**操作系统课程设计报告** 日期：19.6.11

|  |  |  |  |
| --- | --- | --- | --- |
| 实习题目：内存管理 | | | |
| 完成人 | 姓名：龙佳琪 | 班级：06 | 学号2017141461065 |
| 实习内容简要描述 | 1. 问题   使用Windows 2000／XP的API函数，编写一个包含两个线程的进程，一个线程用于模拟内存分配活动，一个线程用于跟踪第一个线程的内存行为，而且要求两个线程之间通过信号量实现同步。模拟内存活动的线程可以从一个文件中读出要进行的内存操作。内存操作包括保留(reserve)一个区域、提交(commit)一个区域、释放(release)一个区域、回收(decommit)一个区域和加锁(lock)与解锁(unlock)一个区域，可以将这些操作编号存放于文件。   1. 实验要求   模拟内存活动的线程可以从一个文件中读出要进行的内存操作，每个内存操作包括如下内容：  （1）时间：操作等待时间。  （2）块数：分配内存的粒度。  （3）操作：包括保留(reserve)一个区域、提交(commit)一个区域、释放(release)一个区域、回收(decommit)一个区域和加锁(lock)与解锁(unlock)一个区域，可以将这些操作编号存放于文件。  （4）保留是指保留进程的虚拟地址空间，而不分配物理存储空间。  （5）提交在内存中分配物理存储空间。  （6）回收是指释放物理内存空间，但在虚拟地址空问仍然保留，它与提交相对应，即可以回收已经提交的内存块。  （7）释放是指将物理存储和虚拟地址空间全部释放，它与保留(reserve)相对应，即可以释放已经保留的内存块。  （8）大小：块的大小。  （9）访问权限：共五种，分别为PAGE\_READONLY,PAGE\_READWRITE，PAGE\_EXECUT，PAGE\_EXECUTE\_READ和PAGE\_EXETUTE\_READWRITE。可以将这些权限编号存放于文件中跟踪线程将页面大小、已使用的地址范围、物理内存总量，以及虚拟内存总量等信息显示出来。 | | |
| 主要代码结构  （附注释） | Makefile.cpp 将操作写入文件    Memory-op.cpp 内存管理操作  struct operation  {  int time;  int block;  int oper;  int protection;  };  struct trace  {  LPVOID start;  long size;  };  DWORD Tracker(LPDWORD lpdwparm); //跟踪分配  void Allocator(); //模拟内存分配 | | |
| 结果分析（或错误原因分析） | 二进制文件的生成：  随机生成结构体中的四个参数，同时由于状态有6种，采用双循环的形式生成操作文件，并且存入opfile操作文件中  for (int j = 0; j < 6; j++)  for (int i = 0; i < 5; i++)  {  op.time = rand() % 1000;  op.block = rand() % 5 + 1;  op.oper = j;  op.protection = i;  fwrite(&op, sizeof(operation), 1, file);  }  内存管理文件：按照要求，设置两个线程且使用两个信号量通知和记录线程。函数 void Allocator()完成模拟分配的功能，而 DWORD Tracker(LPDWORD lpdwparm)用于跟踪输出当前分配的状态，两个线程间通过信号量实现同步。最终的分配信息保存于文件out.txt中。 | | |
| 注释：如果上述表格空间不够，可以另附表格进行说明 | | | |

设计文档：

1. Makefile.cpp

该程序意在生成内存管理的相应操作，并按照要求以结构的形式写入一个文件中，便于管理文件读取并操作。由于内存操作有6种，且操作权限有5种，故在此采用双循环生成。外层循环生成内存的操作：保留、提交、锁、解锁、回收、释放；内存循环生成操作权限：PAGE\_READONLY、PAGE\_READWRITE、PAGE\_EXECUTE、PAGE\_EXECUTE\_READ、PAGE\_EXECUTE\_READWRITE。其他信息采用随机数直接生成。

1. Memory-op.cpp

（1）main（）

创建2个线程和信号量

（2）Tracker（）

打开文件并输出，但是要等到分配完后，也就是要等待Allocator的信号量，通过查阅资料可知，用WaitForSingleObject（）函数生成。

输出系统消息，设置变量类型SYSTEM\_INFO ，通过 GetSystemInfo()可以获取数据

输出内存状况，设置变量类型MEMORYSTATUS ，通过GlobalMemoryStatus()可以获取

输出内存基本信息，设置变量类型MEMORY\_BASIC\_INFORMATION，通过VirtualQuery()获取数据

操作完后释放信号量: ReleaseSemaphore，即告知Allocator ()可以进行下一次分配

（3）Allocator（）

打开文件，自然也是要在等到Tracker的信号后再做出（即要等到（2）中的信号量的释放）；读出文件并获取相关的内容，依次对内存操作的参数进行设置；待全部设置完成后释放信号量（trac）使Tracker可以进行输出

伪代码的实现：

Begin

创建2个进程

Allocator:

打开文件

P(alloc)

读出数据并赋值

V(trac)

End

Tracker:

打开文件

P(trac)

向屏幕输出数据

向输出文件写数据

V(alloc)

End

End

附代码:

Makefile.cpp

#include <fstream>

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#pragma warning(disable:4996);

struct operation

{

int time;

int block;

int oper;

int protection;

};

int main()

{

FILE\* file;

file = fopen("opfile", "wb");

operation op;

//0--保留，1--提交，2--锁，3--解锁，4--回收，5--释放

for (int j = 0; j < 6; j++)

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

{

op.time = rand() % 1000;

op.block = rand() % 5 + 1;

op.oper = j;

op.protection = i;

fwrite(&op, sizeof(operation), 1, file);

}

return 0;

}

Memory-op.cpp

# include <stdio.h>

# include <stdlib.h>

# include <windows.h>

# include <iostream>

# include <fstream>

using namespace std;

#pragma warning(disable:4996)；

struct operation

{

int time;

int block;

int oper;

int protection;

};

struct trace

{

LPVOID start;

long size;

};

HANDLE allo, trac;

DWORD Tracker(LPDWORD lpdwparm)//跟踪分配

{

ofstream outfile;

outfile.open("out.txt");

for (int i = 0; i <= 30; i++)

{

WaitForSingleObject(trac, INFINITE);

outfile << i << endl;

SYSTEM\_INFO info; //系统消息

GetSystemInfo(&info);

outfile << "dwActiveProcessorMask" << "\t" << info.dwActiveProcessorMask << endl;

outfile << "dwAllocationGranularity" << "\t" << info.dwAllocationGranularity << endl;

outfile << "dwNumberOfProcessors" << "\t" << info.dwNumberOfProcessors << endl;

outfile << "dwOemId" << "\t" << info.dwOemId << endl;

outfile << "dwPageSize" << "\t" << info.dwPageSize << endl;

outfile << "dwProcessorType" << "\t" << info.dwProcessorType << endl;

outfile << "lpMaximumApplicationAddress" << "\t" << info.lpMaximumApplicationAddress << endl;

outfile << "lpMinimumapplicationAddress" << "\t" << info.lpMinimumApplicationAddress << endl;

outfile << "wProcessorArchitecture" << "\t" << info.wProcessorArchitecture << endl;

outfile << "wProcessorLevel" << "\t" << info.wProcessorLevel << endl;

outfile << "wProcessorRevision" << "\t" << info.wProcessorRevision << endl;

outfile << "wReserved" << "\t" << info.wReserved << endl;

outfile << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*" << endl;

//内存状况

MEMORYSTATUS status;

GlobalMemoryStatus(&status);

outfile << "dwAvailPageFile" << "\t" << status.dwAvailPageFile << endl;

outfile << "dwAvailPhys" << "\t" << status.dwAvailPhys << endl;

outfile << "dwAvailVirtual" << "\t" << status.dwAvailVirtual << endl;

outfile << "dwLength" << "\t" << status.dwLength << endl;

outfile << "dwMemoryLoad" << "\t" << status.dwMemoryLoad << endl;

outfile << "dwTotalPageFile" << "\t" << status.dwTotalPageFile << endl;

outfile << "dwTotalPhy" << "\t" << status.dwTotalPhys << endl;

outfile << "dwTotalVirtual" << "\t" << status.dwTotalVirtual << endl;

outfile << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*" << endl;

MEMORY\_BASIC\_INFORMATION mem; //内存基本信息

VirtualQuery(info.lpMinimumApplicationAddress, &mem, sizeof(MEMORY\_BASIC\_INFORMATION));

outfile << "AllocationBase" << "\t" << mem.AllocationBase << endl;

outfile << "AllocationProtect" << "\t" << mem.AllocationProtect << endl;

outfile << "BaseAddress" << "\t" << mem.BaseAddress << endl;

outfile << "Protect" << "\t" << mem.Protect << endl;

outfile << "RegionSize" << "\t" << mem.RegionSize << endl;

outfile << "State" << "\t" << mem.State << endl;

outfile << "Type" << "\t" << mem.Type << endl;

outfile << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*" << endl;

//释放信号量

ReleaseSemaphore(allo, 1, NULL);

}

return 0;

}

void Allocator()//模拟内存分配

{

trace traceArray[5];

int index = 0;

FILE\* file;

file = fopen("opfile", "rb");

operation op;

SYSTEM\_INFO info;

DWORD temp;

GetSystemInfo(&info);

for (int i = 0; i < 30; i++)

{

WaitForSingleObject(allo, INFINITE);

cout << i << ":";

fread(&op, sizeof(operation), 1, file);

Sleep(op.time); //执行时间

GetSystemInfo(&info);

switch (op.protection)

{

case 0:

{

index = 0;

temp = PAGE\_READONLY;

break;

}

case 1:

temp = PAGE\_READWRITE;

break;

case 2:

temp = PAGE\_EXECUTE;

break;

case 3:

temp = PAGE\_EXECUTE\_READ;

break;

case 4:

temp = PAGE\_EXECUTE\_READWRITE;

break;

default:

temp = PAGE\_READONLY;

}

switch (op.oper)

{

case 0: //保留

{

cout << "reserve now " << endl;

traceArray[index].start = VirtualAlloc(NULL, op.block \* info.dwPageSize,

MEM\_RESERVE, PAGE\_NOACCESS);

traceArray[index++].size = op.block \* info.dwPageSize;

cout << "starting address:" << traceArray[index - 1].start << "\t" << "size:"

<< traceArray[index - 1].size << endl;

break;

}

case 1: //提交

{

cout << "commit now " << endl;

traceArray[index].start = VirtualAlloc(traceArray[index].start,

traceArray[index].size, MEM\_COMMIT, temp);

index++;

cout << "starting address:" << traceArray[index - 1].start << "\t" << "size:"

<< traceArray[index - 1].size << endl;

break;

}

case 2: //锁住

{

cout << "lock now" << endl;

cout << "starting address:" << traceArray[index].start << "size:" <<

traceArray[index].size << endl;

if (!VirtualLock(traceArray[index].start, traceArray[index++].size))

cout << GetLastError() << endl;

break;

}

case 3: //解锁

{

cout << "unlock now" << endl;

cout << "starting address:" << traceArray[index].start << "\t" << "size:"

<< traceArray[index].size << endl;

if (!VirtualUnlock(traceArray[index].start, traceArray[index++].size))

cout << GetLastError() << endl;

break;

}

case 4: //回收

{

cout << "decommit now" << endl;

cout << "starting address:" << traceArray[index].start << "\t"

<< "size:" << traceArray[index].size << endl;

if (!VirtualFree(traceArray[index].start, traceArray[index++].size,

MEM\_DECOMMIT))

cout << GetLastError() << endl;

break;

}

case 5: //释放

{

cout << "release now" << endl;

cout << "starting address:" << traceArray[index].start << "\t" <<

"size:" << traceArray[index].size << endl;

if (!VirtualFree(traceArray[index++].start, 0, MEM\_RELEASE))

cout << GetLastError() << endl;

break;

}

default:

cout << "error" << endl;

}

ReleaseSemaphore(trac, 1, NULL); //释放信号量

}

}

//主函数

int main()

{

DWORD dwThread;

HANDLE handle[2];

handle[0] = CreateThread(NULL, 0, (LPTHREAD\_START\_ROUTINE)Tracker, NULL, 0,

&dwThread);

handle[1] = CreateThread(NULL, 0, (LPTHREAD\_START\_ROUTINE)Allocator, NULL, 0,

&dwThread);

allo = CreateSemaphore(NULL, 0, 1, "allo");//信号量

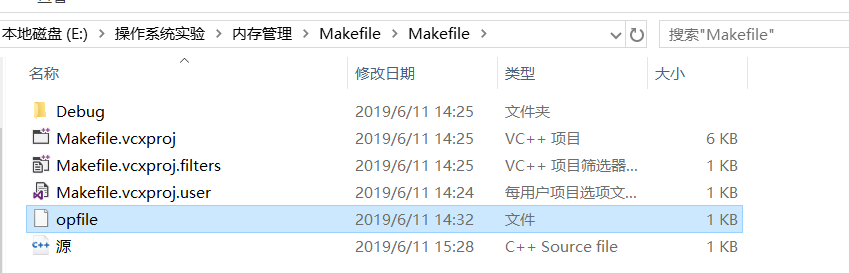
trac = CreateSemaphore(NULL, 1, 1, "trac");//信号量

WaitForMultipleObjects(2, handle, TRUE, INFINITE);

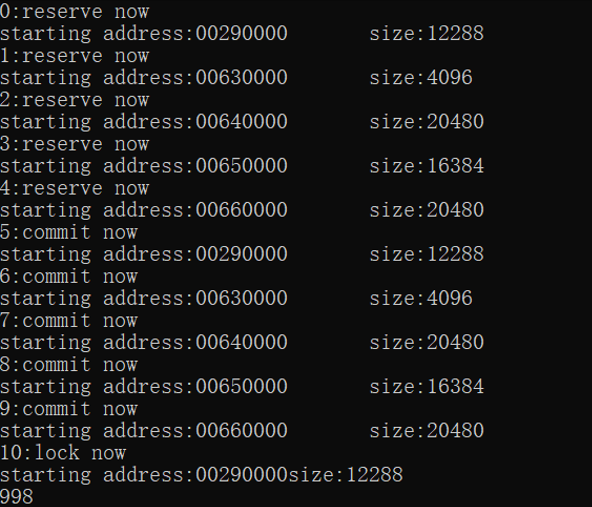
return 0;

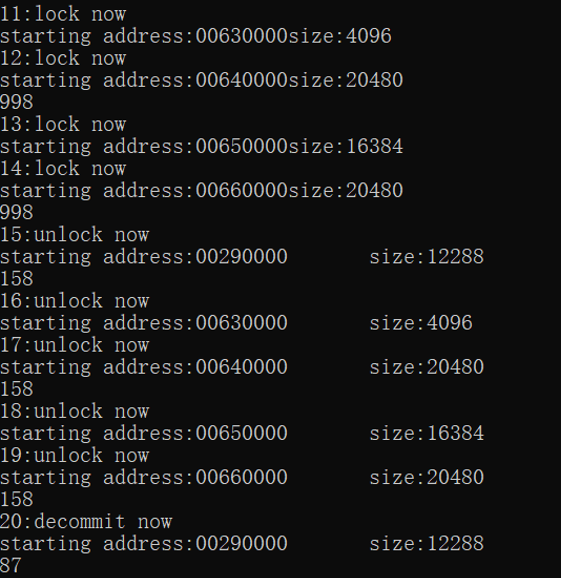
}

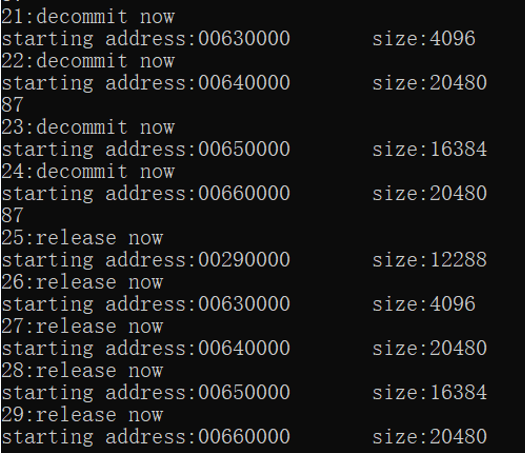
运行结果截屏：



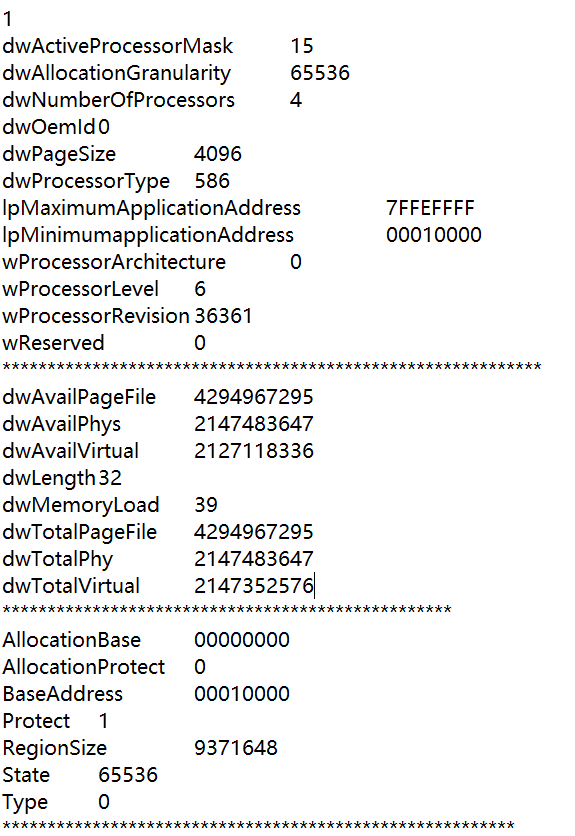
Makefile.cpp运行后所生成的用于控制内存操作执行的文件

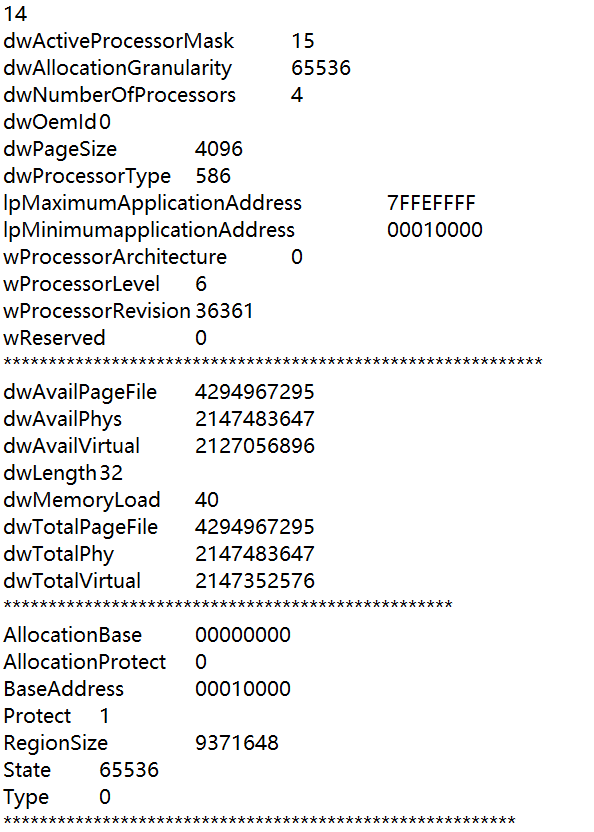






Memory-op.cpp运行结果





部分输出文档中的结果