Комп'ютерний практикум №4
Організація багатозадачності в середовищі Win32 за допомогою процесів і потоків

Виконав:
Студент 2 курсу ФТІ
групи ФІ-92
Поночевний Назар Юрійович

Мета: Ознайомлення з можливостями багатозадачності Win32 та робота з процесами і потоками.

Завдання 10:

Розробити застосунок, який складається з трьох процесів (головного та 2х дочірніх). У одному з дочірніх процесів створюються/відкриваються декілька файлів (в залежності від варіанту): текстові (в залежності від варіанту) або і двійкові (в залежності від варіанту). Інформація про відкриті файли (дескриптори) передається в дочірні процеси на етапі їх створення через командний рядок.

На головний процес покладена роль координатора та інтегратора інформації про роботу обчислень, проведених потоками його дочірніх процесів. Головний процес породжує два дочірні процеси на один яких покладається задача створення та заповнення файлів(суть обробки згідно варіанту), причому параметри файлу передаються від головного по іменованому каналу.

Робота другого дочірнього процесу полягає у наступному:

Він утворює два потоки. Перший з потоків дає старт обробки для другого потоку та визначенні параметрів обробки(наприклад розмір матриці), призупинці та поновленні обробки, отриманні проміжних результатів обробки(згідно варіанту), та передачі головному процесу результатів обробки та часу виконання. Другий потік виконує алгоритмічну реалізацію змісту файлу(сортування, вибірка, перекодування і т.д.) Головний процес виводить результати у вікно. Процеси пов’язані між собою іменованими каналами, або віконними обмінами. Роботу потоків необхідно узгоджувати. Наприклад, якщо на один потік покладається вибірка якихось чисел, а на другий покладають обчислення суми, то їх сума не може бути отримана раніше ніж буде вибрано останнє потрібне число. Узгодження роботи потоків реалізується через механізм зміни відносних пріоритетів потоків, а також через використання подій(Events). При демонстрації роботи передбачити індикацію роботи відповідного процесу та потоку шляхом виведення ідентифікатора та змісту виконання. З цією метою передбачити затримки під час переходу від процесу до процесу та від потоку до потоку, причому початок роботи відповідного процесу чи потоку виводиться інформація, що позначає хто працює.

1. Зчитати дані з файлу, в якому записані блоками записані текстові та числові дані. З цих даних вибрати числові дані та сформувати матрицю MxN.

2. Обчислити добуток всіх колонок масиву, у яких перший елемент більше елементів розташованих на головній і побічній діагоналі. Один потік читає блоки і визначає потрібний , а інший формує.
3. Використовувати іменовані канали для з'єднання процесів.

Код (Main Process):

```
#include <tchar.h>
#include <windows.h>
#include <stdio.h>
#include <strsafe.h>

#define BUFSIZE 4096

DWORD WINAPI InstanceThread(LPVOID);
VOID GetAnswerToRequest(LPTSTR, LPTSTR, LPDWORD);

DWORD MainProcessId = GetCurrentProcessId();

void CreateChild1Process() {
    TCHAR applicationName[] = TEXT("\"D:\\Microsoft Visual
Studio\\Workspace\\SysProga\\Lab4_1\\Debug\\Lab4_1.exe\" \"D:\\Microsoft
Visual Studio\\Workspace\\SysProga\\Lab4_1\\one.txt\" \"D:\\Microsoft
Visual Studio\\Workspace\\SysProga\\Lab4_1\\two.txt\"");
    STARTUPINFO si;
    PROCESS_INFORMATION pi;

    ZeroMemory(&si, sizeof(si));
    si.cb = sizeof(si);
    ZeroMemory(&pi, sizeof(pi));

    // Start the child process.
    if (!CreateProcess(NULL,
        applicationName,
        NULL,
        NULL,
        FALSE,
        0,
        NULL,
        NULL,
        &si,
        &pi)
        )
    {
        printf("CreateProcess failed (%d).\n", GetLastError());
        return;
    }
```

```
    // Close process and thread handles.
    CloseHandle(pi.hProcess);
    CloseHandle(pi.hThread);
}

void CreateChild2Process() {
    TCHAR applicationName[] = TEXT("\"D:\\Microsoft Visual
Studio\\Workspace\\SysProga\\Lab4_2\\Debug\\Lab4_2.exe\" \"D:\\Microsoft
Visual Studio\\Workspace\\SysProga\\Lab4_1\\two.txt\"");
    STARTUPINFO si;
    PROCESS_INFORMATION pi;

    ZeroMemory(&si, sizeof(si));
    si.cb = sizeof(si);
    ZeroMemory(&pi, sizeof(pi));

    // Start the child process.
    if (!CreateProcess(NULL,
        applicationName,
        NULL,
        NULL,
        FALSE,
        0,
        NULL,
        NULL,
        &si,
        &pi)
        )
    {
        printf("CreateProcess failed (%d).\n", GetLastError());
        return;
    }

    // Close process and thread handles.
    CloseHandle(pi.hProcess);
    CloseHandle(pi.hThread);
}

int _tmain(VOID)
{
    BOOL   fConnected = FALSE;
    DWORD  dwThreadId = 0;
    HANDLE hPipe = INVALID_HANDLE_VALUE, hThread = NULL;
    LPCTSTR lpszPipename = TEXT("\\\\.\\pipe\\mynamedpipe");
```

```
    printf("(%d) Parent process running.... \n\n", MainProcessId);

    CreateChild1Process();
    CreateChild2Process();


    _tprintf(TEXT("\n(%d) Pipe Server: Main thread awaiting client
connection on %s\n\n"), MainProcessId, lpszPipename);

    for (;;)
    {
        hPipe = CreateNamedPipe(
            lpszPipename,
            PIPE_ACCESS_DUPLEX,
            PIPE_TYPE_MESSAGE |
            PIPE_READMODE_MESSAGE |
            PIPE_WAIT,
            PIPE_UNLIMITED_INSTANCES,
            BUFSIZE,
            BUFSIZE,
            0,
            NULL);

        if (hPipe == INVALID_HANDLE_VALUE)
        {
            _tprintf(TEXT("CreateNamedPipe failed, GLE=%d.\n"),
GetLastError());
            return -1;
        }

        fConnected = ConnectNamedPipe(hPipe, NULL) ?
            TRUE : (GetLastError() == ERROR_PIPE_CONNECTED);

        if (fConnected)
        {
            printf("\n\n(%d) Client connected, creating a processing
thread.\n", MainProcessId);

            // Create a thread for this client.
            hThread = CreateThread(
                NULL,
                0,
                InstanceThread,
                (LPVOID)hPipe,
                0,
                &dwThreadId);
```

```
            if (hThread == NULL)
            {
                _tprintf(TEXT("CreateThread failed, GLE=%d.\n"),
GetLastError());
                return -1;
            }
            else CloseHandle(hThread);
        }
        else
            CloseHandle(hPipe);
    }

    return 0;
}

DWORD WINAPI InstanceThread(LPVOID lpvParam)
{
    HANDLE hHeap = GetProcessHeap();
    TCHAR* pchRequest = (TCHAR*)HeapAlloc(hHeap, 0, BUFSIZE *
sizeof(TCHAR));
    TCHAR* pchReply = (TCHAR*)HeapAlloc(hHeap, 0, BUFSIZE *
sizeof(TCHAR));

    DWORD cbBytesRead = 0, cbReplyBytes = 0, cbWritten = 0;
    BOOL fSuccess = FALSE;
    HANDLE hPipe = NULL;

    if (lpvParam == NULL)
    {
        printf("\nERROR - Pipe Server Failure:\n");
        printf("   InstanceThread got an unexpected NULL value in
lpvParam.\n");
        printf("   InstanceThread exitting.\n");
        if (pchReply != NULL) HeapFree(hHeap, 0, pchReply);
        if (pchRequest != NULL) HeapFree(hHeap, 0, pchRequest);
        return (DWORD)-1;
    }

    if (pchRequest == NULL)
    {
        printf("\nERROR - Pipe Server Failure:\n");
        printf("   InstanceThread got an unexpected NULL heap
allocation.\n");
        printf("   InstanceThread exitting.\n");
        if (pchReply != NULL) HeapFree(hHeap, 0, pchReply);
```

```
        return (DWORD)-1;
    }

    if (pchReply == NULL)
    {
        printf("\nERROR - Pipe Server Failure:\n");
        printf("   InstanceThread got an unexpected NULL heap
allocation.\n");
        printf("   InstanceThread exitting.\n");
        if (pchRequest != NULL) HeapFree(hHeap, 0, pchRequest);
        return (DWORD)-1;
    }

    hPipe = (HANDLE)lpvParam;

    while (1)
    {
        fSuccess = ReadFile(
            hPipe,
            pchRequest,
            BUFSIZE * sizeof(TCHAR),
            &cbBytesRead,
            NULL);

        if (!fSuccess || cbBytesRead == 0)
        {
            if (GetLastError() == ERROR_BROKEN_PIPE)
            {
                _tprintf(TEXT("\n(%d) InstanceThread: client
disconnected.\n"), MainProcessId);
            }
            else
            {
                _tprintf(TEXT("InstanceThread ReadFile failed,
GLE=%d.\n"), GetLastError());
            }
            break;
        }

        GetAnswerToRequest(pchRequest, pchReply, &cbReplyBytes);
    }

    FlushFileBuffers(hPipe);
    DisconnectNamedPipe(hPipe);
    CloseHandle(hPipe);
```

```
    HeapFree(hHeap, 0, pchRequest);
    HeapFree(hHeap, 0, pchReply);

    return 1;
}

VOID GetAnswerToRequest(LPTSTR pchRequest,
    LPTSTR pchReply,
    LPDWORD pchBytes)
{
    _tprintf(TEXT("\n(%d) Client Request String:\n    \"%s\"\n"),
MainProcessId, pchRequest);

    // Check the outgoing message to make sure it's not too long for the
buffer.
    if (FAILED(StringCchCopy(pchReply, BUFSIZE, TEXT("default answer
from server"))))
    {
        *pchBytes = 0;
        pchReply[0] = 0;
        printf("StringCchCopy failed, no outgoing message.\n");
        return;
    }
    *pchBytes = (lstrlen(pchReply) + 1) * sizeof(TCHAR);
}
```

Код (Child 1 Process):

```
#include <tchar.h>
#include <windows.h>
#include <stdio.h>
#include <strsafe.h>

#define BUFSIZE 4096

DWORD Child1ProcessId = GetCurrentProcessId();

int _tmain(int argc, TCHAR* argv[])
{
    HANDLE hFile;
    HANDLE hAppend;
    DWORD  dwBytesRead, dwBytesWritten, dwPos;
    BYTE   buff[BUFSIZE];
```

```
    if (argc != 3)
    {
        _tprintf(TEXT("Usage: %s <data file> <copy file>\n"), argv[0]);
        return 1;
    }

    printf("(%d) Child1 process running.... \n", Child1ProcessId);

    printf("(%d) Sleep for 5 secs... \n", Child1ProcessId);
    Sleep(5000);

    // Connect to Named Pipe.
    HANDLE hPipe;
    TCHAR  chBuf[BUFSIZE];
    BOOL   fSuccess = FALSE;
    DWORD  cbRead, cbToWrite, cbWritten, dwMode;
    LPCTSTR lpszPipename = TEXT("\\\\.\\pipe\\mynamedpipe");

    while (1)
    {
        hPipe = CreateFile(
            lpszPipename,
            GENERIC_READ |
            GENERIC_WRITE,
            0,
            NULL,
            OPEN_EXISTING,
            0,
            NULL);

        if (hPipe != INVALID_HANDLE_VALUE)
            break;

        if (GetLastError() != ERROR_PIPE_BUSY)
        {
            _tprintf(TEXT("(%d) Could not open pipe. GLE=%d\n"),
Child1ProcessId, GetLastError());
            return -1;
        }

        if (!WaitNamedPipe(lpszPipename, 20000))
        {
            printf("(%d) Could not open pipe: 20 second wait timed
out.", Child1ProcessId);
            return -1;
        }
```

```
    }

    dwMode = PIPE_READMODE_MESSAGE;
    fSuccess = SetNamedPipeHandleState(
        hPipe,
        &dwMode,
        NULL,
        NULL);

    if (!fSuccess)
    {
        _tprintf(TEXT("SetNamedPipeHandleState failed. GLE=%d\n"),
GetLastError());
        return -1;
    }

    // Open the existing file.

    hFile = CreateFile(argv[1],
        GENERIC_READ,
        0,
        NULL,
        OPEN_EXISTING,
        FILE_ATTRIBUTE_NORMAL,
        NULL);

    if (hFile == INVALID_HANDLE_VALUE)
    {
        printf("Could not open One.txt.");
        return 1;
    }

    // Create a new file.

    hAppend = CreateFile(argv[2],
        FILE_WRITE_DATA,
        FILE_SHARE_READ,
        NULL,
        CREATE_ALWAYS,
        FILE_ATTRIBUTE_NORMAL,
        NULL);

    if (hAppend == INVALID_HANDLE_VALUE)
    {
        printf("Could not open Two.txt.");
        return 1;
```

```
    }

    while (ReadFile(hFile, buff, sizeof(buff), &dwBytesRead, NULL)
        && dwBytesRead > 0)
    {
        dwPos = SetFilePointer(hAppend, 0, NULL, FILE_END);
        LockFile(hAppend, dwPos, 0, dwBytesRead, 0);
        WriteFile(hAppend, buff, dwBytesRead, &dwBytesWritten, NULL);
        UnlockFile(hAppend, dwPos, 0, dwBytesRead, 0);
    }

    // Close both files.

    CloseHandle(hFile);
    CloseHandle(hAppend);

    // Send a message to the pipe server.

    TCHAR lpvMessage[MAX_PATH];
    StringCchPrintf(lpvMessage, MAX_PATH, TEXT("(%d) Created file %s"),
Child1ProcessId, argv[2]);

    cbToWrite = (lstrlen(lpvMessage) + 1) * sizeof(TCHAR);

    fSuccess = WriteFile(
        hPipe,
        lpvMessage,
        cbToWrite,
        &cbWritten,
        NULL);

    if (!fSuccess)
    {
        _tprintf(TEXT("(%d) WriteFile to pipe failed. GLE=%d\n"),
Child1ProcessId, GetLastError());
        return -1;
    }

    // Close Pipe.
    CloseHandle(hPipe);

      return 0;
}
```

Код (Child 2 Process):

```
#include <tchar.h>
#include <windows.h>
#include <stdio.h>
#include <strsafe.h>
#include <string>
#include <atlstr.h>

#define BUFSIZE 4096
#define MAX_THREADS 2

DWORD Child2ProcessId = GetCurrentProcessId();
int iMatrix[3][3];
TCHAR sTargetFilePath[MAX_PATH];

HANDLE ghWriteEvent;
DWORD WINAPI FirstThreadFunction(LPVOID lpParam);
DWORD WINAPI SecondThreadFunction(LPVOID lpParam);

HANDLE hPipe;
TCHAR  chBuf[BUFSIZE];
BOOL   fSuccess = FALSE;
DWORD  cbRead, cbToWrite, cbWritten, dwMode;
LPCTSTR lpszPipename = TEXT("\\\\.\\pipe\\mynamedpipe");

int _tmain(int argc, TCHAR* argv[])
{
    if (argc != 2)
    {
        _tprintf(TEXT("Usage: %s <target file>\n"), argv[0]);
        return 1;
    }

    StringCchCopy(sTargetFilePath, MAX_PATH, argv[1]);

    printf("(%d) Child2 process running.... \n", Child2ProcessId);

    printf("(%d) Sleep for 10 secs... \n", Child2ProcessId);
    Sleep(10000);

    // Connect to Named Pipe.
    while (1)
    {
        hPipe = CreateFile(
            lpszPipename,
            GENERIC_READ |
```

```
            GENERIC_WRITE,
            0,
            NULL,
            OPEN_EXISTING,
            0,
            NULL);

        if (hPipe != INVALID_HANDLE_VALUE)
            break;

        if (GetLastError() != ERROR_PIPE_BUSY)
        {
            _tprintf(TEXT("(%d) Could not open pipe. GLE=%d\n"),
Child2ProcessId, GetLastError());
            return -1;
        }

        if (!WaitNamedPipe(lpszPipename, 20000))
        {
            printf("(%d) Could not open pipe: 20 second wait timed
out.", Child2ProcessId);
            return -1;
        }
    }

    dwMode = PIPE_READMODE_MESSAGE;
    fSuccess = SetNamedPipeHandleState(
        hPipe,
        &dwMode,
        NULL,
        NULL);

    if (!fSuccess)
    {
        _tprintf(TEXT("SetNamedPipeHandleState failed. GLE=%d\n"),
GetLastError());
        return -1;
    }

    // Create an Event.
    ghWriteEvent = CreateEvent(
        NULL,
        TRUE,
        FALSE,
        TEXT("WriteEvent")
    );
```

```
    if (ghWriteEvent == NULL)
    {
        printf("CreateEvent failed (%d)\n", GetLastError());
        return 1;
    }

    // Create Threads.
    DWORD   dwThreadIdArray[MAX_THREADS];
    HANDLE  hThreadArray[MAX_THREADS];

    hThreadArray[0] = CreateThread(
        NULL,
        0,
        FirstThreadFunction,
        0,
        0,
        &dwThreadIdArray[0]);

    hThreadArray[1] = CreateThread(
        NULL,
        0,
        SecondThreadFunction,
        0,
        0,
        &dwThreadIdArray[1]);

    if (hThreadArray[0] == NULL || hThreadArray[1] == NULL)
    {
        _tprintf(TEXT("CreateThread Error"));
        ExitProcess(3);
    }

    WaitForMultipleObjects(MAX_THREADS, hThreadArray, TRUE, INFINITE);

    // Close all thread handles and free memory allocations.
    CloseHandle(ghWriteEvent);
    CloseHandle(hThreadArray[0]);
    CloseHandle(hThreadArray[1]);

    // Close Pipe.
    CloseHandle(hPipe);

    return 0;
}
```

```
DWORD WINAPI FirstThreadFunction(LPVOID lpParam)
{
    HANDLE hFile;
    DWORD  dwBytesRead;
    BYTE   buff[BUFSIZE];
    BOOL   bSuccess;

    DWORD Child1ThreadId = GetCurrentThreadId();
    printf("\n(%d - %d) First thread creating matrix...\n",
Child2ProcessId, Child1ThreadId);


    hFile = CreateFile(sTargetFilePath,
        GENERIC_READ,
        0,
        NULL,
        OPEN_EXISTING,
        FILE_ATTRIBUTE_NORMAL,
        NULL);

    if (hFile == INVALID_HANDLE_VALUE)
    {
        printf("Could not open Two.txt \n");
        return 1;
    }

    bSuccess = ReadFile(hFile, buff, BUFSIZE, &dwBytesRead, NULL);
    if (!bSuccess) {
        printf("Error reading \n");
        return 1;
    }

    buff[dwBytesRead] = '\0';

    int i = 0, a = 0, b = 0;
    char digits[10] = { '0', '1', '2', '3', '4', '5', '6', '7', '8', '9'
};
    BOOL wfinished = FALSE;
    while (!wfinished)
    {
        if (buff[i] == '\0')
        {
            wfinished = TRUE;
            break;
        }
        for (int j = 0; j < 10; j++)
```

```
        {
            if (buff[i] == digits[j])
            {
                iMatrix[a][b] = j;
                b++;
                if (b > 2)
                {
                    b = 0;
                    a++;
                }
                if (a > 3)
                    wfinished = TRUE;
                break;
            }
        }
        i++;
    }

    // Add sleep to show sync.
    Sleep(3000);

    _tprintf(TEXT("(%d - %d) Matrix filled by numbers from file\n"),
Child2ProcessId, Child1ThreadId);

    CloseHandle(hFile);

    // Set ghWriteEvent to signaled.

    if (!SetEvent(ghWriteEvent))
    {
        printf("SetEvent failed (%d)\n", GetLastError());
        return 1;
    }

    return 0;
}

DWORD WINAPI SecondThreadFunction(LPVOID lpParam)
{
    std::string sMatrix;
    std::string sMatrixMul;
    TCHAR szMatrixMul[MAX_PATH];
    DWORD dwWaitResult;
    int iMmul = 1;

    DWORD Child2ThreadId = GetCurrentThreadId();
```

```
    printf("(%d - %d) Second thread waiting for write event...\n",
Child2ProcessId, Child2ThreadId);

    dwWaitResult = WaitForSingleObject(
        ghWriteEvent,
        INFINITE);

    switch (dwWaitResult)
    {
    case WAIT_OBJECT_0:
        for (int i = 0; i < 3; i++)
        {
            for (int j = 0; j < 3; j++)
            {
                iMmul *= iMatrix[i][j];
                sMatrix += std::to_string(iMatrix[i][j]) + " ";
            }
            sMatrixMul += std::to_string(iMmul) + " ";
            iMmul = 1;
            sMatrix += "\n";
        }
        sMatrixMul.pop_back();
        printf("(%d - %d) Matrix:\n%s", Child2ProcessId, Child2ThreadId,
sMatrix.c_str());

        // Send a message to the pipe server.

        TCHAR lpvMessage[MAX_PATH];
        _tcscpy_s(szMatrixMul, CA2T(sMatrixMul.c_str()));
        StringCchPrintf(lpvMessage, MAX_PATH, TEXT("(%d - %d) Matrix
multiplications: %s"), Child2ProcessId, Child2ThreadId, szMatrixMul);

        cbToWrite = (lstrlen(lpvMessage) + 1) * sizeof(TCHAR);

        fSuccess = WriteFile(
            hPipe,
            lpvMessage,
            cbToWrite,
            &cbWritten,
            NULL);

        if (!fSuccess)
        {
            _tprintf(TEXT("(%d) WriteFile to pipe failed. GLE=%d\n"),
Child2ProcessId, GetLastError());
            return -1;
```

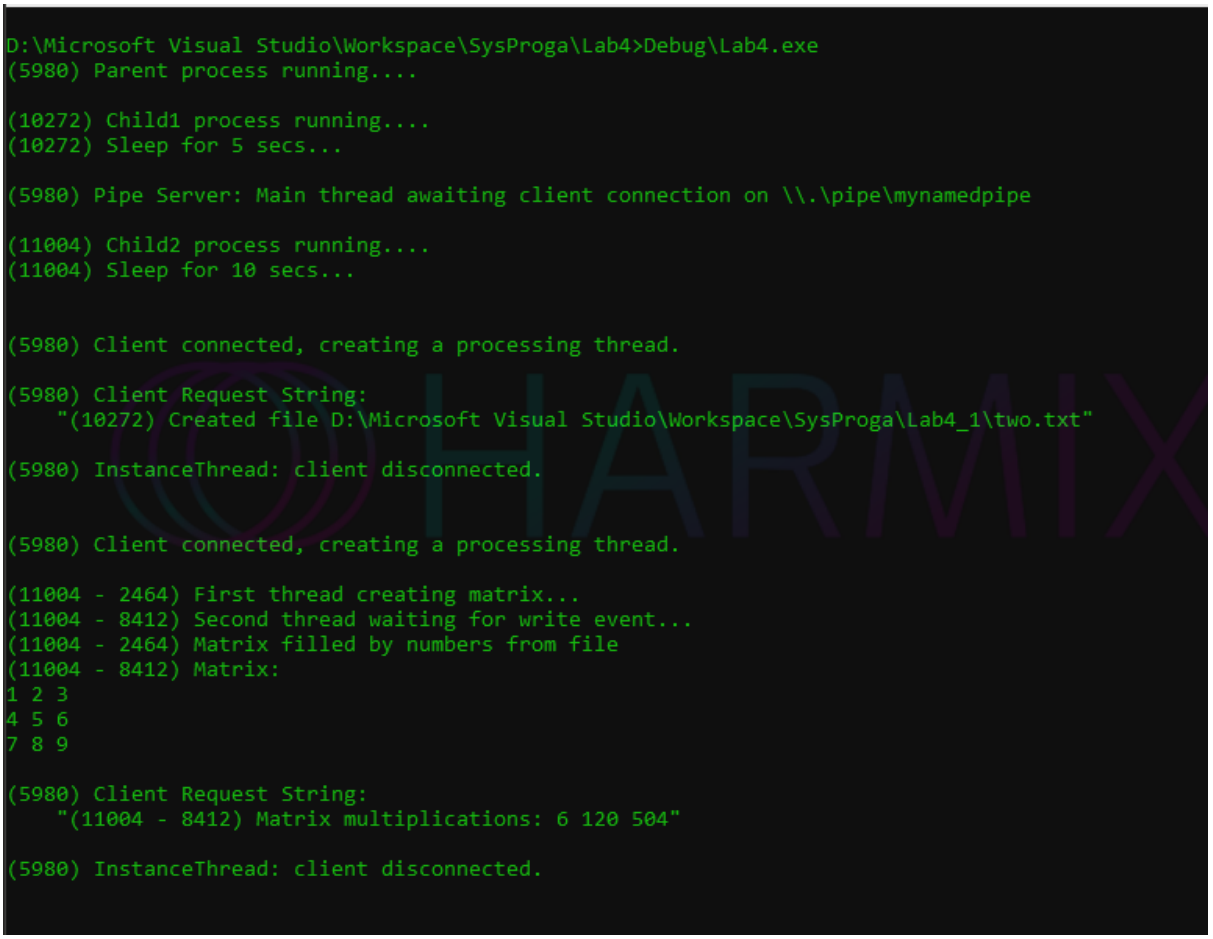
```
        }

        break;

    default:
        printf("Wait error (%d)\n", GetLastError());
        return 1;
    }

    return 0;
}
```

Скріншоти:



Повний код можна знайти у GitHub-репозиторії:
https://github.com/NazarPonochevnyi/Programming-Labs/blob/master/System%20Programming/Lab4/lab4.cpp