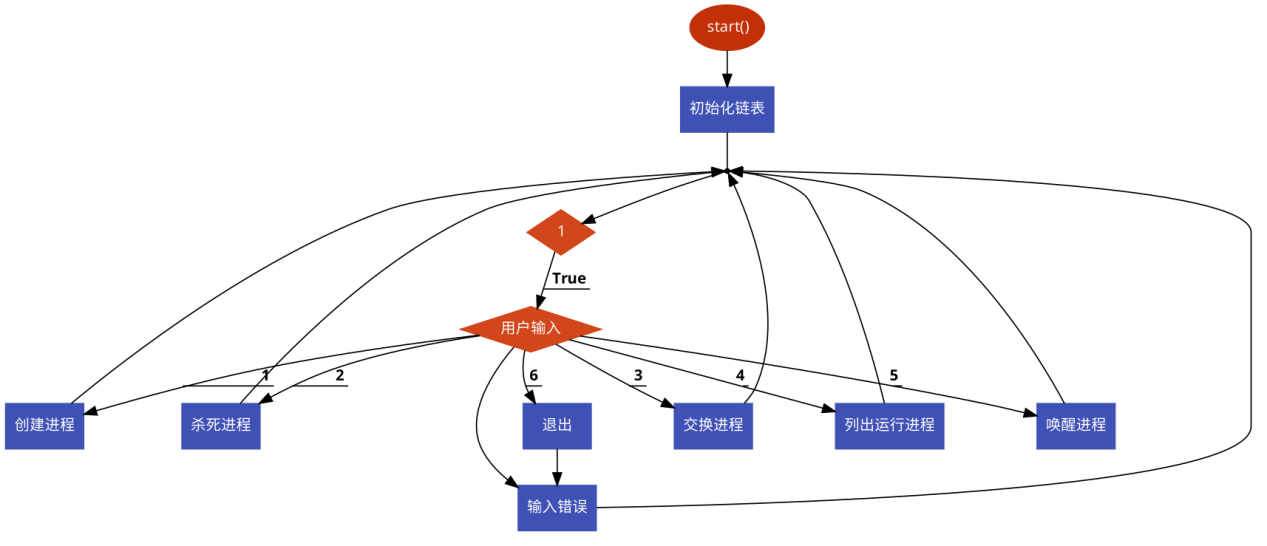
# 实验1

1. 实验分析

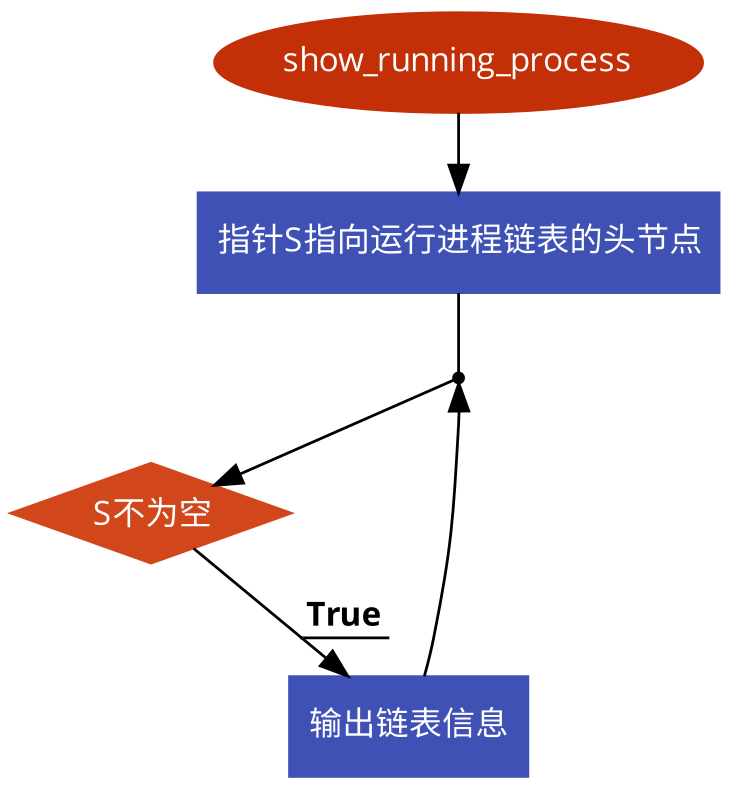
使用struct结构体创建构造PCB, 使用链表分别创建运行进程链表和阻塞进程链表。创建进程为进程分配内存和创建进程PCB，杀死进程时释放内存和PCB数据结构。

1. 流程图

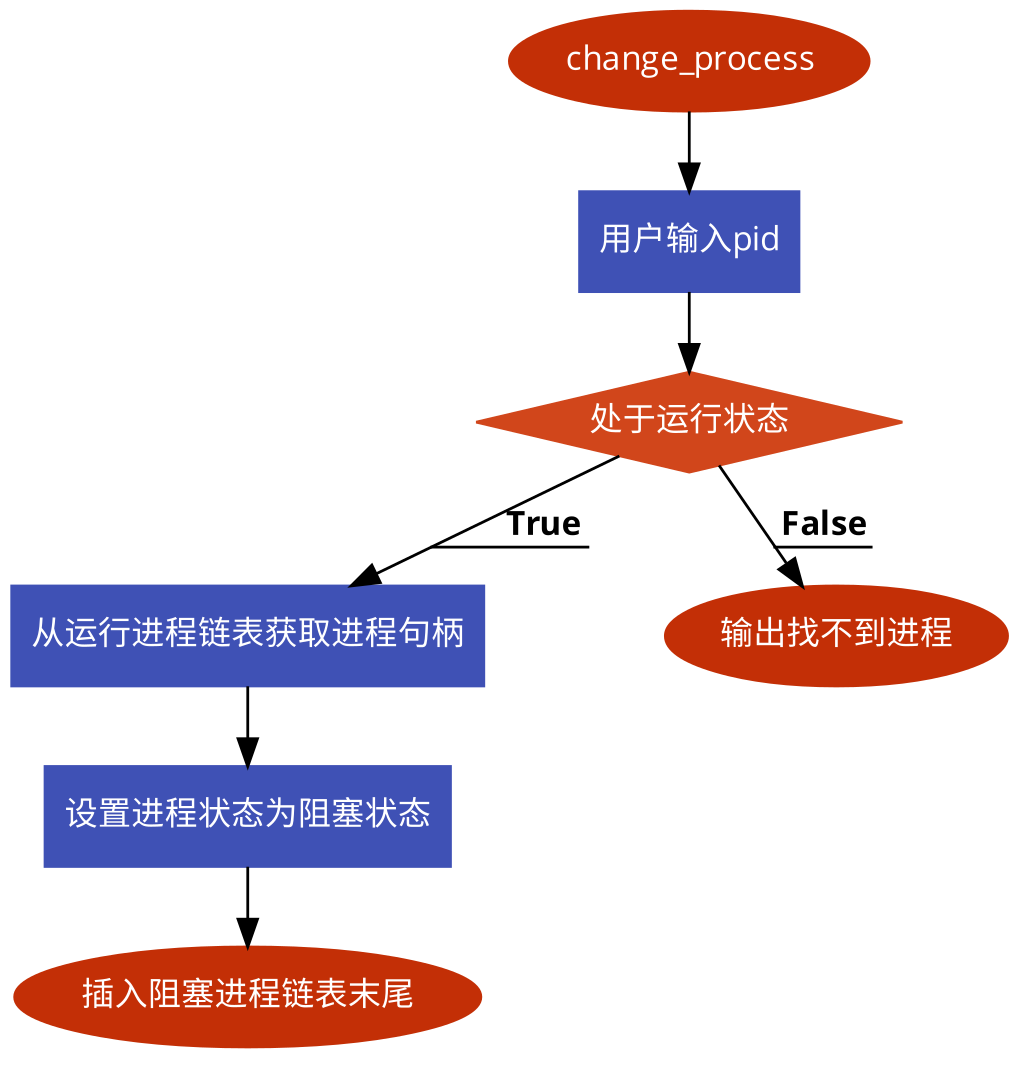
主函数:



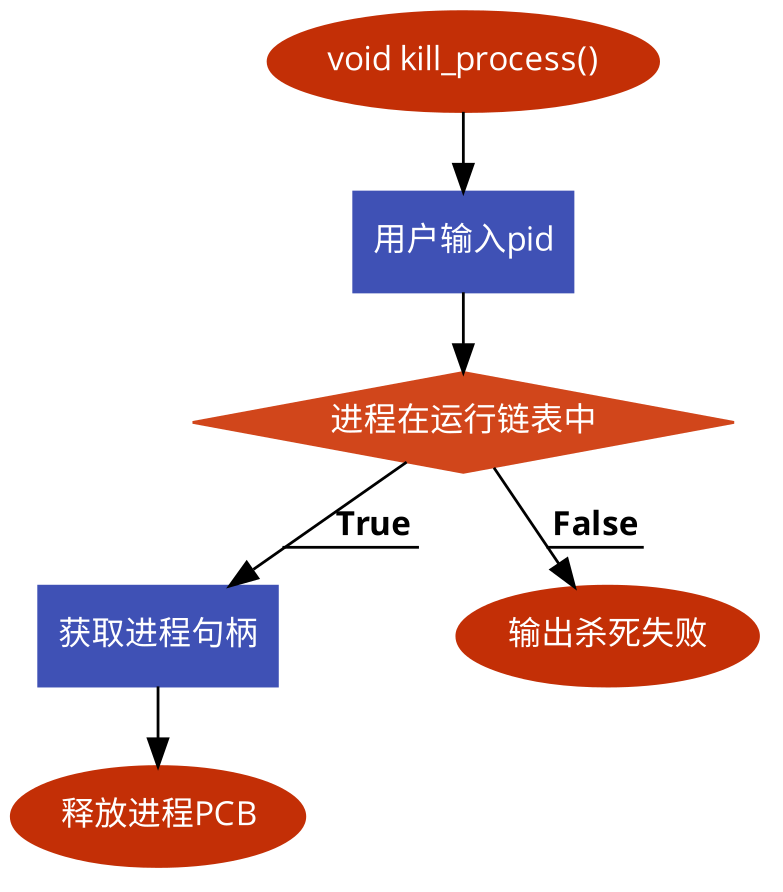
Show\_running\_process函数:



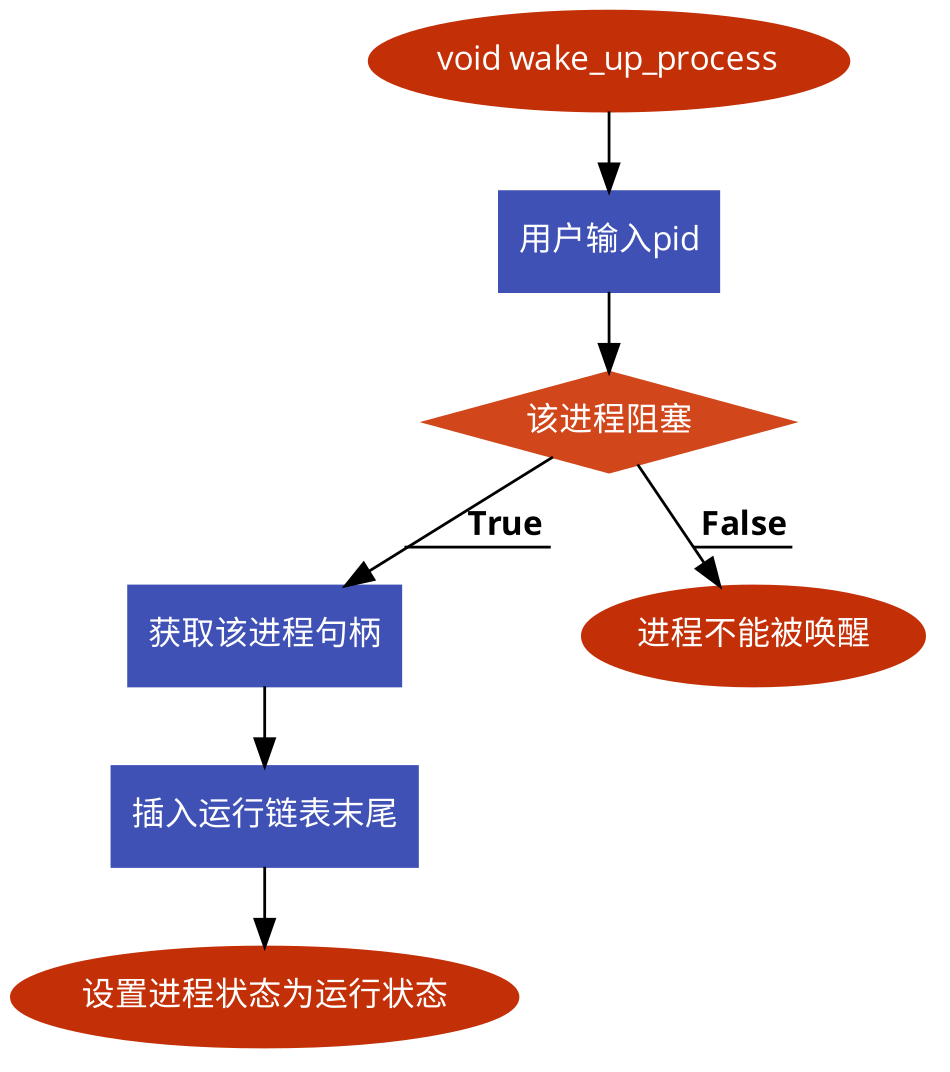
Change process:



Kill process:



Wake up process:



10

1. 源代码

ProcessManagement.h

#ifndef ProcessManagement\_h

#define ProcessManagement\_h

#include <stdio.h>

#define MAX\_SIZE 20

struct \_Tunnel {

int id;

char name[MAX\_SIZE];

char buffer[MAX\_SIZE];

};

*// tunnel to communicate between process*

typedef struct \_Tunnel Tunnel;

enum process\_type{

process\_type\_running = 1000,

process\_type\_block = -1000,

process\_type\_ready = 500

};

struct \_PCB {

int pid;

int size;

int state;

int priority;

char data[20];

*// mutex to communicate*

int mutex;

int state\_internal;

struct \_PCB \*next;

};

typedef struct \_PCB PCB;

PCB \*find\_process(PCB \*list, size\_t pid);

void create\_process(PCB \*running\_list, PCB \*block\_list, size\_t \*size);

void show\_running\_process(PCB \*running\_list);

void change\_process(PCB \*running\_list, PCB \*block\_list, size\_t \*size);

void kill\_process(PCB \*running\_list, size\_t \*size);

void wake\_up\_process(PCB \*running\_list, PCB \*block\_list, size\_t \*size);

int is\_process\_block(PCB \*block\_list, size\_t pid);

int is\_process\_running(PCB \*running\_list, size\_t pid);

#endif

ProcessManagement.c

#include <stdio.h>

#include <stdlib.h>

#include "ProcessManagement.h"

void create\_process(PCB \*process\_list\_runnning, PCB \*process\_list\_block, size\_t \*size)

{

size\_t pid;

printf("input pid:\n");

scanf("%zu",&pid);

if (\*size >= MAX\_SIZE) {

fprintf(stderr, "[-] not enough memory");

exit(EXIT\_FAILURE);

}

*// alloc memory for process*

PCB \*p = (PCB \*)malloc(sizeof(PCB));

if (p == NULL) {

perror("malloc");

}

if (is\_process\_block(process\_list\_block, pid)) {

puts("[+] process is in blocking list\n");

return;

}

if (is\_process\_running(process\_list\_runnning, pid)) {

puts("[+] process is in running list\n");

return;

}

p->pid = pid;

*// get info for process*

printf("input priority:\n");

scanf("%d",&p->priority);

printf("process data:\n");

scanf("%s",p->data);

printf("input size(running time):\n");

scanf("%d",&p->size);

*// set process status*

p->state = process\_type\_running;

\*size = \*size + 1;

*// end list*

p->next = NULL;

PCB \*tmp = process\_list\_runnning;

while (tmp->next != NULL) {

tmp = tmp->next;

}

tmp->next = p;

}

void show\_running\_process(PCB \*process\_list\_runnning)

{

PCB \*s = process\_list\_runnning->next;

while (s != NULL) {

printf("id:%d\n",s->pid);

printf("priority:%d\n",s->priority);

printf("size:%d\n",s->size);

printf("data:%s\n",s->data);

s = s->next;

}

}

void change\_process(PCB \*process\_list\_runnning, PCB \*process\_list\_block, size\_t \*size)

{

size\_t pid;

printf("input id:\n");

scanf("%zu",&pid);

if (\*size == 0) {

printf("[-] no process\n");

return;

}

if (is\_process\_running(process\_list\_runnning, pid)) {

PCB \*p = process\_list\_block;

PCB \*s = find\_process(process\_list\_runnning, pid);

\*size -= 1;

s->next->state = process\_type\_block;

while (p->next != NULL) {

p = p->next;

}

p->next = s->next;

s->next = s->next->next;

p->next->next = NULL;

printf("[+] succeed\n");

} else {

printf("[-] process not found\n");

}

}

void kill\_process(PCB \*process\_list\_runnning, size\_t \*size)

{

size\_t pid;

printf("input id:\n");

scanf("%lu",&pid);

if (\*size == 0) {

printf("[-] no process found\n");

return;

}

if (is\_process\_running(process\_list\_runnning, pid)) {

*// minus counter first*

\*size = \*size - 1;

PCB \*p, \*thread;

p = find\_process(process\_list\_runnning, pid);

thread = p->next;

p->next = p->next->next;

free(thread);

printf("[+] success\n");

} else {

printf("[-] not found\n");

}

}

void wake\_up\_process(PCB \*process\_list\_runnning,

PCB \*process\_list\_block, size\_t \*size){

size\_t pid;

printf("input id:\n");

scanf("%zu",&pid);

PCB \*s = process\_list\_block;

if (s->next == NULL) {

printf("[-] no process found\n");

return;

}

if (is\_process\_block(process\_list\_block, pid)) {

PCB \*p = process\_list\_runnning;

s = find\_process(process\_list\_block, pid);

s->next->state = process\_type\_running;

while (p->next != NULL) {

p = p->next;

}

p->next = s->next;

s->next = s->next->next;

p->next->next = NULL;

\*size = \*size + 1;

printf("[+] success\n");

}else{

printf("[+] process not found\n");

}

}

int is\_process\_running(PCB \*process\_list\_runnning, size\_t pid)

{

int flag = 0;

PCB \*tmp = process\_list\_runnning->next;

while (tmp != NULL) {

if (tmp->pid == pid) {

flag = 1;

break;

}

tmp = tmp->next;

}

return flag;

}

int is\_process\_block(PCB \*process\_list\_block, size\_t pid)

{

int flag = 0;

PCB \*s = process\_list\_block->next;

while (s != NULL) {

if (s->pid == pid) {

*// found*

flag = 1;

break;

}

*// point to next node*

s = s->next;

}

return flag;

}

PCB \*find\_process(PCB \*list, size\_t pid)

{

PCB \*s = list;

while (s->next != NULL) {

*// found*

if (s->next->pid == pid) {

return s;

}

s = s->next;

}

*// not found*

return NULL;

}

Main.c

#include <stdio.h>

#include <stdlib.h>

#include "ProcessManagement.h"

int main(int argc, const char \* argv[])

{

size\_t cnt = 0;

int choose = 1;

PCB \*running\_list = (PCB \*)malloc(sizeof(PCB));

if (running\_list == NULL) {

perror("malloc");

exit(EXIT\_FAILURE);

}

PCB \*block\_list = (PCB \*)malloc(sizeof(PCB));

if (running\_list == NULL) {

perror("malloc");

exit(EXIT\_FAILURE);

}

running\_list->next = NULL;

block\_list->next = NULL;

while (choose) {

fprintf(stdout, "1.create process\n");

fprintf(stdout, "2.kill process\n");

fprintf(stdout, "3.swap process\n");

fprintf(stdout, "4.show process\n");

fprintf(stdout, "5.wakeup process\n");

fprintf(stdout, "6.exit\n");

printf("input:\n");

scanf("%d",&choose);

switch (choose) {

case 1:

create\_process(running\_list, block\_list, &cnt);

break;

case 2:

kill\_process(running\_list, &cnt);

break;

case 3:

change\_process(running\_list, block\_list, &cnt);

break;

case 4:

show\_running\_process(running\_list);

break;

case 5:

wake\_up\_process(running\_list, block\_list, &cnt);

break;

case 6:

exit(EXIT\_SUCCESS);

default:

printf("input error\n");

break;

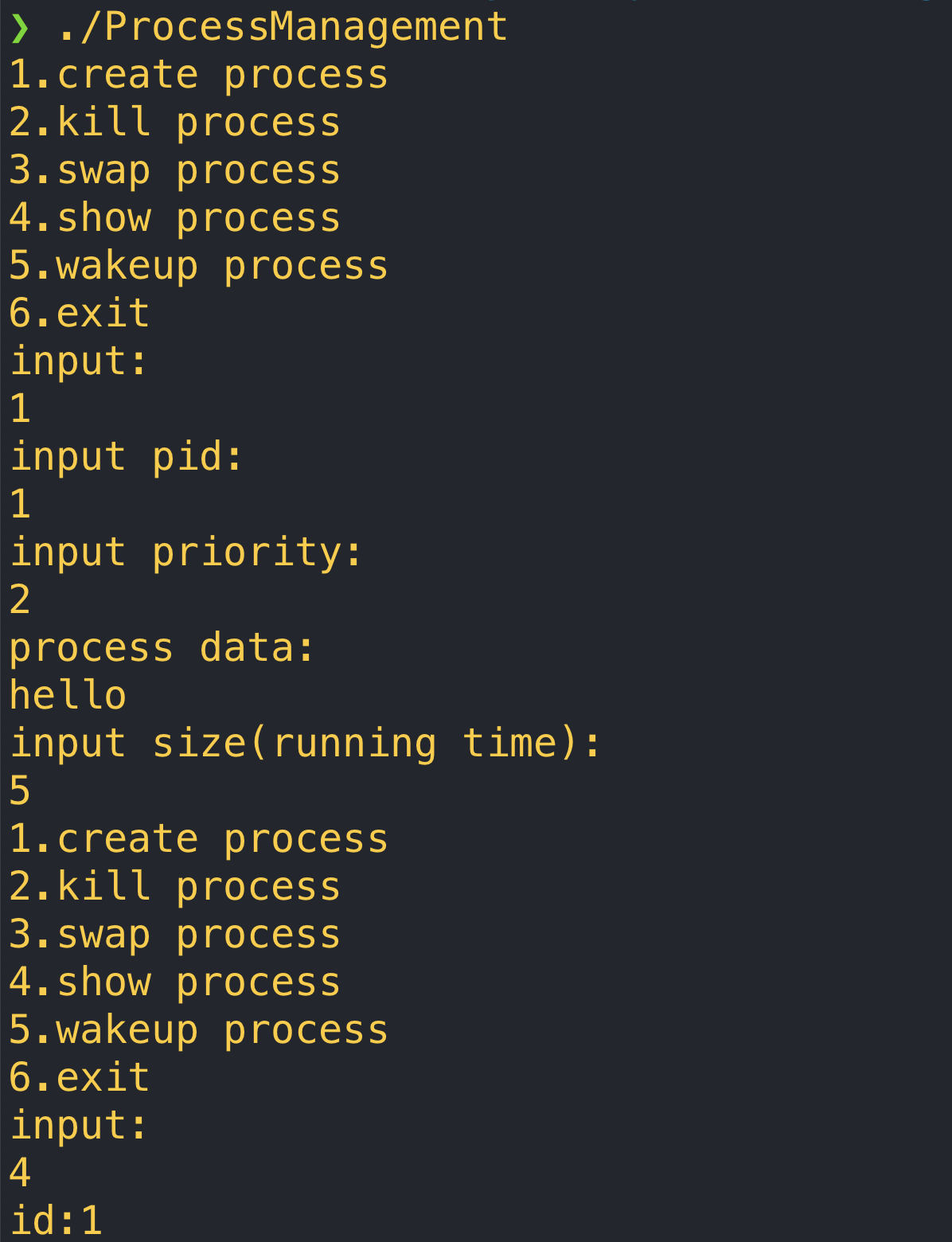
}

}

return 0;

}

1. 实验结果



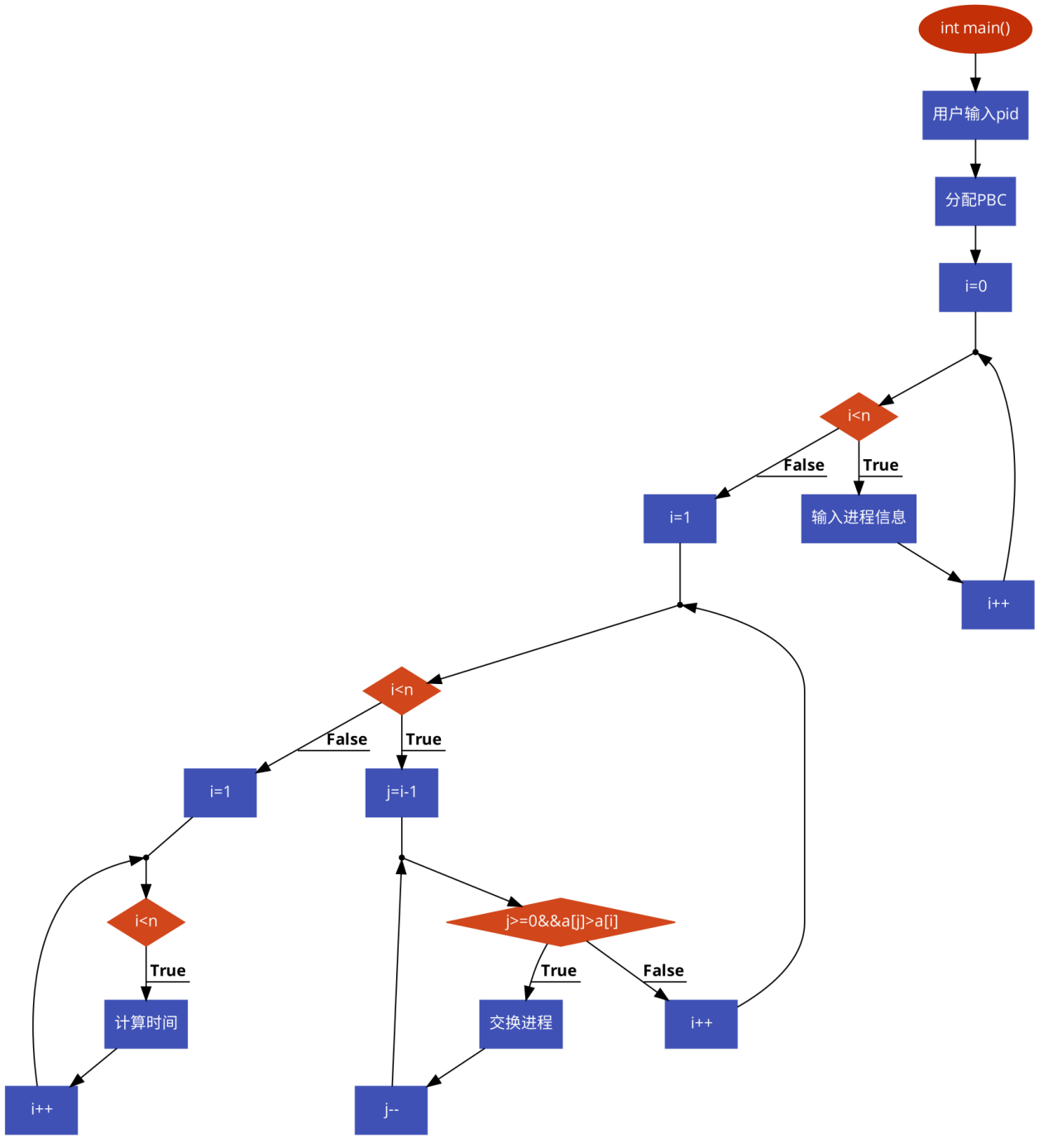
# 实验2

1. 实验分析

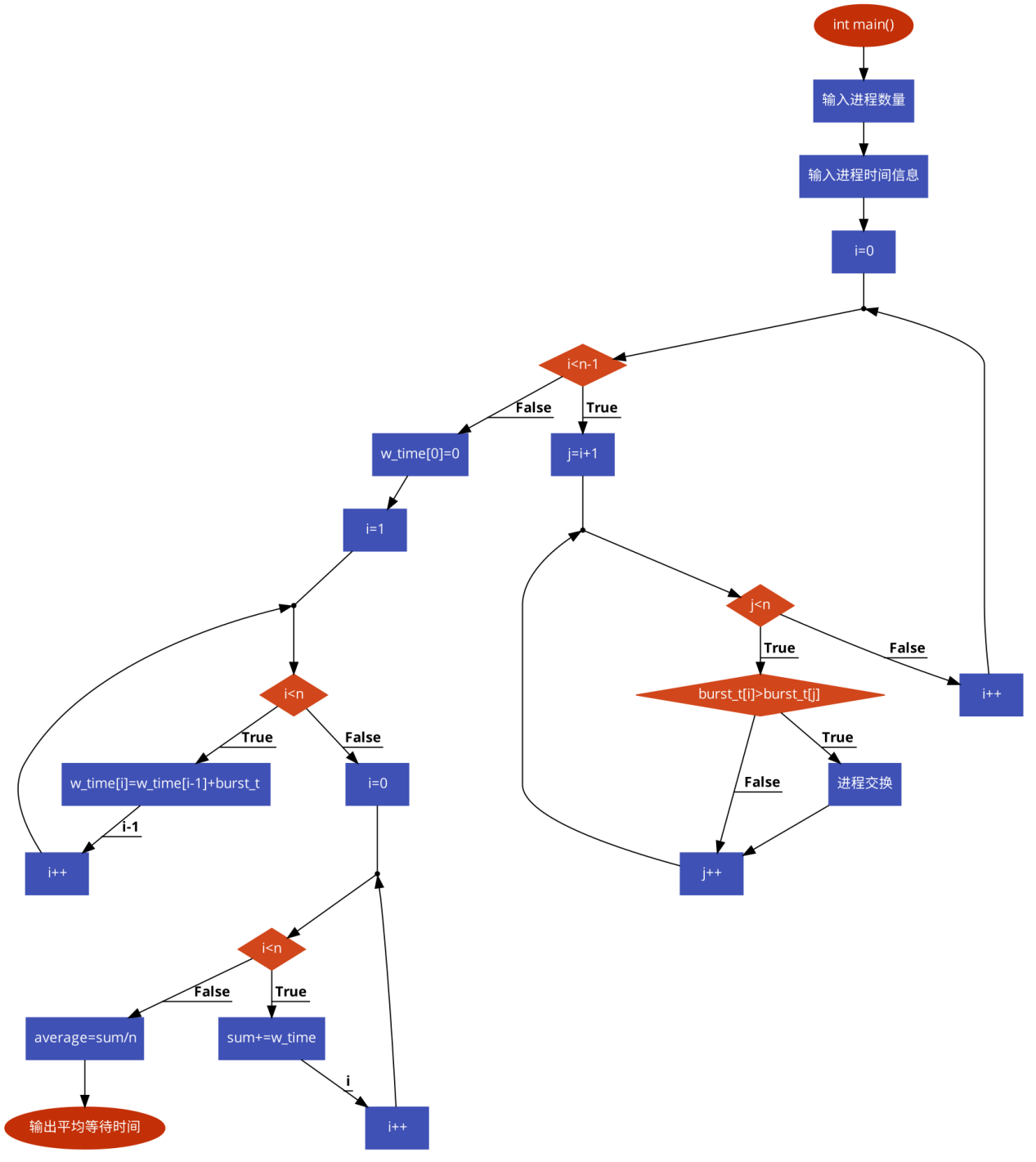
实现先来先服务，进程优先服务算法。

1. 流程图

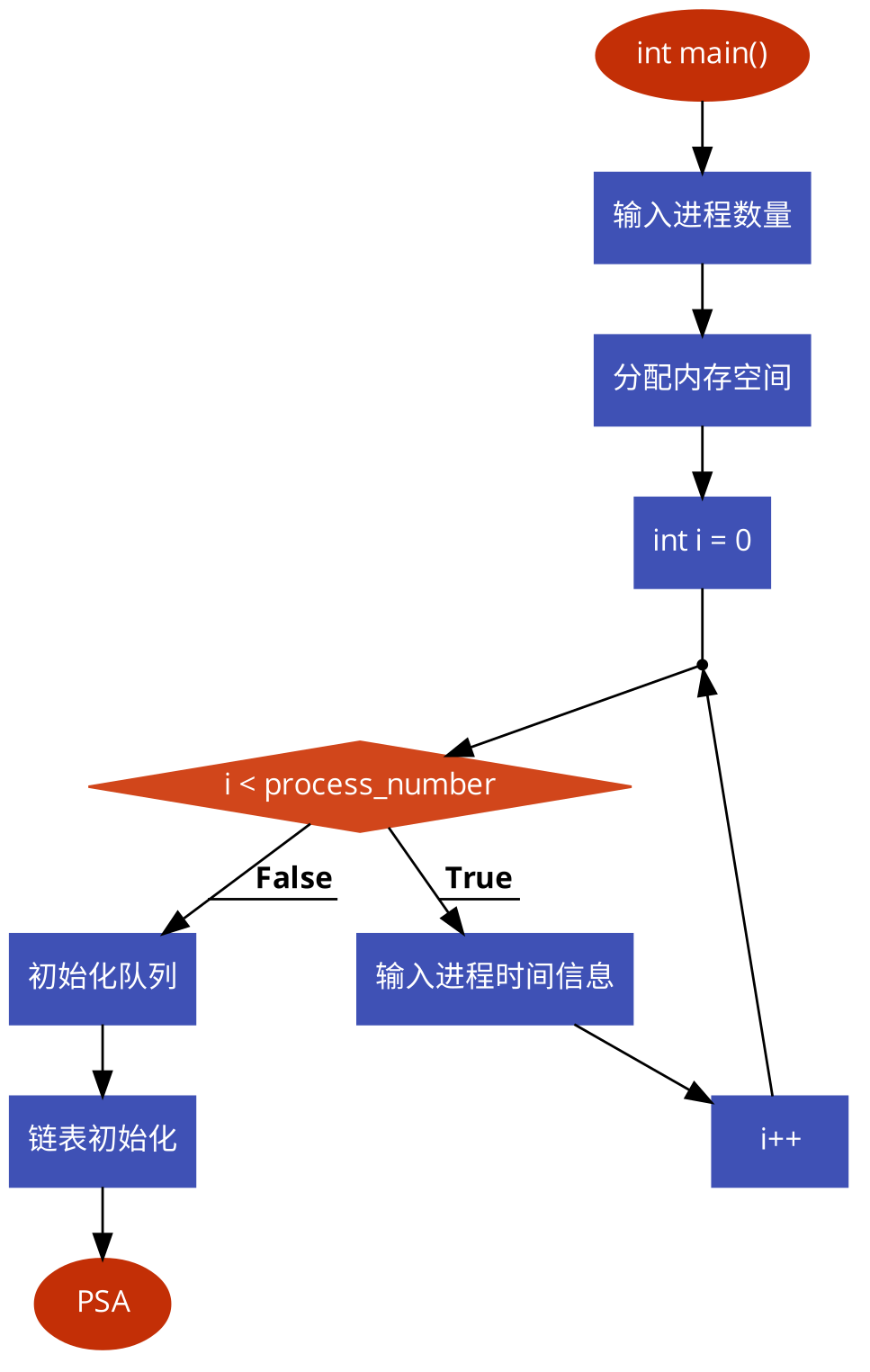
fcfs算法

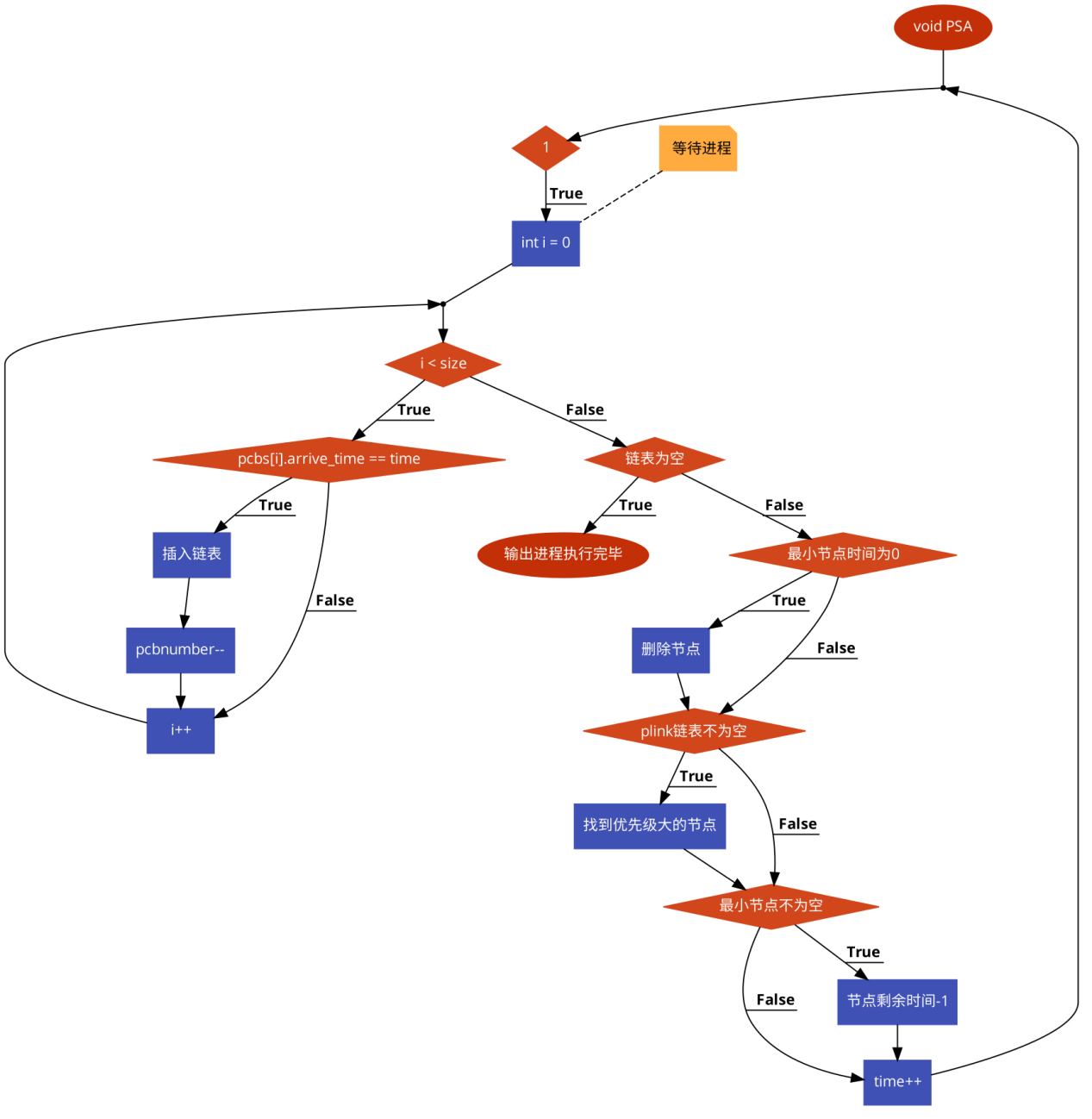


spf算法



PSA算法





1. 源代码

fcfs源代码

#include<stdio.h>

#include<stdlib.h>

void swap(int \*a,int \*b){

int temp;

temp=\*a;

\*a=\*b;

\*b=temp;

}

int main()

{

int n,i,j,w,t,s;

float avt=0,avw=0;

int \*p=NULL,\*a=NULL,\*b=NULL;

printf("Enter the number of processes:\n");

scanf("%d",&n);

a=(int \*)malloc(n\*sizeof(int));

if (a == NULL) {

perror("malloc");

exit(EXIT\_FAILURE);

}

b=(int \*)malloc(n\*sizeof(int));

if (b == NULL) {

perror("malloc");

exit(EXIT\_FAILURE);

}

p=(int \*)malloc(n\*sizeof(int));

if (p == NULL) {

perror("malloc");

exit(EXIT\_FAILURE);

}

for(i=0;i<n;i++){

printf("Enter process id, arrival time, burst time\n");

scanf("%d%d%d",&p[i],&a[i],&b[i]);

}

for(i=1;i<n;i++){

for(j=i-1;j>=0&&a[j]>a[i];j--){

swap(&a[i],&a[j]);

swap(&b[i],&b[j]);

swap(&p[i],&p[j]);

i--;

}

}

s=a[0];

w=s-a[0];

t=b[0]+s-a[0];

avt+=t;

avw+=w;

printf("Process\tAT\tBT\tWT\tTT\n");

printf("%d\t%d\t%d\t%d\t%d\n",p[0],a[0],b[0],w,t);

for(i=1;i<n;i++){

if(a[i]>t){

w=0;

t=t+b[i]-a[i];

printf("%d\t%d\t%d\t%d\t%d\n",p[i],a[i],b[i],w,t);

avw+=w;

avt+=t;

}

s=t+a[i-1];

w=s-a[i];

t=s+b[i]-a[i];

printf("%d\t%d\t%d\t%d\t%d\n",p[i],a[i],b[i],w,t);

avw+=w;

avt+=t;

}

avt/=n;

avw/=n;

printf("\nAverage waiting time = %f\n",avw);

printf("\nAverage turnaround time = %f\n",avt);

return 0;

}

PSA算法源代码

main函数

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include "head.h"

int main()

{

PCB \*pcbs;

Queue queue;

linkList linklist;

int process\_number = 0;

printf("please input process number: ");

scanf("%d", &process\_number);

pcbs = (PCB \*)malloc(sizeof(struct \_PCB) \* process\_number);

if (pcbs == NULL) {

perror("malloc");

exit(EXIT\_FAILURE);

}

memset(pcbs, 0, sizeof(PCB) \* process\_number);

for (int i = 0; i < process\_number; i++) {

printf("name: ");

scanf(" %c",&(pcbs[i].name));

printf("arrive: ");

scanf("%d",&(pcbs[i].arrive\_time));

printf("process->serve\_time: ");

scanf("%d",&(pcbs[i].serve\_time));

pcbs[i].left\_time = pcbs[i].serve\_time;

printf("priority: ");

scanf("%d",&(pcbs[i].Priority));

}

init\_queue(&queue);

ll\_init(&linklist);

PSA(&linklist, pcbs, process\_number);

ll\_destroy(&linklist);

remove\_queue(&queue);

return 0;

}

Process.c

#include <stdio.h>

#include <unistd.h>

#include <assert.h>

#include <string.h>

#include <stdlib.h>

#include <stdbool.h>

#include "head.h"

void PSA(linkList \*plinklist, PCB pcbs[], int size)

{

int time = 0;

int pcbid = 0;

int pcbnumber = size;

Node \*curnode = NULL;

Node \*minnode = NULL;

while (1) {

*//等待进程*

for (int i = 0; i < size; i++) {

if (pcbs[i].arrive\_time == time) {

ll\_push\_back(plinklist, pcbs[i]);

*// printf("queue\_get\_top(pqueue)->next == %p\n", queue\_get\_top(pqueue)->next);*

printf("PCB %c ARRIVAL TIME IS %d\n", pcbs[i].name, time);

pcbnumber--;

*// printf("pcbnumber == %d\n", pcbnumber);*

}

}

if (ll\_is\_empty(plinklist) && pcbnumber == 0) {

printf("进程执行完毕！\n");

break;

}

*// printf("time is %d \n", time);*

*//*

*//执行进程*

*//*

if (minnode != NULL && (minnode->data).left\_time == 0) {

printf("PCB %c FINISH TIME IS %d\n", (minnode->data).name, time);

ll\_remove(plinklist, minnode);

minnode = NULL;

}

if (!ll\_is\_empty(plinklist) && minnode == NULL) {

minnode = NULL;

curnode = plinklist->head;

while(curnode != NULL) {

if (minnode == NULL || curnode->data.Priority < minnode->data.Priority) {

minnode = curnode;

}

curnode = curnode->next;

}

}

if (minnode != NULL) {

--(minnode->data).left\_time;

}

time++;

*// sleep(1);*

}

}

void init\_queue(Queue \*pqueue) {

pqueue->front = NULL;

pqueue->rear = NULL;

pqueue->size = 0;

}

void remove\_queue(Queue \*pqueue)

{

Node \*current\_node = NULL;

Node \*p = NULL;

assert(pqueue != NULL);

if (pqueue->front == NULL) {

return;

}

current\_node = pqueue->front;

while (current\_node != NULL) {

p = current\_node;

current\_node = current\_node->next;

free(p);

p = NULL;

}

}

Node \*new\_node(DataType data)

{

Node \*ret = (Node \*)malloc(sizeof(Node));

if (ret == NULL) {

perror("malloc");

exit(EXIT\_FAILURE);

}

ret->data = data;

ret->next = NULL;

return ret;

}

void queue\_enqueue(Queue \*pqueue, DataType data)

{

Node \*temp = new\_node(data);

if (pqueue->front == NULL) {

pqueue->front = temp;

pqueue->rear = temp;

++pqueue->size;

}

else {

pqueue->rear->next = temp;

pqueue->rear = pqueue->rear->next;

++pqueue->size;

}

}

void queue\_dequeue(Queue \*pqueue)

{

Node \*p = NULL;

assert(pqueue != NULL);

if (pqueue->front == NULL) {

return;

}

p = pqueue->front;

pqueue->front = pqueue->front->next;

free(p);

p = NULL;

++pqueue->size;

}

Node \*queue\_get\_top(Queue \*pqueue)

{

return pqueue->front;

}

bool queue\_is\_empty(const Queue \*pqueue)

{

return pqueue->front == NULL;

}

size\_t queue\_get\_size(const Queue \*pqueue)

{

return pqueue->size;

}

*// linked list init*

void ll\_init(linkList \*plinklist)

{

plinklist->head = NULL;

plinklist->tail = NULL;

plinklist->size = 0;

}

void ll\_destroy(linkList \*plinklist)

{

Node \*current\_node = NULL;

Node \*p = NULL;

if (plinklist->head == NULL) {

return;

}

current\_node = plinklist->head;

while (current\_node != NULL) {

p = current\_node;

current\_node = current\_node->next;

free(p);

p = NULL;

}

}

void ll\_push\_back(linkList \*plinklist, DataType data)

{

Node \*temp = new\_node(data);

if (plinklist->head == NULL) {

*// push*

++plinklist->size;

plinklist->head = temp;

plinklist->tail = temp;

}

else {

++plinklist->size;

plinklist->tail->next = temp;

plinklist->tail = plinklist->tail->next;

}

}

bool ll\_is\_empty(const linkList \*plinkList)

{

return plinkList->head == NULL;

}

size\_t ll\_get\_size(const linkList \*plinklist)

{

return plinklist->size;

}

void ll\_remove(linkList \*plinklist, Node \*pnode)

{

Node \*cur = NULL;

assert(plinklist != NULL);

cur = plinklist->head;

if ((plinklist->head) == NULL) {

return;

}

if ((plinklist->head) == pnode) {

plinklist->head = pnode->next;

free(pnode);

pnode = NULL;

--(plinklist->size);

return;

}

while(cur != NULL) {

if (cur->next == pnode) {

cur->next = pnode->next;

free(pnode);

pnode = NULL;

--(plinklist->size);

break;

}

cur = cur->next;

}

}

SPF算法源代码

#include<stdio.h>

int main()

{

int burst\_t[10],process[10],n,temp,i,j,w\_time[10],sum=0;

float average;

printf("Enter number of process:");

scanf("%d",&n);

printf("\n Enter burst time for each process:");

for(i=0;i<n;i++)

{

printf("\nBurst time of process P%d:",i);

scanf("%d",&burst\_t[i]);

process[i]=i;

}

for(i=0;i<n-1;i++)

{

for(j=i+1;j<n;j++)

{

if(burst\_t[i]>burst\_t[j])

{

temp=burst\_t[i];

burst\_t[i]=burst\_t[j];

burst\_t[j]=temp;

temp=process[i];

process[i]=process[j];

process[j]=temp;

}

}

}

w\_time[0]=0;

for(i=1;i<n;i++)

{

w\_time[i]=w\_time[i-1]+burst\_t[i-1];

}

for(i=0;i<n;i++)

{

sum+=w\_time[i];

}

average=(float)sum/n;

printf("\n Waiting time for each process:-");

for(i=0;i<n;i++)

{

printf("\n Waiting time for process P%d is %d sec.",process[i],w\_time[i]);

}

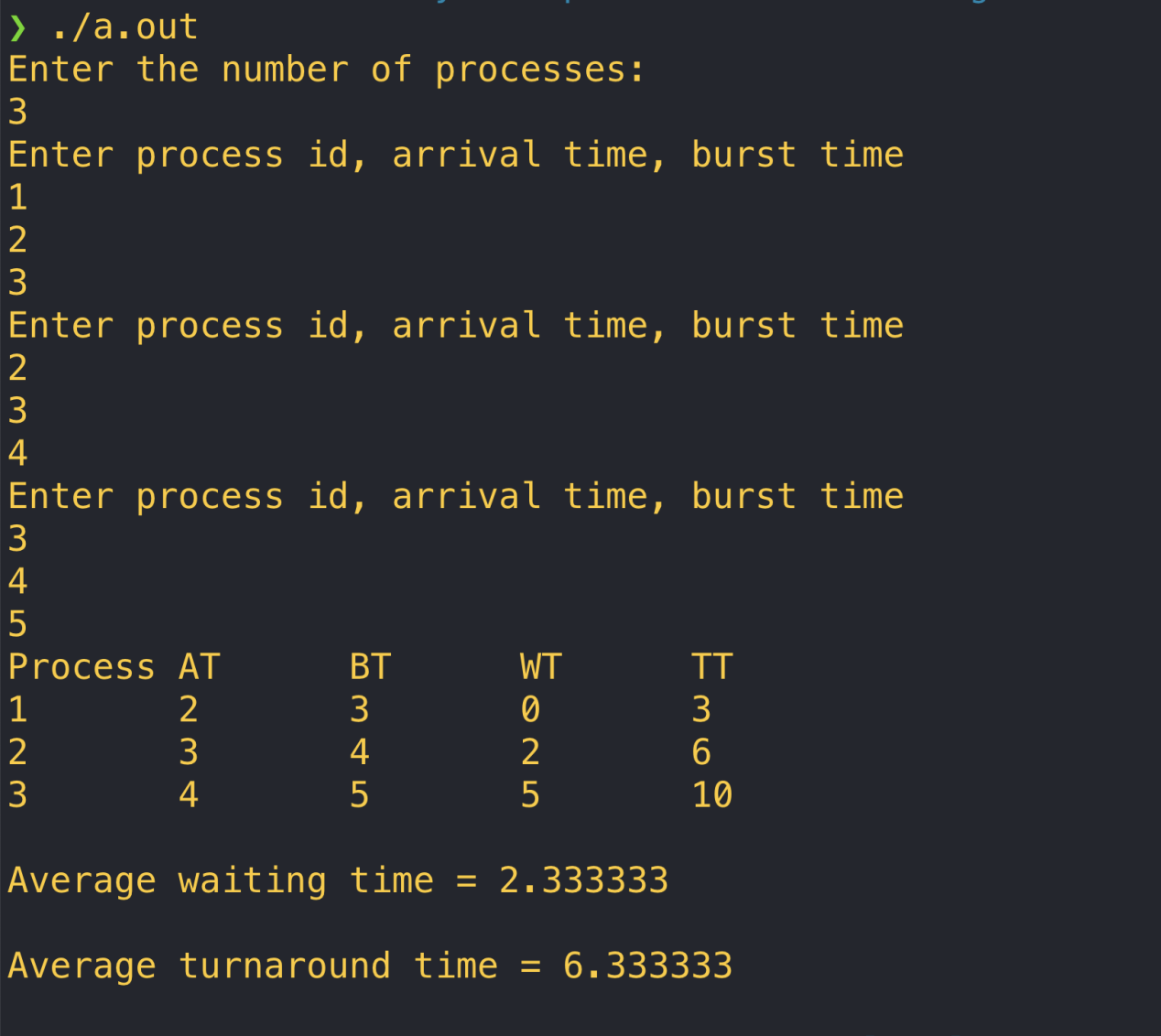
printf("\n Average waiting time is %f sec.",average);

return 0;

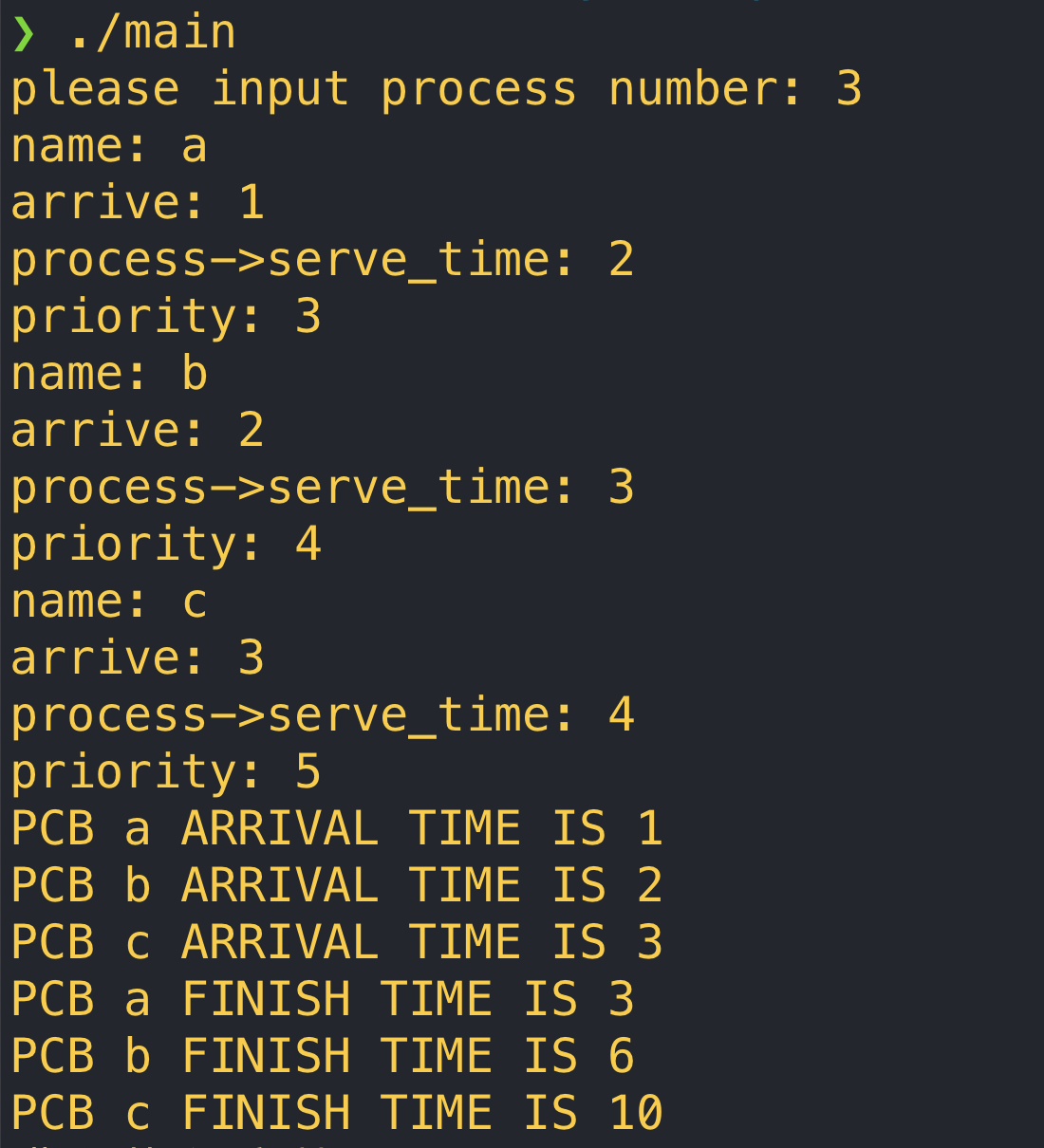
}

1. 实验结果

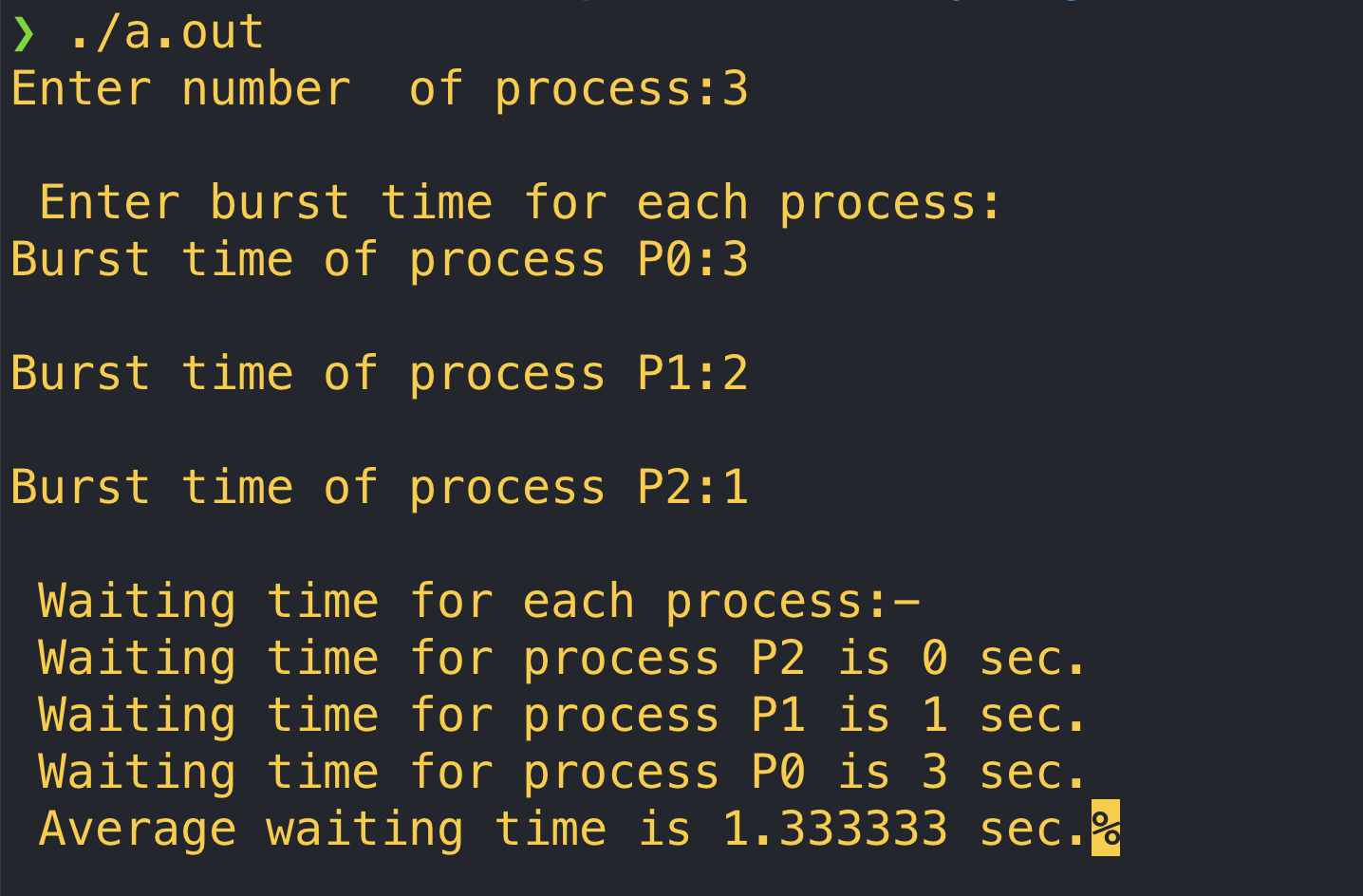
FSFC实验结果



PSA执行结果



SPF运行结果



# 实验3

FIFO算法

源代码:

#include <stdio.h>

int main()

{

int referenceString[10];

int pageFaults = 0, m, n, s, pages, frames;

printf("\nEnter the number of Pages:\t");

scanf("%d", &pages);

printf("\nEnter reference string values:\n");

for(m = 0; m < pages; m++) {

printf("Value No. [%d]:\t", m + 1);

scanf("%d", &referenceString[m]);

}

printf("\n What are the total number of frames:\t");

{

scanf("%d", &frames);

}

int temp[frames];

for(m = 0; m < frames; m++)

{

temp[m] = -1;

}

for(m = 0; m < pages; m++)

{

s = 0;

for(n = 0; n < frames; n++)

{

if(referenceString[m] == temp[n])

{

s++;

pageFaults--;

}

}

pageFaults++;

if((pageFaults <= frames) && (s == 0))

{

temp[m] = referenceString[m];

}

else if(s == 0)

{

temp[(pageFaults - 1) % frames] = referenceString[m];

}

printf("\n");

for(n = 0; n < frames; n++)

{

printf("%d\t", temp[n]);

}

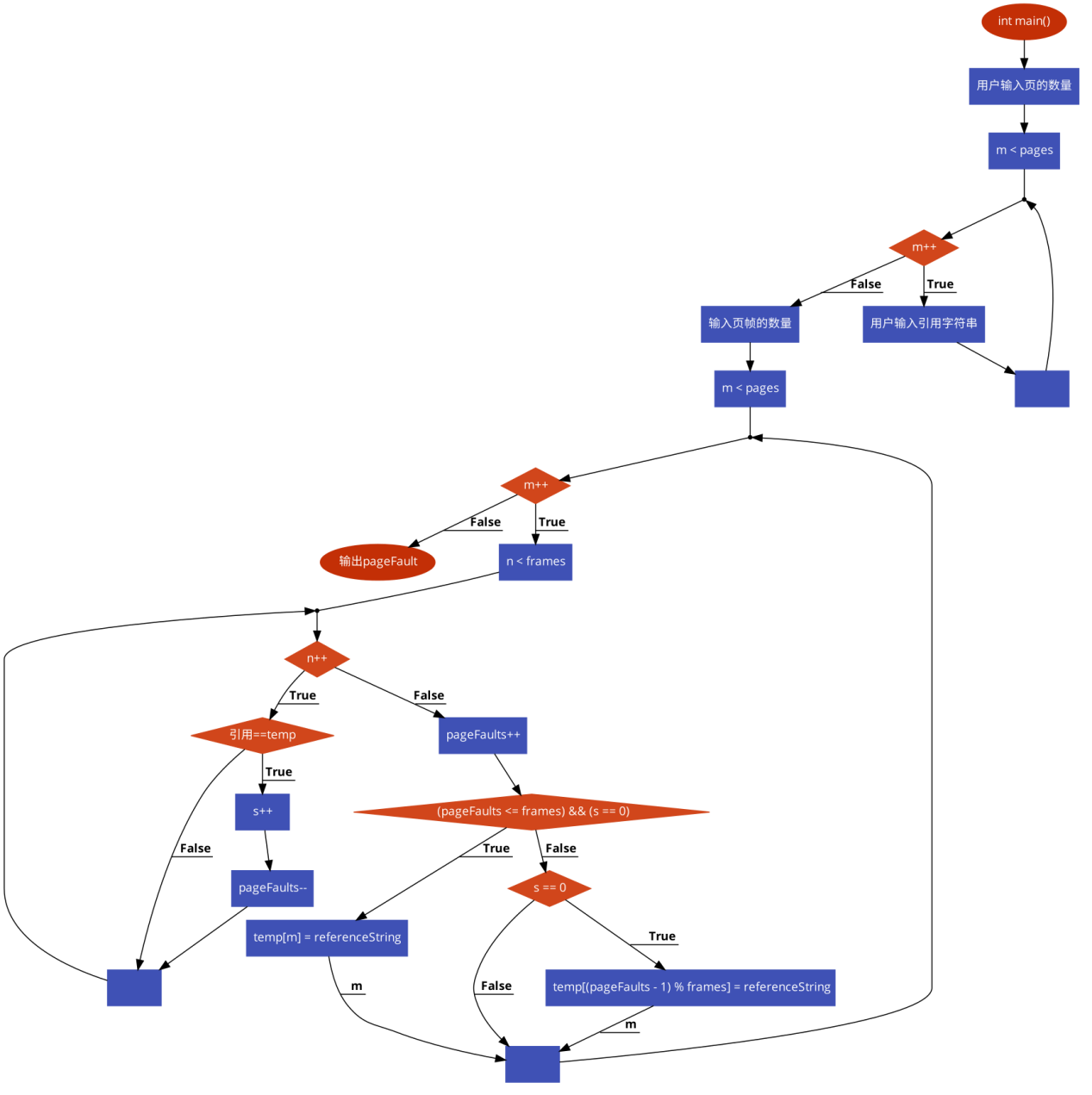
}

printf("\nTotal Page Faults:\t%d\n", pageFaults);

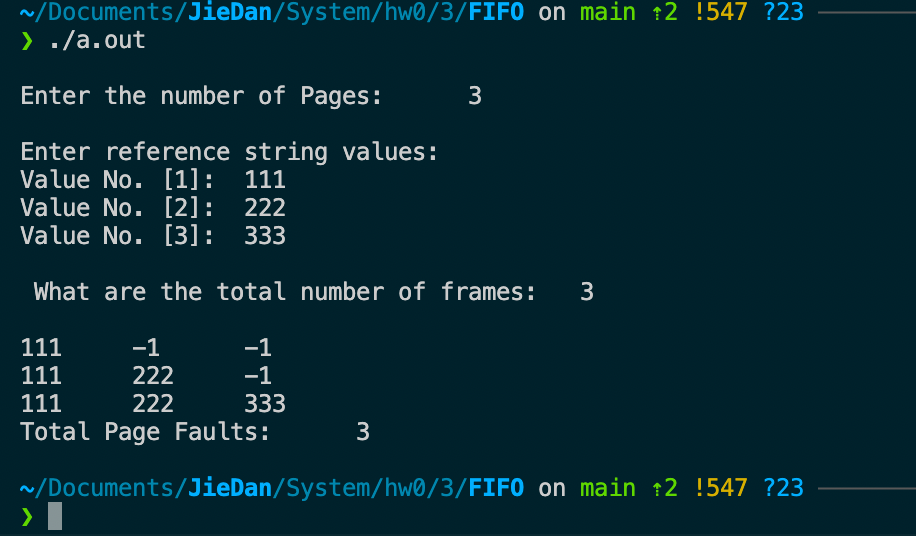
return 0;

}

流程图：



运行结果如下:



LRU缓存算法:

由于使用c语言构造队列，链表，哈希表等数据结构相对困难，为了解约时间和方便编写，这里使用java来编写。

import java.util.Deque;

import java.util.HashSet;

import java.util.LinkedList;

import java.util.Iterator;

public class LRUCache {

private Deque<Integer> doublyQueue;

private HashSet<Integer> hashSet;

private final int CACHE\_SIZE;

LRUCache(int capacity) {

doublyQueue = new LinkedList<>();

hashSet = new HashSet<>();

CACHE\_SIZE = capacity;

}

*/\* Refer the page within the LRU cache \*/*

public void refer(int page) {

if (!hashSet.contains(page)) {

if (doublyQueue.size() == CACHE\_SIZE) {

int last = doublyQueue.removeLast();

hashSet.remove(last);

}

}

else {

doublyQueue.remove(page);

}

doublyQueue.push(page);

hashSet.add(page);

}

*// display contents of cache*

public void display() {

Iterator<Integer> itr = doublyQueue.iterator();

while (itr.hasNext()) {

System.out.print(itr.next() + " ");

}

}

public static void main(String[] args) {

LRUCache cache = new LRUCache(4);

cache.refer(1);

cache.refer(2);

cache.refer(3);

cache.refer(1);

cache.refer(4);

cache.refer(5);

cache.refer(2);

cache.refer(2);

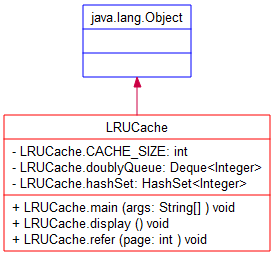
cache.refer(1);

cache.display();

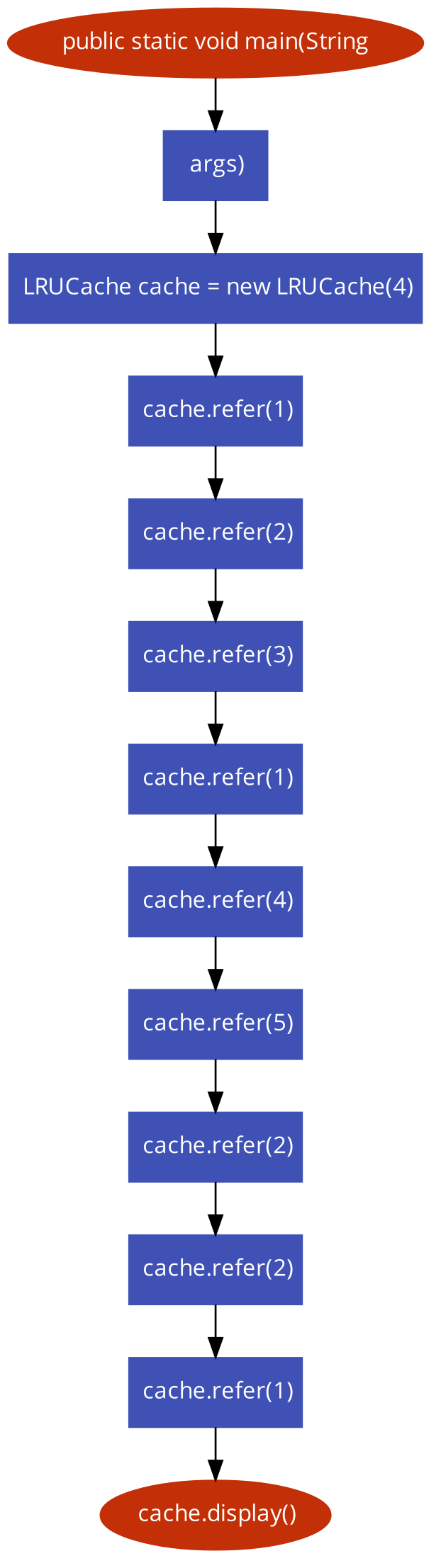
}

}

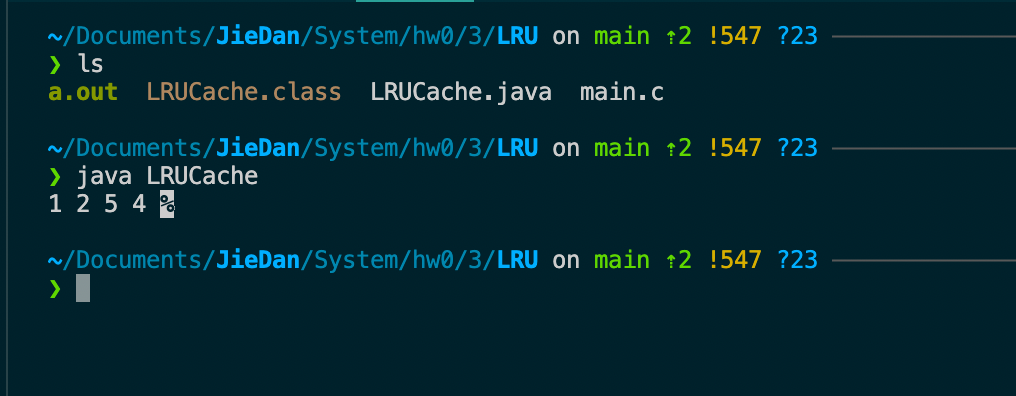
LRUCache类图:



main函数流程图:



运行结果如下:



OPT算法代码如下：

#include <bits/stdc++.h>

using namespace std;

bool search(int key, vector<int>& fr)

{

for (int i = 0; i < fr.size(); i++)

if (fr[i] == key)

return true;

return false;

}

int predict(int pg[], vector<int>& fr, int pn, int index)

{

int res = -1, farthest = index;

for (int i = 0; i < fr.size(); i++) {

int j;

for (j = index; j < pn; j++) {

if (fr[i] == pg[j]) {

if (j > farthest) {

farthest = j;

res = i;

}

break;

}

}

if (j == pn)

return i;

}

return (res == -1) ? 0 : res;

}

void optimalPage(int pg[], int pn, int fn)

{

vector<int> fr;

int hit = 0;

for (int i = 0; i < pn; i++) {

if (search(pg[i], fr)) {

hit++;

continue;

}

if (fr.size() < fn)

fr.push\_back(pg[i]);

else {

int j = predict(pg, fr, pn, i + 1);

fr[j] = pg[i];

}

}

cout << "No. of hits = " << hit << endl;

cout << "No. of misses = " << pn - hit << endl;

}

int main()

{

int pg[] = { 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2 };

int pn = sizeof(pg) / sizeof(pg[0]);

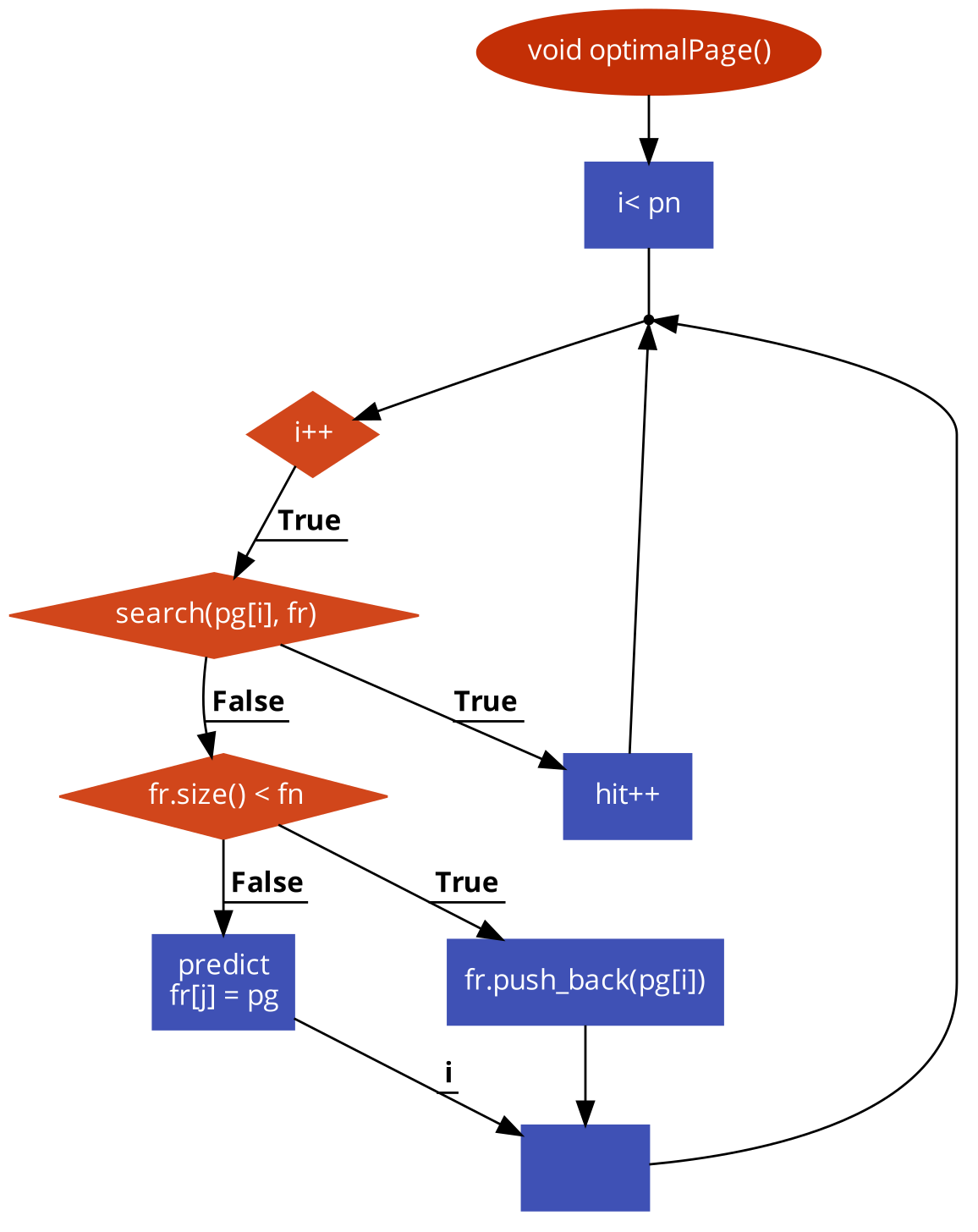
int fn = 4;

optimalPage(pg, pn, fn);

return 0;

}

算法流程图:



运行结果如下:

