**گزارش دستورکار چهارم آزمایشگاه سیستم­های عامل**

نگار موقتیان، 9831062

1. **ارتباط فرآیندها از طریق حافظۀ مشترک**

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

در این تمرین در استفاده از shared memory 5 خانۀ اول استفاده شده­اند. قرارداد می­کنیم writer دو عدد خود را در دو خانۀ اول بنویسد. سپس پس از نوشتن این داده­ها به نشانۀ valid بودن داده خانۀ شماره 4 را 1 کند. پس از آن reader حاصل جمع را در خانۀ 4 حافظه نوشته و به نشانۀ valid بودن نتیجه خانۀ شماره 5 را 1 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 حافظۀ 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 حافظۀ مشترک را تخصیص داده و ID آن را دریافت می­کنیم. همچنین بررسی می­کنیم اگر نتیجۀ این عملیات ناموفق بود پیغام مناسب چاپ شده و برنامه خاتمه یابد.

در ادامه یک رفرنس به حافظۀ 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 حافظۀ مشترک (بر حسب تعداد کاراکتر) و کلید یکتای مربوط به این حافظه تعریف شده­اند. به علاوه پیامی که قرار است بر روی حافظۀ مشترک قرار گیرد مشخص شده­است.

پس از نیز درست مانند برنامۀ قبل در تابع main با استفاده از کلید و طول حافظه­ای که تعریف کرده بودیم حافظۀ مشترک را تخصیص داده و ID آن را دریافت می­کنیم. همچنین بررسی می­کنیم اگر نتیجۀ این عملیات ناموفق بود پیغام مناسب چاپ شده و برنامه خاتمه یابد. در ادامه نیز یک رفرنس به حافظۀ 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 دادۀ مورد نظر را خوانده و چاپ می­کنیم.

حال کار هر دو پردازه با حافظۀ مشترک تمام شده­است، بنابراین این حافظه را آزاد می­کنیم تا پردازه­های 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 reuslt and free the shared memory

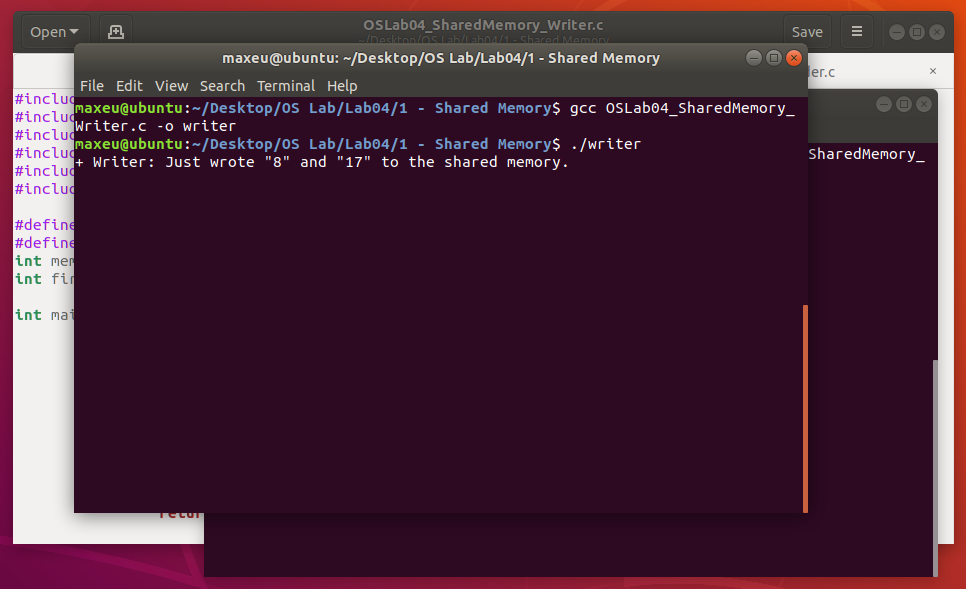
printf("+ Writer: Got result %d from the reader, freeing the shared memory...\n", memory[3]);

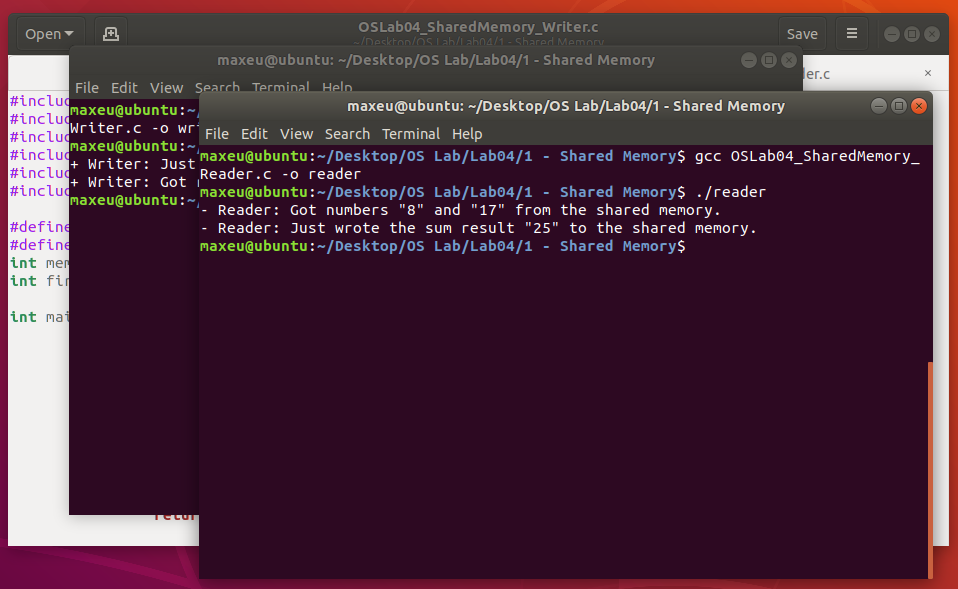
shmctl(memoryID, IPC\_RMID, NULL);

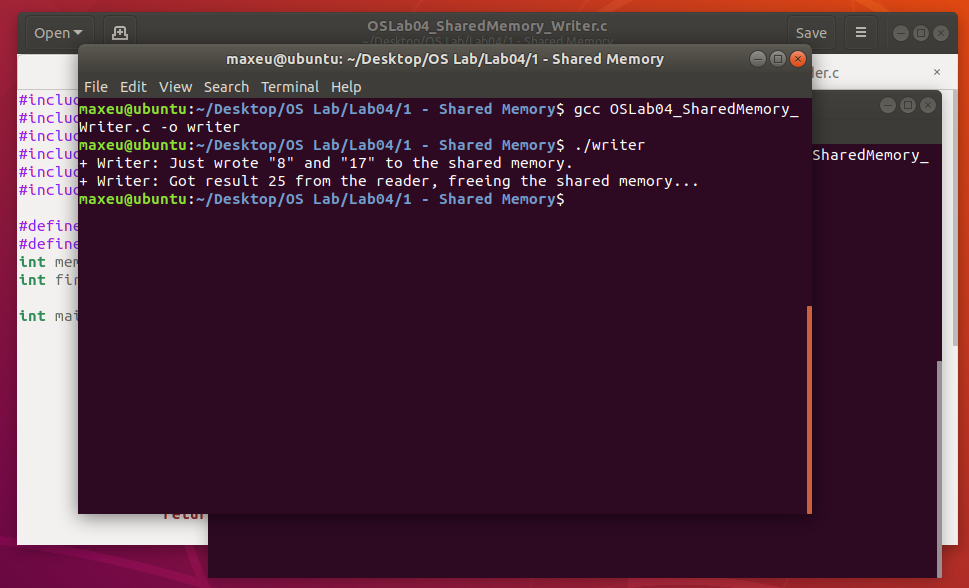
return 0;

}

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



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



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، حافظۀ مشترک را آزاد کرده و کار خود را خاتمه می­دهد.

1. **ارتباط فرآیندها از طریق سیستم Client-Server**

\* به دلیل انجام این آزمایش پیش از جلسۀ 19 فروردین، برنامۀ نوشته شده برای این قسمت مطابق متن اصلی دستورکار (پیاده­سازی یک chat application که در آن امکان مکالمۀ گروهی وجود دارد) می­باشد.

در این قسمت از آزمایش می­خواهیم برنامۀ 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 پردازه ایجاد می­شود و پردازه­ها 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 پوشۀ “Client\_Server Screenshots” ارسال می­گردد.

در کد مربوط به client در ابتدا با اجرای برنامه 3 آرگومان را به عنوان ورودی از کاربر می­گیریم:

1. Server Host Name که آدرس سرور بر روی شبکه می­باشد.
2. Server Port Number که شماره پورتی که سرور بر روی آن در حال اجراست می­باشد.
3. 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 دادۀ ارسال شده توسط آن را بخوانیم.

از آن­جایی که ورودی گرفتن از کاربر و آمدن دادۀ 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 نتیجۀ 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.

پردازۀ 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 موفقت­آمیز 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 شمارۀ پورت سرور است را بررسی می­کنیم. سپس آرایۀ 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 نام­کاربری 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 نحوۀ 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 فرآیندها 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 لولۀ 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 آرایۀ نمایندۀ 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.

1. 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.

1. 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 نحوۀ 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.

