Operating System Course

Report for Lab3

Student Name	Student Number
马洪升	20175188

2018-12

Catalogue

1. Experiment Basic Information

1.1 Theme

Memory Placement Algorithm Analyses

1.2 Purpose

(1) Learn about Operating System Analyze memory management.

(2) Construct a simulation environment.

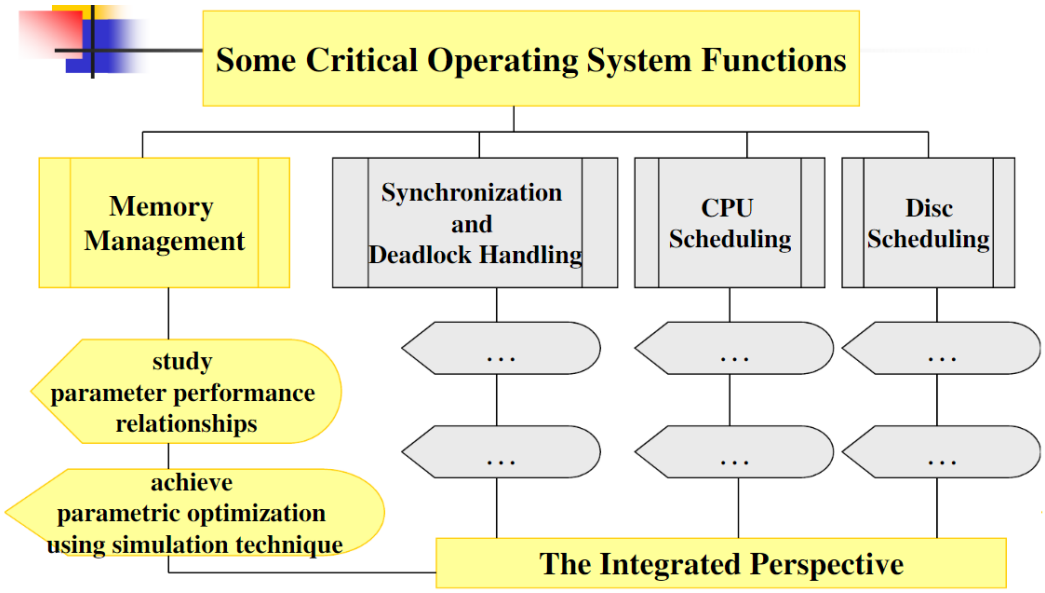
(3) Learn to use First Fit, Best Fit, Next Fit algorithm.

(4) Capture statistics on the state of the memory.

(5) Draw graph to analyze the performance of these three algorithms.

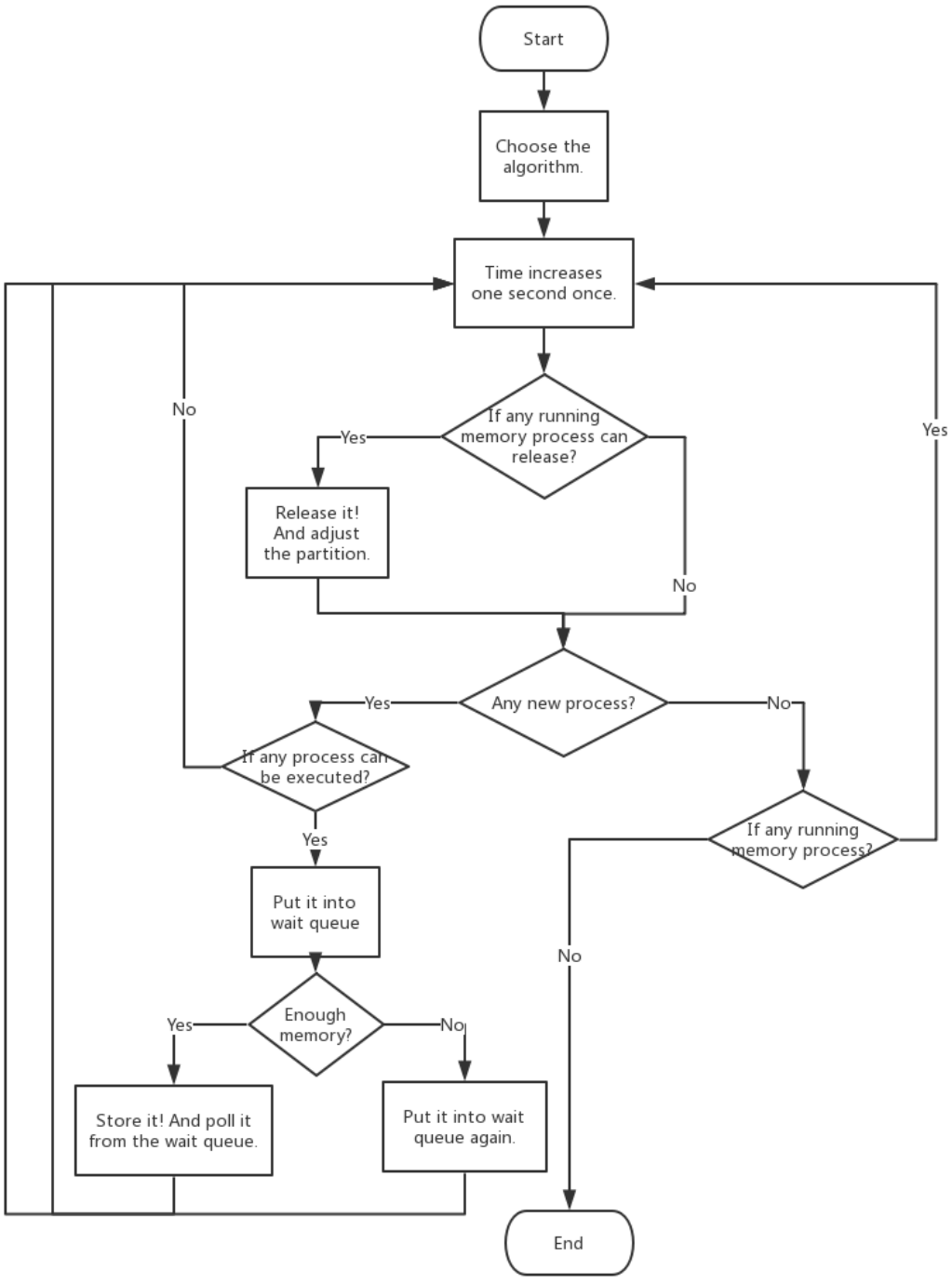
(6) Implement this experiment using Java.

(7) After doing this, we can understand the entire process management.



2. Flowchart & Methodology

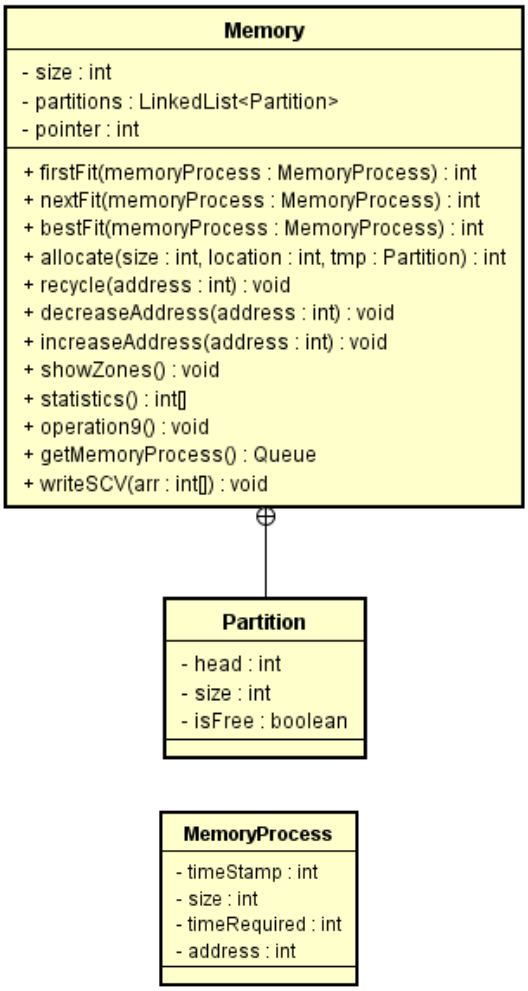
2.1 Flowchart



The above graph is the total flowchart, including memory management and CPU scheduling.

2.2 Methodology

(1) Class Diagram



(2) Memory Structure

From the class diagram, we can see that I create four classes.

MemoryProcess: I use this class to simulate a memory process, it has arrival time, memory size, required time and address.

Partition: I use this class to store every memory process. It has head, size and isFree.

Memory: I use this class to achieve the whole memory management. I create a linked list as a memory, it has 1024 blips. And I add partition into the linked list. This class also has many methods to help me achieve the experiment, I will introduce them soon.

FCFS: I use this class as a scheduling. It can handle the processes like Lab2.

(3) Method to Simulate the Request

a) Declare 3 structures to store processes in different situation.

```
static Queue<MemoryProcess> newProcess; // We use this Queue to store the process that is new.
static ArrayList<MemoryProcess> handlingProcess; // We use this list to store the process that is running.
static Queue<MemoryProcess> waitProcess; // We use this Queue to store the process that doesn't finish.
```

b) Use FCFS to handle the processes.

```
public void FCFSAlgorithm(Memory memory, int algorithm) {
    int currTime = 1;
    // We traverse these three list or queue until they are empty.
    while (!waitProcess.isEmpty() || !newProcess.isEmpty()
|| !handlingProcess.isEmpty()){
        int size = handlingProcess.size();
        if (size !=0 ){
            // Iterate all the processes that are running.
            Iterator<MemoryProcess> iterator = handlingProcess.iterator();
            while (iterator.hasNext()){
                MemoryProcess memoryProcess = iterator.next();
                // If arrival time plus required time is less than current time, release!
                if ((memoryProcess.getTimeStamp() + memoryProcess.getTimeRequired())
<= currTime){
                  // Remove this process from the list.
                    iterator.remove();
                    System.out.println("Current time:  " + currTime + "s");
                    release(memory, memoryProcess);
                }
            }
        }
```

```
        // Ensure if the process's arrival time is less than current time.
        if (!newProcess.isEmpty()){
            if (newProcess.peek().getTimeStamp() <= currTime){
                waitProcess.add(newProcess.poll());
            }
        }
        // Run the process in the wait queue. Allocate them!
        if (!waitProcess.isEmpty()){
            System.out.println("Current time: " + currTime + "s");
            executeProcess(memory, waitProcess.poll(), algorithm);
        }
        // Iterate the time by one second.
        currTime++;
    }
}
```

c) First Fit

```
public int firstFit(MemoryProcess memoryProcess){
    int size = memoryProcess.getSize();
    for (pointer = 0; pointer < partitions.size(); pointer++){
        Partition tmp = partitions.get(pointer);
        // If the condition is satisfied. Store it.
        if (tmp.isFree && (tmp.size > size)){
            int newPointer = allocate(size, pointer, tmp);
            // Put it into handling process list.
            FCFS.handlingProcess.add(memoryProcess);
            // Return the position and assign it to process as an address.
            return newPointer;
        }
    }
    // Condition isn't satisfied. Put it into waiting.
    FCFS.waitProcess.add(memoryProcess);
    System.out.println("Out of memory, delay!");
    return -1;
}
```

d) Next Fit

```
public int nextFit(MemoryProcess memoryProcess){
    int size = memoryProcess.getSize();
    int len = partitions.size();
    // In case beyond the index of the size.
    if (pointer >= len){
        pointer = 0;
    }
    Partition tmp = partitions.get(pointer);
```

```
    // If the head of the zones is free, store it.
    if (tmp.isFree && (tmp.size > size)){
        allocate(size, pointer, tmp);
        pointer++;
        FCFS.handlingProcess.add(memoryProcess);
        return (pointer-1);
    }
    // Traverse the queue from last position to the end of the zones.
    for (; pointer < len ; pointer++){
        tmp = partitions.get(pointer);
        // Find the zone, and it's big enough.
        if (tmp.isFree && (tmp.size > size)){
            allocate(size, pointer, tmp);
            FCFS.handlingProcess.add(memoryProcess);
            return pointer;
        }
    }
    int prePointer = pointer;
    // Traverse from the head of the zones to last position.
    for (pointer = 0; pointer < prePointer; pointer++){
        tmp = partitions.get(pointer);
        //Find the zone, and it's big enough.
        if (tmp.isFree && (tmp.size > size)) {
            allocate(size, pointer, tmp);
            FCFS.handlingProcess.add(memoryProcess);
            return pointer;
        }
    }
    // Condition isn't satisfied. Put it into waiting.
    FCFS.waitProcess.add(memoryProcess);
    System.out.println("Out of memory, delay!");
    return -1;
}
```

e) Best Fit

```
public int bestFit(MemoryProcess memoryProcess){
    int size = memoryProcess.getSize();
    int flag = -1;
    int min = this.size;
    // Traverse the zones, find the most suitable one.
    for (pointer = 0; pointer < partitions.size(); pointer++){
        Partition tmp = partitions.get(pointer);
        if (tmp.isFree && (tmp.size > size)){
            if (min > tmp.size - size){
```

```
                min = tmp.size - size;
                flag = pointer;
            }
        }
    }
    if (flag == -1){
        // Condition isn't satisfied. Put it into waiting.
        FCFS.waitProcess.add(memoryProcess);
        System.out.println("Out of memory, delay!");
        return -1;
    }else {
        FCFS.handlingProcess.add(memoryProcess);
        allocate(size, flag, partitions.get(flag));
        // Return the position and assign it to process as an address.
        return flag;
    }
}
```

f) Encapsulate an allocate method.

```
public int allocate(int size, int location, Partition tmp){
        // Redistribute zone.
        Partition split = new Partition(tmp.head + size, tmp.size - size);
        partitions.add(location + 1, split);
        tmp.size = size;
        tmp.isFree = false;
        System.out.println("Allocate " + size + "Blips successfully.");
        // Count the free amount, free fragment amount, max free fragment size and
the min one.
        int[] statistics = statistics();
        // Write these to csv.
        writeCSV(statistics);
        // Change the address of all handling process.
        decreaseAddress(location);
        showZones();
        return (location);
}
```

g) Encapsulate a recycle method.

```
public void recycle(int address){
        if (address >= partitions.size()){
            System.out.println("No such zone!");
            return;
        }
        Partition tmp = partitions.get(address);
        int size = tmp.size;
```

```
        if (tmp.isFree) {
            System.out.println("Such zone has been free!");
            return;
        }
        // If the reclaimed partition is not the tail partition and the latter
partition is idle,
        // it is merged with the latter partition.
        if (address < partitions.size() - 1 && partitions.get(address + 1).isFree){
            Partition next = partitions.get(address + 1);
            tmp.size += next.size;
            partitions.remove(next);
            // Change the address of all handling process.
            increaseAddress(address);
        }
        // If the reclaimed partition is not the first partition and the previous
partition is idle,
        // it is merged with the previous partition.
        if (address > 0 && partitions.get(address - 1).isFree){
            Partition previous = partitions.get(address - 1);
            previous.size += tmp.size;
            partitions.remove(address);
            // Change the address of all handling process.
            increaseAddress(address);
            address--;
        }
        partitions.get(address).isFree = true;
        System.out.println("Recycle successfully! Recycle " + size + "Blips!");
        // Count the free amount, free fragment amount, max free fragment size and
the min one.
        int[] statistics = statistics();
        // Write to csv.
        writeCSV(statistics);
        showZones();
}
```

3. Result

I add some extra memory processes after the test data. Now I execute the program with Best Fit algorithm. The result is below.

Begin

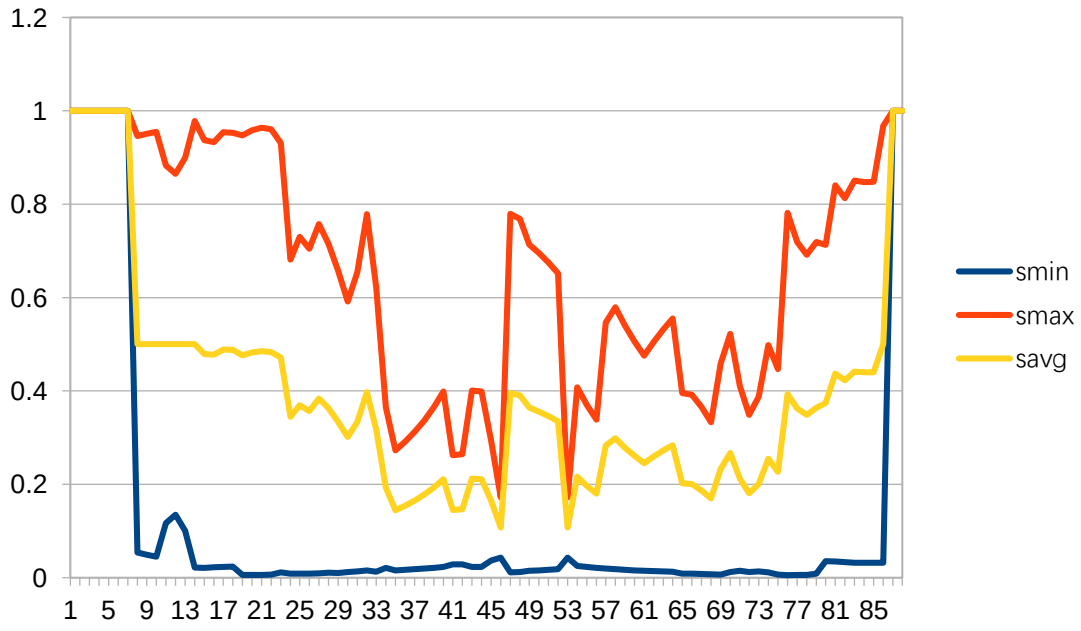
```
-----------------------------------------
1.FirstFit
2.NextFit
3.BestFit
Please choose the algorithm by its number: 1
Current time: 1s
Allocate 16Blips successfully.
-------------------------------------------------------------
Partition ID    Partition head  Partition size  State
-------------------------------------------------------------
0               0               16              false
1               16              1008            true
-------------------------------------------------------------
```

End

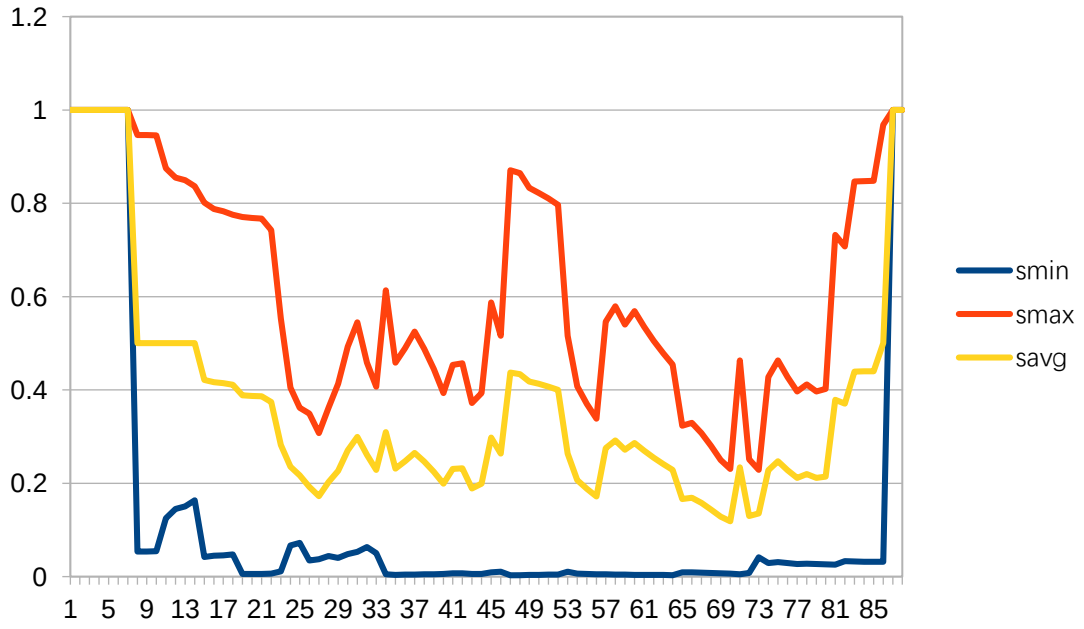
```
-------------------------------------------------------------
Current time:  89s
Recycle successfully! Recycle 16Blips!
-------------------------------------------------------------
Partition ID    Partition head  Partition size  State
-------------------------------------------------------------
0               0               1024            true
-------------------------------------------------------------
```

Statistic

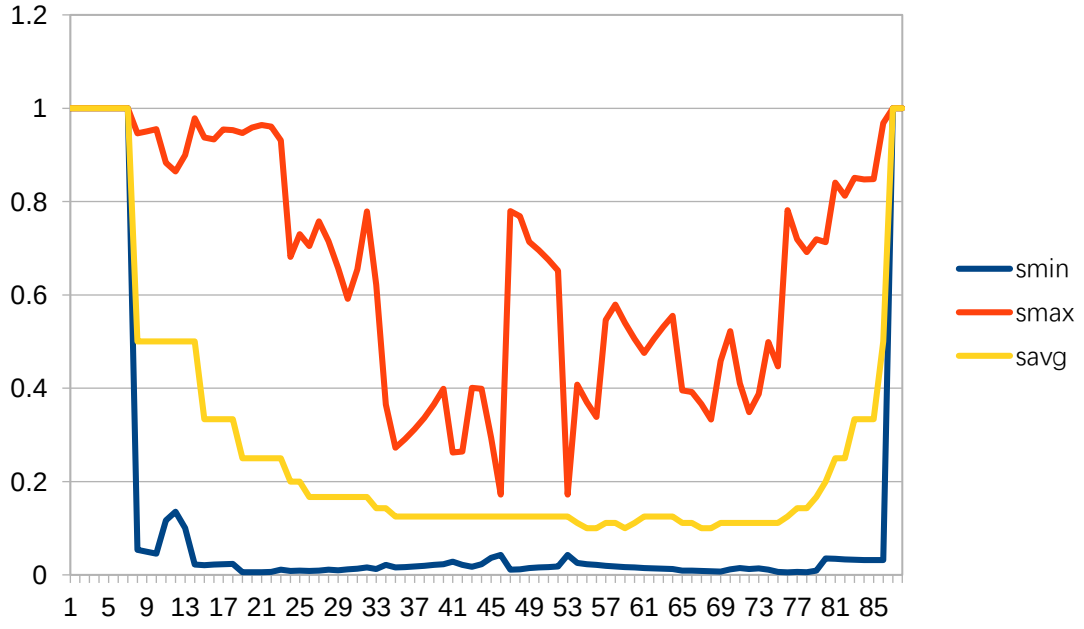
First Fit



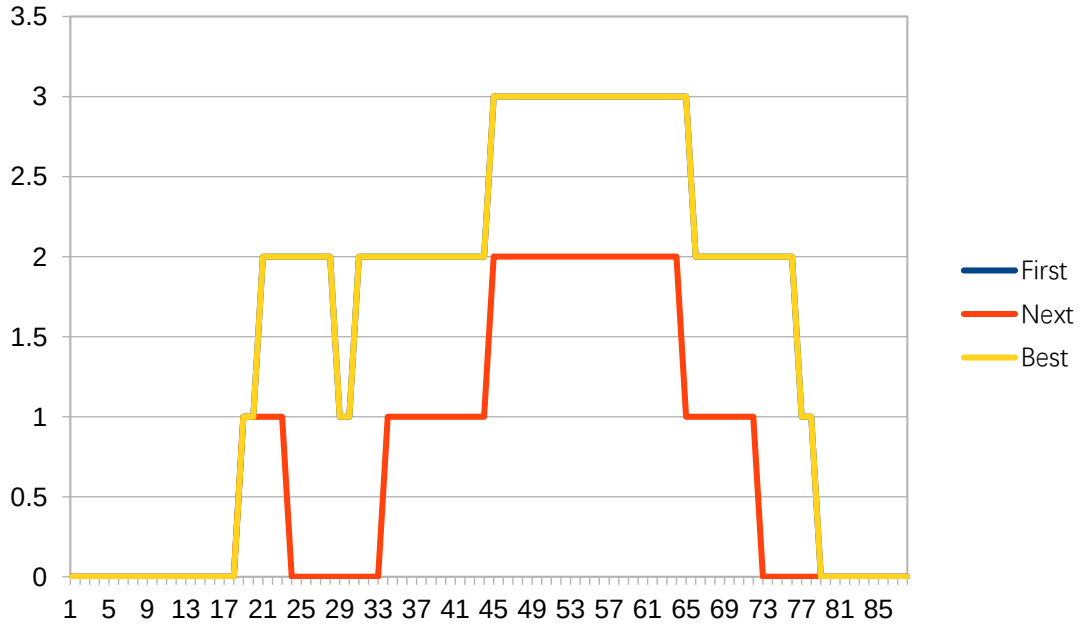
Next Fit



Best Fit



Fragment Analysis



These are the number of fragments which are smaller than 5.

4. Conclusion

4.1 Conclusion

- As we can see from the diagram, Next Fit can make the number of small fragments relatively small under certain circumstances. Maybe in other circumstances, other algorithms will perform better. Next Fit also has its drawbacks, as it makes memory pile up at the end of the linked list, resulting in an uneven list.
- First Fit algorithm tends to take advantage of the free partition of the low address part of memory, thus preserving the large free area of the high address part, which creates conditions for allocating large memory space to the large jobs arriving later. But the low-address portion is constantly partitioned, leaving a lot of small, unused partitions that are difficult to use. Each lookup starts at the lower address portion, which undoubtedly increases the overhead of finding available free partitions.
- Best Fit finds the smallest free partition from all free zones that can meet the job requirements, thus minimizing fragmentation. But it will take more time.

4.2 Comments

- Through this experiment, I know more about what is dynamic partition. And I am more familiar with these three algorithms. And the first fit is better than next

fit because sometimes there are more partitions gather in the end of memory while using next fit.

- The data structure we use as memory is very important, the linked list is much better than other data structure. Every partition in it is connected by pointers.
- I use address to indicate where the process is stored. So it's very important for me to change the pointer correctly. I debug for more 4 hours to find why the pointer is wrong.
- This experiment is very interesting. I regard every request as a memory process, regard linked list as memory, put partition into it to simulate the environment.

4.3 Suggestions

I also think that achieving a memory management by pages is also challenged and interesting. Maybe we can do it!