Міністерство освіти і науки України
Національний технічний університет України
“Київський політехнічний інститут”
Кафедра АСОІУ

ЗВІТ
про виконання комп’ютерного практикуму № 1
з дисципліни
“Операційні системи
Тема: Алокатор пам’яті

Прийняв:
Симоненко А.В.

Виконав:
студент 3-го курсу
гр. ІП-52 ФІОТ
Онбиш Олександр

Київ 2017

1 ОПИС АЛГОРИТМУ

Обраний алгоритм демонструє роботу алокатора пам’яті на основі звязного списку.

Список вміщає у себе вільні блоки, які за необхідність використовуються.

У процесі програми виділяємо пам’ять динамічно. ОС (Linux, etc) збільшує розмір кучі за необхідністю.

Проходить перевірка заданого розміру виділення памяті на степеневість двійки. Тобто, якщо заданий розмір блоку був 10 байтів — то виділиться блок розміром 16 байтів. Якщо 5 байтів — то 8.

Асимптотична складність операцій доступа, виділення та звільнення пам’яті — константне значення (O(1)) , але пошук адреси у звязному списку— лінійна операція (O(n) , де n — кількість адрес у поточному масиві)

Інформаційний блок (META block) має наступні витрати по пам’яті

- 8 байт — під покажчик на наступний блок
- 4 байт - під розмір блоку
- 4 байт — булеве значення (Вільний чи використаний)

Недоліком даного алгоритму є пошук за лінійний час.

Перевагою алгоритму є те, що послідовність декількох блоків, що є вільними у списку, зливаються в один великий блок, задля цілістності блоків, та зберігання менше покажчиків на кожен з блоків.

У разі якщо не вистачає розміру блоку для нововиділеної памяті, то створюється новий блок, і вставляється у кінець даного списку.

2 ПРОГРАМНИЙ КОД

Модуль main.c

```
#include <stdio.h>
#include <string.h>
#include "memoryLib.h"

int main() {
  dump();

  char *testChar = mem_alloc(17);
  int *testInt = mem_alloc(sizeof(int));
  int *testInt_ = mem_alloc(sizeof(int));

  *testInt = 27;
  *testInt_ = 14;
  strcpy(testChar, "hello world!");

  printf("\nNew allocated variables: \n");
  printf("TestChar: %s\n", testChar);
  printf("TestInt: %d\n", *testInt);
  printf("TestInt: %d\n", *testInt_);

  dump();

  mem_free(testInt);

  dump();

  testChar = mem_realloc(testChar, 33);
  strcpy(testChar, "New sentence!");

  printf("TestChar: %s\n", testChar);
  dump();
}
```

Модуль memoryLib.c

```
#include <stdio.h>
#include <unistd.h>
#include <stdbool.h>
#include <string.h>

#include "memoryLib.h"

// describing structure of allocated element (Block)
struct TBlock {
  struct TBlock *next;
  size_t size;
  int free;
};

#define META_SIZE sizeof(struct TBlock)
```

```
typedef struct TBlock TBlock;

// head of list
void *header = NULL;

// check if given size belongs to power of two (e.g 1, 2, 4, 8, 16, 32...)
bool isPowerOfTwo(size_t size) {

 if (size == 0)
  return false;

 while (size != 1)
 {
  if (size%2 != 0)
   return false;

  size = size / 2;
 }

 return true;
}

// compute the next power of two (e.g size=5, function return 8)
size_t getNextPowerOfTwo(size_t size) {

 if (isPowerOfTwo(size)) {
  return size;
 }

 printf("\n(given) Size: %zu\n", size);
 size -= 1;
 size = size | (size >> 1);
 size = size | (size >> 2);
 size = size | (size >> 4);
 size = size | (size >> 8);
 size = size | (size >> 16);
 size = size | (size >> 32);
 size = size | (size >> 64);
 size += 1;
 printf("(changed to) Size: %zu\n", size);
 return size;
}

// find in the list free block, if not found, return null
TBlock *getFreeBlock(TBlock **iterPtr, size_t size) {
 TBlock *current = header;

 while (current && !(current->free && current->size >= size)) {
  *iterPtr = current;
  current = current->next;
 }

 return current;
}

// in case of lack of memory, it allocates new memory
```

```
TBlock *allocateSpace(TBlock *iterPtr, size_t size) {
  // find the program break
  TBlock *block = sbrk(0);

  void *allocatedMemory = sbrk(META_SIZE + size);

  // if it's failed
  if (allocatedMemory == (void*) -1) {
    return NULL;
  }

  // set next block for current block
  if (iterPtr) {
    iterPtr->next = block;
  }

  block->size = size;
  block->next = NULL;
  block->free = 0;

  return block;
}

// split given block if it's possible
TBlock *splitBlock(TBlock *current, size_t size) {

  // if required size equils to the current
  if (size == current->size) {
    return current;
  }

  // split size divided by two
  size_t temp_s = current->size;
  while (temp_s / 2 >= size) {
    temp_s = temp_s / 2;
  }

  // split block and configure relationships
  TBlock *leftBlock = current + size;
  leftBlock->next = current->next;
  leftBlock->size = current->size - size;
  leftBlock->free = 1;

  current->next = leftBlock;
  current->size = size ;

  return current;
}

// allocate memory
void *mem_alloc(size_t size) {

  TBlock *block;

  // align given size
  size = getNextPowerOfTwo(size);
```

```
  if (size <= 0) {
    return NULL;
  }

  // in case of first call
  if (!header) {
    block = allocateSpace(NULL, size);
    if (!block) {
      return NULL;
    }
    header = block;
  } else {
    TBlock *iterPtr = header;
    block = getFreeBlock(&iterPtr, size);

    // if free block was not found, allocate more space,
    // otherwise try to split given free block
    if (!block) {
      block = allocateSpace(iterPtr, size);

      if (!block) {
        return NULL;
      }

    } else {
      // split block
      block = splitBlock(block, size);
      block->free = 0;
    }
  }
  // +1 increments the address by one sizeof(struct(block_meta)).
  return(block+1);
}

// get meta block of current data block
TBlock *getBlock(void *ptr) {
  return (TBlock*)ptr - 1;
}

// merge free blocks if it's possible
void mergeBlocks() {

  TBlock *iterPtr = header;
  TBlock *curr = getFreeBlock(&iterPtr, 0);

  // if there is no free block
  if (!curr) {
    return;
  }

  // if next block is used
  if (!curr->next->free) {
    return;
  }

  TBlock *newBlock = curr;
  size_t total_size = 0;
```

```
  while(curr && (curr->free == 1)) {
    newBlock->next = curr->next;
    total_size += curr->size;
    curr = curr->next;
  }

  newBlock->free = 1;
  newBlock->size = total_size;
}

// free memory
void mem_free(void *addr) {

  if (!addr) {
    return;
  }

  TBlock *block_ptr = getBlock(addr);

  block_ptr->free = 1;

  // merge free blocks
  mergeBlocks();
}

// reallocate memory by given address
void *mem_realloc(void *addr, size_t size) {

  size = getNextPowerOfTwo(size);

  // if size was not given
  if (!size)
    return NULL;

  // free memory by given address
  mem_free(addr);

  // allocate new memory with given size
  void *new_addr = mem_alloc(size);


  // rewrite everything to the new allocated memory
  memmove(new_addr, addr, size);

  return new_addr;
}

// print out current state of memory
void dump(){
  printf("\nSummary of memory:\n");
  TBlock *current = header;
  printf("----------------------\n");
  while (current) {
    printf("Address: %p -- Size: %zu -- Free: %s -- Next Block: %p\n", current, current->size, current->free ? "Yes" : "No", current-
>next);
    current = current->next;
```

```
  }
  printf("----------------------\n");
}
```

Модуль alloclib.h

```
#include <stdlib.h>
#include <stdio.h>

void *mem_alloc(size_t);
void *mem_realloc(void*, size_t);
void mem_free(void *);
void mergeBlocks();
void dump();
```

3 РЕЗУЛЬТАТ ПРОГРАМИ

```
make all
gcc -w main.c memoryLib.c -o output
./output

Summary of memory:
----------------------
----------------------

(given) Size: 17
(changed to) Size: 32

New allocated variables:
TestChar: hello world!
TestInt: 27
TestInt: 14

Summary of memory:
----------------------
Address: 0x10ebbd000 -- Size: 32 -- Free: No -- Next Block: 0x10ebbd038
Address: 0x10ebbd038 -- Size: 4 -- Free: No -- Next Block: 0x10ebbd054
Address: 0x10ebbd054 -- Size: 4 -- Free: No -- Next Block: 0x0
----------------------

Summary of memory:
----------------------
Address: 0x10ebbd000 -- Size: 32 -- Free: No -- Next Block: 0x10ebbd038
Address: 0x10ebbd038 -- Size: 4 -- Free: Yes -- Next Block: 0x10ebbd054
Address: 0x10ebbd054 -- Size: 4 -- Free: No -- Next Block: 0x0
----------------------

(given) Size: 33
(changed to) Size: 64
TestChar: New sentence!

Summary of memory:
----------------------
Address: 0x10ebbd000 -- Size: 36 -- Free: Yes -- Next Block: 0x10ebbd054
Address: 0x10ebbd054 -- Size: 4 -- Free: No -- Next Block: 0x10ebbd070
Address: 0x10ebbd070 -- Size: 64 -- Free: No -- Next Block: 0x0
----------------------
```