

1

- **Run the code example and explain the output.** My_pipe is an array of two integers, and we create a pipe with that array then we check if it returns an error. After that fork is called, and in the child process the first element of the pipe array is closed because we won't be reading, but writing into the write part (second element) of the pipe array. Then a file that is associated with the write end of the pipe is created, followed by making the calling thread sleep for 2 seconds. After that fprintf writes to that file "Are you my parent?" and the child returns 42.
In the parent process we close the write end of the pipe, since we will no longer be writing, only reading. A file is created associated with the read part of the pipe. A string called buffer is created where fgets writes what's been read from the child thread into the string. Printf prints "My child asked" followed by the value of the string buffer, which is the string that was written to the pipe in the child process, "Are you my parent?". wait(&status) stores status information in the int to which it points, which is the int status in this case. The value that is printed after the string "And then returned" is what was returned in the child process, and is retrieved by using WEXITSTATUS(status).e
- **How the output changes if the sleep statement in child process is moved before fgets statement of the parent?** Nothing changes. Even when the sleep command is removed it still works, but without the two second delay. I think this is because we use wait to ensure that the parent waits for the child to finish.

2

- **What happens when you run the echo command?** It passes the text "HI FIFO" to the named pipe as an argument.
- **What happens if you run echo command first and then cat command?** The process is sent a SIGPIPE (broken pipe: write to pipe with no readers) signal because the FIFO hasn't been opened for reading by running cat first, so what is written with echo won't be read into the FIFO.
- **Use man fifo(7), where is the data is sent through FIFO store?** Data is not written to the filesystem, but passed internally by the kernel. So it doesn't have contents on the filesystem; the filesystem is used as a reference points for other processes wanting to access the pipe using a name in the file system.

3

- **Do we need to synchronize the two programs to ensure that the read operation will always happen after the write operation?** Yes, and it can be done by using a named semaphore that is shared between both of the processes so that it will tell the read operation to sem_wait for the sem_post signal from the write operation before executing the critical block of the read operation.

4

- **What happens when you try to quit the program by using CTRL + C?. Explain the result.** The first argument of signal specifies which signals to catch, and the second argument of signal is the handler if the signal in the first argument is caught; in this case the function my_quit_handler is called whenever the SIGINT (ctrl c) is caught. Signal also blocks the default behavior if the signal, so in this case SIGINT won't be able to terminate the process.
- **What is the signal number that CTRL + C send?** The signal number of SIGINT (ctrl c) is 2.

5

- **What is the output of the program?**
count = 1 in handler 1
count = 10 in handler 2
count = 101 in main handler
- **Explain the output, specify how variable count is increased in child and parent processes.** In the main process signal() catches SIGUSR1 signals and calls the function handler1; this is a function that increments the global variable count (set to 0) by 1, prints out its value and sends the SIGUSR1 to the current process by using kill().
Then a process is created, and in the child process a new function, handler2, is set as the handler for the SIGUSR1 signal; then kill() sends the SIGUSR1 signal to the parent process, meaning handler1 is called since the signal is sent to the parent process. In handler 1 count is incremented from 0 to 1,

so the new value of count in the parent process will be 1. After printing out the value of count, the signal is sent to pid, which in this case is the child, so the handler2 function is called and adds 10 to count, which is 0 because the change made to count was in the parent process and not copied over to the child. I think the while(1); at the end of the child block is to make sure that the handler2 function has enough time to run before the main exits, I tried replacing while(1); with sleep(1) and it worked, so it seems like it makes sure that the child has enough time to finish executing the handler2 function before the program exits. At the end, add 100 to count and print out the value if the process is the parent; here the value will be 101 because count in the parent is 1 and 10 in the child.

7

- **What is the output if you run both at the same time calling shm_test1.c first?**

```
a_string = "I am a buffer in the shared memory area"
an_array[] = {42, 1, 4, 9, 16}
a_ptr = 140722425614928 = "I am a string allocated on main's stack!"
```

```
a_string = "I am a buffer in the shared memory area"
an_array[] = {42, 1, 4, 9, 16}
Segmentation fault (core dumped)
```

The only writing operation happening inside shm_test2 that conflicts with a writing operation inside shm_test1 is where index 0 of an_array[] is set; in shm_test2 it's set to 42, while it's value is 0x0 in shm_test1. Therefore when the shm_test2 is called last, it will overwrite the value 0 set by shm_test1 that was called first, to 42.

- **What is the output if you run both at the same time calling shm_test2.c first?**

```
a_string = "I am a buffer in the shared memory area"
an_array[] = {0, 1, 4, 9, 16}
Segmentation fault (core dumped)
```

```
a_string = "I am a buffer in the shared memory area"
an_array[] = {0, 1, 4, 9, 16}
a_ptr = 140722228167152 = "I am a string allocated on main's stack!"
```

When the shm_test1 is called last, it will overwrite the value 42 set by shm_test2 that was called first, to 0.

- **Why is shm_test2.c causing a segfault? How could this be fixed?** Because a_ptr is a pointer, and it will not have the same pointer address in the two separate processes created by shm_test1 and shm_test2. It's not advisable to use pointers in shared memory structures because of the fact that their addresses will be different in each implementation of the shared structure the pointer resides in. Here we must change char *a_ptr to char a_ptr[SIZE], and in the shm_test1 we have to replace the line shared_mem->a_ptr = my_string; to strcpy(shared_mem->a_ptr, my_string);
- **What happens if the two applications both try to read and set a variable at the same time? (e.g. shared mem → count++)?** Race condition, can be avoided by for example sharing a named semaphore between shm_test1 and shm_test2 in order to control the access of the critical blocks by using sem_wait and sem_post.
- **How can a shared memory space be deleted from the system?** shm_unlink("/ITF22519LabsIPC"); ITF22519LabsIPC must be replaced with the actual name, but in this task that is the name.

8

- **How long do semaphores last in the kernel?** POSIX named semaphores have kernel persistence: if not removed by sem_unlink(3), a semaphore will exist until the system is shut down.
- **What causes them to be destroyed?** sem_unlink(const char *name); or system shutdown.
- **What is the basic process for creating and using named semaphores? (list the functions**

that would need to be called and their order).

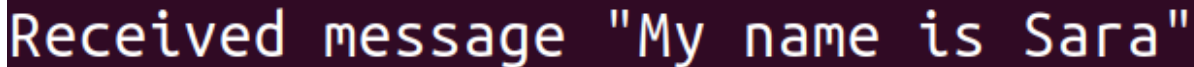
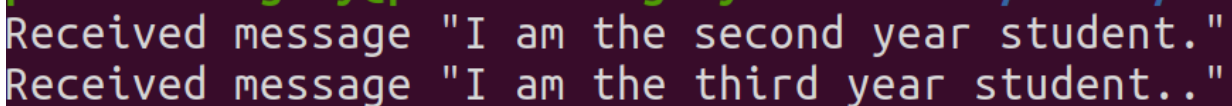
1. Create an uninitialized semaphore `sem_t *nameOfSem`
2. `nameOfSem = sem_open(const char *name, int oflag, mode_t mode, unsigned int value);`
OR
`nameOfSem = sem_open(const char *name, int oflag);`
3. Use `sem_wait(nameOfSem);` where you want to introduce the access lock of the critical block, and `sem_post(nameOfSem);` to unlock the `sem_wait(nameOfSem);`
4. `sem_close(nameOfSem);` to close the opened semaphore.
5. `sem_unlink(const char *name);` to delete the semaphore from the system.

10

- **What are the six types of sockets?**
SOCK_STREAM: Provides sequenced, reliable, two-way, connection-based byte streams. An out-of-band data transmission mechanism may be supported.
SOCK_DGRAM: Supports datagrams (connectionless, unreliable messages of a fixed maximum length).
SOCK_SEQPACKET: Provides a sequenced, reliable, two-way connection-based data transmission path for datagrams of fixed maximum length; a consumer is required to read an entire packet with each input system call.
SOCK_RAW: Provides raw network protocol access.
SOCK_RDM: Provides a reliable datagram layer that does not guarantee ordering.
SOCK_PACKET: Obsolete and should not be used in new programs; see `packet(7)`.
- **What are the two domains that can be used for local communications?** AF_UNIX, AF_LOCAL (synonym for AF_UNIX).

11

- **What is the output from each program?**

A terminal window with a dark background and light-colored text. The text reads: "Received message 'My name is Sara'".A terminal window with a dark background and light-colored text. The text reads: "Received message 'I am the second year student.'" followed by "Received message 'I am the third year student..'" on the next line.

The output in test1 is the string sent in test2, and the output in test2 is the two strings from test1. The reason the string that was sent second from test1 is written before the one that was sent first is because the second string has a higher priority.

- **What happens if you start them in the opposite order?** If I execute test2 and then test1 a couple of seconds after, the output doesn't change; if I wait some seconds more the order of the messages from test1 doesn't follow the correct priority, so string1 is printed before string2. I'm not sure which one is the outcome you're expecting in this question.

12

- **Do we need to synchronize the two programs to ensure read operations will always happen after write operations?** No, the read will wait for the write because `mq_receive()` blocks the program from executing any more code, and waits for the queue to be filled before unblocking the program and letting it continue executing the code below `mq_receive()`.
From the man page about `mq_receive`: If the queue is empty, then, by default, `mq_receive()` blocks until a message becomes available, or the call is interrupted by a signal handler.