

گزارش دستورکار چهارم آزمایشگاه سیستم‌های عامل

نگار موقتیان، ۹۸۳۱۰۶۲

۱. ارتباط فرآیندها از طریق حافظه مشترک

در این قسمت از آزمایش می‌خواهیم برنامه‌ای بنویسیم که در آن دو فرآیند مجزا (Reader و Writer) از طریق یک حافظه مشترک با یکدیگر ارتباط برقرار کنند. دسترسی به این قسمت از حافظه توسط یک کلید یکتا و مشترک میان دو پردازنده انجام می‌شود.

در این تمرین در استفاده از shared memory ۵ خانه اول استفاده شده‌اند. قرارداد می‌کنیم writer دو عدد خود را در دو خانه اول بنویسد. سپس پس از نوشتن این داده‌ها به نشانه valid بودن داده خانه شماره ۴ را ۱ کند. پس از آن reader حاصل جمع را در خانه ۴ حافظه نوشته و به نشانه valid بودن نتیجه خانه شماره ۵ را ۱ کند. یعنی به طور خلاصه:

1. First number
2. Second number
3. Sum result
4. Writer valid bit
5. Reader valid bit

در ادامه کد مربوط به فرآیند reader و توضیحات مربوط به آن آمده‌است.

در ابتدای برنامه کتابخانه‌های مورد نیاز اضافه شده‌اند.

سپس دو مقدار ثابت به عنوان طول حافظه مشترک (بر حسب تعداد کاراکتر) و کلید یکتای مربوط به این حافظه تعریف شده‌اند.

پس از آن در تابع main با استفاده از کلید و طول حافظه‌ای که تعریف کرده بودیم حافظه مشترک را تخصیص داده و ID آن را دریافت می‌کنیم. همچنین بررسی می‌کنیم اگر نتیجه این عملیات ناموفق بود پیغام مناسب چاپ شده و برنامه خاتمه یابد.

در ادامه یک رفرنس به حافظه مشترک را توسط تابع shmat در اختیار می‌گیریم و باز هم موفقیت آمیز بودن این عملیات را بررسی می‌کنیم.

حال تا زمانی که مطمئن شویم writer داده‌های خود را در حافظه نوشته منتظر می‌مانیم و سپس مقدار دو عدد نوشته شده را می‌خوانیم.

در نهایت حاصل جمع دو عدد خوانده شده را بر روی حافظه نوشته و valid بودن این حاصل جمع را اعلام می‌کنیم.

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<string.h>
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/shm.h>

#define MEMSIZE 64
#define KEY 1234
int memoryID;

int main() {
    // initiate the shared memory using the shared key defined above
    if ((memoryID = shmget(KEY, MEMSIZE * sizeof(int), IPC_CREAT | 0666)) < 0) {
        printf("ERR: Failed to allocate the shared memory.\n");
        return 1;
    }

    // retrieve the shared memory
    int *memory;
    if ((memory = shmat(memoryID, NULL, 0)) == (int *) -1) {
        printf("ERR: Failed to retrieve the shared memory.\n");
        return 1;
    }

    // get the numbers from the shared memory
    while (!memory[4]);
    printf("- Reader: Got numbers \"%d\" and \"%d\" from the shared memory.\n",
memory[0], memory[1]);

    // write the result to the shared memory
    int sum = memory[0] + memory[1];
    memory[3] = sum, memory[5] = 1;
    printf("- Reader: Just wrote the sum result \"%d\" to the shared memory.\n",
sum);

    return 0;
}
```

حال به بررسی کد مربوط به writer می‌پردازیم.

در ابتدای برنامه کتابخانه‌های مورد نیاز اضافه شده‌اند.

سپس مانند کد مربوط به فرآیند reader دو مقدار ثابت به عنوان طول حافظه مشترک (بر حسب تعداد کاراکتر) و کلید یکتای مربوط به این حافظه تعریف شده‌اند. به علاوه پیامی که قرار است بر روی حافظه مشترک قرار گیرد مشخص شده‌است.

پس از نیز درست مانند برنامه قبل در تابع main با استفاده از کلید و طول حافظه‌ای که تعریف کرده بودیم حافظه مشترک را تخصیص داده و ID آن را دریافت می‌کنیم. همچنین بررسی می‌کنیم اگر نتیجه این عملیات ناموفق بود پیغام مناسب چاپ شده و برنامه خاتمه یابد. در ادامه نیز یک رفرنس به حافظه مشترک را توسط تابع shmat در اختیار می‌گیریم و باز هم موفقیت آمیز بودن این عملیات را بررسی می‌کنیم.

پس از آن دو عدد مورد نظر را طبق قرارداد بالا بر روی حافظه نوشته و valid بودن آن‌ها را اعلام می‌کنیم.

سپس تا زمانی که reader حاصل جمع دو عدد را بر روی حافظه بنویسد منتظر می‌مانیم و پس از آن داده مورد نظر را خوانده و چاپ می‌کنیم.

حال کار هر دو پردازش با حافظه مشترک تمام شده‌است، بنابراین این حافظه را آزاد می‌کنیم تا پردازش‌های دیگر بتوانند در صورت نیاز از آن استفاده کنند.

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/shm.h>

#define MEMSIZE 64
#define KEY 1234
int memoryID;
int firstNumber = 8, secondNumber = 17;

int main() {
    // initiate the shared memory using the shared key defined above
    if ((memoryID = shmget(KEY, MEMSIZE * sizeof(int), IPC_CREAT | 0666)) < 0) {
        printf("ERR: Failed to allocate the shared memory.\n");
        return 1;
    }

    // retrieve the shared memory
    int *memory;
```

```

    if ((memory = shmat(memoryID, NULL, 0)) == (int *) -1) {
        printf("ERR: Failed to retrieve the shared memory.\n");
        return 1;
    }

    // write the numbers to the shared memory
    memory[0] = firstNumber, memory[1] = secondNumber;
    printf("+ Writer: Just wrote \"%d\" and \"%d\" to the shared memory.\n", firstNumber,
secondNumber);
    memory[4] = 1;

    while (!memory[5]);

    // print the result and free the shared memory
    printf("+ Writer: Got result %d from the reader, freeing the shared memory...\n",
memory[3]);
    shmctl(memoryID, IPC_RMID, NULL);

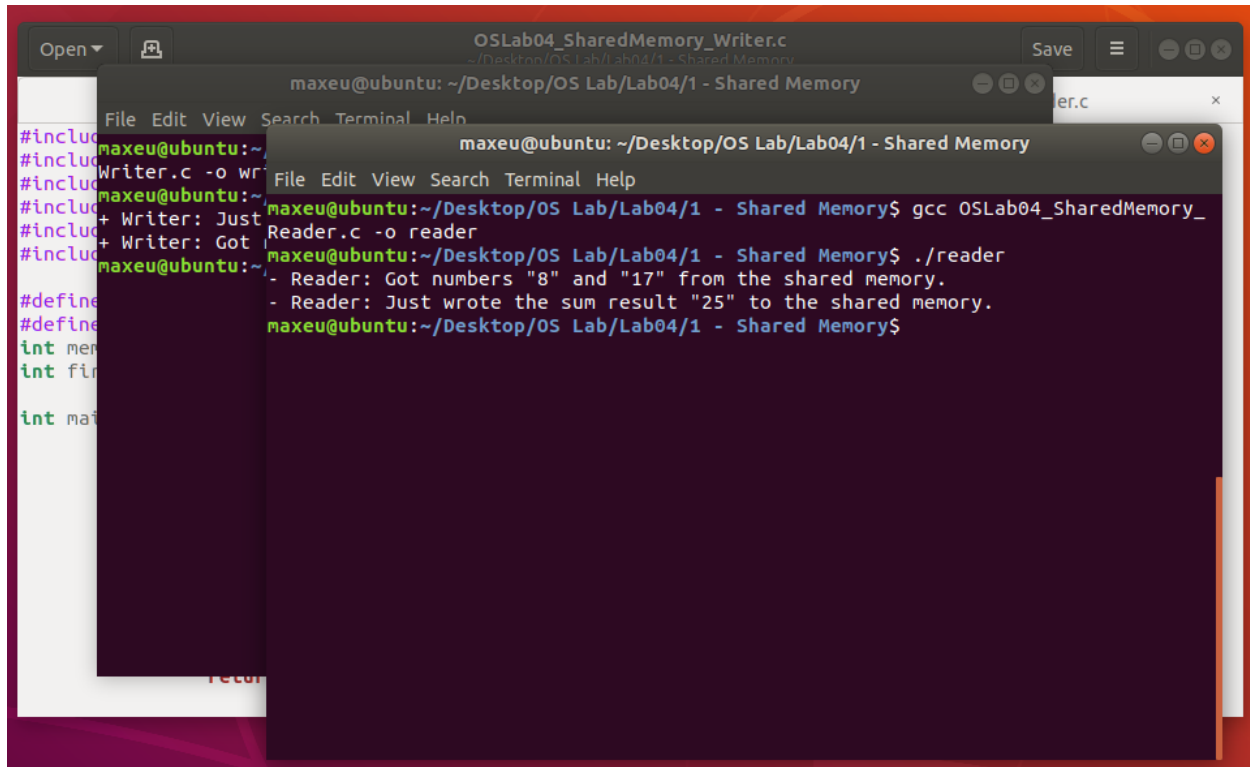
    return 0;
}

```

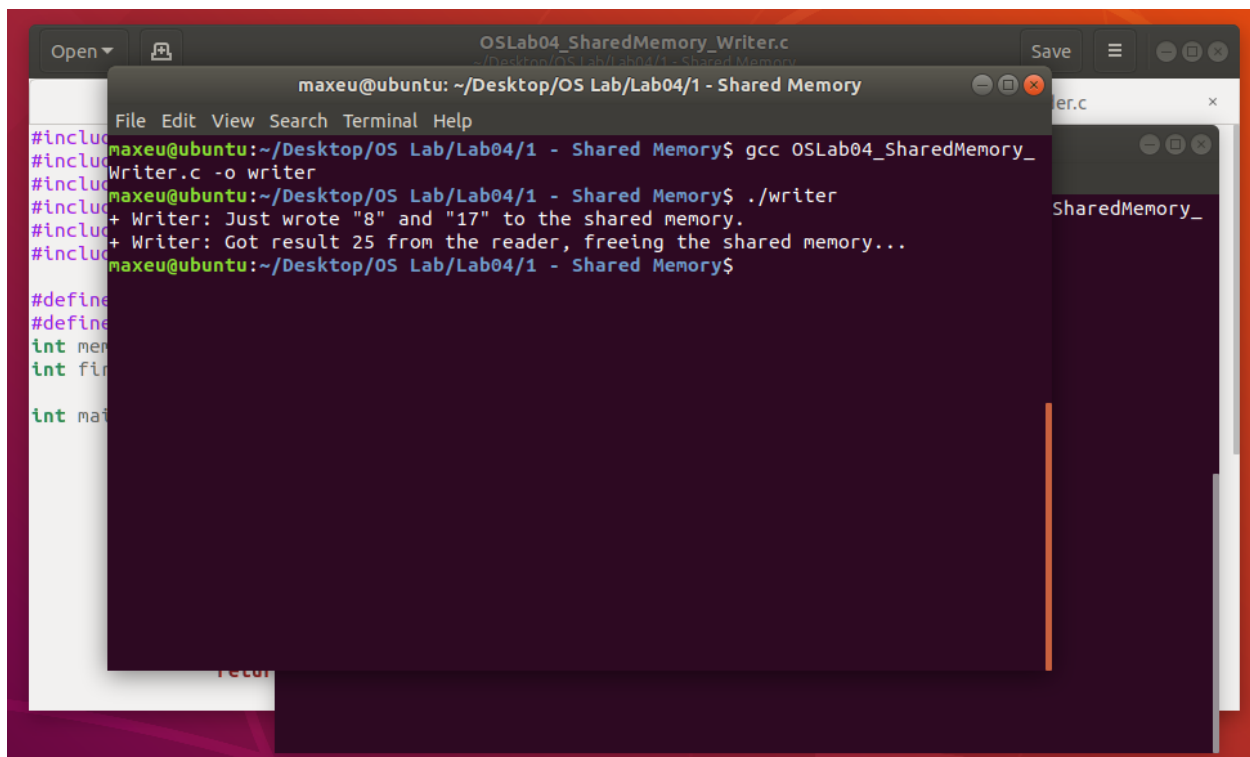
حال می‌توانیم برنامه‌ها را اجرا کنیم. داده ابتدا باید بر روی حافظهٔ مشترک نوشته شود تا بتواند توسط reader خوانده شود. بنابراین ابتدا کد مربوط به فرآیند writer را کامپایل و اجرا می‌کنیم.

The screenshot shows a terminal window titled "maxeu@ubuntu: ~/Desktop/OS Lab/Lab04/1 - Shared Memory". The user has compiled the program "OSLab04_SharedMemory_Writer.c" into an executable named "writer" using the command `gcc OSLab04_SharedMemory_Writer.c -o writer`. Then, they executed the program with `./writer`, which printed the output: `+ Writer: Just wrote "8" and "17" to the shared memory.`

سپس کد مربوط به فرآیند reader را کامپایل و اجرا می‌کنیم تا داده نوشته شده توسط writer را بخواند.



```
OSLab04_SharedMemory_Writer.c
~/Desktop/OS Lab/Lab04/1 - Shared Memory
maxeu@ubuntu: ~/Desktop/OS Lab/Lab04/1 - Shared Memory
File Edit View Search Terminal Help
maxeu@ubuntu:~/Desktop/OS Lab/Lab04/1 - Shared Memory$ gcc OSLab04_SharedMemory_
+ Writer: Just
Reader.c -o reader
+ Writer: Got
maxeu@ubuntu:~/Desktop/OS Lab/Lab04/1 - Shared Memory$ ./reader
- Reader: Got numbers "8" and "17" from the shared memory.
- Reader: Just wrote the sum result "25" to the shared memory.
maxeu@ubuntu:~/Desktop/OS Lab/Lab04/1 - Shared Memory$
```



```
OSLab04_SharedMemory_Writer.c
~/Desktop/OS Lab/Lab04/1 - Shared Memory
maxeu@ubuntu: ~/Desktop/OS Lab/Lab04/1 - Shared Memory
File Edit View Search Terminal Help
maxeu@ubuntu:~/Desktop/OS Lab/Lab04/1 - Shared Memory$ gcc OSLab04_SharedMemory_
Writer.c -o writer
maxeu@ubuntu:~/Desktop/OS Lab/Lab04/1 - Shared Memory$ ./writer
+ Writer: Just wrote "8" and "17" to the shared memory.
+ Writer: Got result 25 from the reader, freeing the shared memory...
maxeu@ubuntu:~/Desktop/OS Lab/Lab04/1 - Shared Memory$
```

همانطور که مشاهده می‌شود reader اعداد writer را به درستی خوانده و حاصل جمع آن‌ها را برای writer فرستاده‌است. writer نیز حاصل جمع را دریافت کرده، حافظه مشترک را آزاد کرده و کار خود را خاتمه می‌دهد.

۲. ارتباط فرآیندها از طریق سیستم Client-Server

* به دلیل انجام این آزمایش پیش از جلسه ۱۹ فروردین، برنامه نوشته شده برای این قسمت مطابق متن اصلی دستورکار (پیاده‌سازی یک chat application که در آن امکان مکالمه گروهی وجود دارد) می‌باشد.

در این قسمت از آزمایش می‌خواهیم برنامه یک chat application را بنویسیم که در آن دو فرآیند مجزا (Client و Server) از طریق سیستم خادم-خدمتگذار و با استفاده از socket programming با یکدیگر ارتباط برقرار می‌کنند. در این روش یک سوکت مشترک میان دو پردازش ایجاد می‌شود و پردازش‌ها می‌توانند بر روی این سوکت داده‌ای را نوشته و یا از روی آن داده‌ای را بخوانند.

* به دلیل طولانی بودن کد این بخش از آزمایش، کد به صورت مستقیم در اینجا آورده نمی‌شود و تنها توضیحات مربوط به آن داده می‌شود. همچنین اسکرین‌شات‌های مربوط به این قسمت در پوشه Client_Server “Screenshots ارسال می‌گردد.

در کد مربوط به client در ابتدا با اجرای برنامه ۳ آرگومان را به عنوان ورودی از کاربر می‌گیریم:

۱. Server Host Name که آدرس سرور بر روی شبکه می‌باشد.

۲. Server Port Number که شماره پورتی که سرور بر روی آن در حال اجراست می‌باشد.

۳. Client Name که در حقیقت نام کاربری‌ای است که client از طریق آن شناخته می‌شود.

پس از آن مطابق کد موجود در دستورکار سوکت مربوطه را ایجاد کرده و به سرور متصل می‌شویم. حال می‌توانیم برای سرور داده‌ای را ارسال کرده یا داده ارسال شده توسط آن را بخوانیم.

از آن جایی که ورودی گرفتن از کاربر و آمدن داده جدید از سرور ممکن است به طور همزمان و در هر لحظه انجام شود ناچاریم از یک thread مجزا برای انجام یکی از این دو کار استفاده کنیم تا بتوانند به صورت همروند اجرا شوند.

بنابراین یک thread جدید ایجاد شده‌است که مسئول گرفتن داده‌ها از سرور است (از طریق تابع readUtil). این داده ممکن است نتیجه عملیاتی باشد که کاربر آن را درخواست داده و یا پیام جدید باشد که از طرف یکی از

گروه‌هایی که در آن عضو است آمده. دریافت این داده‌ها از طریق سوکت نیز درست مانند آنچه در دستورکار آمده انجام می‌شود.

پردازه اصلی نیز در ابتدا یک بار Client Name را برای سرور می‌فرستد تا سرور پس از این او را با این نام بشناسد. سپس در هر مرحله یک ورودی از کاربر گرفته و آن را برای سرور ارسال می‌کند. پردازش دستورات در سمت سرور انجام می‌گیرد و در سمت کاربر تنها دستور quit بررسی می‌شود، تا در صورت آمدن آن اجرای برنامه را متوقف کنیم.

حال به بررسی کد مربوط به writer می‌پردازیم.

در ابتدای این برنامه دو مقدار ثابت با عنوان MAXGROUPS، حداکثر تعداد گروه‌ها و MAXUSERS، حداکثر تعداد کاربران هر گروه تعریف شده‌اند.

پس از آن یک struct با نام group ایجاد کرده‌ایم که اطلاعات مربوط به هر گروه را ذخیره‌سازی می‌کند. این اطلاعات عبارت‌اند از ID گروه، لیست سوکت‌های کاربران موجود در گروه، لیست نام کاربری کاربران موجود در گروه و آرایه‌ای که مشخص می‌کند هر کدام از اندیس‌های لیست‌های بالا پر است یا خالی (مقدار 0 به معنای پر و مقدار 1 به معنای خالی بودن آن است).

پس از آن توابع مختلف برای مدیریت اضافه شدن کاربران به گروه، ترک کردن گروه و فرستادن پیام به گروه نوشته شده‌اند (این توابع کامنت گذاری شده‌اند تا مشخص باشد در هر مرحله چه عملی انجام می‌شود). انجام عملیات در هر یک از این توابع می‌تواند موفقیت آمیز باشد یا با مشکلی مواجه شود، بنابراین در صورت موفقیت‌آمیز بودن مقدار 0 و در غیر این صورت مقدار 1- برگردانده می‌شود تا پاسخ مناسب را بر حسب آن به کلاینت ارسال کنیم.

در تابع main نیز مانند کد نوشته شده در client آرگومان ورودی که شماره پورت سرور است را بررسی می‌کنیم. سپس آرایه گروه‌ها را با مقادیر پیش فرض مقداردهی می‌کنیم.

پس از آن مانند دستور کار ابتدا ارتباط را برقرار کرده و سپس روی پورت مربوطه گوش می‌کنیم و منتظر می‌مانیم تا یک client به سرور متصل شود. حال تا زمانی که برنامه terminate شود کلاینت‌های جدید را accept کرده و به ازای هر یک از آن‌ها یک thread می‌سازیم که مسئول مدیریت آن کلاینت خاص است.

مدیریت کلاینت‌ها در تابع clientHandler انجام می‌شود. ورودی این تابع شماره سوکت مربوط به کلاینت است. پس از این می‌توانیم از طریق این سوکت داده‌هایی را از کلاینت دریافت کرده یا برای او ارسال کنیم.

در شروع نام‌کاری را از کلاینت دریافت کرده و ذخیره می‌کنیم. پس از آن تا زمانی که کلاینت دستور quit را اجرا نکرده از او query ای که کاربر وارد کرده را دریافت می‌کنیم. سپس قسمت دستور را از این query جدا می‌کنیم. برای آشنایی با نحوه tokenize کردن رشته‌ها از لینک زیر کمک گرفته شده‌است:

<https://stackoverflow.com/questions/266357/tokenizing-strings-in-c>

حال بررسی می‌کنیم که دستور وارد شده کدام یک از دستورات تعریف شده‌است و بر اساس آن باقی آرگومان‌ها را نیز جدا کرده و تابع مربوطه را صدا می‌زنیم. در نهایت نیز بر اساس مقدار برگردانده شده توسط تابع فوق یک متن به عنوان پاسخ برای کلاینت تنظیم کرده و آن را برای کلاینت ارسال می‌کنیم.

ارتباط فرایندها از طریق خط لوله

در این قسمت از آزمایش می‌خواهیم برنامه‌ای بنویسیم که در آن دو فرآیند مجزا (Parent و Child) از طریق دو خط لوله با یکدیگر ارتباط برقرار کنند. در این برنامه فرآیند پدر توسط یک خط لوله پیغامی را به فرآیند فرزند می‌فرستد، سپس فرآیند فرزند پیغام را پردازش کرده و پاسخ آن را بر روی خط لوله دیگری می‌نویسد تا فرآیند پدر آن را بخواند.

در ادامه کد مربوط به برنامه و توضیحات مربوط به آن آمده‌است.

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<sys/types.h>
#include<string.h>

void toggleCase(char *str) {
    for (int i = 0; str[i] != '\0'; i++)
        str[i] += (str[i] >= 'a' && str[i] <= 'z') ? -32 : (str[i] >= 'A' && str[i] <= 'Z') ? 32 : 0;
}

#define MSGCNT 4
int mainPipe[2], toggledPipe[2]; // 1 -> write, 0 -> read
pid_t pid;
```



```

int main() {
    // initiate pipelines
    if (pipe(mainPipe) < 0 || pipe(toggledPipe) < 0) {
        printf("ERR: Failed to initiate pipes.\n");
        return 1;
    }

    // create the child process
    if ((pid = fork()) < 0) {
        printf("ERR: Failed to fork child process.\n");
        return 1;
    }

    if (pid > 0) {
        // parent process will only write to the "main pipe" and read from the
        "toggled pipe"
        close(mainPipe[0]);
        close(toggledPipe[1]);

        char* messages[MSGCNT] = {
            "Hello!",
            "This IS a mESSAGE",
            "from Parent PROCESS to CHILD process",
            "hope You're doing ALright"
        };

        for (int i = 0; i < MSGCNT; i++) {
            printf("+ Sending: %s\n", messages[i]);
            write(mainPipe[1], messages[i], strlen(messages[i]) + 1);

            char buff[128];
            int bytes = read(toggledPipe[0], buff, sizeof(buff));
            printf("- Received: %s\n", buff);
        }
    }
    else {
        // child process will only read from the "main pipe" and write to the
        "toggled pipe"
        close(mainPipe[1]);
        close(toggledPipe[0]);

        for (int i = 0; i < MSGCNT; i++) {
            char buff[128];
            int bytes = read(mainPipe[0], buff, sizeof(buff));

            toggleCase(buff);
            write(toggledPipe[1], buff, strlen(buff) + 1);
        }
    }

    return 0;
}

```

در ابتدای برنامه کتابخانه‌های مورد نیاز اضافه شده‌اند.

سپس تابعی تعریف شده‌است که با استفاده از روابط میان کاراکترهای ASCII حروف بزرگ یک رشته را به حروف کوچک و حروف کوچک آن را به حروف بزرگ تبدیل می‌کند.

پس از آن تعداد پیام‌ها مشخص شده و دو آرایه نماینده خط لوله‌ها ایجاد شده‌اند (اندیس 0 آن‌ها برای خواندن از خط لوله و اندیس 1 آن‌ها برای نوشتن بر روی آن به کار می‌روند). برای این کار نیاز به دو خط لوله داریم زیرا خط لوله به صورت یک طرفه عمل می‌کند و دو فرآیند نمی‌توانند به طور همزمان بر روی آن داده بنویسند و بخوانند.

در تابع main ابتدا خط لوله‌ها را راه‌اندازی کرده و فرایند فرزند را fork می‌کنیم.

۱. قسمت مربوط به فرآیند پدر: فرآیند پدر تنها بر روی mainPipe نوشته و از روی toggledPipe داده را می‌خواند، بنابراین قسمتی از خط لوله که به آن نیاز نداریم (mainPipe[0] و toggledPipe[1]) را در این فرآیند می‌بندیم.

سپس پیغام‌هایی که می‌خواهیم برای فرآیند فرزند ارسال کنیم را تعریف کرده و یکی یکی آن‌ها را بر روی mainPipe نوشته و نتیجه را از روی toggledPipe می‌خوانیم.

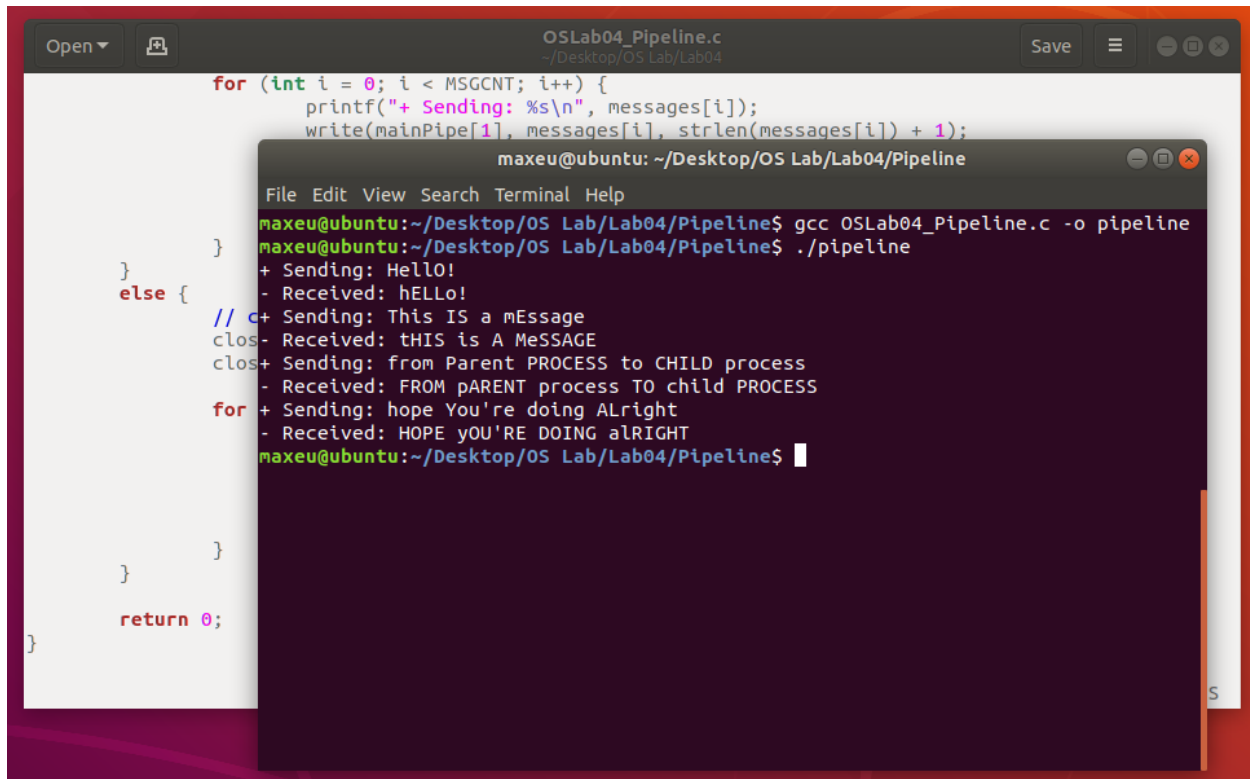
۲. قسمت مربوط به فرآیند فرزند: در این قسمت نیز ابتدا قسمتی از خط لوله که به آن نیاز نداریم (mainPipe[1] و toggledPipe[0]) را می‌بندیم، زیرا فرآیند فرزند تنها بر روی toggledPipe نوشته و از روی mainPipe داده را می‌خواند.

سپس به تعداد پیغام‌ها آن‌ها را از روی mainPipe خوانده، از طریق تابع toggle حروف آن را برعکس کرده و نتیجه را بر روی toggledPipe می‌نویسیم.

برای آشنایی با نحوه ایجاد و کارکردن با pipeline ها در زبان C از لینک زیر کمک گرفته شده‌است:

<https://www.geeksforgeeks.org/pipe-system-call/>

خروجی این برنامه در شکل زیر آمده‌است.



The image shows a code editor window titled "OSLab04_Pipeline.c" with the following C code:

```
for (int i = 0; i < MSGCNT; i++) {  
    printf("+ Sending: %s\n", messages[i]);  
    write(mainPipe[1], messages[i], strlen(messages[i]) + 1);  
}  
}  
else {  
    // c  
    clos+ Sending: This IS a mESsage  
    clos- Received: tHIS is A MeSSAGE  
    clos+ Sending: from Parent PROCESS to CHILD process  
    - Received: FROM pARENT process TO child PROCESS  
    for+ Sending: hope You're doing ALright  
    - Received: HOPE you'RE DOING alRIGHT  
    maxeu@ubuntu:~/Desktop/OS Lab/Lab04/Pipeline$  
  
    }  
}  
return 0;  
}
```

Overlaid on the code editor is a terminal window titled "maxeu@ubuntu: ~/Desktop/OS Lab/Lab04/Pipeline". The terminal shows the compilation and execution of the program:

```
maxeu@ubuntu:~/Desktop/OS Lab/Lab04/Pipeline$ gcc OSLab04_Pipeline.c -o pipeline  
maxeu@ubuntu:~/Desktop/OS Lab/Lab04/Pipeline$ ./pipeline  
+ Sending: Hello!  
- Received: hELLO!  
+ Sending: This IS a mESsage  
- Received: tHIS is A MeSSAGE  
+ Sending: from Parent PROCESS to CHILD process  
- Received: FROM pARENT process TO child PROCESS  
+ Sending: hope You're doing ALright  
- Received: HOPE you'RE DOING alRIGHT  
maxeu@ubuntu:~/Desktop/OS Lab/Lab04/Pipeline$
```