Deepak R **18ETCS002041**

Laboratory 8

Title of the Laboratory Exercise: Programs for memory management algorithms

1. **Introduction and Purpose of Experiment**

   In a multiprogramming system, the user part of memory must be further subdivided to accommodate multiple processes. This task of subdivision is carried out dynamically done by the operating system known as memory management. By solving these problems students will become familiar with the implementations of memory management algorithms in dynamic memory partitioning scheme of operating system.

2. **Aim and Objectives**

   Aim

   - To develop a simulator for memory management algorithms

   Objectives

   At the end of this lab, the student will be able to

   - Apply memory management algorithms wherever required
   - Develop simulators for the algorithms

3. **Experimental Procedure**

   - Analyse the problem statement
   - Design an algorithm for the given problem statement and develop a flowchart/pseudo-code
   - Implement the algorithm in C language
   - Compile the C program
   - Test the implemented program
   - Document the Results
   - Analyse and discuss the outcomes of your experiment

4. Questions

Implement a simulator for the memory management algorithms with the provision of compaction and garbage collection

a) **First fit**

b) **Best fit**

c) **Worst fit**

5. Calculations/Computations/Algorithms

First Fit:

In the first fit, the partition is allocated which is first sufficient from the top of Main Memory.

1- Input memory blocks with size and processes with size.

2- Initialize all memory blocks as free.

3- Start by picking each process and check if it can
 be assigned to current block.

4- If size-of-process <= size-of-block if yes then
 assign and check for next process.

5- If not then keep checking the further blocks

Best Fit:

Best fit allocates the process to a partition which is the smallest sufficient partition among the free available partitions.

1- Input memory blocks and processes with sizes.

2- Initialize all memory blocks as free.

3- Start by picking each process and find the
 minimum block size that can be assigned to
 current process i.e., find min(bockSize[1],
 blockSize[2],.....blockSize[n]) >
 processSize[current], if found then assign

it to the current process.

5- If not then leave that process and keep checking
the further processes.

Worst Fit:

Worst Fit allocates a process to the partition which is largest sufficient among the freely available partitions available in the main memory. If a large process comes at a later stage, then memory will not have space to accommodate it.

1- Input memory blocks and processes with sizes.

2- Initialize all memory blocks as free.

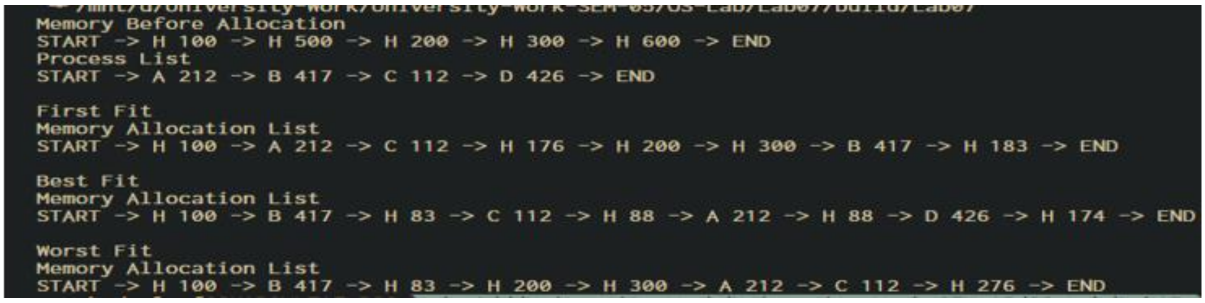
3- Start by picking each process and find the
maximum block size that can be assigned to
current process i.e., find max(bockSize[1],
blockSize[2],.....blockSize[n]) >
processSize[current], if found then assign
it to the current process.

5- If not then leave that process and keep checking
the further processes.

6. Presentation of Results

```
#include <algorithm>
#include <iostream>
#include <iterator>
#include <list>

using block_list_t = std::list<std::pair<char, int>>;
using proc_list_t = std::list<std::pair<char, int>>;

std::ostream& operator<<(std::ostream& out, std::list<std::pair<char, int>> list) {
    out << "START -> ";
    for (auto [mem_name, mem_size] : list) {
        out << mem_name << " " << mem_size << " -> ";
    }
    out << "END" << std::endl;

    return out;
}

void worst_fit(block_list_t blist, proc_list_t plist) {
    std::cout << "Worst Fit" << std::endl;

    for (auto [proc_name, proc_mem] : plist) {
        block_list_t::iterator worst = blist.end();
        for (auto it = blist.begin(); it != blist.end(); it++) {
            auto [block_name, block_size] = *it;
            if (block_name != 'H')
                continue;

            if (block_size >= proc_mem) {
                if (worst == blist.end())
                    worst = it;
                else if (block_size > worst->second)
                    worst = it;
            }
        }

        // found the worst fit
        if (worst != blist.end()) {
            auto [block_name, block_size] = *worst;
            int rem_mem = block_size - proc_mem;
            if (rem_mem > 0) {
                (*worst) = {proc_name, proc_mem};
                blist.insert(std::next(worst), {'H', rem_mem});
            } else if (rem_mem == 0) {
                (*worst) = {proc_name, proc_mem};
            }
        }
    }

    std::cout << "Memory Allocation List" << std::endl;
    std::cout << blist;
}

void best_fit(block_list_t blist, proc_list_t plist) {
    std::cout << "Best Fit" << std::endl;

    for (auto [proc_name, proc_mem] : plist) {
        block_list_t::iterator best = blist.end();
        for (auto it = blist.begin(); it != blist.end(); it++) {
            auto [block_name, block_size] = *it;
            if (block_name != 'H')
                continue;

            if (block_size >= proc_mem) {
                if (best == blist.end())
                    best = it;
                else if (block_size < best->second)
                    best = it;
            }
        }

        // found the best fit
        if (best != blist.end()) {
            auto [block_name, block_size] = *best;
            int rem_mem = block_size - proc_mem;
            if (rem_mem > 0) {
                (*best) = {proc_name, proc_mem};
                blist.insert(std::next(best), {'H', rem_mem});
            } else if (rem_mem == 0) {
                (*best) = {proc_name, proc_mem};
            }
        }
    }

    std::cout << "Memory Allocation List" << std::endl;
    std::cout << blist;
}

void first_fit(block_list_t blist, proc_list_t plist) {
    std::cout << "First Fit" << std::endl;

    for (auto [proc_name, proc_mem] : plist) {
        for (auto it = blist.begin(); it != blist.end(); it++) {
            auto [block_name, block_size] = *it;
            if (block_name != 'H')
                continue;

            int rem_mem = block_size - proc_mem;
            if (rem_mem > 0) {
                (*it) = {proc_name, proc_mem};
                blist.insert(std::next(it), {'H', rem_mem});
                break;
            } else if (rem_mem == 0) {
                (*it) = {proc_name, proc_mem};
                break;
            }
        }
    }

    std::cout << "Memory Allocation List" << std::endl;
    std::cout << blist;
}

int main(int argc, char* argv[]) {
    block_list_t block_list{{'H', 100}, {'H', 500}, {'H', 200}, {'H', 300}, {'H', 600}};
    proc_list_t proc_list{{'A', 212}, {'B', 417}, {'C', 112}, {'D', 426}};

    std::cout << "Memory Before Allocation" << std::endl;
    std::cout << block_list;
    std::cout << "Process List" << std::endl;
    std::cout << proc_list;

    std::cout << std::endl;
    first_fit(block_list, proc_list);

    std::cout << std::endl;
    best_fit(block_list, proc_list);

    std::cout << std::endl;
    worst_fit(block_list, proc_list);

    return EXIT_SUCCESS;
}
```

7. Analysis and Discussions



```
  ~/mnt/d/University-Work/University-Work-SEM-05/OS-Lab/Lab07/build/Lab07
Memory Before Allocation
START -> H 100 -> H 500 -> H 200 -> H 300 -> H 600 -> END
Process List
START -> A 212 -> B 417 -> C 112 -> D 426 -> END

First Fit
Memory Allocation List
START -> H 100 -> A 212 -> C 112 -> H 176 -> H 200 -> H 300 -> B 417 -> H 183 -> END

Best Fit
Memory Allocation List
START -> H 100 -> B 417 -> H 83 -> C 112 -> H 88 -> A 212 -> H 88 -> D 426 -> H 174 -> END

Worst Fit
Memory Allocation List
START -> H 100 -> B 417 -> H 83 -> H 200 -> H 300 -> A 212 -> C 112 -> H 276 -> END
```

Figure 0-1 Output

8. Conclusions

Memory management is the functionality of an operating system which handles or manages primary memory and moves processes back and forth between main memory and disk during execution. Memory management keeps track of each and every memory location, regardless of either it is allocated to some process or it is free. It checks how much memory is to be allocated to processes. It decides which process will get memory at what time. It tracks whenever some memory gets freed or unallocated and correspondingly it updates the status.

9. Comments

1. Limitations of Experiments

The disadvantage with first fit memory management algorithm is that the extra space cannot be used by any other process.

2. Limitations of Results

Worst Fit: A process entering first may be allocated the largest memory space but if another process of larger memory requirement is to be allocated, space cannot be found. This is a serious drawback here.

3. Learning happened

Learnt how memory is allocated or not allocated to each process based on the block size ad process size.

The algorithm which is easy to implement need not be the best.

4. Recommendations

Worst fit algorithm is not recommended to be implemented in the real world as it has many disadvantages.