과목명: 시스템프로그래밍

담당 교수명: 박운상

<<Assignment 1>>

**서강대학교 컴퓨터공학과**

**20161662 허남규**

Table of Contents

[1.프로그램 개요 3](#_Toc510016498)

[2.프로그램 설명 3](#_Toc510016499)

[2.1.프로그램 흐름도 (Pseudo-code) 3](#_Toc510016500)

[3.모듈 정의 4](#_Toc510016501)

[3.1.dir [dir.h] 4](#_Toc510016502)

[3.1.1. 함수 설명 4](#_Toc510016503)

[3.2.hashtable [hashtable.h] 4](#_Toc510016504)

[3.2.1. 함수 설명 4](#_Toc510016505)

[3.3.help [help.h] 6](#_Toc510016506)

[3.3.1. 함수 설명 6](#_Toc510016507)

[3.4.history [history.h] 6](#_Toc510016508)

[3.4.1. 함수 설명 6](#_Toc510016509)

[3.5.memory [memory.h] 7](#_Toc510016510)

[3.5.1. 함수 설명 7](#_Toc510016511)

[4.전역 변수 8](#_Toc510016512)

[5.특이 사항 8](#_Toc510016513)

[5.1.Inline 함수 8](#_Toc510016514)

[5.2.Macro 정의를 이용한 모듈 테스트 8](#_Toc510016515)

[6.코드 8](#_Toc510016516)

[6.1.main.c 8](#_Toc510016517)

[6.2.dir.h 12](#_Toc510016518)

[6.3.dir.c 12](#_Toc510016519)

[6.4.hashtable.h 13](#_Toc510016520)

[6.5.hashtable.c 14](#_Toc510016521)

[6.6.help.h 16](#_Toc510016522)

[6.7.help.c 16](#_Toc510016523)

[6.8.history.h 16](#_Toc510016524)

[6.9.history.c 17](#_Toc510016525)

[6.10.memory.h 18](#_Toc510016526)

[6.11.memory.c 19](#_Toc510016527)

# 프로그램 개요

이번 프로젝트의 최종 목표는 Linux OS 상에서 C 표준 라이브러리 함수를 기반으로 SIC/XE 가상 머신을 구현하는 것입니다. 이번 프로그램에서는 명령어 입출력, 메모리 조작 및 출력, 그리고 연산자 mnemonic을 opcode 값으로 변환하는 테이블 등 SIC/XE 가상 머신의 기반이 되는 요소들을 구현했습니다.

# 프로그램 설명

## 프로그램 흐름도 (Pseudo-code)



# 모듈 정의

이 프로그램은 OOP 설계 원칙에 따라 모든 기능을 데이터와 그에 대한 함수로 모듈화하여 만들었습니다. 구현 언어가 C 언어로 제한되어 있기 때문에 클래스를 사용하는 대신 각각의 모듈을 파일로 분리했습니다. 파일로 분리된 모듈 내에는 구조체와 new, free 함수, 그리고 해당 구조체 타입을 첫 인자로 갖는 기타 함수를 통해 OOP 방법론을 수용하였습니다.

아래 각각의 모듈에 대해 핵심 구조체와 함수에 관한 설명을 추가했습니다. 내부적인 구현과 변수, 지역 함수 등을 제외하고 헤더에 정의된 요소에 관해서만 서술하였습니다.

**Note.** 모듈 간의 독립성을 향상시키고 향후에 유닛 테스트를 쉽게 진행할 수 있도록, 모든 출력 기능은 output stream을 이용하거나 또는 문자열을 반환하는 형태로 구현했습니다.

## dir [dir.h]

현재 디렉토리를 출력하기 위한 함수를 담고 있는 모듈입니다.

### 함수 설명

void fprint\_dir(FILE \*out);

*out* output stream으로 현재 디렉토리를 출력합니다.

## hashtable [hashtable.h]

연산자 mnemonic – opcode에 해당하는 key – value 쌍을 저장하기 위한 해쉬 테이블을 제공하는 모듈입니다.

### 함수 설명

typedef struct \_HashTable { … } HashTable;

해쉬 테이블 데이터를 담기 위한 구조체입니다.

**Implementation Note.** 내부 데이터는 <<데이터 쌍을 담은 TableNode로 이루어진 linked-list>에 대한 reference를 담은 BucketNode의 배열>로 저장됩니다.

HashTable \*new\_hash\_table();

필요한 메모리를 할당받아 새로운 해쉬 테이블을 초기화하여 반환합니다.

void free\_hash\_table(HashTable \*table);

해쉬 테이블에 쓰인 메모리를 해제합니다.

void add\_to\_hash\_table(HashTable \*table, char \*key, Value value);

해쉬 테이블에 새로운 key – value 쌍을 추가합니다.

int find\_from\_hash\_table(HashTable \*table, char \*key, Value \*value);

해쉬 테이블 내에서 특정 key를 찾아 그와 연결된 값을 pass-by-reference로 반환하고 성공 여부를 반환합니다.

void fprint\_hash\_table(FILE \*out, HashTable \*table);

해쉬 테이블을 지정한 output stream에 출력합니다.

## help [help.h]

이 프로그램에서 사용할 수 있는 명령어에 관한 도움말 출력 기능을 제공하는 모듈입니다.

### 함수 설명

void fprint\_help(FILE \*out);

도움말을 지정된 output stream에 출력합니다.

## history [history.h]

문자열을 저장하여 지금까지 저장된 문자열을 출력하는 기록을 제공하는 모듈로, 이번 프로그램에서는 사용자 입력 기록을 출력하는 history 명령어 구현에 쓰입니다.

### 함수 설명

typedef struct \_History { … } History;

문자열 기록 데이터를 담는 구조체입니다.

**Implementation Note.** 내부 데이터는 크기가 정해진 문자열 (문자 배열)의 linked-list로 저장됩니다.

History \*new\_history();

필요한 메모리를 할당받아 새로운 문자열 기록을 초기화하여 반환합니다.

void free\_history(History \*history);

문자열 기록에 쓰인 메모리를 해제합니다.

void add\_history(History \*history, char \*string);

기록에 새로운 문자열을 추가합니다.

void fprint\_history(FILE \*out, History \*history);

기록된 문자열을 지정된 output stream에 모두 출력합니다.

int has\_history(History \*history);

기록된 문자열이 있는지 반환합니다.

## memory [memory.h]

정해진 크기 (1MB)에 해당하는 메모리를 담을 수 있는 메모리 블록 (Block)에 대한 읽기, 쓰기, 초기화, 출력 기능을 제공하는 모듈입니다.

### 함수 설명

typedef struct \_Block { … } Block;

1MB에 해당하는 메모리를 저장하는 구조체입니다.

**Implementation Note.** 내부 데이터는unsigned char 배열로 저장됩니다.

Block \*new\_memory\_block();

필요한 메모리를 할당받아 새로운 메모리 블록을 초기화하여 반환합니다.

// void free\_memory\_block();

메모리 블록은 별다른 내부 작업 없이 free 함수를 이용하여 메모리를 해제할 수 있습니다.

void set\_memory(Block \*block, int location, unsigned char value);

메모리 블록 내 지정된 주소에 새로운 값을 저장합니다.

void fill\_memory(Block \*block, int start, int end, unsigned char value);

메모리 블록 내 시작 주소부터 끝 주소까지 (inclusive) 새로운 값을 저장합니다.

char \*dump\_memory(Block \*block, int start, int end);

메모리 블록 내 시작 주소부터 끝 주소까지 dump 형식으로 출력된 문자열을 반환합니다.

끝 주소 인자로 -1 값이 넘겨질 경우 끝 주소를 지정하지 않은 것으로 간주하여 시작 주소부터 160개의 값을 출력합니다.

시작 주소와 끝 주소 인자로 모두 -1 값이 넘겨질 경우, 이전까지 출력한 주소 다음부터 160개의 값을 출력합니다.

이때 반환되는 문자열은 동적 할당된 것이므로 사용 후 꼭 메모리를 해제해야 합니다.

void reset\_memory(Block \*block);

메모리 블록 내의 값을 모두 0으로 초기화합니다.

# 전역 변수

이 프로그램은 전역 변수를 사용하지 않도록 설계되었습니다. 이는 모듈 간 독립성을 유지하여 충돌을 방지하고 모듈 단위의 테스트를 쉽게 할 수 있기 위함입니다.

# 특이 사항

## Inline 함수

main.c 및 다른 모듈 내에는 readability 향상을 위한 몇가지 지역 함수가 있습니다. 이들 중 일부는 컴파일러 최적화를 위해 static inline으로 선언했습니다. 해당 기능은 C99 표준에서 소개된 기능이지만 실행 환경인 cspro의 gcc 버젼 (5.4.0)의 기본 C언어 컴파일 표준이 GNU11이기 때문에 별다른 컴파일 옵션을 추가하지 않았습니다.

## Macro 정의를 이용한 모듈 테스트

모든 모듈의 module\_name.c 파일에는 모듈 테스트를 위한 main 함수가 정의되어 있습니다. 이는TEST라는 매크로 정의를 선언해야만 컴파일 단위에 포함이 되도록 IFDEF, ENDIF 전처리문으로 둘러쌓여 있습니다. 해당 main 함수를 이용해서 모듈을 테스트하려면 gcc 컴파일 명령어에 -D TEST 옵션을 추가해야 합니다. 이 경우 main 기호가 다중으로 선언되는 문제로 인해 다른 모듈과 링크를 하여 테스트할 수 없습니다. 모듈 단위의 테스트가 목적이기 때문에 위와 같은 방법으로 전처리기를 사용했습니다.

# 코드

## main.c

#include "dir.h"

#include "hashtable.h"

#include "help.h"

#include "history.h"

#include "memory.h"

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <assert.h>

#define ERROR\_MESSAGE\_LENGTH 2000

#define MAX\_TOKENS 4

#define OPCODE\_FILE "opcode.txt"

#define PROMPT "sicsim> "

static inline int is\_comma(char c);

// Return whether c is a comma. Functionalized for readability.

static inline int hex\_to\_uint(char \*string, unsigned int \*value);

// Convert hex string to uint and save to value. Return success as boolean.

static inline int is\_normal\_letter(char c);

// Return whether c is a normal letter (not comma, space, newline, or tab).

int validate\_and\_remove\_commas(char \*string);

// Return whether the comma sequence is valid and remove commas.

int validate\_and\_tokenize(char \*string, int \*token\_count, char \*\*tokens);

// Return whether the spacing and number of command tokens are valid and

// save the (number of) tokens to token(s)(\_count).

static inline int is\_command(char \*token0, char \*command, char \*alias);

// Return whether the given command tokens correspond to the given command

// or alias.

static inline int get\_line(char \*string);

// Scan one line to string and return whether it is the appropriate length.

void load\_hash\_table(HashTable \*table, FILE \*in);

// Load HashTable for the given opcode input file and save to table.

int main(void) {

History \*history = new\_history();

HashTable \*table = new\_hash\_table();

Block \*block = new\_memory\_block();

char command[COMMAND\_LENGTH];

char processed\_command[COMMAND\_LENGTH];

FILE \*opcode\_in;

if (!(opcode\_in = fopen(OPCODE\_FILE, "r")))

printf("error opening opcode.txt. Continuing without opcodes\n");

else {

load\_hash\_table(table, opcode\_in);

fclose(opcode\_in);

}

while (1) {

printf("%s", PROMPT);

if (!get\_line(command)) {

printf("error: command is too long\n");

continue;

}

strcpy(processed\_command, command);

if (!validate\_and\_remove\_commas(processed\_command)) {

printf("error: command not parsable\n"); continue;

}

int token\_count;

char \*tokens[MAX\_TOKENS];

if (!validate\_and\_tokenize(processed\_command, &token\_count, tokens)) {

printf("error: too many arguments\n"); continue;

}

if (is\_command(tokens[0], "quit", "q"))

return 0;

if (is\_command(tokens[0], "history", "hi")) {

if (token\_count != 1) {

printf("error: history does not take any arguments\n");

continue;

}

int print = has\_history(history);

add\_history(history, command);

if (print) fprint\_history(stdout, history);

continue;

}

if (is\_command(tokens[0], "help", "h")) {

if (token\_count != 1) {

printf("error: help does not take any arguments\n");

continue;

}

fprint\_help(stdout);

} else if (is\_command(tokens[0], "dir", "d")) {

if (token\_count != 1) {

printf("error: dir does not take any arguments\n");

continue;

}

fprint\_dir(stdout);

} else if (is\_command(tokens[0], "dump", "du")) {

int start = -1;

int end = -1;

if (token\_count == 4) {

printf("error: dump takes up to 2 arguments\n");

continue;

}

if (token\_count >= 2) {

unsigned int value;

if (!hex\_to\_uint(tokens[1], &value) || value >= BLOCK\_SIZE) {

printf("error: invalid start address\n");

continue;

}

start = value;

}

if (token\_count >= 3) {

unsigned int value;

if (!hex\_to\_uint(tokens[2], &value) || value >= BLOCK\_SIZE || value < start) {

printf("error: invalid end address\n");

continue;

}

end = value;

}

char \*string = dump\_memory(block, start, end);

printf("%s", string);

free(string);

} else if (is\_command(tokens[0], "edit", "e")) {

if (token\_count != 3) {

printf("error: edit takes exactly 2 arguments\n");

continue;

}

unsigned int address;

if (!hex\_to\_uint(tokens[1], &address) || address >= BLOCK\_SIZE) {

printf("error: invalid address\n");

continue;

}

unsigned int value;

if (!hex\_to\_uint(tokens[2], &value) || value >= 256) {

printf("error: invalid value\n");

continue;

}

set\_memory(block, address, value);

} else if (is\_command(tokens[0], "fill", "f")) {

if (token\_count != 4) {

printf("error: fill takes exactly 3 arguments\n");

continue;

}

unsigned int start;

if (!hex\_to\_uint(tokens[1], &start) || start >= BLOCK\_SIZE) {

printf("error: invalid start address\n"); continue;

}

unsigned int end;

if (!hex\_to\_uint(tokens[2], &end) || end >= BLOCK\_SIZE || end < start) {

printf("error: invalid end address\n"); continue;

}

unsigned int value;

if (!hex\_to\_uint(tokens[3], &value) || value >= 256) {

printf("error: invalid value\n"); continue;

}

fill\_memory(block, start, end ,value);

} else if (!strcmp(tokens[0], "reset")) {

reset\_memory(block);

} else if (!strcmp(tokens[0], "opcode")) {

if (token\_count != 2) {

printf("error: opcode takes exactly 1 argument\n");

continue;

}

Value value;

if (!find\_from\_hash\_table(table, tokens[1], &value)) {

printf("Couldn't find opcode for %s\n", tokens[1]); continue;

} else {

printf("opcode is %02X\n", value.opcode);

}

} else if (!strcmp(tokens[0], "opcodelist")) {

if (token\_count != 1) {

printf("error: opcodelist does not take any arguments\n");

continue;

}

fprint\_hash\_table(stdout, table);

} else {

printf("error: no such command\n");

continue;

}

add\_history(history, command);

}

return 0;

}

static inline int is\_comma(char c) {

return c == ',';

}

static inline int hex\_to\_uint(char \*string, unsigned int \*value) {

int length = strlen(string);

char format\_string[20];

sprintf(format\_string, "%%%dX", length);

if (sscanf(string, format\_string, value) == 1) return 1;

else return 0;

}

static inline int is\_normal\_letter(char c) {

if (c == ' ') return 0;

if (c == '\t') return 0;

if (c == ',') return 0;

if (c == '\n') return 0;

return 1;

}

int validate\_and\_remove\_commas(char \*string) {

int word\_index = 0;

int was\_letter = 0;

int comma\_avail = 0;

for (; \*string; ++string) {

if (is\_normal\_letter(\*string)) {

if (!was\_letter) {

word\_index++;

if (comma\_avail > 0) return 0;

if (word\_index >= 2) comma\_avail = 1;

}

was\_letter = 1;

} else {

was\_letter = 0;

if (is\_comma(\*string)) {

\*string = ' ';

comma\_avail--;

}

if (comma\_avail < 0) return 0;

}

}

return 1;

}

int validate\_and\_tokenize(char \*string, int \*token\_count, char \*\*tokens) {

\*token\_count = 0;

int waiting\_for\_word = 1;

for (; \*string; ++string) {

if (is\_normal\_letter(\*string)) {

if (waiting\_for\_word) {

tokens[(\*token\_count)++] = string;

if (\*token\_count > MAX\_TOKENS) return 0;

waiting\_for\_word = 0;

}

} else {

\*string = '\0';

waiting\_for\_word = 1;

}

}

return 1;

}

static inline int is\_command(char \*token0, char \*command, char \*alias) {

if (!strcmp(token0, command)) return 1;

if (!strcmp(token0, alias)) return 1;

return 0;

}

static inline int get\_line(char \*string) {

char \*p = string;

do {

if (p - string == COMMAND\_LENGTH - 1) return 0;

\*p = getchar();

} while (\*(p++) != '\n');

\*(p - 1) = 0; // places null-char at \n

return 1;

}

void load\_hash\_table(HashTable \*table, FILE \*in) {

unsigned int opcode;

char mnemonic[100];

char operand\_count[100];

while (fscanf(in, "%02X %s %s", &opcode, mnemonic, operand\_count) == 3) {

assert(opcode < 256);

Value value;

value.opcode = opcode;

add\_to\_hash\_table(table, mnemonic, value);

}

}

## dir.h

#include <stdio.h>

void fprint\_dir(FILE \*out);

/\* Description

\* Print the current directory to an output stream

\*/

## dir.c

#include "dir.h"

#include <stdio.h>

#include <dirent.h>

#include <sys/stat.h>

#ifdef TEST

// Manual unit test for this module

// Refer to the README for testing instructions

int main(void) {

fprint\_dir(stdout);

}

#endif

void fprint\_dir(FILE \*out) {

struct dirent \*entry;

DIR \*directory = opendir(".");

// For each file in current directory

while ((entry = readdir(directory))) {

char \*filename = entry->d\_name;

struct stat buffer;

stat(filename, &buffer);

if (filename[0] == '.') continue;

if (entry->d\_type == DT\_DIR) {

fprintf(out, "%s/\t", filename);

continue;

}

fprintf(out, "%s", filename);

if (entry->d\_type == DT\_DIR) fprintf(out, "\\");

else if (buffer.st\_mode & S\_IXUSR) fprintf(out, "\*");

fprintf(out, "\t");

}

fprintf(out, "\n");

}

## hashtable.h

#include <stdio.h>

#define HASH\_TABLE\_SIZE 20

#define HASH\_KEY\_LENGTH 100

typedef struct \_Value {

unsigned char opcode;

} Value;

typedef struct \_BucketNode {

char key[HASH\_KEY\_LENGTH];

Value value;

struct \_BucketNode \*next;

} BucketNode;

typedef struct \_TableNode {

BucketNode \*head; // fake

BucketNode \*last;

} TableNode;

typedef struct \_HashTable {

TableNode nodes[HASH\_TABLE\_SIZE];

} HashTable;

HashTable \*new\_hash\_table();

/\* Description

\* Initiate memory for a valid HashTable.

\*/

void free\_hash\_table(HashTable \*table);

/\* Description

\* Free memory and destory HashTable.

\*/

void add\_to\_hash\_table(HashTable \*table, char \*key, Value value);

/\* Description

\* Add a key-value entry to HashTable.

\*

\* Arguments

\* char \*key: key as a string.

\* Value value: the value to store. To change the fields of the value later

\* on, change the definition of struct Value

\*/

int find\_from\_hash\_table(HashTable \*table, char \*key, Value \*value);

/\* Description

\* Find the value associated with key from table

\*

\* Arguments

\* char \*key: key as string

\* Value \*value: return found value to reference

\*

\* Return

\* A boolean value for whether the key exists

\*/

void fprint\_hash\_table(FILE \*out, HashTable \*table);

/\* Description

\* Prints the hash-table to a human-readable form to an output stream

\*/

## hashtable.c

#include "hashtable.h"

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

/\* Recall these definitions

#define HASH\_TABLE\_SIZE 20

#define HASH\_KEY\_LENGTH 100

typedef struct \_Value {

unsigned char opcode;

} Value;

typedef struct \_BucketNode {

char key[HASH\_KEY\_LENGTH];

Value value;

struct \_BucketNode \*next;

} BucketNode;

typedef struct \_TableNode {

BucketNode \*head; // fake

BucketNode \*last;

} TableNode;

typedef struct \_HashTable {

TableNode nodes[HASH\_TABLE\_SIZE];

} HashTable;

\*/

#ifdef TEST

// Manual unit test for this module

// Refer to the README for testing instructions

int main(void) {

for (int i = 0; i < 10; ++i) {

HashTable \*table = new\_hash\_table();

Value value;

value.opcode = 12;

add\_to\_hash\_table(table, "Hello!", value);

value.opcode = 23;

add\_to\_hash\_table(table, "Hoho!", value);

for (int i = 0; i < 50; ++i) {

char ho[3] = { 0 };

ho[0] = 32 + i;

ho[1] = 32 + i;

Value value;

value.opcode = i;

add\_to\_hash\_table(table, ho, value);

}

printf("Print hash table\n");

fprint\_hash\_table(stdout, table);

printf("\n");

printf("Find non-exisitng\n");

if (find\_from\_hash\_table(table, "Non", &value))

printf("> Found something..? FAIL\n");

else

printf("> Yes! We couldn't find anything\n");

printf("\n");

printf("Find Hello!\n");

if (find\_from\_hash\_table(table, "Hello!", &value))

printf("> Found this: %02X\n", value.opcode);

else

printf("> Couldn't find anything..? FAIL\n");

free\_hash\_table(table);

}

return 0;

}

#endif

// Must free after use!

HashTable \*new\_hash\_table() {

HashTable \*table = malloc(sizeof(HashTable));

for (int i = 0; i < HASH\_TABLE\_SIZE; ++i) {

table->nodes[i].head = malloc(sizeof(BucketNode));

table->nodes[i].head->next = NULL;

table->nodes[i].last = table->nodes[i].head;

}

return table;

}

static inline int hash(char \*key) {

unsigned int hash\_value = 0;

for (; \*key; ++key) {

hash\_value = hash\_value << 1;

hash\_value ^= \*key;

}

return hash\_value % HASH\_TABLE\_SIZE;

}

void free\_hash\_table(HashTable \*table) {

for (int i = 0; i < HASH\_TABLE\_SIZE; ++i) {

BucketNode \*next;

for (BucketNode \*node = table->nodes[i].head; node; node = next) {

next = node->next;

free(node);

}

}

free(table);

}

void add\_to\_hash\_table(HashTable \*table, char \*key, Value value) {

int h = hash(key);

BucketNode \*node = malloc(sizeof(BucketNode));

strcpy(node->key, key);

node->next = NULL;

node->value = value;

table->nodes[h].last->next = node;

table->nodes[h].last = node;

}

int find\_from\_hash\_table(HashTable \*table, char \*key, Value \*value) {

int h = hash(key);

for (BucketNode \*node = table->nodes[h].head->next; node; node = node->next)

if (!(strcmp(key, node->key))) {

\*value = node->value;

return 1;

}

return 0;

}

void fprint\_hash\_table(FILE \*out, HashTable \*table) {

for (int i = 0; i < HASH\_TABLE\_SIZE; ++i) {

fprintf(out, "%d : ", i);

for (BucketNode \*node = table->nodes[i].head->next; node; node = node->next) {

fprintf(out, "[%s,%02X]", node->key, node->value.opcode);

if (node->next) fprintf(out, " -> ");

}

fprintf(out, "\n");

}

}

## help.h

#include <stdio.h>

void fprint\_help(FILE \*out);

/\* Description

\* Print a help dialog to file stream.

\*/

## help.c

#include "help.h"

#include <stdio.h>

#include <stdlib.h>

#ifdef TEST

// Manual unit test for this module

// Refer to the README for testing instructions

int main(void) {

print\_help(stdout);

return 0;

}

#endif

void fprint\_help(FILE \*out) {

const int HELP\_COUNT = 10;

const char \*HELP\_STRINGS[] = {

"h[elp]", "d[ir]",

"q[uit]", "hi[story]",

"du[mp] [start, end]", "e[dit] address, value",

"f[ill] start, end, value", "reset",

"opcode mnemonic", "opcodelist"

};

for (int i = 0; i < HELP\_COUNT; ++i)

fprintf(out, "%s\n", HELP\_STRINGS[i]);

}

## history.h

#include <stdio.h>

#define COMMAND\_LENGTH 200

typedef struct \_HistoryNode {

struct \_HistoryNode \*next;

char string[COMMAND\_LENGTH];

} HistoryNode;

typedef struct \_History {

HistoryNode \*head; // fake head

HistoryNode \*last;

} History;

/\* Description

\* Stores command history as a linked-list.

\*/

History \*new\_history();

/\* Description

\* Initiate memory for a valid history record.

\*/

void free\_history(History \*history);

/\* Description

\* Free memory and destroy history record.

\*/

void add\_history(History \*history, char \*string);

/\* Description

\* Add a previous command to a history record.

\*

\* Arguments

\* char \*string: the command string

\*/

void fprint\_history(FILE \*out, History \*history);

/\* Description

\* Print the history to an output stream.

\*/

int has\_history(History \*history);

/\* Description

\* Return whether there are any commands in the history record.

\*/

## history.c

#include "history.h"

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

/\* Recall these definitions

#define HISTORY\_LENGTH 200

typedef struct \_HistoryNode {

struct \_HistoryNode \*next;

char string[HISTORY\_LENGTH];

} HistoryNode;

typedef struct \_History {

HistoryNode \*head; // fake head

HistoryNode \*last;

} History;

\*/

#ifdef TEST

// Manual unit test for this module

// Refer to the README for testing instructions

int main(void) {

for (int i = 0; i < 20; ++i) {

History \*history = new\_history();

add\_history(history, "Hello!");

add\_history(history, "Yo!");

add\_history(history, "Hello!");

add\_history(history, "Yo!");

add\_history(history, "Hello!");

add\_history(history, "Yo!");

fprint\_history(stdout, history);

free\_history(history);

}

return 0;

}

#endif

// Make sure to free after use!

History \*new\_history() {

History \*history = malloc(sizeof(History));

history->head = malloc(sizeof(HistoryNode));

history->head->next = NULL;

history->last = history->head;

return history;

}

void free\_history(History \*history) {

HistoryNode \*next;

for (HistoryNode \*node = history->head; node; node = next) {

next = node->next;

free(node);

}

free(history);

}

void add\_history(History \*history, char \*string) {

HistoryNode \*new\_node = malloc(sizeof(HistoryNode));

strcpy(new\_node->string, string);

new\_node->next = NULL;

history->last->next = new\_node;

history->last = new\_node;

}

void fprint\_history(FILE \*out, History \*history) {

int i = 1;

for (HistoryNode \*node = history->head->next; node; node = node->next, ++i)

fprintf(out, "%d\t%s\n", i, node->string);

}

int has\_history(History \*history) {

return history->head != history->last;

}

## memory.h

#define BLOCK\_SIZE 1048576

#define MAX\_DUMP\_LENGTH 1000000

typedef struct \_Block {

unsigned char data[BLOCK\_SIZE];

int current;

} Block;

/\* Description

\* A memory block that stores 1 megabyte of unsigned chars.

\*

\* Members

\* unsigned char data[]:

\* int current: the current position used for dump operations

\*/

Block \*new\_memory\_block();

/\* Description

\* Initiate memory for a valid memory block.

\*/

void set\_memory(Block \*block, int location, unsigned char value);

/\* Description

\* Find the value associated with key from table.

\*

\* Arguments

\* char \*key: key as string

\* Value \*value: return found value to reference

\*

\* Return

\* A boolean value for whether the key exists

\*/

void fill\_memory(Block \*block, int start, int end, unsigned char value);

/\* Description

\* Fill the memory block with a given value from start to end (inclusive).

\*/

char \*dump\_memory(Block \*block, int start, int end);

/\* Description

\* Print the values within the block from start to end (inclusive) in a

\* dump format.

\*

\* Return

\* A dynamically allocated string (make sure to free it)

\*/

void reset\_memory(Block \*block);

/\* Description

\* Fill the memory block with 0s.

\*/

## memory.c

#include "memory.h"

#include <assert.h>

#include <stdio.h>

#include <stdlib.h>

/\* Recall this definition

struct Block {

unsigned char data[BLOCK\_SIZE];

int current;

}

\*/

#ifdef TEST

// Manual unit test for this module

// Refer to the README for testing instructions

int main(void) {

Block \*block = new\_memory\_block();

for (int i = 0; i < BLOCK\_SIZE; ++i)

set\_memory(block, i, i % 256);

printf("Dump Test\n");

for (int i = 0; i < 5; ++i) {

char \*dump\_string = dump\_memory(block, -1, -1);

printf("%s", dump\_string);

free(dump\_string);

}

printf("\nDump Range & Offset Test\n");

{

char \*dump\_string = dump\_memory(block, 23, 48);

printf("%s", dump\_string);

free(dump\_string);

}

for (int i = 0; i < 5; ++i) {

char \*dump\_string = dump\_memory(block, -1, -1);

printf("%s", dump\_string);

free(dump\_string);

}

printf("\nOverflow Test\n");

{

char \*dump\_string = dump\_memory(block, BLOCK\_SIZE - 4, -1);

printf("%s", dump\_string);

free(dump\_string);

}

for (int i = 0; i < 5; ++i) {

char \*dump\_string = dump\_memory(block, -1, -1);

printf("%s", dump\_string);

free(dump\_string);

}

printf("\nRange & Overflow Test\n");

{

char \*dump\_string = dump\_memory(block, 0xFFFD7, 0xFFFE0);

printf("%s", dump\_string);

free(dump\_string);

}

for (int i = 0; i < 5; ++i) {

char \*dump\_string = dump\_memory(block, -1, -1);

printf("%s", dump\_string);

free(dump\_string);

}

printf("\nReset Test\n");

reset\_memory(block);

for (int i = 0; i < 4; ++i) {

char \*dump\_string = dump\_memory(block, -1, -1);

printf("%s", dump\_string);

free(dump\_string);

}

free(block);

return 0;

}

#endif

// Make sure to free after use!

Block \*new\_memory\_block() {

Block \*block = malloc(sizeof(Block));

for (int i = 0; i < BLOCK\_SIZE; ++i)

block->data[i] = '\0';

block->current = 0;

return block;

}

void set\_memory(Block \*block, int location, unsigned char value) {

block->data[location] = value;

}

void fill\_memory(Block \*block, int start, int end, unsigned char value) {

// Inclusive, Namgyu's exception rules

if (start < 0) start = 0;

if (BLOCK\_SIZE <= end) end = BLOCK\_SIZE - 1;

for (int i = start; i <= end; i ++)

block->data[i] = value;

}

static inline int sprintf\_digit(char \*string, int i) {

char \*base = string;

int digit\_34 = i / 16 % 256;

int digit\_12 = i / 4096; assert(digit\_12 < 256);

string += sprintf(string, "%02X%02X0 ", digit\_12, digit\_34);

int offset = string - base;

return offset;

}

static inline int sprintf\_data\_hex(

char \*string, Block \*block, int start, int end, int base\_index)

{

char \*base = string;

for (int i = 0; i < 16; ++i) {

int index = base\_index + i;

int value = block->data[index];

if (start <= index && index <= end)

string += sprintf(string, "%02X ", value);

else

string += sprintf(string, " ");

}

int offset = string - base;

return offset;

}

static inline int sprintf\_data\_char(

char \*string, Block \*block, int start, int end, int base\_index)

{

const int X20 = 32;

const int X7E = 126;

char \*base = string;

for (int i = 0; i < 16; ++i) {

int index = base\_index + i;

int value = block->data[index];

if (index < start || end < index || value < X20 || X7E < value)

string += sprintf(string, ".");

else

string += sprintf(string, "%c", value);

}

int offset = string - base;

return offset;

}

// Make sure to free after use!

char \*dump\_memory(Block \*block, int start, int end) {

char \*string = malloc(sizeof(char) \* MAX\_DUMP\_LENGTH);

char \*full\_string = string;

if (start == -1) start = block->current; else assert(start >= 0);

if (end == -1) end = start + 159;

if (end >= BLOCK\_SIZE) end = BLOCK\_SIZE - 1;

block->current = end + 1;

if (block->current == BLOCK\_SIZE) block->current = 0;

for (int i = start / 16 \* 16; i < end / 16 \* 16 + 16; i += 16) {

string += sprintf\_digit(string, i);

string += sprintf\_data\_hex(string, block, start, end, i);

string += sprintf(string, "; ");

string += sprintf\_data\_char(string, block, start, end, i);

string += sprintf(string, "\n");

}

return full\_string;

}

void reset\_memory(Block \*block) {

for (int i = 0; i < BLOCK\_SIZE; ++i)

block->data[i] = 0;

block->current = 0;

}