

1) Fibonacci using Fork and inter-process communications

Let's use the fork system call to create processes to compute the Fibonacci sequence of a given number. Write a program that takes two parameters: “-F *n* -S *m*”, where *n* is an input number to compute the *n*th number of the Fibonacci sequence and *m* specifies a computing threshold.

- ❖ In general, $\text{fib}(x) = \text{fib}(x-1) + \text{fib}(x-2)$, where $\text{fib}(0) = 0$ and $\text{fib}(1) = 1$
- ❖ **Case 1:** if $(n-1) > m$ or $(n-2) > m$, your program must use the fork system call to recursively spawn child processes to compute the next Fibonacci numbers.
 - ✓ When the child process completes its Fibonacci number computation, it will send the result back to its parent process based on inter-process communications (e.g., shared memory, message queue (mq_open), named pipe (mkfifo), etc.).
 - ✓ The parent process will receive the results by receive messages from its child processes.
 - ✓ The parent process should wait for all its child processes terminated and then close the inter-process communications if existing.
- ❖ **Case 2:** if $(n-1) \leq m$ and $(n-2) \leq m$, your program recursively call `fib_seq(x-1)` to compute the next Fibonacci numbers, where

```
int fib_seq(int x) { /* slow/recursive implementation of Fib*/ int i, rint
    = (rand()%30); double dummy;
    for (i=0; i<rint*100; i++) {dummy=2.345*i*8.765/1.234;}
    if (x==0) return(0); else if (x==1) return(1); else return(fib_seq(x-1)+fib_seq(x-2));
}
```

Your program should follow the given input/output format: “myfib -F *n* -S *m*”. You should output/print ONLY the *n*th Fibonacci number. See the example below.

```
$ myfib -F 6 -S 6
8
$ myfib -F 6 -S 3
8
```

The above two cases will output the same result “8”. The former one (**Case 2**) will do every computation in sequential while the later one (**Case 1**) will create several child processes to do the computation. In a multicore system, the later one should take less time to complete since multiple processes can utilize multiple cores. Hint:

- ❖ You can use C-library call “getopt” to handle your command line input.
- ❖ Be careful about how many child processes to be created.

2) Inter-process communication using signals

Write a program that will utilize signal handlers to intercept keyboard interrupts to manage control of a child process. Your main program should fork and execve a new child process to run the “yes” command (use `man yes` to learn more). You should then register signal handlers for both ctrl-c and ctrl-z in your main program. The ctrl-z signal handler should toggle the child “yes” process, i.e., stop the child process if it is running, or resume the child process if it is stopped, all while keeping the main process running. The ctrl-c signal should kill the child process, and then exit the main process. In both signal handlers, add print statements indicating what the program is doing – i.e. “ctrl-z caught, stopping child process”. Your main program should sit in an infinite loop once the setup has been completed. It should only exit when the user hits ctrl-c, after cleaning up the child process.

1) Fibonacci using Fork and inter-process communications

A program was implemented to achieve the desired functionality as is shown above. Within the main function the “getopt” function was used to take the desired user inputs when calling the compiled program and decide on which method to calculate the respective Fibonacci’s number.

```
int main(int argCount, char *argVariable[]){
    int Fvalue, Svalue;
    int opt;
    int x = 0;
    while ((opt = getopt(argCount, argVariable, "F:" "S:")) != -1){
        switch(opt){
            case 'F':
                Fvalue = atoi(optarg);
                break;
            case 'S':
                Svalue = atoi(optarg);
                break;
        }
    }

    //CASE 1 CODE
    if((Fvalue-1)>Svalue||(Fvalue-2)>Svalue){
        Result = forkfib(Fvalue);
    }
    //CASE 2 CODE
    else if ((Fvalue-1)<=Svalue&&(Fvalue-2)<=Svalue){
        Result = case2(Fvalue);
    }

    printf(" \n\n%d\n\n", Result);

    exit (0);
}
```

Figure 1 - A screenshot of Main() within the PA02fib.c source code file.

Case 1

The main function calls reference to the Case 1 function, known as `forkfib()` with the file title PA02fib.c, when entering an F Value and S Value that meet the following criterion:

$(F \text{ value} - 1) > S \text{ value}$ OR $(F \text{ value} - 2) > S \text{ value}$. The function is as shown in the table below and is intended to recursively procreate child processes. The F value passed into `forkfib` is decremented 1 by the child processes and 2 by the parents until a value of 1 or 0 is achieved. Then 1 or 0 is written into a shared memory and the respective process returns to `forkfib` where the parent function returns from a wait state after its child terminates and it sums the previous 2 values per the Fibonacci sequence. The program essentially creates the tree structure depicted below via forking and decrements then exits and sums in a reverse order.

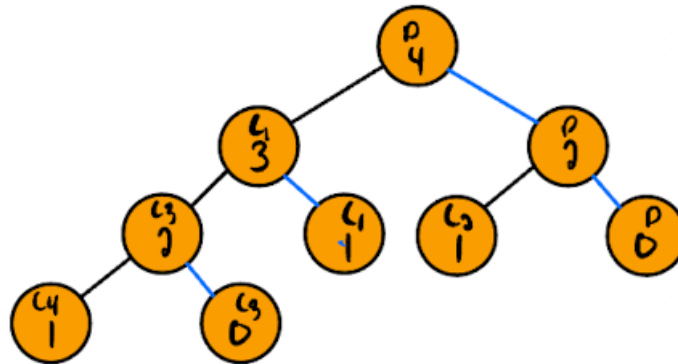


Figure 2 - Tree Structure Depicting the Case 1 Parent-Child Relationship

In the figure above, the blue connection points represent the relationship between parent processes and the black connection is the relationship between a parent and child processes. The value at the bottom of the orange circle is the F Value and in this example this tree was created with an F value of 4 which according to Appendix 1 is a Fibonacci Value of 3. For convenience, Appendix 1 of this document contains the first few Fibonacci numbers and their respective ratio, which approaches the so called “Golden Ratio” as the Fibonacci numbers increase.

This user requirement specification was attempted using the `forkfib()` function depicted on the following page. The function will decrement to create the correct number of parent and child processes. Prior forking, it establishes four shared memory locations, the four locations are intended to be used: as an array to store the values written in the conditional ($n == 0 \parallel n == 1$) statement, an indexer for this array, an array for the resultant values from the $\text{fib}(n) = \text{fib}(n-1) + \text{fib}(n-2)$ equation, and finally an indexer for the resultant array. Communication between these processes using the shared memory locations is successful. Therefore, the first sub-objective of having the child process complete the computation and then writing the result into shared memory location is achieved. The second sub-objective to allow the parent processes to receive the results using inter process communications was achieved as the parents are able to successfully read from shared memory. Finally, the third sub-objective, for the parent processes to wait for its child processes to terminate then close the inter-process communication was achieved. This is completed using the `waitpid()` library call and the `shmtcl()` library call coupled with the `IPC_RMID` argument, which will remove and delete the shared memory after the last (parent) process detaches from the shared memory.

Although these above sub-objectives were achieved the indexing within the shared memory is incorrect and leads to inconsistent behavior within the program. For example, the first time the program is called the behavior is different from the subsequent calls. The second and every time after that `-F 3 -S 1` is called it produces the correct answer. The first time only that `-F 4 -S 1` is called it is only correct the first time. As is shown in the figure below.

```

zacharymontoya@UbuntuVivado:/mnt/hgfs/01-Fall2022/ECE437-OperatingSystems/Homework (PA Assignments)/SharePoint - Home
• work (Programming Assignments - PA)/PA02/Code$ gcc -o PA02fib PA02fib.c
zacharymontoya@UbuntuVivado:/mnt/hgfs/01-Fall2022/ECE437-OperatingSystems/Homework (PA Assignments)/SharePoint - Home
• work (Programming Assignments - PA)/PA02/Code$ PA02fib -F 3 -S 1

0
zacharymontoya@UbuntuVivado:/mnt/hgfs/01-Fall2022/ECE437-OperatingSystems/Homework (PA Assignments)/SharePoint - Home
• work (Programming Assignments - PA)/PA02/Code$ PA02fib -F 3 -S 1

2
zacharymontoya@UbuntuVivado:/mnt/hgfs/01-Fall2022/ECE437-OperatingSystems/Homework (PA Assignments)/SharePoint - Home
• work (Programming Assignments - PA)/PA02/Code$ PA02fib -F 4 -S 1

3
zacharymontoya@UbuntuVivado:/mnt/hgfs/01-Fall2022/ECE437-OperatingSystems/Homework (PA Assignments)/SharePoint - Home
• work (Programming Assignments - PA)/PA02/Code$ PA02fib -F 4 -S 1

0

```

Figure 3 - A Screenshot of the `PA02fib.c` results for `-F 3 -S 1` and `-F 4 -S 1`.

As previously mentioned, this behavior is attributed to improper indexing with the shared memory array. The source code for case 1 is as follows.

```

25 //CASE1
26 int forkfib(int n){
27     //Setup for Resultant
28     int *ResArray;
29     int ResSMID;
30     key_t Reskey = ftok("ResSharedMemorFib3",65); //Grabbing Key
31     ResSMID = shmget(Reskey, SHMSZ, IPC_CREAT | 0666); // Creating Shared Memory Location
32     ResArray = (int *)shmat(ResSMID, 0, 0); // Attaching
33     //Setup for Result Indexer
34     int *IndexerRes;
35     int IndexerResSMID;
36     key_t Indexerreskey = ftok("IndexerResSharedMemoryFib3",65); //Grabbing Key
37     IndexerResSMID = shmget(Indexerreskey, SHMSZ, IPC_CREAT | 0666); // Creating Shared Memory Location
38     IndexerRes = (int *)shmat(IndexerResSMID, 0, 0); // Attaching
39     //Setup for Adder Indexer
40     int *IndexerArray;
41     int IndexersMID;
42     key_t Indexerkey = ftok("IndexerSharedMemoryFib3",65); //Grabbing Key
43     IndexersMID = shmget(Indexerkey, SHMSZ, IPC_CREAT | 0666); // Creating Shared Memory Location
44     IndexerArray = (int *)shmat(IndexersMID, 0, 0); // Attaching
45     //Setup for Adder Array
46     int shmid;
47     int *array;
48     key_t key = ftok("ArraySharedMemoryFib3",65); //Grabbing Key
49     shmid = shmget(key, SHMSZ, IPC_CREAT | 0666); // Creating Shared Memory Location
50     array = (int *)shmat(shmid, 0, 0); // Attaching
51
52     if(n == 0 || n == 1){
53         array[IndexerArray[0]] = n;
54         IndexerArray[0] = IndexerArray[0] + 1;
55         return(0);
56     }
57     else{
58         pid_t ChildPid = fork();
59         if(ChildPid == 0){ //child
60             forkfib(n-1);
61             exit(0);
62         }
63         else{ //parent
64             waitpid(ChildPid,NULL,0);
65             forkfib(n-2);
66             int ChildIndex = IndexerArray[0];
67             ChildIndex = ChildIndex - 2;
68             int ParentIndex = IndexerArray[0];
69             ParentIndex = ParentIndex - 1;
70             int childResult = array[ChildIndex];
71             int parentResult = array[ParentIndex];
72             int Fibn = childResult + parentResult;
73             ResArray[100] = ResArray[100] + Fibn;
74             // Removing Memory Location
75             int rmidIndexer = shmctl(IndexersMID, IPC_RMID, NULL);
76             int rmidArray = shmctl(shmid, IPC_RMID, NULL);
77             int rmidResArray = shmctl(ResSMID, IPC_RMID, NULL);
78             int rmidResIndexer = shmctl(IndexerResSMID, IPC_RMID, NULL);
79             return(ResArray[100]);
80         }
81     }
82 }

```

Figure 4 - A Screenshot of the forkfib() function within the PA02fib.c source code file.

Case 2:

The main function calls reference to the Case 2 function when entering an F Value and S Value that meet the following criterion: $(F \text{ value} - 1) \leq S \text{ value}$ & $(F \text{ value} - 2) \leq S \text{ value}$. The function is as follows and returns the following result for the main function to print to the terminal:

```
// CASE2 - slow/recursive implementation of Fib
int case2(int x) {
    int i, rint = (rand()%30);
    double dummy;
    for (i=0; i<rint*100; i++)dummy=2.345*i*8.765/1.234;
    if (x==0)return(0);
    else if (x==1)return(1);
    else return(case2(x-1)+case2(x-2));
}
```

Figure 5 - A screenshot of the case2() function within the PA02fib.c source code file.

```
zacharymontoya@UbuntuVivado:/mnt/hgfs/01-Fall2022/ECE437-OperatingSystems/Homewo
rk (PA Assignments)/Local Copies/PA02 Copy/Code$ gcc -o PA02fib PA02fib.c
zacharymontoya@UbuntuVivado:/mnt/hgfs/01-Fall2022/ECE437-OperatingSystems/Homewo
rk (PA Assignments)/Local Copies/PA02 Copy/Code$ PA02fib3 -F 4 -S 4

3 ←

zacharymontoya@UbuntuVivado:/mnt/hgfs/01-Fall2022/ECE437-OperatingSystems/Homewo
rk (PA Assignments)/Local Copies/PA02 Copy/Code$ PA02fib3 -F 10 -S 10

55 ←
```

Figure 6 - A screenshot of the return values from case2() within the PA02fib.c source code file.

As is shown above, the single process recursive implementation for calculating the Fibonacci number successfully returns the correct Fibonacci number per Appendix 1 of this document.

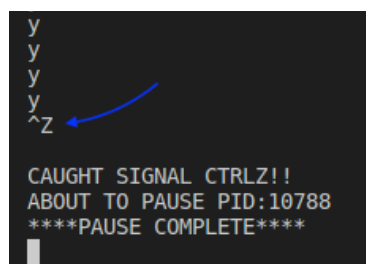
2) Inter-process communication using signals

This program functions as intended and is described within the first page of this document. It is comprised of a main process that forks into a parent and child. The child code and parent code are delineated using the returned value from the fork, 0 for the child and PID of the child to the parent. This allows for separate codes to be executed as the parent and child processes both continue. Signal handlers were setup within the scope of the parent code after the fork() because having them prior to the fork created erroneous process behavior when both the parent and child were handling signals. User defined signal interrupts were used but they were mapped to the keys CTRL+C and CTRL+Z, the interrupt signals are SIGINT and SIGSTP respectively. These are some of the interrupt signals that can be caught and handled¹.

The parent process will run infinity in a while loop that does absolutely nothing while immediately after the fork the child processes with execve into the simple GNU program “yes”. This program will infinitely print new lines of “y”, or any other string passed to it when called until the user interrupts the program. This is a program created by David MacKenzie[1], that is used to pipe the infinite “y”s into another program that may be asking the user a question that remains within UNIX systems is perfect for this situation. The child immediately erupts into a wall of “y”s and can be paused using CTRL+Z, as the interrupt is handled using the code below and the program pauses see the figure below.

```
void userDef_signalHandlerZ(int sigZ){//CTRL+Z is the toggle INT
printf("\n\nCAUGHT SIGNAL CTRLZ!!");
if(ChildPIDValue>0){
    if(ToggleBit<1){//CHILD IS RUNNING IF TOGGLEBIT==0
        printf("\nABOUT TO PAUSE PID:%d",ChildPIDValue);
        kill(ChildPIDValue,SIGSTOP);
        ToggleBit = 1;
        printf("\n****PAUSE COMPLETE****\n");
    }
    else{//CHILD IS STOPPED IF TOGGLEBIT==1
        printf("\nABOUT TO RESUME PID:%d",ChildPIDValue);
        kill(ChildPIDValue,SIGCONT);
        ToggleBit = 0;
        printf("\n****RESUME COMPLETE****\n");
    }
}
}
```

Figure 7 - A screenshot of the CTRL+Z Interrupt Handler.



```
y
y
y
y
^Z
CAUGHT SIGNAL CTRLZ!!
ABOUT TO PAUSE PID:10788
****PAUSE COMPLETE****
```

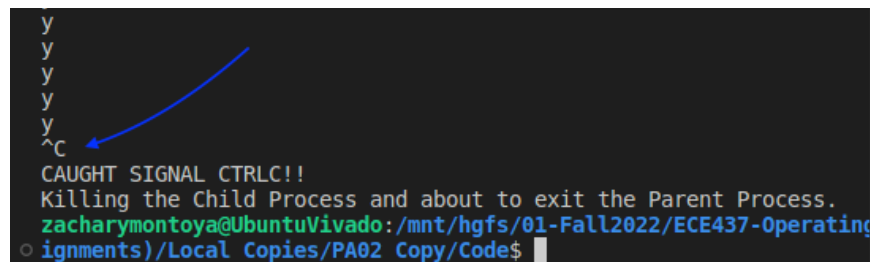
Figure 8 - A screenshot of a CTRL+Z signal being handled.

¹Note that SIGKILL and SIGSTOP which is different from SIGSTP can NOT be handled.

When CTRL+C is pressed, the interrupt is handled using the following code which will send a SIGKILL interrupt to the child process and exit, as is shown in the figures below.

```
void userDef_signalHandlerC(int sigC){//CTRL+C is to terminate child and exit the parent.  
    printf("\nCAUGHT SIGNAL CTRLC!!");  
    if(ChildPIDValue>0){  
        printf("\nKilling the Child Process and about to exit the Parent Process.\n");  
        kill(ChildPIDValue,SIGKILL);  
        exit(0);  
    }  
}
```

Figure 9 - A screenshot of the CTRL+C Interrupt Handler.



```
y  
y  
y  
y  
y  
y  
y  
^C  
CAUGHT SIGNAL CTRLC!!  
Killing the Child Process and about to exit the Parent Process.  
zacharymontoya@UbuntuVivado: /mnt/hgfs/01-Fall2022/ECE437-Operating  
Systems/Local Copies/PA02 Copy/Code$
```

Figure 10 - A screenshot of a CTRL+C signal being handled.

1) Bibliography

[1] N. Montfort, “The Trivial Program ‘yes,’” p. 12.

2) Appendix 1 – Fibonacci Numbers

The ratio is calculated via the following equation:

$$\text{Ratio} = \frac{\text{Fibonacci (F Value)}}{\text{Fibonacci (F Value - 1)}}$$

Table 1 - First 50 Fibonacci Numbers

F Value	Fibonacci Number	Ratio
0	0	
1	1	
2	1	1.0000000000000000
3	2	2.0000000000000000
4	3	1.5000000000000000
5	5	1.6666666666666670
6	8	1.6000000000000000
7	13	1.6250000000000000
8	21	1.6153846153846200
9	34	1.6190476190476200
10	55	1.6176470588235300
11	89	1.6181818181818200
12	144	1.6179775280898900
13	233	1.6180555555555600
14	377	1.6180257510729600
15	610	1.6180371352785100
16	987	1.6180327868852500
17	1597	1.6180344478216800
18	2584	1.6180338134001300
19	4181	1.6180340557275500
20	6765	1.6180339631667100
21	10946	1.6180339985218000
22	17711	1.6180339850173600
23	28657	1.6180339901756000
24	46368	1.6180339882053200
25	75025	1.6180339889579000
26	121393	1.6180339886704400
27	196418	1.6180339887802400
28	317811	1.6180339887383000

F Value	Fibonacci Number	Ratio
29	514229	1.6180339887543200
30	832040	1.6180339887482000
31	1346269	1.6180339887505400
32	2178309	1.6180339887496500
33	3524578	1.6180339887499900
34	5702887	1.6180339887498600
35	9227465	1.6180339887499100
36	14930352	1.6180339887498900
37	24157817	1.6180339887499000
38	39088169	1.6180339887498900
39	63245986	1.6180339887499000
40	102334155	1.6180339887498900
41	165580141	1.6180339887498900
42	267914296	1.6180339887498900
43	433494437	1.6180339887498900
44	701408733	1.6180339887498900
45	1134903170	1.6180339887498900
46	1836311903	1.6180339887498900
47	2971215073	1.6180339887498900
48	4807526976	1.6180339887498900
49	7778742049	1.6180339887498900
50	12586269025	1.6180339887498900