MEMORY ALLOCATION STRATEGIES REPORT

Discussion & Analysis

Mohammed Abdullah H Almahdawi
RMIT | Operating System Principles

Table of Contents

Introduction

In this report, three memory allocation strategies will be discussed and analysed. These are First Fit, Best Fit, and Worst Fit. In addition, a software has been developed to showcase the results of implementing the mentioned strategies. This software will give a very intensive insights about how each of the allocation strategies work. Also, the performance of each of the mentioned strategies will be measure based on the results of executing the software against a strategy.

Optimization Objective

The optimization objectives of using these memory allocation strategies are many. First, the operating system has the ability in managing the high number of processes and their memory. Second, the use of one strategy over the other will determine how fast the operating system would perform when allocating or searching in the free memory list for a memory block. In addition, the efficiency of allocating processes to the memory is an important aspect that the operating system should consider when deciding between strategies. This report will extensively study each strategy and conclude the most optimized strategy based on the results and the experimental evidence gather by the software.

Software Overview

The way this software works is to have two lists, which are allocated memory list and freed memory list. Once the software receive an input file, it allocates memory blocks for 1000 lines from the file using the function sbrk(). This function will help allocating memory in the heap by giving the size, which is the line's size. Once the allocation of each memory block, the software keeps track of the starting address and the size of that allocated block. After allocating 1000 memory blocks, the software will randomly delete 500 memory blocks and keep track of their starting address and size in the freed memory list. Also, the consecutive memory block in the heap will be merged and presented in the free memory list as one memory block. Then the software repeats the same actions on the rest of the input file until no more line is being read.

Input/Output Overview

The input that has been used on testing is from dominictarr GitHub account [1]. Three files contain thousands of names are inputted to the software. The first file is first-names.txt, and this file contains 4945 names. The second file that has been used for testing is middle-names.txt, and it contains 3897 names. The last file which is the largest file is names.txt and it contains almost 22000 names. The software will output the total number of bytes has been allocated to memory using sbrk(). Also, the output file will have number of nodes in the freed memory list, the number of allocated names using sbrk(), the address and size of each node in the freed memory list, and the address, size, and content of each node in the allocated memory list.

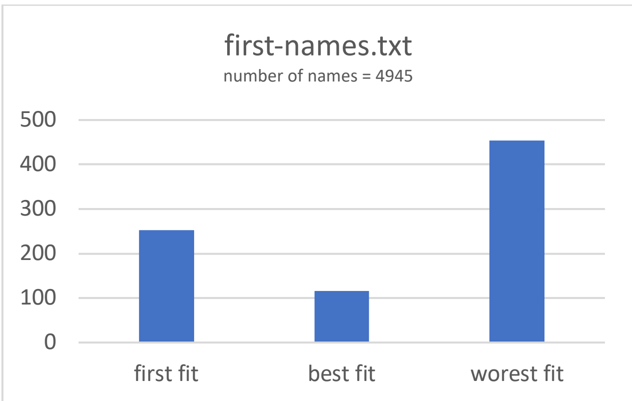

Figure 1.1

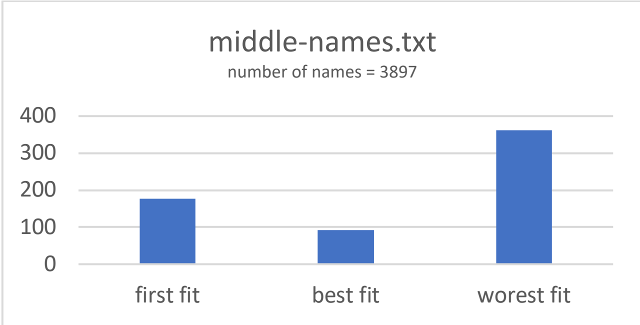

Figure 1.2

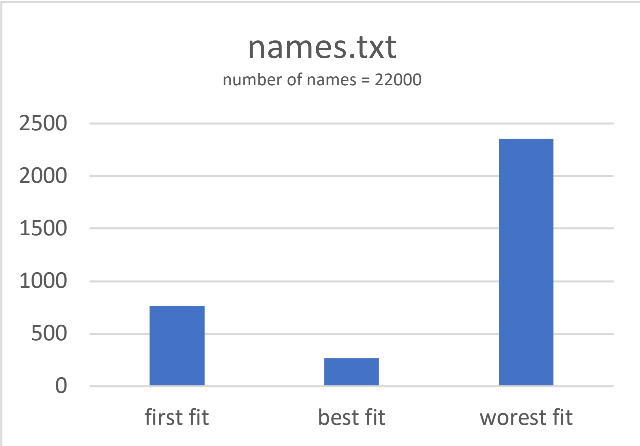

Figure 1.3

Analysis

To get an accurate analysis, each input file has been tested on each strategy five times. The results from all the five outputs have been averaged to get a more accurate result. Also, the consistency on the results will increase for many reasons. One of which is that we randomly delete 500 memory blocks and since random deletion may cause consecutive memory blocks to be compacted. Thus, this merge on the memory blocks will lead to different result each time the software runs.

Figure 1.1, 1.2 and 1.3 are representing the average number of nodes in the freed memory list for each file and each strategy.

Looking at the histograms in all the Figures 1.1 – 1.3, we see a trend in these histograms that are similar to each other. Worst fit startegy is having the highest number of nodes in the free memory list on all of the files. However, the more the file input has many lines, the more the number of nodes in the freed list gets bigger. This means that having an input file, that has 4000 strings, will result in 350 useless memory blocks in the freed list. If the input file has five times higher the number of strings in the previous example, then the useless memory blocks will exceed 2400 blocks. Therefore, worst fit by far is the worst memory allocation strategy among the three mentioned for many reasons. We already touched on the useless holes, which are spread out memory blocks in the heap that have a very small size; these blocks are useless because they cannot be merged with other holes since mostly they are not consecutive in memory, and most of the time these holes are smaller than the size of regular data to be allocated. Another reason that worst fit is the worst strategy is that worst fit will search for the biggest memory block in the freed memory list, and this will take higher time for the operating system to search all the nodes in the list to get the biggest memory block. Therefore, the bigger the list is the higher time it will take the operating system to get the biggest memory block. As a result, worst fit strategy is not comparable to the other strategies in terms of its complexity time and the number of memory holes making it the worst.

Comparing worst fit to first fit and best fit strategies, first fit strategies will only look the first memory block that will satisfy the size it is looking for; best fit strategy will look for the closest possible memory block size it is looking for. However, looking at the figures 1.1-1.3, we notice that first fit always has double the number of useless memory holes than best fit. This does not mean that first fit is the worst strategy, but it shows that it wastes many memory blocks. An advantage of first fit will be that its time complexity is most probably lower than best fit and worst fit strategies since it searches for the first memory block that satisfies the needed size. Although first fit has massive memory wastages, worst fit is not comparable to first fit because its memory wastages are more than five times first fit's memory wastages number.

On the other hands, based on the figures 1.1-1.3 and best fit definition, best fit has the minimum number of memory holes in the heap compared to the maximum number of holes, worst fit, and it is also less than half of the memory wastages when using first fit. Although best fit strategy might have a slightly lower memory holes than first fit, the time complexity is very high in best fit strategy. This is the biggest disadvantage of best fit since it needs to search all the nodes in the freed memory list to fine the best suitable memory block.

Conclusion

In conclusion, while worst fit has the maximum number of wastages, best fit has the minimum number of wastages. Also, first fit illustrated some flaws and number of memory holes that are more than doubled best fit's memory holes. However, first fit has the lowest number of searches among all the three strategies and this makes it the best performer strategy for the operating system. Also, it has low rate of memory wastages compered to worst fit. Therefore, first fit is the best strategy compared to best fit and worst fit.

Bibliography

[1] https://github.com/dominictarr/random-name

app.cpp

```
#include "alloc_block.h"
#include "max_block.h"
#include <list>
#include <string>
#include <string.h>
#include <vector>
#include <fstream>
#include <sstream>
#include <iostream>
#include <ctime>
#include <iterator>
#include <stdlib.h>
#include <sys/queue.h>
#include <unistd.h>
using namespace std;

// args options
enum alloc_strategies
{
    FF, //  first fit
    BF, //  best fit
    WF, //  worest fit
    INVALID
};

// retreiving args
alloc_strategies op_return(string str)
{
    if (str == "-ff")
        return FF;
    else if (str == "-bf")
        return BF;
    else if (str == "-wf")
        return WF;
    return INVALID;
}

// file reading
void readFile(vector<string> *arr, string filename)
{

    ofstream f;
    string line;

    int i = 0;
    // given file in canvas
    std::ifstream infile(filename);

    if (infile.is_open())
    {
```

```
        // loop through the file and get every line
        while (getline(infile, line))
        {
            std::istringstream iss(line);
            string word;
            // pushing the numbers to the above vars
            // as the arrangment explaind in the assignment specs
            if (!(iss >> word))
            {
                break;
            }
            arr->push_back(word);
            i++;
        }
    }
    f.close();
}

bool comp(const alloc_block *a, const alloc_block *b)
{
    return (void *)a->bword < (void *)b->bword;
}

void writeFile(int totalAllocationInByte, int totalAllocationTimes,
list<alloc_block *> allocMBList, list<alloc_block *> freedMBList, string filename)
{

    std::ofstream f;
    f.open(filename);

    // file writing
    f << "***[Memory Allocated Size: " << totalAllocationInByte << " Bytes]***\n";
    f << "***[Memory Allocations: " << totalAllocationTimes << "
Allocations]***\n";
    f << "***[Nodes in Free List: " << freedMBList.size() << " Nodes]***\n\n";

    // standerd output
    cout << "***[Memory Allocated Size: " << totalAllocationInByte << "
Bytes]***\n";
    cout << "***[Memory Allocations: " << totalAllocationTimes << "
Allocations]***\n";
    cout << "***[Nodes in Free List: " << freedMBList.size() << " Nodes]***\n\n";

    // file writing
    f << "freedMBList\n";
    f << "Address\t\t\tNode Size\n";

    // standerd output
    cout << "freedMBList\n";
    cout << "Address\t\t\tNode Size\n";
```

```
    for (list<alloc_block *>::iterator node = freedMBList.begin(); node !=
freedMBList.end(); ++node)
    {
        // file writing
        f << (void *)((*node)->bword) << "\t\t\t" << (*node)->bsize << "\n";
        // standerd output
        cout << (void *)((*node)->bword) << "\t\t\t" << (*node)->bsize << "\n";
    }

    // file writing
    f << "\n\n";
    f << "allocMBList\n";
    f << "Address\t\t\tNode Size\t\tContent\n";

    // standerd output
    cout << "\n\n";
    cout << "allocMBList\n";
    cout << "Address\t\t\tNode Size\t\tContent\n";

    for (list<alloc_block *>::iterator node = allocMBList.begin(); node !=
allocMBList.end(); ++node)
    {
        // file writing
        f << (void *)((*node)->bword) << "\t\t\t" << (*node)->bsize << "\t\t\t";
        // standerd output
        cout << (void *)((*node)->bword) << "\t\t\t" << (*node)->bsize << "\t\t\t";

        char buffer[(*node)->bsize + 1];
        snprintf(buffer, sizeof(buffer),
                 "%s", (*node)->bword);

        // file writing
        f << buffer << "\n";
        // standerd output
        cout << buffer << "\n";
    }

    f.close();
}

void allocateMemory(string &name, list<alloc_block *> *allocMBList, int
*totalAllocationInByte, int *totalAllocationinTimes)
{
    // set the size
    size_t bsize = name.length();

    // allocate a new memory block with the name size
    void *bword;
    bword = sbrk(bsize);

    //copy from the name from the file to the specific memory block
```

```
    strncpy((char *)bword, name.c_str(), bsize);

    // push the newly allocated memory and size to the list
    alloc_block *mB = (alloc_block *)malloc(sizeof(struct alloc_block));
    mB->bsize = (int)bsize;
    mB->bword = (char *)bword;
    allocMBList->push_back(mB);

    // add size to totalAllocation for counting the bytes in the heap
    *totalAllocationInByte = *totalAllocationInByte + bsize;
    *totalAllocationinTimes = *totalAllocationinTimes + 1;
}

void compactMemory(list<alloc_block *> *freedMBList)
{
    // sort the list
    freedMBList->sort(comp);

    list<alloc_block *>::iterator it1 = freedMBList->begin(), it2 = ++freedMBList-
>begin();

    // iterate the sorted list
    while (it2 != freedMBList->end())
    {

        // checking consecitive free memory block
        if ((*it1)->bword + (*it1)->bsize == (*it2)->bword)
        {
            // merge blocks
            (*it1)->bsize = (*it1)->bsize + (*it2)->bsize;

            // erease the merged node
            freedMBList->erase(it2++);
        }
        else
            ++it1, ++it2;
    }
}

void deallocateMemory(int length, int maxLength, int counter, list<alloc_block *>
*allocMBList, list<alloc_block *> *freedMBList)
{
    int remaining_length = length - counter;

    // length check for remaining names and should be limited to a max of maxLength
names to be freed
    if (remaining_length > maxLength)
        remaining_length = maxLength;

    // free maxLength memory each iterationLength iteration
    for (int j = 0; j < remaining_length; j++)
```

```
    {

        // create iterator
        list<alloc_block *>::iterator it = allocMBList->begin();

        // random num
        std::srand(std::time(nullptr));
        int num = std::rand() % allocMBList->size();

        // advance iterator by num (get num-th node)
        advance(it, num);

        // add the removed to the freed list
        alloc_block *mB = (alloc_block *)malloc(sizeof(struct alloc_block));
        mB->bsize = (int)(*it)->bsize;
        mB->bword = (char *)(*it)->bword;
        freedMBList->push_back(mB);

        // remove from alloc list
        allocMBList->erase(it);
    }

    // compact consective memory in free list
    compactMemory(freedMBList);
}

void splitMemory(list<alloc_block *>::iterator it, string &name, list<alloc_block
*> *allocMBList, list<alloc_block *> *freedMBList, bool *flagMB)
{
    // set the size
    size_t bsize = name.length();

    //copy from the name from the file to the specific memory block
    strncpy((char *)(*it)->bword, name.c_str(), bsize);

    // get the remaining size from that block
    int remainingSize = (*it)->bsize - (int)bsize;

    // store the new name in allocatMBList
    alloc_block *mB = (alloc_block *)malloc(sizeof(struct alloc_block));
    mB->bsize = (int)bsize;
    mB->bword = (char *)(*it)->bword;
    allocMBList->push_back(mB);

    // if remaining memory block store it back to freedMBList
    if (remainingSize > 0)
    {
        // char *word = static_cast<char *>(());
        alloc_block *mBF = (alloc_block *)malloc(sizeof(struct alloc_block));
        mBF->bsize = remainingSize;
        mBF->bword = (char *)(*it)->bword + bsize;
```

```
        freedMBList->push_back(mBF);
    }

    // delete the old memory block from the list
    freedMBList->erase(it);

    // compact consective memory in free list
    compactMemory(freedMBList);

    // set the flag so it's not allocated again in the next if block
    *flagMB = true;
}

void firstFit(string &name, list<alloc_block *> *allocMBList, list<alloc_block *>
*freedMBList, bool *flagMB)
{
    // finding first fit
    for (list<alloc_block *>::iterator it = freedMBList->begin(); it !=
freedMBList->end(); ++it)
    {
        // checking the size of both the name from the file and the allocated
memory size in freedMBList
        if (name.length() <= (unsigned)(*it)->bsize)
        {
            splitMemory(it, name, allocMBList, freedMBList, flagMB);
            break;
        }
    }
}

void worstFit(string &name, list<alloc_block *> *allocMBList, list<alloc_block *>
*freedMBList, bool *flagMB)
{
    // max name size
    int max = 0;
    // temp iterator
    list<alloc_block *>::iterator temp;

    // iterating through freedMBList to get the max value and store that MB
    for (list<alloc_block *>::iterator it = freedMBList->begin(); it !=
freedMBList->end(); ++it)
    {
        if ((*it)->bsize > max)
        {
            max = (int)(*it)->bsize;
            temp = it;
        }
    }

    // checking the size of both the name from the file and the allocated memory
size in freedMBList
```

```
    if (name.length() <= (unsigned)max && *temp != NULL)
    {
        splitMemory(temp, name, allocMBList, freedMBList, flagMB);
    }
}

void bestFit(string &name, list<alloc_block *> *allocMBList, list<alloc_block *>
*freedMBList, bool *flagMB)
{
    // flag
    int i = -1;
    // temp iterator
    list<alloc_block *>::iterator temp;

    // iterating through freedMBList to get the exact match andd replace it with
name
    for (list<alloc_block *>::iterator it = freedMBList->begin(); it !=
freedMBList->end(); ++it)
    {
        // checking the size of both the name from the file and the allocated
memory size in freedMBList
        if (name.length() <= (unsigned)(*it)->bsize)
        {
            // first found satisfying size from list
            if (i == -1)
            {
                i = (*it)->bsize;
                temp = it;
            }
            // ideal size
            else if ((*temp)->bsize > (*it)->bsize)
            {
                i = (*it)->bsize;
                temp = it;
            }
        }
    }
    if (i != -1)
    {
        splitMemory(temp, name, allocMBList, freedMBList, flagMB);
    }
}

void run(vector<string> *arr, alloc_strategies strategy, string outFile)
{
    // lists
    list<alloc_block *> allocMBList, freedMBList;
    // length of file received
    int length = arr->size();
    // loop counter
    int i = 0;
```

```
    // free limit => 500
    int deletionLength = 500;
    // iteration length => 1000
    int insertionLength = 1000;
    // total allocated bytes
    int totalAllocatedMemoryInByte = 0, totalAllocatedMemoryTime = 0;

    // main loop for all name list size
    while (i < length)
    {
        // each 1000 allocations in allocMBList
        if (i % insertionLength == 0 && i > 0)
        {
            deallocateMemory(length, deletionLength, i, &allocMBList,
&freedMBList);
        }

        // flag if memory block is found in freedMBList
        bool foundMB = false;

        // checking if not empty then we try use memory block
        if (freedMBList.size() > 0)
        {
            switch (strategy)
            {
            case FF:
                firstFit(arr->at(i), &allocMBList, &freedMBList, &foundMB);
                break;
            case BF:
                bestFit(arr->at(i), &allocMBList, &freedMBList, &foundMB);
                break;
            case WF:
                worstFit(arr->at(i), &allocMBList, &freedMBList, &foundMB);
                break;
            case INVALID:
                break;
            }
        }

        // if name didn't find a fit in freedMBList
        if (!foundMB)
        {
            allocateMemory(arr->at(i), &allocMBList, &totalAllocatedMemoryInByte,
&totalAllocatedMemoryTime);
        }
        i++;
    }
    writeFile(totalAllocatedMemoryInByte, totalAllocatedMemoryTime, allocMBList,
freedMBList, outFile);
}
```

```
int main(int argc, char *argv[])
{

    // vector for file reading and arguments
    vector<string> content;
    vector<string> arguments(argv + 1, argv + argc);

    // // help menu
    string help_menu = "Choose one of the following strategies as an argument:\n\t-
ff \t<source file> <destination file>\tFirst-Fit\n\t-bf \t<source file>
<destination file>\tBest-Fit\n\t-wf \t<source file> <destination file>\tWorst-Fit";

    // check if argument are given then enter the switch statement
    if (arguments.size() == 3)
    {
        readFile(&content, arguments[1]);

        switch (op_return(arguments[0]))
        {
        case FF:
            cout << "*********[First Fit]*********\n"
                 << endl;
            run(&content, FF, arguments[2]);
            break;
        case BF:
            cout << "*********[Best Fit]*********\n"
                 << endl;
            run(&content, BF, arguments[2]);
            break;
        case WF:
            cout << "*********[Worest Fit]*********\n"
                 << endl;
            run(&content, WF, arguments[2]);
            break;
        case INVALID:
            cout << help_menu << endl;
            break;
        default:
            break;
        }
    }
    else
    {
        cout << help_menu << endl;
    }

    return 0;
}
```

alloc_block.h

```
#ifndef ALLOCATION_STRUCT
#define ALLOCATION_STRUCT

struct alloc_block
{
    int bsize;
    char *bword;
    bool operator==(const alloc_block &ab)
    {
        return bsize == ab.bsize &&
               bword == ab.bword;
    }
};

#endif
```

max_block.h

```
#ifndef MAX_BLOCK_STRUCT
#define MAX_BLOCK_STRUCT

#include "alloc_block.h"

struct max_block
{
    int length;
    alloc_block *block;
};

#endif
```

Makefile

```
all:
    g++ -std=c++11 -Wall  -o app app.cpp
```