Lab Report of Project 7
Contiguous Memory Allocation

Name: Li Dongyue   Student Number: 516030910502
Computer System Engineering (Undergraduate), Fall 2018

December 29, 2018

1 Introduction

This project involves managing a contiguous region of memory of size *MAX* where addresses may range from 0 . . . *MAX*-1. Our program must respond to several different requests:

1. Request for a contiguous block of memory `(RQ)`
2. Release of a contiguous block of memory `(RL)`
3. Compact unused holes of memory into one single block `(C)`
4. Report the regions of free and allocated memory `(STAT)`
5. Exit the contiguous memory allocator `(X)`

2 Implementation

In this chapter, I will explain how I implement the contiguous memory allocation. First, I will explain how I simulate the memory allocation which does not really allocate memory and just keep the track of allocated memory information. Second, I will explain how I implement each function of the allocator.

2.1 Memory Information Representation

To set up, I use two lists `simulatedMemory` and `memoryName` and a integer number `fragNum` for memory information representation. In this way, memory are taken as fragments composed of unallocated memory(empty fragment) and process memories(process fragments). The list `simulatedMemory` contains the information of each fragments(empty fragments and process fragments). The list `memoryName` contains the information of name of each fragments. For empty fragment, the name is "empty". For process fragment, the name is the process name. The `fragNum` is for recording how many fragments in the memory.

```
int simulatedMemory[100];
char * memoryName[100];
int fragNum; // memory fragment number
char emptyName[10] = "Unused";
```

allocator.c

Therefore, I just keep the records of memory status information instead of really allocating information in memory. Given these information, I am able to do the allocation operation.

2.2 Memory Allocation Operation

In my program, the allocator repeatedly take user's input and perform different task according to that input. I set a variable `status` to detect the status of the allocator. If the user inputs `X`, which means he wants to exit, the `status` variable will turn 0 and the allocator will stop precessing.

```
int main(int argc, char *argv[])
{
    memSize = atoi(argv[1]);;

    for (int i=0; i<100; i++) simulatedMemory[i] = 0;
    fragNum = 1;
```

```c
    simulatedMemory[0] = memSize;
    memoryName[0] = emptyName;

    int status = 1;

  /*Excute input lines*/
  while (status)
  {
        char* cmd = malloc(sizeof(char)*30);
        printf("allocator>");
        fgets(cmd, 30, stdin);

        char * pch;
        pch = strtok (cmd," ");

    if(cmd[0] == 'C')
    {
      compactMemory();
    }
    else if (cmd[1] == 'Q')
    {
            char* name;
            int size;
            char* flag;

            name = strtok(NULL, " ");
            size = atoi(strtok(NULL, " "));
            flag = strtok(NULL, " ");

      allocateMemory(name, size , flag);
    }
    else if (cmd[1] == 'L')
    {
      char* name;
            name = strtok(NULL, " ");
            releaseMemory(name);
        }
    else if(cmd[0] == 'S')
    {
        reportMemory();
    }
        else if(cmd[0] == 'X')
        {
            status = 0;
        }
  }
    return 0;
}
```

allocator.c

2.2.1 Allocate Memory

We have three ways to allocate contiguous memory: **First Fit**, **Best Fit**, **Worst Fit**.

First Fit

For first fit, I iterate through the list `simulatedMemory` and find the first empty one that can contain the process memory and update the information accordingly(update the lists `simulatedMemory` and `memoryName` and make `fragNum` plus one).

```c
if (flag[0] == 'F')
    {
        int i;
        for (i=0; i<fragNum; i++)
        {
            if (!strcmp(memoryName[i], "Unused"))
            {
                if (size < simulatedMemory[i])
                {
                    for (int j=fragNum; j>i; j--)
                    {
                        memoryName[j] = memoryName[j-1];
                        simulatedMemory[j] = simulatedMemory[j-1];
                    }
                    simulatedMemory[i+1] = simulatedMemory[i] - size;
```

```
                    simulatedMemory[i] = size;

                    char * tmpName = malloc(sizeof(char)*strlen(name));
                    strcpy(tmpName, name);
                    memoryName[i] = tmpName;
                    fragNum = fragNum + 1;
                    break;
                }
                else if (size == simulatedMemory[i])
                {
                    memoryName[i] = name;
                    break;
                }
            }
        }
        if (i == fragNum) printf("Memory Request Denied.\n");
        return;
    }
```

allocator.c

Best Fit

For best fit, I need to iterate through the whole list `simulatedMemory` and keep the track of the minimum sized empty fragment that can contain the process information. Then I allocate the process to the minimum sized empty fragment and update the information accordingly(update the lists `simulatedMemory` and `memoryName` and make `fragNum` plus one).

```
if (flag[0] == 'B')
    {
        int tmpIndex = -1;
        int tmpSize = memSize+1;
        for (int i=0; i<fragNum; i++)
        {
            if (!strcmp(memoryName[i], "Unused"))
            {
                if (simulatedMemory[i]>=size && simulatedMemory[i]<tmpSize)
                {
                    tmpIndex = i;
                    tmpSize = simulatedMemory[i];
                }
            }
        }
        if (tmpIndex == -1)
        {printf("Memory Request Denied.\n"); return;}
        if (size < tmpSize)
        {
            for (int j=fragNum; j>tmpIndex; j--)
            {
                memoryName[j] = memoryName[j-1];
                simulatedMemory[j] = simulatedMemory[j-1];
            }
            simulatedMemory[tmpIndex+1] = simulatedMemory[tmpIndex] - size;
            simulatedMemory[tmpIndex] = size;

            char * tmpName = malloc(sizeof(char)*10);
            strcpy(tmpName, name);
            memoryName[tmpIndex] = tmpName;
            fragNum = fragNum + 1;
        }
        else if (size == tmpSize)
        {
            memoryName[tmpIndex] = name;
        }
    }
```

allocator.c

Worst Fit

Similar to best fit, in this time, I need to iterate through the whole list `simulatedMemory` and keep the track of the maximum sized empty fragment that can contain the process information. Then I allocate the process to the maximum sized empty fragment and update the information accordingly(update the lists `simulatedMemory` and `memoryName` and make `fragNum` plus one).

```
if (flag[0] == 'W')
    {
```

```c
        int tmpIndex = -1;
        int tmpSize = size-1;
        for (int i=0; i<fragNum; i++)
        {
            if (!strcmp(memoryName[i], "Unused"))
            {
                if (simulatedMemory[i]>tmpSize)
                {
                    tmpIndex = i;
                    tmpSize = simulatedMemory[i];
                }
            }
        }
        if (tmpIndex == -1)
        {printf("Memory Request Denied.\n"); return;}
        if (size < tmpSize)
        {
            for (int j=fragNum; j>tmpIndex; j--)
            {
                memoryName[j] = memoryName[j-1];
                simulatedMemory[j] = simulatedMemory[j-1];
            }
            simulatedMemory[tmpIndex+1] = simulatedMemory[tmpIndex] - size;
            simulatedMemory[tmpIndex] = size;

            char * tmpName = malloc(sizeof(char)*10);
            strcpy(tmpName, name);
            memoryName[tmpIndex] = tmpName;
            fragNum = fragNum + 1;
        }
        else if (size == tmpSize)
        {
            memoryName[tmpIndex] = name;
        }
    }
```

allocator.c

2.2.2 Release Memory

For releasing memory, I need to delete the information of corresponding process in the lists `memoryName` and `simulatedMemory` and update the information accordingly(update the lists `simulatedMemory` and `memoryName` and make `fragNum` minus one).

After deleting, I also need to check whether the fragment next to the deleted process fragment is empty. If yes, I need to compact them together and update the information.

```c
void releaseMemory(char* name)
{
    for (int i=0; i<fragNum; i++)
    {
        if (memoryName[i][1] == name[1])
        {
            memoryName[i] = emptyName;
            if (i>0 && !strcmp(memoryName[i-1], "Unused"))
            {
                simulatedMemory[i-1] += simulatedMemory[i];
                for (int j=i; j<fragNum-1; j++)
                {
                    simulatedMemory[j] = simulatedMemory[j+1];
                    memoryName[j] = memoryName[j+1];
                }
                fragNum--;
                i--;
            }
            if (i<fragNum-1 && !strcmp(memoryName[i+1], "Unused"))
            {
                simulatedMemory[i] += simulatedMemory[i+1];
                for (int j=i+1; j<fragNum-1; j++)
                {
                    simulatedMemory[j] = simulatedMemory[j+1];
                    memoryName[j] = memoryName[j+1];
                }
                fragNum--;
            }
```

```
                break;
            }
        }
}
```

allocator.c

2.2.3 Compact Memory

For compacting memory, we need to compact all empty fragment in one empty fragment. I do this by two steps:

- First, I move each process fragment to the front of the simulated memory and next to another process fragment. In this step, I move all the empty fragments to the tail of the simulated memory, which makes it easy for me to compact memory.

```
for (int i=0; i<fragNum; i++)
    {
        if (strcmp(memoryName[i], "Unused"))
        {
            int index = i;
            while(index>0 && !strcmp(memoryName[index-1], "Unused"))
            {
                memoryName[index-1] = memoryName[index];
                int tmp = simulatedMemory[index-1];
                simulatedMemory[index-1] = simulatedMemory[index];

                memoryName[index] = emptyName;
                simulatedMemory[index] = tmp;
                index--;
            }
        }
    }
```

allocator.c

- Second, I compact all the empty memories that have been stacked at the tail of simulated memory to one empty memory, adding their size together and creating one big empty fragment.

```
for (int i=0; i<fragNum; i++)
    {
        if (!strcmp(memoryName[i], "Unused"))
        {
            int index = i;
            int emptyMemSize = 0;
            while (index < fragNum)
            {
                emptyMemSize += simulatedMemory[index++];
            }

            simulatedMemory[i] = emptyMemSize;
            fragNum = i+1;
            break;
        }
    }
```

allocator.c

2.2.4 Report Memory Status

If the user type input ``STAT'', the program will print out the memory status composed of allocated memory range and memory fragment name.

In my program, to print out information, I need to iterate through the lists `simulatedMemory` and `memoryName` and print out each fragment information accordingly. Then I allocate the process to the minimize sized empty fragment and update the information accordingly(update the lists `simulatedMemory` and `memoryName` and make `fragNum` plus one).

```

void reportMemory()
{
    int curMem = 0;
    for (int i=0; i<fragNum; i++)
```

```
    {
        if (!strcmp(memoryName[i], "Unused"))
        {
            printf("[%d:%d] Unused\n", curMem, curMem+simulatedMemory[i]-1);
        }
        else
        {
            printf("[%d:%d] Process %s\n", curMem, curMem+simulatedMemory[i]-1, memoryName[i]);
        }
        curMem = curMem + simulatedMemory[i];
    }
}
```

allocator.c

3 How to run this program

To run this program, please compile it with `gcc` and run it with `./allocator MAX_MEMSIZE` where `MAX_MEMSIZE` is user assigned memory size.