Operating systems and system programming

Project

- 5 projects, three of which must be completed
- Scoring 0 5 p, 7+, and at least 1 point for each project.
- A maximum of 2 people are allowed in each group
- The deadline is the last week of this course

Project 1 Mutual exclusion and synchronization with semaphores

POSIX semaphores come in two forms: named semaphores and unnamed semaphores. Unnamed semaphores are mainly used for synchronization between threads, and can also be used for synchronization between processes (generated by fork). Named semaphores can be used for synchronization between processes and between threads. Named semaphores mainly include sem_open, sem_post, sem_wait, sem_close, and sem_unlink.

sem_t *sem_open(const char *name, int oflag);
sem_t *sem_open(const char *name, int oflag, mode_t mode, unsigned int value);
sem_open() creates a new POSIX semaphore or opens an existing semaphore. The semaphore is identified by *name*. The *oflag* argument specifies flags that control the operation of the call. If O_CREAT is specified in *oflag*, then the semaphore is created if it does not already exist. If both O_CREAT and O_EXCL are specified in *oflag*, then an error is returned if a semaphore with the given name already exists. If O_CREAT is specified in *oflag*, then two additional arguments must be supplied. The mode argument specifies the permissions to be placed on the new semaphore. The value argument specifies the initial value for the new semaphore. If O_CREAT is specified, and a semaphore with the given name already exists, then mode and value are ignored.

int sem_wait(sem_t *sem);
sem_wait() decrements (locks) the semaphore pointed to by sem. If the semaphore's value is greater than zero, then the decrement proceeds, and the function returns, immediately. If the semaphore currently has the value zero, then the call blocks until either it becomes possible to perform the decrement., or a signal handler interrupts the call.

int sem_post(sem_t *sem);
sem_post() increments (unlocks) the semaphore pointed to by sem. If the semaphore's value consequently becomes greater than zero, then another process or thread blocked in a sem_wait call will be woken up and proceed to lock the semaphore.

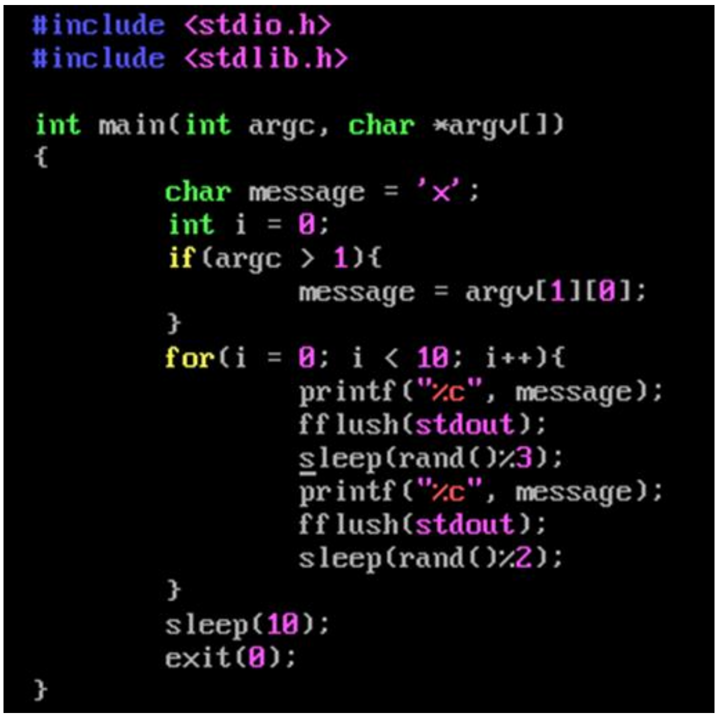
int sem_close(sem_t *sem);
sem_close() closes the named semaphore referred to by sem, allowing any resources that the system has allocated to the calling process for this semaphore to be freed.

int sem_unlink(const char *name);

sem_unlink() removes the named semaphore referred to by name. The semaphore name is removed immediately. The semaphore is destroyed once all other processes that have the semaphore open close it.

Task1

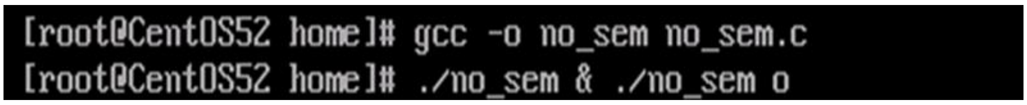
Observe the situation when mutex is not used through an example (assuming the file is named no_sem.c).



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

int main(int argc, char *argv[])
{
        char message = 'x';
        int i = 0;
        if(argc > 1){
                message = argv[1][0];
        }
        for(i = 0; i < 10; i++){
                printf("%c", message);
                fflush(stdout);
                sleep(rand()%3);
                printf("%c", message);
                fflush(stdout);
                sleep(rand()%2);
        }
        sleep(10);
        exit(0);
}
```

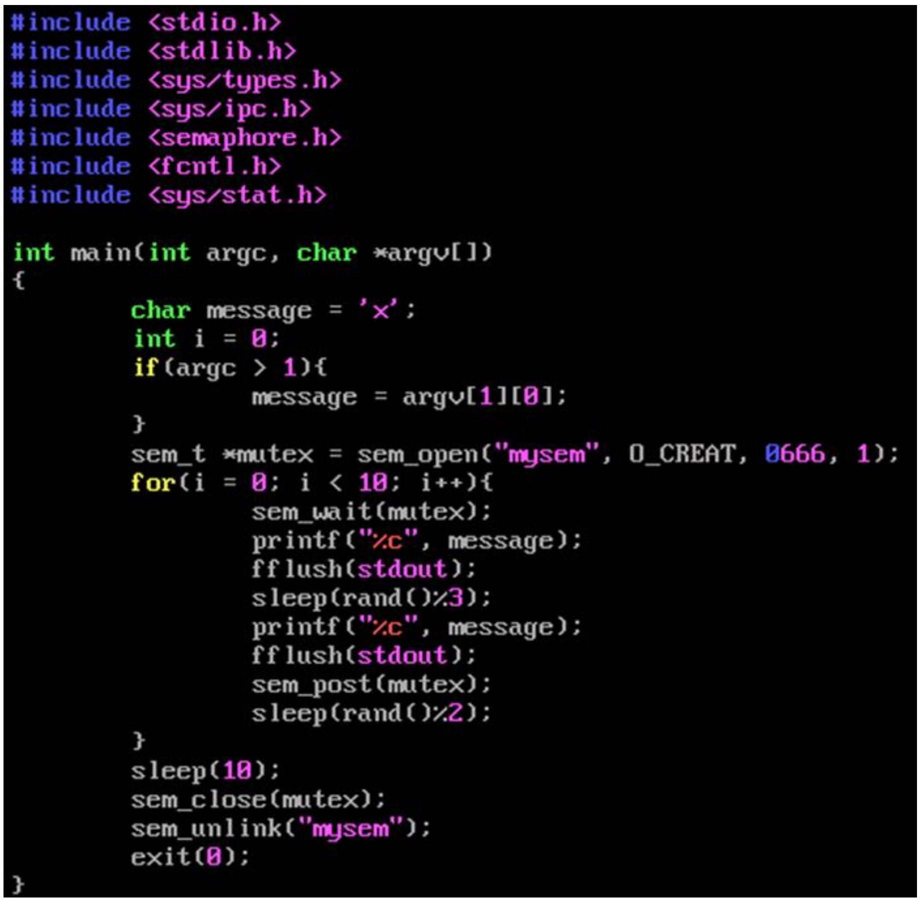
Compile and link, run two processes at the same time. Observe the occurrence rules of X and O, and analyze the reasons.



```
[root@CentOS52 home]# gcc -o no_sem no_sem.c
[root@CentOS52 home]# ./no_sem & ./no_sem o
```

Task2

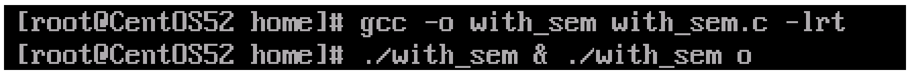
Use semaphores to mutex critical resources (assuming the file is named with_sem.c)



```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <semaphore.h>
#include <fcntl.h>
#include <sys/stat.h>

int main(int argc, char *argv[])
{
        char message = 'x';
        int i = 0;
        if(argc > 1){
                message = argv[1][0];
        }
        sem_t *mutex = sem_open("mysem", O_CREAT, 0666, 1);
        for(i = 0; i < 10; i++){
                sem_wait(mutex);
                printf("%c", message);
                fflush(stdout);
                sleep(rand()%3);
                printf("%c", message);
                fflush(stdout);
                sem_post(mutex);
                sleep(rand()%2);
        }
        sleep(10);
        sem_close(mutex);
        sem_unlink("mysem");
        exit(0);
}
```

Compile and link, run two processes at the same time. Observe the occurrence rules of X and O, and analyze the reasons.



```
[root@CentOS52 home]# gcc -o with_sem with_sem.c -lrt
[root@CentOS52 home]# ./with_sem & ./with_sem o
```

Task3

Simulate playing chess with semaphores, where red and black take turns.

Write two C language programs, black_chess.c and red_chess.c, to simulate that red moves and black moves in the process of playing chess respectively. Rules of movement: red first and black second; red and black take turns to move, until the 10th step, the red side wins and the black side loses.

Programming ideas:

Set up two synchronization semaphores.

(1) ***hei***: The initial value is 1, which means that the black side has already moved, and it is the red side's turn to move (meeting the chess rule "red first, black second").

(2) ***hong***: The initial value is 0, which means that the red side has not moved yet.

Before the red chess moves, test the semaphore ***hei*** to determine whether the black side has already moved. If so, it is the red side's turn to move, otherwise it is blocked and waits for the black side to move. Since the initial value of ***hei*** is 1, it must be the red side go first. After the red side moves, it sets the semaphore to notify the black side to move.

Before the black side moves, it first tests the semaphore ***hong*** to determine whether the red side has already moved. If so, it is the turn of the black side to move, otherwise block and wait for the red side to move. Since the initial value of ***hong*** is 0, the black side will not move before the red side moves. After the black side moves, set the semaphore ***hei*** to notify the red side to move.

Submission

1. Observe the occurrence rules of X and O in Task1 and Task2.
2. Code: black_chess.c and red_chess.c
3. Report: post and analyze the results.

https://linux.die.net/man/7/sem_overview

Project 2 Process synchronization implementation

Solve problems related to process synchronization with semaphores, such as: producer-consumer problems, philosophers' meals, etc.

1 The producer-consumer problem

A group of producer processes provides products to a group of consumer processes. The two types of processes share a bounded buffer pool composed of ***n*** buffers. The producer processes put products into the empty buffer pool, and the consumer processes take products from the buffer pool with data and consume them.

As long as the buffer pool is not full, the producer process can send products to the buffer pool; as long as the buffer pool is not empty, the consumer process can take products from the buffer pool. A producer process can only put one product at a time; a consumer process can only take one product at a time.

However, the producer process is prohibited from re-delivering products to the full buffer pool, and the consumer process is also prohibited from taking products from the empty buffer pool.

In order to prevent repeated operations on the buffer pool, it is stipulated that only one subject can access the buffer pool at any time.

2 The dining-philosophers problem

Consider five philosophers who spend their lives thinking and eating. The philosophers share a circular table surrounded by five chairs, each belonging to one philosopher. In the center of the table is a bowl of rice, and the table is laid with five single chopsticks (Figure 1). When a philosopher thinks, she does not interact with her colleagues. From time to time, a philosopher gets hungry and tries to pick up the two chopsticks that are closest to her (the chopsticks that are between her and her left and right neighbors). A philosopher may pick up only one chopstick at a time. Obviously, she cannot pick up a chopstick that is already in the hand of a neighbor. When a hungry philosopher has both her chopsticks at the same time, she eats without releasing the chopsticks. When she is finished eating, she puts down both chopsticks and starts thinking again.

The dining-philosophers problem is considered a classic synchronization problem neither because of its practical importance nor because computer scientists dislike philosophers but because it is an example of a large class of concurrency-control problems. It is a simple representation of the need to allocate several resources among several processes in a deadlock-free and starvation-free manner.

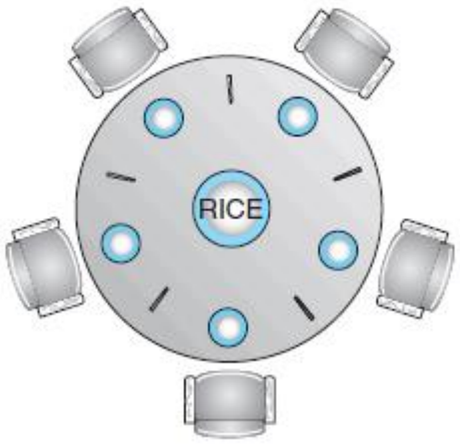


Fig 1. The situation of the dining philosophers

Task1

Write a C program to solve the producer-consumer problem based on semaphores.

Task2

Write a C program to solve the dining-philosophers problem based on semaphores.

Submission

1. Code: The programs of Task1 and Task2
2. Report: post and explain the results

Project 3 Job Scheduling Algorithm Simulation

The scheduling algorithms used in this project default to non-preemptive scheduling. The test data used in the experiment is shown in the table below.

Job number	Arrival time	Execution time	Priority
1	800	50	0
2	815	30	1
3	830	25	2
4	835	20	2
5	845	15	2
6	700	10	1
7	820	5	0

The data structure of jobs:

```
typedef struct node
{
    int number; // job number
    int reach_time;// arrival time
    int need_time;// execution time
    int privilege;// priority
    float excellent;// response ratio
```

```
	int start_time;// start time
	int wait_time;// waiting time
	int visited;// if the job is accessed
	bool isreached;// if the job has arrived
}job;
```

The instruction of important functions

void initial_jobs()
initialize all job information

void reset_jinfo()
reset all job information

int findminjob(job jobs[],int count)
Find the job with the shortest execution time. The input parameters are all job information and the total number of jobs. The output is the job id with the shortest execution time.

int findrearlyjob(job jobs[],int count)
Find the job that arrived earliest. The input parameters are all job information and the total number of jobs. Return the id of the job that arrived earliest.

void readJobdata()
read the basic information of all jobs

void FCFS()
FCFS algorithm

void SFJschdulejob(job jobs[],int count)
Shortest-Job-First Scheduling algorithm. The input parameters are all job information and the total number of jobs.

Code for reference

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
const int MAXJOB=50; // maximum number of jobs
//data structure of jobs
typedef struct node
{
	int number; // job number
	int reach_time;// arrival time
	int need_time;// execution time
	int privilege;// priority
	float excellent;// response ratio
	int start_time;// start time
```

```
    int wait_time;// waiting time
    int visited;// if the job is accessed
    bool isreached;// if the job has arrived
}job;
job jobs[MAXJOB];//job sequence
int quantity;//number of jobs
//initialize job sequence
void initial_jobs()
{
    int i;
    for(i=0;i<MAXJOB;i++)
    {
        jobs[i].number=0;
        jobs[i].reach_time=0;
        jobs[i].privilege=0;
        jobs[i].excellent=0;
        jobs[i].start_time=0;
        jobs[i].wait_time=0;
        jobs[i].visited=0;
        jobs[i].isreached=false;
    }
    quantity=0;
}
//reset all job information
void reset_jinfo()
{
    int i;
    for(i=0;i<MAXJOB;i++)
    {
        jobs[i].start_time=0;
        jobs[i].wait_time=0;
        jobs[i].visited=0;
    }
}
// Find the shortest job that has reached but not yet been executed. If not return -1
int findminjob(job jobs[],int count)
{
    int minjob=-1;//=jobs[0].need_time;
    int minloc=-1;
    for(int i=0;i<count;i++)
    {
        if(minloc==-1){
            if(    jobs[i].isreached==true && jobs[i].visited==0){
            minjob=jobs[i].need_time;
```

```
                minloc=i;
                }
            }
            else if(minjob>jobs[i].need_time&&jobs[i].visited==0&&jobs[i].isreached==true)
            {
                minjob=jobs[i].need_time;
                minloc=i;
            }
        }
        return minloc;
}
//Find the job that arrived ealiest
int findrearlyjob(job jobs[],int count)
{
        int rearlyloc=-1;
        int rearlyjob=-1;
        for(int i=0;i<count;i++)
        {
            if(rearlyloc==-1){
                if(jobs[i].visited==0){
                rearlyloc=i;
                rearlyjob=jobs[i].reach_time;
                }
            }
            else if(rearlyjob>jobs[i].reach_time&&jobs[i].visited==0)
            {
                rearlyjob=jobs[i].reach_time;
                rearlyloc=i;
            }
        }
        return rearlyloc;
}
//read all job data
void readJobdata()
{
        FILE *fp;
        char fname[20];
        int i;
        //input the name of test file
        printf("please input job data file name\n");
        scanf("%s",fname);
        if((fp=fopen(fname,"r"))==NULL)
        {
            printf("error, open file failed, please check filename:\n");
```

```
	}
	else
	{
		//read job information sequentially
		while(!feof(fp))
		{
	if(fscanf(fp,"%d %d %d %d",&jobs[quantity].number,&jobs[quantity].reach_time,&jobs[qua
ntity].need_time,&jobs[quantity].privilege)==4)
			quantity++;
		}
		//print job information
		printf("output the origin job data\n");
		printf("-------------------------------------------------------------
----\n");
		printf("\tjobID\treachtime\tneedtime\tprivilege\n");
		for(i=0;i<quantity;i++)
		{
	printf("\t%-8d\t%-8d\t%-8d\t%-
8d\n",jobs[i].number,jobs[i].reach_time,jobs[i].need_time,jobs[i].privilege);
		}
	}
}
//FCFS
void FCFS()
{
	int i;
	int current_time=0;
	int loc;
	int total_waitime=0;
	int total_roundtime=0;
	loc=findrearlyjob(jobs,quantity);
	//print job stream
	printf("\n\nFCFS job stream\n");
	printf("-----------------------------------------------------------------
---\n");
	printf("\tjobID\treachtime\tstarttime\twaittime\troundtime\n");
	current_time=jobs[loc].reach_time;
	// Each loop finds the first arriving job and prints information about it
	for(i=0;i<quantity;i++)
	{
		if(jobs[loc].reach_time>current_time)
		{
			jobs[loc].start_time=jobs[loc].reach_time;
			current_time=jobs[loc].reach_time;
		}
		else
		{
			jobs[loc].start_time=current_time;
		}
		jobs[loc].wait_time=current_time-jobs[loc].reach_time;
	printf("\t%-8d\t%-8d\t%-8d\t%-8d\t%-
8d\n",loc+1,jobs[loc].reach_time,jobs[loc].start_time,jobs[loc].wait_time,
			jobs[loc].wait_time+jobs[loc].need_time);
		jobs[loc].visited=1;
		current_time+=jobs[loc].need_time;
		total_waitime+=jobs[loc].wait_time;
		total_roundtime=total_roundtime+jobs[loc].wait_time+jobs[loc].need_time;
		// Get the first arriving job among the remaining jobs
		loc=findrearlyjob(jobs,quantity);
	}
	printf("total waiting time:%-8d total turnaround time:%-8d\n", total_waitime,
total_roundtime);
	printf("average waiting time: %4.2f average turnaround time: %4.2f\n",
(float)total_waitime/(quantity),(float)total_roundtime/(quantity));
}
// Shortest-Job-First Scheduling algorithm.
void SFJschdulejob(job jobs[],int count)
{

}
// Highest Response Ratio Next
//response ratio=turnaround time/execution time
void HRRFschdulejob()
{
}
// Priority scheduling algorithm
void HPF(job jobs[])
{

}

int main()
{
	initial_jobs();
	readJobdata();
	FCFS();
	reset_jinfo();
	SFJschdulejob(jobs, quantity);
```

```
    system("pause");
    return 0;
}
```

Task1

Check whether the job information is read correctly by printing the information of the program.

Task2

Run the FCFS algorithm to check whether the operation result is correct.

Task 3

Supplement the code of Shortest-Job-First Scheduling algorithm, and calculate its waiting time and turnaround time (total and average).

Task4

Referring to the implementation method of the above algorithms, write the code for highest response ratio next algorithm and priority schedule algorithm.

Submission

1. Code: The program for this project
2. Report: post and explain the results

Project 4 Simulation of Dynamic Partition Allocation

Assume that in the initial state, the available memory space is 640KB, and there is the following request sequence:

•job1 applies for 130KB。
•job2 applies for 60KB。
•job3 applies for 100KB。
•job2 releases 60KB。
•job4 applies for 200KB。
•job3 releases 100KB。
•job1 releases 130KB。
•job5 applies for 140KB。
•job6 applies for 60KB。
•job7 applies for 50KB。
•job6 releases 60KB。

Task

The allocation process alloc( ) and the recycling process free( ) for dynamic partition are respectively implemented based on the first-fit algorithm and the best-fit algorithm in C language. Free partitions are managed through free partition chains. When allocating memory, the system gives priority to using the lower space of the free area. Shows the status of free partition chains after each allocation and recycling.

Submission

1. Code: The program for this project
2. Report: post and explain the results

Project 5 Simulate the page frame replacement

1. **Important data structures**
   1) **Page table**

```
typedef struct
{
  int vmn; // virtual page number
  int pmn; //physical page number
  int exist; //Whether to occupy a physical block
  int time;
  }vpage_item;
  vpage_item page_table[VM_PAGE];
```

time item is added to the page table for the replacement algorithm to select victim pages. In different replacement algorithms, ***time*** has different meanings.

In the LRU algorithm, ***time*** is the most recent access time. Every time the virtual page is accessed, ***time*** is set as the current access time. When select victim pages, select the one with the smallest time value, that is, the one that has not been used for the longest time.

In the FIFO algorithm, ***time*** is the time when the virtual page enters the memory. This flag is only set when the virtual page is brought into memory from external memory. When select victim pages, select the virtual page with the smallest ***time***, that is, the earliest virtual page that enters memory.

In the OPT algorithm, ***time*** has no meaning.

   2) **Physical page bitmap**

vpage_item * ppage_bitmap[PM_PAGE]

Code for reference

```
#include "stdafx.h"
#include<stdio.h>
#include<stdlib.h>
#define VM_PAGE 7           /*Number of virtual pages*/
#define PM_PAGE 4               /* Number of memory blocks allocated to the job */
#define TOTAL_INSERT 18 /*Number of virtual page replacements*/
typedef struct
{
    int vmn;
    int pmn;
    int exist;
    int time;
}vpage_item;
vpage_item page_table[VM_PAGE];

vpage_item* ppage_bitmap[PM_PAGE];
```

```
int vpage_arr[TOTAL_INSERT] = { 1,2,3,4,2,6,2,1,2,3,7,6,3,2,1,2,3,6 }; // The access order virtual
pages


void init_data() //initialize data
{
	for (int i = 0; i<VM_PAGE; i++)
	{
		page_table[i].vmn = i + 1;
		page_table[i].pmn = -1;
		page_table[i].exist = 0;
		page_table[i].time = -1;

	}
	for (int i = 0; i<PM_PAGE; i++) /*initialize the physical page map*/
	{
		ppage_bitmap[i] = NULL;
	}
}

void FIFO()/*FIFO page replacement algorithem*/
{
	int k = 0;
	int i;
	int sum = 0;
	int missing_page_count = 0;
	int current_time = 0;
	bool isleft = true;	/* Whether there are remaining physical blocks */
	while (sum < TOTAL_INSERT)
	{
		if (page_table[vpage_arr[sum] - 1].exist == 0)
		{
			missing_page_count++;
			if (k < 4)
			{
				if (ppage_bitmap[k] == NULL) /*find a free block*/
				{
					ppage_bitmap[k] = &page_table[vpage_arr[sum] - 1];
					ppage_bitmap[k]->exist = 1;
					ppage_bitmap[k]->pmn = k;
					ppage_bitmap[k]->time = current_time;
					k++;
				}
			}
```

```
                else
                {
                    int temp = ppage_bitmap[0]->time;
                    int j = 0;
                    for (i = 0; i < PM_PAGE; i++)
                    {
                        if (ppage_bitmap[i]->time < temp)
                        {
                            temp = ppage_bitmap[i]->time;
                            j = i;
                        }
                    }
                    ppage_bitmap[j]->exist = 0;
                    ppage_bitmap[j] = &page_table[vpage_arr[sum] - 1]; /*update page table
*/
                    ppage_bitmap[j]->exist = 1;
                    ppage_bitmap[j]->pmn = j;
                    ppage_bitmap[j]->time = current_time;
                }
            }
            current_time++;
            sum++;
        }
        printf("The number of page faults of FIFO is:%d\t Page fault rate:%f\t The number of
replacement:%d\tReplacement rate:%f", missing_page_count, missing_page_count /
(float)TOTAL_INSERT, missing_page_count - 4, (missing_page_count - 4) /
(float)TOTAL_INSERT);
}
void LRU()
{
}
void OPT()
{
}

int main()
{
    int a;
    printf("Please choose page replacement algorithm： 1.FIFO\t2.LRU\t3.OPT\t0. quit\n");
    do
    {
        scanf_s("%d", &a);
        switch (a)
        {
```

```
		case 1:
			init_data();
			FIFO();
			break;
		case 2:
			init_data();
			LRU();
			break;
		case 3:
			init_data();
			OPT();
			break;
		}
	} while (a != 0);
	return 0;
}
```

Task

Complete the LRU and OPT algorithm code

Submission

1) Report: post the experimental results; analyze the reasons for the experimental results, and analyze the differences in the page fault rates of the three replacement algorithms.

2) The code for this project.