University of Waterloo
Faculty of Engineering
Department of Electrical and Computer Engineering

ECE 254
Laboratory 4

Prepared by
Jinming Zhang
UW Student ID Number: 20613667
UW User ID: Jm3zhang @edu.uwaterloo.ca
2B Computer Engineering
and
Lun Jing
UW Student ID Number: 20558988
UW User ID: l7jing@edu.uwaterloo.ca
2B Computer Engineering

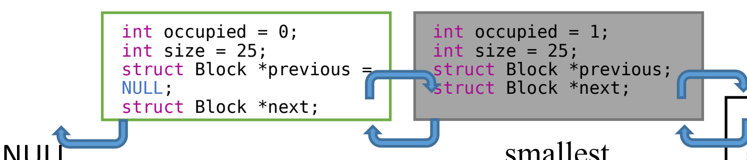
4 December 2017

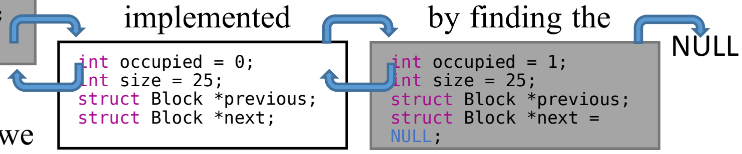
Section 1: Statement of the problem to be discussed in the report

In this lab, we had implemented two methods to allocate the chunk of free memory we requested dynamically from the system. The methods we used here are the best fit algorithm and the worst fit algorithm. The way we tackled this lab is to implement two memory initialization functions (one for best fit and one for worst fit) as a starter and we implement the two main allocation functions (one for best fit and one for worst fit) to manage the the chunk of free memory we requested. After those implementations, we moved to the de-allocate methods because in order for our memory block to work, our memory block should be able to allocate and de-allocate. We also implement the functions to count the external fragmented in our memory blocks as the utility functions for our lab. As this report concerned, we will discuss four problems. We will start with the descriptions of the data structure and algorithms we used to implement the allocation algorithms for our memory block and the descriptions for our test cases. After that, we will do a comparison to the result from running the two allocation method and we will conclude the comparison. The experimental data will be used to support our conclusions.

Section 2: Descriptions of the data structure and algorithms to implement the allocation algorithms

The data structure we used to implement the allocation algorithms is linked list. The reason why we chose linked list is because we both agree on the fact that linked list can be used to represent the memory block we request quite well. For the clarity purpose, we used double linked list in this lab which has two pointers in each node of the linked list point to both pervious node and next node. The node we have for the linked list consists a previous pointer points to the previous node, a next pointer points to the next node, an occupied integer indicates if the current node (represents a particular memory block) is occupied or not (0 means empty and 1 means occupied) and a size integer indicates the size of this particular memory block in the big memory block we allocated. A better view for the double linked listed we used shows as below:





The way the best fit allocation algorithm is implemented by finding the smallest memory block in the big memory block we have that can fit in a particular memory block which need to allocate a spot in the big memory block we have. We have used a while loop for finding the smallest memory block and for each loop, we compare the size of that particular memory block to the smallest memory size variable. We also need to make sure that smallest memory block can hold the new block which requests the memory. If this is smaller and can fit in the new block, we update the smallest memory block to keep track of it. A code example for that is as below:

```
currentBlock = MemBlock;
smallestBlock = currentBlock;
// Do linear search to find and update the block with smallest size
while (currentBlock != NULL ){
     if(currentBlock->occupied == 0 && currentBlock->size < smallest_size &&
(currentBlock->size)-struct_size >= size_alloc){
            smallestBlock = currentBlock;
```

```
            smallest_size = currentBlock->size;
        }
        currentBlock = currentBlock->next;
}
```

(Note: the `size_alloc` variable here is the sum of the block size, the size a structure needs for storage and the delta)

If we can't find smallest memory block which can fit, we will just print "Do not find best fit block". But if we find it, we need to insert this new block which is the second part of the allocation algorithm. We need to split the block to two parts, one part is to store the structure or the node of the new block and the other part is to the actually data. After that, we need to update the pointers, and the nodes attribute data. A code example for that is as below:

```
splitBlock = (block_ptr *)((size_t)smallestBlock + size_alloc);
// update the Block ptr
block_ptr *tmp = smallestBlock->next;
smallestBlock->next = splitBlock;
splitBlock->next = tmp;
if(tmp){
     tmp->previous = splitBlock;
}
splitBlock->previous = smallestBlock;
// update the block size
smallestBlock->size = size_alloc;
splitBlock->size = smallest_size - size_alloc;
// update the state
splitBlock->occupied = 0;
smallestBlock->occupied = 1;
```

(Note: The new block actually size is the smallest block size minus the size a structure needed)
The worst fit algorithm is in similar fashion but instead of finding the smallest block to fit in, it tries to find the biggest block to fit in. This is how we implement the allocation algorithms.

Section 3: Testing scenario description

Part A: Allocate and Deallocate

This part will basically shows how allocation and deallocation work:

Test case:

```
p1 = best_fit_alloc(66);
p2 = best_fit_alloc(9);
best_fit_dealloc(p1);
p3 = best_fit_alloc(21);
best_fit_dealloc(p2);
best_fit_dealloc(p3);
```

best_fit output:

```
[l7jing@ecelinux1 lab4]$ ./a.out 0
 Start Best fit test
 Base Value 6610960
Address: 0 | Size: 1024 | Using: 0 | Next: -6610960 |

my Stucture Size: sz=24
Address: 0 | Size: 92 | Using: 1 | Next: 92 |
Address: 92 | Size: 932 | Using: 0 | Next: -6610960 |

Address: 0 | Size: 92 | Using: 1 | Next: 92 |
Address: 92 | Size: 36 | Using: 1 | Next: 128 |
Address: 128 | Size: 896 | Using: 0 | Next: -6610960 |

Address: 0 | Size: 92 | Using: 1 | Next: 92 |
Address: 92 | Size: 36 | Using: 1 | Next: 128 |
Address: 128 | Size: 48 | Using: 1 | Next: 176 |
Address: 176 | Size: 848 | Using: 0 | Next: -6610960 |

Address: 0 | Size: 92 | Using: 1 | Next: 92 |
Address: 92 | Size: 36 | Using: 0 | Next: 128 |
Address: 128 | Size: 48 | Using: 1 | Next: 176 |
Address: 176 | Size: 848 | Using: 0 | Next: -6610960 |

Address: 0 | Size: 92 | Using: 1 | Next: 92 |
Address: 92 | Size: 932 | Using: 0 | Next: -6610960 |

Address: 0 | Size: 1024 | Using: 0 | Next: -6610960 |

check frag
num = 0
```

worst_fit output:
Base Value 36687888
Address: 0 | Size: 92 | Using: 1 | Next: 92 |
Address: 92 | Size: 932 | Using: 0 | Next: -36687888 |

Address: 0 | Size: 92 | Using: 1 | Next: 92 |
Address: 92 | Size: 36 | Using: 1 | Next: 128 |
Address: 128 | Size: 896 | Using: 0 | Next: -36687888 |

Address: 0 | Size: 92 | Using: 0 | Next: 92 |
Address: 92 | Size: 36 | Using: 1 | Next: 128 |
Address: 128 | Size: 896 | Using: 0 | Next: -36687888 |

Address: 0 | Size: 92 | Using: 0 | Next: 92 |
Address: 92 | Size: 36 | Using: 1 | Next: 128 |
Address: 128 | Size: 48 | Using: 1 | Next: 176 |
Address: 176 | Size: 848 | Using: 0 | Next: -36687888 |

Address: 0 | Size: 128 | Using: 0 | Next: 128 |
Address: 128 | Size: 48 | Using: 1 | Next: 176 |
Address: 176 | Size: 848 | Using: 0 | Next: -36687888 |

Address: 0 | Size: 1024 | Using: 0 | Next: -36687888 |

To make calculation check more easily, we track the base value of the starting node and make address values more easily to calculate.

Part B: External Fragmentation Comparison

To test how external fragmentation, it is unrealistic to manually allocate and then deallocate many times and then compare, which will be a huge workload. Instead, we first divide the block into different pieces, and then make each part unoccupied, then we start to allocate another group and see the result. In details:

```
best_fit_memory_init(2048);     // initizae 2KB

for(i = 0; i < 20; i++){
        size = 50 + i * 5;
        p[i] = best_fit_alloc(size);
}

tmp = MemBlock;
```

```
// cut the memeory into many blocks with different sizes
while(tmp){

        tmp->occupied = 0;
        tmp = tmp->next;
        // make each block available again, such that they will not merge together
}


for(i = 0; i < 30; i++){
        size = 10 + 30*(i%4 + 1);
        q[i] = best_fit_alloc(size);
        print_list(0);
        if(q[i]){
                can_fit++;        // used to track how many request can be satisfied
        }
}

num = best_fit_count_extfrag(50);
// the blocks that have size less than 50 will be counted.
```

For best fit and worst fit, everything is same in this part except the function calls. Note that we set the parameter of best_fit_count_extfrag as 50 bytes, which is very small. And here is the results:

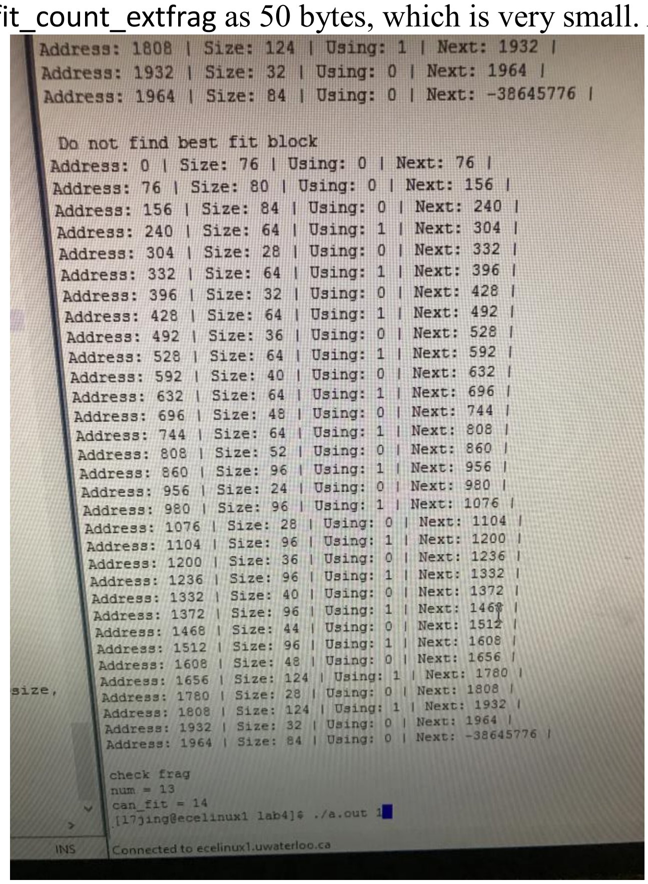


Figure 1. best_fit

From the diagram, we see that there are 12 blocks have size less than 50, but there are still two block with size larger than 100.

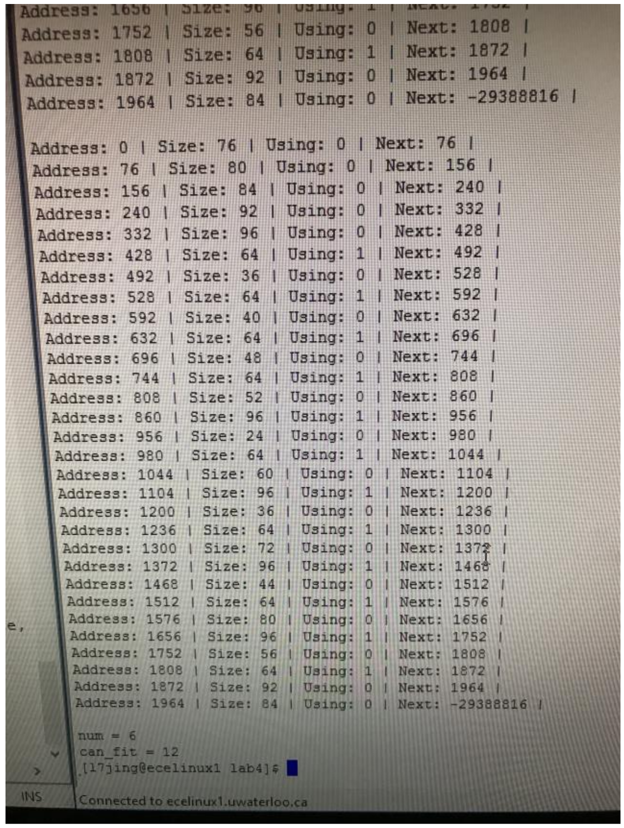


Figure 2. worst_fit

As we can see that when we have small initialized memory size, best_fit will have a little more external fragments, however, best_fit can satisfy more request than worst_fit.

In another case, when we have initialized size large enough:

```
best_fit_memory_init(8192);     // initizae 2KB

    for(i = 0; i < 20; i++){
        size = 380 + 10*(i%3 + 1);
        p[i] = best_fit_alloc(size);
    }

    tmp = MemBlock;
    // cut the memeory into many blocks with different sizes
    while(tmp){
```

```
            tmp->occupied = 0;
            tmp = tmp->next;
            // make each block available again, such that they will not merge together
    }


    for(i = 0; i < 30; i++){
            size = 40 + 4*(i%4 + 1);
            q[i] = best_fit_alloc(size);
            print_list(0);
            if(q[i]){
                    can_fit++;          // used to track how many request can be satisfied
            }
    }

    num = best_fit_count_extfrag(380);
```

The output in this case is

Best fit test
num = 37
can_fit = 30

Best fit test
num = 50
can_fit = 30

Section 4: Conclusion

In conclusion, when the memory size is small, best_fit may cause some more fragments, but it allow to run more tasks. Worst_fit always start allocation from the largest block. In this case, when the total memory size is large, and more tasks allocation requests come, the worst_fit will create more fragments than best_fit.

Generally, the best_fit will manage the memory space more efficiently than worst_case. Also, it can guarantee that more allocation requirements could be satisfied.