

CS39002 Operating Systems Laboratory
Spring 2024

Lab Assignment: 3
Date of submission: 24-Jan -2024

IPC using pipe and dup

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 command-line 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.

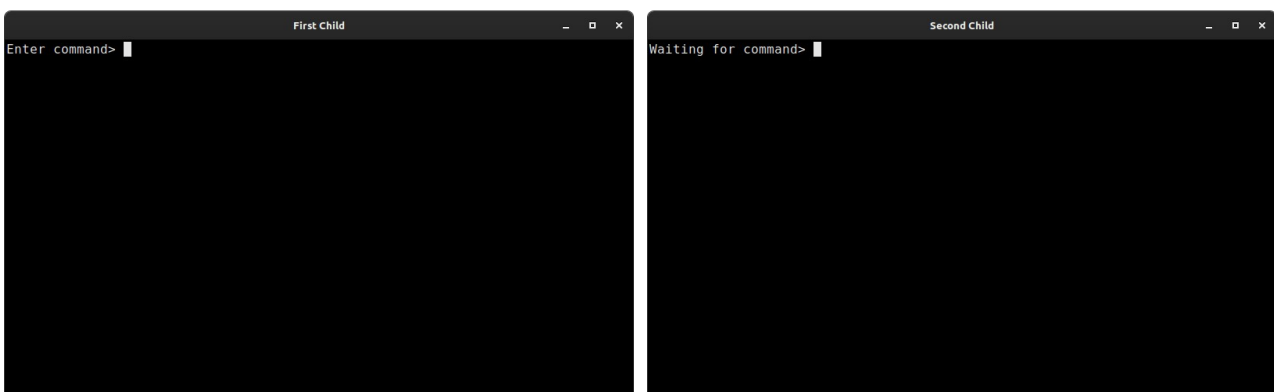
That's all indeed.

Submit the single C source file *CSE.c* to the CSE Moodle server before it is too late.

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 -a -p /
Enter command> w
Enter 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 -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
23:57:55 up 8:41, 3 users, load average: 0.18, 0.19, 0.18
USER TTY FROM LOGIN@ IDLE JCPU PCPU WHAT
abhij :0 :0 15:49 ?xdm? 9:22 0.01s /usr/lib/gdm3/g
abhij pts/0 :0 15:49 7:31m 0.07s 0.07s -bin/tcsh
abhij 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 -a -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 -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
abhij :0 :0 15:49 ?xdm? 9:37 0.01s /usr/lib/gdm3/g
abhij pts/0 :0 15:49 7:34m 0.07s 0.07s -bin/tcsh
abhij 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 -a -p /
Enter command> w
Enter command> swaprole
Waiting for command> ps
PID TTY TIME CMD
10908 pts/3 00:00:00 CSE
10978 pts/3 00:00:00 ps
Waiting for command> who
abhij :0 2024-01-20 15:49 (:0)
abhij pts/0 2024-01-20 15:49 (:0)
abhij 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 -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
abhij :0 :0 15:49 ?xdm? 9:37 0.01s /usr/lib/gdm3/g
abhij pts/0 :0 15:49 7:34m 0.07s 0.07s -bin/tcsh
abhij 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>
Enter command> █
```

Another **swaprole**, 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).

The image shows four terminal windows arranged in a 2x2 grid. The top-left window is titled 'First Child' and shows the execution of 'swaprole' and 'ps' commands. The top-right window is titled 'Second Child' and shows the execution of 'ls -a -p /' and 'w' commands, followed by 'swaprole' and 'ps' commands. The bottom-left window is titled 'First Child' and shows the execution of 'cal 2024'. The bottom-right window is titled 'Second Child' and shows a calendar for the year 2024, followed by 'swaprole' and 'ps' commands. The output of 'ps' in the bottom-right window shows the process list for the Second Child, including the 'swaprole' process and its children.

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

The image shows two terminal windows. The left window is titled 'First Child' and shows the execution of 'swaprole' and 'ps' commands, followed by 'exit'. The right window is titled 'Second Child' and shows the execution of 'ls -a -p /' and 'w' commands, followed by 'swaprole' and 'ps' commands, and finally 'exit'. The output of 'ps' in the right window shows the process list for the Second Child, including the 'swaprole' process and its children, and the message 'Second child terminated'.

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

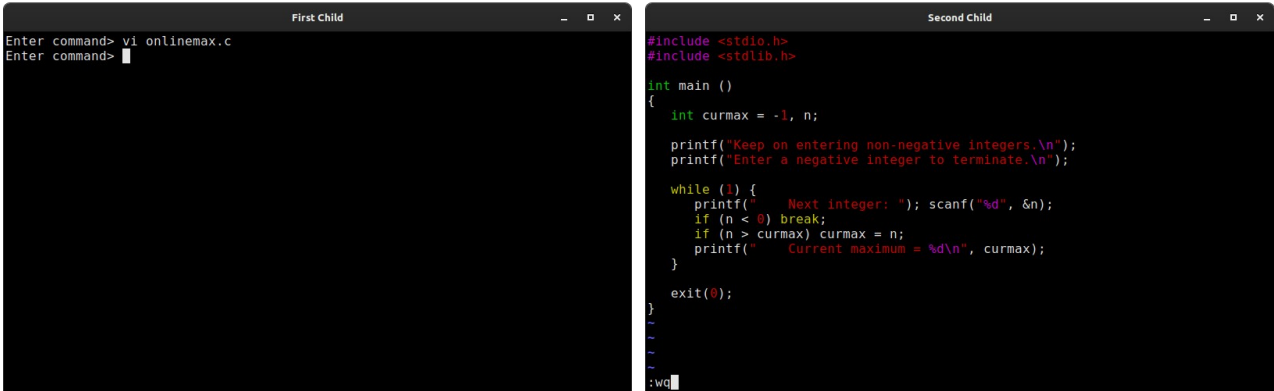
The image shows two terminal windows. The left window is titled 'First Child' and shows the execution of 'swaprole' and 'ps' commands, followed by 'exit'. The right window is titled 'Second Child' and shows the execution of 'ls -a -p /' and 'w' commands, followed by 'swaprole' and 'ps' commands, and finally 'exit'. The output of 'ps' in the right window shows the process list for the Second Child, including the 'swaprole' process and its children, and the message 'Second child terminated'.

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>

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

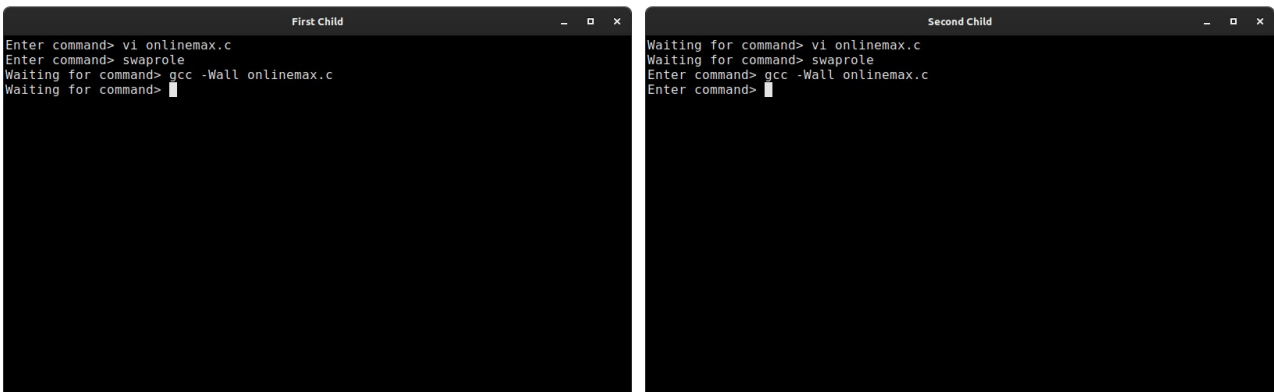
int main ()
{
    int curmax = -1, n;

    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);
}
:wq
```

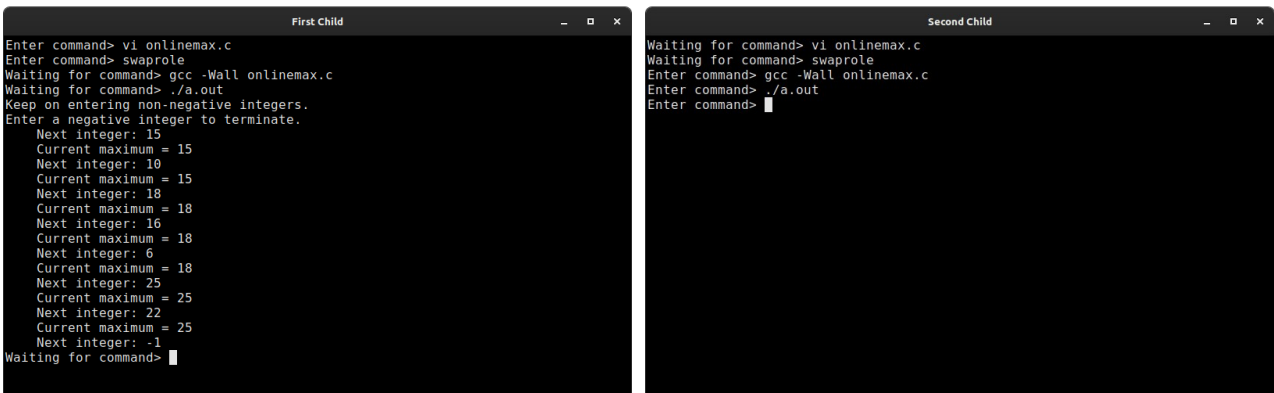
Swaprole and compile.



```
First Child
Enter command> vi onlinemax.c
Enter command> swaprole
Waiting for command> gcc -Wall onlinemax.c
Waiting for command>

Second Child
Waiting for command> vi onlinemax.c
Waiting for command> swaprole
Enter command> gcc -Wall onlinemax.c
Enter command>
```

Run



```
First Child
Enter command> vi onlinemax.c
Enter command> swaprole
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> swaprole
Enter command> gcc -Wall onlinemax.c
Enter command> ./a.out
Enter command>
```

Extras (if you care, NOT for submission)

There are two issues not yet dealt with by the submission part of the assignment, given on the earlier pages. These are for your personal consumption only. If you solve these, you may send a Sales Quotation for your product against a tender floated by FooBar Inc.

- The (current) **E** window redirects its *stdin* to the input end of the pipe, and accepts nothing from the original *stdin*. However, the user can type things against the prompt: `Waiting for command>` To the **E** window, this is garbage, and nothing would happen at this moment. But these inputs are not lost. They will stay in the buffer of the real *stdin*. No running process can straightaway control this, because this happens between the user and the device (terminal).

Later, when the **E** window reinstates the original *stdin* (for example, after a **swaprole** or while running an interactive program by a grandchild), the stored garbage will be delivered to the **E** window. This will certainly have unwelcome effects. For example, after a **swaprole**, the window (now in **C** mode) will treat the garbage as command(s) entered by the user, whereas an interactive program will consume the garbage as its own input.

How can you ensure that the garbage actually goes to something like a *trash bin*, and does not interfere with the normal intended working of the windows? Note that `fflush(stdin)` will not work.

- Handle `^C` and `^Z` in each of the three windows **C**, **S**, and **E** (it is assumed that you run **S** from a terminal window on your desktop). Pressing each of these should print “Type exit in the C window” only once in the respective window, and will have no further effect.

FooBar Inc. needs a single file *CSE.c*.