Assignment4 Report

Section A (30 marks)

1.Definition and description of the Data structures (include the implemented methods)

1.1 Buddy algorithm data structures

- **buddy_system_free_block**

```
//Data Structures of Buddy System Allocation Algorithm
class buddy_system_free_block { //buddy system free block
public:
    int firstAddress;
    int blockLevel; //0,1,2,3,4,5,6,7,8,9,10
    buddy_system_free_block *next;
    buddy_system_free_block *prev;
```

```
    buddy_system_free_block(int first_address, int block_level);
};

buddy_system_free_block::buddy_system_free_block(int first_Address, int block_leve
l) {
    firstAddress = first_Address;
    blockLevel = block_level;
    next = nullptr;
    prev = nullptr;
}
```

- each `buddy_system_free_block` has
- `firstAddress` of the free block
- `blockLevel` //0(size:1), 1(size:2), 2(size:4), 3(size:8), 4(size:16), 5(size:32), 6(size:64), 7(size:128), 8(size:256), 9(size:512), 10(size:1024)
- `next` a pointer linked with the next `buddy_system_free_block` in the same `buddy_system_free_list`
- `prev` a pointer linked with the previous `buddy_system_free_block` in the same `buddy_system_free_list`

- **buddy_system_free_list**

```
class buddy_system_free_list { // buddy system free list
public:
    int blockLevel; //0,1,2,3,4,5,6,7,8,9,10
    buddy_system_free_block *head;
};
```

- each buddy_system_free_list has
- `blockLevel` indicate the block level of the `buddy_system_free_list`/0(size:1), 1(size:2), 2(size:4), 3(size:8), 4(size:16), 5(size:32), 6(size:64), 7(size:128), 8(size:256), 9(size:512), 10(size:1024)

1.2 allocated information of specific sequence number's data structures

- **page_info**

```
class page_info { // page information
public:
    int pageFrameStatus; // -1：unUse, 0:initial allocated, 1: freed, 2: evicted,
 3: LRU inactive list, 4: LRU active list
    int physicalAddress;
```

```
    page_info(int page_frame_status, int physical_address);
};

page_info::page_info(int page_frame_status, int physical_address) {
    pageFrameStatus = page_frame_status; // -1：unUse, 0:initial allocated, 1: fre
ed, 2: evicted, 3: LRU inactive list, 4: LRU active list
    physicalAddress = physical_address;
}
```

- each `page_info` record the information of a specific page of a “A” command
- `pageFrameStatus`
  - -1: unUse
  - 0: initial allocated
  - 1: freed
  - 2: evicted
  - 3: in LRU inactive list
  - 4: in LRU active list
- `physicalAddress` record the physical address of a page
  - if the page is freed or evicted, then `physicalAddress = -1`

- **buddy_system_allocated_block**

```
class buddy_system_allocated_block { // the allocated information of a seqNum
public:
    int blockSize;
    int sizeOfInternalSegment;
    int numOfFreed;
    map<int, page_info *> pagesMapInformation;

    buddy_system_allocated_block(int num_of_page, buddy_system_free_block *block);
};

buddy_system_allocated_block::buddy_system_allocated_block(int num_of_page, buddy_
system_free_block *newBlock) {
    blockSize = pow(2, newBlock->blockLevel);
    sizeOfInternalSegment = blockSize - num_of_page;
    numOfFreed = 0;

    page_info *pageInfo;
    int pageStatus = 0;
    for (int i = 0; i < blockSize; i++) {
        if (i < num_of_page) {
            pageStatus = 0;
        } else {
```

```
            pageStatus = -1;
        }

        pageInfo = new page_info(pageStatus, newBlock->firstAddress + i);
        pagesMapInformation.insert(pair<int, page_info *>(i, pageInfo));
    }
}
```

- record all the allocated information of a “A” command with a specific sequence number, like log
- `blockSize`:
- `sizeOfInternalSegment`: size of internal fragment
- `numOfFreed`: indicate the number of page which is already freed
- `map<int, page_info *> pagesMapInformation`: maintain a map structure, which make it easier to operate the page information

1.3 LRU data structures

- **LRU_item**

```
class LRU_item {
public:
    int seqNum;
    int pageFrameOffset;
    int physicalAddress;
    LRU_item *next;
    LRU_item *prev;

    LRU_item(int sequence_num, int page_frame_offset, int physical_address);
};

LRU_item::LRU_item(int sequence_num, int page_frame_offset, int physical_address)
 {
    seqNum = sequence_num;
    pageFrameOffset = page_frame_offset;
    physicalAddress = physical_address;
    next = nullptr;
    prev = nullptr;
}
```

- each LRU_item is an item in the LRU inactive list or LRU active list; it record the information:
  - `seqNum`, the sequence number of the command “A”
  - `pageFrameOffset`, page’s offset of a allocated block

        - `physicalAddress` :physical address
        - `next` a pointer linked with the next **LRU_item** in the LRU inactive list or LRU active list
        - `pre` a pointer linked with the previous **LRU_item** in the LRU inactive list or LRU active list
- **LRU_list**
    - the data structure used to maintain LRU inactive list or LRU active list
    - `numOfItems`: the number of the items in the list
    - `maxCapacity`: (250) the maximum capacity of the list
    - `listHead` a pointer point to the head of the list
    - `listTail` a pointer point to the tail of the list

```
class LRU_list {
public:
    int numOfItems;
    int maxCapacity;
    LRU_item *listHead;
    LRU_item *listTail;
    map<int, LRU_item *> addressMap;

    LRU_list(int max_capacity);

    LRU_item *addToHead(LRU_item *newLRUItem);

    LRU_item *deleteItem(int);

    LRU_item *deleteTailItem();

};

LRU_list::LRU_list(int max_capacity) {
    numOfItems = 0;
    maxCapacity = max_capacity;
    listHead = nullptr;
    listTail = nullptr;
}
```

    - public method of the class `LRU_list`
        - **addToHead**
            - add a LRU_item to the head of the list

```
LRU_item *LRU_list::addToHead(LRU_item *newLRUItem) {
    LRU_item *returnItem = nullptr;

    if (numOfItems == maxCapacity) {
        returnItem = deleteTailItem();
        returnItem->prev = nullptr;
        returnItem->next = nullptr;
    }

    if (numOfItems == 0) {
        listHead = newLRUItem;
        listTail = newLRUItem;
    } else {
        listHead->prev = newLRUItem;
        newLRUItem->next = listHead;
        newLRUItem->prev = nullptr;
        listHead = newLRUItem;
    }

    numOfItems++;
    addressMap.insert(pair<int, LRU_item *>(newLRUItem->physicalAddre
ss, newLRUItem));


    return returnItem;
}
```

- **deleteItem**
  - delete an item from the list with specific physical address

```


LRU_item *LRU_list::deleteItem(int physicalAddress) {
    LRU_item *relatedItem = nullptr;

    if (addressMap.count(physicalAddress) == 0) {
        return relatedItem;
    }

    relatedItem = addressMap.find(physicalAddress)->second;

    if (relatedItem->prev == nullptr) {
        listHead = relatedItem->next;
    } else {
        relatedItem->prev->next = relatedItem->next;
    }

    if (relatedItem->next == nullptr) {
        listTail = relatedItem->prev;
    } else {
        relatedItem->next->prev = relatedItem->prev;
    }

    numOfItems--;
```

```
    addressMap.erase(physicalAddress);

    relatedItem->prev = nullptr;
    relatedItem->next = nullptr;

    return relatedItem;
}
```

- **deleteTailItem**
  - delete the tail item from the list

```
LRU_item *LRU_list::deleteTailItem() {
    LRU_item *relatedItem = listTail;

    if (numOfItems == 0) {
        relatedItem = nullptr;
        return relatedItem;
    } else if (numOfItems == 1) {
        listHead = nullptr;
        listTail = nullptr;
    } else {
        listTail->prev->next = nullptr;
        listTail = listTail->prev;
    }

    numOfItems--;
    addressMap.erase(relatedItem->physicalAddress);

    relatedItem->prev = nullptr;
    relatedItem->next = nullptr;
    return relatedItem;
}
```

1.4 Buddy_System

- As the following code shows. The `buddy_system_free_area[11]` , `inactiveList` and `activeList` of the LRU system is stored in the class `Buddy_System`
- also the allocated information of each sequence number `seqNUM_allocated_logs` is is stored in the class `Buddy_System`

```
class Buddy_System {
public:
    buddy_system_free_list buddy_system_free_area[11];
    map<int, buddy_system_allocated_block *> seqNUM_allocated_logs;
    int buddy_system_block_size[11] = {1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 102
4};
    LRU_list *inactiveList;
    LRU_list *activeList;
```

```
    Buddy_System();

    void allocate(int, int);

    void access(int, int);

    void free(int, int);

    buddy_system_free_block *buddy_system_allocate(int);

    void buddy_system_deallocate(int);

    void buddy_system_divide_larger_block(int);

    int compute_buddy_address(int, int);

    void buddy_system_check_merge(int);

    void buddy_system_insert_block_to_one_list(int, int);

    void printStateOfBuddyAndLRU();
};

Buddy_System::Buddy_System() {
    for (int i = 0; i < 11; i++) {
        buddy_system_free_area[i].blockLevel = i;
        buddy_system_free_area[i].head = nullptr;
    }

    inactiveList = new LRU_list(250);
    activeList = new LRU_list(250);

    buddy_system_insert_block_to_one_list(0, 9);
}
```

- some of the logic is implemented by the following functions:

```
buddy_system_free_block *Buddy_System::buddy_system_allocate(int numOfPage) {
    //determine the block size
    int blockSizeLevel = -1;
    for (int i = 0; i < 11; i++) {
        if (numOfPage <= buddy_system_block_size[i]) {
            blockSizeLevel = i;
            break;
        }
    }

    if (!buddy_system_free_area[blockSizeLevel].head) {
        //split the larger block to produce small blocks
        buddy_system_divide_larger_block(blockSizeLevel + 1);
    }

    //evict inactive list to free more pages
    if (!buddy_system_free_area[blockSizeLevel].head) {
```

```
        while (!buddy_system_free_area[blockSizeLevel].head && inactiveList->numOf
Items > 0) {
            LRU_item *returnedLRUItem = inactiveList->deleteTailItem();

            buddy_system_deallocate(returnedLRUItem->physicalAddress);
            buddy_system_allocated_block *relatedBlock;
            relatedBlock = seqNUM_allocated_logs.find(returnedLRUItem->seqNum)->se
cond;
            relatedBlock->pagesMapInformation[returnedLRUItem->pageFrameOffset]->p
hysicalAddress = -1;
            relatedBlock->pagesMapInformation[returnedLRUItem->pageFrameOffset]->p
ageFrameStatus = 2;

            buddy_system_check_merge(0);
        }
    }

    //if still need mare pages, we need evict active list to free more pages
    if (!buddy_system_free_area[blockSizeLevel].head) {
        while (!buddy_system_free_area[blockSizeLevel].head && activeList->numOfIt
ems > 0) {
            LRU_item *returnedLRUItem = activeList->deleteTailItem();

            buddy_system_deallocate(returnedLRUItem->physicalAddress);
            buddy_system_allocated_block *relatedBlock;
            relatedBlock = seqNUM_allocated_logs.find(returnedLRUItem->seqNum)->se
cond;
            relatedBlock->pagesMapInformation[returnedLRUItem->pageFrameOffset]->p
hysicalAddress = -1;
            relatedBlock->pagesMapInformation[returnedLRUItem->pageFrameOffset]->p
ageFrameStatus = 2;

            buddy_system_check_merge(0);
        }
    }


    //allocated a free block from the buddy system
    buddy_system_free_block *result;
    result = buddy_system_free_area[blockSizeLevel].head;

    buddy_system_free_area[blockSizeLevel].head = buddy_system_free_area[blockSize
Level].head->next;
    if (buddy_system_free_area[blockSizeLevel].head) {
        buddy_system_free_area[blockSizeLevel].head->prev = nullptr;
    }

    return result;
}

void Buddy_System::buddy_system_deallocate(int physicalAddress) {
    buddy_system_insert_block_to_one_list(physicalAddress, 0);
}

void Buddy_System::buddy_system_divide_larger_block(int nextLevel) {
    if (nextLevel == 10) {
        cout << "no space to allocate****************************************"
 << endl;
```

```
        return;
    }

    if (!buddy_system_free_area[nextLevel].head) {
        //recursion of the buddy_system_divide_larger_block method
        buddy_system_divide_larger_block(nextLevel + 1);
    }

    if (!buddy_system_free_area[nextLevel].head) {
        return;
    }

    //update: insert 2 small blocks to the buddy system
    buddy_system_insert_block_to_one_list(buddy_system_free_area[nextLevel].head->
firstAddress, nextLevel - 1);
    buddy_system_insert_block_to_one_list(buddy_system_free_area[nextLevel].head->
firstAddress + pow(2, nextLevel) / 2,
                                          nextLevel - 1);

    //update: delete the bigger block from the buddy system
    buddy_system_free_area[nextLevel].head = buddy_system_free_area[nextLevel].hea
d->next;
    if (buddy_system_free_area[nextLevel].head) {
        buddy_system_free_area[nextLevel].head->prev = nullptr;
    }
}

//used to compute the first address of a block which can used to merge with anothe
r block of a specific physical address
int Buddy_System::compute_buddy_address(int startAddress, int blockLevel) {
    int mergeBlockLevel;
    if (blockLevel == 0) {
        mergeBlockLevel = 1;
    } else {
        int temp = pow(2, blockLevel - 1);
        mergeBlockLevel = temp << 1;
    }
    int buddyAddress = startAddress ^ mergeBlockLevel;
    return buddyAddress;
}

void Buddy_System::buddy_system_check_merge(int blockLevel) {
    bool mergeHappened = false;

    buddy_system_free_block *scanPointer;
    scanPointer = buddy_system_free_area[blockLevel].head;

    while (scanPointer != nullptr) {
        if (scanPointer->next == nullptr) {
            break;
        }

        if (scanPointer->next->firstAddress == compute_buddy_address(scanPointer->
firstAddress, blockLevel)) {
            //merge 2 blocks

            buddy_system_insert_block_to_one_list(scanPointer->firstAddress, block
Level + 1);
```

```
            if (scanPointer->prev == nullptr) {
                buddy_system_free_area[blockLevel].head = scanPointer->next->next;
                if (scanPointer->next->next != nullptr) {
                    scanPointer->next->next->prev = nullptr;
                }
            } else {
                scanPointer->prev->next = scanPointer->next->next;
                if (scanPointer->next->next != nullptr) {
                    scanPointer->next->next->prev = scanPointer->prev;
                }
            }

            scanPointer = scanPointer->next->next;
            mergeHappened = true;
        } else {
            //cannot merge
            scanPointer = scanPointer->next;
        }
    }

    if (blockLevel == 10 || !mergeHappened) {
        return;
    }

    if (mergeHappened) {
        buddy_system_check_merge(blockLevel + 1);
    }
}

void Buddy_System::buddy_system_insert_block_to_one_list(int firstAddress, int blo
ckLevel) {
    buddy_system_free_block *newBlock = new buddy_system_free_block(firstAddress,
 blockLevel);
    buddy_system_free_block *scanPointer;
    scanPointer = buddy_system_free_area[blockLevel].head;

    if (buddy_system_free_area[blockLevel].head == nullptr) {
        buddy_system_free_area[blockLevel].head = newBlock;
    }

    while (scanPointer != nullptr) {
        if (scanPointer->firstAddress > firstAddress) {
            if (scanPointer->prev == nullptr) {
                buddy_system_free_area[blockLevel].head = newBlock;
                newBlock->prev = nullptr;
                newBlock->next = scanPointer;
                scanPointer->prev = newBlock;
            } else {
                scanPointer->prev->next = newBlock;
                newBlock->prev = scanPointer->prev;
                newBlock->next = scanPointer;
                scanPointer->prev = newBlock;
            }
            break;
        } else if (scanPointer->firstAddress < firstAddress) {
            if (scanPointer->next == nullptr) {
                scanPointer->next = newBlock;
```

```
                newBlock->prev = scanPointer;
                newBlock->next = nullptr;
                break;
            } else {
                scanPointer = scanPointer->next;
            }
        } else {
            cout << "buddy system insert error" << endl;
            break;
        }
    }
}
```

- `allocate(int seqNum, int numOfPage)` **method**
  - used to handle the command A <seqNum> <numOfPages>

```
void Buddy_System::allocate(int seqNum, int numOfPage) {
    //allocated a new block of pages from the buddy system
    buddy_system_free_block *result;
    result = buddy_system_allocate(numOfPage);

    buddy_system_allocated_block *allocatedBlock;
    allocatedBlock = new buddy_system_allocated_block(numOfPage, result);
    this->seqNUM_allocated_logs.insert(pair<int, buddy_system_allocated_block
*>(seqNum, allocatedBlock));

    // add the new allocated pages to the active list
    for (int offset = 0; offset < numOfPage; offset++) {
        access(seqNum, offset);
        access(seqNum, offset);
    }

    for (int offset = numOfPage; offset < allocatedBlock->blockSize; offset+
+) {
        access(seqNum, offset);
        access(seqNum, offset);
    }
}
```

- `access(int seqNum, int offset)` **method**
  - used to handle the command X <seqNum> <offsetOfPages>

```
void Buddy_System::access(int seqNum, int offset) {
    //get the stored information of the seqNum
    buddy_system_allocated_block *allocatedBlock;
    allocatedBlock = seqNUM_allocated_logs.find(seqNum)->second;

    page_info *pageInformation;
    pageInformation = allocatedBlock->pagesMapInformation.find(offset)->secon
d;
```

```
    if (offset >= (allocatedBlock->blockSize - allocatedBlock->sizeOfInternal
Segment)) {
        cout << "seqnum" << seqNum << "offset" << offset << "spurious access"
<< endl;
    }

    LRU_item *newLRUItem;

    if (pageInformation->pageFrameStatus == 0 || pageInformation->pageFrameSt
atus == -1) {
        //first access: add the page to the head of the inactive list
        pageInformation->pageFrameStatus = 3;
        newLRUItem = new LRU_item(seqNum, offset, pageInformation->physicalAd
dress);
        LRU_item *returnedLRUItem = inactiveList->addToHead(newLRUItem);

        if (returnedLRUItem != nullptr) {
            buddy_system_deallocate(returnedLRUItem->physicalAddress);
            buddy_system_allocated_block *relatedBlock;
            relatedBlock = seqNUM_allocated_logs.find(returnedLRUItem->seqNu
m)->second;
            relatedBlock->pagesMapInformation[returnedLRUItem->pageFrameOffse
t]->physicalAddress = -1;
            relatedBlock->pagesMapInformation[returnedLRUItem->pageFrameOffse
t]->pageFrameStatus = 2;

            buddy_system_check_merge(0);
        }
    } else if (pageInformation->pageFrameStatus == 1) {
        //already freed
        cout << "seqNum " << seqNum << " offset " << offset << " spurious acc
ess" << endl;
    } else if (pageInformation->pageFrameStatus == 2) {
        //handle page fault
        buddy_system_free_block *newBlock;
        newBlock = buddy_system_allocate(1);

        pageInformation->physicalAddress = newBlock->firstAddress;
        pageInformation->pageFrameStatus = 3;

        //add to inactive List
        newLRUItem = new LRU_item(seqNum, offset, pageInformation->physicalAd
dress);
        LRU_item *returnedLRUItem = inactiveList->addToHead(newLRUItem);

        if (returnedLRUItem != nullptr) {
            buddy_system_deallocate(returnedLRUItem->physicalAddress);
            buddy_system_allocated_block *relatedBlock;
            relatedBlock = seqNUM_allocated_logs.find(returnedLRUItem->seqNu
m)->second;
            relatedBlock->pagesMapInformation[returnedLRUItem->pageFrameOffse
t]->physicalAddress = -1;
            relatedBlock->pagesMapInformation[returnedLRUItem->pageFrameOffse
t]->pageFrameStatus = 2;

            buddy_system_check_merge(0);
        }
```

```
    } else if (pageInformation->pageFrameStatus == 3) {
        //move from inActiveList to the ActiveList
        newLRUItem = inactiveList->deleteItem(pageInformation->physicalAddres
s);
        LRU_item *returnedLRUItem;
        returnedLRUItem = activeList->addToHead(newLRUItem);

        if (returnedLRUItem != nullptr) {
            inactiveList->addToHead(returnedLRUItem);
            buddy_system_allocated_block *relatedBlock;
            relatedBlock = seqNUM_allocated_logs.find(returnedLRUItem->seqNu
m)->second;
            relatedBlock->pagesMapInformation[returnedLRUItem->pageFrameOffse
t]->pageFrameStatus = 3;
        }

        pageInformation->pageFrameStatus = 4;
    } else if (pageInformation->pageFrameStatus == 4) {
        //already in ActiveList do nothing
    }
}
```

- `free(int seqNum, int numOfPage)` **method**
  - used to handle the command F <seqNum> <numOfPages>

```
void Buddy_System::free(int seqNum, int numOfPage) {
    //get the stored information of the seqNum
    buddy_system_allocated_block *allocatedBlock;
    allocatedBlock = seqNUM_allocated_logs.find(seqNum)->second;
    page_info *pageInformation;

    //already freed
    if (allocatedBlock->numOfFreed == (allocatedBlock->blockSize - allocatedB
lock->sizeOfInternalSegment)) {
        return;
    }

    int collectedPages = 0;
    int pageOffset = allocatedBlock->numOfFreed;

    while (collectedPages < numOfPage &&
           allocatedBlock->numOfFreed < (allocatedBlock->blockSize - allocate
dBlock->sizeOfInternalSegment)) {
        pageInformation = allocatedBlock->pagesMapInformation.find(pageOffse
t)->second;
        if (pageInformation->pageFrameStatus == 0) {
            buddy_system_deallocate(pageInformation->physicalAddress);
        } else if (pageInformation->pageFrameStatus == 2) {
            //already evicted
        } else if (pageInformation->pageFrameStatus == 3) {
            //delete from inactive list
            inactiveList->deleteItem(pageInformation->physicalAddress);
            //deallocate to the buddy system
```

```
            buddy_system_deallocate(pageInformation->physicalAddress);
        } else if (pageInformation->pageFrameStatus == 4) {
            //delete from inactive list
            activeList->deleteItem(pageInformation->physicalAddress);
            //deallocate to the buddy system
            buddy_system_deallocate(pageInformation->physicalAddress);
        }

        pageInformation->pageFrameStatus = 1;
        pageInformation->physicalAddress = -1;

        collectedPages++;
        pageOffset++;
        allocatedBlock->numOfFreed++;
    }

    if (allocatedBlock->numOfFreed == (allocatedBlock->blockSize - allocatedB
lock->sizeOfInternalSegment)) {
        //the freed operation of the internal fragment
        for (int i = allocatedBlock->numOfFreed; i < allocatedBlock->blockSiz
e; i++) {
            if (allocatedBlock->pagesMapInformation.find(i)->second->physical
Address != -1) {
                buddy_system_deallocate(allocatedBlock->pagesMapInformation.f
ind(i)->second->physicalAddress);
                inactiveList->deleteItem(allocatedBlock->pagesMapInformation.
find(i)->second->physicalAddress);
                activeList->deleteItem(allocatedBlock->pagesMapInformation.fi
nd(i)->second->physicalAddress);
            }
        }
    }
    buddy_system_check_merge(0);
}
```

1.5 main function

```
int main() {

    //read data from file
    char type[5000];
    int seqno[5000];
    int num[5000];
    char buff[255];
    FILE *fp = NULL;
    if ((fp = fopen("./A0248373X-assign4-input.dat", "r")) == NULL) {
        printf("file cannot open!");
        return 0;
    } else {
        int i = 0;
        while (!feof(fp)) {
            if (fgets(buff, 128, fp) != NULL) {
                char *split;
                split = strtok(buff, "\t");
```

```
                type[i] = split[0];
                split = (strtok(NULL, "\t"));
                seqno[i] = atoi(split);
                split = (strtok(NULL, "\t"));
                num[i] = atoi(split);
                i++;
            }
        }
        fclose(fp);
    }

    Buddy_System *buddySystem = new Buddy_System();

    for (int i = 0; i <= 4146; i++) {
        cout << i << " begin" << endl;
        if (type[i] == 'A') {
            buddySystem->allocate(seqno[i], num[i]);
        } else if (type[i] == 'X') {
            buddySystem->access(seqno[i], num[i]);
        } else if (type[i] == 'F') {
            buddySystem->free(seqno[i], num[i]);
        }
        cout << i << " end" << endl;
    }

    cout << "End of the buddy system!" << std::endl;

    buddySystem->printStateOfBuddyAndLRU();
    return 0;
}
```

2.Result

Print the final buddy lists, as well as the final state of the two LRU lists.

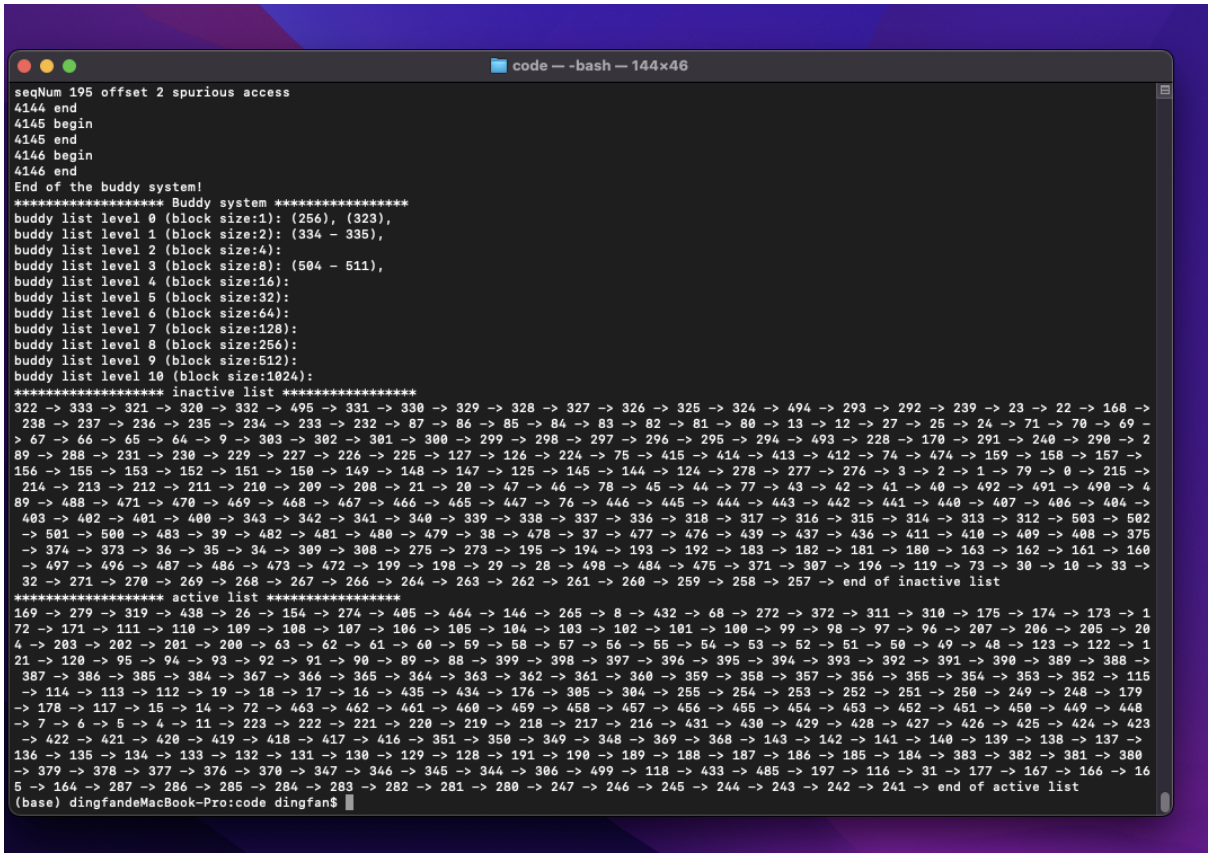
- At the end of the simulation, my simulator is to dump the final buddy lists, as well as the final state of the two LRU lists.
- The following method is used to format and print these out clearly for easy reading.

```
void Buddy_System::printStateOfBuddyAndLRU() {
    cout << "****************** Buddy system *****************" << endl;

    cout << "buddy list level " << 0 << " (block size:" << pow(2, 0) << ")" << ":
 ";
    buddy_system_free_block *temp1 = buddy_system_free_area[0].head;
    while (temp1 != nullptr) {
        cout << "(" << temp1->firstAddress << ")" << ", ";
        temp1 = temp1->next;
    }
    cout << endl;
```

```
    for (int i = 1; i < 11; i++) {
        cout << "buddy list level " << i << " (block size:" << pow(2, i) << ")" <<
": ";
        buddy_system_free_block *temp = buddy_system_free_area[i].head;
        while (temp != nullptr) {
            cout << "(" << temp->firstAddress << " - " << temp->firstAddress + pow
(2, i) - 1 << ")" << ", ";
            temp = temp->next;
        }
        cout << endl;
    }

    cout << "****************** inactive list *****************" << endl;
    LRU_item *temp = inactiveList->listHead;
    while (temp != nullptr) {
        cout << temp->physicalAddress << " -> ";
        temp = temp->next;
    }
    cout << "end of inactive list" << endl;


    cout << "****************** active list *****************" << endl;
    temp = activeList->listHead;
    while (temp != nullptr) {
        cout << temp->physicalAddress << " -> ";
        temp = temp->next;
    }
    cout << "end of active list" << endl;

}
```

Compile and run to show the result

- run `make`
- run `.\main.o`



Section B (15 marks)

B-1. (5 marks) Consider the x86-64 4-level paging with 4Kbyte pages. What is the total amount of storage required if every entry in the page tables at all levels were filled with valid entries? Compare this with a scenario where the process virtual space has only one valid page (at the end of the hierarchy).

solution:

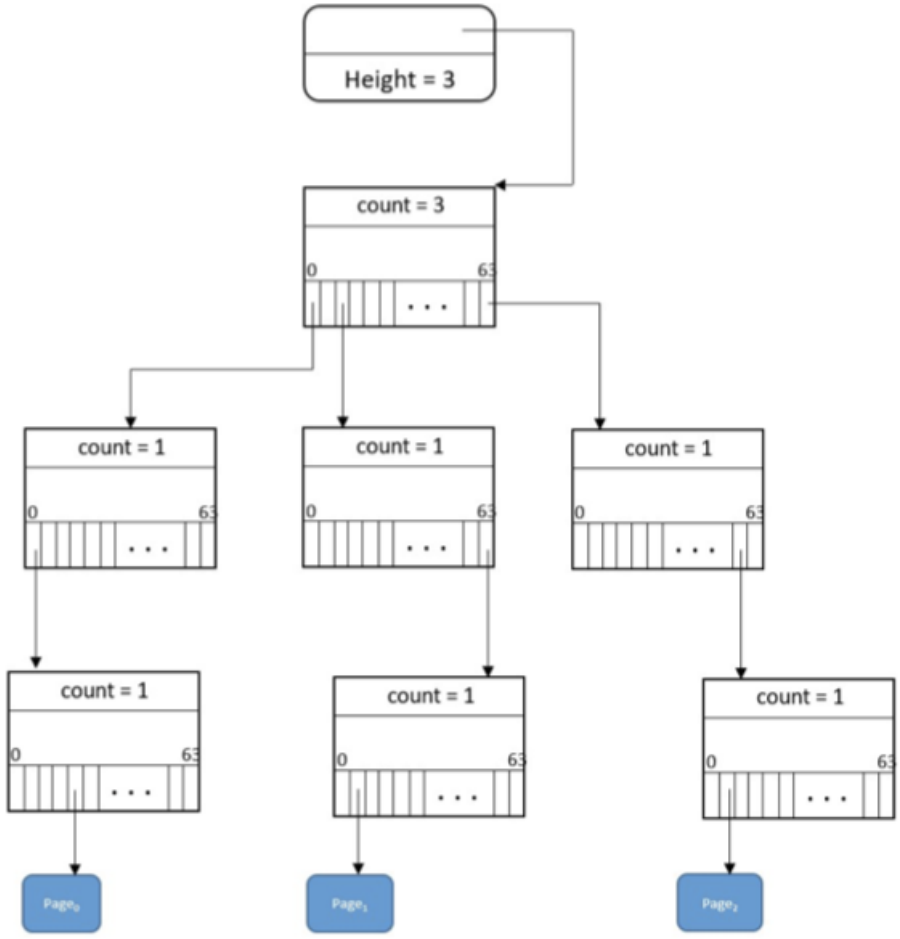
1. if every entry in the page tables at all levels were filled with valid entries

   total amount of storage required $= (1 + 2^9 + 2^9 * 2^9 + 2^9 * 2^9 * 2^9 + 2^9 * 2^9 * 2^9 * 2^9) * 4KB = 256.5TB$

2. compared with a scenario where the process virtual space has only one valid page(at the end of the hierarchy)

   $2^9 * 4 * 8 + 2^8 * 2^4 = 20480byte = 20KB$

B-2. (5 marks) Given the following Linux radix tree, what are the indexes of the three pages (in base 10)? Remember to show your workings.



solution:

- Page0: 5-1=4
- Page1: 64*64*2 + 64*63 +2 -1=12225
- Page2: 646463 + 64*62 +2 -1 =262017

B-3. (5 marks) Consider the following compare-and-exchange atomic instruction (abstracted as a function – in other words, assume that the following function is atomic): `int cas(int *ptr, int oldval, int newval);` If the integer pointed to by `ptr` is equal to `oldval`, then the

content pointed to by `ptr` will be replaced with `newval`, and a '1' will be returned by `cas`. Otherwise, a '0' is returned. Using this function/instruction, show how a spinlock can be implemented by giving the C code for the spin_lock() and spin_unlock() functions. Write down any assumptions that you make.

- Solution:
- Assumptions:
  - When `lock=0`, it is unlocked
  - When `lock=1`,it is locked. When the lock is locked, the lock has the process ID of the lock holder. It is used to check the validation when attempting the unlock operation.
  - The PID of each process is greater than `0`

```
#define UNLOCK 0
typedef int spinlock_t;

int spin_lock(spinlock_t *lock) {
    int pid = (int) getpid();
    while (!cas((int *)lock, UNLOCK, pid))
    return 1;
}

int spin_unlock(spinlock_t *lock) {
    int curval = (int) *lock;
    int pid = (int) getpid();
    if (curval != pid)
      return -1;
    *lock = (spinlock_t) 0;
    return 1;
}
```