Laborator 6

Gavan Eduard
CR3.2A

Acest document descrie experienta mea in realizarea laboratorului 6 la disciplina Sisteme de Operare.
Tema a presupus implementarea comunicarii intre doua procese folosind memorie partajata si un mutex pentru sincronizare. Programul ruleaza pe Windows si utilizeaza API-uri precum **CreateFileMappingA**, **MapViewOfFile** si **CreateMutexA** pentru a crea o zona de memorie comuna si a controla accesul la aceasta.

Primul proces creeaza memoria si mutex-ul, apoi lanseaza automat al doilea proces folosind **CreateProcessA**. Ambele procese executa aceeasi logica pe zona de memorie partajata, incrementand o valoare comuna intr-un mod sincronizat.

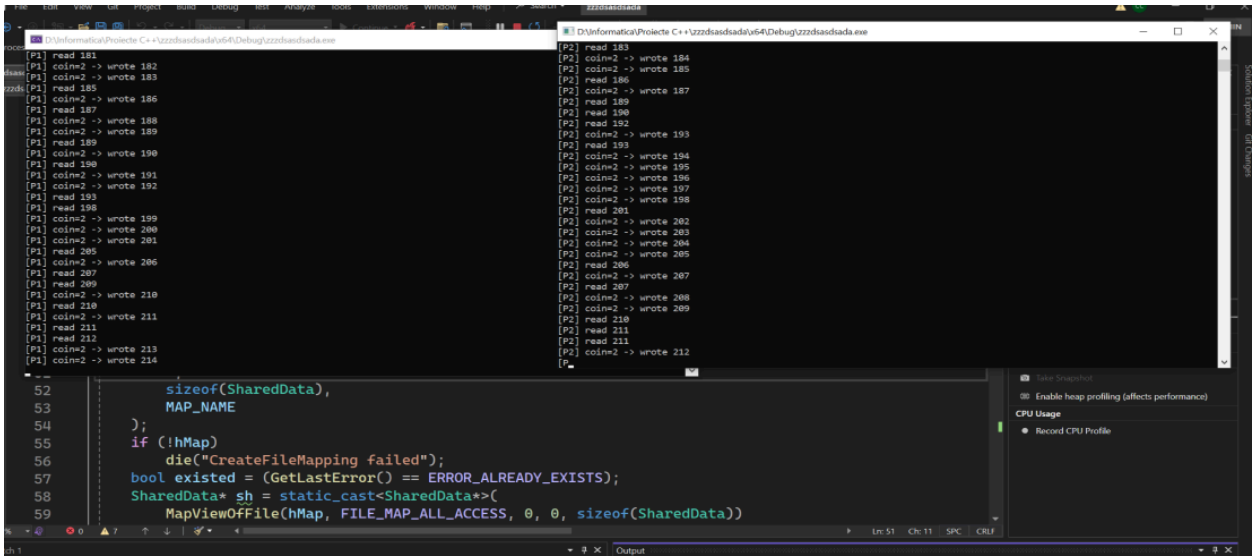
Codul final al programului:

```
// Primul proces creeaza memoria+mutexul si lanseaza automat al doilea proces.
// Al doilea proces se conecteaza si numara impreuna cu primul.
#include <windows.h>
#include <iostream>
#include <string>
#include <cstdlib>
#include <ctime>
#define MAX_N 1000
using namespace std;
struct SharedData {
    int value;
};
static void die(const string& msg) {
    cerr << msg << " (error=" << GetLastError() << ")\n";
    ExitProcess(1);
}
static bool spawn_second_process() {
    char exePath[MAX_PATH];
    GetModuleFileNameA(NULL, exePath, MAX_PATH);
    string cmd = string("\"") + exePath + "\" --child";
    // buffer mutabil pentru CreateProcessA
    char cmdMutable[512];
    strcpy_s(cmdMutable, cmd.c_str());
    STARTUPINFOA si{};
    si.cb = sizeof(si);
    PROCESS_INFORMATION pi{};
    BOOL ok = CreateProcessA(
        NULL,
        cmdMutable,
        NULL, NULL,
```

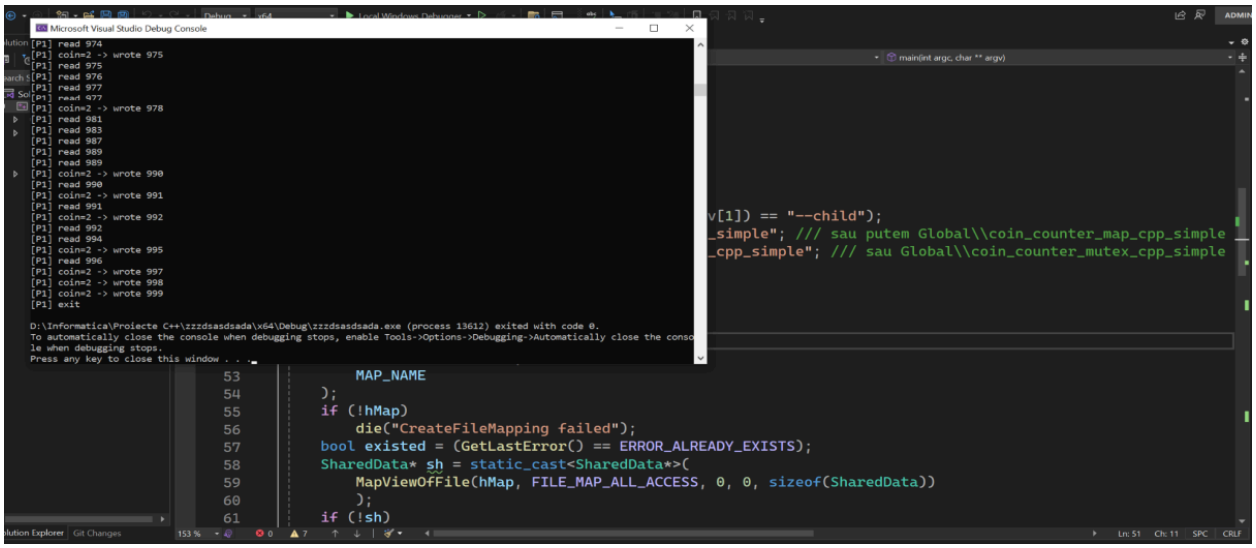
```
        FALSE,
        CREATE_NEW_CONSOLE,
        NULL, NULL,
        &si, &pi
    );
    if (!ok)
        return false;
    CloseHandle(pi.hThread);
    CloseHandle(pi.hProcess);
    return true;
}

int main(int argc, char** argv) {
    bool isChild = (argc >= 2 && std::string(argv[1]) == "--child");
    const char* MAP_NAME = "coin_counter_map_cpp_simple"; /// sau putem Global\\coin_counter_map_cpp_simple
-> insa merge doar daca ii dam run ca administrator
    const char* MUTEX_NAME = "coin_counter_mutex_cpp_simple"; /// sau Global\\coin_counter_mutex_cpp_simple
-> (run as administrator ca sa mearga)
    HANDLE hMap = CreateFileMappingA(
        INVALID_HANDLE_VALUE,
        NULL,
        PAGE_READWRITE,
        0,
        sizeof(SharedData),
        MAP_NAME
    );
    if (!hMap)
        die("CreateFileMapping failed");
    bool existed = (GetLastError() == ERROR_ALREADY_EXISTS);
    SharedData* sh = static_cast<SharedData*>(
        MapViewOfFile(hMap, FILE_MAP_ALL_ACCESS, 0, 0, sizeof(SharedData))
        );
    if (!sh)
        die("MapViewOfFile failed");
    HANDLE hMutex = CreateMutexA(NULL, FALSE, MUTEX_NAME);
    if (!hMutex)
        die("CreateMutex failed");
    // daca NU exista mapping-ul, inseamna ca suntem primul proces
    int id = existed ? 2 : 1;
    // initializare doar in primul proces
    if (!existed) {
        WaitForSingleObject(hMutex, INFINITE);
        sh->value = 0;
        ReleaseMutex(hMutex);
        // primul proces porneste automat al doilea
        if (!spawn_second_process()) {
            cerr << "Warning: nu am putut porni al doilea proces automat.\n";
            cerr << "Poti rula manual acelasi exe inca o data.\n";
        }
    }
    // seed random diferit pe proces
    unsigned seed = (unsigned)time(NULL)
        ^ (unsigned)GetCurrentProcessId()
        ^ (unsigned)(id * 12345);
```

```
    std::srand(seed);
    while (true) {
        DWORD w = WaitForSingleObject(hMutex, INFINITE);
        if (w != WAIT_OBJECT_0) die("WaitForSingleObject failed");
        int elem_curent = sh->value;
        if (elem_curent >= MAX_N) {
            ReleaseMutex(hMutex);
            break;
        }
        cout << "[P" << id << "] read " << elem_curent << "\n";
        while (elem_curent < MAX_N) {
            int coin = (rand() % 2) + 1; // 1 sau 2
            if (coin != 2)
                break;
            elem_curent++;
            sh->value = elem_curent;
            cout << "[P" << id << "] coin=2 -> wrote " << elem_curent << "\n";
        }
        ReleaseMutex(hMutex);
        Sleep(50);
    }
    UnmapViewOfFile(sh);
    CloseHandle(hMutex);
    CloseHandle(hMap);
    cout << "[P" << id << "] exit\n";
    return 0;
}
```



Aceasta captura arata ambele procese in executie simultana. Se observa cum P1 si P2 citesc si scriu alternativ aceeasi valoare in memoria partajata, folosind mutex-ul pentru sincronizare.



In aceasta captura putem vedea finalizarea procesului P1 pana la valoarea 999, moment in care programul se opreste. Incrementarea sincronizata pana la 1000 confirma functionarea corecta.

Github: https://github.com/edigolan/Cerinta6_SO