Учреждение образования

«БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ»

Кафедра интеллектуальных информационных технологий

Отчет по лабораторной работе №4

по курсу «Операционные системы»

по теме «Изучение алгоритмов изучения памяти»

|  |  |
| --- | --- |
| Выполнили студенты группы 321701: | Самович В. М.  Мотолянец К. А. |
| Проверил: | Сальников Д. А. |

Минск 2024

Лабораторная работа №4

«Изучение алгоритмов распределения памяти»

Вариант 5

Цель работы: Изучить алгоритмы распределения памяти и особенности их реализации.

Задание: Реализовать менеджер памяти со странично-сегментным разбиением.

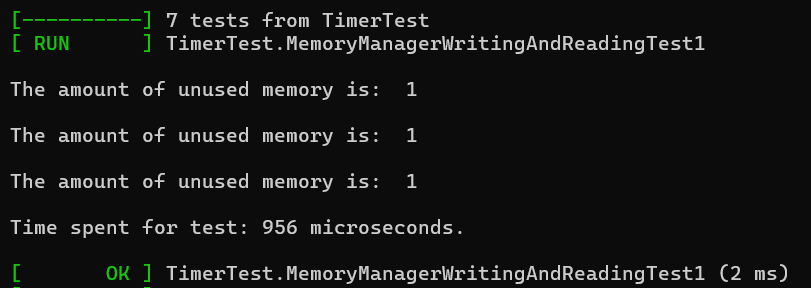
Описание метода странично-сегментного разбиения памяти

Метод странично-сегментного разбиения памяти представляет собой комбинацию страничного и сегментного распределения памяти и, вследствие этого, сочетает в себе достоинства обоих подходов. Виртуальное пространство процесса делится на сегменты, а каждый сегмент в свою очередь делится на виртуальные страницы, которые нумеруются в пределах сегмента.

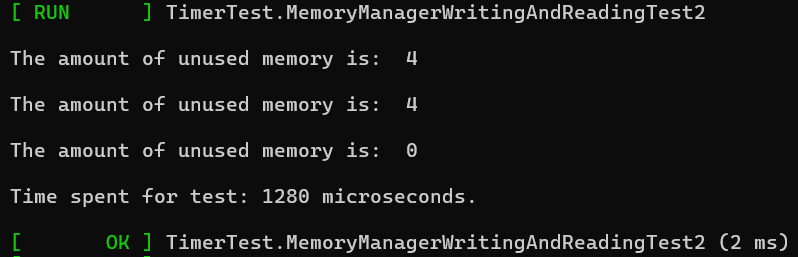
Структуры, реализованные для выполнения задачи

1. Структура Page – описывает страницу в контексте странично-сегментного разбиения памяти.
2. Структура Segment – описывает сегмент в контексте странично-сегментного разбиения памяти.
3. Структура List для описания примитивного списка для создания таблиц страниц и таблиц сегментов.
4. Структура Location для хранения логического адреса, выделенного блока памяти.
5. Структура MemoryManager для описания тестового интерфейса взаимодействия пользователя с системой разбиения памяти.

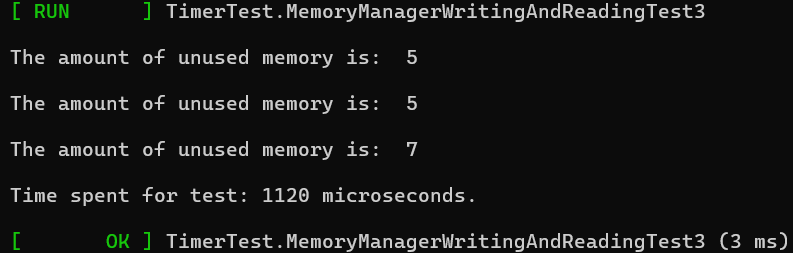
Тестирование



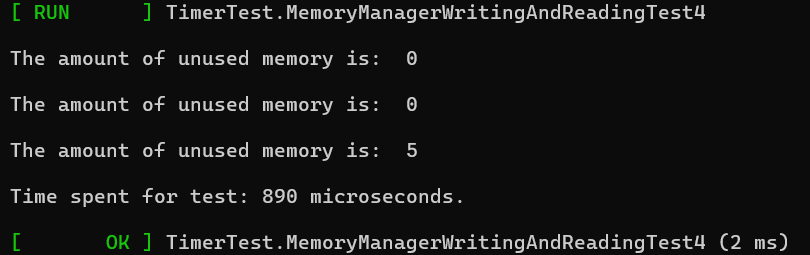
Тест 1



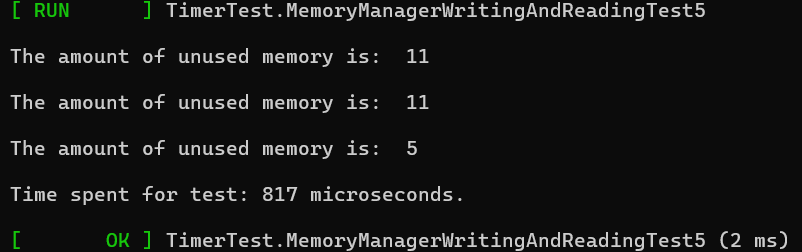
Тест 2



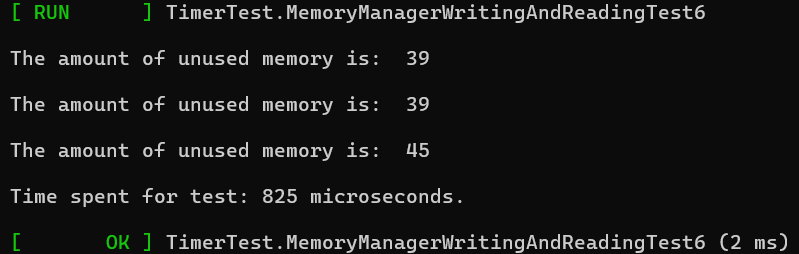
Тест 3



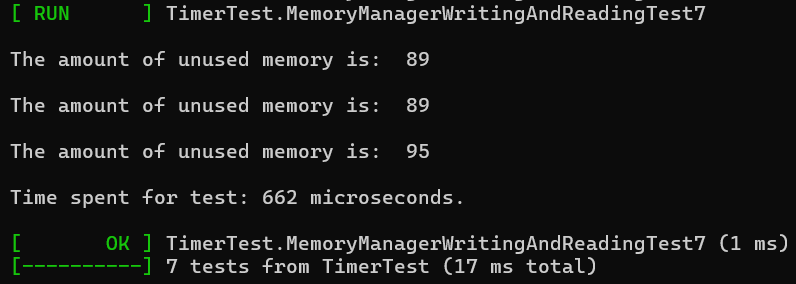
Тест 4



Тест 5



Тест 6



Тест 7

Результаты тестов

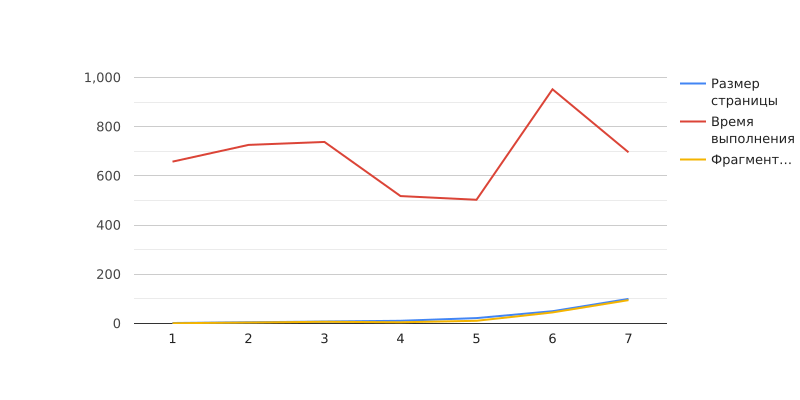


Рисунок 1. Результаты выполнения тестов

На рисунке 1 по оси X – номера тестов, по оси Y – численное значение результатов теста. Синим цветом обозначен график размера страницы. Жёлтым – фрагментация (максимальная за время выполнения теста), которая наблюдается в тестовом примере. Красным – время выполнения в микросекундах.

Выводы

1. Странично-сегментное разбиение памяти позволяет практически избавится как от внутренней, так и от внешней сегментации.
2. Время выполнения тестов с различным размером и количеством страниц и сегментов приблизительно совпадает. Это связано с тем, что сложность вычислений при нахождении адреса необходимого блока не изменятся от размеров менеджера памяти.
3. Сложность алгоритма вычисления адреса блока памяти значительно сложнее, чем у чисто сегментного или чисто страничного способов разбиения памяти. Также сложнее реализация данного алгоритма.

Листинг кода

#include <stdlib.h>

#include <stdio.h>

#include <memory.h>

const bool ENABLE\_WARNINGS = true;

struct Segment;

struct Page;

struct List

{

List\* Next;

unsigned int Id;

void\* Content;

void Free()

{

if (Content) free(Content);

}

void InitializeList(unsigned int id, void\* content)

{

Id = id;

Content = content;

Next = NULL;

}

void AddListItem(List\* list, void\* nextItemContent)

{

List\* iterator = list;

while (iterator->Next)

{

iterator = iterator->Next;

}

List\* newListItem = (List\*)malloc(sizeof(List));

if (newListItem != NULL)

{

newListItem->Next = NULL;

newListItem->Content = nextItemContent;

newListItem->Id = iterator->Id + 1;

}

else exit(EXIT\_FAILURE);

iterator->Next = newListItem;

}

bool RemoveListItemById(List\*\* list, int itemId)

{

List\* iterator = \*list;

if (itemId == (\*list)->Id)

{

\*list = (\*list)->Next;

free(iterator);

return true;

}

while (iterator)

{

if (iterator->Next && iterator->Next->Id == itemId)

{

List\* toRemove = iterator->Next;

iterator->Next = iterator->Next->Next;

free(toRemove);

return true;

}

iterator = iterator->Next;

}

return false;

}

void\* FindListItemContentById(List\* list, int itemId)

{

List\* iterator = list;

while (iterator)

{

if (iterator->Id == itemId) return iterator->Content;

iterator = iterator->Next;

}

return NULL;

}

Page\* FindPageInListById(List\* list, unsigned int id)

{

void\* searchResult = FindListItemContentById(list, id);

if (searchResult) return (Page\*)searchResult;

return NULL;

}

Segment\* FindSegmentInListById(List\* list, unsigned int id)

{

void\* searchResult = FindListItemContentById(list, id);

if (searchResult) return (Segment\*)searchResult;

return NULL;

}

void PrintListItemsIndexies(List\* list)

{

List\* iterator = list;

printf("Id in list: ");

while (iterator)

{

printf("%i ", iterator->Id);

iterator = iterator->Next;

}

printf("\n");

}

};

struct Location

{

unsigned int SegmentId;

unsigned int PageId;

unsigned int Offset;

unsigned int Size;

void Initialize(unsigned int segmentId, unsigned int pageId, unsigned int offset, unsigned int size)

{

SegmentId = segmentId;

PageId = pageId;

Offset = offset;

Size = size;

}

};

struct Page

{

void\* StartAddr;

unsigned int SegmentId;

unsigned int PageId;

unsigned int Offset;

unsigned int Size;

bool IsEmpty;

List\* Locations;

void Initialize(unsigned int size, unsigned int segmentId, unsigned int pageId)

{

Size = size;

StartAddr = malloc(Size);

Offset = 0;

IsEmpty = true;

Locations = NULL;

SegmentId = segmentId;

PageId = pageId;

if (StartAddr == NULL)

{

if (ENABLE\_WARNINGS) printf("%s \n", "Unable to allocate memory for the page.");

}

}

void Free()

{

free(StartAddr);

free(Locations);

}

Location\* WriteData(void\* sourceAddr, unsigned int size)

{

if (Offset + size > Size)

{

if (ENABLE\_WARNINGS) printf("%s %i %s %i %s %i \n", "Not enough space to write the data. ", Offset, " + ", size, " > ", Size);

return NULL;

}

void\* destinationAddr = (char\*)StartAddr + Offset;

memcpy(destinationAddr, sourceAddr, size);

Location\* location = (Location\*) malloc(sizeof(Location));

location->Initialize(SegmentId, PageId, Offset, size);

Offset += size;

IsEmpty = false;

if (Locations)

{

Locations->AddListItem(Locations, location);

return location;

}

Locations = (List\*)malloc(sizeof(List));

Locations->InitializeList(1, location);

return location;

}

bool ReadData(unsigned int readOffset, unsigned int size, void\* destinationAddr)

{

if (readOffset + size > Size)

{

if (ENABLE\_WARNINGS) printf("%s %i %s %i %s %i \n", "Unable to read data outside of page. ", readOffset, " + ", size, " > ", Size);

return false;

}

void\* sourceAddr = (char\*) StartAddr + readOffset;

memcpy(destinationAddr, sourceAddr, size);

return true;

}

bool ReadData(Location\* location, void\* destinationAddr)

{

return ReadData(location->Offset, location->Size, destinationAddr);

}

bool FreePart(unsigned int fragmentOffset, unsigned int fragmentSize)

{

if (fragmentOffset + fragmentSize > Size)

{

if (ENABLE\_WARNINGS) printf("%s %i %s %i %s %i \n", "Unable to read data outside of page. ", fragmentOffset, " + ", fragmentSize, " > ", Size);

return false;

}

void\* destinationAddr = (char\*) StartAddr + fragmentOffset;

void\* sourceAddr = (char\*) destinationAddr + fragmentSize;

unsigned int endPartSize = Size - fragmentSize - fragmentOffset;

memcpy(destinationAddr, sourceAddr, endPartSize);

Offset -= fragmentSize;

DeleteLocation(fragmentOffset);

return true;

}

void DeleteLocation(unsigned int offset)

{

List\* iterator = Locations;

List\* next = iterator->Next;

bool wasRemoved = false;

unsigned int removedSize = 0;

while (iterator)

{

if (wasRemoved)

{

((Location\*)iterator)->Offset -= removedSize;

}

else if (((Location\*)iterator)->Offset == offset)

{

removedSize = ((Location\*)iterator)->Size;

Locations->RemoveListItemById(&Locations, iterator->Id);

iterator = next;

wasRemoved = true;

}

if (!next) return;

iterator = iterator->Next;

next = next->Next;

}

}

void\* GetFirstFree()

{

return (char\*) StartAddr + Offset;

}

unsigned int CountUnusedMemory()

{

return Size - Offset;

}

bool HasUnusedMemory()

{

if (Offset == Size) return false;

return true;

}

void PrintPageReport()

{

printf("%s %i %s %i %s %i \n", "Page ", PageId, " has used ", Offset, " out of ", Size);

}

};

struct Segment

{

List\* Pages;

unsigned int SegmentId;

unsigned int LastPageId;

unsigned int PageSize;

unsigned int PageCount;

unsigned int MaxPageCount;

void Initialize(unsigned int segmentId, unsigned int pageSize, unsigned int maxPageCount)

{

SegmentId = segmentId;

Pages = NULL;

LastPageId = 0;

PageSize = pageSize;

PageCount = 0;

MaxPageCount = maxPageCount;

AddPage();

}

void AddPage()

{

LastPageId++;

Page\* page = (Page\*) malloc(sizeof(Page));

page->Initialize(PageSize, SegmentId, LastPageId);

PageCount++;

if (!Pages)

{

Pages = (List\*) malloc(sizeof(List));

Pages->InitializeList(LastPageId, page);

return;

}

Pages->AddListItem(Pages, page);

}

bool HasEnoughPlaceToWriteData(unsigned int size)

{

List\* iterator = Pages;

unsigned int potentialSpace = 0;

while (iterator)

{

potentialSpace += ((Page\*)iterator)->CountUnusedMemory();

iterator = iterator->Next;

}

if (potentialSpace < size) return false;

return true;

}

unsigned int NeededSize(unsigned int remainedSize, unsigned int memoryLeft)

{

if (memoryLeft > remainedSize) return remainedSize;

return memoryLeft;

}

List\* WriteData(void\* sourceAddr, unsigned int size)

{

List\* destinationLocations = (List\*)malloc(sizeof(List));

unsigned int listId = 1;

List\* iterator = Pages;

unsigned int writtenSize = 0;

while (iterator && writtenSize < size)

{

Page\* page = (Page\*)iterator->Content;

if (!page->HasUnusedMemory())

{

iterator = iterator->Next;

continue;

}

int neededSize = NeededSize(size - writtenSize, page->CountUnusedMemory());

Location\* newLocation = page->WriteData((char\*) sourceAddr + writtenSize, neededSize);

if (writtenSize == 0)

{

destinationLocations->InitializeList(listId, newLocation);

writtenSize += neededSize;

iterator = iterator->Next;

continue;

}

destinationLocations->AddListItem(destinationLocations, newLocation);

writtenSize += neededSize;

iterator = iterator->Next;

}

if (writtenSize == size) return destinationLocations;

int neededSize = size - writtenSize;

int numberOfPages = neededSize / PageSize;

if (neededSize % PageSize != 0) numberOfPages++;

for (int i = 0; i < numberOfPages; i++) AddPage();

iterator = Pages;

while (iterator)

{

Page\* page = (Page\*)iterator->Content;

if (!page->HasUnusedMemory())

{

iterator = iterator->Next;

continue;

}

neededSize = NeededSize(size - writtenSize, page->CountUnusedMemory());

Location\* newLocation = page->WriteData((char\*) sourceAddr + writtenSize, neededSize);

if (writtenSize == 0)

{

destinationLocations->InitializeList(listId, newLocation);

writtenSize += neededSize;

iterator = iterator->Next;

continue;

}

destinationLocations->AddListItem(destinationLocations, newLocation);

writtenSize += neededSize;

iterator = iterator->Next;

}

return destinationLocations;

}

unsigned int CalculateBlockSize(List\* locations)

{

int size = 0;

while (locations)

{

size += ((Location\*)(locations->Content))->Size;

locations = locations->Next;

}

return size;

}

void ReadData(List\* locations, void\* destinationAddr)

{

int blockSize = CalculateBlockSize(locations);

void\* temp = malloc(blockSize);

int offset = 0;

while (locations)

{

Location\* location = (Location\*)(locations->Content);

Page\* page = Pages->FindPageInListById(Pages, location->PageId);

page->ReadData(location, (char\*) temp + offset);

offset += location->Size;

locations = locations->Next;

}

if (temp) memcpy(destinationAddr, temp, offset);

}

void RemoveData(List\* locations)

{

while (locations)

{

Location\* location = (Location\*)(locations->Content);

Page\* page = Pages->FindPageInListById(Pages, location->PageId);

page->FreePart(location->Offset, location->Size);

if (page->Offset == 0) Pages->RemoveListItemById(&Pages, page->PageId);

locations = locations->Next;

}

}

void FreeAllPages()

{

List\* next;

while (Pages)

{

next = Pages->Next;

((Page\*)Pages->Content)->Free();

Pages = next;

}

}

unsigned int CalculateUnusedMemory()

{

unsigned int unusedMemory = 0;

List\* iterator = Pages;

while (iterator)

{

unusedMemory += ((Page\*)iterator->Content)->CountUnusedMemory();

iterator = iterator->Next;

}

return unusedMemory;

}

void Free()

{

FreeAllPages();

Pages->Free();

}

};

struct MemoryManager

{

List\* Segments;

unsigned int LastSegmentId;

unsigned int PageSize;

unsigned int MaxPageCount;

unsigned int MaxSegmentCount;

void Initialize(unsigned int pageSize, unsigned int maxPageCount, unsigned int maxSegmentCount)

{

LastSegmentId = 1;

PageSize = pageSize;

MaxPageCount = maxPageCount;

MaxSegmentCount = maxSegmentCount;

Segment\* segment = (Segment\*) malloc(sizeof(Segment));

segment->Initialize(LastSegmentId, PageSize, MaxPageCount);

Segments = (List\*) malloc(sizeof(List\*));

Segments->InitializeList(LastSegmentId, segment);

}

List\* WriteData(unsigned int segmentId, void\* sourceAddr, unsigned int size)

{

Segment\* foundSegment = Segments->FindSegmentInListById(Segments, segmentId);

if (!foundSegment)

{

printf("%s \n", "No segment exist with such Id.");

return NULL;

}

if (!foundSegment->HasEnoughPlaceToWriteData(size))

{

printf("%s \n", "Found segment can't resize to satisfy memory requirements.");

return NULL;

}

return foundSegment->WriteData(sourceAddr, size);

}

void AddSegment()

{

if (MaxSegmentCount <= LastSegmentId)

{

printf("%s", "Unable to add segment because of memory restrictions.");

return;

}

LastSegmentId++;

Segment\* segment = (Segment\*)malloc(sizeof(Segment));

segment->Initialize(LastSegmentId, PageSize, MaxPageCount);

Segments->AddListItem(Segments, segment);

}

void ReadData(int segmentId, List\* locations, void\* destinationAddr)

{

Segment\* foundSegment = Segments->FindSegmentInListById(Segments, segmentId);

if (!foundSegment)

{

printf("%s \n", "No segment exist with such Id.");

return;

}

foundSegment->ReadData(locations, destinationAddr);

}

void RemoveData(int segmentId, List\* locations)

{

Segment\* foundSegment = Segments->FindSegmentInListById(Segments, segmentId);

if (!foundSegment)

{

printf("%s \n", "No segment exist with such Id.");

return;

}

foundSegment->RemoveData(locations);

}

void FreeAllSegments()

{

List\* next;

while (Segments)

{

next = Segments->Next;

((Segment\*)Segments->Content)->Free();

Segments = next;

}

}

unsigned int CalculateUnusedMemory()

{

unsigned int unusedMemory = 0;

List\* iterator = Segments;

while (iterator)

{

unusedMemory += ((Segment\*)iterator->Content)->CalculateUnusedMemory();

iterator = iterator->Next;

}

return unusedMemory;

}

void Free()

{

FreeAllSegments();

Segments->Free();

}

};