Национальный технический университет Украины

«Киевский политехнический институт»

Факультет информатики и вычислительной техники

Кафедра вычислительной техники

Лабораторная работа №1

по курсу «Операционные системы»

Аллокатор памяти общего назначения (часть 1)

Выполнил

студент группы ИВ-73

Захожий Игорь

Киев-2010

Задание на работу

Разработать аллокатор общего назначения, используя за основу описанный выше базовый вариант алгоритма, принимая во внимания следующие условия:

1. Области памяти можно выделять любым доступным способом.
2. Функции mem_alloc(), mem_realloc() и mem_free() должны соответствовать приведенным выше прототипам.
3. Адреса памяти, возвращаемые функциями mem_alloc() и mem_realloc(), должны быть выровнены на границу в 4 байта.
4. Попытаться уменьшить время поиска свободного блока памяти и время освобождения занятого блока.
5. Попытаться уменьшить фрагментацию памяти.

Написать функцию mem_dump(), которая должна выводить на консоль состояние областей памяти.

Выполнение

Описание алгоритма работы функций аллокатора

```
void *mem_alloc(size_t size);
```

1. Выравнивание размера до 4 байт.
2. Поиск подходящего по размеру сводного блока (выделяемый размер и 12 байт на заголовок блока). Если подходящий сводный блок не найден – функция возвращает NULL.
3. Изменение заголовка следующего блока (размер предыдущего блока).
4. Изменение размера свободного блока.
5. Разделение свободного блока (запись заголовка(4 байта – флаг занятости блока, 4 байта – размер блока, 4 байта – размер предыдущего блока) выделенного блока после свободного).
6. Возврат ссылки на выделенный блок.

```
void mem_free(void *ptr);
```

1. Если ссылка равна NULL – возврат.
2. Если блок свободен – возврат.
3. Если предыдущий и следующий блок свободны – склеить их с текущим.
4. Если предыдущий блок свободен, а следующий занят – склеить предыдущий блок с текущим.
5. Если следующий блок свободен, а предыдущий занят – склеить текущий блок со следующим.
6. Иначе пометить текущий блок как свободный.

```
void *mem_realloc(void *ptr, size_t size);
```

1. Если ссылка равна NULL – вызов функции mem_alloc и возврат возвращенной функцией ссылки.
2. Выравнивание размера до 4 байт.
3. Если два соседних блока свободны и суммарный размер трех блоков превышает или равен нужному – изменение заголовка предыдущего блока и ,если суммарный размер трех блоков превышает нужный размер, создание заголовка свободного блока. Изменение заголовка следующего после следующего блока, если такой есть (размер предыдущего блока).
4. Если два соседних блока свободны и суммарный размер трех блоков превышает или равен нужному – изменение заголовка предыдущего блока и ,если суммарный размер трех блоков превышает нужный размер, создание заголовка свободного блока. Изменение заголовка следующего после следующего блока, если такой есть (размер предыдущего блока). Возврат ссылки на предыдущий блок.
5. Если предыдущий блок свободен и суммарный размер двух блоков превышает или равен нужному – изменение заголовка предыдущего блока и ,если суммарный размер двух блоков превышает нужный размер, создание заголовка свободного блока. Изменение заголовка

следующего блока, если такой есть (размер предыдущего блока). Возврат ссылки на предыдущий блок.

6. Если следующий блок свободен и суммарный размер двух блоков превышает или равен нужному – изменение заголовка текущего блока и ,если суммарный размер двух блоков превышает нужный размер, создание заголовка свободного блока. Изменение заголовка следующего после следующего блока, если такой есть (размер предыдущего блока). Возврат ссылки на текущий блок.
7. Если два соседних блока заняты и размер текущего блока равен нужному - возврат ссылки на текущий блок.
8. Если два соседних блока заняты и размер текущего блока меньше нужного – изменение размера текущего блока, создание нового заголовка свободного блока. Изменение заголовка следующего блока (размер предыдущего блока). Возврат ссылки на текущий блок.
9. Иначе вызов функции mem_alloc. Если возвращенная функцией ссылка не равна NULL, копирование данных из старого блока в новый. Возврат ссылки на новый блок.

Оценка времени поиска свободного блока памяти

Поскольку использовался алгоритм линейного поиска, то и время поиска будет тем больше, чем больше размер выделенной памяти. В худшем случае при поиске свободного блока потребуется проверить N заголовков.

Оценка времени освобождения занятого блока

Здесь нет поиска, поэтому освобождение происходит быстро, но время освобождения сильно зависит от варианта развития событий. Если блок окружен пустыми блоками – самое длительное время работы, т.к. необходимо склеивать блоки. Если же блок окружен занятыми блоками – самое короткое время, т.к. блок просто помечается как свободный.

Оценка расхода памяти для хранения служебной информации

В каждом блоке используется 12 байт для хранения служебной информации. Для N блоков потребуется (N*12) байт дополнительной информации.

Описание достоинств и недостатков разработанного аллокатора

Самым существенным недостатком является сильная фрагментация памяти, которая приводит к существенному замедлению работы. Также линейный поиск свободного блока негативно влияет на скорость работы.

Листинг аллокатора памяти общего назначения

Lab1.cpp

```
#include "stdafx.h"
#include "Allocator.h"
#include <iostream>
#include <stdlib.h>
#include <time.h>

using namespace std;

void test(Allocator* allocator, size_t mSize, void* ptrs[], size_t* ptrsCount, size_t iterCount)
{
	srand((unsigned) time(NULL));
	for (int i = 0; i < iterCount; i++)
	{
		cout << "************************************************************\n";
		cout << (i + 1);
		cout << " iteration\n";
		int a;
		if (*ptrsCount == 0)
		{
			a = 0;
		}
		else
```

```
		{
			a = (int) rand() * 3 / RAND_MAX;
			if (a == 3)
			{
				a--;
			}
		}
		switch (a)
		{
			case 0:
			{
				size_t s = (size_t)( rand() * (mSize / 20) / RAND_MAX);
				if (s == 0)
				{
					s++;
				}
				cout << "mem_alloc(";
				cout << s;
				cout << ")\n";
				void* ptr = allocator->mem_alloc(s);
				if (ptr != NULL)
				{
					ptrs[*ptrsCount] = ptr;
					*ptrsCount = *ptrsCount + 1;
				}
				cout << "ptr = ";
				cout << ptr;
				cout << "\n";
				break;
			}
			case 1:
			{
				size_t e = (size_t) (rand() * (*ptrsCount) / RAND_MAX);
				cout << "mem_free(";
				cout << ptrs[e];
				cout << ")\n";
				allocator->mem_free(ptrs[e]);
				for (int i = e + 1; i < *ptrsCount; i++)
				{
					ptrs[i - 1] = ptrs[i];
				}
				*ptrsCount = *ptrsCount - 1;
				break;
			}
			case 2:
			{
				size_t s = (size_t) (rand() * (mSize / 20) / RAND_MAX);
				srand((unsigned) time(NULL));
				size_t e = (size_t) (rand() * (*ptrsCount) / RAND_MAX);
				cout << "mem_realloc(";
				cout << ptrs[e];
				cout << ", ";
				cout << s;
				cout << ")\n";
				void* ptr = allocator->mem_realloc(ptrs[e], s);
				if (ptr != NULL)
				{
					ptrs[e] = ptr;
				}
				cout << "ptr = ";
				cout << ptr;
				cout << "\n";
				break;
			}
		}
		allocator->mem_dump();
		cout << "*********************************************************\n";
	}
}

int _tmain(int argc, _TCHAR* argv[])
{
	const size_t mSize = 4096;
	Allocator allocator(mSize);
	void* ptrs[341];
	size_t ptrsCount = 0;
	int a = -1;
	while (a != 0)
	{
		cout << "*********************************************************\n";
		cout << "Memory size: ";
		cout << mSize;
```

```
cout << "\n";
allocator.mem_dump();
cout << "******************************************************\n";
cout << "Choose the action:\n";
cout << "1 - Test allocator\n";
cout << "2 - Allocate memory\n";
if (ptrsCount != 0)
{
	cout << "3 - Free memory\n";
	cout << "4 - Reallocate memory\n";
}
cout << "0 - Exit\n";
cout << "> ";
cin >> a;
switch (a)
{
	case 1:
	{
		size_t iterCount;
		cout << "Enter the number of iterations.\n";
		cout << "> ";
		cin >> iterCount;
		test(&allocator, mSize, ptrs, &ptrsCount, iterCount);
		break;
	}
	case 2:
	{
		size_t size;
		cout << "Enter the size of memory to allocate.\n";
		cout << "> ";
		cin >> size;
		void* ptr = allocator.mem_alloc(size);
		if (ptr != NULL)
		{
			ptrs[ptrsCount++] = ptr;
		}
		cout << "ptr = ";
		cout << ptr;
		cout << "\n";
		break;
	}
	case 3:
	{
		cout << "Choose block of memory to free:\n";
		for (int i = 0; i < ptrsCount; i++)
		{
			cout << i;
			cout << " - ";
			cout << ptrs[i];
			cout << "\n";
		}
		cout << "> ";
		int n;
		cin >> n;
		if (n < ptrsCount)
		{
			allocator.mem_free(ptrs[n]);
			for (int i = n + 1; i < ptrsCount; i++)
			{
				ptrs[i - 1] = ptrs[i];
			}
			ptrsCount--;
		}
		break;
	}
	case 4:
	{
		cout << "Choose block of memory to reallocate:\n";
		for (int i = 0; i < ptrsCount; i++)
		{
			cout << i;
			cout << " - ";
			cout << ptrs[i];
			cout << "\n";
		}
		cout << "> ";
		int n;
		cin >> n;
		if (n < ptrsCount)
		{
			size_t size;
			cout << "Enter the size of memory to reallocate.\n";
```

```
                                        cout << "> ";
                                        cin >> size;
                                        void* ptr = allocator.mem_realloc(ptrs[n], size);
                                        if (ptr != NULL)
                                        {
                                                ptrs[n] = ptr;
                                        }
                                        cout << "ptr = ";
                                        cout << ptr;
                                        cout << "\n";
                                }
                                break;
                        }
                }
                cin.get();
        }
        return 0;
}
```

Allocator.h

```
#pragma once

class Allocator
{

public:

        Allocator(size_t memorySize);
        ~Allocator();

        void mem_dump();
        void *mem_alloc(size_t size);
        void mem_free(void *ptr);
        void *mem_realloc(void *ptr, size_t size);

private:

        size_t memorySize;
        void *memory;

};
```

Allocator.cpp

```
#include "StdAfx.h"
#include "Allocator.h"
#include <iostream>

using namespace std;

struct header
{
        bool isBusy;
        size_t size;
        size_t prevSize;
};

Allocator::Allocator(size_t memorySize)
{
        this->memorySize = memorySize;
        memory = new char[this->memorySize];
        struct header* head = (header*) memory;
        head->isBusy = false;
        head->prevSize = 0;
        head->size = this->memorySize - sizeof(header);
}

Allocator::~Allocator(void)
{
        delete[] memory;
}

void mem_copy(size_t ptr, size_t ptrc, size_t length)
{
        size_t* p = (size_t*) ptr;
        size_t* cp = (size_t*) ptrc;
        for (int i = 0; i < (length / 4); i++)
        {
                *cp = *p;
                p++;
                cp++;
        }
}
```

```
void Allocator::mem_dump()
{
	cout << "Memory dump (Memory size: ";
	cout << memorySize;
	cout << " bytes):\n";
	struct header* head = (header*) memory;
	int i = 1;
	size_t sum = 0;
	while (head < ((header*) ((size_t) memory + memorySize)))
	{
		cout << "Block #";
		cout << i;
		cout << ":\t";
		if (head->isBusy)
		{
			cout << "Busy";
		}
		else
		{
			cout << "Free";
		}
		cout << "\t";
		cout << head->size;
		cout << " bytes";
		cout << "\t";
		cout << ((void*) ((size_t)head + 12));
		cout << "\n";
		sum = sum + head->size + sizeof(header);
		head = (header*) ((size_t) head + head->size + sizeof(header));
		i++;
	}
	cout << "Real memory size: ";
	cout << sum;
	cout << "\n";
}

void* Allocator::mem_alloc(size_t size)
{
	size_t size4 = size;
	while ((size4 % 4) != 0)
	{
		size4++;
	}
	size_t* ptr = new size_t;
	*ptr = NULL;
	struct header* head = (header*) memory;
	while ((*ptr == NULL) && ((size_t)head < ((size_t) memory + memorySize)))
	{
		while ((((head->isBusy) && ((size_t) head < ((size_t) memory + memorySize))) || ((!head-
>isBusy) && (head->size < size4))) && ((size_t)head < ((size_t) memory + memorySize)))
		{
			head = (header*) ((size_t) head + head->size + sizeof(header));
		}
		if ((size_t)head < ((size_t) memory + memorySize))
		{
			if (head->size >= (size4 + sizeof(header)))
			{
				struct header* newHead = (header*) ((size_t) head + head->size - size4);
				struct header* nextHead = (header*) ((size_t) head + head->size +
sizeof(header));
				if (((size_t) nextHead) < ((size_t) memory) + memorySize)
				{
					nextHead->prevSize = size4;
				}
				head->size = head->size - size4 - sizeof(header);
				newHead->isBusy = true;
				newHead->size = size4;
				newHead->prevSize = head->size;
				*ptr = (size_t) newHead + sizeof(header);
			}
			else
			{
				if (head->size == size4)
				{
					head->isBusy = true;
					*ptr = (size_t) head + sizeof(header);
				}
				else
				{
					head = (header*) ((size_t) head + head->size + sizeof(header));
				}
```

```
                        }
                }
        }
        return (void*) *ptr;
}

void Allocator::mem_free(void *ptr)
{
        if (ptr == NULL)
        {
                return;
        }
        struct header* head = (header*) ((size_t) ptr - sizeof(header));
        if (!head->isBusy)
        {
                return;
        }
        struct header* prevHead = (header*) ((size_t) head - head->prevSize - sizeof(header));
        struct header* nextHead = (header*) ((size_t) head + head->size + sizeof(header));
        if ((((size_t) prevHead) >= ((size_t) memory)) && (((size_t) nextHead) < ((size_t) memory +
memorySize)))
        {
                if ((!prevHead->isBusy) && (!nextHead->isBusy))
                {
                        struct header* nextNextHead = (header*) ((size_t) nextHead + nextHead->size +
sizeof(header));
                        prevHead->size = prevHead->size + head->size + nextHead->size + 2 * sizeof(header);
                        if (((size_t) nextNextHead) < ((size_t) memory + memorySize))
                        {
                                nextNextHead->prevSize = prevHead->size;
                        }
                }
                else
                {
                        if (!prevHead->isBusy)
                        {
                                prevHead->size = prevHead->size + head->size + sizeof(header);
                                nextHead->prevSize = prevHead->size;
                        }
                        else
                        {
                                if (!nextHead->isBusy)
                                {
                                        struct header* nextNextHead = (header*) ((size_t) nextHead + nextHead-
>size + sizeof(header));
                                        head->isBusy = false;
                                        head->size = head->size + nextHead->size + sizeof(header);
                                        if (((size_t) nextNextHead) < ((size_t) memory + memorySize))
                                        {
                                                nextNextHead->prevSize = head->size;
                                        }
                                }
                                else
                                {
                                        head->isBusy = false;
                                }
                        }
                }
        }
        else
        {
                if (((size_t) prevHead) >= ((size_t) memory))
                {
                        if (!prevHead->isBusy)
                        {
                                prevHead->size = prevHead->size + head->size + sizeof(header);
                        }
                        else
                        {
                                head->isBusy = false;
                        }
                }
                else
                {
                        if (((size_t) nextHead) < ((size_t) memory + memorySize))
                        {
                                if (!nextHead->isBusy)
                                {
                                        struct header* nextNextHead = (header*) ((size_t) nextHead + nextHead-
>size + sizeof(header));
                                        head->isBusy = false;
                                        head->size = head->size + nextHead->size + sizeof(header);
```

```
                                        if (((size_t) nextNextHead) < ((size_t) memory + memorySize))
                                        {
                                                nextNextHead->prevSize = head->size;
                                        }
                                }
                                else
                                {
                                        head->isBusy = false;
                                }
                        }
                        else
                        {
                                head->isBusy = false;
                        }
                }
        }
}

void* Allocator::mem_realloc(void *ptr, size_t size)
{
        size_t* rePtr = new size_t;
        *rePtr = NULL;
        if (ptr == NULL)
        {
                *rePtr = (size_t) mem_alloc(size);
        }
        else
        {
                size_t size4 = size;
                while ((size4 % 4) != 0)
                {
                        size4++;
                }
                struct header* head = (header*) ((size_t) ptr - sizeof(header));
                size_t prevSize = head->size;
                if (size4 != head->size)
                {
                        struct header* prevHead = (header*) ((size_t) head - head->prevSize - sizeof(header));
                        struct header* nextHead = (header*) ((size_t) head + head->size + sizeof(header));
                        if ((((size_t) prevHead) >= ((size_t) memory)) && (((size_t) nextHead) < ((size_t)
memory + memorySize)))
                        {
                                if ((!prevHead->isBusy) && (!nextHead->isBusy))
                                {
                                        if ((prevHead->size + head->size + nextHead->size + 2 * sizeof(header))
>= size4)
                                        {
                                                size_t sumSize = prevHead->size + head->size + nextHead->size +
2 * sizeof(header);
                                                prevHead->isBusy = true;
                                                prevHead->size = size4;
                                                *rePtr = (size_t) prevHead + sizeof(header);
                                                struct header* nextNextHead = (header*) ((size_t) nextHead +
nextHead->size + sizeof(header));
                                                size_t copySize;
                                                if (prevSize > size4)
                                                {
                                                        copySize = prevSize;
                                                }
                                                else
                                                {
                                                        copySize = size4;
                                                }
                                                mem_copy((size_t) ptr, *rePtr, copySize);
                                                if (sumSize != size4)
                                                {
                                                        struct header* newHead = (header*) ((size_t) prevHead +
prevHead->size + sizeof(header));
                                                        newHead->isBusy = false;
                                                        newHead->prevSize = prevHead->size;
                                                        newHead->size = sumSize - newHead->prevSize -
sizeof(header);
                                                        if ((size_t) nextNextHead < ((size_t) memory +
memorySize))
                                                        {
                                                                nextNextHead->prevSize = newHead->size;
                                                        }
                                                }
                                                else
                                                {
                                                        if ((size_t) nextNextHead < ((size_t) memory +
memorySize))
```

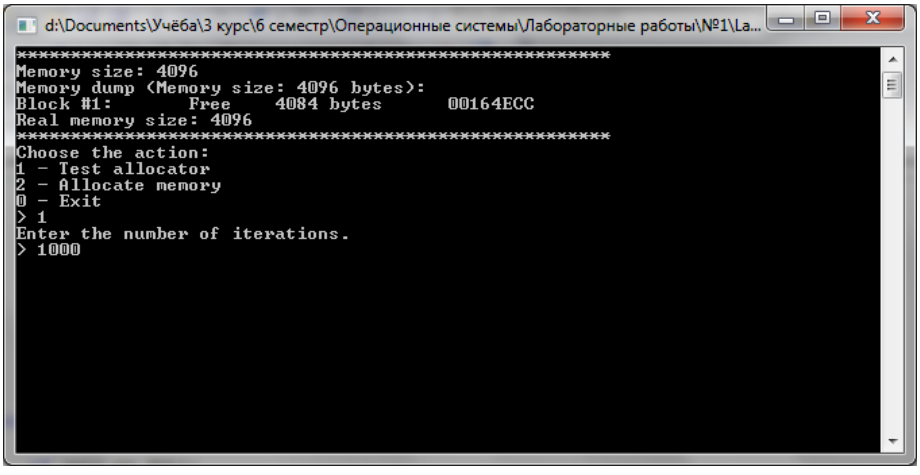
```
                                                {
                                                        nextNextHead->prevSize = prevHead->size;
                                                }
                                        }
                                }
                                else
                                {
                                        *rePtr = (size_t) mem_alloc(size);
                                        if (*rePtr != NULL)
                                        {
                                                mem_copy((size_t) ptr, *rePtr, head->size);
                                                mem_free(ptr);
                                        }
                                }
                        }
                        else
                        {
                                if (!prevHead->isBusy)
                                {
                                        if ((prevHead->size + head->size + sizeof(header)) >= size4)
                                        {
                                                size_t sumSize = prevHead->size + head->size +
sizeof(header);
                                                prevHead->isBusy = true;
                                                prevHead->size = size4;
                                                *rePtr = (size_t) prevHead + sizeof(header);
                                                size_t copySize;
                                                if (prevSize > size4)
                                                {
                                                        copySize = prevSize;
                                                }
                                                else
                                                {
                                                        copySize = size4;
                                                }
                                                mem_copy((size_t) ptr, *rePtr, copySize);
                                                if (sumSize != size4)
                                                {
                                                        struct header* newHead = (header*) ((size_t)
prevHead + prevHead->size + sizeof(header));
                                                        newHead->isBusy = false;
                                                        newHead->prevSize = prevHead->size;
                                                        newHead->size = sumSize - prevHead->size -
sizeof(header);
                                                        nextHead->prevSize = newHead->size;
                                                }
                                                else
                                                {
                                                        nextHead->prevSize = prevHead->size;
                                                }
                                        }
                                        else
                                        {
                                                *rePtr = (size_t) mem_alloc(size);
                                                if (*rePtr != NULL)
                                                {
                                                        mem_copy((size_t) ptr, *rePtr, head->size);
                                                        mem_free(ptr);
                                                }
                                        }
                                }
                                else
                                {
                                        if (!nextHead->isBusy)
                                        {
                                                if ((head->size + nextHead->size + sizeof(header)) >=
size4)
                                                {
                                                        size_t sumSize = head->size + nextHead->size +
sizeof(header);
                                                        head->size = size4;
                                                        *rePtr = (size_t) head + sizeof(header);
                                                        struct header* nextNextHead = (header*) ((size_t)
nextHead + nextHead->size + sizeof(header));
                                                        if (sumSize != size4)
                                                        {
                                                                struct header* newHead = (header*)
((size_t) head + head->size + sizeof(header));
                                                                newHead->isBusy = false;
                                                                newHead->prevSize = head->size;
                                                                newHead->size = sumSize - head->size -
sizeof(header);
```

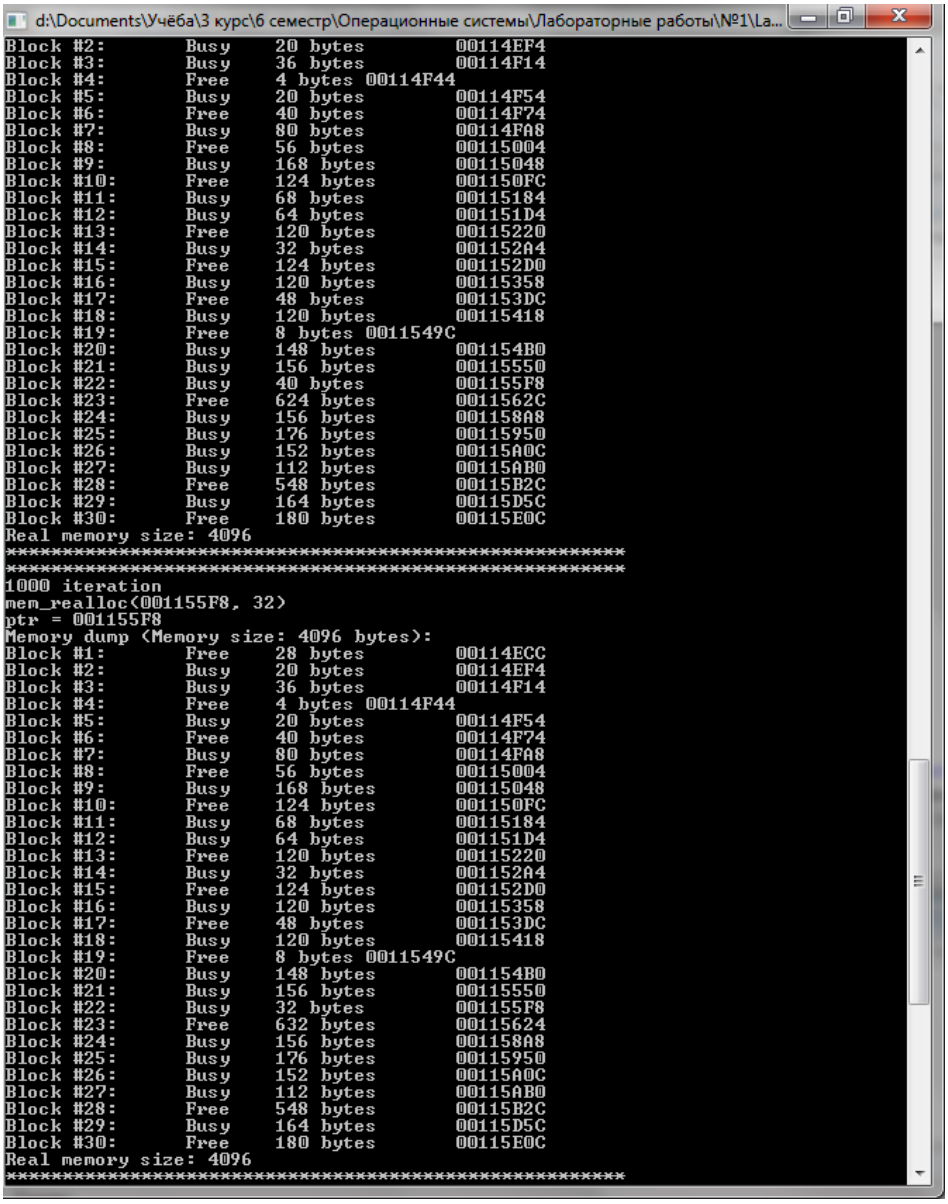
```
                                                        if (((size_t) nextNextHead) < ((size_t)
memory + memorySize))
                                                        {
                                                                nextNextHead->prevSize = newHead-
>size;
                                                        }
                                                }
                                                else
                                                {
                                                        if (((size_t) nextNextHead) < ((size_t)
memory + memorySize))
                                                        {
                                                                nextNextHead->prevSize = head-
>size;
                                                        }
                                                }
                                        }
                                        else
                                        {
                                                *rePtr = (size_t) mem_alloc(size);
                                                if (*rePtr != NULL)
                                                {
                                                        mem_copy((size_t) ptr, *rePtr, head-
>size);
                                                        mem_free(ptr);
                                                }
                                        }
                                }
                                else
                                {
                                        if (size4 > head->size)
                                        {
                                                *rePtr = (size_t) mem_alloc(size);
                                                if (*rePtr != NULL)
                                                {
                                                        mem_copy((size_t) ptr, *rePtr, head-
>size);
                                                        mem_free(ptr);
                                                }
                                        }
                                        else
                                        {
                                                struct header* newHead = (header*) ((size_t) head
+ size4 + sizeof(header));
                                                newHead->isBusy = false;
                                                newHead->size = head->size - size4 -
sizeof(header);
                                                head->size = size4;
                                                newHead->prevSize = head->size;
                                                nextHead->prevSize = newHead->size;
                                                *rePtr = (size_t) ptr;
                                        }
                                }
                        }
                }
                else
                {
                        if (((size_t) prevHead) >= ((size_t) memory))
                        {
                                if (!prevHead->isBusy)
                                {
                                        if ((prevHead->size + head->size + sizeof(header)) >= size4)
                                        {
                                                size_t sumSize = prevHead->size + head->size +
sizeof(header);
                                                prevHead->isBusy = true;
                                                prevHead->size = size4;
                                                *rePtr = (size_t) prevHead + sizeof(header);
                                                size_t copySize;
                                                if (prevSize > size4)
                                                {
                                                        copySize = prevSize;
                                                }
                                                else
                                                {
                                                        copySize = size4;
                                                }
                                                mem_copy((size_t) ptr, *rePtr, copySize);
                                                if (sumSize != size4)
                                                {
```

```
                                                        struct header* newHead = (header*) ((size_t)
prevHead + prevHead->size + sizeof(header));
                                                        newHead->isBusy = false;
                                                        newHead->prevSize = prevHead->size;
                                                        newHead->size = sumSize - prevHead->size -
sizeof(header);
                                                }
                                        }
                                        else
                                        {
                                                *rePtr = (size_t) mem_alloc(size);
                                                if (*rePtr != NULL)
                                                {
                                                        mem_copy((size_t) ptr, *rePtr, head->size);
                                                        mem_free(ptr);
                                                }
                                        }
                                }
                                else
                                {
                                        if (size4 > head->size)
                                        {
                                                *rePtr = (size_t) mem_alloc(size);
                                                if (*rePtr != NULL)
                                                {
                                                        mem_copy((size_t) ptr, *rePtr, head->size);
                                                        mem_free(ptr);
                                                }
                                        }
                                        else
                                        {
                                                struct header* newHead = (header*) ((size_t) head + size4
+ sizeof(header));
                                                newHead->isBusy = false;
                                                newHead->size = head->size - size4 - sizeof(header);
                                                head->size = size4;
                                                newHead->prevSize = head->size;
                                                *rePtr = (size_t) ptr;
                                        }
                                }
                        }
                        else
                        {
                                if (((size_t) nextHead) < ((size_t) memory + memorySize))
                                {
                                        if (!nextHead->isBusy)
                                        {
                                                if ((head->size + nextHead->size + sizeof(header)) >=
size4)
                                                {
                                                        size_t sumSize = head->size + nextHead->size +
sizeof(header);
                                                        head->size = size4;
                                                        *rePtr = (size_t) head + sizeof(header);
                                                        struct header* nextNextHead = (header*) ((size_t)
nextHead + nextHead->size + sizeof(header));
                                                        if (sumSize != size4)
                                                        {
                                                                struct header* newHead = (header*)
((size_t) head + head->size + sizeof(header));
                                                                newHead->isBusy = false;
                                                                newHead->prevSize = head->size;
                                                                newHead->size = sumSize - head->size -
sizeof(header);
                                                                if (((size_t) nextNextHead) < ((size_t)
memory + memorySize))
                                                                {
                                                                        nextNextHead->prevSize = newHead-
>size;
                                                                }
                                                        }
                                                        else
                                                        {
                                                                if (((size_t) nextNextHead) < ((size_t)
memory + memorySize))
                                                                {
                                                                        nextNextHead->prevSize = head-
>size;
                                                                }
                                                        }
                                                }
                                                else
```

```
                                                {
                                                        *rePtr = (size_t) mem_alloc(size);
                                                        if (*rePtr != NULL)
                                                        {
                                                                mem_copy((size_t) ptr, *rePtr, head-
>size);
                                                                mem_free(ptr);
                                                        }
                                                }
                                        }
                                        else
                                        {
                                                *rePtr = (size_t) mem_alloc(size);
                                                if (*rePtr != NULL)
                                                {
                                                        mem_copy((size_t) ptr, *rePtr, head->size);
                                                        mem_free(ptr);
                                                }
                                        }
                                }
                                else
                                {
                                        if (size4 > head->size)
                                        {
                                                *rePtr = NULL;
                                        }
                                        else
                                        {
                                                struct header* newHead = (header*) ((size_t) head + size4
+ sizeof(header));
                                                newHead->prevSize = size4;
                                                newHead->isBusy = false;
                                                newHead->size = head->size - size4 - sizeof(header);
                                                head->size = size4;
                                        }
                                }
                        }
                }
                else
                {
                        *rePtr = (size_t) ptr;
                }
        }
        return (void*) *rePtr;
}
```

Пример работы аллокатора





Выводы

В результате выполнения данной работы был написан простейший аллокатор памяти. Он реализует три основных функции: добавление, освобождение и изменение размеров блоков памяти. Использован линейный поиск свободного блока. В результате длительной работы аллокатора наблюдается фрагментация памяти. Аллокатор также имеет режим тестирования. В этом режиме производится N итераций, в которых случайным образом вызываются функции аллокатора со случайными параметрами и выводится дамп памяти. Управление производится через консольный интерфейс.