

### Q1] Shared Memory :

One producer and  $n$  consumers share a memory  $M[]$  capable of storing two int variables. The producer generates items (random integers) in  $M[1]$  for a predetermined number  $t$  of times. For each item generated, the producer specifies in  $M[0]$  the consumer (an integer in the range  $1, 2, 3, \dots, n$ ) for which the item written in  $M[1]$  is meant. The designated consumer reads  $M[1]$ , and sets  $M[0]$  to 0, indicating that the item is consumed. After all of the  $t$  items are generated and consumed, the producer writes  $-1$  to  $M[0]$ . After reading this special value of  $M[0]$ , each consumer prints some aggregate information, and terminates. Finally, the producer terminates too after printing some aggregate information.

In order to implement this set of actions, write a C program **prodcons.c**. The parent process (call it  $P$ ) plays the role of the producer.  $P$  reads  $n$  (the number of consumers) and  $t$  (the number of items to be produced) from the user or as command-line arguments. Then,  $P$  creates a shared-memory segment  $M$  capable of storing two int variables.  $P$  also initializes  $M[0]$  to 0 (implying that no item is available for consumption at the beginning).  $P$  then forks  $n$  child processes  $C_1, C_2, \dots, C_n$  which play the roles of the  $n$  consumers. These child processes (or consumers) are numbered  $1, 2, \dots, n$ . After this,  $P$  goes to a production loop, and each  $C_i$  goes to a consumption loop. The loops run until all of the  $t$  items are produced and consumed. These loops work as follows.

#### Production Loop :

For each  $i = 1, 2, \dots, t$ , the producer  $P$  (parent in our case) generates a random 3-digit int value item and a random consumer  $c$  in the range  $1, 2, \dots, n$ .  $P$  waits (busy wait) until  $M[0]$  becomes 0. When  $M[0]$  becomes 0,  $P$  sets  $M[0]$  to  $c$  and  $M[1]$  to item (in that order). An optional delay (you can use `usleep()`) between setting  $M[0]$  and setting  $M[1]$  should be used if a compile-time macro `SLEEP` is set.

After producing  $t$  items,  $P$  waits (busy wait) until  $M[0]$  becomes 0 (that is, the last item is consumed by the designated consumer child).  $P$  then writes  $-1$  to  $M[0]$ , and waits until all of the  $n$  child processes terminate.  $P$  then prints, for each consumer  $c$ , the count of items produced for  $c$ , and the sum of these items.

$P$  finally removes the shared-memory segment  $M$ , and exits.

#### Consumption Loop :

The  $c$ -th consumer waits until  $M[0]$  becomes  $c$  or  $-1$ . If  $M[0]$  becomes  $c$ , the consumer reads  $M[1]$  as the next item meant for it. When  $M[0]$  becomes  $-1$ , the consumption loop is broken. The number of items read by the consumer and the sum of these items are then printed, and the child process terminates.

**Compile Time Tags :**

The default behavior of your program should be to print only an initial message and the final statistics. If the compile-time flag `VERBOSE` is set, then the production and the consumption of each item should also be printed (see the Sample Output). Another compile-time flag `SLEEP` (already mentioned above) dictates whether there is no delay between the setting of `M[0]` and the setting of `M[1]` by the producer (this should be the default behavior if the flag is not set) or there is a small delay (of 1–10 microseconds) between these two assignments. This delay simulates preemption of `P` (which would otherwise be very difficult to reproduce), and highlights the necessity of synchronization for this producer-consumer problem

Submit a single C source file `prodcons.c`.

## Sample Output

```
$ gcc -Wall prodcons.c ; ./a.out
n = 5
t = 100

Producer is alive

Consumer 1 is alive
Consumer 2 is alive
Consumer 3 is alive

Consumer 5 is alive
Consumer 4 is alive
Consumer 5 has read 25 items: Checksum = 13488
Consumer 2 has read 23 items: Checksum = 13657
Consumer 1 has read 17 items: Checksum = 10204
Consumer 4 has read 18 items: Checksum = 10798
Consumer 3 has read 17 items: Checksum = 7715

Producer has produced 100 items
17 items for Consumer 1: Checksum = 10204
23 items for Consumer 2: Checksum = 13657
17 items for Consumer 3: Checksum = 7715
18 items for Consumer 4: Checksum = 10798
25 items for Consumer 5: Checksum = 13488

$ gcc -Wall -DVERBOSE prodcons.c ; ./a.out
n = 4
t = 10

Producer is alive
Producer produces 288 for Consumer 2

Consumer 1 is alive
Consumer 2 is alive

Consumer 2 reads 288

Producer produces 281 for Consumer 3

Consumer 3 is alive
Consumer 3 reads 281

Producer produces 326 for Consumer 4

Consumer 4 is alive
Consumer 4 reads 326

Producer produces 535 for Consumer 2

Consumer 2 reads 535

Producer produces 505 for Consumer 1

Consumer 1 reads 505

Producer produces 848 for Consumer 2

Consumer 2 reads 848

Producer produces 799 for Consumer 3

Consumer 3 reads 799

Producer produces 828 for Consumer 4

Consumer 4 reads 828

Producer produces 884 for Consumer 4

Consumer 4 reads 884

Producer produces 688 for Consumer 4

Consumer 4 reads 688
Consumer 1 has read 1 items: Checksum = 505
Consumer 2 has read 3 items: Checksum = 1671
Consumer 3 has read 2 items: Checksum = 1080
Consumer 4 has read 4 items: Checksum = 2726

Producer has produced 10 items
1 items for Consumer 1: Checksum = 505
3 items for Consumer 2: Checksum = 1671
2 items for Consumer 3: Checksum = 1080
4 items for Consumer 4: Checksum = 2726

$ gcc -Wall -DVERBOSE -DSLEEP prodcons.c ; ./a.out
n = 2
t = 10

Producer is alive
Producer produces 846 for Consumer 1

Consumer 1 is alive

Consumer 1 reads 0
Consumer 2 is alive

Producer produces 648 for Consumer 1

Consumer 1 reads 846

Producer produces 889 for Consumer 1

Consumer 1 reads 648

Producer produces 861 for Consumer 2

Consumer 2 reads 889

Producer produces 913 for Consumer 1

Consumer 1 reads 861

Producer produces 924 for Consumer 2

Consumer 2 reads 913

Producer produces 450 for Consumer 1

Consumer 1 reads 924

Producer produces 671 for Consumer 1

Consumer 1 reads 450

Producer produces 168 for Consumer 2

Consumer 2 reads 671

Producer produces 364 for Consumer 2

Consumer 2 reads 168
Consumer 1 has read 6 items: Checksum = 3729
Consumer 2 has read 4 items: Checksum = 2641

Producer has produced 10 items
6 items for Consumer 1: Checksum = 4417
4 items for Consumer 2: Checksum = 2317
```

## Q2 . Pipes

In this assignment, you write a single C program CSE.c. Compile the program to an executable file called CSE. The program deals with three initial processes called C, S, and E. They work as follows.

### Supervisor (S):

This is the parent process. This process creates the necessary pipe(s), and then forks two child processes henceforth referred to as the “First Child” and the “Second Child”, respectively. Each of these child processes opens an xterm to run ./CSE itself for interacting with the user.

### Command-input child (C):

This child keeps on reading lines of commands from the user in its own xterm, and sending the commands verbatim to the other child.

### Execute-command child (E):

This child keeps on reading the commands sent by the other child, and executing them in the foreground (by forking child processes) in its own **xterm**. The parent S initiates First Child in the C mode and the Second Child in the E mode. It then waits until both the child processes terminate.

Each command supplied by the user can be one of the following:

1. A standard Linux command (like ls, ps, who, cat) with any number of command-line parameters but without pipes (like |) or redirections (like >, >>, and <).
2. The special command exit that terminates both the child processes.
3. Another special command swaprole that swaps the roles (C and E) of the two child processes. That is, when the user types this command (without any arguments) to the C child, the C child sends this command verbatim to the E child. After this, the C child switches to the E (execute-command) mode, and the E child switches to the C (command-input) mode. This command may be supplied by the user any number of times in the appropriate C windows.

All of the processes C, S, and E run the same executable file ./CSE. If ./CSE is run without any commandline arguments, it is the parent process (S). After the parent forks the First Child in C mode or the Second Child in E mode, each child execs xterm which in turn runs ./CSE with appropriate command-line arguments, so that each child knows its role (mode) along with other relevant information (like pipe fd's).

Follow the instructions given below in order to implement CSE.c.

- An xterm can be opened with a customized title to run an executable file (in our case, ./CSE) as

```
xterm -T "Customized Title" -e ./CSE arg1 arg2 arg3 . . .
```

- Child C reads user commands from its stdin. It sends the command to Child E via a pipe. Child E reads each command from the pipe, and forks a (grand)child for executing the command (and itself waits until the grandchild terminates). The grandchild prints the output of the command to the

stdout of E. This pipe must be created by the parent S, and the corresponding file descriptors must be sent to the child processes as command-line arguments. Two child processes cannot themselves establish an unnamed pipe between them. Do not involve the parent S in any child-to-child communication.

- The stdout of C should be dup-ed as the write-end of the pipe. Likewise, the stdin of E should be dup-ed as the read-end of the pipe. This allows the two child processes to use high-level I/O primitives like scanf, printf, fgets, fputs, . . . This however prevents the Child C from printing a prompt like Enter Command> to the terminal. Use the un-dup-ed stderr for that purpose. This is not the intended use of stderr, but nobody will mind.
- A swaprole command will necessitate the restoring of the original stdin or stdout. This can be achieved by maintaining copies of the original stdin and of the original stdout (can be done by dup once at the beginning).
- Use fflush() whenever it is necessary to flush an output buffer (like stdout) immediately.

There are two more problems to solve. The solutions are outlined below.

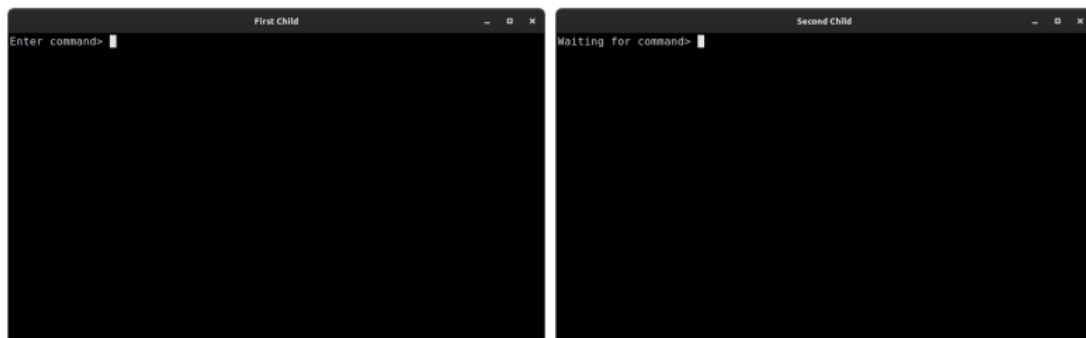
1. Suppose that the user supplies the command **swaprole** in the C window. Immediately after this input, this situation may happen: C writes the command to the pipe, switches role to E, and reads from the pipe, before the earlier E gets a chance to read from the pipe. This should not happen. Do not use sleep to delay the earlier C (and give the earlier E time to read from the pipe). Use another pipe instead (to be created by S before forking).

2. Suppose that the user supplies an interactive command. When E lets this command get executed by a (grand)child process, the stdin of E and of the grandchild is the pipe's read end. That is, the inputs for the interactive command should come from the other window C. This can be unimaginably bad (for example, you will go mad if you open a text editor in one window and type in another window). To solve this problem, note that the (interactive) command is not executed by E itself, so E can afford to continue with the dup-ed definition of stdin. The grandchild which actually does the running of the command should restore the original stdin for taking user inputs in the E window itself. This strategy will also relieve E and the grandchild from any contention over user inputs. That's all indeed. Submit the single C source file CSE.c

### Sample Run of CSE

Run `./CSE` from a terminal (let its prompt be `$`). This prints the following lines to the terminal, and opens two **xterms**. Note the pipe file descriptors (3 and 4) for the communication between the child processes.

```
$ gcc -Wall -o CSE CSE.c
$ ./CSE
+++ CSE in supervisor mode: Started
+++ CSE in supervisor mode: pfd = [3 4]
+++ CSE in supervisor mode: Forking first child in command-input mode
+++ CSE in supervisor mode: Forking second child in execute mode
```



You type some commands in the **First Child**, and the **Second Child** executes them.

First Child

Enter command- ls /  
Enter command- ls -la -p /  
Enter command- w  
Enter command- █

Second Child

Waiting for command- ls /  
bin cdrom etc lib lib64 lost-found mnt proc run snap sys usr  
boot dev home lib32 libx32 media opt root/sbin srv tmp var  
Waiting for command- ls -la -p /  
./ boot/ etc/ lib32 lost-found/ opt/ run/ srv/ usr/  
./ cdrom/ home/ lib64 media/ proc/ sbin sys/ var/  
bin dev/ lib libx32 mnt/ root/ snap/ tmp/  
Waiting for command- w  
23:57:55 up 8:41, 3 users, load average: 0.18, 0.19, 0.18  
USER TTY FROM LOGIN# IDLE JCPU PCPU WHAT  
abhlj :0 :0 15:49 7xdt7 9:22 0.01s /usr/lib/gdm3/g  
abhlj pts/0 :0 15:49 7:31m 0.07s 0.07s -bin/tcsh  
abhlj pts/1 :0 15:52 51.00s 0.23s 0.00s ./CSE  
Waiting for command- █

Then, you supply the special command **swaprole**.

First Child

Enter command- ls /  
Enter command- ls -la -p /  
Enter command- w  
Enter command- swaprole  
Waiting for command- █

Second Child

Waiting for command- ls /  
bin cdrom etc lib lib64 lost-found mnt proc run snap sys usr  
boot dev home lib32 libx32 media opt root/sbin srv tmp var  
Waiting for command- ls -la -p /  
./ boot/ etc/ lib32 lost-found/ opt/ run/ srv/ usr/  
./ cdrom/ home/ lib64 media/ proc/ sbin sys/ var/  
bin dev/ lib libx32 mnt/ root/ snap/ tmp/  
Waiting for command- w  
00:00:46 up 8:44, 3 users, load average: 0.20, 0.16, 0.17  
USER TTY FROM LOGIN# IDLE JCPU PCPU WHAT  
abhlj :0 :0 15:49 7xdt7 9:37 0.01s /usr/lib/gdm3/g  
abhlj pts/0 :0 15:49 7:34m 0.07s 0.07s -bin/tcsh  
abhlj pts/1 :0 15:52 36.00s 0.21s 0.00s ./CSE  
Waiting for command- swaprole  
Enter command- █

The **Second Child** reads some commands, and the **First Child** runs them.

First Child

Enter command- ls /  
Enter command- ls -la -p /  
Enter command- w  
Enter command- swaprole  
Waiting for command- ps  
PID TTY TIME CMD  
10988 pts/3 00:00:00 CSE  
10978 pts/3 00:00:00 ps  
Waiting for command- who  
abhlj :0 2024-01-20 15:49 (:0)  
abhlj pts/0 2024-01-20 15:49 (:0)  
abhlj pts/1 2024-01-20 15:52 (:0)  
Waiting for command- abc  
\*\*\* Unable to execute command  
Waiting for command-  
Waiting for command- █

Second Child

Waiting for command- ls /  
bin cdrom etc lib lib64 lost-found mnt proc run snap sys usr  
boot dev home lib32 libx32 media opt root/sbin srv tmp var  
Waiting for command- ls -la -p /  
./ boot/ etc/ lib32 lost-found/ opt/ run/ srv/ usr/  
./ cdrom/ home/ lib64 media/ proc/ sbin sys/ var/  
bin dev/ lib libx32 mnt/ root/ snap/ tmp/  
Waiting for command- w  
00:00:46 up 8:44, 3 users, load average: 0.20, 0.16, 0.17  
USER TTY FROM LOGIN# IDLE JCPU PCPU WHAT  
abhlj :0 :0 15:49 7xdt7 9:37 0.01s /usr/lib/gdm3/g  
abhlj pts/0 :0 15:49 7:34m 0.07s 0.07s -bin/tcsh  
abhlj pts/1 :0 15:52 36.00s 0.21s 0.00s ./CSE  
Waiting for command- swaprole  
Enter command- ps  
Enter command- who  
Enter command- abc  
Enter command- █

Another **swapprole**, and the first child recursively invokes **./CSE**. Two additional **xterms** are opened. The new **xterms** communicate between them leaving the first two **xterms** untouched. Note that the new **xterms** are using different pipe file descriptors (9 and 10).

First Child

```

Enter command- ls /
Enter command- ls -a -p /
Enter command- w
Enter command- swapprole
Waiting for command- ps
  PID TTY          TIME CMD
 10988 pts/3    00:00:00 CSE
 10978 pts/3    00:00:00 ps
Waiting for command- who
abhi|  :0          2024-01-20 15:49 (:0)
abhi|  pts/0      2024-01-20 15:49 (:0)
abhi|  pts/1      2024-01-20 15:52 (:0)
Waiting for command- abc
*** Unable to execute command
Waiting for command-
Waiting for command- swapprole
Enter command- ./CSE
Enter command-

```

Second Child

```

bin  cdrom  etc  lib  lib64  lost+found  mnt  proc  run  snap  sys  usr
boot  dev  home  lib32  libx32  media  opt  root /sbin  srv  tmp  var
./  boot/  etc/  lib32  lost+found/  opt/  run/  srv/  usr/
./  cdrom/  home/  lib64  media/  proc/ /sbin  sys/  var/
bin  dev/  lib  libx32  mnt/  root/  snap/  tmp/
Waiting for command- w
00:00:46 up 8:44, 3 users, load average: 0.20, 0.16, 0.17
USER      TTY      FROM            LOGIN@   IDLE   JCPU   PCPU   WHAT
abhi|    :0          :0              15:49   7xdm?  9:37   0.01s  /usr/lib/gdm3/g
abhi|    pts/0      :0              15:49   7:34m  0.07s  -bin/tcsh
abhi|    pts/1      :0              15:52  36.00s  0.21s  0.00s  ./CSE
Waiting for command- swapprole
Enter command- ps
Enter command- who
Enter command- abc
Enter command-
Enter command- swapprole
Waiting for command- ./CSE
+++ CSE in supervisor mode: Started
+++ CSE in supervisor mode: pfd = [9 10]
+++ CSE in supervisor mode: Forking first child in command-input mode
+++ CSE in supervisor mode: Forking second child in execute mode

```

First Child

```

Enter command- cal 2024
Enter command-

```

Second Child

```

 7  8  9 10 11 12 13   5  6  7  8  9 10 11   2  3  4  5  6  7  8
14 15 16 17 18 19 20 12 13 14 15 16 17 18   9 10 11 12 13 14 15
21 22 23 24 25 26 27 19 20 21 22 23 24 25 16 17 18 19 20 21 22
28 29 30                26 27 28 29 30 31      23 24 25 26 27 28 29
                                   30

      July          August          September
Su Mo Tu We Th Fr Sa Su Mo Tu We Th Fr Sa Su Mo Tu We Th Fr Sa
 1  2  3  4  5  6           1  2  3           1  2  3  4  5  6  7
 7  8  9 10 11 12 13     4  5  6  7  8  9 10     8  9 10 11 12 13 14
14 15 16 17 18 19 20 11 12 13 14 15 16 17 15 16 17 18 19 20 21
21 22 23 24 25 26 27 18 19 20 21 22 23 24 22 23 24 25 26 27 28
28 29 30 31              25 26 27 28 29 30 31 29 30

      October       November       December
Su Mo Tu We Th Fr Sa Su Mo Tu We Th Fr Sa Su Mo Tu We Th Fr Sa
 1  2  3  4  5           1  2  3  4  5           1  2  3  4  5  6  7
 6  7  8  9 10 11 12     3  4  5  6  7  8  9     8  9 10 11 12 13 14
13 14 15 16 17 18 19 10 11 12 13 14 15 16 15 16 17 18 19 20 21
20 21 22 23 24 25 26 17 18 19 20 21 22 23 22 23 24 25 26 27 28
27 28 29 30 31          24 25 26 27 28 29 30 29 30 31
Waiting for command-

```

The new **xterms** close by the user input **exit**.

First Child

```

Enter command- ls /
Enter command- ls -a -p /
Enter command- w
Enter command- swapprole
Waiting for command- ps
  PID TTY          TIME CMD
 10988 pts/3    00:00:00 CSE
 10978 pts/3    00:00:00 ps
Waiting for command- who
abhi|  :0          2024-01-20 15:49 (:0)
abhi|  pts/0      2024-01-20 15:49 (:0)
abhi|  pts/1      2024-01-20 15:52 (:0)
Waiting for command- abc
*** Unable to execute command
Waiting for command-
Waiting for command- swapprole
Enter command- ./CSE
Enter command-

```

Second Child

```

Waiting for command- ls -a -p /
./  boot/  etc/  lib32  lost+found/  opt/  run/  srv/  usr/
./  cdrom/  home/  lib64  media/  proc/ /sbin  sys/  var/
bin  dev/  lib  libx32  mnt/  root/  snap/  tmp/
Waiting for command- w
00:00:46 up 8:44, 3 users, load average: 0.20, 0.16, 0.17
USER      TTY      FROM            LOGIN@   IDLE   JCPU   PCPU   WHAT
abhi|    :0          :0              15:49   7xdm?  9:37   0.01s  /usr/lib/gdm3/g
abhi|    pts/0      :0              15:49   7:34m  0.07s  -bin/tcsh
abhi|    pts/1      :0              15:52  36.00s  0.21s  0.00s  ./CSE
Waiting for command- swapprole
Enter command- ps
Enter command- who
Enter command- abc
Enter command-
Enter command- swapprole
Waiting for command- ./CSE
+++ CSE in supervisor mode: Started
+++ CSE in supervisor mode: pfd = [9 10]
+++ CSE in supervisor mode: Forking first child in command-input mode
+++ CSE in supervisor mode: Forking second child in execute mode
+++ CSE in supervisor mode: First child terminated
+++ CSE in supervisor mode: Second child terminated
Waiting for command-

```

Yet another **swapprole**, and the user is going to terminate by entering **exit** in the current **C** window.

First Child

```

Enter command- ls /
Enter command- ls -a -p /
Enter command- w
Enter command- swapprole
Waiting for command- ps
  PID TTY          TIME CMD
 10988 pts/3    00:00:00 CSE
 10978 pts/3    00:00:00 ps
Waiting for command- who
abhi|  :0          2024-01-20 15:49 (:0)
abhi|  pts/0      2024-01-20 15:49 (:0)
abhi|  pts/1      2024-01-20 15:52 (:0)
Waiting for command- abc
*** Unable to execute command
Waiting for command-
Waiting for command- swapprole
Enter command- ./CSE
Enter command- swapprole
Waiting for command-

```

Second Child

```

./  boot/  etc/  lib32  lost+found/  opt/  run/  srv/  usr/
./  cdrom/  home/  lib64  media/  proc/ /sbin  sys/  var/
bin  dev/  lib  libx32  mnt/  root/  snap/  tmp/
Waiting for command- w
00:00:46 up 8:44, 3 users, load average: 0.20, 0.16, 0.17
USER      TTY      FROM            LOGIN@   IDLE   JCPU   PCPU   WHAT
abhi|    :0          :0              15:49   7xdm?  9:37   0.01s  /usr/lib/gdm3/g
abhi|    pts/0      :0              15:49   7:34m  0.07s  -bin/tcsh
abhi|    pts/1      :0              15:52  36.00s  0.21s  0.00s  ./CSE
Enter command- ps
Enter command- who
Enter command- abc
Enter command-
Enter command- swapprole
Waiting for command- ./CSE
+++ CSE in supervisor mode: Started
+++ CSE in supervisor mode: pfd = [9 10]
+++ CSE in supervisor mode: Forking first child in command-input mode
+++ CSE in supervisor mode: Forking second child in execute mode
+++ CSE in supervisor mode: First child terminated
+++ CSE in supervisor mode: Second child terminated
Waiting for command- swapprole
Enter command- exit

```

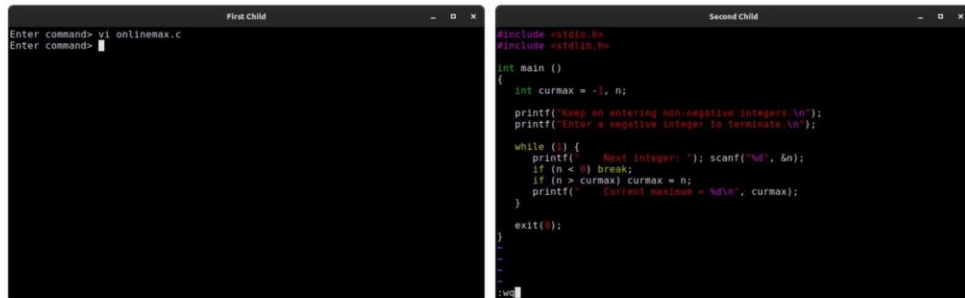


The final lines are written by your original terminal.

```
+++ CSE in supervisor mode: First child terminated
+++ CSE in supervisor mode: Second child terminated
$
```

## Running interactive programs

Edit a C file.



```
First Child
Enter command- vi onlinemax.c
Enter command-
Enter command- swapon

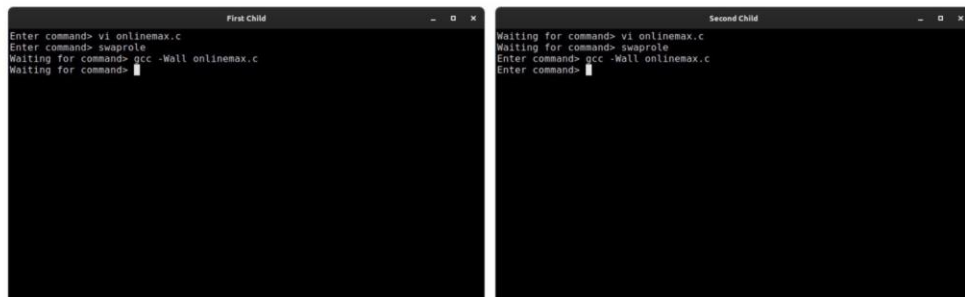
Second Child
#include <stdio.h>
#include <stdlib.h>

int main ()
{
    int curmax = -1, n;

    printf("Keep on entering non-negative integers.\n");
    printf("Enter a negative integer to terminate.\n");

    while (1) {
        printf("Next integer: "); scanf("%d", &n);
        if (n < 0) break;
        if (n > curmax) curmax = n;
        printf("Current maximum = %d\n", curmax);
    }
    exit(0);
}
```

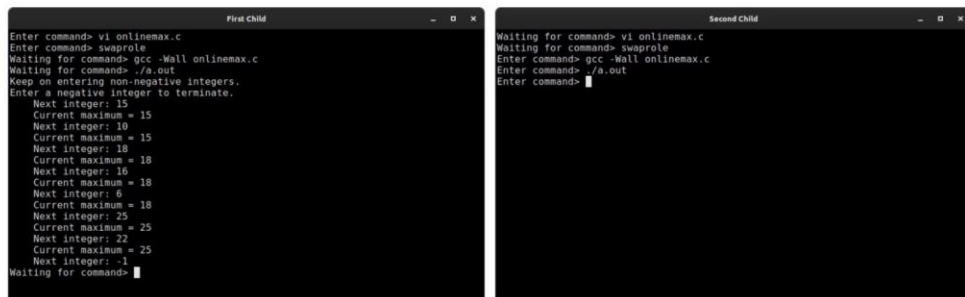
Swaprole and compile.



```
First Child
Enter command- vi onlinemax.c
Enter command- swapon
Waiting for command- gcc -Wall onlinemax.c
Waiting for command-
Waiting for command- ./a.out

Second Child
Waiting for command- vi onlinemax.c
Waiting for command- swapon
Enter command- gcc -Wall onlinemax.c
Enter command-
Enter command- ./a.out
```

Run



```
First Child
Enter command- vi onlinemax.c
Enter command- swapon
Waiting for command- gcc -Wall onlinemax.c
Waiting for command- ./a.out
Keep on entering non-negative integers.
Enter a negative integer to terminate.
Next integer: 15
Current maximum = 15
Next integer: 10
Current maximum = 15
Next integer: 18
Current maximum = 18
Next integer: 16
Current maximum = 18
Next integer: 6
Current maximum = 18
Next integer: 25
Current maximum = 25
Next integer: 22
Current maximum = 25
Next integer: -1
Waiting for command-

Second Child
Waiting for command- vi onlinemax.c
Waiting for command- swapon
Enter command- gcc -Wall onlinemax.c
Enter command- ./a.out
Enter command-
```

### Q3 . IPC using Signals

Think of  $n$  children  $C_1, C_2, C_3, \dots, C_n$  standing in a circle, and playing a game with the parent  $P$  standing at the center of the circle.  $P$  throws a ball to the children in a circular sequence. If the child (say,  $C_i$ ) to which the ball is thrown can catch the ball, then  $C_i$  continues to play. If  $C_i$  misses the ball, then  $C_i$  goes out of the game. After each throw, the ball comes back to  $P$  who then throws the ball to the next (in the circular order) child who is not yet out of the game. Eventually,  $n - 1$  children miss and go out of the game. The remaining child wins the game.

You need to implement this game as a multi-process application, where the processes can communicate with one another by sending signals. You write two programs `parent.c` and `child.c` to simulate the working of the parent and each child process. Suppose that these two programs are compiled to the executable files `parent` and `child`. The program `parent` (which simulates  $P$ ) is run with one command-line argument  $n$  (the number of child processes).  $P$  creates  $n$  child processes  $C_i$  which exec `child`  $i$  for  $i = 1, 2, 3, \dots, n$ .  $P$  also writes, in a text file `childpid.txt`, the child count  $n$  and the PIDs of the  $n$  child processes created by  $P$ . Each child waits for some time (like one second) for  $P$  to finish writing to `childpid.txt`. After this wait, each child reads  $n$  and the  $n$  PIDs from the text file `childpid.txt`. After the child creation,  $P$  waits for some time (like two seconds) so that each child process gets time to read this text file. These waits may be implemented by `sleep()` or `usleep()`, but after this, no waiting based on these functions will be allowed.

The parent  $P$  starts the game by sending `SIGUSR2` to  $C_1$ .  $P$  then enters a loop which continues until only one child is left as the player. Each child  $C_i$ , on the other hand, enters an infinite loop. The body of each loop should contain the single system call `pause()` which lets the calling process wait until it receives a signal (avoid busy waits). The game of throwing balls and catching/missing throws will be implemented by sending and handling signals. In this assignment, we use the three signals `SIGUSR1`, `SIGUSR2`, and `SIGINT` only.

For a child process  $C_i$ , receiving `SIGUSR2` implies that a throw is made to it. It then randomly decides whether it catches the ball (with probability 0.8) or misses the ball (with probability 0.2). If  $C_i$  can catch the ball, it sends `SIGUSR1` to  $P$ . If  $C_i$  fails to catch the ball, it sends `SIGUSR2` to  $P$ . Depending on the type of the signal received from the child  $C_i$  (to which the throw is made), the parent knows whether that child continues to play the game or is out of the game.  $P$  records this information.

After the outcome of a throw is recorded as explained above,  $P$  initiates a printing of the current status of all the  $n$  players. Since  $P$  has all the necessary information, it can do that printing itself. However, as a part of this assignment, this printing should be done by the child processes. This is achieved by sending `SIGUSR1` to the child processes in turn. Recall that to a child process, `SIGUSR2` means that a throw is made to it. On the other hand, the reception of `SIGUSR1` initiates that child to print its current status. The possible status of a child are `PLAYING` (written as `...`), `CATCHMADE` (written as `CATCH`), `CATCHMISSED` (written as `MISS`), and `OUTOFGAME` (written as blank). See the sample at the end to know the format of printing.  $P$  initiates the printing process by sending `SIGUSR1` to  $C_1$ . For each  $i < n$ ,  $C_i$  prints its status, and then sends `SIGUSR1` to the next child  $C_{i+1}$ . The last child  $C_n$  prints its status, but does not send `SIGUSR1` to any other process. However,  $C_n$  takes part in the synchronization activity in a different manner. Until all the child processes finish writing their status, the parent  $P$  must wait before it can make the throw to the next playing child. But you do not know many synchronization primitives at this moment, so let this wait be

accomplished by `waitpid()`. Before sending `SIGUSR1` to `C1`, `P` forks a dummy child process `D` and writes the PID of `D` in a text file `dummyspid.txt`. `P` then waits until `D` exits. When `Cn` is done printing its status, it reads the PID of `D` from the file `dummyspid.txt`, and sends `SIGINT` to `D`. Write `dummy.c` (the code for `D`) that enters an infinite loop of `pause()` at the beginning of its `main()`.

When `D` exits, `P` wakes up, and works out the next playing child process `Cnext` to which the throw is to be made. Note that `P` maintains the information of the status of all child processes. `P` should also keep track of the child process to which the last throw is made. So `P` can determine `Cnext` easily. `P` then sends `SIGUSR2` to `Cnext`, and the game continues as explained above.

After  $n - 1$  child processes miss throws, the parent sends `SIGINT` to all the  $n$  child processes. Only the last playing child process prints a happy message, and exits (see the format in the sample). The other processes exit without printing anything.

Notice that the sequencing of the throw-and-print cycle must be implemented only by signals (and by the `waitpid()` in one case). No other synchronization mechanism is allowed. You may use `fflush(stdout)` to avoid garbled output. But do not use any sleep or `usleep` calls (except only at the very beginning, that is, before the game starts).

You may use the following makefile.

```
all:
    gcc -Wall -o parent parent.c
    gcc -Wall -o child child.c
    gcc -Wall -o dummy dummy.c

run: all
    ./parent 10

clean:
    -rm -f parent child dummy childpid.txt dummyspid.txt
```

Submit a zip/tar/tgz archive containing the files `parent.c`, `child.c`, `dummy.c`, and `makefile`.

### Sample Output

```
$ make run
gcc -Wall -o parent parent.c
gcc -Wall -o child child.c
gcc -Wall -o dummy dummy.c
./parent 10
Parent: 10 child processes created
Parent: Waiting for child processes to read child database
```

[illegible]