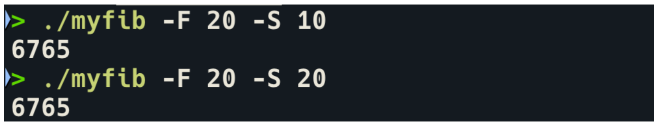
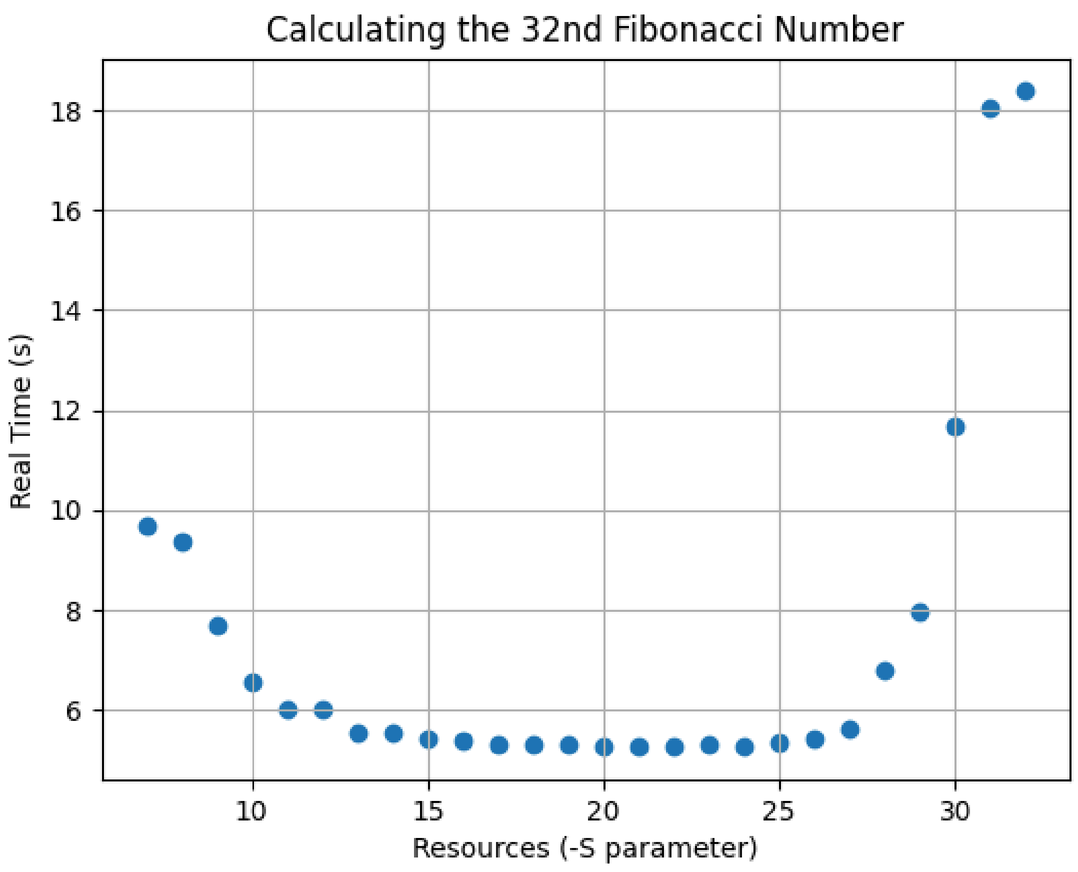
Programming Assignment #02

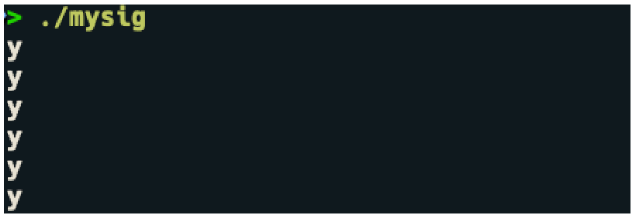
1. Fibonacci using Fork and inter-process communications


```
> ./myfib -F 20 -S 10
6765
> ./myfib -F 20 -S 20
6765
```

- The program calculates the **f**th Fibonacci number with a computing threshold of ***s***
- In the first case, ***f*** = 20 and ***s*** = 10, indicating that the program uses the fib_fork function for Fibonacci numbers greater than the threshold of 10. The computation of F(20) forks new processes for F(19) and F(18) and continue forking processes recursively as long as the Fibonacci index exceeds the threshold (i.e. (***f*** - 1) > ***s*** and (***f*** - 2) > ***s***). For computations of F(11) and below, the program will switch to the fib_seq function to complete the task
- In the second case, ***f*** = 20 and ***s*** = 20, the threshold for switching to fib_seq is now the Fibonacci index 20 itself. Since both F(19) and F(18) are below this threshold, the program directly computes the result using the recursive fib_seq function without any additional processes. The entire Fibonacci sequence is calculated using simple recursion.
- We notice that the difference between ***f*** and ***s*** is indicative of the number of processes that are created by the program, since a single process is created for each recursive call to fib_fork. In our testing, this resulted in failures to run the program when the difference between ***f*** and ***s*** became too large, and the system ran out of pids to assign to new processes. This happened when ***f-s*** was **26** on our particular systems.
- Additionally, we notice that creating too many processes results in a slower execution speed due to the overhead of creating new processes. The figure below shows that when ***f*** and ***s*** are equal (i.e. Fibonacci is calculated sequentially), the program takes the most time to run. As ***s*** decreases, the runtime improves; however, for values of ***s*** below 13, the runtime increases again.

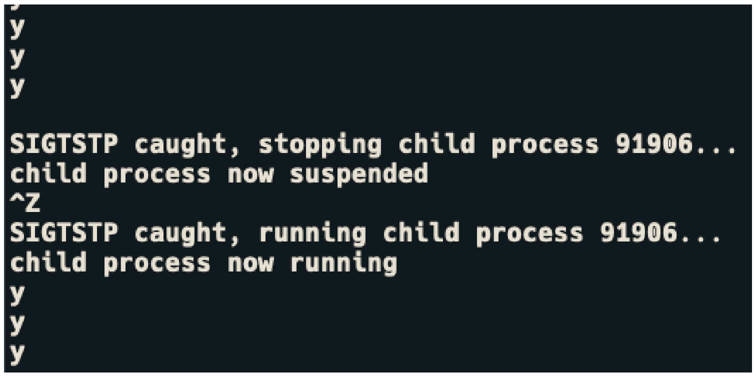


2. Inter-process communication using signals



```
> ./mysig
y
y
y
y
y
y
```

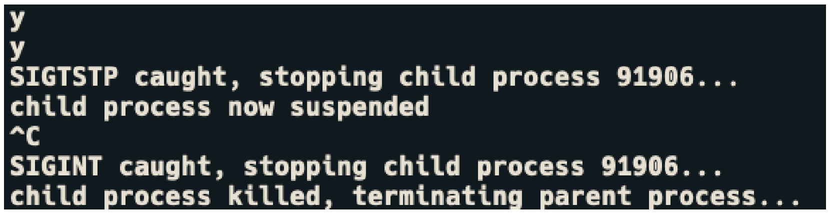
- When the program executes, the parent forks a child process, which uses **execve()** to run the "yes" command, printing "y" repeatedly. The parent enters an infinite loop, waiting for signals like **SIGINT** or **SIGTSTP**



```
y
y
y

SIGTSTP caught, stopping child process 91906...
child process now suspended
^Z
SIGTSTP caught, running child process 91906...
child process now running
y
y
y
```

- When **Ctrl+Z** is pressed, a **SIGSTP** signal is sent to the process. The parent's handle_sigtstp function is triggered, printing a caught message and sending **SIGSTOP** to suspend the child, halting the "y" output. The parent prints the "child process now suspended" message and waits for more signals.
- When **Ctrl+Z** is pressed again, the signal handler for **SIGTSTP** is invoked once more, printing another caught message and resuming the child process with **SIGCONT**. The parent process prints "child process now running" and the child resumes printing "y".



```
y
y
SIGTSTP caught, stopping child process 91906...
child process now suspended
^C
SIGINT caught, stopping child process 91906...
child process killed, terminating parent process...
```

- When **Ctrl+C** is pressed, a **SIGINT** signal is sent to the process. The parent's handle_sigint is invoked, sending **SIGKILL** to terminate the child. After confirming the child's exit, the parent terminates itself, ending the program. We use **SIGKILL** instead of **SIGINT** to stop the child process because the **SIGINT** signal is only handled for running processes.

3. Tasks

Both Matt and Leo developed individual versions of the code for Task 1 and Task 2. After reviewing and discussing both implementations, we decided to adopt Matt's version for the Fibonacci task and use Leo's version for the signal task. Leo wrote the report on Task 2 and the first half of Task 1, while Matt completed the second half of Task 1 and supplemented the report with the benchmark figure.