be reaped(i.e. The process id of the child process is removed from the process table)
- How to create processes using the fork() system call
- How to create various data structures to represent process trees and analyze their effectiveness relative to each other
- I also learned how to dynamically allocate and free memory in this program, something which was a new concept to me since my previous background in Computer Science was in Java! When I was creating child processes in my

program, I had to create data structures for each child to store the results of its computations/finding hidden elements of an array. This required me to use malloc() to create memory dynamically. Furthermore, I had to use free(), when each child process was terminated, to free memory that I had previously allocated.

- I also learned how to create a File using a File pointer, write data to that file, and read data from a file into an Array Data Structure
- I learned how to generate random numbers using the rand() call in the C library

Assumptions Made Based on Problem Description:

- Since it says that each process has a “fixed number of processes, say X”, I assumed that each process in the process tree is only allowed to have **exactly** X Child Processes or no child processes.
- When we say that we want our program to generate no more than “Y” process, I assumed that this does not include the main program since the main process is not technically generated by our program itself.

Program Information/Overview:

- The 1st Input(argv[1]) must be equal to the number of positive integers to place in the File
- The 2nd input(argv[2]) must be equal to the number of Hidden Keys(i.e. integers whose value is -1) to place in the File
- The 3rd Input (argv[3]) must be equal to the maximum number of processes that can be in our process tree. Please note that this DOES NOT INCLUDE the main process. So, if you want your process tree corresponding to this program to have a maximum of 10 processes, set argv[3] to 9. This means that a total of 9 processes will be generated via the fork() call. The main process already exists when we execute this program!
- The 4th Input (argv[4]) must be equal to the fixed number of child processes that each process can have
- The 5th Input (argv[5]) must be equal to the number of Hidden Keys, out of the total number of Hidden Keys in the File, to identify. Hence, argv[5] <= argv[2]
- Predefined Tree Structures that I used in this Program:
 - Heap Data Structure: I used the Heap Data Structure as one of my tree Data Structures for this Program. The Heap Data Structure is a Complete Tree. As per its definition, every level of a Complete Tree is completely filled except the last level. Furthermore, at the last level of the Complete Tree, the nodes are all as far left as possible
 - Left-Skewed Tree: I used the Left-Skewed Tree as one of my data structures for this Program. Essentially, a Left-Skewed Tree is a Tree where the root Node has X Children, the leftmost child of the root node

will have X children, the leftmost child of the leftmost child of the root node will have X Children and so on.

- Right-Skewed Tree: I used the Right-Skewed Tree as one of my data structures for this Program. Essentially, a Right-Skewed Tree is a Tree where the root Node has X Children, the rightmost child of the root node will have X children, the rightmost child of the rightmost child of the root node will have X Children and so on.

Program Analysis/Design Decisions for Tree Data Structures:

To analyze this program, I will break it down into 3 cases and analyze how I constructed the program in each case.

Case 1: Heap Data Structure

The first step I took was to generate a Heap Data Structure. Since a Heap Data Structure is composed of nodes, I used a structure in C to represent a node. We know that each Node in the heap has X children. If the Maximum Number of Processes we can create is less than X, then we cannot even create X children for our root node. Hence, our heap is just 1 node which is the root node.

However, if the maximum number of processes(let's call this Y) we can create is greater than or equal to X, since each Node has **exactly** X Children or no children, then the number of Processes we can create in our Heap Data Structure is equal to $Y - Y \bmod X$. The reason for this is that we originally have our root node. At every step, we add X children to a given node in the heap. Hence, at every step, the heap grows by a size of X. So, we can see that the number of processes we are adding to our heap is equal to $0 \bmod X$. Since $Y = Y \bmod X$, we know that $Y - Y \bmod X = 0 \bmod X$. Hence, our goal should be to add $Y - Y \bmod X$ processes to our Heap Data Structure.

Once we have created the data structure, the next step is to partition our array and assign a portion of the original array to each node in the data structure. An important thing to note here is that each node in the data structure represents an individual process. The logic I used is as follows: If a node has 0 children, then we obviously cannot partition the subarray assigned to this node into subarrays to be processed by the children nodes. However, if a node has X children, then we can partition the subarray assigned to this node into X smaller subarrays of roughly equal length. I used a very simple way to partition an subarray by just determining the length of the subarray, looping through the subarray, and then whenever I hit the point where the subarray should be partitioned, I assigned the section of the subarray to the child node. How is this point determined? If the length of the subarray assigned to a node is Z and the node has X children,

there are two cases to analyze. If $Z \geq X$, each partitioned portion of the subarray should have roughly Z/X elements. Hence, whenever I loop through the subarray, I have a counter and whenever the counter reaches Z/X , I assign the subarray to a child node, move onto the next child node, and reset the counter back to 0. However, when $Z < X$, I can only partition the subarray into a maximum of Z portions. Hence, I will do the same process as before but assign the portions of the subarray to only Z Children

Once portions of the subarray are assigned to each node and subsequent child nodes, it is time to run the algorithm. For my algorithm, I have created two separate functions. One function will perform the computations on the array to calculate the average and maximum. The other function will find the hidden elements in the array. The reason I set it up this way is because performing the computations on the array can easily be done in a parallel manner by the processes we generate. The maximum value in a given array is basically the maximum value among the maximum values in each subarray. To calculate the average, we can first calculate the sums in each of the subarrays and this can clearly be done independently. Then, we can take our results from the subarrays and put them together. We can add all the sums together to get a sum for the original array. Furthermore, we can simultaneously calculate the total number of elements in each subarray. We can accomplish this by having each process calculate the total number of elements in its respective subarray. Then, we can put these results that we obtain from our child processes together. Then, we will get the total amount of numbers in our array. Then, we divide the sum by the total amount of numbers to get the average.

However, identifying a certain number of hidden elements in the array cannot be done 100% in a parallel manner. We need one process to wait for the previous process to complete. Why is this the case? Well, if we had all processes running concurrently to identify a certain number of hidden elements in the array, it could well be the case that all these processes identify more than that number.

Consider the following example:

[-1, 2, 3, -1, 4, 5, -1, -1, 6]

This array partitioned into equal subarrays whose sizes are 3 elements would be [-1, 2, 3], [-1, 4, 5], and [-1, -1, 6]. Let's say our program goal is to only identify 2 hidden elements. Well, when running each child process concurrently, the 1st child process will work with [-1, 2, 3], the 2nd child process will work with [-1, 4, 5], and the 3rd child will work with [-1, -1, 6]. Since the requirement is to identify 2 hidden elements, the 1st child will identify 1 (i.e. The item at index 0), the 2nd child will identify 1 (i.e. The item at index 0) and the 3rd child will identify 2 (i.e. The items at indices 0 and 1). Across all our subprocesses, we have identified 4 hidden elements when the requirement was to only find 2. The solution to this is to have our child processes run in some predefined order. Hence, if we run Child Process 1 first, it will identify 1 hidden element. Then, we could run Child Process 2 to identify 1 hidden element. When it comes to

child process 3, we notice that Child Processes 1 and 2 have already identified 2 hidden elements and we don't need to identify any more hidden elements. This can be easily accomplished with pipes as I will explain later on

How I implemented the Computation Function?

To implement the Computation Function, I used IPC Pipes as a Message Passing Protocol between each process and its children.

Each Node had a total of 1 Write Pipe in order to send data to its parent node and Each Node has 1 Read Pipe **PER CHILD NODE** in order to read data from each of its children.

For every Node that had 0 children, the corresponding process would just loop through its assigned subarray, find the maximum and average, and write the values to its Write Pipe.

Obviously a Node with 0 children does not need a Read Pipe.

For every Node that had X children, the corresponding process would loop through all the child Nodes in the tree and recursively call the computation function for each of those child nodes. In order to ensure that all the children run independent of each other, I used the fork() system call to create a child process for every iteration of the for loop. Within the for loop, I used a conditional branch "if pid == 0" and within this branch, I made a recursive call to the compute function and then I wrote exit(0). Hence, the child process will call the compute function recursively and then exit. In the "pid > 0" branch, I wrote "continue". For the recursive call of the compute function, I set the write Pipe for the child process to be equal to the read Pipe corresponding to the parent process and this particular child process. This works because we expect the parent to read from the pipe and the child to write to the pipe. The parent process will just continue to the next iteration of the for loop. So, what we can observe here is, during every iteration of the for loop, exactly 1 child process is created. Hence, since there were X iterations of the for loop corresponding to the X children of the Node, there were X child processes created across all iterations of the for loop.

After the parent process exits the for loop, it will enter into another for loop and make the read() system call for each of its read Pipes. As I mentioned earlier, since the node had X children, it will have X read pipes. The read system call blocks until data is made available in the pipe or until all write file descriptors have closed. We know that the latter case won't occur since I don't close all the write file descriptors.

Once the parent process reads all the data from its read pipes, it will simply perform the computations on all the data to obtain a max and average. We know that each child process will send to the parent the maximum of its assigned subarray. Hence, the parent just needs to take these "maximum values" send to it by each child process and find the maximum of these values. Furthermore, I have programmed each child process to send to the parent process the sum and

total number of values(i.e. The “count”) in its subarray. Hence, to calculate the average, the parent process can just take the sum of all these sums, take the sum of all the “counts”, and simply divide sum / count to get the average. Once the parent process has finished its computations, it will write the data to its parent process via its write pipe.

Furthermore, I have ensured that zombie processes are taken care of in my program. After every read() system call I make from each read Pipe, I call wait(pidArray[i], NULL, 0). Hence, since there are X iterations in the loop where I am making the read() system call, there will be X iterations when I call wait(pidArray[i], NULL, 0). We know that the wait(pidArray[i], NULL, 0) system call places the parent process on the waiting queue until the child process has terminated. Hence, once the child process has finished, the parent process can read the status information of the child and the entry of the child can be removed from the process table. In my Program, when I was doing all the fork() system calls, I created a pidArray and stored all the pids of the generated child processes in the pidArray. Hence, every time I made the read system call for each read Pipe and call wait(pidArray[i], NULL, 0), the parent process will reap the respective child process.

How I implemented the Hidden Identify Function?

Our goal here is to find the first argv[5] Hidden Keys in our file. Similar to the Computation Function, Each Node had a total of 1 Write Pipe in order to send data to its parent node and Each Node has 1 Read Pipe **PER CHILD NODE** in order to read data from each of its children. For a node that had 0 children, that node would just iterate through its subarray to find the hidden keys. I kept track of the indices in the array where the Hidden Key has already been found. I did this to ensure that we don't find the same key twice! Furthermore, for each child process, the maximum number of hidden keys it would find would be equal to R where $R = \text{argv}[5] - \text{Keys Already Found}$ in the previous processes thus far. Obviously, if the subarray for the child process doesn't have $R = \text{argv}[5] - \text{Keys Already Found}$ hidden keys, then it will find less hidden keys.

In this case, since it cannot be 100% parallelizable, I used the waitpid() system call **AFTER** each fork() system call in order for me to figure out how many hidden keys each child process has identified. Then, after the waitpid() system call, I would update a variable that keeps track of the total number of hidden keys we have identified thus far. I would pass this variable into each child process so that each child process would know how many hidden keys have been identified so far and how many more keys need to be identified.

Case 2: Left-Skewed Tree Data Structure

The first step I took was to generate a Left-Skewed Tree Data Structure. Since a Left-Skewed Tree Data Structure is composed of nodes, I used a structure in C to represent a node. We know that each Node in the Left-Skewed Tree has X children. If the Maximum Number of Processes

we can create is less than X , then we cannot even create X children for our Left-Skewed Tree node. Hence, our Left-Skewed Tree is just 1 node which is the root node.

However, if the maximum number of processes(let's call this Y) we can create is greater than or equal to X , since each Node has **exactly** X Children or no children, then the number of Processes we can create in our Left-Skewed Tree Data Structure is equal to $Y - Y \bmod X$. The reason for this is that we originally have our root node. At every step, we add X children to a given node in the Left-Skewed Tree. Hence, at every step, the Left-Skewed Tree grows by a size of X . So, we can see that the number of processes we are adding to our Left-Skewed Tree is equal to $0 \bmod X$. Since $Y = Y \bmod X$, we know that $Y - Y \bmod X = 0 \bmod X$. Hence, our goal should be to add $Y - Y \bmod X$ processes to our Left-Skewed Tree Data Structure. To make it easier, I defined a recursive function that added X children to the tree a total of Y / X times where Y / X is integer division. During each recursive call, I'd add X children to a node, then I'd access the first child of the node, and recursively add X children to that first child.

Once we have created the data structure, the next step is to partition our array and assign a portion of the original array to each node in the data structure. An important thing to note here is that each node in the data structure represents an individual process. The logic I used is as follows: If a node has 0 children, then we obviously cannot partition the subarray assigned to this node into subarrays to be processed by the children nodes. However, if a node has X children, then we can partition the subarray assigned to this node into X smaller subarrays of roughly equal length. I used a very simple way to partition an subarray by just determining the length of the subarray, looping through the subarray, and then whenever I hit the point where the subarray should be partitioned, I assigned the section of the subarray to the child node. How is this point determined? If the length of the subarray assigned to a node is Z and the node has X children, there are two cases to analyze. If $Z \geq X$, each partitioned portion of the subarray should have roughly Z/X elements. Hence, whenever I loop through the subarray, I have a counter and whenever the counter reaches Z/X , I assign the subarray to a child node, move onto the next child node, and reset the counter back to 0. However, when $Z < X$, I can only partition the subarray into a maximum of Z portions. Hence, I will do the same process as before but assign the portions of the subarray to only Z Children

Once portions of the subarray are assigned to each node and subsequent child nodes, it is time to run the algorithm. For my algorithm, I have created two separate functions. One function will perform the computations on the array to calculate the average and maximum. The other function will find the hidden elements in the array. The reason I set it up this way is because performing the computations on the array can easily be done in a parallel manner by the processes we generate. The maximum value in a given array is basically the maximum value among the maximum values in each subarray. To calculate the average, we can first calculate the sums in each of the subarrays and this can clearly be done independently. Then, we can take our

results from the subarrays and put them together. We can add all the sums together to get a sum for the original array. Furthermore, we can simultaneously calculate the total number of elements in each subarray. We can accomplish this by having each process calculate the total number of elements in its respective subarray. Then, we can put these results that we obtain from our child processes together. Then, we will get the total amount of numbers in our array. Then, we divide the sum by the total amount of numbers to get the average.

However, identifying a certain number of hidden elements in the array cannot be done 100% in a parallel manner. We need one process to wait for the previous process to complete. Why is this the case? Well, if we had all processes running concurrently to identify a certain number of hidden elements in the array, it could well be the case that all these processes identify more than that number.

Consider the following example:

[-1, 2, 3, -1, 4, 5, -1, -1, 6]

This array partitioned into equal subarrays whose sizes are 3 elements would be [-1, 2, 3], [-1, 4, 5], and [-1, -1, 6]. Let's say our program goal is to only identify 2 hidden elements. Well, when running each child process concurrently, the 1st child process will work with [-1, 2, 3], the 2nd child process will work with [-1, 4, 5], and the 3rd child will work with [-1, -1, 6]. Since the requirement is to identify 2 hidden elements, the 1st child will identify 1 (i.e. The item at index 0), the 2nd child will identify 1 (i.e. The item at index 0) and the 3rd child will identify 2 (i.e. The items at indices 0 and 1). Across all our subprocesses, we have identified 4 hidden elements when the requirement was to only find 2. The solution to this is to have our child processes run in some predefined order. Hence, if we run Child Process 1 first, it will identify 1 hidden element. Then, we could run Child Process 2 to identify 1 hidden element. When it comes to child process 3, we notice that Child Processes 1 and 2 have already identified 2 hidden elements and we don't need to identify any more hidden elements. This can be easily accomplished with pipes as I will explain later on

How I implemented the Computation Function?

To implement the Computation Function, I used IPC Pipes as a Message Passing Protocol between each process and its children.

Each Node had a total of 1 Write Pipe in order to send data to its parent node and Each Node has 1 Read Pipe **PER CHILD NODE** in order to read data from each of its children.

For every Node that had 0 children, the corresponding process would just loop through its assigned subarray, find the maximum and average, and write the values to its Write Pipe.

Obviously a Node with 0 children does not need a Read Pipe.

For every Node that had X children, the corresponding process would loop through all the child Nodes in the tree and recursively call the computation function for each of those child nodes. In order to ensure that all the children run independent of each other, I used the `fork()` system call to create a child process for every iteration of the for loop. Within the for loop, I used a conditional branch “if `pid == 0`” and within this branch, I made a recursive call to the compute function and then I wrote `exit(0)`. Hence, the child process will call the compute function recursively and then exit. In the “`pid > 0`” branch, I wrote “continue”. For the recursive call of the compute function, I set the write Pipe for the child process to be equal to the read Pipe corresponding to the parent process and this particular child process. This works because we expect the parent to read from the pipe and the child to write to the pipe. The parent process will just continue to the next iteration of the for loop. So, what we can observe here is, during every iteration of the for loop, exactly 1 child process is created. Hence, since there were X iterations of the for loop corresponding to the X children of the Node, there were X child processes created across all iterations of the for loop.

After the parent process exits the for loop, it will enter into another for loop and make the `read()` system call for each of its read Pipes. As I mentioned earlier, since the node had X children, it will have X read pipes. The read system call blocks until data is made available in the pipe or until all write file descriptors have closed. We know that the latter case won't occur since I don't close all the write file descriptors.

Once the parent process reads all the data from its read pipes, it will simply perform the computations on all the data to obtain a max and average. We know that each child process will send to the parent the maximum of its assigned subarray. Hence, the parent just needs to take these “maximum values” send to it by each child process and find the maximum of these values. Furthermore, I have programmed each child process to send to the parent process the sum and total number of values(i.e. The “count”) in its subarray. Hence, to calculate the average, the parent process can just take the sum of all these sums, take the sum of all the “counts”, and simply divide sum / count to get the average. Once the parent process has finished its computations, it will write the data to its parent process via its write pipe.

Furthermore, I have ensured that zombie processes are taken care of in my program. After every `read()` system call I make from each read Pipe, I call `wait(pidArray[i], NULL, 0)`. Hence, since there are X iterations in the loop where I am making the `read()` system call, there will be X iterations when I call `wait(pidArray[i], NULL, 0)`. We know that the `wait(pidArray[i], NULL, 0)` system call places the parent process on the waiting queue until the child process has terminated. Hence, once the child process has finished, the parent process can read the status information of the child and the entry of the child can be removed from the process table. In my Program, when I was doing all the `fork()` system calls, I created a `pidArray` and stored all the pids of the generated child processes in the `pidArray`. Hence, every time I made the read system call for each

read Pipe and call `wait(pidArray[i], NULL, 0)`, the parent process will reap the respective child process.

How I implemented the Hidden Identify Function?

Our goal here is to find the first `argv[5]` Hidden Keys in our file. Similar to the Computation Function, Each Node had a total of 1 Write Pipe in order to send data to its parent node and Each Node has 1 Read Pipe **PER CHILD NODE** in order to read data from each of its children. For a node that had 0 children, that node would just iterate through its subarray to find the hidden keys. I kept track of the indices in the array where the Hidden Key has already been found. I did this to ensure that we don't find the same key twice! Furthermore, for each child process, the maximum number of hidden keys it would find would be equal to R where $R = \text{argv}[5] - \text{Keys Already Found}$ in the previous processes thus far. Obviously, if the subarray for the child process doesn't have $R = \text{argv}[5] - \text{Keys Already Found}$ hidden keys, then it will find less hidden keys.

In this case, since it cannot be 100% parallelizable, I used the `waitpid()` system call **AFTER** each `fork()` system call in order for me to figure out how many hidden keys each child process has identified. Then, after the `waitpid()` system call, I would update a variable that keeps track of the total number of hidden keys we have identified thus far. I would pass this variable into each child process so that each child process would know how many hidden keys have been identified so far and how many more keys need to be identified.

Case 3: Right-Skewed Tree Data Structure

The first step I took was to generate a Right-Skewed Tree Data Structure. Since a Right-Skewed Tree Data Structure is composed of nodes, I used a structure in C to represent a node. We know that each Node in the Right-Skewed Tree Data Structure has X children. If the Maximum Number of Processes we can create is less than X , then we cannot even create X children for our Right-Skewed Tree node. Hence, our Right-Skewed Tree is just 1 node which is the root node.

However, if the maximum number of processes(let's call this Y) we can create is greater than or equal to X , since each Node has **exactly** X Children or no children, then the number of Processes we can create in our Right-Skewed Tree Data Structure is equal to $Y - Y \bmod X$. The reason for this is that we originally have our root node. At every step, we add X children to a given node in the Right-Skewed Tree. Hence, at every step, the Right-Skewed Tree grows by a size of X . So, we can see that the number of processes we are adding to our Right-Skewed Tree is equal to $0 \bmod X$. Since $Y = Y \bmod X$, we know that $Y - Y \bmod X = 0 \bmod X$. Hence, our goal should be to add $Y - Y \bmod X$ processes to our Right-Skewed Tree Data Structure. To make it easier, I defined a recursive function that added X children to the tree a total of Y / X times

where Y / X is integer division. During each recursive call, I'd add X children to a node, then I'd access the right-most child of the node, and recursively add X children to that right-most child.

Once we have created the data structure, the next step is to partition our array and assign a portion of the original array to each node in the data structure. An important thing to note here is that each node in the data structure represents an individual process. The logic I used is as follows: If a node has 0 children, then we obviously cannot partition the subarray assigned to this node into subarrays to be processed by the children nodes. However, if a node has X children, then we can partition the subarray assigned to this node into X smaller subarrays of roughly equal length. I used a very simple way to partition an subarray by just determining the length of the subarray, looping through the subarray, and then whenever I hit the point where the subarray should be partitioned, I assigned the section of the subarray to the child node. How is this point determined? If the length of the subarray assigned to a node is Z and the node has X children, there are two cases to analyze. If $Z \geq X$, each partitioned portion of the subarray should have roughly Z/X elements. Hence, whenever I loop through the subarray, I have a counter and whenever the counter reaches Z/X , I assign the subarray to a child node, move onto the next child node, and reset the counter back to 0. However, when $Z < X$, I can only partition the subarray into a maximum of Z portions. Hence, I will do the same process as before but assign the portions of the subarray to only Z Children

Once portions of the subarray are assigned to each node and subsequent child nodes, it is time to run the algorithm. For my algorithm, I have created two separate functions. One function will perform the computations on the array to calculate the average and maximum. The other function will find the hidden elements in the array. The reason I set it up this way is because performing the computations on the array can easily be done in a parallel manner by the processes we generate. The maximum value in a given array is basically the maximum value among the maximum values in each subarray. To calculate the average, we can first calculate the sums in each of the subarrays and this can clearly be done independently. Then, we can take our results from the subarrays and put them together. We can add all the sums together to get a sum for the original array. Furthermore, we can simultaneously calculate the total number of elements in each subarray. We can accomplish this by having each process calculate the total number of elements in its respective subarray. Then, we can put these results that we obtain from our child processes together. Then, we will get the total amount of numbers in our array. Then, we divide the sum by the total amount of numbers to get the average.

However, identifying a certain number of hidden elements in the array cannot be done 100% in a parallel manner. We need one process to wait for the previous process to complete. Why is this the case? Well, if we had all processes running concurrently to identify a certain number of hidden elements in the array, it could well be the case that all these processes identify more than that number.

Consider the following example:

[-1, 2, 3, -1, 4, 5, -1, -1, 6]

This array partitioned into equal subarrays whose sizes are 3 elements would be [-1, 2, 3], [-1, 4, 5], and [-1, -1, 6]. Let's say our program goal is to only identify 2 hidden elements. Well, when running each child process concurrently, the 1st child process will work with [-1, 2, 3], the 2nd child process will work with [-1, 4, 5], and the 3rd child will work with [-1, -1, 6]. Since the requirement is to identify 2 hidden elements, the 1st child will identify 1 (i.e. The item at index 0), the 2nd child will identify 1 (i.e. The item at index 0) and the 3rd child will identify 2 (i.e. The items at indices 0 and 1). Across all our subprocesses, we have identified 4 hidden elements when the requirement was to only find 2. The solution to this is to have our child processes run in some predefined order. Hence, if we run Child Process 1 first, it will identify 1 hidden element. Then, we could run Child Process 2 to identify 1 hidden element. When it comes to child process 3, we notice that Child Processes 1 and 2 have already identified 2 hidden elements and we don't need to identify any more hidden elements. This can be easily accomplished with pipes as I will explain later on

How I implemented the Computation Function?

To implement the Computation Function, I used IPC Pipes as a Message Passing Protocol between each process and its children.

Each Node had a total of 1 Write Pipe in order to send data to its parent node and Each Node has 1 Read Pipe **PER CHILD NODE** in order to read data from each of its children.

For every Node that had 0 children, the corresponding process would just loop through its assigned subarray, find the maximum and average, and write the values to its Write Pipe.

Obviously a Node with 0 children does not need a Read Pipe.

For every Node that had X children, the corresponding process would loop through all the child Nodes in the tree and recursively call the computation function for each of those child nodes. In order to ensure that all the children run independent of each other, I used the fork() system call to create a child process for every iteration of the for loop. Within the for loop, I used a conditional branch "if pid == 0" and within this branch, I made a recursive call to the compute function and then I wrote exit(0). Hence, the child process will call the compute function recursively and then exit. In the "pid > 0" branch, I wrote "continue". For the recursive call of the compute function, I set the write Pipe for the child process to be equal to the read Pipe corresponding to the parent process and this particular child process. This works because we expect the parent to read from the pipe and the child to write to the pipe. The parent process will just continue to the next iteration of the for loop. So, what we can observe here is, during every iteration of the for loop, exactly 1 child process is created. Hence, since there were X iterations of the for loop

corresponding to the X children of the Node, there were X child processes created across all iterations of the for loop.

After the parent process exits the for loop, it will enter into another for loop and make the read() system call for each of its read Pipes. As I mentioned earlier, since the node had X children, it will have X read pipes. The read system call blocks until data is made available in the pipe or until all write file descriptors have closed. We know that the latter case won't occur since I don't close all the write file descriptors.

Once the parent process reads all the data from its read pipes, it will simply perform the computations on all the data to obtain a max and average. We know that each child process will send to the parent the maximum of its assigned subarray. Hence, the parent just needs to take these "maximum values" send to it by each child process and find the maximum of these values. Furthermore, I have programmed each child process to send to the parent process the sum and total number of values(i.e. The "count") in its subarray. Hence, to calculate the average, the parent process can just take the sum of all these sums, take the sum of all the "counts", and simply divide sum / count to get the average. Once the parent process has finished its computations, it will write the data to its parent process via its write pipe.

Furthermore, I have ensured that zombie processes are taken care of in my program. After every read() system call I make from each read Pipe, I call wait(pidArray[i], NULL, 0). Hence, since there are X iterations in the loop where I am making the read() system call, there will be X iterations when I call wait(pidArray[i], NULL, 0). We know that the wait(pidArray[i], NULL, 0) system call places the parent process on the waiting queue until the child process has terminated. Hence, once the child process has finished, the parent process can read the status information of the child and the entry of the child can be removed from the process table. In my Program, when I was doing all the fork() system calls, I created a pidArray and stored all the pids of the generated child processes in the pidArray. Hence, every time I made the read system call for each read Pipe and call wait(pidArray[i], NULL, 0), the parent process will reap the respective child process.

How I implemented the Hidden Identify Function?

Our goal here is to find the first argv[5] Hidden Keys in our file. Similar to the Computation Function, Each Node had a total of 1 Write Pipe in order to send data to its parent node and Each Node has 1 Read Pipe **PER CHILD NODE** in order to read data from each of its children. For a node that had 0 children, that node would just iterate through its subarray to find the hidden keys. I kept track of the indices in the array where the Hidden Key has already been found. I did this to ensure that we don't find the same key twice! Furthermore, for each child process, the maximum number of hidden keys it would find would be equal to R where $R = \text{argv}[5] - \text{Keys}$

Already Found in the previous processes thus far. Obviously, if the subarray for the child process doesn't have $R = \text{argv}[5] - \text{Keys Already Found hidden keys}$, then it will find less hidden keys.

In this case, since it cannot be 100% parallelizable, I used the `waitpid()` system call **AFTER** each `fork()` system call in order for me to figure out how many hidden keys each child process has identified. Then, after the `waitpid()` system call, I would update a variable that keeps track of the total number of hidden keys we have identified thus far. I would pass this variable into each child process so that each child process would know how many hidden keys have been identified so far and how many more keys need to be identified.

When `argv[5]` hidden keys have been identified, each process will just send the number of hidden keys to its parent via a write pipe and terminate. Once the parent receives this, it will know that `argv[5]` hidden keys have been identified and it terminates. This results in a cascading termination that eventually results in the main process exiting the function and printing the `argv[5]` hidden keys it has found.

Runtime Analysis Comparing How the Program Runs for Each Tree Data Structure

For this table, I kept X as a constant and set it equal to 5. I also set the number of hidden keys we want to identify to be 5

| Data Structure | List Size | # of Processes Created | Runtime |
|----------------|-----------|------------------------|----------|
| Heap | 5000 | 5 | 0.008291 |
| | | 25 | 0.010500 |
| | | 50 | 0.009766 |
| | | 125 | 0.009451 |
| | | 625 | 0.010655 |
| | 10000 | 5 | 0.014452 |
| | | 25 | 0.016778 |
| | | 50 | 0.015554 |
| | | 125 | 0.014628 |
| | | 625 | 0.017334 |

| | | | |
|--|---------|-----|----------|
| | 100000 | 5 | 0.066367 |
| | | 25 | 0.065955 |
| | | 50 | 0.064639 |
| | | 125 | 0.062226 |
| | | 625 | 0.063955 |
| | 1000000 | 5 | 0.500997 |
| | | 25 | 0.508387 |
| | | 50 | 0.509124 |
| | | 125 | 0.492326 |
| | | 625 | 0.495479 |

For this table, I kept X as a constant and set it equal to 5. I also set the number of hidden keys we want to identify to be 5

| Data Structure | List Size | # of Processes Created | Runtime |
|------------------|-----------|------------------------|----------|
| Left-Skewed Tree | 5000 | 5 | 0.008611 |
| | | 25 | 0.008554 |
| | | 50 | 0.009073 |
| | | 125 | 0.010145 |
| | | 450 | 0.009871 |
| | 10000 | 5 | 0.013453 |
| | | 25 | 0.011569 |
| | | 50 | 0.012662 |
| | | 125 | 0.015185 |
| | | 450 | 0.016142 |
| | 100000 | 5 | 0.063358 |
| | | 25 | 0.063428 |

| | | | |
|--|---------|-----|----------|
| | | 50 | 0.064923 |
| | | 125 | 0.066603 |
| | | 450 | 0.064114 |
| | 1000000 | 5 | 0.493879 |
| | | 25 | 0.490009 |
| | | 50 | 0.498224 |
| | | 125 | 0.499278 |
| | | 450 | 0.506591 |

For this table, I kept X as a constant and set it equal to 5. I also set the number of hidden keys we want to identify to be 5

| Data Structure | List Size | # of Processes Created | Runtime |
|-------------------|-----------|------------------------|----------|
| Right-Skewed Tree | 5000 | 5 | 0.008298 |
| | | 25 | 0.008894 |
| | | 50 | 0.010022 |
| | | 125 | 0.010083 |
| | | 450 | 0.009135 |
| | 10000 | 5 | 0.014392 |
| | | 25 | 0.015185 |
| | | 50 | 0.015546 |
| | | 125 | 0.016636 |
| | | 450 | 0.017552 |
| | 100000 | 5 | 0.066114 |
| | | 25 | 0.067458 |
| | | 50 | 0.066549 |

| | | | |
|--|---------|-----|----------|
| | | 125 | 0.065496 |
| | | 450 | 0.065893 |
| | 1000000 | 5 | 0.502569 |
| | | 25 | 0.497542 |
| | | 50 | 0.500642 |
| | | 125 | 0.498987 |
| | | 450 | 0.489159 |
| | | | |

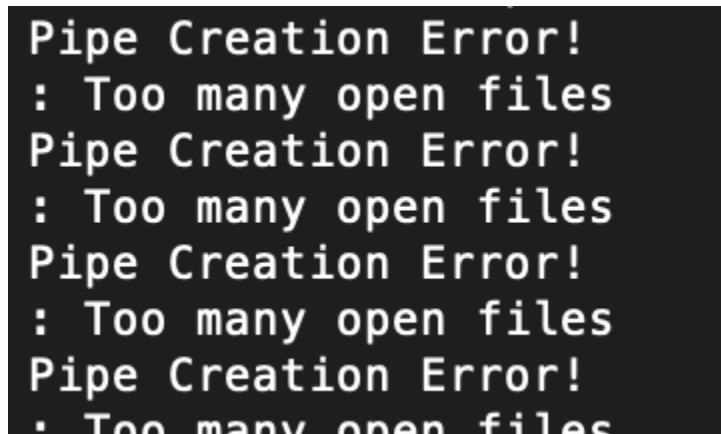
Based on the observations in the data table, it is evident that the Left-Skewed Process Tree has the shortest runtime on the Engsoft Machine(Ubuntu 20.0.4 Operating System)

Furthermore, based on my experimental data, in regards to increasing the number of processes, the trend on my Operating System seems to be that a small number of processes(i.e. Close to 5 processes) and a large number of processes(i.e. Roughly near 450-625) tend to have a larger runtime. When I use around 125 processes, my experimental data seems to suggest that the runtime has hit a minimum. My proposed explanation for this is that when we are using a smaller number of processes, each node in the process tree will be assigned a larger portion of the original array. Hence, performing the operations on that portion of the array and finding the hidden keys will obviously take more time. When we are using a larger number of processes, although each node in the process tree is assigned a relatively smaller portion of the original array, there will be a larger overhead due the fact that we will require more pipes to communicate between the processes, we will need to use the wait/waitpid system call more often for the parent processes to reap the child processes, and the Operating System will have to perform context switching more often. Performing context switching, depending on the hardware on that particular system, will certainly incur some overhead cost and this will increase the runtime.

With respect to the size of the input list, the experimental data still shows that the Left-Skewed Data Structure performs far better than its counterparts(i.e. The Heap Data Structure and the Right-Skewed Tree Data Structure). When the size of the input list increases, the Left Skewed Tree Data Structure has better runtimes.

Hence, my conclusion from this experiment is that, to optimize the runtime of the program/algorithm, we will need to use the Left-Skewed Data Structure and set the amount of processes to 125.

Furthermore, my experiments also rendered another interesting finding. For the program, I realized that there is a limit to the number of pipes I could create and the error I was receiving was “Pipe Creation Error! : Too many open files” I have included a screenshot below



This most likely occurred because, as my tree structure grew larger and larger, I needed more pipes to communicate between processes. When a tree grows larger and larger, the number of levels/layers in the tree will be more. In my program, each parent Node will need X read pipes to read data from its children node in the next level. Each child node will need 1 write pipe to write data back to the parent process in the previous level of the tree. Hence, the more levels there are in a process tree data structure, we will need more Pipes to be open in order to send data between the parent and children process. However, since the pipe buffer is of a finite size. After Linux Version 2.6.11, according to the manual, the pipe capacity is 65536 bytes. However, since my program creates a very large number of pipe as the tree structure gets larger and larger, it is evident that the 65536 bytes will be exhausted at some point. Hence, this is why I obtained the “Pipe Creation Error!” It has nothing to do with the algorithmic logic of my program.

4. In each program I have developed, I have explained how zombies are prevented. Furthermore, in each program, the possibility of orphan processes doesn't occur since I ensure that each parent executes a `waitpid()` system call at some point in its lifecycle to reap its child process. Please refer to the above explanations for more detail into how I did this.
5. Based on the observations in the data table, it is evident that the Left-Skewed Process Tree has the shortest runtime on the Engsoft Machine(Ubuntu 20.0.4 Operating System) Furthermore, based on my experimental data, in regards to increasing the number of processes, the trend on my Operating System seems to be that a small number of processes(i.e. Close to 5 processes) and a large number of processes(i.e. Roughly near 450-625) tend to have a larger runtime. When I use around 125 processes, my experimental data seems to suggest that the runtime has hit a minimum. My proposed explanation for this is that when we are using a smaller number of processes, each node

in the process tree will be assigned a larger portion of the original array. Hence, performing the operations on that portion of the array and finding the hidden keys will obviously take more time. When we are using a larger number of processes, although each node in the process tree is assigned a relatively smaller portion of the original array, there will be a larger overhead due the fact that we will require more pipes to communicate between the processes, we will need to use the wait/waitpid system call more often for the parent processes to reap the child processes, and the Operating System will have to perform context switching more often. Performing context switching, depending on the hardware on that particular system, will certainly incur some overhead cost and this will increase the runtime. With respect to the size of the input list, the experimental data still shows that the Left-Skewed Data Structure performs far better than its counterparts(i.e. The Heap Data Structure and the Right-Skewed Tree Data Structure). When the size of the input list increases, the Left Skewed Tree Data Structure has better runtimes. Hence, my conclusion from this experiment is that, to optimize the runtime of the program/algorithm, we will need to use the Left-Skewed Data Structure and set the amount of processes to 125.

6. You cannot create an arbitrary number of processes in Linux and there are limitations. There is a limit set by the kernel per user in linux. However this limit can be overridden and we can change the maximum limit for the current user by running “ulimit -u {upper limit of processes we can create}”, The initial number of processes that can be created varies by system, but if the system is 64 bit we can have an upper theoretical limit of 4194303 processes, and if the system is 32 bit then we can have an upper theoretical limit of 32767 processes. We can also check the number of processes that we can create in our current user by running “ulimit -u”.
To investigate the limit of the number of processes we can create per user, I executed the “ulimit-u” command on my Engsoft Machine. I have included my results below to provide some insight into this limit that has been set on the Engsoft Machine

```

Welcome to Ubuntu 20.04.2 LTS (GNU/Linux 5.4.0-105-generic x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:    https://landscape.canonical.com
 * Support:       https://ubuntu.com/advantage

System information as of Wed 06 Apr 2022 08:21:59 PM EDT

System load:  0.07          Processes:            1193
Usage of /:   2.1% of 457.45GB Users logged in:      3
Memory usage: 7%          IPv4 address for enp0s9:  128.6.238.3
Swap usage:   0%          IPv4 address for enp6s4f0: 192.168.2.59

96 updates can be installed immediately.
2 of these updates are security updates.
To see these additional updates run: apt list --upgradable

*** System restart required ***
Last login: Wed Apr  6 09:59:50 2022 from 68.192.77.250
[rr1133@engsoft:~$ ulimit -u
1000
rr1133@engsoft:~$ █

```

To test this theory, I created a very simple program to experimentally determine the number of processes that can be in my process tree for any arbitrary program. I have included the program below:

```

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <stdarg.h>
#include <signal.h>
#include <time.h>

int main(void){
    for(int i = 0; i < 1000; i ++){
        pid_t pid = fork();
        if(pid < 0){
            perror("Fork Failed!\n");
        }
    }
}

```

```

        else if (pid == 0){
            continue;
        }
        else if (pid > 0){
            waitpid(pid, NULL, 0);
            exit(0);
        }
    }
}

```

When I executed this program in my engsoft machine, this is what I observed

```

r113@engsoft:~/pc/Project#ls ./systemTest.out
Current PID IS: 564447
Current PID IS: 564448
Current PID IS: 564449
Current PID IS: 564450
Current PID IS: 564451
Current PID IS: 564452
Current PID IS: 564453
Current PID IS: 564454
Current PID IS: 564455
Current PID IS: 564456
Current PID IS: 564457
Current PID IS: 564458
Current PID IS: 564459
Current PID IS: 564460
Current PID IS: 564461
Current PID IS: 564462
Current PID IS: 564463
Current PID IS: 564464
Current PID IS: 564465
Current PID IS: 564466
Current PID IS: 564467

```

```

Current PID IS: 564468
Current PID IS: 564469
Current PID IS: 564470
Current PID IS: 564471
Current PID IS: 564472
Current PID IS: 564473
Current PID IS: 564474
Current PID IS: 564475
Current PID IS: 564476
Current PID IS: 564477
Current PID IS: 564478
Current PID IS: 564479
Current PID IS: 564480
Current PID IS: 564481
Current PID IS: 564482
Current PID IS: 564483
Current PID IS: 564484
Current PID IS: 564485
Current PID IS: 564486
Current PID IS: 564487
Current PID IS: 564488
Current PID IS: 564489
Current PID IS: 564490
Current PID IS: 564491
Current PID IS: 564492
Current PID IS: 564493
Current PID IS: 564494
Current PID IS: 564495
Current PID IS: 564496
Current PID IS: 564497
Current PID IS: 564498
Current PID IS: 564499
Current PID IS: 564500
Current PID IS: 564501
Current PID IS: 564502
Current PID IS: 564503
Current PID IS: 564504
Current PID IS: 564505
Current PID IS: 564506
Current PID IS: 564507
Current PID IS: 564508
Current PID IS: 564509
Current PID IS: 564510
Current PID IS: 564511
Current PID IS: 564512
Current PID IS: 564513
Current PID IS: 564514
Current PID IS: 564515
Current PID IS: 564516
Current PID IS: 564517
Current PID IS: 564518
Current PID IS: 564519
Current PID IS: 564520
Current PID IS: 564521
Current PID IS: 564522
Current PID IS: 564523
Current PID IS: 564524
Current PID IS: 564525
Current PID IS: 564526
Current PID IS: 564527
Current PID IS: 564528
Current PID IS: 564529
Current PID IS: 564530
Current PID IS: 564531
Current PID IS: 564532
Current PID IS: 564533
Current PID IS: 564534
Current PID IS: 564535
Current PID IS: 564536
Current PID IS: 564537
Current PID IS: 564538
Current PID IS: 564539
Current PID IS: 564540
Current PID IS: 564541
Current PID IS: 564542
Current PID IS: 564543
Current PID IS: 564544

```

| | | |
|---------|---------|--------|
| Current | PID 151 | 564622 |
| Current | PID 151 | 564623 |
| Current | PID 151 | 564625 |
| Current | PID 151 | 564626 |
| Current | PID 151 | 564627 |
| Current | PID 151 | 564628 |
| Current | PID 151 | 564631 |
| Current | PID 151 | 564632 |
| Current | PID 151 | 564633 |
| Current | PID 151 | 564634 |
| Current | PID 151 | 564635 |
| Current | PID 151 | 564636 |
| Current | PID 151 | 564637 |
| Current | PID 151 | 564638 |
| Current | PID 151 | 564639 |
| Current | PID 151 | 564641 |
| Current | PID 151 | 564642 |
| Current | PID 151 | 564643 |
| Current | PID 151 | 564645 |
| Current | PID 151 | 564646 |
| Current | PID 151 | 564647 |
| Current | PID 151 | 564648 |
| Current | PID 151 | 564649 |
| Current | PID 151 | 564650 |
| Current | PID 151 | 564651 |
| Current | PID 151 | 564652 |
| Current | PID 151 | 564653 |
| Current | PID 151 | 564654 |
| Current | PID 151 | 564655 |
| Current | PID 151 | 564656 |
| Current | PID 151 | 564657 |
| Current | PID 151 | 564658 |
| Current | PID 151 | 564661 |
| Current | PID 151 | 564662 |
| Current | PID 151 | 564663 |
| Current | PID 151 | 564664 |
| Current | PID 151 | 564665 |
| Current | PID 151 | 564666 |
| Current | PID 151 | 564667 |
| Current | PID 151 | 564668 |
| Current | PID 151 | 564669 |
| Current | PID 151 | 564670 |
| Current | PID 151 | 564671 |
| Current | PID 151 | 564672 |
| Current | PID 151 | 564673 |
| Current | PID 151 | 564674 |
| Current | PID 151 | 564675 |
| Current | PID 151 | 564676 |
| Current | PID 151 | 564677 |
| Current | PID 151 | 564678 |
| Current | PID 151 | 564679 |
| Current | PID 151 | 564680 |
| Current | PID 151 | 564681 |
| Current | PID 151 | 564682 |
| Current | PID 151 | 564683 |
| Current | PID 151 | 564684 |
| Current | PID 151 | 564685 |
| Current | PID 151 | 564686 |
| Current | PID 151 | 564687 |
| Current | PID 151 | 564688 |
| Current | PID 151 | 564689 |
| Current | PID 151 | 564690 |
| Current | PID 151 | 564691 |
| Current | PID 151 | 564692 |
| Current | PID 151 | 564693 |
| Current | PID 151 | 564694 |
| Current | PID 151 | 564695 |
| Current | PID 151 | 564696 |
| Current | PID 151 | 564697 |
| Current | PID 151 | 564698 |
| Current | PID 151 | 564699 |
| Current | PID 151 | 564700 |

Current PID is: 564853
Current PID is: 564854
Current PID is: 564855
Current PID is: 564856
Current PID is: 564857
Current PID is: 564858
Current PID is: 564859
Current PID is: 564860
Current PID is: 564861
Current PID is: 564862
Current PID is: 564863
Current PID is: 564864
Current PID is: 564865
Current PID is: 564866
Current PID is: 564867
Current PID is: 564868
Current PID is: 564869
Current PID is: 564870
Current PID is: 564871
Current PID is: 564872
Current PID is: 564873
Current PID is: 564874
Current PID is: 564875
Current PID is: 564876
Current PID is: 564877
Current PID is: 564878
Current PID is: 564879
Current PID is: 564880
Current PID is: 564881
Current PID is: 564882
Current PID is: 564883
Current PID is: 564884
Current PID is: 564885
Current PID is: 564886
Current PID is: 564887
Current PID is: 564888
Current PID is: 564889
Current PID is: 564890
Current PID is: 564891
Current PID is: 564892
Current PID is: 564893
Current PID is: 564894
Current PID is: 564895
Current PID is: 564896
Current PID is: 564897
Current PID is: 564898
Current PID is: 564899
Current PID is: 564900
Current PID is: 564901
Current PID is: 564902
Current PID is: 564903
Current PID is: 564904
Current PID is: 564905
Current PID is: 564906
Current PID is: 564907
Current PID is: 564908
Current PID is: 564909
Current PID is: 564910
Current PID is: 564911
Current PID is: 564912
Current PID is: 564913
Current PID is: 564914
Current PID is: 564915
Current PID is: 564916
Current PID is: 564917
Current PID is: 564918
Current PID is: 564919
Current PID is: 564920
Current PID is: 564921
Current PID is: 564922
Current PID is: 564923
Current PID is: 564924
Current PID is: 564925
Current PID is: 564926
Current PID is: 564927
Current PID is: 564928
Current PID is: 564929
Current PID is: 564930

| | | |
|---------|---------|--------|
| Current | PID 151 | 151088 |
| Current | PID 151 | 151089 |
| Current | PID 151 | 151090 |
| Current | PID 151 | 151091 |
| Current | PID 151 | 151092 |
| Current | PID 151 | 151093 |
| Current | PID 151 | 151094 |
| Current | PID 151 | 151095 |
| Current | PID 151 | 151096 |
| Current | PID 151 | 151097 |
| Current | PID 151 | 151098 |
| Current | PID 151 | 151099 |
| Current | PID 151 | 151100 |
| Current | PID 151 | 151101 |
| Current | PID 151 | 151102 |
| Current | PID 151 | 151103 |
| Current | PID 151 | 151104 |
| Current | PID 151 | 151105 |
| Current | PID 151 | 151106 |
| Current | PID 151 | 151107 |
| Current | PID 151 | 151108 |
| Current | PID 151 | 151109 |
| Current | PID 151 | 151110 |
| Current | PID 151 | 151111 |
| Current | PID 151 | 151112 |
| Current | PID 151 | 151113 |
| Current | PID 151 | 151114 |
| Current | PID 151 | 151115 |
| Current | PID 151 | 151116 |
| Current | PID 151 | 151117 |
| Current | PID 151 | 151118 |
| Current | PID 151 | 151119 |
| Current | PID 151 | 151120 |
| Current | PID 151 | 151121 |
| Current | PID 151 | 151122 |
| Current | PID 151 | 151123 |
| Current | PID 151 | 151124 |
| Current | PID 151 | 151125 |
| Current | PID 151 | 151126 |
| Current | PID 151 | 151127 |
| Current | PID 151 | 151128 |
| Current | PID 151 | 151129 |
| Current | PID 151 | 151130 |
| Current | PID 151 | 151131 |
| Current | PID 151 | 151132 |
| Current | PID 151 | 151133 |
| Current | PID 151 | 151134 |
| Current | PID 151 | 151135 |
| Current | PID 151 | 151136 |
| Current | PID 151 | 151137 |
| Current | PID 151 | 151138 |
| Current | PID 151 | 151139 |
| Current | PID 151 | 151140 |
| Current | PID 151 | 151141 |
| Current | PID 151 | 151142 |
| Current | PID 151 | 151143 |
| Current | PID 151 | 151144 |
| Current | PID 151 | 151145 |
| Current | PID 151 | 151146 |
| Current | PID 151 | 151147 |
| Current | PID 151 | 151148 |
| Current | PID 151 | 151149 |
| Current | PID 151 | 151150 |
| Current | PID 151 | 151151 |
| Current | PID 151 | 151152 |
| Current | PID 151 | 151153 |
| Current | PID 151 | 151154 |
| Current | PID 151 | 151155 |
| Current | PID 151 | 151156 |
| Current | PID 151 | 151157 |
| Current | PID 151 | 151158 |
| Current | PID 151 | 151159 |
| Current | PID 151 | 151160 |
| Current | PID 151 | 151161 |
| Current | PID 151 | 151162 |
| Current | PID 151 | 151163 |
| Current | PID 151 | 151164 |
| Current | PID 151 | 151165 |
| Current | PID 151 | 151166 |
| Current | PID 151 | 151167 |
| Current | PID 151 | 151168 |
| Current | PID 151 | 151169 |
| Current | PID 151 | 151170 |
| Current | PID 151 | 151171 |
| Current | PID 151 | 151172 |
| Current | PID 151 | 151173 |
| Current | PID 151 | 151174 |
| Current | PID 151 | 151175 |
| Current | PID 151 | 151176 |
| Current | PID 151 | 151177 |
| Current | PID 151 | 151178 |
| Current | PID 151 | 151179 |
| Current | PID 151 | 151180 |
| Current | PID 151 | 151181 |
| Current | PID 151 | 151182 |
| Current | PID 151 | 151183 |
| Current | PID 151 | 151184 |
| Current | PID 151 | 151185 |
| Current | PID 151 | 151186 |
| Current | PID 151 | 151187 |
| Current | PID 151 | 151188 |
| Current | PID 151 | 151189 |
| Current | PID 151 | 151190 |
| Current | PID 151 | 151191 |
| Current | PID 151 | 151192 |
| Current | PID 151 | 151193 |
| Current | PID 151 | 151194 |
| Current | PID 151 | 151195 |
| Current | PID 151 | 151196 |
| Current | PID 151 | 151197 |
| Current | PID 151 | 151198 |
| Current | PID 151 | 151199 |
| Current | PID 151 | 151200 |

| | | |
|---------|-----|----------|
| Current | P10 | 15S163 |
| Current | P10 | 15S163.1 |
| Current | P10 | 15S164 |
| Current | P10 | 15S164.1 |
| Current | P10 | 15S165 |
| Current | P10 | 15S165.1 |
| Current | P10 | 15S166 |
| Current | P10 | 15S166.1 |
| Current | P10 | 15S169 |
| Current | P10 | 15S169.1 |
| Current | P10 | 15S171 |
| Current | P10 | 15S171.1 |
| Current | P10 | 15S172 |
| Current | P10 | 15S172.1 |
| Current | P10 | 15S173 |
| Current | P10 | 15S173.1 |
| Current | P10 | 15S174 |
| Current | P10 | 15S174.1 |
| Current | P10 | 15S176 |
| Current | P10 | 15S176.1 |
| Current | P10 | 15S178 |
| Current | P10 | 15S178.1 |
| Current | P10 | 15S179 |
| Current | P10 | 15S179.1 |
| Current | P10 | 15S181 |
| Current | P10 | 15S181.1 |
| Current | P10 | 15S182 |
| Current | P10 | 15S182.1 |
| Current | P10 | 15S183 |
| Current | P10 | 15S183.1 |
| Current | P10 | 15S186 |
| Current | P10 | 15S186.1 |
| Current | P10 | 15S188 |
| Current | P10 | 15S188.1 |
| Current | P10 | 15S189 |
| Current | P10 | 15S189.1 |
| Current | P10 | 15S191 |
| Current | P10 | 15S191.1 |
| Current | P10 | 15S195 |
| Current | P10 | 15S195.1 |
| Current | P10 | 15S198 |
| Current | P10 | 15S198.1 |
| Current | P10 | 15S199 |
| Current | P10 | 15S199.1 |
| Current | P10 | 15S201 |
| Current | P10 | 15S201.1 |
| Current | P10 | 15S203 |
| Current | P10 | 15S203.1 |
| Current | P10 | 15S206 |
| Current | P10 | 15S206.1 |
| Current | P10 | 15S208 |
| Current | P10 | 15S208.1 |
| Current | P10 | 15S209 |
| Current | P10 | 15S209.1 |
| Current | P10 | 15S214 |
| Current | P10 | 15S214.1 |
| Current | P10 | 15S216 |
| Current | P10 | 15S216.1 |
| Current | P10 | 15S218 |
| Current | P10 | 15S218.1 |
| Current | P10 | 15S219 |
| Current | P10 | 15S219.1 |
| Current | P10 | 15S221 |
| Current | P10 | 15S221.1 |
| Current | P10 | 15S223 |
| Current | P10 | 15S223.1 |
| Current | P10 | 15S224 |
| Current | P10 | 15S224.1 |
| Current | P10 | 15S226 |
| Current | P10 | 15S226.1 |
| Current | P10 | 15S228 |
| Current | P10 | 15S228.1 |
| Current | P10 | 15S229 |
| Current | P10 | 15S229.1 |
| Current | P10 | 15S230 |
| Current | P10 | 15S230.1 |
| Current | P10 | 15S231 |
| Current | P10 | 15S231.1 |
| Current | P10 | 15S233 |
| Current | P10 | 15S233.1 |
| Current | P10 | 15S234 |
| Current | P10 | 15S234.1 |
| Current | P10 | 15S236 |
| Current | P10 | 15S236.1 |
| Current | P10 | 15S237 |
| Current | P10 | 15S237.1 |
| Current | P10 | 15S238 |
| Current | P10 | 15S238.1 |
| Current | P10 | 15S239 |
| Current | P10 | 15S239.1 |
| Current | P10 | 15S240 |
| Current | P10 | 15S240.1 |

| | |
|---------|----------------|
| Current | PID 1S: 463118 |
| Current | PID 1S: 463119 |
| Current | PID 1S: 463120 |
| Current | PID 1S: 463121 |
| Current | PID 1S: 463122 |
| Current | PID 1S: 463123 |
| Current | PID 1S: 463124 |
| Current | PID 1S: 463125 |
| Current | PID 1S: 463126 |
| Current | PID 1S: 463127 |
| Current | PID 1S: 463128 |
| Current | PID 1S: 463129 |
| Current | PID 1S: 463130 |
| Current | PID 1S: 463131 |
| Current | PID 1S: 463132 |
| Current | PID 1S: 463133 |
| Current | PID 1S: 463134 |
| Current | PID 1S: 463135 |
| Current | PID 1S: 463136 |
| Current | PID 1S: 463137 |
| Current | PID 1S: 463138 |
| Current | PID 1S: 463139 |
| Current | PID 1S: 463140 |
| Current | PID 1S: 463141 |
| Current | PID 1S: 463142 |
| Current | PID 1S: 463143 |
| Current | PID 1S: 463144 |
| Current | PID 1S: 463145 |
| Current | PID 1S: 463146 |
| Current | PID 1S: 463147 |
| Current | PID 1S: 463148 |
| Current | PID 1S: 463149 |
| Current | PID 1S: 463150 |
| Current | PID 1S: 463151 |
| Current | PID 1S: 463152 |
| Current | PID 1S: 463153 |
| Current | PID 1S: 463154 |
| Current | PID 1S: 463155 |
| Current | PID 1S: 463156 |
| Current | PID 1S: 463157 |
| Current | PID 1S: 463158 |
| Current | PID 1S: 463159 |
| Current | PID 1S: 463160 |
| Current | PID 1S: 463161 |
| Current | PID 1S: 463162 |
| Current | PID 1S: 463163 |
| Current | PID 1S: 463164 |
| Current | PID 1S: 463165 |
| Current | PID 1S: 463166 |
| Current | PID 1S: 463167 |
| Current | PID 1S: 463168 |
| Current | PID 1S: 463169 |
| Current | PID 1S: 463170 |
| Current | PID 1S: 463171 |
| Current | PID 1S: 463172 |
| Current | PID 1S: 463173 |
| Current | PID 1S: 463174 |
| Current | PID 1S: 463175 |
| Current | PID 1S: 463176 |
| Current | PID 1S: 463177 |
| Current | PID 1S: 463178 |
| Current | PID 1S: 463179 |
| Current | PID 1S: 463180 |
| Current | PID 1S: 463181 |
| Current | PID 1S: 463182 |
| Current | PID 1S: 463183 |
| Current | PID 1S: 463184 |
| Current | PID 1S: 463185 |
| Current | PID 1S: 463186 |
| Current | PID 1S: 463187 |
| Current | PID 1S: 463188 |
| Current | PID 1S: 463189 |
| Current | PID 1S: 463190 |
| Current | PID 1S: 463191 |
| Current | PID 1S: 463192 |
| Current | PID 1S: 463193 |
| Current | PID 1S: 463194 |

```

Current PID IS: 565396
Current PID IS: 565397
Current PID IS: 565398
Current PID IS: 565399
Current PID IS: 565400
Current PID IS: 565401
Current PID IS: 565402
Current PID IS: 565403
Current PID IS: 565404
Current PID IS: 565405
Current PID IS: 565406
Current PID IS: 565407
Current PID IS: 565408
Current PID IS: 565409
Current PID IS: 565410
Current PID IS: 565411
Current PID IS: 565412
Current PID IS: 565413
Current PID IS: 565414
Current PID IS: 565415
Current PID IS: 565416
Current PID IS: 565417
Current PID IS: 565418
Current PID IS: 565419
Current PID IS: 565420
Current PID IS: 565421
Current PID IS: 565422
Current PID IS: 565423
Current PID IS: 565424
Current PID IS: 565425
Current PID IS: 565426
Current PID IS: 565427
Current PID IS: 565428
Current PID IS: 565429
Current PID IS: 565430
Current PID IS: 565431
Current PID IS: 565432
Current PID IS: 565433
Current PID IS: 565434
Current PID IS: 565435
Current PID IS: 565436
Current PID IS: 565437
Current PID IS: 565438
Current PID IS: 565439
Current PID IS: 565440
Current PID IS: 565441
Current PID IS: 565442
Current PID IS: 565443
Current PID IS: 565444
Current PID IS: 565445
Current PID IS: 565446
Current PID IS: 565447
Current PID IS: 565448
Current PID IS: 565449
Fork Failed!
! Resource temporarily unavailable
Current PID IS: 565449
Fork Failed!
! Resource temporarily unavailable
Current PID IS: 565449
Fork Failed!
! Resource temporarily unavailable
rr133@engsoft:~/pc/Project#16

```

```

Fork Failed!
: Resource temporarily unavailable
Current PID IS: 565449
Fork Failed!
: Resource temporarily unavailable
Current PID IS: 565449
Fork Failed!
: Resource temporarily unavailable

```

Conclusions that I Drew From the Output Listed Above:

- The pid of the main process was 564447.
- The pid of the last process generated has a pid of 565449
- The terminal output shows that every child process had a pid that was 1 greater than its parent process
- Hence, I can conclude that the number of processes generated was $565449 - 564447 = 1002$ which is roughly close to the 1000 value given to us by “ulimit -u”

To further test my hypothesis, I ran my program to generate 997 processes by making the following modification to the test program;

```

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>

```

```
#include <sys/wait.h>
#include <stdarg.h>
#include <signal.h>
#include <time.h>
int main(void){
    for(int i = 0; i < 997; i ++){
        pid_t pid = fork();
        if(pid < 0){
            perror("Fork Failed!\n");
        }
        else if (pid == 0){
            continue;
        }
        else if (pid > 0){
            waitpid(pid, NULL, 0);
            exit(0);
        }
    }
}
```

```
Current PID is: 568604
Current PID is: 568605
Current PID is: 568606
Current PID is: 568607
Current PID is: 568608
Current PID is: 568609
Current PID is: 568610
Current PID is: 568611
Current PID is: 568612
Current PID is: 568613
Current PID is: 568614
Current PID is: 568615
Current PID is: 568616
Current PID is: 568617
Current PID is: 568618
Current PID is: 568619
Current PID is: 568620
Current PID is: 568621
Current PID is: 568622
Current PID is: 568623
Current PID is: 568624
Current PID is: 568625
Current PID is: 568626
Current PID is: 568627
Current PID is: 568628
Current PID is: 568629
Current PID is: 568630
Current PID is: 568631
Current PID is: 568632
Current PID is: 568633
Current PID is: 568634
Current PID is: 568635
Current PID is: 568636
Current PID is: 568637
Current PID is: 568638
Current PID is: 568639
Current PID is: 568640
Current PID is: 568641
Current PID is: 568642
Current PID is: 568643
Current PID is: 568644
Current PID is: 568645
Current PID is: 568646
Current PID is: 568647
Current PID is: 568648
Current PID is: 568649
Current PID is: 568650
Current PID is: 568651
Current PID is: 568652
Current PID is: 568653
Current PID is: 568654
Current PID is: 568655
Current PID is: 568656
Current PID is: 568657
Current PID is: 568658
Current PID is: 568659
Current PID is: 568660
Current PID is: 568661
Current PID is: 568662
Current PID is: 568663
Current PID is: 568664
Current PID is: 568665
Current PID is: 568666
Current PID is: 568667
Current PID is: 568668
Current PID is: 568669
Current PID is: 568670
Current PID is: 568671
Current PID is: 568672
Current PID is: 568673
Current PID is: 568674
Current PID is: 568675
Current PID is: 568676
Current PID is: 568677
Current PID is: 568678
Current PID is: 568679
Current PID is: 568680
Current PID is: 568681
rr1133@engsoft:~/pc/Project#1$
```

I have included the end of my terminal output just to prove that there was no error with the fork(). Hence, I was able to create 997 processes without any error!

Hence, it is clear that for any program I run on my engsoft machine, the maximum number of processes generated in the process tree corresponding to this program is roughly close to 1000.

7. To first answer this question, I first did some experimentation to determine the theoretical limit regarding the highest possible Process Identifier of a Process that can run on an Operating System just out of curiosity. I was using the Ubuntu 20.0.4(i.e. Linux Distribution) Operating System since this is what Engsoft machines use. I ran the following two commands on the terminal: “cd /proc/sys/kernel” followed by “cat pid_max”. My results are shown below in a screenshot

```
[rr1133@engsoft:~$ cd /proc/sys/kernel  
[rr1133@engsoft:/proc/sys/kernel$ cat pid_max  
4194304  
rr1133@engsoft:/proc/sys/kernel$ █
```

This shows that the maximum value of the pid of a process can be 4194303. Once we have used up the pid of 4194303, the next process that is created on our Operating System will NOT receive a pid of 4194304. Instead, what the Operating System will do, is that it will wrap back around to 1 and will start looking for an unused pid to assign to the process.

What is the maximum number of processes you can create before your system crashes?

- Although the maximum value of the pid of a process can certainly be 4194303, our Operating System, as shown by my screenshots in Question 6, will only allow us to create 1000 processes when we are running a program and once we exceed this limit, the fork() system call will fail.

Variation Problem:

- Code on Github

What I Learned From Problem:

- How to create a Process Tree and use IPC Pipes to pass information between Nodes in the Process Tree. I had to use multiple pipes in my program to send data between parent process and child process and vice versa. In lecture we learned that IPC pipes are unidirectional and half duplex(i.e. Data can travel one way at a time), I had to make two pipes for each process. One pipe would enable a process to retrieve data from its child process and the second pipe would enable a process to send data to its parent process.
- How to ensure that child processes are terminated accordingly and are not remaining in the system as zombies as I am performing a Depth First Search Traversal of the process tree. To do this I had to use the kill() system call in conjunction with the waitpid() system call. After receiving data from its child process, I had the process send a SIGKILL signal to the child process via kill(). Since SIGKILL cannot be caught or handled, the child process will be killed.

Furthermore, I used the waitpid() system call to enable the child process to be reaped(i.e. The process id of the child process is removed from the process table)

- How to create processes using the fork() system call
- How to create various data structures to represent process trees and analyze their effectiveness relative to each other
- I also learned how to dynamically allocate and free memory in this program, something which was a new concept to me since my previous background in Computer Science was in Java! When I was creating child processes in my program, I had to create data structures for each child to store the results of its computations/finding hidden elements of an array. This required me to use malloc() to create memory dynamically. Furthermore, I had to use free(), when each child process was terminated, to free memory that I had previously allocated.
- I also learned how to create a File using a File pointer, write data to that file, and read data from a file into an Array Data Structure
- I learned how to generate random numbers using the rand() call in the C library

Assumptions Made Based on Problem Description:

- Since it says that each process has a “fixed number of processes, say X”, I assumed that each process in the process tree is only allowed to have **exactly** X Child Processes or no child processes.
- When we say that we want our program to generate no more than “Y” process, I assumed that this does not include the main program since the main process is not technically generated by our program itself.

Program Inputs:

- The 1st Input(argv[1]) must be equal to the number of positive integers to place in the File
- The 2nd input(argv[2]) must be equal to the number of Hidden Keys(i.e. integers whose value is -1) to place in the File
- The 3rd Input (argv[3]) must be equal to the maximum number of processes that can be in our process tree. Please note that this INCLUDES the main process. So, if you want your process tree corresponding to this program to have a maximum of 10 processes, set argv[3] to 10. This means that a total of 9 processes will be generated via the fork() call. The main process already exists when we execute this program!
- The 4th Input (argv[4]) must be equal to the fixed number of child processes that each process can have

Design Description

In Part 3 of this Problem 1, I mentioned how I had created a separate function for identifying the Hidden Keys in the text file. Since I had already created a separate function for performing this task, I simply just reused that function for this problem.

How I implemented the Hidden Identify Function?

Our goal here is to find the first 3 Hidden Keys in our file. Similar to the Computation Function, Each Node had a total of 1 Write Pipe in order to send data to its parent node and Each Node has 1 Read Pipe **PER CHILD NODE** in order to read data from each of its children.

For a node that had 0 children, that node would just iterate through its subarray to find the hidden keys. I kept track of the indices in the array where the Hidden Key has already been found. I did this to ensure that we don't find the same key twice! Furthermore, for each child process, the maximum number of hidden keys it would find would be equal to R where $R = 3 - \text{Keys Already Found}$ in the previous processes thus far. Obviously, if the subarray for the child process doesn't have $R = 3 - \text{Keys Already Found}$ hidden keys, then it will find less hidden keys.

In this case, since it cannot be 100% parallelizable, I used the `waitpid()` system call **AFTER** each `fork()` system call in order for me to figure out how many hidden keys each child process has identified. Then, after the `waitpid()` system call, I would update a variable that keeps track of the total number of hidden keys we have identified thus far. I would pass this variable into each child process so that each child process would know how many hidden keys have been identified so far and how many more keys need to be identified.

When 3 hidden keys have been identified, each process will just send the number of hidden keys to its parent via a write pipe and terminate. Once the parent receives this, it will know that 3 hidden keys have been identified and it terminates. This results in a cascading termination that eventually results in the main process exiting the function and printing the 3 hidden keys it has found.

The program is able to find the **first 3** hidden keys because of the way the array is partitioned among the child processes. For any given process, if it has X child processes, it will take the subarray it has been assigned and assign equal segments of the subarray to the child processes from left to right. Let's define the variable L to represent the length of each segment that is assigned to a child process and let $A[\text{left: right}]$ represent the subarray that the parent process has been assigned. What this means is that the first child process will get $A[\text{left: left} + L]$, the second child process will get $A[\text{left} + L: \text{left} + 2L]$ and so on. My algorithm logic uses a for loop and creates the first child process before the 2nd child process is created. Likewise, the 2nd child process is created before the 3rd child process is created. This means that the child process that is assigned the "leftmost" segment of the subarray $A[\text{left: right}]$ will always be executed first. We can use an inductive argument to show that this will apply to the entire tree. For any subarray

assigned to a process, it will create a child process to analyze the leftmost segment of the subarray first.

Problem 2

Assumptions Made When Writing Program:

- When it says to generate the process tree in Scheme 1, in order to obtain the exit information from Process A(i.e. The root node of the Tree), I needed to have another Process which served as the Main Process and as the Parent of Process A. This Main Process really didn't do anything special when I developed the program. I just used it as a way to obtain the status information for Process A and print out its termination information.
- For this problem, I assumed that Process A creates Process B before it creates Process C.

Design Process

- Design Decision was to simply create a predefined Process Tree with the 4 Nodes shown in the Scheme 1. Then, using the Depth First Search Algorithm, I traversed the Tree and created a process every time I "visited" a Node during my Depth First Search Traversal. Then, if my node had child nodes, I would implement a for loop, use a fork() system call to create a child process, put the parent node in waiting queue using waitpid() until child process is finished executing, and proceed to "visit" the child node(i.e. When we visit the child node, the child process will be running)
- waitpid() assures that when all the children processes are done and the Depth First Search is coming back up the tree, the parent process will reap each child process(i.e. The pid of the child process will be removed from the process table). Since I am using a loop to execute one child at a time and then have the parent reap the child when it's done, this is very similar to Depth First Search Traversal of a Tree.

What I Learned From This Program

- How to create a Process Tree
- How to use the Depth First Search Algorithm to traverse a Process Tree, create Processes as I am traversing the Tree, and ensure that all child processes don't become zombies and are reaped by the parent process using the waitpid() system call.
- What happens when we issue a terminating signal to a process in a process tree given a particular state of the process tree(i.e. Which process are running, which processes are waiting, etc)

****Code on Github, Also Attached as C File to Submission Along with MakeFile**

1. When the kill() system call(i.e. Kill -9 <pid>) is used to send a terminating signal to Process A, Process A will be terminated. The thing to analyze here is what happens to the remaining processes in our process tree(i.e. Process B, Process C, and Process D).

Case 1: We send the terminating signal to Process A before Process A has even created any children

This is a rather trivial case to analyze because Process A hasn't even created any child processes yet. Hence, at the time we sent the terminating signal to Process A, Process A and the main process(i.e. The parent process of Process A) were the only processes of our program that were running on our system. Hence, when we terminate process A, we can notice that our main program has a `waitpid()` system call. Since the Flags used here are `WCONTINUED` and `WUNTRACED`, the `waitpid()` system call will be blocking. Hence, once Process A terminates, the main process will reap Process A from the process table(i.e. the entry of Process A will be removed from the process table).

Case 2: We send the terminating signal to Process A after Process A has created child Process B but before child Process B has created Process D and before Process A has created Process C

In this case, Process A has already had the opportunity to create a child process B. However, we have sent a terminating signal to Process A. Hence, when we terminate process A, we can notice that our main program has a `waitpid()` system call. Since the Flags used here are `WCONTINUED` and `WUNTRACED`, the `waitpid()` system call will be blocking. Hence, once Process A terminates, the main process will reap Process A from the process table(i.e. the entry of Process A will be removed from the process table). obviously the main process(i.e. The parent process of Process A) is running on our system. However, since Process B is still running in our system, it will now be an orphan process and will be adopted by the init process. The init process always runs on the `wait()` system call so, when Process B terminates, it will be reaped by init. Since Process B is running in our system, it will create a child Process(i.e. Process D). When Process D is finished executing, since Process B calls `waitpid()` system call and the Flags used here are `WCONTINUED` and `WUNTRACED`, the `waitpid()` system call will be blocking. Hence, once Process D terminates, Process B will reap Process D from the process table(i.e. The entry of Process D will be removed from the process table)

Case 3: We send the terminating signal to Process A after Process A has created child Process B. During this time, Process B has already created Process D.

In this case, Process A has already had the opportunity to create a child process B. Furthermore, child process B has already had the opportunity to create child process D. In this case when we send a terminating signal to Process A, Processes A, B, and D and obviously the main process(i.e. The parent process of Process A) are running in our system. Hence, when we terminate process A, we can notice that our main program has a `waitpid()` system call. Since the Flags used here are `WCONTINUED` and

WUNTRACED, the waitpid() system call will be blocking. Hence, once Process A terminates, the main process will reap Process A from the process table(i.e. the entry of Process A will be removed from the process table). However, since Process B is still running in our system, it will now be an orphan process and will be adopted by the init process. The init process always runs on the wait() system call so, when Process B terminates, it will be reaped by init. When Process D is finished executing, since Process B calls waitpid() system call and the Flags used here are WCONTINUED and WUNTRACED, the waitpid() system call will be blocking. Hence, once Process D terminates, Process B will reap Process D from the process table(i.e. The entry of Process D will be removed from the process table). Then, once Process B terminates, it will be reaped by init(i.e. The entry of Process B will be removed from the process table).

Case 4: We send a terminating signal to Process A after Process A has created Process C. In this case, Process A has already had the opportunity to create two child processes(i.e. Child Process B and Child Process C). In this case, when we send a terminating signal to Process A, Process B has already been reaped by Process A and, subsequently, Process D has already been reaped by Process B. Thus, the only processes that are alive and running in our system at this period of time is Process A and Process C and obviously the main process(i.e. The parent process of Process A). When we send a terminating signal to Process A, Process A will be terminated and the main process(i.e. The parent process of Process) will reap Process A(i.e. The entry of Process A will be removed from the process table). When Process C is still executing in our system and Process A has been terminated, Process C will become an orphan process and will be adopted by init. The init process always runs on the wait() system call so, when Process C terminates, it will be reaped by init.

2.

```
rr1133@engsoft:~/pc/Project#1/Question2$ ./Question2.out
Process A is currently executing and the pid of this process is 393451
Process A is now going to wait for B
Process B is currently executing and the pid of this process is 393454
Process B is now going to wait for D
We are now in Process 393450 and We will now print the Process Tree(i.e. pstree) Starting from its Parent
bash(391960)—Question2.out(393450)—Question2.out(393451)—Question2.out(393454)—Question2.out(393455)
└─sh(393456)—pstree(393457)
We are now in Process 393450 and We will now print the Process Tree(i.e. pstree) Starting from Itself
Question2.out(393450)—Question2.out(393451)—Question2.out(393454)—Question2.out(393455)
└─sh(393458)—pstree(393459)
We are now in Process 393450 and We will now print the Process Tree(i.e. pstree) Starting from its Child
Question2.out(393451)—Question2.out(393454)—Question2.out(393455)
Process D is currently executing and the pid of this process is 393455
Process D is now going to exit!
Child with PID = 393455 terminated normally, exited status = 10
Process B is now going to exit!
Child with PID = 393454 terminated normally, exited status = 4
Process A is now going to wait for C
Process C is currently executing and the pid of this process is 393477
We are now in Process 393450 and We will now print the Process Tree(i.e. pstree) Starting from its Parent
bash(391960)—Question2.out(393450)—Question2.out(393451)—Question2.out(393477)
└─sh(393479)—pstree(393480)
We are now in Process 393450 and We will now print the Process Tree(i.e. pstree) Starting from Itself
Question2.out(393450)—Question2.out(393451)—Question2.out(393477)
└─sh(393481)—pstree(393482)
We are now in Process 393450 and We will now print the Process Tree(i.e. pstree) Starting from its Child
Question2.out(393451)—Question2.out(393477)
Process C is now going to exit!
Child with PID = 393477 terminated normally, exited status = 6
Process A is now going to exit!
Child with PID = 393451 terminated normally, exited status = 2
The Process Tree will now be printed in its entirety
A
  B
    D
  C
rr1133@engsoft:~/pc/Project#1/Question2$
```

I have modified my program in Question 2 in order to display the process trees so I can analyze it and answer this question.

Based on my terminal output, I can draw the following conclusions:

- Whenever we call the `pstree(getpid())` command, we are asking the operating system to print the process tree starting from the current process. Furthermore, based on my experiments, it is clear that the operating system only displays processes that are RUNNING not on the waiting queue. I wanted to display processes that were RUNNING so I modified my `sleep()` system calls to ensure that the `pstree()` function would be called when the processes are all running and not on the waiting queue. I called the `pstree()` function twice. Once when processes A, B, and D were running and the second time was then Process A and C were running.
 - Analysis of First call to `pstree()` (i.e. When Processes A, B, and D were running):
 - As shown by my terminal output, I displayed the process tree in 3 different ways. The first process tree I displayed started from the parent process of the main process. To do this, I called

`ps-tree(getppid())` since `getppid()` returns the pid of the parent process. As shown, the first process is `bash(391960)` which is the pid of the shell. This is expected since I executed my program on the terminal. Hence, the `bash` shell has to be the parent of the main process. The child of `bash(391960)` is `Question2(393450)` and we know this is valid since our terminal output shows us that `393450` is the pid of the main process. The child of `Question2(393450)` are `Question2(393451)`, `sh(393456)`. This makes sense because the main process creates Process A, which has a pid of `393451`, using the `fork()` system call. Furthermore, it makes use of the shell to call `ps-tree`. Hence, it is expected that the `sh(393456)` is the child process of `Question2(393450)`. Finally, `ps-tree(393457)` is a child process of `sh(393456)` and this makes sense because the shell executes the `ps-tree` command. Process A has a child process `Question2(393454)`. This makes sense because Process A creates Process B, which has a pid of `393454`, using the `fork()` system call. Process B has a child process `Question2(393455)`. This makes sense because Process B creates Process D, which has a pid of `393455`, using the `fork()` system call.

- The second process tree I displayed started from the main process. To do this, I called `ps-tree(getpid())` since `getpid()` returns the pid of the calling process. The first item in the process tree is `Question 2(393450)` and this makes sense because the main process has a pid of `393450`. The children of `Question2(393450)` are `Question2(393451)`, `sh(393458)`. This makes sense because the main process creates Process A, which has a pid of `393451`, using the `fork()` system call. Furthermore, it makes use of the shell to call `ps-tree`. Hence, it is expected that the `sh(393458)` is the child process of `Question2(393450)`. Finally, `ps-tree(393459)` is a child process of `sh(393458)` and this makes sense because the shell executes the `ps-tree` command. Process A has a child process `Question2(393454)`. This makes sense because Process A creates Process B, which has a pid of `393454`, using the `fork()` system call. Process B has a child process `Question2(393455)`. This makes sense because Process B creates Process D, which has a pid of `393455`, using the `fork()` system call.
- The third process tree I displayed started from the child process of the main Process (i.e. Process A). To do this, I called `ps-tree(pid)` since `pid`, for a parent process, returns the pid of the child process that it has created via the `fork()` system call. Hence, in the main

method in the main program, we use the `fork()` system call to create Process A. Hence, `pid`, for the main process, will be equal to the `pid` of Process A. The first item in the process tree is `Question 2(393451)` and this makes sense because the main process creates Process A, which has a `pid` of 393451, using the `fork()` system call. Process A has a child process `Question2(393454)`. This makes sense because Process A creates Process B, which has a `pid` of 393454, using the `fork()` system call. Process B has a child process `Question2(393455)`. This makes sense because Process B creates Process D, which has a `pid` of 393455, using the `fork()` system call.

- Analysis of Second Call to `ps-tree()`(i.e. When Processes A and C are running)
 - As shown by my terminal output, I displayed the process tree in 3 different ways. The first process tree I displayed started from the parent process of the main process. To do this, I called `ps-tree(getppid())` since `getppid()` returns the `pid` of the parent process. As shown, the first process is `bash(391960)` which is the `pid` of the shell. This is expected since I executed my program on the terminal. Hence, the `bash` shell has to be the parent of the main process. The child of `bash(391960)` is `Question2(393450)` and we know this is valid since our terminal output shows us that 393450 is the `pid` of the main process. The child of `Question2(393450)` are `Question2(393451)`, `sh(393479)`. This makes sense because the main process creates Process A, which has a `pid` of 393451, using the `fork()` system call. Furthermore, it makes use of the shell to call `ps-tree`. Hence, it is expected that the `sh(393479)` is the child process of `Question2(393450)`. Finally, `ps-tree(393480)` is a child process of `sh(393479)` and this makes sense because the shell executes the `ps-tree` command. Process A has a child process `Question2(393477)`. This makes sense because Process A creates Process C, which has a `pid` of 393477, using the `fork()` system call.
 - The second process tree I displayed started from the main process. To do this, I called `ps-tree(getpid())` since `getpid()` returns the `pid` of the calling process. The first item in the process tree is `Question 2(393450)` and this makes sense because the main process has a `pid` of 393450. The children of `Question2(393450)` are `Question2(393451)`, `sh(393481)`. This makes sense because the main process creates Process A, which has a `pid` of 393451, using the `fork()` system call. Furthermore, it makes use of the shell to call `ps-tree`. Hence, it is expected that the `sh(393481)` is the child

process of Question2(393450). Finally, pstree(393482) is a child process of sh(393481) and this makes sense because the shell executes the pstree command. Process A has a child process Question2(393477). This makes sense because Process A creates Process C, which has a pid of 393477, using the fork() system call.

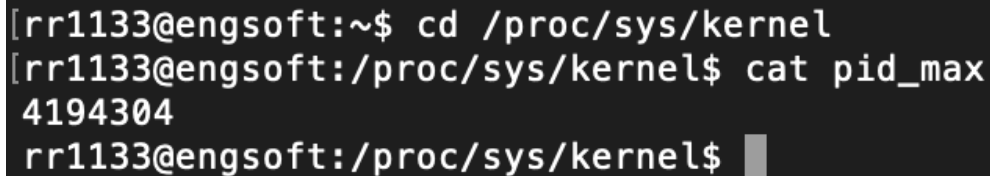
- The third process tree I displayed started from the child process of the main Process (i.e. Process A). To do this, I called pstree(pid) since pid, for a parent process, returns the pid of the child process that it has created via the fork() system call. Hence, in the main method in the main program, we use the fork() system call to create Process A. Hence, pid, for the main process, will be equal to the pid of Process A. The first item in the process tree is Question 2(393451) and this makes sense because the main process creates Process A, which has a pid of 393451, using the fork() system call. Process A has a child process Question2(393477). This makes sense because Process A creates Process C, which has a pid of 393477, using the fork() system call.

Conclusions that I can draw from this experiment:

- Calling pstree with getpid() will print the process tree starting from the calling process since getpid() returns the pid of the calling process. Hence, the calling process and all of its child processes, grandchild processes, etc that are not on the waiting queue will be printed via the pstree() command. Furthermore, when we call pstree(), we will need to ask the shell to invoke this command on our behalf. Hence, the shell process will also be a child of the calling process and the pstree command will be a child of the shell process since it was invoked by the shell.
- Calling pstree with pid will print the process tree starting from the child process of the calling process since, after a fork() call, the calling process will get the pid of the child process it has created. . Hence, the child processes, grandchild processes, etc of the calling process that are not on the waiting queue will be printed via the pstree() command
- Calling pstree with getppid() will print the process tree starting from the parent process of the calling process since getppid() returns the pid of the parent process of the calling process. Hence, the calling process and all of its child processes, grandchild processes, etc that are not on the waiting queue will be printed via the pstree() command. The parent process of the calling process will also be displayed Furthermore, when we call pstree(), we will need to ask the shell to invoke this command on our behalf. Hence, the shell process will also be a child of the calling process and the pstree command will be a child of the shell process since it was invoked by the shell.

3.

To first answer this question, I first did some experimentation to determine the theoretical limit regarding the highest possible Process Identifier of a Process that can run on an Operating System just out of curiosity. I was using the Ubuntu 20.04(i.e. Linux Distribution) Operating System since this is what Engsoft machines use. I ran the following two commands on the terminal: “cd /proc/sys/kernel” followed by “cat pid_max”. My results are shown below in a screenshot



```
[rr1133@engsoft:~$ cd /proc/sys/kernel  
[rr1133@engsoft:/proc/sys/kernel$ cat pid_max  
4194304  
rr1133@engsoft:/proc/sys/kernel$
```

This shows that the maximum value of the pid of a process can be 4194303. Once we have used up the pid of 4194303, the next process that is created on our Operating System will NOT receive a pid of 4194304. Instead, what the Operating System will do, is that it will wrap back around to 1 and will start looking for an unused pid to assign to the process.

Now, I wanted to test to see what is the number of processes that my main program could create. So, what this means is that I am trying to find out the number of processes that can be forked() by my calling process(i.e. The main program). To figure this out, I needed to query the RLIMIT_NPROC variable which, according to the Linux Manual, is defined as follows: It is a system-instituted limit on the number of processes that can be created by the calling process(i.e. The main program). So, if the number of processes belonging to the current process is greater than or equal to this limit, then fork() will fail with the error EAGAIN, indicating that we have hit a system-imposed limit in creating child processes.

To determine the value of RLIMIT_NPROC, I executed the following program on my Engsoft machine

```
#include <stdio.h>  
#include <sys/resource.h>  
int main() {
```



```

    struct rlimit queried_information;
    getrlimit(RLIMIT_NPROC, &queried_information);
    printf("%lu\n", queried_information.rlim_cur);
}

```

This was my result

```

[rr1133@engsoft:~/pc/Project#1$ ./testLimits.out
The value of RLIMIT_NPROC IS: 1000
rr1133@engsoft:~/pc/Project#1$ █

```

Hence, it is clear that, since RLIMIT_NPROC is 1000, our process tree can have a maximum number of 1000 processes. Regardless of the data structure that we use to create the process tree(i.e. Heap Data Structure, Left Skewed Tree Data Structure, Right Skewed Tree Data Structure), the process tree must have a maximum of 1000 nodes where each node corresponds to an individual process.

I wanted to further test whether the RLIMIT_NPROC was returning the correct information. To do this, I created a very simple program to experimentally determine the number of processes that can be in my process tree for any arbitrary program. I have included the program below:

```

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <stdarg.h>
#include <signal.h>
#include <time.h>

int main(void){
    for(int i = 0; i < 1000; i ++){
        pid_t pid = fork();
        if(pid < 0){
            perror("Fork Failed!\n");
        }
    }
}

```

```

        else if (pid == 0){
            continue;
        }
        else if (pid > 0){
            waitpid(pid, NULL, 0);
            exit(0);
        }
    }
}

```

When I executed this program in my engsoft machine, this is what I observed

```

r113@engsoft:~/pc/Project#ls ./systemTest.out
Current PID IS: 564447
Current PID IS: 564448
Current PID IS: 564449
Current PID IS: 564450
Current PID IS: 564451
Current PID IS: 564452
Current PID IS: 564453
Current PID IS: 564454
Current PID IS: 564455
Current PID IS: 564456
Current PID IS: 564457
Current PID IS: 564458
Current PID IS: 564459
Current PID IS: 564460
Current PID IS: 564461
Current PID IS: 564462
Current PID IS: 564463
Current PID IS: 564464
Current PID IS: 564465
Current PID IS: 564466
Current PID IS: 564467

```

```

Current PID IS: 564466
Current PID IS: 564467
Current PID IS: 564468
Current PID IS: 564469
Current PID IS: 564470
Current PID IS: 564471
Current PID IS: 564472
Current PID IS: 564473
Current PID IS: 564474
Current PID IS: 564475
Current PID IS: 564476
Current PID IS: 564477
Current PID IS: 564478
Current PID IS: 564479
Current PID IS: 564480
Current PID IS: 564481
Current PID IS: 564482
Current PID IS: 564483
Current PID IS: 564484
Current PID IS: 564485
Current PID IS: 564486
Current PID IS: 564487
Current PID IS: 564488
Current PID IS: 564489
Current PID IS: 564490
Current PID IS: 564491
Current PID IS: 564492
Current PID IS: 564493
Current PID IS: 564494
Current PID IS: 564495
Current PID IS: 564496
Current PID IS: 564497
Current PID IS: 564498
Current PID IS: 564499
Current PID IS: 564500
Current PID IS: 564501
Current PID IS: 564502
Current PID IS: 564503
Current PID IS: 564504
Current PID IS: 564505
Current PID IS: 564506
Current PID IS: 564507
Current PID IS: 564508
Current PID IS: 564509
Current PID IS: 564510
Current PID IS: 564511
Current PID IS: 564512
Current PID IS: 564513
Current PID IS: 564514
Current PID IS: 564515
Current PID IS: 564516
Current PID IS: 564517
Current PID IS: 564518
Current PID IS: 564519
Current PID IS: 564520
Current PID IS: 564521
Current PID IS: 564522
Current PID IS: 564523
Current PID IS: 564524
Current PID IS: 564525
Current PID IS: 564526
Current PID IS: 564527
Current PID IS: 564528
Current PID IS: 564529
Current PID IS: 564530
Current PID IS: 564531
Current PID IS: 564532
Current PID IS: 564533
Current PID IS: 564534
Current PID IS: 564535
Current PID IS: 564536
Current PID IS: 564537
Current PID IS: 564538
Current PID IS: 564539
Current PID IS: 564540
Current PID IS: 564541
Current PID IS: 564542
Current PID IS: 564543
Current PID IS: 564544

```

| | | | |
|---------|-----|-----|--------|
| Current | PID | 151 | 564622 |
| Current | PID | 151 | 564623 |
| Current | PID | 151 | 564625 |
| Current | PID | 151 | 564626 |
| Current | PID | 151 | 564627 |
| Current | PID | 151 | 564628 |
| Current | PID | 151 | 564631 |
| Current | PID | 151 | 564632 |
| Current | PID | 151 | 564633 |
| Current | PID | 151 | 564634 |
| Current | PID | 151 | 564635 |
| Current | PID | 151 | 564636 |
| Current | PID | 151 | 564637 |
| Current | PID | 151 | 564638 |
| Current | PID | 151 | 564639 |
| Current | PID | 151 | 564641 |
| Current | PID | 151 | 564642 |
| Current | PID | 151 | 564643 |
| Current | PID | 151 | 564644 |
| Current | PID | 151 | 564645 |
| Current | PID | 151 | 564646 |
| Current | PID | 151 | 564647 |
| Current | PID | 151 | 564648 |
| Current | PID | 151 | 564649 |
| Current | PID | 151 | 564651 |
| Current | PID | 151 | 564653 |
| Current | PID | 151 | 564654 |
| Current | PID | 151 | 564655 |
| Current | PID | 151 | 564656 |
| Current | PID | 151 | 564657 |
| Current | PID | 151 | 564658 |
| Current | PID | 151 | 564659 |
| Current | PID | 151 | 564661 |
| Current | PID | 151 | 564662 |
| Current | PID | 151 | 564663 |
| Current | PID | 151 | 564664 |
| Current | PID | 151 | 564665 |
| Current | PID | 151 | 564666 |
| Current | PID | 151 | 564667 |
| Current | PID | 151 | 564668 |
| Current | PID | 151 | 564669 |
| Current | PID | 151 | 564671 |
| Current | PID | 151 | 564672 |
| Current | PID | 151 | 564673 |
| Current | PID | 151 | 564676 |
| Current | PID | 151 | 564677 |
| Current | PID | 151 | 564678 |
| Current | PID | 151 | 564679 |
| Current | PID | 151 | 564680 |
| Current | PID | 151 | 564681 |
| Current | PID | 151 | 564682 |
| Current | PID | 151 | 564683 |
| Current | PID | 151 | 564684 |
| Current | PID | 151 | 564685 |
| Current | PID | 151 | 564686 |
| Current | PID | 151 | 564687 |
| Current | PID | 151 | 564688 |
| Current | PID | 151 | 564689 |
| Current | PID | 151 | 564690 |
| Current | PID | 151 | 564691 |
| Current | PID | 151 | 564692 |
| Current | PID | 151 | 564693 |
| Current | PID | 151 | 564694 |
| Current | PID | 151 | 564695 |
| Current | PID | 151 | 564697 |
| Current | PID | 151 | 564698 |
| Current | PID | 151 | 564699 |

Current PID is: 564853
Current PID is: 564854
Current PID is: 564855
Current PID is: 564856
Current PID is: 564857
Current PID is: 564858
Current PID is: 564859
Current PID is: 564860
Current PID is: 564861
Current PID is: 564862
Current PID is: 564863
Current PID is: 564864
Current PID is: 564865
Current PID is: 564866
Current PID is: 564867
Current PID is: 564868
Current PID is: 564869
Current PID is: 564870
Current PID is: 564871
Current PID is: 564872
Current PID is: 564873
Current PID is: 564874
Current PID is: 564875
Current PID is: 564876
Current PID is: 564877
Current PID is: 564878
Current PID is: 564879
Current PID is: 564880
Current PID is: 564881
Current PID is: 564882
Current PID is: 564883
Current PID is: 564884
Current PID is: 564885
Current PID is: 564886
Current PID is: 564887
Current PID is: 564888
Current PID is: 564889
Current PID is: 564890
Current PID is: 564891
Current PID is: 564892
Current PID is: 564893
Current PID is: 564894
Current PID is: 564895
Current PID is: 564896
Current PID is: 564897
Current PID is: 564898
Current PID is: 564899
Current PID is: 564900
Current PID is: 564901
Current PID is: 564902
Current PID is: 564903
Current PID is: 564904
Current PID is: 564905
Current PID is: 564906
Current PID is: 564907
Current PID is: 564908
Current PID is: 564909
Current PID is: 564910
Current PID is: 564911
Current PID is: 564912
Current PID is: 564913
Current PID is: 564914
Current PID is: 564915
Current PID is: 564916
Current PID is: 564917
Current PID is: 564918
Current PID is: 564919
Current PID is: 564920
Current PID is: 564921
Current PID is: 564922
Current PID is: 564923
Current PID is: 564924
Current PID is: 564925
Current PID is: 564926
Current PID is: 564927
Current PID is: 564928
Current PID is: 564929
Current PID is: 564930

[illegible]

| | |
|---------|----------------|
| Current | PID 1S: 463118 |
| Current | PID 1S: 463119 |
| Current | PID 1S: 463120 |
| Current | PID 1S: 463121 |
| Current | PID 1S: 463122 |
| Current | PID 1S: 463123 |
| Current | PID 1S: 463124 |
| Current | PID 1S: 463125 |
| Current | PID 1S: 463126 |
| Current | PID 1S: 463127 |
| Current | PID 1S: 463128 |
| Current | PID 1S: 463129 |
| Current | PID 1S: 463130 |
| Current | PID 1S: 463131 |
| Current | PID 1S: 463132 |
| Current | PID 1S: 463133 |
| Current | PID 1S: 463134 |
| Current | PID 1S: 463135 |
| Current | PID 1S: 463136 |
| Current | PID 1S: 463137 |
| Current | PID 1S: 463138 |
| Current | PID 1S: 463139 |
| Current | PID 1S: 463140 |
| Current | PID 1S: 463141 |
| Current | PID 1S: 463142 |
| Current | PID 1S: 463143 |
| Current | PID 1S: 463144 |
| Current | PID 1S: 463145 |
| Current | PID 1S: 463146 |
| Current | PID 1S: 463147 |
| Current | PID 1S: 463148 |
| Current | PID 1S: 463149 |
| Current | PID 1S: 463150 |
| Current | PID 1S: 463151 |
| Current | PID 1S: 463152 |
| Current | PID 1S: 463153 |
| Current | PID 1S: 463154 |
| Current | PID 1S: 463155 |
| Current | PID 1S: 463156 |
| Current | PID 1S: 463157 |
| Current | PID 1S: 463158 |
| Current | PID 1S: 463159 |
| Current | PID 1S: 463160 |
| Current | PID 1S: 463161 |
| Current | PID 1S: 463162 |
| Current | PID 1S: 463163 |
| Current | PID 1S: 463164 |
| Current | PID 1S: 463165 |
| Current | PID 1S: 463166 |
| Current | PID 1S: 463167 |
| Current | PID 1S: 463168 |
| Current | PID 1S: 463169 |
| Current | PID 1S: 463170 |
| Current | PID 1S: 463171 |
| Current | PID 1S: 463172 |
| Current | PID 1S: 463173 |
| Current | PID 1S: 463174 |
| Current | PID 1S: 463175 |
| Current | PID 1S: 463176 |
| Current | PID 1S: 463177 |
| Current | PID 1S: 463178 |
| Current | PID 1S: 463179 |
| Current | PID 1S: 463180 |
| Current | PID 1S: 463181 |
| Current | PID 1S: 463182 |
| Current | PID 1S: 463183 |
| Current | PID 1S: 463184 |
| Current | PID 1S: 463185 |
| Current | PID 1S: 463186 |
| Current | PID 1S: 463187 |
| Current | PID 1S: 463188 |
| Current | PID 1S: 463189 |
| Current | PID 1S: 463190 |
| Current | PID 1S: 463191 |
| Current | PID 1S: 463192 |
| Current | PID 1S: 463193 |
| Current | PID 1S: 463194 |


```

Current PID IS: 565396
Current PID IS: 565397
Current PID IS: 565398
Current PID IS: 565399
Current PID IS: 565400
Current PID IS: 565401
Current PID IS: 565402
Current PID IS: 565403
Current PID IS: 565404
Current PID IS: 565405
Current PID IS: 565406
Current PID IS: 565407
Current PID IS: 565408
Current PID IS: 565409
Current PID IS: 565410
Current PID IS: 565411
Current PID IS: 565412
Current PID IS: 565413
Current PID IS: 565414
Current PID IS: 565415
Current PID IS: 565416
Current PID IS: 565417
Current PID IS: 565418
Current PID IS: 565419
Current PID IS: 565420
Current PID IS: 565421
Current PID IS: 565422
Current PID IS: 565423
Current PID IS: 565424
Current PID IS: 565425
Current PID IS: 565426
Current PID IS: 565427
Current PID IS: 565428
Current PID IS: 565429
Current PID IS: 565430
Current PID IS: 565431
Current PID IS: 565432
Current PID IS: 565433
Current PID IS: 565434
Current PID IS: 565435
Current PID IS: 565436
Current PID IS: 565437
Current PID IS: 565438
Current PID IS: 565439
Current PID IS: 565440
Current PID IS: 565441
Current PID IS: 565442
Current PID IS: 565443
Current PID IS: 565444
Current PID IS: 565445
Current PID IS: 565446
Current PID IS: 565447
Current PID IS: 565448
Current PID IS: 565449
Fork Failed!
! Resource temporarily unavailable
Current PID IS: 565449
Fork Failed!
! Resource temporarily unavailable
Current PID IS: 565449
Fork Failed!
! Resource temporarily unavailable
rr133@engsoft:~/pc/Project#16

```

```

Fork Failed!
: Resource temporarily unavailable
Current PID IS: 565449
Fork Failed!
: Resource temporarily unavailable
Current PID IS: 565449
Fork Failed!
: Resource temporarily unavailable

```

Conclusions that I Drew From the Output Listed Above:

- The pid of the main process was 564447.
- The pid of the last process generated has a pid of 565449
- The terminal output shows that every child process had a pid that was 1 greater than its parent process
- Hence, I can conclude that the number of processes generated was $565449 - 564447 = 1002$ which is roughly close to the RLIMIT_NPROC value

To further test my hypothesis, I ran my program to generate 997 processes by making the following modification to the test program;

```

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

```

```

#include <stdarg.h>
#include <signal.h>
#include <time.h>
int main(void){
    for(int i = 0; i < 997; i ++){
        pid_t pid = fork();
        if(pid < 0){
            perror("Fork Failed!\n");
        }
        else if (pid == 0){
            continue;
        }
        else if (pid > 0){
            waitpid(pid, NULL, 0);
            exit(0);
        }
    }
}

```

```

Current PID IS: 568604
Current PID IS: 568605
Current PID IS: 568606
Current PID IS: 568607
Current PID IS: 568608
Current PID IS: 568609
Current PID IS: 568610
Current PID IS: 568611
Current PID IS: 568612
Current PID IS: 568613
Current PID IS: 568614
Current PID IS: 568615
Current PID IS: 568616
Current PID IS: 568617
Current PID IS: 568618
Current PID IS: 568619
Current PID IS: 568620
Current PID IS: 568621
Current PID IS: 568622
Current PID IS: 568623
Current PID IS: 568624
Current PID IS: 568625
Current PID IS: 568626
Current PID IS: 568627
Current PID IS: 568628
Current PID IS: 568629
Current PID IS: 568630
Current PID IS: 568631
Current PID IS: 568632
Current PID IS: 568633
Current PID IS: 568634
Current PID IS: 568635
Current PID IS: 568636
Current PID IS: 568637
Current PID IS: 568638
Current PID IS: 568639
Current PID IS: 568640
Current PID IS: 568641
Current PID IS: 568642
Current PID IS: 568643
Current PID IS: 568644
Current PID IS: 568645
Current PID IS: 568646
Current PID IS: 568647
Current PID IS: 568648
Current PID IS: 568649
Current PID IS: 568650
Current PID IS: 568651
Current PID IS: 568652
Current PID IS: 568653
Current PID IS: 568654
Current PID IS: 568655
Current PID IS: 568656
Current PID IS: 568657
Current PID IS: 568658
Current PID IS: 568659
Current PID IS: 568660
Current PID IS: 568661
Current PID IS: 568662
Current PID IS: 568663
Current PID IS: 568664
Current PID IS: 568665
Current PID IS: 568666
Current PID IS: 568667
Current PID IS: 568668
Current PID IS: 568669
Current PID IS: 568670
Current PID IS: 568671
Current PID IS: 568672
Current PID IS: 568673
Current PID IS: 568674
Current PID IS: 568675
Current PID IS: 568676
Current PID IS: 568677
Current PID IS: 568678
Current PID IS: 568679
Current PID IS: 568680
Current PID IS: 568681
rr1133@engsoft:~/pc/Project#1$ █

```

I have included the end of my terminal output just to prove that there was no error with the `fork()`. Hence, I was able to create 997 processes without any error!

From my aforementioned experimentation, I was able to conclude that `RLIMIT_NPROC` was giving me the correct information as the Linux Manual stated that the `RLIMIT_NPROC` value, according to the Linux Manual, is defined as follows: It is a system-instituted limit on the number of processes that can be created by the calling process(i.e. The main program).

Hence, it is clear that for any program I run on my engsoft machine, the maximum number of processes generated in the process tree corresponding to this program is roughly close to 1000.