ИТМО Кафедра Информатики и прикладной математики

Отчет по лабораторной работе №2 «Управление памятью»

Выполнил: студент группы P3217
Плюхин Дмитрий
Преподаватель: Зыков А. Г.

2017 год

1. Задание

Используя алгоритмы из задания по дисциплине «Комбинаторные алгоритмы» реализуйте исследование методов управления памятью тремя способами:

1. Используя кучи.
2. Отображаемые файлы.
3. Базовые указатели.

Сравните временные характеристики. Выводы. Отчёт.

2. Листинг основной части программы

Файл main.cpp содержит первый тест производительности, а именно, в нем организуется сравнение способов формирования бинарного дерева поиска на основе данных во внешнем файле : сначала бинарное дерево строится только с использованием отдельной кучи, затем замеряется время его построения с использованием отдельной кучи и механизма MMF, наконец, запускается модификация того же алгоритма с использованием базовых указателей вместо копирования строк. Далее приведены только функции, отвечающие за построение бинарного дерева.

```
treeNodePointer fillTreeHeap(const char* fileName, HANDLE hNode){
    treeNodePointer rootPointer = NULL, nodePointer;
    char identifier[ID_MAX_LENGTH];
    ifstream identifiers(fileName);
    if (identifiers == NULL){
      return NULL;
    }
    while (identifiers){
      identifiers.getline(identifier, ID_MAX_LENGTH);
      nodePointer = (treeNodePointer)HeapAlloc(hNode, HEAP_ZERO_MEMORY, NODE_SIZE);
      strncpy(nodePointer->identifier,identifier,ID_SIZE);
      InsertTree(&rootPointer, nodePointer);
    }
    return rootPointer;
}

treeNodePointer fillTreeMMF(const char* fileName, HANDLE hNode, LPVOID mappedFile,DWORD
fileSize){
    treeNodePointer rootPointer = NULL, nodePointer;
    char* fileContent = (char*)mappedFile;
    DWORD old_pos = 0;
    DWORD pos = -1;
    char identifier[ID_MAX_LENGTH];
    DWORD i = 1;
    while(true){
      old_pos = pos+1;
      pos = find(fileContent, '\n', old_pos, fileSize);
      if (pos == 0) {
        break;
      }
      nodePointer = (treeNodePointer)HeapAlloc(hNode, HEAP_ZERO_MEMORY, NODE_SIZE);
      fileContent[pos] = '\0';
      strncpy(nodePointer->identifier,fileContent+old_pos,ID_SIZE);
      fileContent[pos] = '\n';
      InsertTree(&rootPointer, nodePointer);
      i++;
    }
    return rootPointer;
}

treeNodeBPointer fillTreeBases(const char* fileName, HANDLE hNode, LPVOID mappedFile,DWORD
fileSize){
    treeNodeBPointer rootPointer = NULL, nodePointer;
    char* fileContent = (char*)mappedFile;
    DWORD old_pos = 0;
    DWORD pos = -1;
    char identifier[ID_MAX_LENGTH];
    DWORD i = 1;
    while(true){
      old_pos = pos+1;
      pos = find(fileContent, '\n', old_pos, fileSize);
```

```
      if (pos == 0) {
        break;
      }
      nodePointer = (treeNodeBPointer)HeapAlloc(hNode, HEAP_ZERO_MEMORY, NODEB_SIZE);
      fileContent[pos] = '\0';
      nodePointer->identifier = old_pos;
      InsertTree(&rootPointer, nodePointer, fileContent);
      i++;
    }
    return rootPointer;
}
```

Файл sort.cpp реализует сравнение методов управления памятью на примере чтения внешнего файла, его сортировки и записи результата на диск. Этот алгоритм реализуется при помощи кучи с использованием явного чтения и записи на диск, при помощи механизма MMF с применением отображения содержимого файла на адресное пространство процесса и при помощи базовых указателей – в этом варианте применяется сортировка не самого файла, а индексного, содержащего указатели на записи в основном файле. Далее приведена только главная функция, касающаяся непосредственно рассматриваемых способов управления памятью.

```
int main(int argc, char* argv[]){
  cout << "Test sorting external file (time in seconds):" << endl;

  HANDLE hHeap = NULL;
  treeNodePointer rootPointer;
  int numberOfIdentifiers = 0;
  DWORD fileSize;
  DWORD KStart, KSize;

  HANDLE hFile = INVALID_HANDLE_VALUE, hMap = NULL;
  LPVOID pFile = NULL;
  LPVOID mappedFile = NULL, mappedXFile = NULL;
  DWORD * pSizes;
  float tmp;

  char tmpFileName[PATH_MAX_LEN];
  char idxFileName[PATH_MAX_LEN];
  strncpy(tmpFileName,argv[1],PATH_MAX_LEN);
  strcat(tmpFileName,".sorted");

  strncpy(idxFileName,argv[1],PATH_MAX_LEN);
  strcat(idxFileName,".idx");

  showTableHead();
  int koefficient = 100000;
  for (int i = 1; i <= 10; i++){
    generateFile(i*koefficient,argv[1],RECORD_SIZE-1);
    //Heap
    CopyFile(argv[1],tmpFileName,TRUE);
    hFile = CreateFile(tmpFileName, GENERIC_READ | GENERIC_WRITE, 0, NULL, OPEN_EXISTING, 0,
NULL);
    fileSize = GetFileSize(hFile,NULL);
    showCell((float)fileSize/1048576);
    CloseHandle(hFile);
    float fTimeStart = clock()/(float)CLOCKS_PER_SEC;
    hHeap = HeapCreate(HEAP_GENERATE_EXCEPTIONS | HEAP_NO_SERIALIZE, NODE_HEAP_ISIZE, 0);

    char* identifier = (char*)HeapAlloc(hHeap,HEAP_ZERO_MEMORY,fileSize);
    ifstream identifiers(tmpFileName);
    if (identifiers == NULL){
      return 1;
    }
    identifiers.read(identifier,fileSize);
    identifiers.close();

    qsort(identifier,fileSize/RECORD_SIZE,RECORD_SIZE,KeyCompare);

    ofstream outfile(tmpFileName);
    if (outfile == NULL){
      return 1;
    }
    outfile.write(identifier,fileSize);
    outfile.close();
    float fTimeStop = clock()/(float)CLOCKS_PER_SEC;

    showCell(fTimeStop - fTimeStart);
```

```
    if (hHeap != NULL) HeapDestroy(hHeap);
    hHeap = NULL;
    DeleteFile(tmpFileName);

    //MMF
    CopyFile(argv[1],tmpFileName,TRUE);
    fTimeStart = clock()/(float)CLOCKS_PER_SEC;
    hFile = CreateFile(tmpFileName, GENERIC_READ | GENERIC_WRITE, 0, NULL, OPEN_EXISTING, 0,
NULL);
    hMap = CreateFileMapping(hFile, NULL, PAGE_READWRITE, 0, fileSize, NULL);
    mappedFile = MapViewOfFile(hMap, FILE_MAP_ALL_ACCESS, 0, 0, 0);

    qsort(mappedFile,fileSize/RECORD_SIZE,RECORD_SIZE,KeyCompare);

    UnmapViewOfFile(mappedFile);
    CloseHandle(hMap);
    CloseHandle(hFile);
    fTimeStop = clock()/(float)CLOCKS_PER_SEC;
    showCell(fTimeStop - fTimeStart);
    DeleteFile(tmpFileName);

    //Based pointers
    generateFileEx(i*koefficient,argv[1],32,64);
    CopyFile(argv[1],tmpFileName,TRUE);
    HANDLE hInFile, hInMap;
    HANDLE hXFile, hXMap;
    DWORD fileSizeOld = fileSize;
    fTimeStart = clock()/(float)CLOCKS_PER_SEC;
    hInFile = CreateFile(tmpFileName,GENERIC_READ | GENERIC_WRITE,0, NULL, OPEN_EXISTING, 0,
NULL);
    fileSize = GetFileSize(hInFile,NULL);
    hInMap = CreateFileMapping(hInFile,NULL,PAGE_READWRITE,0,0,NULL);
    mappedFile = MapViewOfFile(hInMap,FILE_MAP_ALL_ACCESS,0,0,0);
    char* fileContent = (char*)mappedFile;


    hXFile = CreateFile(idxFileName,GENERIC_READ |
GENERIC_WRITE,FILE_SHARE_READ,NULL,CREATE_ALWAYS,0,NULL);
    DWORD es = IDX_ENTRY_SIZE;
    DWORD xFileSize = koefficient*es*i;
    hXMap = CreateFileMapping(hXFile,NULL,PAGE_READWRITE,0,xFileSize,NULL);
    mappedXFile = MapViewOfFile(hXMap,FILE_MAP_ALL_ACCESS,0,0,xFileSize);

    char* xfileContent = (char*)mappedXFile;
    DWORD basedDataPointer = 0;
    DWORD old_pos = 0;
    DWORD pos = -1;
    char key[8];
    char tmp;

    while(true){
      old_pos = pos+1;
      pos = find(fileContent, '\n', old_pos, fileSize);
      if (pos == 0) {
        break;
      }
      strncpy(key,fileContent+old_pos,8);
      strncpy(xfileContent,key,8);
      xfileContent+=8;
      strncpy(xfileContent,(char*)(&old_pos),4);
      xfileContent+=4;
    }

    UnmapViewOfFile(mappedFile);
    CloseHandle(hInMap);
    CloseHandle(hInFile);
    UnmapViewOfFile(mappedXFile);
    CloseHandle(hXMap);
    CloseHandle(hXFile);


    hFile = CreateFile(idxFileName, GENERIC_READ | GENERIC_WRITE, 0, NULL, OPEN_EXISTING, 0,
NULL);
    hMap = CreateFileMapping(hFile, NULL, PAGE_READWRITE, 0, xFileSize, NULL);
    mappedFile = MapViewOfFile(hMap, FILE_MAP_ALL_ACCESS, 0, 0, 0);

    qsort(mappedFile,xFileSize/es,es,KeyCompare);
```

```
    UnmapViewOfFile(mappedFile);
    CloseHandle(hMap);
    CloseHandle(hFile);
    fTimeStop = clock()/(float)CLOCKS_PER_SEC;
    showCell((fTimeStop - fTimeStart)/(1.0+(float)(fileSize - fileSizeOld)/fileSize)," |");
    drawHorizontalLine(130);
    DeleteFile(tmpFileName);
    DeleteFile(idxFileName);

  }
  return 0;
}
```

3. Результаты работы программы

В результате работы программы получены следующие результаты:

```
g++ main.cpp -o build/main
build/main data/loop.dat
Test creating binary tree from the data in external file (time in seconds):
```

Size (Mb)	Only Heap	Heap + MMF	Heap + MMF + Based pointers
1.80976	0.29	0.196	0.282
3.62769	0.682	0.472	0.585
5.43922	1.091	0.976	1.025
7.24239	1.507	1.167	1.733
9.05164	2.08	1.521	2.359
10.8649	2.669	1.964	3.047
12.6696	2.969	2.709	3.238
14.4866	3.714	3.082	3.884
16.2991	4.363	3.543	4.564
18.1313	4.54501	3.905	5.425

```
g++ sort.cpp -o build/sort
build/sort data/loop.dat
Test sorting external file (time in seconds):
```

Size (Mb)	Heap	MMF	MMF + Based pointers
4.57764	0.189	0.099	0.122437
9.15527	0.468	0.243	0.234155
13.7329	0.553	0.31	0.314784
18.3105	0.858	0.393	0.399297
22.8882	1.011	0.620999	0.531487
27.4658	1.057	0.739	0.606254
32.0435	1.417	0.744001	0.698569
36.6211	1.479	0.856003	0.828752
41.1987	1.78	1.12	0.952168
45.7764	1.892	1.107	1.02113

4. Вывод

Таким образом, существует три способа управления памятью в Windows: использование куч с динамически распределяемой памятью – такой способ является оптимальным, если необходимо обрабатывать множество составных однотипных объектов, среди которых могут быть, например, экземпляры различных классов или структур. Однако использование кучи становится неэффективным, если требуется простая обработка файла размером в несколько мегабайт и создание в качестве результата нового файла (например, смена кодировки или сортировка) – в этом случае более эффективным окажется использование механизма MMF (Memory mapped files), который существенно упрощает доступ к содержимому файла на диске и даже позволяет осуществить взаимодействие двух процессов через выделение общей области памяти. Третий же способ – использование базовых указателей – может быть применен совместно с механизмом MMF, когда требуется сохранение ссылок на отдельные фрагменты внешнего файла для последующего доступа.