
# COSC1114 Operating System Principles

## Assignment 2

By

Arone Susau s3728977

# Table of Contents

# Introduction

This report will cover the experiment and analysis of the first and middle name data sets to conclude which allocation algorithm, First Pick, Best Pick and Worst Pick is the best allocation strategy.

The first pick algorithm will pick the first memory block available that contains the required size space to allocate the required value. The best pick algorithm will select a memory block that contains a size that is exactly the size required. The worst pick allocation strategy will select the memory block with the largest size for allocation. Each allocation strategy will split any memory block selected that's size is greater than the size of the current value.

# Experiment

Each dataset will have their names read into memory and stored in a list. Then the allocation strategy will execute to determine if a deallocated memory block fits the requirement of the strategy, otherwise a new memory block will be created.

This process will repeat until the names list is empty, and then the contents of the allocated and deallocated memory block lists will be stored in a collection of CSV files. The "*combined.cvs*" file will contain an aggregate of all the data from all the name files and allocation strategy types, although individual execution files will remain under the results directory if individual analysis or cross referencing is required.

# Results

Specific results and the files can be found under the results directory in the project folder.

## First Fit

| METRIC | RESULT |
|---|---|
| FILE | first-names |
| TOTAL MEMORY USAGE | 24155 |
| AVERAGE FREEMB SIZE | 5.25 |
| AVERAGE ALLOCMB SIZE | 8.05 |

| METRIC | RESULT |
|---|---|
| FILE | middle-names |
| TOTAL MEMORY USAGE | 19179 |
| AVERAGE FREEMB SIZE | 5.73 |
| AVERAGE ALLOCMB SIZE | 7.84 |

## Best Fit

| METRIC | RESULT |
|---|---|
| FILE | first-names |
| TOTAL MEMORY USAGE | 23970 |
| AVERAGE FREEMB SIZE | 6.45 |
| AVERAGE ALLOCMB SIZE | 8.05 |

| METRIC | RESULT |
|---|---|
| FILE | middle-names |
| TOTAL MEMORY USAGE | 18993 |
| AVERAGE FREEMB SIZE | 6.80 |
| AVERAGE ALLOCMB SIZE | 7.84 |

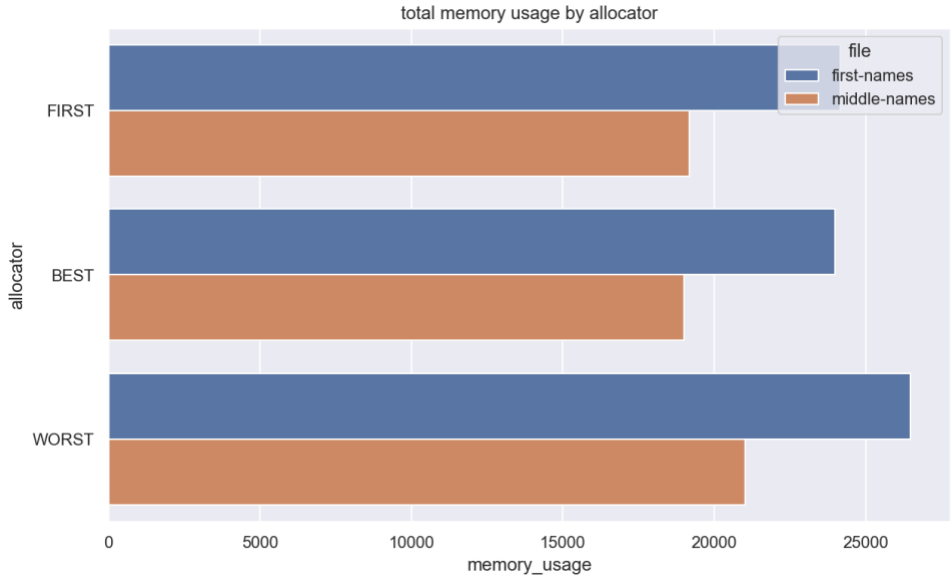
## Worst Fit

| METRIC | RESULT |
|---|---|
| FILE | first-names |
| TOTAL MEMORY USAGE | 26480 |
| AVERAGE FREEMB SIZE | 3.89 |
| AVERAGE ALLOCMB SIZE | 8.05 |

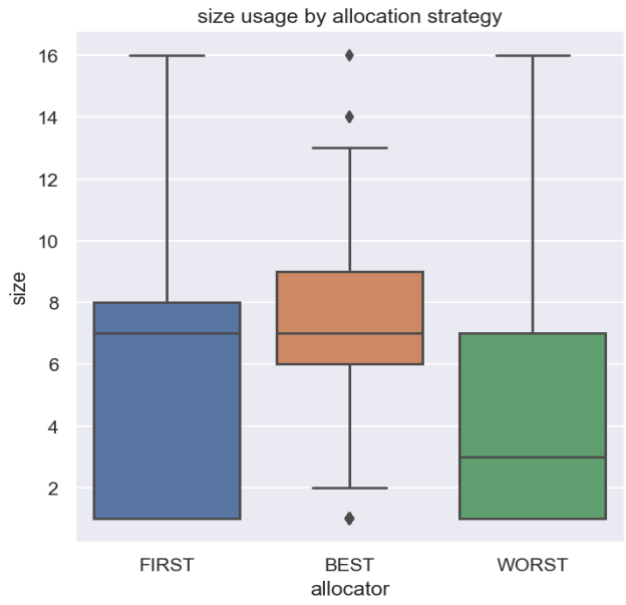
| METRIC | RESULT |
|---|---|
| FILE | middle-names |
| TOTAL MEMORY USAGE | 21020 |
| AVERAGE FREEMB SIZE | 4.26 |
| AVERAGE ALLOCMB SIZE | 7.84 |

# Analysis

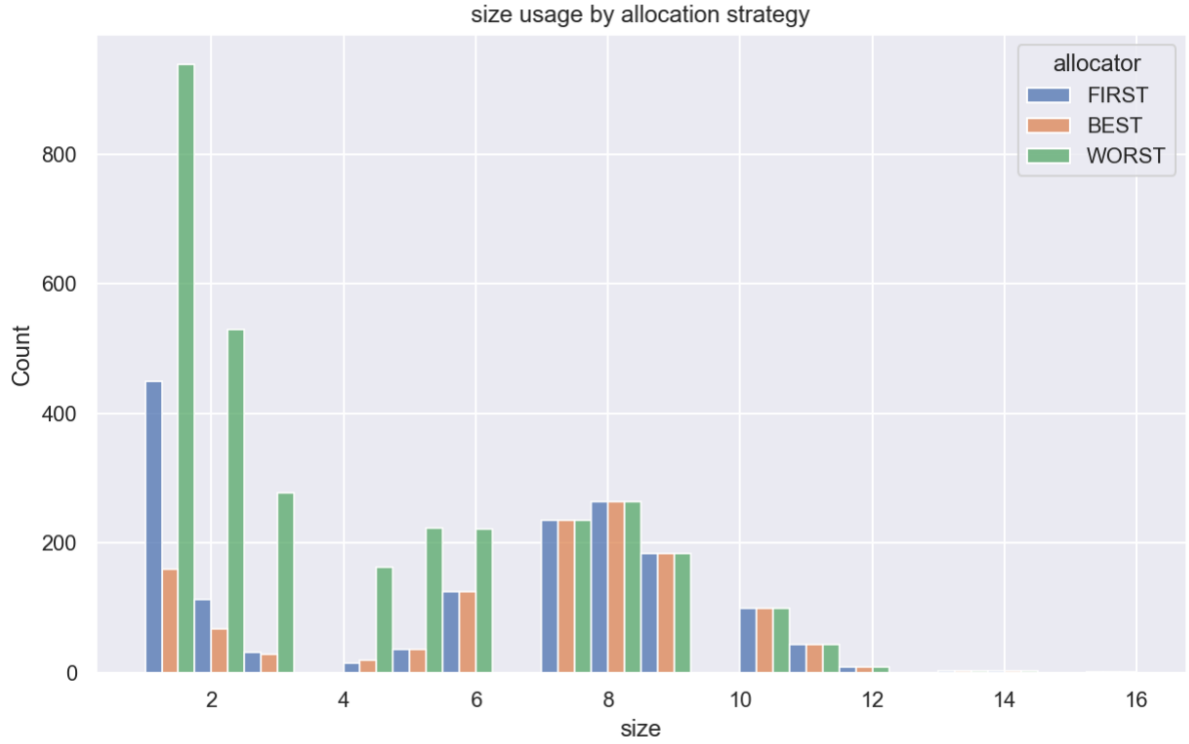
Viewing allocation strategies performance in terms of memory usage, we can see that both the first fit and best fit strategies had a similar result, followed by worst fit using the most memory. The results would appear to align with general intuition as the worst pick strategy will consume the largest memory blocks first, leaving smaller free memory blocks available as the program progresses. The increased number of smaller memory blocks will increase the likeliness that there will not be a larger memory block available if required, hence a new memory block will then be created.




The distribution of sizes of free memory blocks also provides some insight into the inner workings of the allocation strategy. As seen in the graph below, we can confirm our prior expectation that worst fit does indeed create a smaller average of free memory block sizes, sitting at 3.89, compared to first and best fit at 5.74 and 6.45 respectively.



It would also appear that first and worst fit have a similar range of free memory block sizes, with values within one standard deviation between 2-8. Best fit is the clear separator in the group with an average size of 6.8 and values within one standard deviation being between 6-9.




The worst fit algorithm also creates a significantly larger number of free memory blocks, particularly around the smaller size. Followed by first fit then best fit in terms of number of memory blocks.

# Conclusion

Best fit is undeniably the best allocation strategy out of the trio as it uses the least amount of memory, as well as creates the least amount of free memory blocks. Of those memory blocks it does create, they are on average between 6-9 in size, which for our current use case of names is preferable.

Both the first and worst fit use a larger total memory usage and create significantly smaller, somewhat useless, free memory blocks.

# Bibliography

- No materials outside of the course work were used to write this report.

# Code

```
// * * * * * * * * * * * * * * * *
// 2020 Semester 2 - Assignemnt 2
// Author: Arone Susau
// Course: COSC1114
// Student#: s3728977
// * * * * * * * * * * * * * * * *

#ifndef FILE_IO
#define FILE_IO

#include <unistd.h>
#include <cstdlib>
#include <string>
#include <list>
#include <fstream>
#include <iostream>
#include <cstring>
#include <sstream>

#include "memory_block.h"
#include "types.h"

using namespace std;

// Reads in contents of a file line by line,
// and assigns is value into the provided array.
//
// char* path   - Path to file
// names &names - List to store names
void readFile(char* path, names &names);

// Writes contents of AllocMBList to file.
//
// char* path          - Path to file
// char* type          - Allocator type
// memory_list &results - List of stored memory_blocks
void writeAllocMBFile(char* path, char* type, memory_list &results);
```

```
// Writes contents of freeMBList to file.
//
// char* path          - Path to file
// char* type          - Allocator type
// memory_list &results - List of stored memory_blocks
void writeFreeMBFile(char* path, char* type, memory_list &results);

// Writes total memory usage information to file.
//
// char* path - Path to file
// char* type - Allocator type
// int total  - Total memory usage
void writeHeaderFile(char* path, char* type, int total);

// Writes total memory usage information to file.
//
// char* path              - Path to file
// char* type              - Allocator type
// memory_list &freeMBList  - List of free memory_blocks
// memory_list &allocMBList - List of allocated memory_blocks
void writeCombinedFile(char* path, char* type, memory_list &freeMBList, memory_list &allocMBList);

#endif
// * * * * * * * * * * * * * * * *
// 2020 Semester 2 - Assignemnt 2
// Author: Arone Susau
// Course: COSC1114
// Student#: s3728977
// * * * * * * * * * * * * * * * *

#ifndef MAIN_PROGRAM
#define MAIN_PROGRAM

#include "./types.h"
#include "./strategies.h"
#include "./memory_block.h"
#include "./file_io.h"

#define PROGRAM_SUCCESS 0
```

```
#define RANDOM_MAX 500

// Executes the allocation algorithm
//
// names &names                  - List of names
// memory_list &allocMBList - List of currently allocated memory blocks
// memory_list &freeMBList  - List of currently deallocated memory blocks
void run(names &names, memory_list &allocMBList, memory_list &freeMBList);

// Randomly deallocates a number of memory blocks.
//
// memory_list &allocMBList - List of currently allocated memory blocks
// memory_list &freeMBList  - List of currently deallocated memory blocks
void removeRandomBlocks(memory_list &allocMBList, memory_list &freeMBList);

// Searches for and joins memory blocks with contiguous starting
// and ending positions.
//
// memory_list &freeMBList  - List of currently deallocated memory blocks
void joinContiguousBlocks(memory_list &freeMBList);

#endif
// * * * * * * * * * * * * * * * *
// 2020 Semester 2 - Assignemnt 2
// Author: Arone Susau
// Course: COSC1114
// Student#: s3728977
// * * * * * * * * * * * * * * * *

#ifndef MEMORY_BLOCK
#define MEMORY_BLOCK

#include <string>

using namespace std;

typedef struct _memory_block {
  int size;
  void* address;
} memory_block;
```

```
bool operator< (const memory_block& a, const memory_block& b);
bool operator> (const memory_block& a, const memory_block& b);
bool operator== (const memory_block& a, const memory_block& b);
#endif
// * * * * * * * * * * * * * * * *
// 2020 Semester 2 - Assignemnt 2
// Author: Arone Susau
// Course: COSC1114
// Student#: s3728977
// * * * * * * * * * * * * * * * *

#ifndef STRATEGIES
#define STRATEGIES

#include <string>
#include <list>
#include <stdlib.h>
#include <iostream>
#include <unistd.h>
#include <stdio.h>
#include <unistd.h>
#include <math.h>
#include <cstring>
#include <algorithm>

#include "memory_block.h"
#include "types.h"

#define FIRST "FIRST"
#define BEST "BEST"
#define WORST "WORST"

#define ALLOC_MAX 1000

class strategies {

  public:
    string type = "";
    int name_index = 0;
```

```
int total_memory = 0;
int max_names = 0;

// Determines the allocation strategy specified and executes it
//
// names &names                    - List of names
// memory_list &allocMBList - List of currently allocated memory blocks
// memory_list &freeMBList  - List of currently deallocated memory blocks
void allocate(names &names, memory_list &allocMBList, memory_list &freeMBList);

// Executes the First Pick allocation strategy
//
// names &names            - List of names
// memory_list &allocMBList - List of currently allocated memory blocks
// memory_list &freeMBList  - List of currently deallocated memory blocks
void first_pick(names &names, memory_list &allocMBList, memory_list &freeMBList);

// Executes the Best Pick allocation strategy
//
// names &names            - List of names
// memory_list &allocMBList - List of currently allocated memory blocks
// memory_list &freeMBList  - List of currently deallocated memory blocks
void best_pick(names &names, memory_list &allocMBList, memory_list &freeMBList);

// Executes the Worst Pick allocation strategy
//
// names &names            - List of names
// memory_list &allocMBList - List of currently allocated memory blocks
// memory_list &freeMBList  - List of currently deallocated memory blocks
void worst_pick(names &names, memory_list &allocMBList, memory_list &freeMBList);

// Creates a new memory block
//
// names &names            - List of names
// memory_list &allocMBList - List of currently allocated memory blocks
void newBlock(names &names, memory_list &allocMBList);

// Allocates a memory block with a specified size
//
// memory_iter it          - Iterator of position of memory block
```

```
    // names &names            - List of names
    // memory_list &allocMBList - List of currently allocated memory blocks
    // memory_list &freeMBList  - List of currently deallocated memory blocks
    // size                    - Size of memory block to be allocated
    void allocateBlock(memory_iter it, names &names, memory_list &allocMBList, memory_list &freeMBList, int
size);

    // Splits two memory blocks
    //
    // names &names            - List of names
    // memory_iter it          - Iterator of position of memory block
    // memory_list &freeMBList  - List of currently deallocated memory blocks
    void splitBlocks(memory_iter it, memory_list &freeMBList, int size);

    // Finds the largest sized deallocated memory block
    //
    // memory_list &freeMBList  - List of currently deallocated memory blocks
    int maxSize(memory_list &freeMBList);

};

#endif
// * * * * * * * * * * * * * * * *
// 2020 Semester 2 - Assignemnt 2
// Author: Arone Susau
// Course: COSC1114
// Student#: s3728977
// * * * * * * * * * * * * * * * *

#ifndef TYPES
#define TYPES

#include <list>

#include "memory_block.h"

#define MAX_CHAR 50
#define MAX_NAMES 4945

using names = char**;
```

```
using memory_list = list<memory_block*>;
using memory_iter = list<memory_block*>::iterator;

#endif
// * * * * * * * * * * * * * * * *
// 2020 Semester 2 - Assignemnt 2
// Author: Arone Susau
// Course: COSC1114
// Student#: s3728977
// * * * * * * * * * * * * * * * *

#include "./../headers/file_io.h"

void readFile(char* path, names &names) {
  char base[] ="./data/";
  char ext[] =".txt";

  stringstream dest;
  dest << base << path << ext;
  ifstream file(dest.str().c_str());

  string line;
  int count = 0;

  while (getline(file, line)) {
    names[count] = new char[MAX_CHAR];
    strcpy(names[count], line.c_str());
    count++;
  }

  file.close();
}

void writeAllocMBFile(char* path, char* type, memory_list &results) {
  char base[] ="./results/";
  char ext[] =".csv";

  ofstream file;
  stringstream dest;
  dest << base << path << "/alloc-" << type << ext;
```

```
  file.open(dest.str());

  file << "address,size,contents"  << endl;

  for (auto mb : results) {
    file << (intptr_t) mb->address << "," << mb->size << "," << (char *) mb->address << endl;
  }

  file.close();
}

void writeFreeMBFile(char* path, char* type, memory_list &results) {
  char base[] ="./results/";
  char ext[] =".csv";

  std::ofstream file;
  stringstream dest;
  dest << base << path << "/free-" << type << ext;
  file.open(dest.str());

  file << "address,size"  << endl;

  for (auto mb : results) {
    file << (intptr_t) mb->address << "," << mb->size << endl;
  }

  file.close();
}

void writeHeaderFile(char* path, char* type, int total) {
  char dest[] ="./results/totals.csv";
  std::ofstream file;
  file.open(dest, ios_base::app);
  file << path << "," << type << "," << total << endl;
  file.close();
}

void writeCombinedFile(char* path, char* type, memory_list &freeMBList, memory_list &allocMBList) {
  char base[] ="./results/";
```

```
  std::ofstream file;
  stringstream dest;
  dest << base << "combined.csv";
  file.open(dest.str(), ios_base::app);

  for (auto mb : allocMBList) {
    file << path << "," << type << "," << "ALLOC" << "," <<  (intptr_t) mb->address << "," << mb->size << "," <<
(char *) mb->address << endl;
  }

  for (auto mb : freeMBList) {
    file << path << "," << type << "," << "FREE" << "," << (intptr_t) mb->address << "," << mb->size << "," << endl;
  }

  file.close();
}

// * * * * * * * * * * * * * * * *
// 2020 Semester 2 - Assignemnt 2
// Author: Arone Susau
// Course: COSC1114
// Student#: s3728977
// * * * * * * * * * * * * * * * *

#include "./../headers/main.h"

static strategies strat;

int main(int argc, char** argv) {

  char* path = argv[1];
  strat.type = argv[2];
  strat.max_names = stoi(argv[3]);

  names names = new char*[strat.max_names];
  memory_list allocMBList, freeMBList;

  readFile(path, names);
  run(names, allocMBList, freeMBList);
  writeAllocMBFile(path, argv[2], allocMBList);
```

```
  writeFreeMBFile(path, argv[2], freeMBList);
  writeHeaderFile(path, argv[2], strat.total_memory);
  writeCombinedFile(path, argv[2], freeMBList, allocMBList);

  for (auto mb: allocMBList) {
    free(mb);
  }

  for (auto mb: freeMBList) {
    free(mb);
  }

  return PROGRAM_SUCCESS;
}

void run(names &names, memory_list &allocMBList, memory_list &freeMBList) {

  while (strat.name_index != strat.max_names) {
    strat.allocate(names, allocMBList, freeMBList);
    removeRandomBlocks(allocMBList, freeMBList);
    freeMBList.sort();
    joinContiguousBlocks(freeMBList);
  }
}

void removeRandomBlocks(memory_list &allocMBList, memory_list &freeMBList) {
  for (auto i = 0; i < RANDOM_MAX; ++i) {
    int index = rand() % allocMBList.size();
    memory_iter it = allocMBList.begin();
    advance(it, index);
    freeMBList.push_back(*it);
    allocMBList.erase(it);
  }
}

void joinContiguousBlocks(memory_list &freeMBList) {
  for (auto it = freeMBList.begin(); it != freeMBList.end(); ++it) {
    memory_block* current = *it;

    auto result = find_if(freeMBList.begin(), freeMBList.end(), [current](memory_block* mb)->bool {
```

```
      return current != mb && current == mb;
    });

    if (result != freeMBList.end()) {
      current->size += (*result)->size;
      freeMBList.erase(result);
    }
  }
}

// * * * * * * * * * * * * * * * *
// 2020 Semester 2 - Assignemnt 2
// Author: Arone Susau
// Course: COSC1114
// Student#: s3728977
// * * * * * * * * * * * * * * * *

#include "../headers/memory_block.h"

bool operator< (const memory_block& a, const memory_block& b) {
    return (intptr_t) a.address < (intptr_t) b.address;
}

bool operator> (const memory_block& a, const memory_block& b) {
    return (intptr_t) a.address > (intptr_t) b.address;
}

bool operator== (const memory_block& a, const memory_block& b) {
    return (intptr_t) a.address + a.size == (intptr_t) b.address;
}
// * * * * * * * * * * * * * * * *
// 2020 Semester 2 - Assignemnt 2
// Author: Arone Susau
// Course: COSC1114
// Student#: s3728977
// * * * * * * * * * * * * * * * *

#include "../headers/strategies.h"

void strategies::allocate(names &names, memory_list &allocMBList, memory_list &freeMBList) {
```

```
  if (type == FIRST) {
    first_pick(names, allocMBList, freeMBList);
  } else if (type == BEST) {
    best_pick(names, allocMBList, freeMBList);
  } else if (type == WORST) {
    worst_pick(names, allocMBList, freeMBList);
  }
}

void strategies::first_pick(names &names, memory_list &allocMBList, memory_list &freeMBList) {

  for (int i = 0; i < ALLOC_MAX && name_index != max_names; ++i) {

    // Find first matching memory_block size from freeMB
    int target = strlen(names[name_index]) + 1;
    auto it = find_if(freeMBList.begin(), freeMBList.end(),
      [target] (memory_block* m)->bool {
        return m->size >= target;
      });

    // If match found, allocate the block otherwise create a new one
    it != freeMBList.end() ?
      allocateBlock(it, names, allocMBList, freeMBList, target) :
      newBlock(names, allocMBList);
  }
}

void strategies::best_pick(names &names, memory_list &allocMBList, memory_list &freeMBList) {

  for (int i = 0; i < ALLOC_MAX && name_index != max_names; ++i) {

    int limit = maxSize(freeMBList);
    int target = strlen(names[name_index]) + 1;
    auto match = false;

    // Select a desired size and increment if no match found
    for (int desired = target; desired <= limit && !match; ++desired) {

      // Find best matching memory_block size from freeMB
      auto it = find_if(freeMBList.begin(), freeMBList.end(),
```

```
      [desired] (memory_block* m)->bool {
        return m->size == desired;
      });

    // If match found, allocate the block otherwise create a new one
    if (it != freeMBList.end()) {
      match = true;
      allocateBlock(it, names, allocMBList, freeMBList, target);
    }
  }

  if (!match) {
    newBlock(names, allocMBList);
  }
 }
}

void strategies::worst_pick(names &names, memory_list &allocMBList, memory_list &freeMBList) {
 for (int i = 0; i < ALLOC_MAX && name_index != max_names; ++i) {

  int limit = maxSize(freeMBList);
  int target = strlen(names[name_index]) + 1;

  // Find worst matching memory_block size from freeMB
  if (limit >= target) {

    auto it = find_if(freeMBList.begin(), freeMBList.end(),
      [limit] (memory_block* m)->bool {
        return m->size == limit;
      });

    // If match found, allocate the block otherwise create a new one
    allocateBlock(it, names, allocMBList, freeMBList, target);
  } else {
    newBlock(names, allocMBList);
  }
 }
}

void strategies::newBlock(names &names, memory_list &allocMBList) {
```

```
  auto size = strlen(names[name_index]) + 1;
  void* address = sbrk(0);
  sbrk(size);
  strcpy((char*) address, names[name_index]);
  memory_block* m = (memory_block*) malloc(sizeof(memory_block));
  m->address = address;
  m->size = size;

  total_memory += size;

  allocMBList.push_back(m);
  name_index++;
}

void strategies::allocateBlock(memory_iter it, names &names, memory_list &allocMBList, memory_list
&freeMBList, int size) {
  memory_block* mb = *it;
  strcpy((char*) mb->address, names[name_index]);

  // Split if memory block greater than name
  if (mb->size > size) {
    splitBlocks(it, freeMBList, size);
  }

  allocMBList.push_back(mb);
  freeMBList.erase(it);
  name_index++;
}

void strategies::splitBlocks(memory_iter it, memory_list &freeMBList, int size) {
  auto mb = *it;
  int remaining = mb->size - size;
  mb->size = size;

  memory_block* m = (memory_block*) malloc(sizeof(memory_block));
  m->address = (void*) ((intptr_t) mb->address + mb->size);
  m->size = remaining;
  freeMBList.push_back(m);
}
```

```
int strategies::maxSize(memory_list &freeMBList) {
  int max = 0;

  for (auto it = freeMBList.begin(); it != freeMBList.end(); ++it) {
    auto mb = *it;
    max = mb->size > max ? mb->size : max;
  }

  return max;
}
```