Note: Problem 1 uses C++ 11 standard and VS 2015 to compile.

1. The Buddy System Code shows as figure 1.1, 1.2 shows

```
/**   allocate memory to process          */
bool Allocate(pair<string, int> &input, Node* root, Node* node, unordered_map<string, Node*> &map, list<Node*> &set) {
    int size = calculate(input.second);                    // calculate the size of the process need in the memory
    Node* temp = node;
    while (temp->get_Val() > size) {
        int split_value = temp->get_Val() / 2;
        temp->set_Flag(true);
        Node* left = new Node(split_value);
        Node* right = new Node(split_value);
        temp->set_Left(left);
        temp->set_Right(right);
        left->set_Father(temp);
        right->set_Father(temp);
        left->set_Brother(right);
        right->set_Brother(left);
        list<Node*>::iterator it = set.begin();
        for (it; it != set.end(); ++it)
            if ((*it) == temp)
                break;
        temp = temp->get_Left();
        set.insert(it,left);
        set.insert(it,right);
        set.erase(it);
    }
    temp->set_Allocate(input.second);
    temp->set_Flag(true);
    map[input.first] = temp;
    return true;
}
```

Figure 1.1

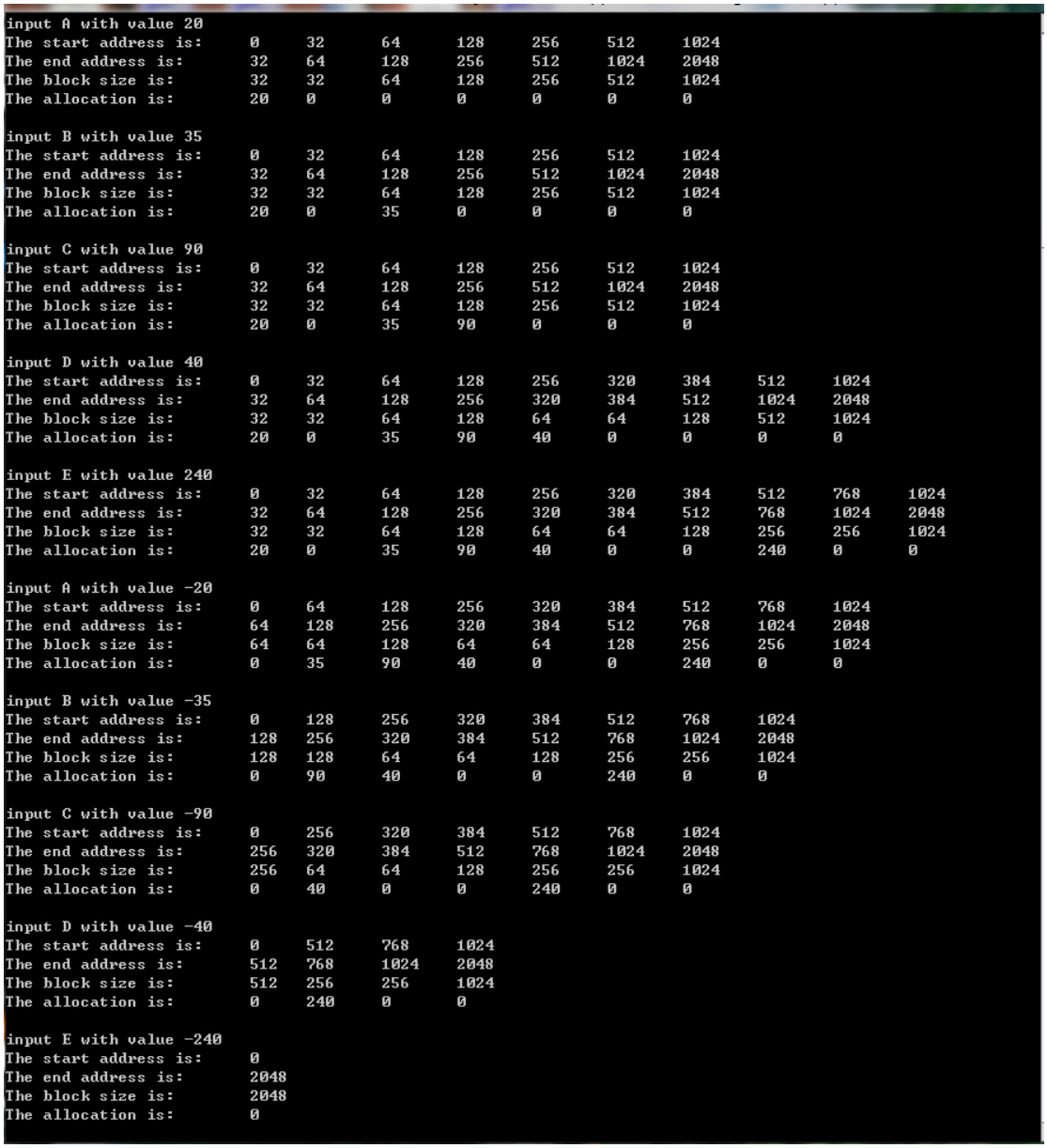
```
/**   release memory of process          */
bool Release(pair<string, int> &input, Node* root, Node* node, unordered_map<string, Node*> &map, list<Node*> &set) {
    Node* temp = map[input.first];
    temp->set_Allocate(0);
    temp->set_Flag(false);

    if (node == root) {
        root->set_Allocate(0);
        root->set_Flag(false);
    }

    while (temp->get_Brother()->get_Flag() == false) {
        Node* child_1 = temp;
        Node* child_2 = temp->get_Brother();
        list<Node*>::iterator it_child1 = set.begin();
        list<Node*>::iterator it_child2 = set.begin();
        for (it_child1; it_child1 != set.end(); ++it_child1)
            if ((*it_child1) == temp)
                break;
        for (it_child2; it_child2 != set.end(); ++it_child2)
            if ((*it_child2) == temp->get_Brother())
                break;
        temp = temp->get_Father();
        temp->set_Allocate(0);
        temp->set_Flag(false);
        set.insert(it_child1, temp);
        set.erase(it_child1);
        set.erase(it_child2);
        delete child_1;
        delete child_2;
        if (set.size() == 1)
            break;
    }
    return true;
}
```

Figure 1.2

The Buddy System output table shows as follows:



2.

9	11	12

0 9 20 32

Because the page table field are 9+11 = 20. The offset is 32 – 20 = 12.

Thus, the page size is 2^12 B = 4KB.

The number of pages size are 2^20

3. The page reference string is (3, 15, 3, 19, 3, 8, 19, 3, 3, 3) or (3, 15, 3, 19, 3, 8, 19, 3, 3, 3, 9)

Condition: PC = 2020, SP = 10192, page size = 512B, instruction size = 4B

PC	Operation	Reference Pages
2020	Load word *8118* (a value of a procedure input parameter) into register *0*	PC: 2020/512 = 3 Data: 8118/512 = 15

2024	Push register *0* onto the stack	PC: 2024/512 = 3 Stack: 10192/512 = 19 SP: 10192 - 4 = 10188
2028	Call a procedure at *4120* (the return address is stacked)	PC: 2028/512 = 3 Data：4120/512 = 8 Stack: 10188/512 = 19
2032	Remove one word (the actual parameter) from the stack and	PC: 2032/512 = 3
2036	Compare the actual parameter to constant *7*	PC: 2036/512 = 3
2040	If equal, jump to *5000*	PC: 2040/512 = 3 Data: 5000/512 = 9 (depend on whether the matching is equal)

4.

1) Explain the value and drawbacks of "non-traditional" hardware architectures for supporting virtual memory (i.e., *inverted page table* and *TLB-only*).

TLB:

Value:

a. TLB with high hit ratio will save significant time of finding a page.

Drawbacks:

a. TLB size is not enough for memory-intensive.
b. TLB is expensive
c. Increase the page size of TLB will reduce time but results fragments of some application do not need so much space.

Inverted Page Tables:

Value:

a. Inverted Page Tables reduce the amount of physical memory need to track virtual-to-physical address translations.

Drawbacks:

a. The Inverted Page Tables do not contain complete information about the logical address space of a process, and that information is required if a referenced page is not currently in memory will cause page faults.
b. Inverted Page Tables may need external page tables may negate the utility of inverted tables because page faults consistently generated by the requirement of a process.

2)

The physical page number of process 1 are 4, 2, 0.

Process ID	Page number	Physical page frame
1	0	4
1	1	2
1	2	0

5.

1) MRU

MRU will result in $3(l+1)$ page faults with the string $\omega=(1,2,3)^k\,(4,5,6)^l$

Proof: MRU is the Most Recently Used page replacement which will replace the page most recently used page out.

1	2	3	… k-1 times …	4	5	6	… l-1 times …
1	1	1	1	1	1	1	1
	2	2	2	2	2	2	2
		3	3	4	5	6	(4,5,6….)
F	F	F		F	F	F	F….F

From this table, when CPU want page 4,5,6, it always not in the block and always is replaced to the block. Thus, the faults time is 3*l. When page 1, 2, 3 come in the memory, there are 3 times faults. Consequently, the total faults times of MRU are 3*(l + 1).

2) LFU

LFU will result in 3[min(k,l)+1] page faults

Proof: LFU is Least Frequency Used page replacement which will replace the Least frequency used page out.

1	2	3	… k-1 times …	4	5	6	… k-2 times …	(K)th			l-k > 0
1	1	1	1	4	5	6	4,5,6	4	4	4	4
	2	2	2	2	2	2	2	2	5	5	5
		3	3	3	3	3	3	3	3	6	6
F	F	F		F	F	F	F….F	F	F	F	

From this table, the page 1,2,3 enter the block cause 3 times page faults. Then, before page 4,5,6 coming, page 1, 2, 3's count are k times. If l < =k, the 4,5,6's count are less than k, then they will be replaced by l times which means the there will be 3*l page faults. If l>k, the page 4,5,6's count are larger than k which means there will be k times page faults. Thus, the total page faults are 3*[min(k, l) + 1].

3) FIFO

FIFO will result in 6 page faults.

Proof: FIFO is First In First Out which means first page in the memory will be replaced out.

1	2	3	… k-1 times …	4	5	6	… l-1 times …
1	1	1	1	4	4	4	1
	2	2	2	2	5	5	2
		3	3	3	3	6	(4,5,6….)
F	F	F		F	F	F	

From this table, the page 1,2,3 cause 3 page faults at first and page 4,5,6 cause another 3 page faults when they come in. Thus, the total page faults is 6.

4) LRU

LRU will result in 6 page faults.

Proof, LRU is Least Recently Used replacement which means it will replace the recently used pages out.

1	2	3	… k-1 times …	4	5	6	… l-1 times …
1	1	1	1	4	4	4	1
	2	2	2	2	5	5	2
		3	3	3	3	6	(4,5,6….)
F	F	F		F	F	F	

From this table, the page 1,2,3 cause 3 page faults at first and page 4,5,6 cause another 3 page faults when they come in at first. Thus, the total page faults is 6.