Contents

1 Basic Test Results

```
============== Tar Content Test ==============
found README
found Makefile
tar content test PASSED!

============== logins ==============
login names mentioned in file:  brahan,danielabayev
Please make sure that these are the correct login names.

============== make Command Test ==============
g++ -Wall -std=c++11 -g -I.   -c -o VirtualMemory.o VirtualMemory.cpp
ar rv libVirtualMemory.a VirtualMemory.o
a - VirtualMemory.o
ranlib libVirtualMemory.a

ar: creating libVirtualMemory.a

make command test PASSED!

============== Static/Global Variables Test ==============
Static/Global test PASSED!

============== Dynamic Allocation Test ==============
Dynamic Allocation test PASSED!

============== Linking Test ==============


Linking PASSED!

Pre-submission passed!
Keep in mind that this script tests only basic elements of your code.
```

2 README

```
brahan, danielabayev
Brahan Wassan(320455116), Daniel Abayev (206224396)
EX: 4

FILES:
VirtualMemory.cpp - virtual memory library implementation
makefile -- a makefile for the program
README -this file

REMARKS:


ANSWERS:

```

3 Makefile

```makefile
CC=g++
CXX=g++
RANLIB=ranlib

LIBSRC=VirtualMemory.cpp
LIBOBJ=$(LIBSRC:.cpp=.o)

INCS=-I.
CFLAGS = -Wall -std=c++11 -g $(INCS)
CXXFLAGS = -Wall -std=c++11 -g $(INCS)

VIRTUALMEMORYLIB = libVirtualMemory.a
TARGETS = $(VIRTUALMEMORYLIB)

TAR=tar
TARFLAGS=-cvf
TARNAME=ex4.tar
TARSRCS=$(LIBSRC) Makefile README

all: $(TARGETS)

$(TARGETS): $(LIBOBJ)
    $(AR) $(ARFLAGS) $@ $^
    $(RANLIB) $@

clean:
    $(RM) $(TARGETS) $(VIRTUALMEMORYLIB) $(OBJ) $(LIBOBJ) *~ *core

depend:
    makedepend -- $(CFLAGS) -- $(SRC) $(LIBSRC)

tar:
    $(TAR) $(TARFLAGS) $(TARNAME) $(TARSRCS)
```

4 VirtualMemory.cpp

```cpp
#include <cstring>
#include "VirtualMemory.h"
#include "PhysicalMemory.h"


#define INIT_VAL 0
#define EXIT_FAIL 0
#define SUCCESS 1
#define TABLE_ROOT 0
#define EMPTY_VAL 0

void clearTable(uint64_t frameIndex);

/**
 * extract bits from an address from position in length of offsetwidth
 * @param address the address
 * @param offsetWidth the width of bits to extract
 * @param position from where to extract(position from lsb to msb)
 * @return the value of these extracted bits
 */
int extract(uint64_t address, int offsetWidth, int position)
{
    return (((1 << offsetWidth) - 1) & (address >> (position - 1)));
}

/**
 * gets the value of an offset of some page-level
 * @param virtualAddress address
 * @param level which page level(0 to TABLES_DEPTH-1) to get its offset from the address
 * @return the value of these extracted bits from page of level - 'level'
 */
uint64_t getOffset(const uint64_t virtualAddress, int level)
{
    int offsetBits = OFFSET_WIDTH;
    if (level == 0)
    {
        offsetBits = VIRTUAL_ADDRESS_WIDTH - (OFFSET_WIDTH * TABLES_DEPTH);
        return extract(virtualAddress, offsetBits, VIRTUAL_ADDRESS_WIDTH - (offsetBits - 1));
    }
    return extract(virtualAddress, offsetBits, ((TABLES_DEPTH - (level)) * offsetBits + 1));
}

/**
 * saves path to page in 'path' array
 * @param page page to save its path
 * @param path array(size of TABLES_DEPTH for path to page)
 */
void fromPageToPath(uint64_t page, uint64_t *path)
{
    for (int level = 0; level < TABLES_DEPTH; level++)
    {
        auto cur = word_t(page % PAGE_SIZE);
        path[TABLES_DEPTH - 1 - level] = cur;
        page = page >> OFFSET_WIDTH;
    }
}

/**
 * return a page number from its path in the hierarchy
 * @param path
 * @return page number according this path
 */
uint64_t fromPathToPage(const uint64_t *path)
{
    uint64_t pageIdx = 0;
    int offsetWidth = OFFSET_WIDTH;
    for (int tableIdx = 0; tableIdx < TABLES_DEPTH; ++tableIdx)
    {
        uint64_t table = path[tableIdx];
        pageIdx = (pageIdx << offsetWidth) + table;
    }
    return pageIdx;
}

/**
 *
 * @param page the page we want to swap in the physical memory
 * @param checkedPage a page that already in pyhsical memory
 * @return the cyclic distance
 */
uint64_t getDistance(uint64_t page, uint64_t checkedPage)
{
    uint64_t dist = 0;
    uint64_t a = page > checkedPage ? page - checkedPage : checkedPage - page;
    uint64_t b = NUM_PAGES - a;
    dist = a > b ? b : a;
    return dist;
}

/**
 * search to evict page(in case we didn't find empty/unused frame in physical memory)
 * @param pageToSwapIn page we want to swap in
 * @param curFrame current frame in physical memory
 * @param curLevel current level in the page tables hierarchy
 * @param father the 'father' table of the current frame
 * @param curLine current line in fathers' table(frame)
 * @param path path to a page we want to check its cyclic distance from pageToSwapIn
 * @param maxDist max distance so far in the search
 * @param pageToEvict page to evict this far in the search
 * @param frameToEvict frame correlated to page to evict this far in the search
 * @param fatherOfEvicted father table of the current evicted page
 * @param rowInFatherTable the row in fathers' table to unlink the evicted page(frame) from it
 */
void
searchToEvict(uint64_t pageToSwapIn, uint64_t curFrame, uint64_t curLevel, uint64_t father, int curLine, uint64_t *path,
              uint64_t &maxDist, uint64_t &pageToEvict, uint64_t &frameToEvict, uint64_t &fatherOfEvicted,
              int &rowInFatherTable)
{
    if (curLevel == TABLES_DEPTH)
    {
        uint64_t checkedPage = fromPathToPage(path);
        uint64_t dist = getDistance(pageToSwapIn, checkedPage);
        if (dist > maxDist)
        {
            maxDist = dist;
            frameToEvict = curFrame;
            pageToEvict = checkedPage;
            fatherOfEvicted = father;
            rowInFatherTable = curLine;
        }
        return;
    }
    word_t val;
    uint64_t pageSize = PAGE_SIZE;
    if (curFrame == TABLE_ROOT)
    {
        pageSize = 1LL << (VIRTUAL_ADDRESS_WIDTH - (OFFSET_WIDTH * TABLES_DEPTH));
    }
    for (uint64_t i = 0; i < pageSize; ++i)
    {
        PMread(curFrame * pageSize + i, &val);
        if (val != 0)
        {
            path[curLevel] = i;
            searchToEvict(pageToSwapIn, val, curLevel + 1, curFrame, i, path, maxDist, pageToEvict, frameToEvict,
                          fatherOfEvicted,
                          rowInFatherTable);
        }
    }
}

/**
 * search for empty/unused frame
 * @param curFrame current frame in physical memory of page table
 * @param curLevel current level in the page tables hierarchy
 * @param father the 'father' table of the current frame
 * @param frameToProtect in case of searching empty frame for our page table child when we also empty!
 * @param curLine current line in fathers' table(frame)
 * @param emptyFrame updated when we find empty frame
 * @param maxUsedFramePlusOne updated when we find max unused frame in p'ysical memory
 */
void searchWithoutEvict(uint64_t curFrame, uint64_t curLevel, uint64_t father, uint64_t frameToProtect, int curLine,
                        uint64_t &emptyFrame,
                        uint64_t &maxUsedFramePlusOne)
{
    if (curLevel == TABLES_DEPTH)
    {
        maxUsedFramePlusOne = maxUsedFramePlusOne > (curFrame + 1) ? maxUsedFramePlusOne : (curFrame + 1);
        return;
    }
    word_t val = 0;
    uint64_t pageSize = PAGE_SIZE;
    int counterEmptyLines = 0;
    if (curFrame == TABLE_ROOT)
    {
        pageSize = 1LL << (VIRTUAL_ADDRESS_WIDTH - (OFFSET_WIDTH * TABLES_DEPTH));
    }
    for (uint64_t i = 0; i < pageSize; ++i)
    {
        PMread(curFrame * pageSize + i, &val);
        maxUsedFramePlusOne = maxUsedFramePlusOne > (curFrame + 1) ? maxUsedFramePlusOne : (curFrame + 1);
        if (val != 0)
        {
            searchWithoutEvict(val, curLevel + 1, curFrame, frameToProtect, i, emptyFrame, maxUsedFramePlusOne);
        }
        else
        {
            counterEmptyLines++;
        }
    }
    if (counterEmptyLines == (int) pageSize && curFrame != TABLE_ROOT && curFrame != frameToProtect)
    {
        emptyFrame = curFrame;
        PMwrite(father * PAGE_SIZE + curLine, 0);
    }
}

/**
 *
 * @param page page to find its frame in PM
 * @param father father of this page in page table hierarchy
 * @return the frame of this page table in physical memory
 */
uint64_t getFrame(uint64_t page, uint64_t father)
{
    uint64_t frame = 0;
    uint64_t emptyFrame = INIT_VAL;
    uint64_t maxUsedFramePlusOne = INIT_VAL;

    searchWithoutEvict(TABLE_ROOT, 0, 0, father, 0, emptyFrame, maxUsedFramePlusOne);
    if (emptyFrame != INIT_VAL)
    {
        frame = emptyFrame;
    }
    else if (maxUsedFramePlusOne != INIT_VAL)
    {
        if (maxUsedFramePlusOne >= NUM_FRAMES)
        {
            uint64_t maxDistanceFromPage = INIT_VAL;
            uint64_t pageToEvict = INIT_VAL;
            uint64_t frameIndexToEvict = INIT_VAL;
            uint64_t path[TABLES_DEPTH];
            uint64_t fatherOfEvicted = INIT_VAL;
            int rowInFatherTable = INIT_VAL;
            searchToEvict(page, TABLE_ROOT, 0, 0, 0, path, maxDistanceFromPage, pageToEvict, frameIndexToEvict,
                          fatherOfEvicted,
                          rowInFatherTable);
            PMevict(frameIndexToEvict, pageToEvict);
            PMwrite(fatherOfEvicted * PAGE_SIZE + rowInFatherTable, 0);
            frame = frameIndexToEvict;
        }
        else
        {
            frame = maxUsedFramePlusOne;
        }
    }
    return frame;
}

/**
 *
 * @param page page in VM to find its pysical address
 * @return the frame in PM of this page
 */
uint64_t getPhysicalAddress(uint64_t page)
{
    uint64_t pagePath[TABLES_DEPTH];
    fromPageToPath(page, pagePath);
    bool isRestored = false;
    uint64_t father;
    word_t child = EMPTY_VAL;
    for (int curLevel = 0; curLevel < TABLES_DEPTH; curLevel++)
    {
        father = child;
        PMread(father * PAGE_SIZE + pagePath[curLevel], (&child));
        if (child == EMPTY_VAL)
        {
            isRestored = true;
            child = getFrame(page, father);
            if (curLevel != TABLES_DEPTH - 1)
            {
                clearTable(child);
            }
            PMwrite(father * PAGE_SIZE + pagePath[curLevel], child);
        }
    }
    if (isRestored)
    {
        PMrestore(child, page);
    }
    return child;
}

```

```
/**
 * clears page table
 * @param frameIndex
 */
void clearTable(uint64_t frameIndex)
{
    for (uint64_t i = 0; i < PAGE_SIZE; ++i)
    {
        PMwrite(frameIndex * PAGE_SIZE + i, 0);
    }
}

void VMinitialize()
{
    clearTable(0);
}


int VMread(uint64_t virtualAddress, word_t *value)
{
    if (virtualAddress >= VIRTUAL_MEMORY_SIZE)
    {
        return EXIT_FAIL;
    }
    uint64_t frame = getPhysicalAddress((virtualAddress >> OFFSET_WIDTH));
    PMread(frame * PAGE_SIZE + getOffset(virtualAddress, TABLES_DEPTH), value);
    return SUCCESS;
}

int VMwrite(uint64_t virtualAddress, word_t value)
{
    if (virtualAddress >= VIRTUAL_MEMORY_SIZE)
    {
        return EXIT_FAIL;
    }
    uint64_t frame = getPhysicalAddress((virtualAddress >> OFFSET_WIDTH));
    PMwrite(frame * PAGE_SIZE + getOffset(virtualAddress, TABLES_DEPTH), value);
    return SUCCESS;
}

```