بسمه تعالی

گزارش‌کار آزمایشگاه سیستم عامل

استاد مربوطه:

مهندس اوا انوری

آزمایش ششم:

Deadlock

اعضای گروه:

فروغ افخمی9831703

پاییز 1401

توضیح کلی ازمایش 7:

در این آزمایش الگوریتم بانکداران بر روی برنامه ای thread-multi اجرا شد. انواعی از منابع در اینجا 5 تعریف شد که تعدادی از مشتریان(5نفر) قصد استفاده مشترک از آنها را دارند. تعداد هر یک از منابع در دسترس به عنوان آرگومان توسط برنامه دریافت می شود. برای پیاده سازی مشتریان متعدد از pthread استفاده شد. هر مشتری به طور تصادفی تعدادی منبع را درخواست می کند یا پس می دهد. در اینجا طبق دستور کار برای درخواست و پس دادن منابع، توابع release_resources و request_resources پیاده سازی شدند. چون این توابع بر روی منابع مشترک خواندن و نوشتن انجام می دهند برای جلوگیری از condition race از قفل mutex استفاده شده تا انحصار متقابل حفظ شود. هنگام درخواست دادن الگوریتم ابتدا چک می کند که منابع کافی در دسترس باشد و سپس safe بودن حالت بعدی را برای پیشگیری از وقوع بن بست چک کرده و در این صورت منابع را به مشتری اختصاص می دهد. در غیر این صورت درخواست مشتری رد می شود .

توضیح ابتدای کد:

```c
#include <time.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <unistd.h>
#include <string.h>
#include <pthread.h>
#include <semaphore.h>

#define NUMBER_OF_RESOURCES 5
#define NUMBER_OF_CUSTOMERS 5
#define MAX_CUSTOMER_ITERATION 10
#define MAX_FILE_NAME "max.txt"

sem_t mutex;

int available[NUMBER_OF_RESOURCES];
int maximum[NUMBER_OF_CUSTOMERS][NUMBER_OF_RESOURCES];
int allocation[NUMBER_OF_CUSTOMERS][NUMBER_OF_RESOURCES] = {0};
int need[NUMBER_OF_CUSTOMERS][NUMBER_OF_RESOURCES];
```

```
bool need_lt_work(int need_i[], int work[]);
bool is_safe_state();
int request_resources(int customer_num, int request[]);
int release_resources(int customer_num, int request[]);
void *customer_thread(int n);
void print_state();
char *req_to_str(int req[]);
```

در ابتدا یک سری تعاریف و مقدار دهی های اولیه انجام شده است. برای مثال ما در اینجا ۵ مشتری و ۵ نوع منبع داریم و هر مشتری میتواند حداکثر 10 بار درخواست منبع گرفتن یا ازاد کردن کند.

همچنین یک فایل max.txt داریم که به صورت دستی مقادیر max را در ان وارد میکنیم و در پوشه که کد قرار دارد قرار میدهیم تا بتوان از محتوای ان استفاده کرد.

از mutex برای قفل کردن و جلوگیری از race condition استفاده میکنیم که بالاتر نیز توضیح داده شد.

حال تعداد منابع در دسترس و ماکسیمم منابع را در یک بردار ذخیره میکنیم.یک سری ماتریس هم داریم که نشان دهنده تعداد allocateکرده و need هر مشتری برای هر منبع میباشد. یک سری تابع نیز تعریف شده که در ادامه به صورت تک به تک توضیح داده خواهند شد.

کد تابع need_lt_work:

```
bool need_lt_work(int need_i[], int work[]) {
    for (int i = 0; i < NUMBER_OF_RESOURCES; i++)
    {
        if (need_i[i] > work[i]) {
            return false;
        }
    }
    return true;
}
```

توضیح تابع need_lt_work :

در این تابع چک میشود که ایا منابع درخواستی مشتری iام کمتر از منابع در دسترس باشد در صورتی که این شرط برای همه منابع برقرار باشد true و در غیر این صورت false برمیگرداند.

کد تابع is_safe_state :

```
bool is_safe_state() {
    int work[NUMBER_OF_RESOURCES];
    for (int i = 0; i < NUMBER_OF_RESOURCES; i++)
    {
        work[i] = available[i];
    }

    bool finish[NUMBER_OF_CUSTOMERS] = {0};
    int finish_count = 0;
    bool changed;
    while (finish_count != NUMBER_OF_CUSTOMERS) {
        changed = false;
        for (int i = 0; i < NUMBER_OF_CUSTOMERS; i++)
        {
            if (!finish[i] && need_lt_work(need[i], work)) {
                for (int j = 0; j < NUMBER_OF_RESOURCES; j++)
                {
                    work[j] += allocation[i][j];
                }

                finish[i] = true;
                finish_count++;
                changed = true;
            }
        }
        if (!changed) {
            return false;
        }
    }
    return true;
}
```

توضیح تابع is_safe_state :

در این تابع چک میکنیم که ایا در صورت تخصیص منابع به یک درخواست باز هم در حالت safe میمانیم یا خیر؟ بدین منظور سعی میکنیم دنباله ای از روند اجرای ترد ها را پیدا کنیم که دچار deadlock نشوند اگر توانستیم چنین ترتیبی را پیدا کنیم یعنی در حالت امن هستیم و مشکلی برای تخصیص منابع نداریم پس کار را ادامه میدهیم اما اگر چنین ترتیبی پیدا نشد یعنی حالت امن وجود ندارد پس نمیتوانیم منابع را به صورت امن به ترد درخواست کننده بدهیم پس منابع را از آن پس میگیریم.

ابتدا بردار منابع موجود را با یک for داخل یک بردار دیگر به نام work ریخته ایم. سپس یک بردار finishداریم که در صورت تخصیص منابع به ان مشتری خانه مربوط به مشتری را true میکنیم.

در واقع ما باید هر !5*5! حالت ممکن را بررسی کنیم که ایا در هیچ کدام حالت امنی وجود دارد یا نه درصورتی که وجود داشت trueرا returnمیکنیم یعنی در حالت ایمن هستیم.
دو متغیر changed , finish_count نیز تعریف میکنیم.حال با کمک دو حلقه for , while این !5*5!را بررسی میکنیم.در هر بار iterateکردن متغیر changedرا false میگذاریم.
حال چک میکنیم اگر کار مشتری i ام تمام نشده است و تعداد منابع درخواستی ان کمتر از منابع موجود است (need_lt_work) منبع را به ان تخصیص میدهیم و در ادامه و اتمام کارش منابع را در تکه کد زیر ازاد میکند .

```
                for (int j = 0; j < NUMBER_OF_RESOURCES; j++)
                {
                    work[j] += allocation[i][j];
                }
```

پس حال کار مشتریi ام تمام شده است و خانه مربوط به ان را در بردار finish ،true میکنیم. changed را نیز trueمیکنیم چون تغییر کرده و finish_countرا یکی جلو میبریم.
در صورتی که changed بعد از for ،false بماند یعنی اجرای انتخاب شده ایمن نیست و falseرا برمیگردانیم.در غیر این صورت در خارج whileاگر changed،trueباشد یعنی اجرای ایمن یافته ایم .

کد تابع request_resources :

```
int request_resources(int customer_num, int request[]) {

    for (int i = 0; i < NUMBER_OF_RESOURCES; i++)
    {
        if (request[i] > need[customer_num][i] || request[i] > available[i]) {
            return -1;
        }
    }

    sem_wait(&mutex);

    for (int i = 0; i < NUMBER_OF_RESOURCES; i++)
    {
        available[i] -= request[i];
        allocation[customer_num][i] += request[i];
        need[customer_num][i] -= request[i];
    }

    if (is_safe_state()) {
        sem_post(&mutex);
        return 0;
```

```
    }

    for (int i = 0; i < NUMBER_OF_RESOURCES; i++)
    {
        available[i] += request[i];
        allocation[customer_num][i] -= request[i];
        need[customer_num][i] += request[i];
    }
    sem_post(&mutex);
    return -1;
}
```

توضیح تابع request_resources:

در تابع resources_request الگوریتم بانکداران را اجرا میکنیم. بدین صورت که این تابع شماره یک ریسمان و مقدار منابعی که درخواست کرده را به عنوان ورودی میگیرد و الگوریتم را به کمک آنها شروع میکند .ابتدا بررسی میکند اگر تعداد منابع درخواست شده از تعداد منابع دردسترس و یا تعداد منابعی که ادعا شده است این ترد نیاز داریم(need) بیشتر باشد این درخواست در همینجا خاتمه میابد1- را خروجی میدهیم.در غیر این صورت اگر مقدار درخواستی مجاز باشد ابتدا منابع را با استفاده از mutex قفل میکنیم سپس فرض میکنیم که منبع را تخصیص داده ایم در نتیجه مقدار درخواستی را ازneed کم میکنیم و به allocation اضافه میکنیم و از مقدار منابع در دسترس نیز کم میکنیم.

حال چک میکنیم اگر شرایط امن باشد قفل را ازاد میکنیم و 0 را returnمیکنیم یعنی موفق بوده ایم.

اگر در حالت امن نبودیم باید اعمال انجام شده را معکوس کنیم یعنی مقدار درخواستی را به need اضافه میکنیم واز allocation کم میکنیم و به مقدار منابع در دسترس نیز اضافه میکنیم.قفل را ازاد میکنیم و 1- را خروجی میدهیم.

کد تابع release_requests:

```
int release_resources(int customer_num, int request[]) {
    sem_wait(&mutex);

    for (int i = 0; i < NUMBER_OF_RESOURCES; i++)
    {
        available[i] += request[i];
        allocation[customer_num][i] -= request[i];
        need[customer_num][i] += request[i];
    }

    sem_post(&mutex);
}
```

توضیح release_requests:

اگر توانستیم منابع را تخصیص بدهیم یعنی از مقدار need ریسمان مورد نظر کم شده و اگر حالتی پیش بیاید که مقدار need یک ریسمان 0 شود یعنی این ریسمان کارش به طور کلی تمام شده است درنتیجه میتواند منابعی را که Allocateکرده است آزاد کند. برای این کار از تابع resources_release استفاده میکنیم که در آن منابعی که ریسمان مورد نظر گرفته بوده به اندازه max درخواست های مشتری است را به منابع دردسترس اضافه میکنیم. قبل انجام این کار باید منابع را قفل کنیم و سپس منابع را ازاد کنیم تا تداخل و race conditionرخ ندهد.

کد customer_thread:

```
void *customer_thread(int n) {
    int req[NUMBER_OF_RESOURCES];
    bool req_rel;
    int done;
    for (int i = 0; i < MAX_CUSTOMER_ITERATION; i++)
    {
        req_rel = rand() % 2;
        if (req_rel == 1) { // request
            for (int j = 0; j < NUMBER_OF_RESOURCES; j++)
            {
                req[j] = rand() % (need[n][j] + 1);
            }
            done = request_resources(n, req);
            printf("Customer %d Requests [%s\b] -> %s\n", n + 1,
req_to_str(req), done == 0 ? "accepted" : "not accepted");

        } else { // release
            for (int j = 0; j < NUMBER_OF_RESOURCES; j++)
            {
                req[j] = rand() % (allocation[n][j] + 1);
            }
            done = release_resources(n, req);
            printf("Customer %d Releases [%s\b]\n", n + 1, req_to_str(req));
        }
    }

}
```

در این تابع ما کد مربوط به ساخت ریمسان هر مشتری را مینویسم. ابتدا یک بردار داریم به نام resکه مشخص میکند مشتری میخواهد چه تعداد از هر منبع را ازاد کند یا بگیرد.حال به صورت رندم انتخاب میکنیم که ایا

میخواهیم منابع را ازاد کنیم یا بگیریم. اگر مشتری بخواد منبع بگیرد مقدار requestرا به صورت رندم انتخاب میکنیم و تابع request_resources را صدا میزنیم و با توجه به خروجی تابع چاپ میکنیم که ایا درخواست قبول شده است یا خیر.
اگر مشتری بخواهد منبع ازاد کند مقدار requestرا به صورت رندم انتخاب میکنیم تا این مقدار منبع ازاد شود و تابع release_resources را صدا میزنیم و با توجه به خروجی تابع چاپ میکنیم که ایا درخواست قبول شده است یا خیر.

کد تابع req_to_str:

```
char *req_to_str(int req[]) {
    char *ret = malloc(100);
    char buf[5] = {0};
    for (int i = 0; i < NUMBER_OF_RESOURCES; i++)
    {
        sprintf(buf, "%d", req[i]);
        strcat(ret, buf);
        strcat(ret, " ");
    }
    return ret;
}
```

این تابع درخواست های کاربران را به صورت stringچاپ میکنید.

کد تابع print_state:

```
void print_state() {
    printf("Available:\n");
    for (int i = 0; i < NUMBER_OF_RESOURCES; i++)
    {
        printf("%d ", available[i]);
    }
    printf("\n");

    printf("Maximum:\n");
    for (int i = 0; i < NUMBER_OF_CUSTOMERS; i++)
    {
        for (int j = 0; j < NUMBER_OF_RESOURCES; j++)
        {
            printf("%d ", maximum[i][j]);
        }
        printf("\n");
    }
```

```
    printf("Allocation:\n");
    for (int i = 0; i < NUMBER_OF_CUSTOMERS; i++)
    {
        for (int j = 0; j < NUMBER_OF_RESOURCES; j++)
        {
            printf("%d ", allocation[i][j]);
        }
        printf("\n");
    }
    printf("Need:\n");
    for (int i = 0; i < NUMBER_OF_CUSTOMERS; i++)
    {
        for (int j = 0; j < NUMBER_OF_RESOURCES; j++)
        {
            printf("%d ", need[i][j]);
        }
        printf("\n");
    }
    printf("====================\n");
}
```

این تابع ماتریس های مربوط به maximum,allocation,need,available را چاپ میکند.

کد تابع main:

```
int main(int argc, char const *argv[])
{
    srand(time(NULL));

    if (argc < NUMBER_OF_RESOURCES + 1) {
        printf("Not enough arguments\n");
        return EXIT_FAILURE;
    }
    for (int i = 0; i < NUMBER_OF_RESOURCES; i++)
    {
        available[i] = atoi(argv[i + 1]);
    }
    FILE *f_ptr = fopen(MAX_FILE_NAME,"r");
    if (f_ptr == NULL)
    {
        printf("%s not found.\n", MAX_FILE_NAME);
        return 0;
    }
    for (int i = 0; i < NUMBER_OF_CUSTOMERS; i++)
```

```
    {
        for (int j = 0; j < NUMBER_OF_RESOURCES; j++)
        {
            fscanf(f_ptr, "%d", &maximum[i][j]);
            need[i][j] = maximum[i][j];
        }
    }
    print_state();

    sem_init(&mutex, 0, 1);

    pthread_t customer_threads[NUMBER_OF_CUSTOMERS];
    for (int i = 0; i < NUMBER_OF_RESOURCES; i++)
    {
        pthread_create(&customer_threads[i], NULL, (void *) customer_thread,
(void *)(intptr_t) i);
    }
    for (int i = 0; i < 5; i++)
    {
        pthread_join(customer_threads[i], NULL);
    }

    return 0;
}
```

توضیح کد main:

در ابتدا دستور srand((time(Null)) را مینویسیم چون از randدر توابع استفاده کرده ایم.

حال چک میکنیم تعداد ارگومان های ورودی که همان مقادیر available منابع هستند به درستی وارد شده باشند.اگر نباشد پیغامی چاپ میکنیم که ورودی نادرست وارد شده است.

سپس از atoi استفاده میکنیم که مقادیر رشته ورودی را به integer تبدیل کنیم و در بردار available میریزیم.

حال فایلmax.txt را به صورت readباز میکنیم و مقادیررا با استفاده از fscanf میخوانیم و در ماتریس maximum قرار میدهیم و مقدار need را هم برابر maximum قرار میدهیم.همچنین در صورت اشکال در باز کردن فایل exceptionرا هندل کرده ایم.

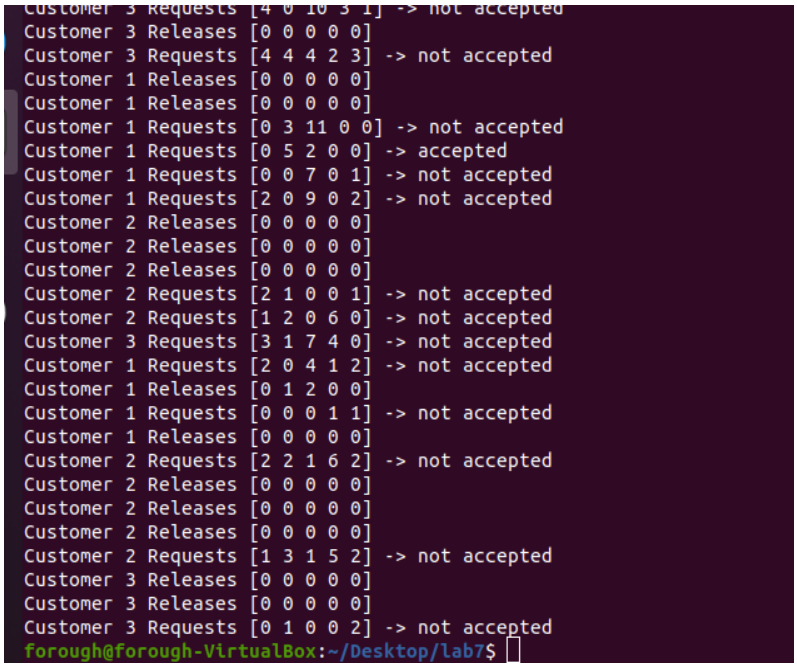
سپس تابع print_state را صدا میزنیم تا مقادیر ماتریس ها و بردار ها را مشاهده کنیم.همچنین mutex را مقدار دهی اولیه میکنیم مقدار اولیه اش را 1 میگذاریم.

توضیح ارگومان دوم در sem_init:

آرگومان pshared نشان می دهد که آیا این سمافور باید باشد یا خیر؟ بین رشته های یک فرآیند یا بین فرآیندها به اشتراک گذاشته می شود.اگر pshared دارای مقدار 0 باشد، سمافور بین آنها به اشتراک گذاشته می شودرشته‌های یک فرآیند، و باید در آدرسی قرار گیرندکه برای همه رشته ها قابل مشاهده است .
سپس ریسمان های مربوط به مشتری ها را میسازیم. در pthread_create ارگومان سوم نشان دهنده تابعی است که با ان ریسمان ما startمیخورد که در اینجا customer_thread است که در ان به صورت رندم هر مشتری درخواست میکند منبعی را بگیرد یا ازاد کند.
همچنین intptr_t یک عدد صحیح علامت گذاری شده از نوع memsize است که می تواند با خیال راحت یک اشاره گر را بدون توجه به ظرفیت پلت فرم ذخیره کند.
همچنین تابع pthread_join() منتظر می ماند تا یک thread خاتمه یابد، ریسمان را جدا می کند و سپس وضعیت خروج رشته را برمی گرداند. اگر پارامتر وضعیت NULL باشد، وضعیت خروج رشته ها برگردانده نمیشود.این تابع باید نوشته شود زیرا در غیر این صورت ریسمان های zombieایجاد میشود.

نتیجه ازمایش:

```
forough@forough-VirtualBox:~/Desktop/lab7$ ./banker 10 8 12 9 5
Available:
10 8 12 9 5
Maximum:
2 5 11 1 2
2 7 1 6 2
9 6 10 6 4
1 1 6 3 3
2 8 11 1 5
Allocation:
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
Need:
2 5 11 1 2
2 7 1 6 2
9 6 10 6 4
1 1 6 3 3
2 8 11 1 5
====================
Customer 4 Requests [1 0 0 1 1] -> accepted
Customer 4 Releases [1 0 0 1 1]
Customer 4 Releases [0 0 0 0 0]
Customer 4 Releases [0 0 0 0 0]
Customer 4 Releases [0 0 0 0 0]
Customer 4 Requests [1 0 4 2 2] -> accepted
Customer 4 Requests [0 1 1 0 0] -> accepted
Customer 4 Requests [0 0 0 1 1] -> accepted
Customer 4 Requests [0 0 1 0 0] -> accepted
Customer 4 Requests [0 0 0 0 0] -> accepted
Customer 5 Requests [1 7 1 0 5] -> not accepted
Customer 5 Requests [1 7 11 0 0] -> not accepted
Customer 5 Requests [1 1 1 0 2] -> accepted
Customer 5 Requests [1 0 4 0 2] -> not accepted
Customer 5 Requests [1 0 2 1 2] -> not accepted
Customer 5 Requests [0 2 9 0 3] -> not accepted
Customer 5 Requests [1 3 2 0 1] -> not accepted
Customer 5 Releases [0 0 1 0 0]
Customer 5 Requests [0 3 7 1 1] -> not accepted
Customer 5 Requests [0 2 5 1 2] -> not accepted
Customer 3 Releases [0 0 0 0 0]
Customer 3 Requests [0 2 2 0 4] -> not accepted
Customer 3 Requests [0 6 8 5 3] -> not accepted
Customer 3 Requests [4 0 10 3 1] -> not accepted
Customer 3 Releases [0 0 0 0 0]
Customer 3 Requests [4 4 4 2 3] -> not accepted
```


Customer 3 Requests [4 0 10 3 1] -> not accepted
Customer 3 Releases [0 0 0 0 0]
Customer 3 Requests [4 4 4 2 3] -> not accepted
Customer 1 Releases [0 0 0 0 0]
Customer 1 Releases [0 0 0 0 0]
Customer 1 Requests [0 3 11 0 0] -> not accepted
Customer 1 Requests [0 5 2 0 0] -> accepted
Customer 1 Requests [0 0 7 0 1] -> not accepted
Customer 1 Requests [2 0 9 0 2] -> not accepted
Customer 2 Releases [0 0 0 0 0]
Customer 2 Releases [0 0 0 0 0]
Customer 2 Releases [0 0 0 0 0]
Customer 2 Requests [2 1 0 0 1] -> not accepted
Customer 2 Requests [1 2 0 6 0] -> not accepted
Customer 3 Requests [3 1 7 4 0] -> not accepted
Customer 1 Requests [2 0 4 1 2] -> not accepted
Customer 1 Releases [0 1 2 0 0]
Customer 1 Requests [0 0 0 1 1] -> not accepted
Customer 1 Releases [0 0 0 0 0]
Customer 2 Requests [2 2 1 6 2] -> not accepted
Customer 2 Releases [0 0 0 0 0]
Customer 2 Releases [0 0 0 0 0]
Customer 2 Releases [0 0 0 0 0]
Customer 2 Requests [1 3 1 5 2] -> not accepted
Customer 3 Releases [0 0 0 0 0]
Customer 3 Releases [0 0 0 0 0]
Customer 3 Requests [0 1 0 0 2] -> not accepted
forough@forough-VirtualBox:~/Desktop/lab7$