Project 7: Contiguous Memory Allocation

Yanjie Ze 519021910706

1 Introduction

In OS lectures, we have learned different algorithms for contiguous memory allocation. This project will involve managing a contiguous region of memory of size *MAX* where addresses may range from 0 to MAX−1. Our program will implement four different requests:

1. Request for a contiguous block of memory
2. Release of a contiguous block of memory
3. Compact unused holes of memory into one single block
4. Report the regions of free and allocated memory

1.1 Commands

First, our program will be passed the initial amount of memory at startup. For example, the following initializes the program with 1 MB (1,048,576 bytes) of memory:

```
./allocator 1048576
```

Once our program has started, it will present the user with the following prompt:

```
allocator>
```

Then, there are five commands needed to be implemented:

`RQ` (request), `RL` (release), `C` (compact), `STAT` (status report), and `X` (exit)

In our implementation, we make the command `C` invisible, which menas the compact is executed automatically.

1.1.1 RQ

A request for 40,000 bytes will appear as follows:

```
allocator>RQ P0 40000 W
```

The first parameter to the RQ command is the new process that requires the memory, followed by the amount of memory being requested, and finally the strategy. (In this situation, “W” refers to worst fit.)

And we support three allocation strategies:

- B, best fit
- W, worst fit
- F, first fit

1.1.2 RL

Similarly, a release will appear as:

```
allocator>RL P0
```

This command will release the memory that has been allocated to process P0.

1.1.3 STAT

The `STAT` command for reporting the status of memory is entered as:

```
allocator>STAT
```

Given this command, your program will report the regions of memory that are allocated and the regions that are unused. For example, one possible arrangement of memory allocation would be as follows:

```
Addresses [0:315000] Process P1
Addresses [315001: 512500] Process P3
Addresses [512501:625575] Unused
Addresses [625575:725100] Process P6
Addresses [725001] . . .
```

1.1.4 X

The `X` command for exiting the program is entered as:

```
allocator>X
```

Then the program will end the execution.

1.2 Allocating Memory

Our program will allocate memory using one of the three approaches high lighted in Section 9.2.2, depending on the flag that is passed to the RQ command. The flags are:

- F—first fit
- B—best fit
- W—worst fit

Simulations have shown that both first fit and best fit are better than worst fit in terms of decreasing time and storage utilization. Neither first fit nor best fit is clearly better than the other in terms of storage utilization, but first fit is generally faster.

1.2.1 First Fit

First Fit allocates the first hole that is big enough. Searching can start either at the beginning of the set of holes or at the location where the previous first-fit search ended. We can stop searching as soon as we find a free hole that is large enough.

1.2.2 Best Fit

Best Fit allocate the smallest hole that is big enough. We must search the entire list, unless the list is ordered by size. This strategy produces the smallest leftover hole.

1.2.3 Worst Fit

Worst Fit allocates the largest hole. Again, we must search the entire list,unless it is sorted by size. This strategy produces the largest leftover hole, which may be more useful than the smaller leftover hole from a best-fit approach.

1.3 Compaction

In my implementation, the user **do not need** to enter `C` command to compact the hole.

This is because **when freeing the process, we will check the hole and compact them automatically.**

For example, if we have four separate holes of size 550 KB, 375 KB, 1,900 KB, and 4,500 KB, our program will combine these four holes into one large hole of size 7,325 KB.

2 Implementation Methods and Details

2.1 Basic Data Structure

To make the allocation of processes more simple, we create a new struct called `process`, as shown below.

It has five attributes:

- **start**, the start position
- **end**, the end position

- **capacity**, the space of the process. This is also equal to:

$$compacity = end - start$$

- **id**, the pid of the process. **We use -1 to assign unused space.**
- **next**, a pointer which points to the adjacent process.

```
typedef struct Process
{
    long start;
    long end;
    long capacity;
    int id;
    struct Process *next;
    // when id=-1, this means "unused".
} process;
```

2.2 Basic Idea

In this implementation, we actually treat every space as a struct **process**.

For example, if the space now is all unused, then there exists only one process. The attributes of this process are:

- start=0,
- end=space capacity,
- capacity=start-end
- id=-1
- next=NULL

Thus, we can represent any space allocation and process by this struct, and distinguish them by their id.

From this view, the memory space consists of several **process(the struct we define)**.

We can use a **link list** to link them together.

2.3 `main()`: Initialize And Parse

We will first introduce how we construct `main()`, and then introduce the implementation of three allocation strategies.

First, we read the argument passed from the command line and create the head node of the link list.

The first node is the node that represents the whole space.

```
// Memory Initialize
    long max_space;
    max_space = atoi(argv[1]);
    process *memory=malloc(sizeof(process));
    memory->start = 0;
    memory->end =  max_space-1;
    memory->compacity = max_space;//compacity = end - start + 1
    memory->next = NULL;
    memory->id = -1;

    process **memory_head=malloc(sizeof(process*));
    *memory_head=memory;
```

Second, parse the command.

`X` command is implemented directly:

```
// "X": exit
      if(instruction[0]=='X')
      {
          running = 0;
          continue;
      }
```

`STAT` command is parsed in a trivial way. It's implemented like this because this form ensures no bug.

Noe that the function `show_state()` will be shown in the following section.

```
// "STAT": print states

 if(instruction[0]=='S'&&instruction[1]=='T'&&instruction[2]=='A'&&instruct
ion[3]=='T')
            show_state(memory_head);
```

`RQ` command is somewhat complex.

Recall that the RQ command has the shape like this:

```
allocator>RQ P0 40000 W
```

Therefore, we need to get the id of the process, the needed space of the process and the allocation strategy.

The parsing codes are shown below. The three algorithms will be introduced in the following section.

```
 // "RQ": request
        if(instruction[0]=='R'&&instruction[1]=='Q')
        {
            temp = strdup(instruction);
            strsep(&temp, " ");
            char *process_name=strsep(&temp, " ");
            process_name++; // skip "P"

            int process_id = atoi(process_name);
            long process_size = atoi(strsep(&temp, " "));
            char *insert_mode = strsep(&temp, " ");

            process *new_process=malloc(sizeof(process));
            new_process->id = process_id;
            new_process->compacity = process_size;
```

```
            // execute algorithm
            int allocation_success=0;
            if(insert_mode[0]=='F')
                allocation_success = first_fit(memory_head, new_process);
            else if (insert_mode[0]=='B')
                allocation_success = best_fit(memory_head, new_process);
            else if (insert_mode[0]=='W')
                allocation_success = worst_fit(memory_head, new_process);
            else
            {
                printf("Error: No such mode.\n");
                continue;
            }
            if(allocation_success)
                printf("Allocate P%d Success.\n", new_process->id);
            else
                printf("Allocate P%d Fail!\n", new_process->id);
        }
```

`RL` command needs to parse out the id of the process. The function `release_process()` is shown in the following section.

```
if(instruction[0]=='R'&&instruction[1]=='L')
        {
            temp = strdup(instruction);
            strsep(&temp, " ");
            char *process_name=strsep(&temp, " ");
            process_name++; // skip "P"
            int process_id = atoi(process_name);

            int release_success = release_process(memory_head, process_id);

            if(release_success)
                printf("Release P%d success.\n", process_id);
            else
```

```
                printf("Release P%d fail.\n", process_id);
        }
```

After introducing the several parts of `main()`, we have the whole implementation of `main()`, as shown below.

```
int main(int argc, char* argv[])
{
    // Memory Initialize
    long max_space;
    max_space = atoi(argv[1]);
    process *memory=malloc(sizeof(process));
    memory->start = 0;
    memory->end =  max_space-1;
    memory->compacity = max_space;//compacity = end - start + 1
    memory->next = NULL;
    memory->id = -1;

    process **memory_head=malloc(sizeof(process*));
    *memory_head=memory;


    int running=1;
    char instruction[SIZE];
    char *temp;

    while (running)
    {
        printf("allocator>");
        fgets(instruction, SIZE, stdin); //会读入换行符

        // "X": exit
        if(instruction[0]=='X')
        {
            running = 0;
            continue;
```

```
        }

        // "STAT": print states

 if(instruction[0]=='S'&&instruction[1]=='T'&&instruction[2]=='A'&&instruct
ion[3]=='T')
            show_state(memory_head);

        // "RQ": request
        if(instruction[0]=='R'&&instruction[1]=='Q')
        {
            temp = strdup(instruction);
            strsep(&temp, " ");
            char *process_name=strsep(&temp, " ");
            process_name++; // skip "P"

            int process_id = atoi(process_name);
            long process_size = atoi(strsep(&temp, " "));
            char *insert_mode = strsep(&temp, " ");

            process *new_process=malloc(sizeof(process));
            new_process->id = process_id;
            new_process->compacity = process_size;

            // execute algorithm
            int allocation_success=0;
            if(insert_mode[0]=='F')
                allocation_success = first_fit(memory_head, new_process);
            else if (insert_mode[0]=='B')
                allocation_success = best_fit(memory_head, new_process);
            else if (insert_mode[0]=='W')
                allocation_success = worst_fit(memory_head, new_process);
            else
            {
                printf("Error: No such mode.\n");
                continue;
            }
```

```
            if(allocation_success)
                printf("Allocate P%d Success.\n", new_process->id);
            else
                printf("Allocate P%d Fail!\n", new_process->id);
        }

        if(instruction[0]=='R'&&instruction[1]=='L')
        {
            temp = strdup(instruction);
            strsep(&temp, " ");
            char *process_name=strsep(&temp, " ");
            process_name++; // skip "P"
            int process_id = atoi(process_name);

            int release_success = release_process(memory_head, process_id);

            if(release_success)
                printf("Release P%d success.\n", process_id);
            else
                printf("Release P%d fail.\n", process_id);
        }
    }
}
```

2.4 first_fit() and algorithm

We introduce the concrete implementation steps here. **First Fit** is implemented with the following steps:

1. Use the link list to loop over
2. Find the first free space that satisfies the requirement.(represented as a process)
3. Determine whether this allocation will use up this free hole
4. If so, just change the old process's attributes.
5. If not, change the old process's attributes, create a new process, and insert it.

The codes below are exactly based on these procedures.

```
int first_fit(process **memory_head, process* new_process)
{
    process *temp_process=malloc(sizeof(process));
    temp_process = (*memory_head);

    if(temp_process->id==-1)// 第一次分配
    {
        if(temp_process->compacity>=new_process->compacity)//满足分配情况
        {
            new_process->start = 0;
            new_process->end = new_process->compacity - 1;
            temp_process->compacity -= new_process->compacity;
            temp_process->start = new_process->end + 1;

            if(temp_process->compacity==0)//用完unused block
            {
                new_process->next=NULL;
                free(temp_process);
            }
            else
                new_process->next = temp_process;

            *memory_head = new_process;
            return 1;//第一次分配，分配成功
        }
        else
            return 0;//分配失败
    }

    process *prev_process=malloc(sizeof(process));

    while (temp_process->next!=NULL)
    {
        prev_process = temp_process;
        temp_process = temp_process->next;
        if(temp_process->id!=-1)
```

```
            continue;

        // find an unused hole
        if(temp_process->compacity<new_process->compacity)
            continue;

        //allocation
        temp_process->compacity -= new_process->compacity;
        new_process->start = temp_process->start;
        new_process->end = new_process->start + new_process->compacity - 1;
        prev_process->next = new_process;
        temp_process->start = new_process->end + 1;
        if(temp_process->compacity==0)
        {
            new_process->next=temp_process->next;
            free(temp_process);
        }
        else
        {
            new_process->next = temp_process;
        }

        return 1;// 分配成功
    }

    return 0;
}
```

2.5 best_fit() and algorithm

Best Fit is surely different from **First Fit**. We need to find the smallest hole that satisfies, which means we need to traverse all the process.

The steps we implement **Best fit** are:

1. Traverse the link list and get **the smallest hole** that satisfies the requirement.

2. Record the start of this best process.(the process's id is -1, to reperesent the free space)
3. After finishing traversing, restart from the head of the link list.
4. Go to the start we record and make the allocation.
5. When allocating, determing whether we use up this free hole.
6. If so, just change the old process's attributes.
7. If not, change the old process's attributes, create the new process, insert it.

That's all.

Based on this, we implement as follows:

```
int best_fit(process **memory_head, process* new_process)
{
    process *temp_process=malloc(sizeof(process));
    temp_process = (*memory_head);

    if(temp_process->id==-1)// 第一次分配
    {
        if(temp_process->compacity>=new_process->compacity)//满足分配情况
        {
            new_process->start = 0;
            new_process->end = new_process->compacity - 1;
            temp_process->compacity -= new_process->compacity;
            temp_process->start = new_process->end + 1;

            if(temp_process->compacity==0)//用完unused block
            {
                new_process->next=NULL;
                free(temp_process);
            }
            else
                new_process->next = temp_process;

            *memory_head = new_process;
            return 1;//第一次分配，分配成功
```

```
    }
    else
        return 0;//分配失败
}

process *prev_process=malloc(sizeof(process));
int minimal_cp = 0;
while (temp_process->next!=NULL)
{
    prev_process = temp_process;
    temp_process = temp_process->next;
    if(temp_process->id!=-1)
        continue;

    // find an unused hole
    if(temp_process->compacity<new_process->compacity)
        continue;

    if(minimal_cp==0)// first find
        minimal_cp=temp_process->compacity;

    if(temp_process->compacity<minimal_cp)
        minimal_cp = temp_process->compacity;
}

if(minimal_cp==0)
    return 0;

temp_process = (*memory_head);
while (temp_process->next!=NULL)
{
    prev_process = temp_process;
    temp_process = temp_process->next;
    if(temp_process->id!=-1)
        continue;

    // find an unused hole
```

```
        if(temp_process->compacity<new_process->compacity)
            continue;

        if(temp_process->compacity==minimal_cp)
        {
        //allocation
        temp_process->compacity -= new_process->compacity;
        new_process->start = temp_process->start;
        new_process->end = new_process->start + new_process->compacity - 1;
        prev_process->next = new_process;
        temp_process->start = new_process->end + 1;
        if(temp_process->compacity==0)
        {
            new_process->next=temp_process->next;
            free(temp_process);
        }
        else
        {
            new_process->next = temp_process;
        }

        return 1;// 分配成功
        }
    }
    return 0;
}
```

2.6 worst_fit() and algorithm

Worst Fit is similar to **Best Fit**. They are actually two side of the allocation.

Therefore, the algorithm we implement is similar.

The steps we implement **Worst fit** are:

1. Traverse the link list and get **the largest hole** that satisfies the requirement.
2. Record the start of this best process.(the process's id is -1, to reperesent the free space)
3. After finishing traversing, restart from the head of the link list.
4. Go to the start we record and make the allocation.
5. When allocating, determing whether we use up this free hole.
6. If so, just change the old process's attributes.
7. If not, change the old process's attributes, create the new process, insert it.

That's all.

Based on this, we implement as follows:

```
int worst_fit(process **memory_head, process* new_process)
{
    process *temp_process=malloc(sizeof(process));
    temp_process = (*memory_head);

    if(temp_process->id==-1)// 第一次分配
    {
        if(temp_process->compacity>=new_process->compacity)//满足分配情况
        {
            new_process->start = 0;
            new_process->end = new_process->compacity - 1;
            temp_process->compacity -= new_process->compacity;
            temp_process->start = new_process->end + 1;

            if(temp_process->compacity==0)//用完unused block
            {
                new_process->next=NULL;
                free(temp_process);
            }
            else
                new_process->next = temp_process;

            *memory_head = new_process;
            return 1;//第一次分配，分配成功
```

```
    }
    else
        return 0;//分配失败
}

process *prev_process=malloc(sizeof(process));
int max_cp = 0;
while (temp_process->next!=NULL)
{
    prev_process = temp_process;
    temp_process = temp_process->next;
    if(temp_process->id!=-1)
        continue;

    // find an unused hole
    if(temp_process->compacity<new_process->compacity)
        continue;

    if(max_cp==0)// first find
        max_cp=temp_process->compacity;

    if(temp_process->compacity>max_cp)
        max_cp = temp_process->compacity;
}

if(max_cp==0)
    return 0;

temp_process = (*memory_head);
while (temp_process->next!=NULL)
{
    prev_process = temp_process;
    temp_process = temp_process->next;
    if(temp_process->id!=-1)
        continue;

    // find an unused hole
```

```
        if(temp_process->compacity<new_process->compacity)
            continue;

        if(temp_process->compacity==max_cp)
        {
        //allocation
        temp_process->compacity -= new_process->compacity;
        new_process->start = temp_process->start;
        new_process->end = new_process->start + new_process->compacity - 1;
        prev_process->next = new_process;
        temp_process->start = new_process->end + 1;
        if(temp_process->compacity==0)
        {
            new_process->next=temp_process->next;
            free(temp_process);
        }
        else
        {
            new_process->next = temp_process;
        }

        return 1;// 分配成功
        }
    }
    return 0;
}
```

2.7 release_process() : release and compact

`release_process()` release teh process by its process id.

Simply, we traverse the link list and find the corresponding process by its process id, and set the id of it as **-1**.

What's more, we do compression after **releasing** the process.

The compression algorithm is:

1. Traverse the link list, and record the previous process when traversing.
2. If the previous process's id==-1, and the current process's id ==-1, we do compression.
3. When we do compression: We remove the previous process and change the current process's attributes:

$$Assume\ two\ process: p_0(previous), p_1(current)$$

$$p_1 \rightarrow start = p_0 \rightarrow start$$

$$p_1 \rightarrow capacity = p_0 \rightarrow capacity + p_1 \rightarrow capacity$$

That's all.

Based on this, we implement the function `release_process()`.

```
int release_process(process **memory_head, int process_id)
{
    process *temp_process=malloc(sizeof(process));
    process *prev_process=malloc(sizeof(process));
    temp_process = (*memory_head);
    prev_process = NULL;

    while(temp_process!=NULL)
    {
        if(temp_process->id==process_id)
            temp_process->id = -1;
        temp_process = temp_process->next;
    }

    //compress unused block
    temp_process = (*memory_head);

    while(temp_process!=NULL)
    {
        if(temp_process->id==-1)
        {
```

```
            if(prev_process!=NULL&&prev_process->id==-1)
            {
                prev_process->compacity += temp_process->compacity;
                prev_process->end = temp_process->end;
                prev_process->next = temp_process->next;
                free(temp_process);
                temp_process = prev_process->next;
            }
            else if(prev_process==NULL&&temp_process->next==NULL)
            {
                temp_process->id=-1;
                return 1;
            }
            else
            {
                prev_process = temp_process;
                temp_process = temp_process->next;
            }
        }
        else
        {
            prev_process = temp_process;
            temp_process = temp_process->next;
        }

    }
    return 1;
}
```

2.8 show_state()

First, we create a function `show_process_state` . The function displays one process's all information.

We need to **distinguish** between the unused space and the real process here, by **the process id.**

```
void show_process_state(process *single_process)
{
    int id = single_process->id;
    int cp = single_process->compacity;
    if(id==-1)
        if(cp==1)
            printf("Addresses [%ld] Unused\n", single_process->start);
        else
            printf("Addresses [%ld:%ld] Unused\n", single_process->start,
single_process->end);
    else
        if(cp==1)
            printf("Addresses [%ld] Process P%d\n", single_process->start,
single_process->id);
        else
            printf("Addresses [%ld:%ld] Process P%d\n", single_process-
>start, single_process->end, single_process->id);
}
```

Then, by using this function, we can write out `show_state()`.

We traverse the whole link list and call `show_process_state()` for each process.

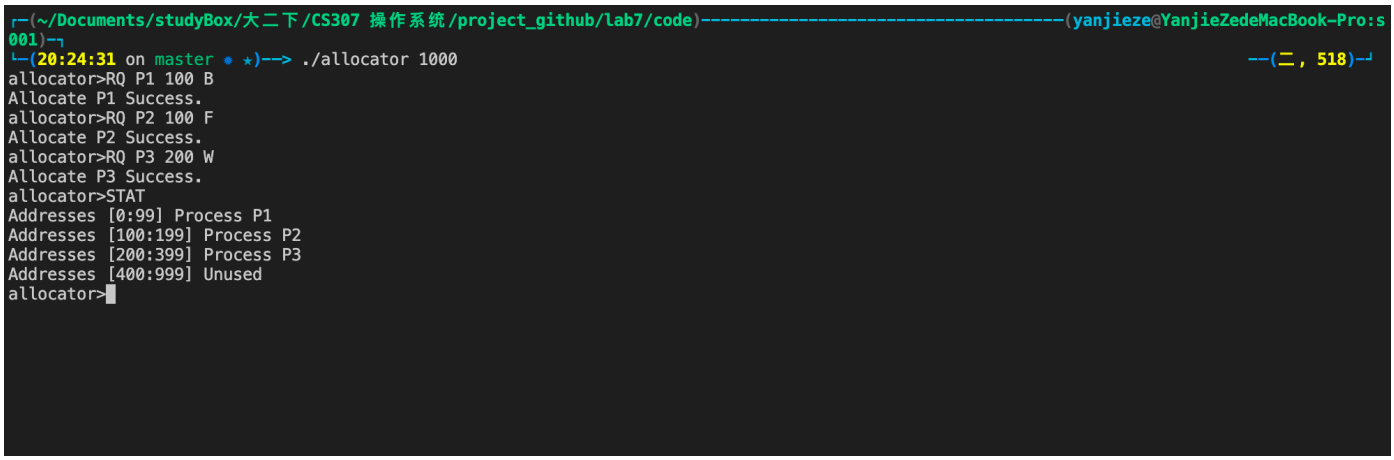
```
void show_state(process **memory_head)
{
    process *temp_process=malloc(sizeof(process));
    temp_process = (*memory_head);
    show_process_state(temp_process);

    while (temp_process->next!=NULL)
    {
        temp_process = temp_process->next;
        show_process_state(temp_process);
    }
}
```

3 Program Results

3.1 RQ

We test three allocation algorithms, and use `STAT` to show the status.

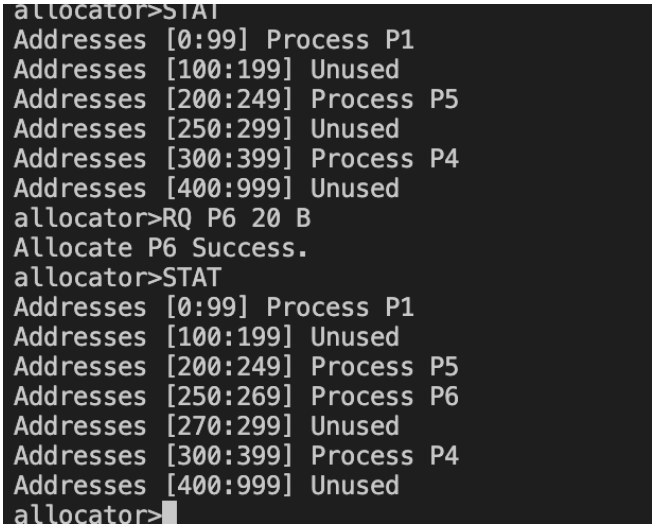


This illustrates the comman situation can work.

We further test the spectial situation for each algorithm.

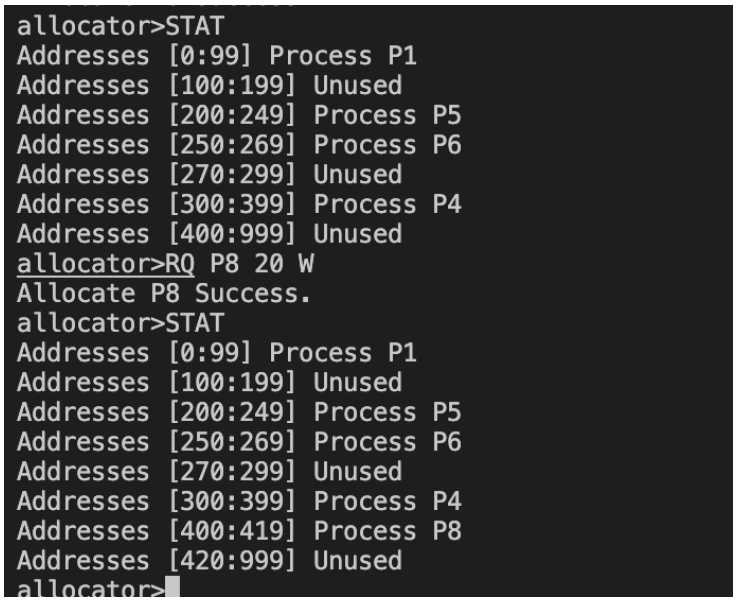
3.2 RQ-Best Fit

Best Fit selects the best space successfully.



3.3 RQ-Worst Fit

Worst Fit selects the largest suitable space.



3.4 RQ-First Fit

First Fit selects the first suitable space.

```
allocator>STAT
Addresses [0:99] Process P1
Addresses [100:199] Unused
Addresses [200:249] Process P5
Addresses [250:269] Process P6
Addresses [270:299] Unused
Addresses [300:399] Process P4
Addresses [400:419] Process P8
Addresses [420:999] Unused
allocator>RQ P2 10 F
Allocate P2 Success.
allocator>STAT
Addresses [0:99] Process P1
Addresses [100:109] Process P2
Addresses [110:199] Unused
Addresses [200:249] Process P5
Addresses [250:269] Process P6
Addresses [270:299] Unused
Addresses [300:399] Process P4
Addresses [400:419] Process P8
Addresses [420:999] Unused
```

3.5 RL

We release P1 successfully.

```
allocator>STAT
Addresses [0:99] Process P1
Addresses [100:109] Process P2
Addresses [110:199] Unused
Addresses [200:249] Process P5
Addresses [250:269] Process P6
Addresses [270:299] Unused
Addresses [300:399] Process P4
Addresses [400:419] Process P8
Addresses [420:999] Unused
allocator>RL p1
Release P1 success.
allocator>STAT
Addresses [0:99] Unused
Addresses [100:109] Process P2
Addresses [110:199] Unused
Addresses [200:249] Process P5
Addresses [250:269] Process P6
Addresses [270:299] Unused
Addresses [300:399] Process P4
Addresses [400:419] Process P8
Addresses [420:999] Unused
```

3.6 Compact

When we do releasing, the space is compacted perfectly.

```
allocator>STAT
Addresses [0:99] Unused
Addresses [100:109] Process P2
Addresses [110:199] Unused
Addresses [200:249] Process P5
Addresses [250:269] Process P6
Addresses [270:299] Unused
Addresses [300:399] Process P4
Addresses [400:419] Process P8
Addresses [420:999] Unused
allocator>RL P5
Release P5 success.
allocator>STAT
Addresses [0:99] Unused
Addresses [100:109] Process P2
Addresses [110:249] Unused
Addresses [250:269] Process P6
Addresses [270:299] Unused
Addresses [300:399] Process P4
Addresses [400:419] Process P8
Addresses [420:999] Unused
```

3.6 STAT

Work perfectly.

```
allocator>STAT
Addresses [0:99] Unused
Addresses [100:109] Process P2
Addresses [110:199] Unused
Addresses [200:249] Process P5
Addresses [250:269] Process P6
Addresses [270:299] Unused
Addresses [300:399] Process P4
Addresses [400:419] Process P8
Addresses [420:999] Unused
```

3.7 X

Finnaly,we exit the program by `X`.

```
allocator>RL P5
Release P5 success.
allocator>STAT
Addresses [0:99] Unused
Addresses [100:109] Process P2
Addresses [110:249] Unused
Addresses [250:269] Process P6
Addresses [270:299] Unused
Addresses [300:399] Process P4
Addresses [400:419] Process P8
Addresses [420:999] Unused
allocator>X
┌─(~/Documents/studyBox/大二下/CS307 操作系统/project_github/lab7/code)----------------------------------(yanjieze@YanjieZedeMacBook-Pro:s
001)─┐
└─(20:36:10 on master ● ★)--> ▉                                                                          ──(二, 518)─┘
```

4 Conclusion and Thoughts

Project 7 guides us to finish the simulation of continous memory allocation and implement three significant allcation strategies: **Best Fit, Worst Fit, First Fit**.

The memory allocation is a very basic and important problem in Operating System. After many operations, the memory space may have losts of holes. By this project, we get a very deep knowledge of memory allocation.

What's more, the design of implementation in this project is very interesting. I use a **process** to represent all the space and the process, which makes me easy to handle them together. A good data structue makes the result nice but only needs us to do half of the work.

In conclusion, we go deep into the memory allocation algorithm in Project 7, and practise the skill of using programs to solve problems.

Thanks for Prof. Wu and all TAs!