操作系统原理实习二

【实习题目1】：

遍历当前系统中的所有进程，并能用树状结构显示进程之间的父子关系。

【题目分析、相关原理及设计思路】：

题目分析：需要对当前系统中的所有进程进行遍历，通过进程ID找到进程之间的父子关系，用树的形式输出进程的关系。

相关原理：通过Windows系统自带的API函数中的进程函数可以对进程进行访问修改，系统中的进程可以创建一个自己的进程，作为自己的子进程运行，从而两个进程之间具有父子关系，并能通过进程id找出父进程或子进程。

设计思路：通过进程快照遍历当前进程，在遍历过程中根据进程之间的父子关系将进程信息存储到树的结点信息中，最后遍历输出树结构。

【详细步骤和相应截图】：

1、定义两个结构体用于保存进程信息和树结点信息；

//定义进程信息

struct ProcessInfo

{

DWORD pid;

DWORD parentpid;

wstring name;

};

//定义进程树节点

struct ProcessTreeNode

{

ProcessInfo info;

vector<ProcessTreeNode>children;

};

2、创建进程快照，遍历进程，构建进程树；

//遍历进程信息并返回进程树根节点

ProcessTreeNode getProcessTree()

{

ProcessTreeNode root;

root.info = {0,0,L"System"};//系统进程根节点

//创建进程快照

HANDLE hSnapshot = CreateToolhelp32Snapshot(TH32CS\_SNAPPROCESS, 0);

if (hSnapshot == INVALID\_HANDLE\_VALUE)

{

cout << "error" << endl;

return root;

}

//遍历进程信息

PROCESSENTRY32 pe32 = { sizeof(pe32) };

if (!Process32First(hSnapshot, &pe32))

{

cerr << "error" << endl;

CloseHandle(hSnapshot);

return root;

}

do

{

ProcessInfo info = { pe32.th32ProcessID,pe32.th32ParentProcessID,pe32.szExeFile };

if (info.pid == 0)

continue; //过滤空闲进程

ProcessTreeNode\* parent = &root;//查找父节点

if (info.pid != 4)

{

// 除了System进程(pid = 4)外，其他进程都有父进程

queue<ProcessTreeNode\*> q;

q.push(&root);

while (!q.empty())

{

ProcessTreeNode\* node = q.front();

q.pop();

if (node->info.pid == info.parentpid)

{

parent = node;

break;

}

for (auto& child : node->children)

{

q.push(&child);

}

}

}

//添加当前进程节点到父节点的子节点列表中

ProcessTreeNode node;

node.info = info;

parent->children.push\_back(node);

} while (Process32Next(hSnapshot, &pe32));

CloseHandle(hSnapshot);

return root;

}

3、打印进程树；

//打印进程树

void printProcessTreeNode(ProcessTreeNode& node, int depth)

{

wcout << wstring(depth \* 4, L' ') << "|---" << node.info.name << " " << node.info.pid << endl;

for (auto& child : node.children)

{

printProcessTreeNode(child, depth + 1);

}

}

4、主函数；

int main()

{

ProcessTreeNode root = getProcessTree();

printProcessTreeNode(root, 0);

return 0;

}

5、运行结果；

图片包含 文本

描述已自动生成

【实习中遇到的困难及解决方法、实习心得或课程建议】：

困难：如何将进程信息和树结点信息连接起来；

解决办法：定义两个结构体分别用于存储进程信息和树结点信息，在创建快照，遍历进程的时候，通过一个队列作为媒介将进程信息以及进程关系通过树结构连接。

实习心得：了解了Windows系统下API进程访问函数的使用，对进程之间的父子关系有了更深入了解，通过树形结构可视化的将进程间的关系展示出来，对进程的具体操作方法有了实践性认识。

【实习题目2】：

设计一个程序，可以做为主进程和子进程两种模式运行，做为主进程创建共享内存并初始化，然后启动 3 个子进程，最后等待子进程结束，并输出共享内存中的内容；做为子进程，首先打开共享内存，然后以临界区的形式对共享内存中的值加 1，重复 10000 次。

【题目分析、相关原理及设计思路】：

题目分析：通过改变主函数的argc值可以选择程序是作为主进程还是子进程运行

相关原理：共享内存是一种最快的进程间通信方式，它允许多个进程访问同一块物理内存空间。在Windows下，可以使用CreateFileMapping和OpenFileMapping函数来创建或打开一个命名的文件映射对象（也就是共享内存），然后使用MapViewOfFile函数来将文件映射对象映射到当前进程的地址空间。

信号量一种用于控制多个进程对共享资源的访问的同步机制。通过维护一个计数器控制进程使用资源的顺序。在Windows下，可以使用CreateSemaphore和OpenSemaphore函数来创建和打开一个命名的信号量对象，使用WaitForSingleObject和ReleaseSemaphore函数来获取或释放信号量。

临界区是指对共享资源进行访问或修改的代码段，在任意时刻只能有一个线程执行该代码段。如果有多个线程同时想要执行临界区代码，则必须按照某种规则排队等待。使用信号量可以实现临界区。

设计思路：主进程创建一个共享内存，在共享内存中定义一个整型变量，创建信号量对象，循环创建三次子进程，在子进程中打开共享内存和信号量对象，循环10000次执行获取信号量，将整型变量+1，再释放信号量，等所有子进程结束后输出共享内存中的内容。

【详细步骤和相应截图】：

1、主进程编写；

if (argc == 1)

{

HANDLE h = CreateFileMapping(NULL, NULL, PAGE\_READWRITE, 0, Memery\_size, TEXT("ShareMemery"));

void\* pt = MapViewOfFile(h, FILE\_MAP\_WRITE, 0, 0, 0); //尾填0扩展到末尾

\*(int\*)pt = 0;

HANDLE ph[3];

TCHAR szCommandLine[] = TEXT("实习二2.exe sub");

for (int i = 0; i < 3; i++)

{

STARTUPINFO si{ sizeof(si) };

PROCESS\_INFORMATION pi;

CreateProcess(NULL, szCommandLine, NULL, NULL, FALSE, NULL, NULL, NULL, &si, &pi);

ph[i] = pi.hProcess;

}

WaitForMultipleObjects(3, ph, TRUE, -1);

cout << "共享内存的内容为:" << \*(int\*)pt;

UnmapViewOfFile(pt);

CloseHandle(h);

}

2、子进程编写；

else

{

HANDLE hShare = OpenFileMapping(FILE\_MAP\_ALL\_ACCESS, FALSE, TEXT("ShareMemery"));

void\* pt = MapViewOfFile(hShare, FILE\_MAP\_WRITE, 0, 0, 0);

HANDLE hCriticalSection = CreateSemaphore(NULL, 1, 1, TEXT("CriticalSection"));

for (int i = 0; i < 10000; i++)

{

WaitForSingleObject(hCriticalSection, -1);

\*(int\*)pt += 1;

ReleaseSemaphore(hCriticalSection, 1, NULL);

}

CloseHandle(hCriticalSection);

UnmapViewOfFile(hShare);

CloseHandle(hShare);

}

3、运行结果；

文本

描述已自动生成

【实习中遇到的困难及解决方法、实习心得或课程建议】：

困难：怎么控制程序作为主进程或子进程运行。

解决方法：main函数的argc参数的值决定了当前主函数中的程序是作为主进程还是子进程运行，等于1时是主程序，其他则是子程序，则通过一个if else语句可以将主进程和子进程运行分开。

实习心得：通过这个题目，了解了程序作为主进程和子进程运行的区别，知道了如何控制程序作为主进程和子进程运行，知道了共享内存的创建和访问方法，以及信号量控制实现临界区的方法。

【实习题目3】：

基于线程的服务器和多进程客户端，实现简单四则运算。可以通过网络通信也可以通过共享内存结合信号量进行通迅，同时请求的客户端进程数可以超过最大 worker 数，当没有请求时 worker 需要等待。

【题目分析、相关原理及设计思路】：

题目分析：需要实现两个程序，服务器和客户端，通过共享内存和信号量结合进行通讯，服务器中的线程数固定，客户端可以超过线程数，并实现工作状态全满是的请求等待。

相关原理：通过共享内存和信号量实现临界区可以实现基于线程的服务器和客户端通信，通过共享内存数据的交互传输获取请求和结果，通过信号量控制线程的运行状态。

设计思路：服务器端需要创建线程用于处理客户端的请求，使用共享内存进行通讯缓冲，同时创建信号量控制对共享内存的访问，其中的线程根据传来的信号量对请求进行实现，同时释放共享内存并返回信号量，客户端则创建信号量，确定信号量状态，打开共享内存，放入操作请求，等待服务器回应，结束后释放信号量。

【详细步骤和相应截图】：

1、实现服务器端：

#include <iostream>

#include <Windows.h>

#include <thread>

#include <mutex>

#include <condition\_variable>

#include <vector>

#include <queue>

#include <sstream>

#include <sysinfoapi.h>

using namespace std;

// 客户端请求数据结构

struct RequestData {

int v; //标记内存空间是否可用

double a; // 操作数1

double b; // 操作数2

char op; // 操作符

};

// 服务器处理结果数据结构

struct ResultData {

int v; //标记内存空间是否可用

double result; // 计算结果

};

struct IndexHead

{

int headIndex;

int tailIndex;

};

const int MAX\_WORKERS = 4; // 最大worker数

const int MAX\_REQUESTS = 20; // 最大请求队列长度

const int SHARED\_MEM\_SIZE = sizeof(IndexHead) + MAX\_REQUESTS \* sizeof(RequestData); // 共享内存大小

const TCHAR\* SHARED\_MEM\_NAME = L"SharedMemory";

const TCHAR\* SEM\_NAME = L"WorkSemaphore";

HANDLE workSem; // 请求信号量

vector<thread> workers; // worker线程池

//worker线程函数

void worker(int id)

{

while (true)

{

WaitForSingleObject(workSem, INFINITE);

HANDLE hMemory = OpenFileMapping(FILE\_MAP\_ALL\_ACCESS, FALSE, SHARED\_MEM\_NAME);

if (hMemory == nullptr)

{

printf("线程 %d,内存访问失败！\n", id);

ReleaseSemaphore(workSem, 1, NULL);

Sleep(1000);

continue;

}

void\* pData = MapViewOfFile(hMemory, FILE\_MAP\_ALL\_ACCESS, 0, 0, 0);

if (pData == nullptr)

{

CloseHandle(hMemory);

ReleaseSemaphore(workSem, 1, NULL);

printf("线程 %d,建立映射失败！\n", id);

Sleep(10);

continue;

}

IndexHead\* pIndex = (IndexHead\*)pData;

if (((IndexHead\*)pIndex)->headIndex == ((IndexHead\*)pIndex)->tailIndex)

{

UnmapViewOfFile(pData);

CloseHandle(hMemory);

ReleaseSemaphore(workSem, 1, NULL);

Sleep(10);

continue;

}

printf("线程 %d,正在响应\n", id);

int id = ((IndexHead\*)pIndex)->headIndex;

((IndexHead\*)pIndex)->headIndex = (((IndexHead\*)pIndex)->headIndex + 1) % MAX\_REQUESTS;

RequestData req;

req.a = ((RequestData\*)((char\*)pData + sizeof(IndexHead) + sizeof(RequestData) \* id))->a;

req.b = ((RequestData\*)((char\*)pData + sizeof(IndexHead) + sizeof(RequestData) \* id))->b;

req.op = ((RequestData\*)((char\*)pData + sizeof(IndexHead) + sizeof(RequestData) \* id))->op;

UnmapViewOfFile(pData);

CloseHandle(hMemory);

ReleaseSemaphore(workSem, 1, NULL);

ResultData res;

switch (req.op)

{

case '+':

{

res.result = req.a + req.b;

}break;

case '-':

{

res.result = req.a - req.b;

}break;

case '\*':

{

res.result = req.a \* req.b;

}break;

case '/':

{

if (req.b == 0)

{

res.result = 2147483647;

}

else

{

res.result = req.a / req.b;

}

}break;

}

WaitForSingleObject(workSem, INFINITE);

hMemory = OpenFileMapping(FILE\_MAP\_ALL\_ACCESS, FALSE, SHARED\_MEM\_NAME);

pData = MapViewOfFile(hMemory, FILE\_MAP\_ALL\_ACCESS, 0, 0, 0);

ResultData\* pRes = (ResultData\*)((char\*)pData + sizeof(IndexHead) + sizeof(RequestData) \* id);

((ResultData\*)pRes)->result = res.result;

((ResultData\*)pRes)->v = 2;

UnmapViewOfFile(pData);

CloseHandle(hMemory);

ReleaseSemaphore(workSem, 1, NULL);

}

}

int main()

{

HANDLE hShare = CreateFileMapping(NULL, NULL, PAGE\_READWRITE, 0, SHARED\_MEM\_SIZE, SHARED\_MEM\_NAME);

if (hShare == nullptr)

{

printf("共享内存创建失败！\n");

return -1;

}

LARGE\_INTEGER liFileSize;

workSem = CreateSemaphore(NULL, 0, 1, SEM\_NAME);

if (workSem == nullptr)

{

CloseHandle(hShare);

printf("信号量创建失败！\n");

return -1;

}

void\* pData = MapViewOfFile(hShare, FILE\_MAP\_ALL\_ACCESS, 0, 0, 0);

if (pData == nullptr)

{

//UnmapViewOfFile(pData);

CloseHandle(workSem);

CloseHandle(hShare);

printf("内存映射失败！\n");

return -1;

}

ZeroMemory(pData, SHARED\_MEM\_SIZE);

UnmapViewOfFile(pData);

pData = MapViewOfFile(hShare, FILE\_MAP\_ALL\_ACCESS, 0, 0, sizeof(IndexHead));

((IndexHead\*)pData)->headIndex = 0;

((IndexHead\*)pData)->tailIndex = 0;

UnmapViewOfFile(pData);

for (int i = 0; i < MAX\_WORKERS; i++)

{

workers.emplace\_back(worker, i);

}

cout << "就绪\n";

ReleaseSemaphore(workSem, 1, NULL);

while (true);

CloseHandle(workSem);

CloseHandle(hShare);

return 0;

}

2、实现客户端：

#include <iostream>

#include <Windows.h>

#include <vector>

#include <queue>

#include <thread>

#include <mutex>

#include <condition\_variable>

#include <sstream>

#include <string>

using namespace std;

// 客户端请求数据结构

struct RequestData {

int v; //标记内存空间是否可用

double a; // 操作数1

double b; // 操作数2

char op; // 操作符

};

// 服务器处理结果数据结构

struct ResultData {

int v; //标记内存空间是否可用

double result; // 计算结果

};

struct IndexHead

{

int headIndex;

int tailIndex;

};

const int MAX\_WORKERS = 4; // 最大worker数

const int MAX\_REQUESTS = 20; // 最大请求队列长度

const int SHARED\_MEM\_SIZE = sizeof(IndexHead) + MAX\_REQUESTS \* sizeof(RequestData); // 共享内存大小

const TCHAR\* SHARED\_MEM\_NAME = L"SharedMemory";

const TCHAR\* SEM\_NAME = L"WorkSemaphore";

HANDLE workSem; // 请求信号量

int main()

{

workSem = OpenSemaphore(SEMAPHORE\_ALL\_ACCESS, FALSE, SEM\_NAME);

if (workSem == nullptr)

{

// CloseHandle(workSem);

cerr << "信号量打开失败，请检查服务器是否启动！\n";

system("pause");

return -1;

}

while (true)

{

RequestData req;

printf("请输入计算式:\n");

cin >> req.a >> req.op >> req.b;

printf("请求服务器中\n");

WaitForSingleObject(workSem, INFINITE);

HANDLE hMemory = OpenFileMapping(FILE\_MAP\_ALL\_ACCESS, FALSE, SHARED\_MEM\_NAME);

if (hMemory == NULL)

{

printf("内存访问失败！\n");

ReleaseSemaphore(workSem, 1, NULL);

Sleep(1000);

continue;

}

void\* pData = MapViewOfFile(hMemory, FILE\_MAP\_ALL\_ACCESS, 0, 0, 0);

if (pData == nullptr)

{

ReleaseSemaphore(workSem, 1, NULL);

CloseHandle(hMemory);

printf("内存映射失败！\n");

continue;

}

if (((IndexHead\*)pData)->headIndex == (((IndexHead\*)pData)->tailIndex + 1) % MAX\_REQUESTS)

{

UnmapViewOfFile(pData);

ReleaseSemaphore(workSem, 1, NULL);

CloseHandle(hMemory);

Sleep(10);

continue;

}

int id = ((IndexHead\*)pData)->tailIndex;

((IndexHead\*)pData)->tailIndex = (((IndexHead\*)pData)->tailIndex + 1) % MAX\_REQUESTS;

RequestData\* pReq = (RequestData\*)((char\*)pData + sizeof(IndexHead) + sizeof(RequestData) \* id);

pReq->v = 1;

pReq->a = req.a;

pReq->b = req.b;

pReq->op = req.op;

UnmapViewOfFile(pData);

CloseHandle(hMemory);

ReleaseSemaphore(workSem, 1, NULL);

Sleep(1000);

while (true)

{

WaitForSingleObject(workSem, INFINITE);

hMemory = OpenFileMapping(FILE\_MAP\_ALL\_ACCESS, FALSE, SHARED\_MEM\_NAME);

pData = MapViewOfFile(hMemory, FILE\_MAP\_ALL\_ACCESS, 0, 0, 0);

ResultData\* pRes = (ResultData\*)((char\*)pData + sizeof(IndexHead) + sizeof(RequestData) \* id);

if (pRes->v == 2)

{

cout << "答案：" << pRes->result << "\n";

pRes->v = 0;

IndexHead\* pIndex = (IndexHead\*)pData;

while (pIndex->headIndex == id)

{

pRes = (ResultData\*)((char\*)pData + sizeof(IndexHead) + sizeof(RequestData) \* id);

if (pRes->v == 0)

{

pIndex->headIndex = (pIndex->headIndex + 1) % MAX\_REQUESTS;

id = pIndex->headIndex;

}

else

{

UnmapViewOfFile(pData);

break;

}

}

ReleaseSemaphore(workSem, 1, NULL);

break;

}

else

{

UnmapViewOfFile(pData);

}

CloseHandle(hMemory);

ReleaseSemaphore(workSem, 1, NULL);

Sleep(100);

}

}

CloseHandle(workSem);

return 0;

}

3、运行结果：

屏幕上有字

描述已自动生成

【实习中遇到的困难及解决方法、实习心得或课程建议】：

困难：怎么传输请求和结果的同时传输线程状态。

解决方法：创建两个结构体，一个用于存储计算式的数字以及操作符，另一个存储答案，两个结构体都有一个标志量用于标记当前线程是否处于可用状态，在服务端每次解决请求线程结束后都将答案和标记量一同传输给客户端，客户端可以通过检索共享内存中的标记量知道当前请求是否进行完，从而执行输出操作和信号量释放操作。

实习心得：通过这道题目，了解了如何利用共享内存和信号量结合实现基于线程的服务器和客户端通讯，对线程有了更直观的了解，对共享内存和信号量的操作也更熟练，认识到进程与线程的关系，服务器端的一个进程有多个线程，不同进程的线程间要利用消息通讯的方法实现同步。