



دانشگاه صنعتی امیرکبیر
(پلی تکنیک تهران)

گزارش کار آزمایشگاه سیستم عامل – شماره ۵

استفاده از مکانیزم های ارتباط بین فرآیندها

حسنا اویارحسینی – ۹۸۲۳۰۱۰

استاد درس: جناب آقای مهندس کیخا

نیمسال دوم سال تحصیلی ۱۴۰۰-۰۱

بخش ۱) ارتباط دو فرآیند از طرق حافظه مشترک:

```
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
int main()
{
    key_t key = ۲۵۲۵;;
    // shmget returns an identifier in shmid
    int shmid = shmget(key, sizeof(int) * ۲, ۰۶۶۶|IPC_CREAT);
    // shmat to attach to shared memory
    int *nums = (int*) shmat(shmid, (void*)۰, ۰);
    printf("Write Data : ");
    scanf("%d", &nums[۰]);

    printf("Write Data : ");
    scanf("%d", &nums[۱]);
    printf("Data written in memory: %d, %d\n", nums[۰], nums[۱]);

    //detach from shared memory
    shmdt(nums);
    return ۰;
}
```

برای انجام اینکار دو برنامه یکی برای read و یکی برای write مینویسیم. ابتدا به توضیح برنامه write می‌پردازیم.

در این برنامه ابتدا یک حافظه مشترک با کلید ۲۵۲۵ ساخته میشود سپس این حافظه مشترک در قالب `int*` یعنی در واقع یک آرایه به طول ۲ به متغیر `nums` نسبت داده می‌شود و در نهایت دو عدد از ورودی گرفته شده و مقدار هر یک در یکی از دوخانه آرایه `nums` نوشته میشود. به این صورت دو عدد در داخل حافظه مشترک ثبت میشود.

در قسمت بعد به توضیح کد مربوط به پردازش read می‌پردازیم:

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

int main()
{
    key_t key = ۲۵۲۵;

    // shmget returns an identifier in shmid
    int shmid = shmget(key, sizeof(int) * ۲, ۰۶۶۶|IPC_CREAT);

    // shmat to attach to shared memory
    int *nums = (int*) shmat(shmid, (void*)۰, ۰);
```

```

printf("Sum of data read from memory: %d\n", nums[0] + nums[1]);

//detach from shared memory
shmdt(nums);

// destroy the shared memory
shmctl(shmid,IPC_RMID,NULL);

return 0;
}

```

در این کد نیز مجددا همانند کد قبلی یک حافظه مشترک با key ۲۵۲۵ ساخته میشود که چون کلید این حافظه با کلید حافظه برنامه قبلی یکسان است این دو حافظه به یک محل اشاره می کنند. سپس این حافظه مشترک در قالب `int*` به متغیر `nums` نسبت داده می شود و در نهایت به کمک `printf` مقدار دو عددی که در محتوای حافظه مشترک باشد خوانده و حاصل جمع آنها نمایش داده میشود.

در ادامه خروجی این برنامه را مشاهده میکنید که ابتدا ۲ عدد در پردازش `write` نوشته شده و سپس پردازش `read` اجرا شده و ۲ عدد را از روی حافظه مشترک می خواند و حاصل جمع آنها را چاپ میکند:

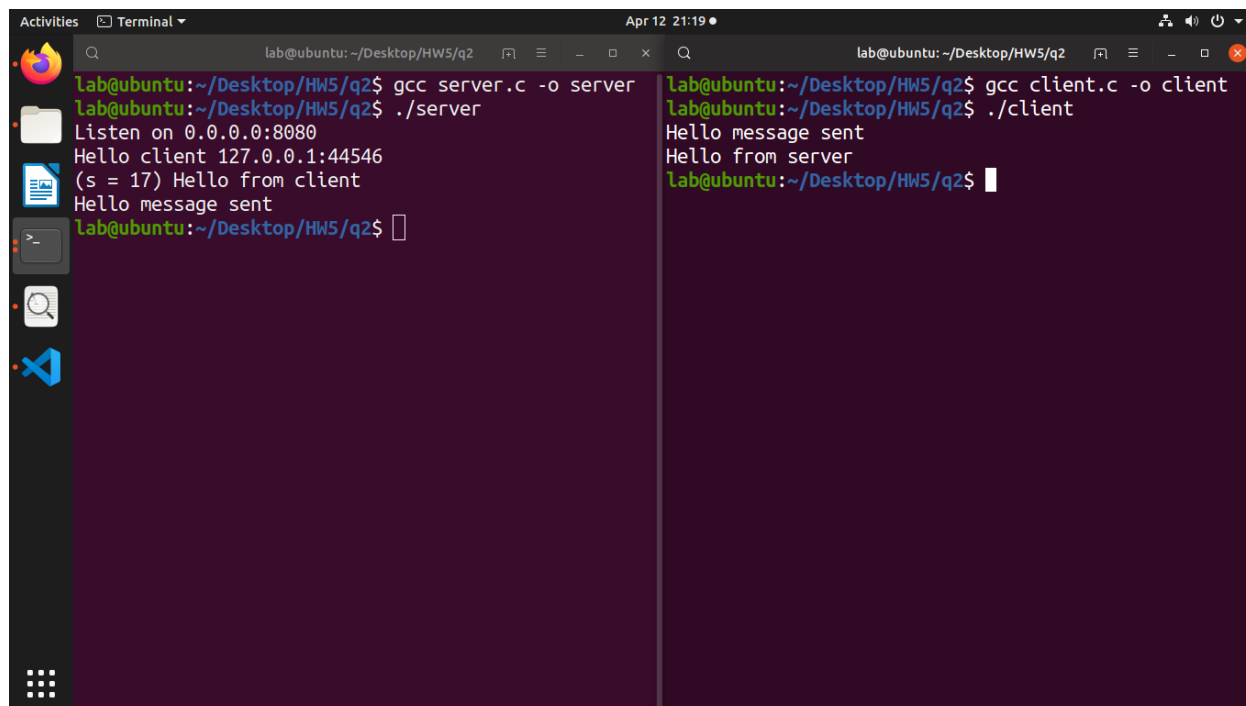
```

lab@ubuntu: ~/Desktop/HW5/q1
lab@ubuntu:~/Desktop/HW5/q1$ gcc writer.c -o writer
lab@ubuntu:~/Desktop/HW5/q1$ ./writer
Write Data : 2
Write Data : 3
Data written in memory: 2, 3
lab@ubuntu:~/Desktop/HW5/q1$ gcc reader.c -o reader
lab@ubuntu:~/Desktop/HW5/q1$ ./reader
Sum of data read from memory: 5
lab@ubuntu:~/Desktop/HW5/q1$

```

بخش ۲) اجرای قطعه کد Server و Client:

کد مربوط به کلاینت و سرور که به کمک حافظه مشترک نوشته شده و در دستور کار وجود دارد را اجرا میکنیم و خروجی را در ادامه مشاهده میکنیم:



```
lab@ubuntu: ~/Desktop/HWS/q2
lab@ubuntu:~/Desktop/HWS/q2$ gcc server.c -o server
lab@ubuntu:~/Desktop/HWS/q2$ ./server
Listen on 0.0.0.0:8080
Hello client 127.0.0.1:44546
(s = 17) Hello from client
Hello message sent
lab@ubuntu:~/Desktop/HWS/q2$

lab@ubuntu:~/Desktop/HWS/q2$ gcc client.c -o client
lab@ubuntu:~/Desktop/HWS/q2$ ./client
Hello message sent
Hello from server
lab@ubuntu:~/Desktop/HWS/q2$
```

بخش ۳) ارتباط دو پردازنده به کمک پایپ لاین:

کد:

```
int main(int argc, char *argv[])
{
    if (argc == 1)
    {
        printf("Enter some string after command!");
        return 1;
    }

    // create pipe:

    int descriptors[2];

    // read:[0] - write:[1]
    if (pipe(descriptors) != 0)
    {
        fprintf(stderr, "pipes failed!\n");
        return 1;
    }

    // fork() child process
```

```

int child = fork();

if (child < 0)
{
    fprintf(stderr, "fork failed!");
    return 1;
}
else if (child != 0)    /* parent */
{
    while (wait(NULL) > 0);

    // close unwanted pipe (write)
    close(descriptors[1]);

    // read from pipe
    char pipelineText[100];
    int n = read(descriptors[0], pipelineText, sizeof(pipelineText) -
1);

    pipelineText[n] = '\0';
    printf("Parent: read from pipe '%s'\n", pipelineText);
    close(descriptors[0]);

    for (int i = 0; pipelineText[i] != '\0'; i++)
    {
        if(pipelineText[i] >= 'a' && pipelineText[i] <= 'z')
        {
            pipelineText[i] = pipelineText[i] - 32;
        }
        else if(pipelineText[i] >= 'A' && pipelineText[i] <= 'Z')
        {
            pipelineText[i] = pipelineText[i] + 32;
        }
    }

    printf("Parent: after processing: %s\n", pipelineText);

}

close(descriptors[0]);

// write from terminal to pipe
printf("Child: writing to pipe '%s'\n", argv[1]);
write(descriptors[1], argv[1], strlen(argv[1]));
close(descriptors[1]);

}
return 0;
}

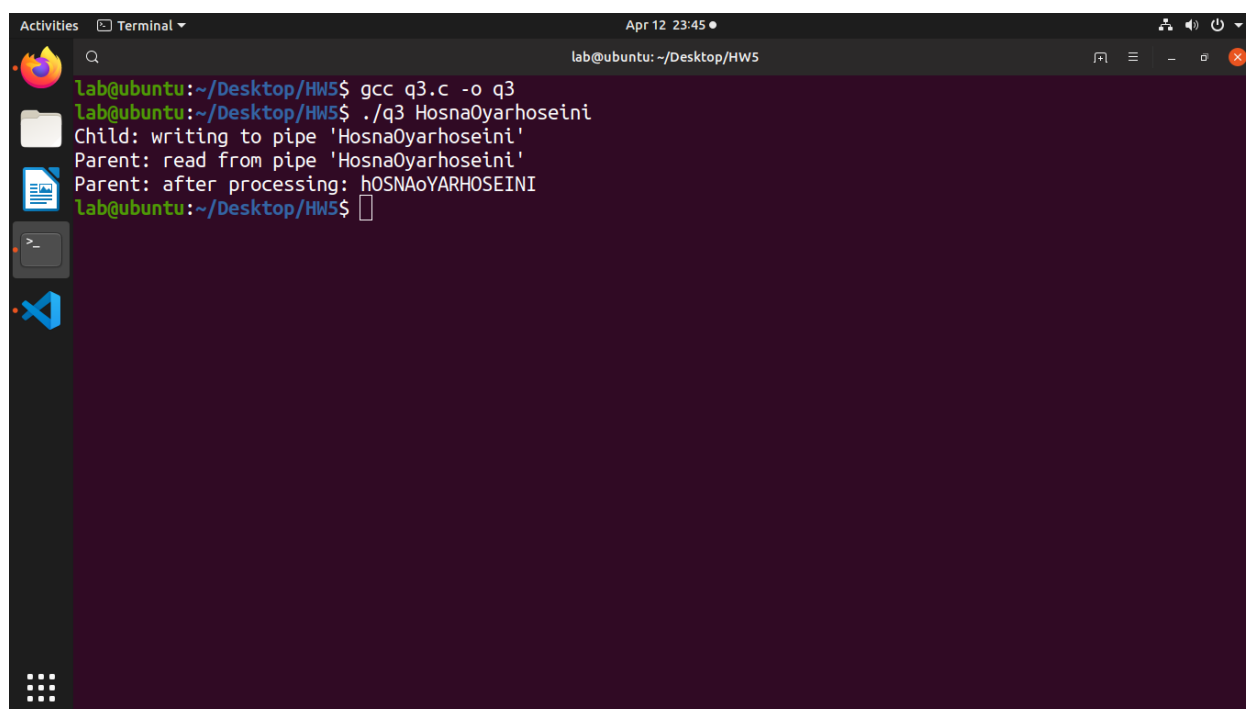
```

پردازش رشته

برای ایجاد این ارتباط یک پایپ لاین میسازیم، میدانیم که هر پایپ لاین دو descriptors به ما میدهد که یکی برای نوشتن در پایپ لاین و دیگری برای خواندن از آن است.

سپس به کمک دستور `fork()` یک پردازش فرزند میسازیم و در آن ابتدا `descriptors` مربوط به خواندن را میبندیم زیرا از طریق پردازش فرزند فقط میخواهیم در پایپ لاین بنویسیم، سپس به کمک دستور `write` رشته ای را که به عنوان آرگومان به برنامه داده شده است را در پایپ لاین مینویسیم. در پردازش پدر ابتدا صبر میکنیم کار فرزند که نوشتن در پایپ لاین باشد تمام شود سپس `descriptors` مربوط به نوشتن را بسته و به کمک دستور `read` و `descriptors` مربوط به آن محتوای موجود در پایپ لاین را میخوانیم و در `pipelineText` میریزیم، در نهایت نیز روی رشته پیمایش کرده و با توجه به کد اسکی هر کاراکتر تشخیص میدهیم که کاراکتر حروف بزرگ است یا کوچک و آن را تغییر میدهیم و نتیجه نهایی را چاپ میکنیم.

خروجی این قطعه کد را در ادامه مشاهده میکنید:



```
lab@ubuntu: ~/Desktop/HW5
lab@ubuntu:~/Desktop/HW5$ gcc q3.c -o q3
lab@ubuntu:~/Desktop/HW5$ ./q3 HosnaOyarhoseini
Child: writing to pipe 'HosnaOyarhoseini'
Parent: read from pipe 'HosnaOyarhoseini'
Parent: after processing: hOSNAoYARHOSEINI
lab@ubuntu:~/Desktop/HW5$
```

روش دیگری که برای این بخش وجود دارد استفاده از دو پایپ لاین است بدین صورت که پردازش ۱ رشته ای را در پایپ لاین ۱ مینویسد سپس پردازش ۲ این رشته را از پایپ لاین ۱ خوانده و همانند قبل به کمک کد اسکی رشته را پردازش کرده و حروف بزرگ و کوچک را تغییر میدهد و نتیجه را در پایپ لاین ۲ مینویسد. در نهایت نیز پردازش ۱ مقدار نهایی را از پایپ لاین ۲ میخواند. فرآیند ساخت و استفاده از پایپ لاین ها همانند قبل خواهد بود که در هنگام خواندن و نوشتن یکی از `descriptors` ها را بسته و فقط با `descriptors` مربوط به عملیات فعلی کار میکنیم، با این تفاوت که دو پایپ لاین داریم.

```

#define BUF_SIZE ۲۵۶

int main(int argc, char *argv[])
{
    int pfd۱[۲];
    int pfd۲[۲];

    ssize_t numRead = -۱;
    /* Note: working under the assumption that the messages
       are of equal length*/
    char* messageOne = argv[۱];
    char* messageTwo ;

    const unsigned int commLen = strlen(messageOne) + ۱;

    char buf[BUF_SIZE];

    // Error handling ... //
    // child \
    switch (fork())
    {

        case ۰:
            printf("\nChild \ executing...\n");
            /* close reading end of first pipe */
            if (close(pfd۱[۰]) == -۱)
            {
                printf("Error closing reading end of pipe ۱.\n");
                _exit(۱);
            }
            /* close writing end of second pipe */
            if (close(pfd۲[۱]) == -۱)
            {
                printf("Error closing writing end of pipe ۲.\n");
                _exit(۱);
            }

            /* write to pipe ۱ */
            if (write(pfd۱[۱], messageOne, commLen) != commLen)
            {
                printf("Error writing to pipe ۱.\n");
                _exit(۱);
            }

            if (close(pfd۱[۱]) == -۱)
            {
                printf("Error closing writing end of pipe ۱.\n");
                _exit(۱);
            }
    }
}

```

```

    /* reading from pipe 1 */
    numRead = read(pfd1[0], buf, commLen);
    if (numRead == -1)
    {
        printf("Error reading from pipe 1.\n");
        _exit(1);
    }

    if (close(pfd1[1]) == -1)
    {
        printf("Error closing reading end of pipe 1.\n");
        _exit(1);
    }

    printf("Message received child ONE: %s\n", buf);
    printf("Exiting child 1...\n");
    _exit(0);

default:
    break;
}

// child 2
switch (fork())
{
    case -1:
        printf("Error forking child 2!\n");
        exit(1);
    case 0:
        printf("\nChild 2 executing...\n");
        /* close reading end of second pipe */
        if (close(pfd2[0]) == -1)
        {
            printf("Error closing reading end of pipe 2.\n");
            _exit(1);
        }
        /* close writing end of first pipe */
        if (close(pfd1[1]) == -1)
        {
            printf("Error closing writing end of pipe 1.\n");
            _exit(1);
        }

        /* read from the first pipe */
        if (read(pfd1[0], buf, commLen) == -1)
        {
            printf("Error reading from pipe 1.\n");
            _exit(EXIT_FAILURE);
        }

        if (close(pfd1[0]) == -1)
        {
            printf("Error closing reading end of pipe 1.\n");

```



```

        _exit(EXIT_FAILURE);
    }

    for (int i = 0; messageOne[i]!='\0'; i++)
    {
        if(messageOne[i] >= 'a' && messageOne[i] <= 'z')
        {
            messageOne[i] = messageOne[i] - ٣٢;
        }
        else if(messageOne[i] >= 'A' && messageOne[i] <= 'Z')
        {
            messageOne[i] = messageOne[i] + ٣٢;
        }
    }
    /* write to the second pipe */
    if (write(pfd٢[١], messageOne, commLen) != commLen)
    {
        printf("Error writing to the pipe.");
        _exit(EXIT_FAILURE);
    }

    if (close(pfd٢[١]) == -١)
    {
        printf("Error closing writing end of pipe ٢.");
        _exit(EXIT_FAILURE);
    }

    printf("Message received child TWO: %s\n", buf);
    printf("Exiting child ٢...\n");
    _exit(EXIT_SUCCESS);

default:
    break;
}

printf("Parent closing pipes.\n");

if (close(pfd١[0]) == -١)
{
    printf("Error closing reading end of the pipe.\n");
    exit(EXIT_FAILURE);
}

if (close(pfd٢[١]) == -١)
{
    printf("Error closing writing end of the pipe.\n");
    exit(EXIT_FAILURE);
}

if (close(pfd٢[0]) == -١)
{
    printf("Error closing reading end of the pipe.\n");

```

پردازش رشته

```

        exit(EXIT_FAILURE);
    }

    if (close(pfd[1]) == -1)
    {
        printf("Error closing writing end of the pipe.\n");
        exit(EXIT_FAILURE);
    }

    printf("Parent waiting for children completion...\n");
    if (wait(NULL) == -1)
    {
        printf("Error waiting.\n");
        exit(EXIT_FAILURE);
    }

    if (wait(NULL) == -1)
    {
        printf("Error waiting.\n");
        exit(EXIT_FAILURE);
    }

    printf("Parent finishing.\n");
    exit(EXIT_SUCCESS);
}

```

در ادامه کد بالا را اجرا و آن را تست میکنیم خروجی کد در عکس زیر مشاهده می شود:

```

lab@ubuntu: ~/Desktop/HWS
lab@ubuntu:~/Desktop/HWS$ gcc q3-1.c -o q3-1
lab@ubuntu:~/Desktop/HWS$ ./q3-1 HosnaOyarhoseini
Piped opened with success. Forking ...
Parent closing pipes.
Parent waiting for children completion...
Child 2 executing...
Child 1 executing...
Message received child TWO: HosnaOyarhoseini
Exiting child 2...
Message received child ONE: hosNaOyARHOSEINI
Exiting child 1...
Parent finishing.
lab@ubuntu:~/Desktop/HWS$

```