

National University Ho Chi Minh City
HO CHI MINH UNIVERSITY OF TECHNOLOGY
FACULTY OF COMPUTER SCIENCE AND ENGINEERING



Operating System - CO2018

Instructor: Assoc. Phạm Hoàng Anh

Class: CC04

Name: Nguyễn Quang Thiện

Student ID: 21522994

Completion Day: March 15th, 2023

Table of Contents

I.	Exercise 3.1	2
1.	<i>Requirement</i>	2
2.	<i>Steps to build and get results</i>	2
3.	<i>Explain each code paragraph</i>	2
4.	<i>Implementing</i>	5
	Source code of exercise 3.1	6
II.	Exercise 3.2	8
1.	<i>Requirement</i>	8
2.	<i>Clarify argc and argv</i>	8
3.	<i>Explain the code paragraph</i>	9
	Serial version	9
	Multithread version	11
4.	<i>Implementing</i>	12
	Source code of exercise 3.2	13
III.	Exercise 3.3	15
1.	<i>Requirement</i>	15
2.	<i>Steps to build and get result</i>	15
3.	<i>Explain each code paragraph</i>	15
4.	<i>Implementing</i>	17
	Source code of exercise 3.3	17

I. Exercise 3.1

1. Requirement

Creating a program that launches two child processes, each of which reads a file and calculates the average ratings of the films it contains. And using the shared memory approach to implement the application.

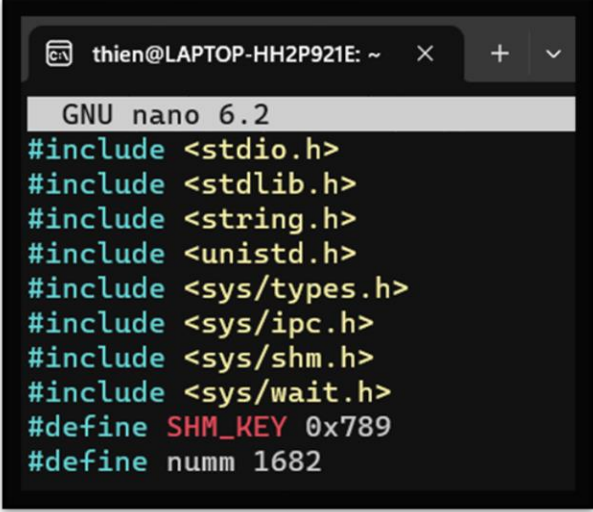
2. Steps to build

To build a program that launches two child processes, each of which reads a file, calculates the average ratings of the movies it contains, and then executes the program using the shared memory technique:

- To keep the total number of ratings for each movie as well as their sum, create a shared memory region.
- For each operation, we'll read a line of the file, add the rating for each movie ID (mID), and then raise the total number of ratings for that particular film.
- Calculating the average rating for each movie after the two processes are complete.

3. Explain each code paragraph

Add the necessary library and the define the SHM_KEY to avoid the conflict, also define the Number of Movies (numm) 1682:



```
thien@LAPTOP-HH2P921E: ~
GNU nano 6.2
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/wait.h>
#define SHM_KEY 0x789
#define numm 1682
```

Designing a data structure (ratingmovie) first to store the data which have count and sum variables to process the exercise:

```
typedef struct{
    int count;
    int sum;
}ratingmovie;
```

Creating readfile function to store the value in an array of numm, where each element's index i contains the total of the ratings and a count of the ratings. The readfile function, which is used to read a text file also read the following information into each line: userID, movieID, rating, timeStamp (in this exercise we just need the movie ID and the rating). After that altered the data at index movie ID (mID) in our shared memory array.

Using fgets(char *str, int n, FILE *fptr) to read each line.

Using strtok(str, sep): Aids in string splitting using preset characters in C strings. **str** is a string to split, **sep** (or separator) is a split character. (“\t” (tab) and NULL are the sep and the str to be used, respectively).

Using atoi(const char *str): Converts a string pointed to by the str parameter to an integer (int). Change the data type string of mID and rating after splitting the string.

```
void readFile(char *filename, ratingmovie*data)
{
    FILE *fptr = fopen(filename, "r");
    char line[512];
    while(fgets(line, sizeof(line), fptr) != NULL)
    {
        int mID, rating;
        char* token = strtok(line, "\t");
        token = strtok(NULL, "\t");
        mID = atoi(token);
        token = strtok(NULL, "\t");
        rating = atoi(token);
        data[mID].count += 1;
        data[mID].sum += rating;
    }
    fclose(fptr);
    exit(EXIT_SUCCESS);
}
```

Spawn the two child process:

Then, shmget will be used to establish a shared memory segment. And the shared memory segment will be attached using shmat after generating an array of size Number of Movies (1682). The array's initial data value will be 0. Thereafter, spawning two child process, each child process will process a single text file.

```
thien@LAPTOP-HH2P921E: ~ × + v
GNU nano 6.2 twochild.c
int main() {
    int shmid;
    ratingmovie *data;

    // Create shared memory segment
    shmid=shmget(SHM_KEY, sizeof(data)*numm, 0644|IPC_CREAT);
    if(shmid < 0)
    {
        perror("shmget");
        return 1;
    }

    // Attach shared memory segment to parent process
    data = (ratingmovie*)shmat(shmid, NULL, 0);
    memset(data, 0, sizeof(data)*numm);
    if(data == (ratingmovie*)-1)
    {
        perror("shmat");
        exit(1);
    }
    pid_t pid1 = fork();
    if (pid1 == 0)
    { //child 1 process
        readFile("/home/thien/movie-100k_1.txt", data);
    }
```

```
    pid_t pid2=fork();
    if(pid2==0)
    { //child 2 process
        readFile("/home/thien/movie-100k_2.txt", data);
    }
    // Parent process
    // Parent process wait 2 child process to finish
    int status;
    waitpid(pid1,&status,0);
    waitpid(pid2,&status,0);
```

Generating the loop to calculate the average rating of the movie:

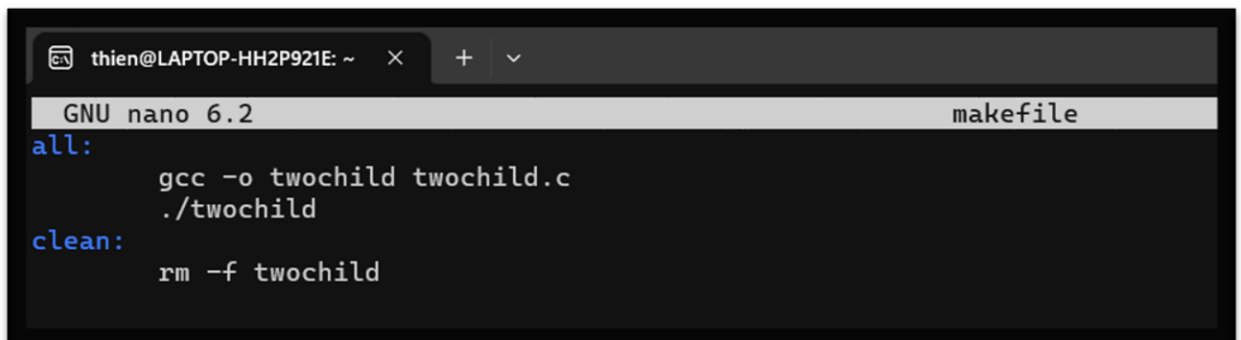
```
// Print the results
for (int i=0; i<numm; i++)
{
    float r = ( (float)data[i].sum ) / ( (float)data[i].count );
    double d = r;
    printf("%d\t\t %.3f\n", i, d);
}
```

And finally:

```
// Detach and remove shared memory segment
if (shmdt(data) == -1)
{
    perror("shmdt");
    exit(EXIT_FAILURE);
}
if (shmctl(shmid, IPC_RMID, NULL) == -1)
{
    perror("shmctl");
    exit(EXIT_FAILURE);
}
return 0;
}
```

4. *Implementing*

First, creating make file:



```
thien@LAPTOP-HH2P921E: ~
GNU nano 6.2 makefile
all:
    gcc -o twochild twochild.c
    ./twochild
clean:
    rm -f twochild
```

Then “make all” to generate the program:

```
thien@LAPTOP-HH2P921E: ~  
thien@LAPTOP-HH2P921E:~$ nano twochild.c  
thien@LAPTOP-HH2P921E:~$ make all  
gcc -o twochild twochild.c  
./twochild  
0          -nan  
1          3.878  
2          3.206  
3          3.033  
4          3.550  
5          3.302  
6          3.577  
7          3.798  
8          3.995  
9          3.896  
10         3.831  
11         3.847  
12         4.386  
13         3.418  
14         3.967  
15         3.778  
16         3.205  
17         3.120  
18         2.800  
19         3.957  
20         3.417  
21         2.762  
22         4.152  
23         4.121  
24         3.448  
25         3.444
```

<https://drive.google.com/file/d/1uu5FXhyYDy5TayC-nQHGR6iY7rq3-cd8/view?usp=sharing> (Results)

Source code of exercise 3.1

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

```

#include <sys/shm.h>
#include <sys/wait.h>
#define SHM_KEY 0x789
#define numm 1682

typedef struct{
    int count;
    int sum;
}ratingmovie;

void readFile(char *filename, ratingmovie*data)
{
    FILE *fptr = fopen(filename, "r");
    char line[512];
    while(fgets(line, sizeof(line), fptr) != NULL)
    {
        int mID, rating;
        char* token = strtok(line, "\t");
        token = strtok(NULL, "\t");
        mID = atoi(token);
        token = strtok(NULL, "\t");
        rating = atoi(token);
        data[mID].count += 1;
        data[mID].sum += rating;
    }
    fclose(fptr);
    exit(EXIT_SUCCESS);
}

int main() {
    int shmid;
    ratingmovie *data;

    // Create shared memory segment
    shmid=shmget(SHM_KEY, sizeof(data)*numm, 0644|IPC_CREAT);
    if(shmid < 0)
    {
        perror("shmget");
        return 1;
    }

    // Attach shared memory segment to parent process
    data = (ratingmovie*)shmat(shmid, NULL, 0);
    memset(data, 0, sizeof(data)*numm);
    if(data == (ratingmovie*)-1)
    {
        perror("shmat");
        exit(1);
    }
    pid_t pid1 = fork();

```



```

if (pid1 == 0)
{ //child 1 process
    readFile("/home/thien/movie-100k_1.txt", data);
}
pid_t pid2=fork();
if(pid2==0)
{ //child 2 process
    readFile("/home/thien/movie-100k_2.txt", data);
}
// Parent process
// Parent process wait 2 child process to finish
int status;
waitpid(pid1,&status,0);
waitpid(pid2,&status,0);

// Print the results
for (int i=0; i<numm; i++)
{
    float r = ( (float)data[i].sum ) / ( (float)data[i].count );
    double d = r;
    printf("%d\t\t %.3f\n", i, d);
}
// Detach and remove shared memory segment
if (shmdt(data) == -1)
{
    perror("shmdt");
    exit(EXIT_FAILURE);
}
if (shmctl(shmid, IPC_RMID, NULL) == -1)
{
    perror("shmctl");
    exit(EXIT_FAILURE);
}
return 0;
}

```

II. Exercise 3.2

1. Requirement

Compared to the serial version, the multi-thread version could accelerate more quickly. The Makefile has at least two targets for the compilation of the two programs, sum serial and sum multi-thread.

2. Clarify argc and argv

In C and C++, main() receives command line arguments through argv and argc. The amount of strings that argv points to will be argc. In reality, this will be 1 in

addition to the amount of arguments because almost all implementations prepend the program name to the array.

3. *Explain the code paragraph*

Serial version

Computing the sum from 1 to n by providing an argument n while compiling `./sum_serial n`. However, the application will prompt the user to correct the command if the number of arguments is different from 2.

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char* argv[])
{
    if(argc != 2)
    {
        printf("Usage: %s n\n", argv[0]);
        return 1;
    }

    long int sum = 0;
    printf("%d\n", atoi(argv[1]));
    for(int i = 0; i <= atoi(argv[1]); i++)
    {
        sum = sum + i;
    }
    printf("The sum from 1 to %d is %ld\n", atoi(argv[1]), sum);
    return 0;
}
```

Result:

```
thien@LAPTOP-HH2P921E: ~ x + v
thien@LAPTOP-HH2P921E:~$ nano sum_serial.c
thien@LAPTOP-HH2P921E:~$ gcc -o sum_serial sum_serial.c
thien@LAPTOP-HH2P921E:~$ ./sum_serial
Usage: ./sum_serial n
thien@LAPTOP-HH2P921E:~$ ./sum_serial 10
10
The sum from 1 to 10 is 55
thien@LAPTOP-HH2P921E:~$ rm -f sum_serial
thien@LAPTOP-HH2P921E:~$ gcc -o sum_serial sum_serial.c
thien@LAPTOP-HH2P921E:~$ ./sum_serial
Usage: ./sum_serial n
thien@LAPTOP-HH2P921E:~$ ./sum_serial 1000000
1000000
The sum from 1 to 1000000 is 500000500000
thien@LAPTOP-HH2P921E:~$
```

Moreover, we can print the time taken to execute by adding the library `<time.h>`

```
thien@LAPTOP-HH2P921E: ~ x + v
thien@LAPTOP-HH2P921E:~$ gcc -o sum_serial1 sum_serial1.c
thien@LAPTOP-HH2P921E:~$ ./sum_serial1 10
10
The sum from 1 to 10 is 55
Time taken to execute in seconds : 0.000040
thien@LAPTOP-HH2P921E:~$ gcc -o sum_serial1 sum_serial1.c
thien@LAPTOP-HH2P921E:~$ ./sum_serial1 1000
1000
The sum from 1 to 1000 is 500500
Time taken to execute in seconds : 0.000059
thien@LAPTOP-HH2P921E:~$ gcc -o sum_serial1 sum_serial1.c
thien@LAPTOP-HH2P921E:~$ ./sum_serial1 1000000
1000000
The sum from 1 to 1000000 is 500000500000
Time taken to execute in seconds : 0.013320
thien@LAPTOP-HH2P921E:~$
```

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main(int argc, char* argv[])
```

```

{
    clock_t start, end;
    double execution_time;
    /* Store start time here */
    start = clock();

    /* Put the main body of your program here */
    if(argc != 2)
    {
        printf("Usage: %s n\n", argv[0]);
        return 1;
    }

    long int sum = 0;
    printf("%d\n", atoi(argv[1]));
    for(int i = 0; i <= atoi(argv[1]); i++)
    {
        sum = sum + i;
    }
    printf("The sum from 1 to %d is %ld\n", atoi(argv[1]), sum);

    /* Program logic ends here */
    end = clock();
    /* Get the time taken by program to execute in seconds */
    execution_time = ((double)(end - start))/CLOCKS_PER_SEC;

    printf("Time taken to execute in seconds : %f\n", execution_time);

    return 0;
}

```

Multithread version

First of all, creating a data structure which include: start number, end number and the result after sum of that thread:

```

struct getsum
{
    int start;
    int end;
    long int result;
};

```

Then, getting the number of threads, the number n that wanting to get the sum to and creating the thread array also the array of argument for each thread:

```

int numthread = atoi(argv[1]);
int n = atoi(argv[2]);
long int sum = 0;

// Create a thread array and array of argument for each thread
pthread_t threads[numthread];
struct getsum gs[numthread];

```

Generating the loop in order to divide the task for each thread:

```

// Create threads and divide the task
for (int i = 0; i < numthread; i++)
{
    gs[i].start = (i * n) / numthread + 1;
    gs[i].end = ( (i + 1) * n ) / numthread;
    pthread_create(&threads[i], NULL, sumthread, &gs[i]);
}

```

When all threads have terminated, connecting them and accumulate the outcomes and printing the output:

```

// Wait for threads to terminate and accumulate the outcomes
for (int i = 0; i < numthread; i++)
{
    pthread_join(threads[i], NULL);
    sum += gs[i].result;
}

printf("Sum from 1 to %d is %ld \n", n, sum);

```

4. *Implementing*

Result:

```
thien@LAPTOP-HH2P921E: ~  
thien@LAPTOP-HH2P921E:~$ nano sum_multithread.c  
thien@LAPTOP-HH2P921E:~$ gcc -o sum_multithread sum_multithread.c  
thien@LAPTOP-HH2P921E:~$ ./sum_multithread  
Usage: ./sum_multithread numthread n  
thien@LAPTOP-HH2P921E:~$ ./sum_multithread 10 10  
Sum from 1 to 10 is 55  
thien@LAPTOP-HH2P921E:~$ rm -f sum_multithread  
thien@LAPTOP-HH2P921E:~$ ./sum_multithread  
-bash: ./sum_multithread: No such file or directory  
thien@LAPTOP-HH2P921E:~$ gcc -o sum_multithread sum_multithread.c  
thien@LAPTOP-HH2P921E:~$ ./sum_multithread  
Usage: ./sum_multithread numthread n  
thien@LAPTOP-HH2P921E:~$ ./sum_multithread 10 1000000  
Sum from 1 to 1000000 is 500000500000  
thien@LAPTOP-HH2P921E:~$
```

Moreover, we can print the time taken to execute by adding the library `<time.h>`

```
thien@LAPTOP-HH2P921E: ~  
thien@LAPTOP-HH2P921E:~$ gcc -o sum_multithread1 sum_multithread1.c  
thien@LAPTOP-HH2P921E:~$ ./sum_multithread1 10 10  
Sum from 1 to 10 is 55  
Time taken to execute in seconds : 0.000982  
thien@LAPTOP-HH2P921E:~$ gcc -o sum_multithread1 sum_multithread1.c  
thien@LAPTOP-HH2P921E:~$ ./sum_multithread1 10 1000  
Sum from 1 to 1000 is 500500  
Time taken to execute in seconds : 0.000754  
thien@LAPTOP-HH2P921E:~$ gcc -o sum_multithread1 sum_multithread1.c  
thien@LAPTOP-HH2P921E:~$ ./sum_multithread1 10 1000000  
Sum from 1 to 1000000 is 500000500000  
Time taken to execute in seconds : 0.002397  
thien@LAPTOP-HH2P921E:~$
```

Source code of exercise 3.2

```
#include <stdio.h>  
#include <stdlib.h>  
#include <pthread.h>  
#include <time.h>  
  
struct getsum  
{  
    int start;  
    int end;
```

```

    long int result;
};
void* sumthread(void* arg)
{
    struct getsum* gs = (struct getsum*)arg;
    long int sum = 0;
    for (int i = gs->start; i <= gs->end; i++)
    {
        sum += i;
    }
    gs->result = sum;
    pthread_exit(NULL);
}
int main(int argc, char* argv[])
{
    clock_t start, end;
    double execution_time;
    /* Store start time here */
    start = clock();

    /* Put the main body of your program here */
    if (argc != 3)
    {
        printf("Usage: %s numthread n \n", argv[0]);
        return 1;
    }

    int numthread = atoi(argv[1]);
    int n = atoi(argv[2]);
    long int sum = 0;

    // Create a thread array and array of argument for each thread
    pthread_t threads[numthread];
    struct getsum gs[numthread];

    // Create threads and divide the task
    for (int i = 0; i < numthread; i++)
    {
        gs[i].start = (i * n) / numthread + 1;
        gs[i].end = ( (i + 1) * n ) / numthread;
        pthread_create(&threads[i], NULL, sumthread, &gs[i]);
    }

    // Wait for threads to terminate and accumulate the outcomes
    for (int i = 0; i < numthread; i++)
    {
        pthread_join(threads[i], NULL);
        sum += gs[i].result;
    }

    printf("Sum from 1 to %d is %ld \n", n, sum);
}

```



```

/* Program logic ends here */
end = clock();
/* Get the time taken by program to execute in seconds */
execution_time = ((double)(end - start))/CLOCKS_PER_SEC;

printf("Time taken to execute in seconds : %f\n", execution_time);

return 0;
}

```

III. Exercise 3.3

1. Requirement

Having some tricks to adapt it for two-way communication by using two pipes due to the pipe being a one-way communication method conventionally. Then, implementing the TODO segment.

2. Steps to build

Create two pipes: one for delivering data from the parent process to the child process, and another for sending data from the child process to the parent process, in order to achieve two-way communication using pipes. And the fork() system function can then be used to spawn a child process.

3. Explain each code paragraph

Generating 2 pipes:

```

static int pipefd1[2], pipefd2[2];
void INIT(void) {
    if (pipe(pipefd1) < 0 || pipe(pipefd2) < 0 )
    {
        perror("pipe");
        exit(EXIT_FAILURE);
    }
}

```

Let **pipefd1** is Parent to Child and **pipefd2** is Child to Parent.

The read end of the pipe, which is used to transfer data to the child process, and the write end, which is used to receive data from the child process, can be closed in the **parent process**.

The read end of the pipe, which is used to transmit data to the parent process, and the write end, which is used to receive data from the parent process, can be closed in the **child process**.

Also, you can write to the write end of the pipe for transferring data to the child process to **transmit data** from the **parent process to the child process**. You can read from the read end of the pipe for **getting data** from the **child process** if you choose to do it.

In a similar manner, you may write to the write end of the pipe for sending data to the parent process to **transfer data** from the **child process to the parent process**. You can read from the read end of the pipe for the **parent process to get data** by doing so.

```
void WRITE_TO_PARENT(void)
{
    /* send parent a message through pipe*/
    // TO DO

    char buffer[1024];
    sprintf(buffer, "Child send message to parent!");
    write(pipefd2[1], buffer, sizeof(buffer));

    printf("Child send message to parent!\n");
}
void READ_FROM_PARENT(void)
{
    /* read message sent by parent from pipe*/
    // TO DO

    char buffer[1024];
    read(pipefd1[0], buffer, sizeof(buffer));

    printf("Child receive message from parent: %s\n", buffer);
}
void WRITE_TO_CHILD(void)
{
    /* send child a message through pipe*/
    // TO DO

    char buffer[1024];
    sprintf(buffer, "Parent send message to child!");
    write(pipefd1[1], buffer, sizeof(buffer));

    printf("Parent send message to child!\n");
}
```

```

void READ_FROM_CHILD(void)
{
    /* read the message sent by child from pipe */
    // TO DO

    char buffer[1024];
    read(pipefd2[0], buffer, sizeof(buffer));

    printf("Parent receive message from child: %s\n", buffer);
}

```

4. *Implementing*

Result:

```

thien@LAPTOP-HH2P921E: ~$ gcc -o pipe pipe.c
thien@LAPTOP-HH2P921E:~$ ./pipe
Child receive message from parent: Parent send message to child!
Parent send message to child!
Child send message to parent!
Parent receive message from child: Child send message to parent!
Parent send message to child!
Child receive message from parent: Parent send message to child!
Child send message to parent!
Parent receive message from child: Child send message to parent!
Parent send message to child!
Child receive message from parent: Parent send message to child!
Child send message to parent!
Parent receive message from child: Child send message to parent!
Parent send message to child!
Child receive message from parent: Parent send message to child!
Child send message to parent!
Parent receive message from child: Child send message to parent!
Parent send message to child!
Child receive message from parent: Parent send message to child!
Child send message to parent!
Parent receive message from child: Child send message to parent!
Alarm clock
thien@LAPTOP-HH2P921E:~$

```

Source code of exercise 3.3

```

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

static int pipefd1[2], pipefd2[2];

```

```

void INIT(void) {
    if (pipe(pipefd1) < 0 || pipe(pipefd2) < 0 )
    {
        perror("pipe");
        exit(EXIT_FAILURE);
    }
}

void WRITE_TO_PARENT(void)
{
    /* send parent a message through pipe*/
    // TO DO

    char buffer[1024];
    sprintf(buffer, "Child send message to parent!");
    write(pipefd2[1], buffer, sizeof(buffer));

    printf("Child send message to parent!\n");
}

void READ_FROM_PARENT(void)
{
    /* read message sent by parent from pipe*/
    // TO DO

    char buffer[1024];
    read(pipefd1[0], buffer, sizeof(buffer));

    printf("Child receive message from parent: %s\n", buffer);
}

void WRITE_TO_CHILD(void)
{
    /* send child a message through pipe*/
    // TO DO

    char buffer[1024];
    sprintf(buffer, "Parent send message to child!");
    write(pipefd1[1], buffer, sizeof(buffer));

    printf("Parent send message to child!\n");
}

void READ_FROM_CHILD(void)
{
    /* read the message sent by child from pipe */
    // TO DO

    char buffer[1024];
    read(pipefd2[0], buffer, sizeof(buffer));

    printf("Parent receive message from child: %s\n", buffer);
}

```

```

int main(int argc, char* argv[])
{
    INIT();
    pid_t pid;
    pid = fork();
    // set a timer, process will end after 10 seconds.
    alarm(10);
    if (pid == 0)
    {
        while (1)
        {
            sleep(rand() % 2 + 1);
            WRITE_TO_CHILD();
            READ_FROM_CHILD();
        }
    }
    else
    {
        while (1)
        {
            sleep(rand() % 2 + 1);
            READ_FROM_PARENT();
            WRITE_TO_PARENT();
        }
    }
    return 0;
}

```