Programming Report

Description:

Exercise 4.22: Implementation of FIFO LRU, and Optimal algorithms. P1-6
Project 1: Implementation of FCFS, SJF, Priority, RR, Priority_RR algorithms. P7-17
Project 2: Banker's algorithm. P17-25.
And I will show you the codes, results, and explanation of each problem.

4.22: Page-Replacement Algorithms.

Functions:

```
//Generates a random page-reference string.
int* random_string();
//Initialization of frames before FIFO, LRU, and Optimal.
void initial_frames(int* frames, int frames_number);
//Checks if the current data exists in the frames for FIFO, LRU, and Optimal.
bool exists(int data, int* frames, int frames_number);
//In order to show the current status of frames.
void print_array(int* array, int length);
//Implements First In First Out (FIFO) page-replacement algorithm
int fifo(int* reference_string, int* frames, int frames_number);
//Implements Least Recently Used (LRU) page-replacement algorithm
int lru(int* reference_string, int* frames, int frames_number);
//Implements Optimal page-replacement algorithm
int optimal(int* reference_string, int* frames, int frames_number);
```

1) I used the function "random_string()" to generate a random page-reference string.
2) "initial_frames()" is necessary to avoid the influence of different algorithms because the "frames[]" is a global variable.
3) "fifo" for First In First Out, "lru" for Least Recently Used, "optimal" for Optimal.

Main Function:

```
int main(int argc, char *argv[]) {
	if (argc < 2) {
		printf("Error: You must provide the number of page frames as an argument.\n");
		exit(EXIT_FAILURE);
	}
	int frames_number = atoi(argv[1]);
	int* frames = (int*)malloc(frames_number * sizeof(int));

	int* reference_string = random_string();
	printf("Random Reference String:\n");
	print_array(reference_string, STRING_NUMBER);
	printf("\n");

	printf("Start Alogorithm of FIFO:\nStatus of frames:\n");
	printf("Page faults of FIFO: %d\n\n", fifo(reference_string, frames, frames_number));

	printf("Start Alogorithm of LRU:\nStatus of frames:\n");
	printf("Page faults of LRU: %d\n\n", lru(reference_string, frames, frames_number));

	printf("Start Alogorithm of Optimal:\nStatus of frames:\n");
	printf("Page faults of Optimal: %d\n\n", optimal(reference_string, frames, frames_number));

	free(frames);
	free(reference_string);
	return 0;
}
```

1) I first generate a random string for page-reference and print it out.
2) Then I also print out the results and the process of iterations of each algorithms.

First In First Out (FIFO):

```
//Implements First In First Out (FIFO) page-replacement algorithm
int fifo(int* reference_string, int* frames, int frames_number) {
        page_faults = 0;
        initial_frames(frames, frames_number);

        int i, j = 0;
        for (i = 0; i < STRING_NUMBER; i++) {
                if (!exists(reference_string[i], frames, frames_number)) {
                        for (j = 0; j < frames_number - 1; j++) {
                                frames[j] = frames[j + 1];
                        }
                        frames[j] = reference_string[i];
                        page_faults++;
                        print_array(frames, frames_number);
                }
        }
        return page_faults;
}
```

I used “print_array()” to print the process of each iteration when it appears page fault.

Least Recently Used (LRU):

```
//Implements Least Recently Used (LRU) page-replacement algorithm
int lru(int* reference_string, int* frames, int frames_number) {
        page_faults = 0;
        initial_frames(frames, frames_number);
        int* close = (int*)malloc(frames_number * sizeof(int));

        int i, j, k = 0;
        while (k < STRING_NUMBER) {
                if (!exists(reference_string[k], frames, frames_number)) {
                        for (i = 0; i < frames_number; i++) {
                                int find = 0;
                                int frame = frames[i];
                                j = k - 1;
                                while (j >= 0) {
                                        if (frame == reference_string[j]) {
                                                find = 1;
                                                close[i] = j;
                                                break;
                                        }
                                        else {
                                                find = 0;
                                        }
                                        j--;
                                }
                                if (!find) {
                                        close[i] = -99;
                                }
                        }
```

```
                    }
                    int least = 99;
                    int repeated;
                    i = 0;
                    while (i < frames_number) {
                            if (close[i] < least) {
                                    repeated = i;
                                    least = close[i];
                            }
                            i++;
                    }
                    frames[repeated] = reference_string[k];
                    page_faults++;
                    print_array(frames, frames_number);
            }
        k++;
        }
        free(close);
        return page_faults;
}
```

I used “print_array()” to print the process of each iteration when it appears page fault.

Optimal:

```
//Implements Optimal page-replacement algorithm
int optimal(int* reference_string, int* frames, int frames_number) {
        page_faults = 0;
        initial_frames(frames, frames_number);
        int* close = (int*)malloc(frames_number * sizeof(int));

        int i, j, k = 0;
        while (k < STRING_NUMBER) {
                if (!exists(reference_string[k], frames, frames_number)) {
                        for (i = 0; i < frames_number; i++) {
                                int find = 0;
                                int frame = frames[i];
                                j = k;
                                while (j < STRING_NUMBER) {
                                        if (frame == reference_string[j]) {
                                                find = 1;
                                                close[i] = j;
                                                break;
                                        }
                                        else {
                                                find = 0;
                                        }
                                        j++;
                                }
                                if (!find) {
                                        close[i] = 99;
                                }
                        }
                        int maximum = -99;
                        int repeated;
                        i = 0;
```

```
                        while (i < frames_number) {
                                if (maximum < close[i]) {
                                        repeated = i;
                                        maximum = close[i];
                                }
                                i++;
                        }
                        frames[repeated] = reference_string[k];
                        page_faults++;
                        print_array(frames, frames_number);
                }
                k++;
        }
        free(close);
        return page_faults;
}
```

I used "print_array()" to print the process of each iteration when it appears page fault.

Compile (run.sh):

```
echo "gcc -o page-replacement page-replacement.c"
echo "./page-replacement 3"
echo ""
echo ""
gcc -o page-replacement page-replacement.c
./page-replacement 3
```

Results:

```
xiaoqi@xiaoqi:~/Desktop/OS/HW5/10.44$ ./run.sh
gcc -o page-replacement page-replacement.c
./page-replacement 3


Random Reference String:
0 5 5 2 1 7 3 5 2 1 8 5 9 7 2 7 6 4 7 5

Start Alogorithm of FIFO:
Status of frames:
    0
  0 5
0 5 2
5 2 1
2 1 7
1 7 3
7 3 5
3 5 2
5 2 1
2 1 8
1 8 5
8 5 9
5 9 7
9 7 2
7 2 6
2 6 4
6 4 7
4 7 5
Page faults of FIFO: 18
```

```
Start Alogorithm of LRU:
Status of frames:
0
0 5
0 5 2
1 5 2
1 7 2
1 7 3
5 7 3
5 2 3
5 2 1
8 2 1
8 5 1
8 5 9
7 5 9
7 2 9
7 2 6
7 4 6
7 4 5
Page faults of LRU: 17

Start Alogorithm of Optimal:
Status of frames:
0
5
5 2
5 2 1
5 2 7
5 2 3
5 2 1
5 2 8
5 2 9
5 2 7
5 6 7
5 4 7
Page faults of Optimal: 12
```

You can see my process of each iteration when happens page fault of each algorithm.

Other Codes:

```
//Generates a random page-reference string.
int* random_string() {
	int* string = (int*)malloc(STRING_NUMBER * sizeof(int));
	srand(time(NULL));
	for (int i = 0; i < STRING_NUMBER; i++) {
		string[i] = rand() % 10;
	}
	return string;
}

//Initialization of frames before FIFO, LRU, and Optimal.
void initial_frames(int* frames, int frames_number) {
	for (int i = 0; i < frames_number; i++) {
		frames[i] = -1;
	}
}

//Checks if the current data exists in the frames for FIFO, LRU, and Optimal.
bool exists(int data, int* frames, int frames_number) {
	int k = 0;
	while (k < frames_number) {
		if (frames[k] == data) {
			return true;
		}
		k++;
	}
	return false;
}
```

```
//In order to show the current status of frames.
void print_array(int* array, int length) {
	for (int i = 0; i < length; i++) {
		if(array[i] == -1) {
			printf("  ");
		}
		else {
			printf("%d ", array[i]);
		}
	}
	printf("\n");
}
```

Project 1: Scheduling Algorithms.

Notice:

1) This project is mainly based on the codes from the source code of the textbook, CPU.c, cpu.h, driver.c, list.c, list,h, task.h, schedulers.h, Makefile, schedules.txt.
2) My task is to implement schedule_fcfs.c, schedule_sjf.c, schedule_priority.c, schedule_rr.c, schedule_priority_rr.c while implementing the function “add()”, and “schedule()” in the file “schedulers.h”.
3) Once we finished implementation of those algorithms, we need to use its Makefile to compile, such as: “make fcfs”...
4) Finally, in order to run a program, we also need a file including the information of processes as an argument, such as: “./fcfs schedules.txt”.

First Come First Served (FCFS):

```
//Structure from "list.h".
struct node *taskList = NULL;

//Implementation of the function "add()" in schedulers.h.
//Add the current task to the taskList.
void add(char *name, int priority, int burst) {
        //Struture from "task.h".
        Task* task = (Task*)malloc(sizeof(Task));

        //Allocate the memory of the name (char*) first.
        task->name = (char*)malloc(sizeof(char) * (strlen(name) + 1));
        strcpy(task->name, name);
        task->priority = priority;
        task->burst = burst;

        //Function from "list.h".
        insert(&taskList, task);
}

//Pick the next task to execute with FCFS.
Task* pickNextTask() {
        struct node *lastNode = taskList;
        while(1) {
                if(!lastNode->next) {
                        break;
                }
                lastNode = lastNode->next;
        }
        return lastNode->task;
}
```

```
//Implementation of the function "schedule()" in schedulers.h.
void schedule() {
	//Used for calculating the average of:
	//Turnaround Time = Exit Time - Arrival Time,
	//Waiting Time = Turnaround Time - Burst Time,
	//Response Time = Time when the process gets CPU - Arrival Time.
	int timeCounter = 0;
	int processCounter = 0;
	int turnaroundCounter = 0;
	int waitingCounter = 0;
	int responseCounter = 0;

	while(taskList) {
		Task *task = pickNextTask();
		run(task, task->burst);

		//Calculating:
		timeCounter += task->burst;
		processCounter++;
		task->turnaround = timeCounter;
		task->waiting = task->turnaround - task->burst;
		task->response = timeCounter;

		turnaroundCounter += task->turnaround;
		waitingCounter += task->waiting;
		responseCounter += task->response;
		delete(&taskList, task);
	}
	printf("Average Turnaround Time = %d\n", turnaroundCounter / processCounter);
	printf("Average Waiting Time = %d\n", waitingCounter / processCounter);
	printf("Average Response Time = %d\n", responseCounter / processCounter);
}
```

1) I implement the function “add()”, and “schedule()” from “schedulers.h”.
2) I also added a “pickNextTask()” implements the main idea of algorithms.
3) I also added a lot of calculation of the Turnaround Time, Waiting Time, Response Time in the function “schedule()”. And this will be print out in the results.
4) Because the screenshot of the algorithms are quite big, the typesetting is a bit ugly, I’m sorry for that.

Shortest Job First:

```c
//This program SJF onlys changes some part of the function
//"pickNextTask()" from program FCFS.

//Structure from "list.h".
struct node *taskList = NULL;

//Implementation of the function "add()" in schedulers.h.
//Add the current task to the taskList.
void add(char *name, int priority, int burst) {
        //Struture from "task.h".
        Task* task = (Task*)malloc(sizeof(Task));
        //Allocate the memory of the name (char*) first.
        task->name = (char*)malloc(sizeof(char) * (strlen(name) + 1));
        strcpy(task->name, name);
        task->priority = priority;
        task->burst = burst;

        //Function from "list.h".
        insert(&taskList, task);
}

//Pick the next task (shortest task in the queue) to execute with SJF.
Task* pickNextTask() {
        Task* shortest_task = taskList->task;
        struct node* Node = taskList;

        //Find the task has the shortest burst time.
        while(Node) {
                if(Node->task->burst <= shortest_task->burst) {
                        shortest_task = Node->task;
                }
                Node = Node->next;
        }
        return shortest_task;
}
```

```c
//Implementation of the function "schedule()" in schedulers.h.
void schedule() {
        //Used for calculating the average of:
        //Turnaround Time = Exit Time - Arrival Time,
        //Waiting Time = Turnaround Time - Burst Time,
        //Response Time = Time when the process gets CPU - Arrival Time.
        int timeCounter = 0;
        int processCounter = 0;
        int turnaroundCounter = 0;
        int waitingCounter = 0;
        int responseCounter = 0;

        while(taskList) {
                Task *task = pickNextTask();
                run(task, task->burst);

                //Calculating:
                timeCounter += task->burst;
                processCounter++;
                task->turnaround = timeCounter;
                task->waiting = task->turnaround - task->burst;
                task->response = timeCounter;

                turnaroundCounter += task->turnaround;
                waitingCounter += task->waiting;
                responseCounter += task->response;
                delete(&taskList, task);
        }
        printf("Average Turnaround Time = %d\n", turnaroundCounter / processCounter);
        printf("Average Waiting Time = %d\n", waitingCounter / processCounter);
        printf("Average Response Time = %d\n", responseCounter / processCounter);
}
```

Priority:

```
//This program Priority onlys changes some part of the function
//"pickNextTask()" from program SJF.

//Structure from "list.h".
struct node *taskList = NULL;

//Implementation of the function "add()" in schedulers.h.
//Add the current task to the taskList.
void add(char *name, int priority, int burst) {
	//Struture from "task.h".
	Task* task = (Task*)malloc(sizeof(Task));
	//Allocate the memory of the name (char*) first.
	task->name = (char*)malloc(sizeof(char) * (strlen(name) + 1));
	strcpy(task->name, name);
	task->priority = priority;
	task->burst = burst;

	//Function from "list.h".
	insert(&taskList, task);
}

//Pick the next task (highest priority in the queue) to execute.
Task* pickNextTask() {
	Task* highest_priority_task = taskList->task;
	struct node* Node = taskList;

	//Find the task has the highest priority.
	while(Node) {
		if(Node->task->priority >= highest_priority_task->priority) {
			highest_priority_task = Node->task;
		}
		Node = Node->next;
	}
	return highest_priority_task;
}
```

```
//Implementation of the function "schedule()" in schedulers.h.
void schedule() {
	//Used for calculating the average of:
	//Turnaround Time = Exit Time - Arrival Time,
	//Waiting Time = Turnaround Time - Burst Time,
	//Response Time = Time when the process gets CPU - Arrival Time.
	int timeCounter = 0;
	int processCounter = 0;
	int turnaroundCounter = 0;
	int waitingCounter = 0;
	int responseCounter = 0;

	while(taskList) {
		Task *task = pickNextTask();
		run(task, task->burst);

		//Calculating:
		timeCounter += task->burst;
		processCounter++;
		task->turnaround = timeCounter;
		task->waiting = task->turnaround - task->burst;
		task->response = timeCounter;

		turnaroundCounter += task->turnaround;
		waitingCounter += task->waiting;
		responseCounter += task->response;

		delete(&taskList, task);
	}
	printf("Average Turnaround Time = %d\n", turnaroundCounter / processCounter);
	printf("Average Waiting Time = %d\n", waitingCounter / processCounter);
	printf("Average Response Time = %d\n", responseCounter / processCounter);
```

Round Robin:

```
//In order to perform the RR algorithm, we have to consider its
//remaining burst everytime, so I add an element "int remaining_burst"
//in the struct of Task from task.h.

//Structure from "list.h".
struct node *taskList = NULL;
struct node *next_node;

//Implementation of the function "add()" in schedulers.h.
//Add the current task to the taskList.
void add(char *name, int priority, int burst) {
        //Struture from "task.h".
        Task* task = (Task*)malloc(sizeof(Task));
        //Allocate the memory of the name (char*) first.
        task->name = (char*)malloc(sizeof(char) * (strlen(name) + 1));
        strcpy(task->name, name);
        task->priority = priority;
        task->burst = task->remaining_burst = burst;
        //In order to get the response time of each process.
        task->response = 0;
        //Function from "list.h".
        insert(&taskList, task);
}

//Pick the next task to execute with RR.
Task* pickNextTask() {
        Task* temp = next_node->task;
        if (next_node->next) {
                next_node = next_node->next;
        }
        else {
                next_node = taskList;
        }
        return temp;
}
```

```
//Implementation of the function "schedule()" in schedulers.h.
void schedule() {
        //Used for calculating the average of:
        //Turnaround Time = Exit Time - Arrival Time,
        //Waiting Time = Turnaround Time - Burst Time,
        //Response Time = Time when the process gets CPU - Arrival Time.
        int timeCounter = 0;
        int processCounter = 0;
        int turnaroundCounter = 0;
        int waitingCounter = 0;
        int responseCounter = 0;

        next_node = taskList;
        while(taskList) {
                Task *task = pickNextTask();

                //Calculating Response Time.
                if (task->response == 0) {
                        task->response = timeCounter;
                        responseCounter += task->response;
                }

                //Quantum is 10 milliseconds.
                int slice;
                if (QUANTUM < task->remaining_burst) {
                        slice = QUANTUM;
                }
                else {
                        slice = task->remaining_burst;
                }

                run(task, slice);
                task->remaining_burst -= slice;
                timeCounter += slice;
```

```
                //Once a process is finished, we will record
                //its turnaround and waiting time.
                if (!task->remaining_burst) {
                        processCounter++;
                        task->turnaround = timeCounter;
                        task->waiting = task->turnaround - task->burst;

                        turnaroundCounter += task->turnaround;
                        waitingCounter += task->waiting;
                        delete(&taskList, task);
                }
        }
        printf("Average Turnaround Time = %d\n", turnaroundCounter / processCounter);
        printf("Average Waiting Time = %d\n", waitingCounter / processCounter);
        printf("Average Response Time = %d\n", responseCounter / processCounter);
}
```

Priority with Round Robin:

```
//Structure from "list.h".
//MAX_PRIORITY = 10.
struct node *taskList[MAX_PRIORITY + 1];
struct node *next_node;

//Implementation of the function "add()" in schedulers.h.
//Add the current task to the taskList.
void add(char *name, int priority, int burst) {
        //Struture from "task.h".
        Task* task = (Task*)malloc(sizeof(Task));
        //Allocate the memory of the name (char*) first.
        task->name = (char*)malloc(sizeof(char) * (strlen(name) + 1));
        strcpy(task->name, name);
        task->priority = priority;
        task->burst = task->remaining_burst = burst;

        //In order to get the response time of each process.
        task->response = 0;
        //Function from "list.h".
        insert(&taskList[priority], task);
}

//Pick the next task to execute with RR.
//We can not use the global varaible taskList now,
//because taskList is a pointer that has MAX_PRIORITY+1 elements.
Task* pickNextTask(struct node* tl) {
        Task* temp = next_node->task;
        if (next_node->next) {
                next_node = next_node->next;
        }
        else {
                next_node = tl;
        }
        return temp;
}
```

```c
//Implementation of the function "schedule()" in schedulers.h.
void schedule() {
	//Used for calculating the average of:
	//Turnaround Time = Exit Time - Arrival Time,
	//Waiting Time = Turnaround Time - Burst Time,
	//Response Time = Time when the process gets CPU - Arrival Time.
	int timeCounter = 0;
	int processCounter = 0;
	int turnaroundCounter = 0;
	int waitingCounter = 0;
	int responseCounter = 0;

	//From higher priority to lower priority.
	for (size_t i = MAX_PRIORITY; i >= MIN_PRIORITY; i--) {
		next_node = taskList[i];
		while(taskList[i]) {
			Task *task = pickNextTask(taskList[i]);

			//Calculating Response Time.
			if (task->response == 0) {
				task->response = timeCounter;
				responseCounter += task->response;
			}

			//Quantum is 10 milliseconds.
			int slice;
			if (QUANTUM < task->remaining_burst) {
				slice = QUANTUM;
			}
			else {
				slice = task->remaining_burst;
			}
```

```c
			run(task, slice);
			task->remaining_burst -= slice;
			timeCounter += slice;

			//Once a process is finished, we will record
			//its turnaround and waiting time.
			if (!task->remaining_burst) {
				processCounter++;
				task->turnaround = timeCounter;
				task->waiting = task->turnaround - task->burst;

				turnaroundCounter += task->turnaround;
				waitingCounter += task->waiting;
				delete(&taskList[i], task);
			}
		}
	}
	printf("Average Turnaround Time = %d\n", turnaroundCounter / processCounter);
	printf("Average Waiting Time = %d\n", waitingCounter / processCounter);
	printf("Average Response Time = %d\n", responseCounter / processCounter);
}
```

Compile (run.sh):

```
make fcfs
make sjf
make priority
make rr
make priority_rr
echo ""
echo "--------------------fcfs----------------------"
./fcfs schedule.txt
echo ""
echo "--------------------sjf-----------------------"
./sjf schedule.txt
echo ""
echo "------------------priority--------------------"
./priority schedule.txt
echo ""
echo "--------------------rr------------------------"
./rr schedule.txt
echo ""
echo "-----------------priority_rr------------------"
./priority_rr schedule.txt
```

My results are mainly based on the information of processes in schedule.txt.

Results:

```
xiaoqi@xiaoqi:~/Desktop/OS/HW5/Scheduling$ ./run.sh
gcc -Wall -c driver.c
gcc -Wall -c list.c
gcc -Wall -c CPU.c
gcc -Wall -c schedule_fcfs.c
gcc -Wall -o fcfs driver.o schedule_fcfs.o list.o CPU.o
gcc -Wall -c schedule_sjf.c
gcc -Wall -o sjf driver.o schedule_sjf.o list.o CPU.o
gcc -Wall -c schedule_priority.c
gcc -Wall -o priority driver.o schedule_priority.o list.o CPU.o
gcc -Wall -c schedule_rr.c
gcc -Wall -o rr driver.o schedule_rr.o list.o CPU.o
gcc -Wall   -c -o schedule_priority_rr.o schedule_priority_rr.c
gcc -Wall -o priority_rr driver.o schedule_priority_rr.o list.o CPU.o

--------------------fcfs----------------------
Running task = [T1] [4] [20] for 20 units.
Running task = [T2] [3] [25] for 25 units.
Running task = [T3] [3] [25] for 25 units.
Running task = [T4] [5] [15] for 15 units.
Running task = [T5] [5] [20] for 20 units.
Running task = [T6] [1] [10] for 10 units.
Running task = [T7] [3] [30] for 30 units.
Running task = [T8] [10] [25] for 25 units.
Average Turnaround Time = 94
Average Waiting Time = 73
Average Response Time = 94
```

```
---------------------sjf------------------------
Running task = [T6] [1] [10] for 10 units.
Running task = [T4] [5] [15] for 15 units.
Running task = [T1] [4] [20] for 20 units.
Running task = [T5] [5] [20] for 20 units.
Running task = [T2] [3] [25] for 25 units.
Running task = [T3] [3] [25] for 25 units.
Running task = [T8] [10] [25] for 25 units.
Running task = [T7] [3] [30] for 30 units.
Average Turnaround Time = 82
Average Waiting Time = 61
Average Response Time = 82

-------------------priority---------------------
Running task = [T8] [10] [25] for 25 units.
Running task = [T4] [5] [15] for 15 units.
Running task = [T5] [5] [20] for 20 units.
Running task = [T1] [4] [20] for 20 units.
Running task = [T2] [3] [25] for 25 units.
Running task = [T3] [3] [25] for 25 units.
Running task = [T7] [3] [30] for 30 units.
Running task = [T6] [1] [10] for 10 units.
Average Turnaround Time = 96
Average Waiting Time = 75
Average Response Time = 96
```

```
----------------------rr-------------------------
Running task = [T8] [10] [25] for 10 units.
Running task = [T7] [3] [30] for 10 units.
Running task = [T6] [1] [10] for 10 units.
Running task = [T5] [5] [20] for 10 units.
Running task = [T4] [5] [15] for 10 units.
Running task = [T3] [3] [25] for 10 units.
Running task = [T2] [3] [25] for 10 units.
Running task = [T1] [4] [20] for 10 units.
Running task = [T8] [10] [25] for 10 units.
Running task = [T7] [3] [30] for 10 units.
Running task = [T5] [5] [20] for 10 units.
Running task = [T4] [5] [15] for 5 units.
Running task = [T3] [3] [25] for 10 units.
Running task = [T2] [3] [25] for 10 units.
Running task = [T1] [4] [20] for 10 units.
Running task = [T8] [10] [25] for 5 units.
Running task = [T7] [3] [30] for 10 units.
Running task = [T3] [3] [25] for 5 units.
Running task = [T2] [3] [25] for 5 units.
Average Turnaround Time = 130
Average Waiting Time = 109
Average Response Time = 45
```

```
------------------priority_rr--------------------
Running task = [T8] [10] [25] for 10 units.
Running task = [T8] [10] [25] for 10 units.
Running task = [T8] [10] [25] for 5 units.
Running task = [T5] [5] [20] for 10 units.
Running task = [T4] [5] [15] for 10 units.
Running task = [T5] [5] [20] for 10 units.
Running task = [T4] [5] [15] for 5 units.
Running task = [T1] [4] [20] for 10 units.
Running task = [T1] [4] [20] for 10 units.
Running task = [T7] [3] [30] for 10 units.
Running task = [T3] [3] [25] for 10 units.
Running task = [T2] [3] [25] for 10 units.
Running task = [T7] [3] [30] for 10 units.
Running task = [T3] [3] [25] for 10 units.
Running task = [T2] [3] [25] for 10 units.
Running task = [T7] [3] [30] for 10 units.
Running task = [T3] [3] [25] for 5 units.
Running task = [T2] [3] [25] for 5 units.
Running task = [T6] [1] [10] for 10 units.
Average Turnaround Time = 106
Average Waiting Time = 85
Average Response Time = 70
```

You can see my process of each iterations and final average Turnaround Time, Waiting Time, Response Time of each algorithm.

Project 2: Banker's Algorithm.

Functions:

```c
//Inilization from the textbook.
#define NUMBER_OF_CUSTOMERS 5
#define NUMBER_OF_RESOURCES 4
int available[NUMBER_OF_RESOURCES];
int maximum[NUMBER_OF_CUSTOMERS][NUMBER_OF_RESOURCES];
int allocation[NUMBER_OF_CUSTOMERS][NUMBER_OF_RESOURCES];
int need[NUMBER_OF_CUSTOMERS][NUMBER_OF_RESOURCES];

#define INIT_FILE "resources.txt"

int is_leq(int* a, int* b, int n);
//Check if the whether the operation is safe to execute.
int is_safe();
//Implementation of request resources: RQ.
//Return: 0 for success, non-zero for errors.
int request_resources(int customer, int request[NUMBER_OF_RESOURCES]);
//Implemetation of release resources: RL.
//Return: 0 for success, non-zero for errors.
int release_resources(int customer, int release[NUMBER_OF_RESOURCES]);
//Checker of request resources.
void request_wrapper();
//Checker of release resources.
void release_wrapper();
//Display the compile usage of this program.
void display_initial_usage();
//Display the usage of this program, how to use it.
void display_usage();
//Display the information of current resurces.
void display_resources();
//Initialization of resources from the textbook.
int init(int argc, char *argv[], const char *resources_file);
```

1) The Banker's algorithm is mainly based on 3 parts: Request Resources, Release Resources, and Safe Status Check.
2) My functions "request_resources()" and "request_wrapper()" implements the Request Resources, "release_resources()" and "release_wrapper()" implements the Release Resources.
3) "is_safe()" plays the role of checking a status after doing an operation is safe or not, if the result is false, the program will not do the operation.
4) I also used "display_initial_usage()", "display_usage()", and "display_resources" to give the user important information during the program.

Main Function:

```
int main(int argc, char *argv[]) {
	if(init(argc, argv, INIT_FILE) != 0) {
		display_initial_usage();
		return 0;
	}
	char op[5];
	display_usage();
	printf(">> ");
	while(scanf("%s", op) == 1) {
		if(strcmp(op, "RQ") == 0) {
			request_wrapper();
		}
		else if(strcmp(op, "RL") == 0) {
			release_wrapper();
		}
		else if(strcmp(op, "*") == 0) {
			display_resources();
		}
		else if(strcmp(op, "help") == 0) {
			display_usage();
		}
		else if(strcmp(op, "exit") == 0) {
			break;
		}
		else {
			printf("Invalid Command! You may use 'help' to check the operations y
		}
		printf(">> ");
	}
	return 0;
}
```

1) The main function mainly create a terminal where you can enter command of operations to execute programs.
2) There is a necessary resource file “resources.txt” for reading the initial data of resources, which is just a copy form the textbook. If the resource file is not in the directory, it will report error, and tell you the “resources.txt” is necessary for compiling.
3) There are several operations you can do in the terminal created by the program. Firstly, you have RQ and RL for requesting resources and releasing resources following the usage from the textbook. You also have “*” to display the current information of Available, Maximum, Allocated, Needed resources for each customer.
4) You also have “help” to display the usage and examples of the operations you can do, and “exit” for exiting the terminal and close the program.

Request Resources:

```
//Implementation of request resources: RQ.
//Return: 0 for success, non-zero for errors.
int request_resources(int customer, int request[NUMBER_OF_RESOURCES]) {
        if(customer < 0 || customer >= NUMBER_OF_CUSTOMERS) {
                printf("Invalid customer: %d\n", customer);
                return -1;
        }
        int err = 0;
        for(int i = 0; i != NUMBER_OF_RESOURCES; ++i) {
                if(request[i] < 0 || request[i] > need[customer][i]) {
                        printf("Invalid number of resources to request:\n(customer = %d, reso
                                        customer, i, need[customer][i], request[i]);
                        err = -1;
                }
                if(request[i] > available[i]) {
                        printf("No enough resources to allocate:\n(customer = %d, resource =
                                        customer, i, available[i], request[i]);
                        err = -2;
                }
                if(err != 0) {//rollback
                        while(i--) {
                                available[i] += request[i];
                                allocation[customer][i] -= request[i];
                                need[customer][i] += request[i];
                        }
                        return err;
                }
                // allocate resources
                available[i] -= request[i];
                allocation[customer][i] += request[i];
                need[customer][i] -= request[i];
        }
```

```
        if(!is_safe()) {
                // rollback
                printf("Unsafe state after request!\n");
                for(int i = 0; i != NUMBER_OF_RESOURCES; ++i) {
                        available[i] += request[i];
                        allocation[customer][i] -= request[i];
                        need[customer][i] += request[i];
                }
                return -3;
        }
        return 0;
}
```

```
//Checker of request resources.
void request_wrapper() {
        int request[NUMBER_OF_RESOURCES], customer;
        scanf("%d", &customer);
        for(int i = 0; i != NUMBER_OF_RESOURCES; ++i) {
                scanf("%d", &request[i]);
        }
        if(request_resources(customer, request) != 0) {
                printf("FAILED.\n");
        }
        else {
                printf("SUCCEEDED.\n");
        }
}
```

Release Resources:

```
//Implemetation of release resources: RL.
//Return: 0 for success, non-zero for errors.
int release_resources(int customer, int release[NUMBER_OF_RESOURCES]) {
        if(customer < 0 || customer >= NUMBER_OF_CUSTOMERS) {
                printf("Invalid customer: %d\n", customer);
                return -1;
        }
        for(int i = 0; i != NUMBER_OF_RESOURCES; ++i) {
                if(release[i] < 0 || release[i] > allocation[customer][i]) {
                        printf("Invalid number of resources to release:\n(customer = %d, res
                                        customer, i, allocation[customer][i], release[i]);
                        // rollback
                        while(i--) {
                                allocation[customer][i - 1] += release[i - 1];
                                available[i] -= release[i];
                        }
                        return -1;
                }
                // release resources
                allocation[customer][i] -= release[i];
                available[i] += release[i];
        }
        return 0;
}
```

```
//Checker of release resources.
void release_wrapper() {
        int release[NUMBER_OF_RESOURCES], customer;
        scanf("%d", &customer);
        for(int i = 0; i != NUMBER_OF_RESOURCES; ++i) {
                scanf("%d", &release[i]);
        }
        if(release_resources(customer, release) != 0) {
                printf("FAILED.\n");
        }
        else {
                printf("SUCCEEDED.\n");
        }
}
```

Safety Checker:

```
//Check if the whether the operation is safe to execute.
int is_safe() {
	int work[NUMBER_OF_RESOURCES], finish[NUMBER_OF_CUSTOMERS];
	memcpy(work, available, NUMBER_OF_RESOURCES * sizeof(int));
	memset(finish, 0, NUMBER_OF_CUSTOMERS * sizeof(int));
	for(int round = 0; round != NUMBER_OF_CUSTOMERS; ++round) {
		int flag = 0;
		for(int i = 0; i != NUMBER_OF_CUSTOMERS; ++i) {
			if(finish[i] == 0 && is_leq(need[i], work, NUMBER_OF_RESOURCES)) {
				flag = 1;
				finish[i] = 1;
				for(int j = 0; j != NUMBER_OF_RESOURCES; ++j) {
					work[j] += allocation[i][j];
				}
				break;
			}
		}
		if(!flag) {
			return 0;
		}
	}
	return 1;
}
```

It plays the role of checking a status after doing an operation is safe or not, if the result is false, the program will not do the operation.

Compile (run.sh):

```
echo "gcc -o bankers bankers.c"
echo "./bankers 10 6 7 8"
echo "You can also use your arguments by './bankers <arguments>'"
echo ""
echo ""

gcc -o bankers bankers.c
./bankers 10 6 7 8
```

I set the default instances of each type of resources are 10, 6, 7, 8 respectively. You can also use your arguments by simply run and program use 4 arguments represents the number of instances.

Results:

```
xiaoqi@xiaoqi:~/Desktop/OS/HW5/Bankers$ ./run.sh
gcc -o bankers bankers.c
./bankers 10 6 7 8
You can also use your arguments by './bankers <arguments>'


Operations you can do:
<1>Request resources: RQ <index of customer> <request resources of each type>
   For example: '>> RQ 0 3 1 2 1': the 0th customer requests resources 3 1 2 1 respectively.
<2>Release resources: RL <index of customer> <request resources of each type>
   For example: '>> RL 0 1 1 1 1': the 0th customer release resources 1 1 1 1 respectively.
<3>Display resources: '*'
   It will show you the current informations (Availible, Maximum, Allocated, Needed) of each customer.
<4>Help: 'help'
   It will show you this information (usage) again.
<5>Exit: 'exit'
   Close and exit the program.
```

```
>> RQ 0 1 1 1 1
SUCCEEDED.
>> RQ 1 2 2 2 2
SUCCEEDED.
>> RQ 2 3 3 3 3
Invalid number of resources to request:
(customer = 2, resource = 0, need = 2, to request = 3)
FAILED.
>> RQ 0 1 1 1 1
SUCCEEDED.
>> RQ 1 2 2 2 2
Invalid number of resources to request:
(customer = 1, resource = 1, need = 0, to request = 2)
FAILED.
```

```
>> *
Availbale resources:
6 2 3 4
Maximum resources for each customer:
0: 6 4 7 3
1: 4 2 3 2
2: 2 5 3 3
3: 6 3 3 2
4: 5 6 7 5

Allocated resources for each customer:
0: 2 2 2 2
1: 2 2 2 2
2: 0 0 0 0
3: 0 0 0 0
4: 0 0 0 0

Needed resources for each customer:
0: 4 2 5 1
1: 2 0 1 0
2: 2 5 3 3
3: 6 3 3 2
4: 5 6 7 5
>> exit
xiaoqi@xiaoqi:~/Desktop/OS/HW5/Bankers$
```

1) You can see after I run this program by "run.sh", it will give you the usage to tell you what operations you can do in the terminal and give you some examples. You can use "help" to get the information again.
2) You can see when I request too much resources, it will reach an unsafe status and give you the current information. Also, if you release too much resources, it will reach an unsafe status and give you the current information.
3) After I used "*", it displayed the current information of Available, Maximum, Allocated, Needed resources for each customer.

Other Codes:

```
//Display the compile usage of this program.
void display_initial_usage() {
	printf("The number of resourses type is %d.\n", NUMBER_OF_RESOURCES);
	printf("You need to enter the number of instances of each type.\n");
	printf("For example: './bankers 10 6 7 8' for number of type = 4.\n");
}

//Display the usage of this program, how to use it.
void display_usage() {
	printf("Operations you can do:\n");
	printf("<1>Request resources: RQ <index of customer> <request resources of each type>\n");
	printf("   For example: '>> RQ 0 3 1 2 1': the 0th customer requests resources 3 1 2 1 respectively.\
	printf("<2>Release resources: RL <index of customer> <request resources of each type>\n");
	printf("   For example: '>> RL 0 1 1 1 1': the 0th customer release resources 1 1 1 1 respectively.\n'
	printf("<3>Display resources: '*'\n");
	printf("   It will show you the current informations (Availible, Maximum, Allocated, Needed) of each
	printf("<4>Help: 'help'\n");
	printf("   It will show you this information (usage) again.\n");
	printf("<5>Exit: 'exit'\n");
	printf("   Close and exit the program.\n\n");
}

//Display the information of current resurces.
void display_resources() {
	printf("Availbale resources:\n");
	for(int i = 0; i != NUMBER_OF_RESOURCES; ++i) {
		printf("%d ", available[i]);
	}
	printf("\n");
	printf("Maximum resources for each customer:\n");
	for(int customer = 0; customer != NUMBER_OF_CUSTOMERS; ++customer) {
		printf("%d: ", customer);
		for(int r = 0; r != NUMBER_OF_RESOURCES; ++r) {
			printf("%d ", maximum[customer][r]);
		}
```

```
		printf("\n");
	}
	printf("\n");
	printf("Allocated resources for each customer:\n");
	for(int customer = 0; customer != NUMBER_OF_CUSTOMERS; ++customer) {
		printf("%d: ", customer);
		for(int r = 0; r != NUMBER_OF_RESOURCES; ++r) {
			printf("%d ", allocation[customer][r]);
		}
		printf("\n");
	}
	printf("\n");
	printf("Needed resources for each customer:\n");
	for(int customer = 0; customer != NUMBER_OF_CUSTOMERS; ++customer) {
		printf("%d: ", customer);
		for(int r = 0; r != NUMBER_OF_RESOURCES; ++r) {
			printf("%d ", need[customer][r]);
		}
		printf("\n");
	}
}
```

```
//Initialization of resources from the textbook.
int init(int argc, char *argv[], const char *resources_file) {
        if(argc != 1 + NUMBER_OF_RESOURCES) {
                printf("Incorrect number of parameters.\n");
                return -1;
        }
        for(int i = 0; i != NUMBER_OF_RESOURCES; ++i) {
                available[i] = atoi(argv[i + 1]);
        }
        FILE *f = fopen(resources_file, "r");
        if(f == NULL) {
                printf("Unable to open file: %s\n", resources_file);
                printf("The resoureces.txt is the initialization from the textbook\n");
                return -2;
        }
        for(int c = 0; c != NUMBER_OF_CUSTOMERS; ++c) {
                for(int r = 0; r != NUMBER_OF_RESOURCES; ++r) {
                        fscanf(f, "%d", &maximum[c][r]);
                        need[c][r] = maximum[c][r];
                }
        }
        fclose(f);
        return 0;
}

int is_leq(int* a, int* b, int n) {
        for(int i = 0 ; i != n; ++i) {
                if(a[i] > b[i]) {
                        return 0;
                }
        }
        return 1;
}
```

That’s the end of this report, thank you very much for your attention!
Xiaoqi LIU 999009335