Белорусский государственный технологический университет
Факультет информационных технологий
Кафедра программной инженерии

Лабораторная работа 6

По дисциплине «Операционные системы»

На тему «Синхронизация»

Выполнил:
Студент 3 курса 9 группы
Павлович Ян Андреевич
Преподаватель: Савельева М.Г.

Минск, 2025

Введение

Целью данной работы является приобретение практических навыков работы с процессами и потоками в операционных системах Windows и Linux. Особое внимание уделяется изучению механизмов синхронизации и их влиянию на производительность и корректность работы приложений.

В рамках работы необходимо разработать набор консольных программ, демонстрирующих работу различных средств синхронизации, включая критические секции, мьютексы, семафоры, события, условные переменные и барьеры. Для платформы Windows предполагается реализовать синхронизацию потоков с использованием критических секций, управление процессами через мьютексы и бинарные семафоры, координацию запуска дочерних процессов с объектами событий, задачу «производитель–потребитель» с критическими секциями и условными переменными, а также параллельную обработку матриц с применением барьеров и атомарных операций. Кроме того, программы должны обеспечивать вывод статистики и состояния синхронизации, а также контроль доступа к общим ресурсам.

Для платформы Linux необходимо создать аналогичные приложения на основе POSIX API с учетом особенностей потоков и процессов в данной ОС, а также реализовать задачи с использованием POSIX Threads и межпроцессных механизмов. В завершение работы проводится сравнительный анализ поведения потоков и процессов при различных методах синхронизации, эффективности используемых механизмов, влияния синхронизации на производительность, блокировки и конкуренцию за ресурсы, а также различий в реализации между Windows и Linux.

Инструменты и материалы:

- Parallels Desktop
- Командная оболочка cmd и PowerShell
- Process Explorer
- Компилятор g++ и среда Visual Studio
- Утилиты Linux: ps, top, cat /proc, kill, sleep
- Документация по WinAPI и POSIX Threads, материалы по синхронизации в ОС

1 Windows

1.1 Приложение Lab-06a

Приложение Lab-06a запускает два дочерних потока A и B. Все потоки (main, A и B) выполняют циклы по 90 итераций, в ходе которых выводятся имена потоков, номера текущих итераций и одна из букв имени пользователя (аналогично тому, как это реализовывалось в лабораторной работе №4). Между итерациями выполняется задержка в 100 мс.

```
=== LAB-06a APPLICATION: WINDOWS THREAD SYNCHRONIZATION ===
Username: yan
Number of iterations: 90
Delay: 100 ms
Critical section: iterations 30-60
============================================================

Threads created:
  - Thread A (child)
[Thread A] Thread started. TID: 5036, PID: 6544
[Thread A] iteration:   1, PID: 6544, TID: 5036, Character: y
  - Thread B (child)

Starting main thread...
```

Рисунок 1.1 – Результат работы Lab-06a

В данном приложении имеется разделяемый ресурс, требующий эксклюзивного доступа: это промежуток с 30 по 60 итерации любого потока. Выполнение этих итераций допускается только одним потоком одновременно. При этом выполнение потоком A итераций с 30 по 60 и потоком B итераций с 61 по 90 является допустимым, так как разделяемый ресурс используется только одним потоком.

```
[Thread A] iteration:  59, PID: 6544, TID: 5036, Character: a
[Thread A] iteration:  60, PID: 6544, TID: 5036, Character: n
[Thread A] <<< Exiting critical section (iteration 60)
[Thread A] iteration:  61, PID: 6544, TID: 5036, Character: y
[Main   ] >>> Entering critical section (iteration 30)
[Main   ] iteration:  30, PID: 6544, TID: 7624, Character: n
[Main   ] iteration:  31, PID: 6544, TID: 7624, Character: y
```

Рисунок 1.2 – Результат работы Lab-06a

Для обеспечения корректной работы с разделяемым ресурсом синхронизация потоков осуществляется с помощью механизма критической секции.

```
[Thread B] iteration:  88, PID: 6544, TID: 5764, Character: y
[Thread B] iteration:  89, PID: 6544, TID: 5764, Character: a
[Thread B] iteration:  90, PID: 6544, TID: 5764, Character: n
[Thread B] Thread finished. TID: 5764

=== EXECUTION COMPLETED ===
All threads have finished successfully.
Critical section ensured exclusive access
to iterations 30-60 for each thread.
```

Рисунок 1.3 – Результат работы Lab-06a

Программа реализуется в виде консольного приложения, код которого представлен в Листинге А. После компиляции и запуска приложения наблюдается параллельная работа трех потоков (Main, Поток А и Поток В), которые выводят символы поочередно, демонстрируя чередование строк вывода.

На рисунках 1.1-1.3 показан пример работы приложения Lab-06a. При достижении 30-й итерации происходит вход в критическую зону, и одновременно в ней может выполняться только один поток.

1.2 Приложение Lab-06b

Приложение Lab-06b создаёт два дочерних процесса A и B. Главные потоки всех трёх процессов (main, A и B) выполняют циклы по 90 итераций, при этом в своих отдельных консольных окнах они выводят имя процесса, номер текущей итерации и одну из букв имени пользователя. Между итерациями выполняется задержка 100 миллисекунд.

В приложении предусмотрен разделяемый ресурс, требующий эксклюзивного доступа: это диапазон с 30-й по 60-ю итерацию каждого процесса. Эти итерации могут выполняться только одним процессом одновременно. Для организации синхронизации используется механизм мьютекса.

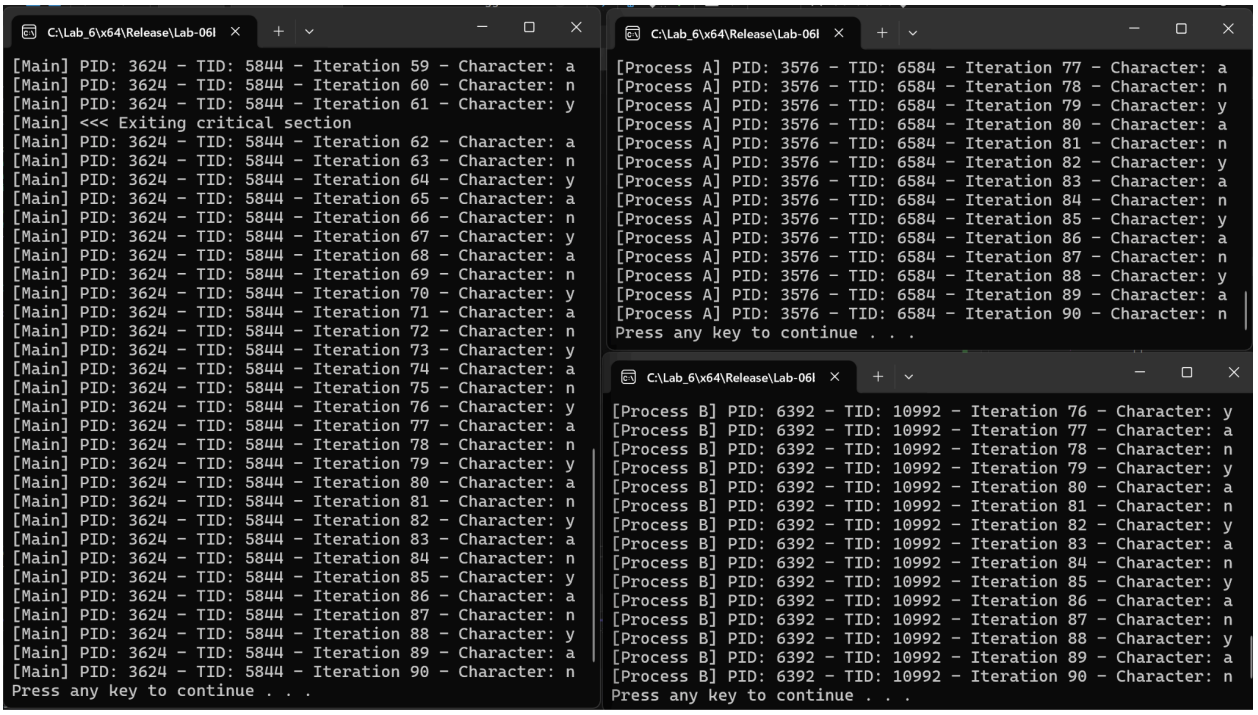

Рисунок 1.4 – Результат работы Lab-06b

Программа реализуется как консольное приложение, код которого представлен в Листинге Б. При запуске главный процесс создаёт именованный мьютекс "Global\Lab06bMutex" и запускает два дочерних процесса, каждый со своей консолью. Все три процесса параллельно

выполняют цикл из 90 итераций, выводя идентификаторы процессов и буквы из имени пользователя с задержкой 100 мс.

Синхронизация ключевого участка выполняется на итерациях 30–60, которые формируют критическую зону. При достижении 30-й итерации процесс захватывает мьютекс с помощью WaitForSingleObject, получая исключительный доступ к критической зоне. Другие процессы при этом блокируются до освобождения мьютекса. По завершении 60-й итерации мьютекс освобождается через ReleaseMutex, позволяя следующему процессу войти в критическую зону. Итерации 1–29 и 61–90 выполняются параллельно, а 30–60 — строго последовательно, что обеспечивает корректное взаимное исключение через механизм мьютекса WinAPI.

На рисунке 1.4 показана работа приложения.

1.3 Приложение Lab-06c

Приложение Lab-06c строится по аналогии с Lab-06b, но использует другой механизм синхронизации. В этом случае управление доступом к разделяемому ресурсу осуществляется с помощью бинарного семафора.

Программа создается как консольное приложение, код которого представлен в Листинге B. Работа приложения показана на рисунке 1.5

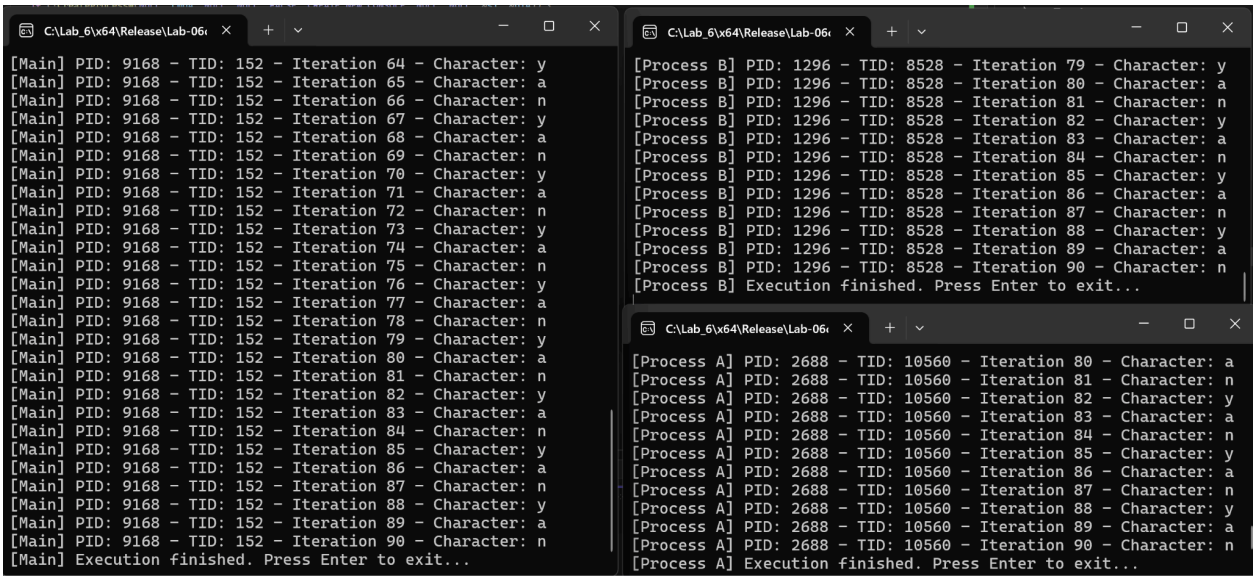

Рисунок 1.5 – Результат работы Lab-06c

После запуска главный процесс создает именованный бинарный семафор и запускает два дочерних процесса. Логика работы аналогична предыдущим приложениям: все процессы выполняют циклы и выводят идентификаторы и буквы имени пользователя.

Однако использование семафора приводит к тому, что вывод разных процессов в консоли может накладываться друг на друга, так как семафор обеспечивает синхронизацию, но не гарантирует строгого чередования строк вывода.

1.4 Приложение Lab-06d

Приложение Lab-06d запускает два дочерних процесса A и B, каждый с собственной консолью для вывода. Главные потоки всех трёх процессов — основной (main) и дочерние (A, B) — выполняют цикл из 90 итераций. На каждой итерации выводится имя процесса, номер текущей итерации и один символ из имени пользователя (аналогично лабораторной работе №4), после чего выполняется задержка 100 миллисекунд.

Работа программы разделена на два этапа. На первом этапе итерации с 1 по 15 выполняются исключительно родительским процессом, в то время как дочерние процессы A и B находятся в ожидании. Синхронизация этого поведения обеспечивается с помощью объекта события (Event) Windows API.

На втором этапе, начиная с 16-й итерации, родительский процесс активирует событие, позволяя дочерним процессам начать выполнение. С этого момента все три процесса работают параллельно, выполняя оставшиеся итерации (16–90) в своих консольных окнах. Таким образом, событие служит механизмом координации запуска параллельной фазы после завершения эксклюзивной работы родительским процессом.

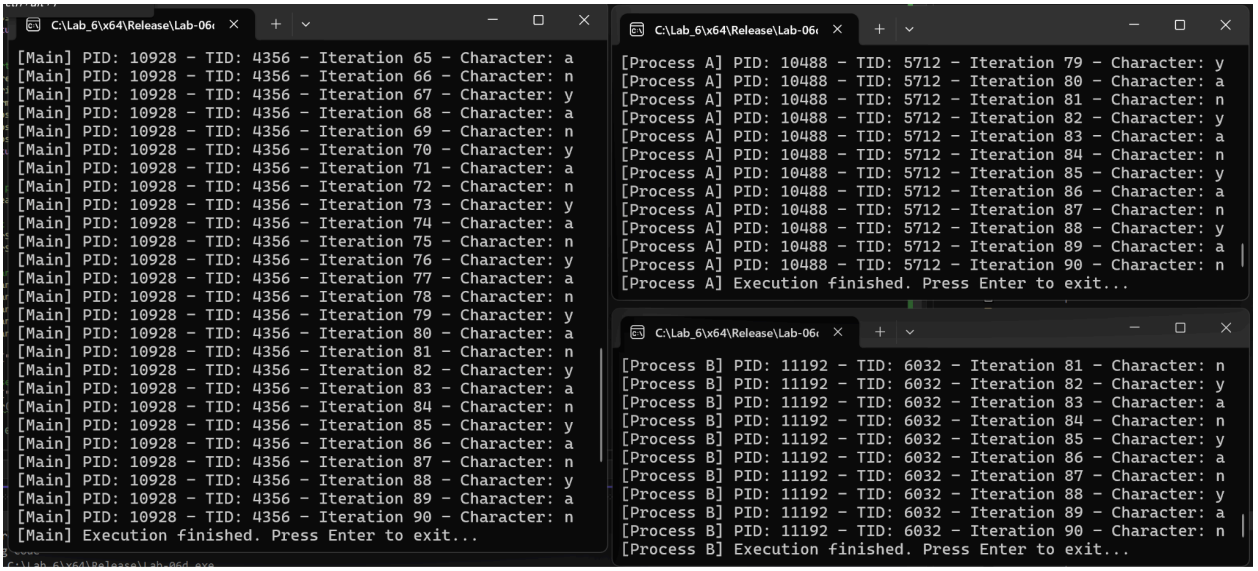

Рисунок 1.6 – Результат работы Lab-06d

Программа создаётся как консольное приложение, код представлен в Листинге Г. В ходе выполнения наблюдается, что на первых 15 итерациях работает только главный процесс, а после сигнала от него дочерние процессы начинают параллельное выполнение.

На рисунке 1.6 показан пример работы приложения Lab-06d. Из-за отсутствия синхронизации вывода между процессами с помощью мьютекса строки из разных консолей могут накладываться друг на друга, создавая визуальный беспорядок. При этом логическая последовательность выполнения каждой программы сохраняется.

Анализ выполненных заданий позволяет сделать следующие выводы: синхронизация потоков проще в реализации, так как они работают в рамках

одного процесса и разделяют общее адресное пространство, что позволяет использовать быстрые механизмы, например, критические секции. Межпроцессная синхронизация требует использования именованных объектов, таких как мьютексы, семафоры или события.

2 Linux

2.1 Приложение Lab-06a

Данное приложение полностью повторяет функциональность версии для Windows, однако вместо использования критической секции для синхронизации потоков применяется мьютекс. Такой подход позволяет обеспечить взаимное исключение при работе с разделяемыми ресурсами, аналогично механизму критической секции, но с использованием объектов синхронизации, подходящих для межпроцессного взаимодействия.

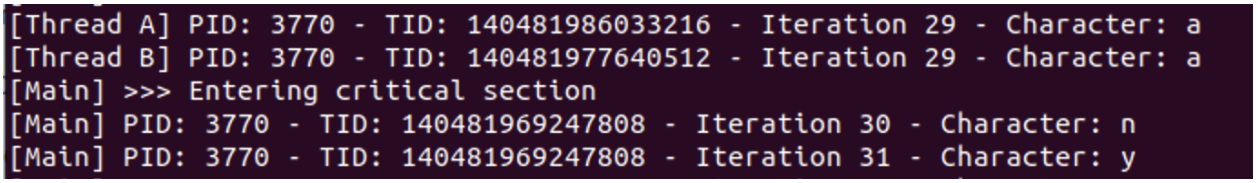

Рисунок 2.1 – Результат работы Lab-06a

Программа реализуется как консольное приложение, код которого представлен в Листинге Д. В процессе работы каждый поток выполняет цикл с выводом имени потока, номера итерации и символа из имени пользователя, а мьютекс гарантирует, что критические участки выполняются только одним потоком одновременно.

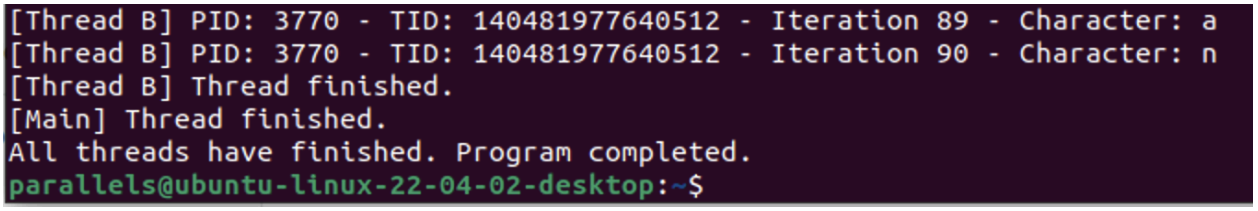

Рисунок 2.2 – Результат работы Lab-06a

После запуска приложения наблюдается поведение, идентичное версии для Windows: потоки выполняются параллельно, но доступ к защищённым участкам кода осуществляется строго последовательно, что обеспечивает корректность работы программы и предотвращает одновременное использование разделяемого ресурса несколькими потоками.

На рисунках 2.1 и 2.2 представлено выполнение приложения Lab-06a, демонстрирующее правильное чередование потоков при работе с мьютексом.

2.2 Приложение Lab-06b

Приложение Lab-06b полностью повторяет функциональность версии Lab-06c для Windows, обеспечивая аналогичное поведение потоков и процессов.

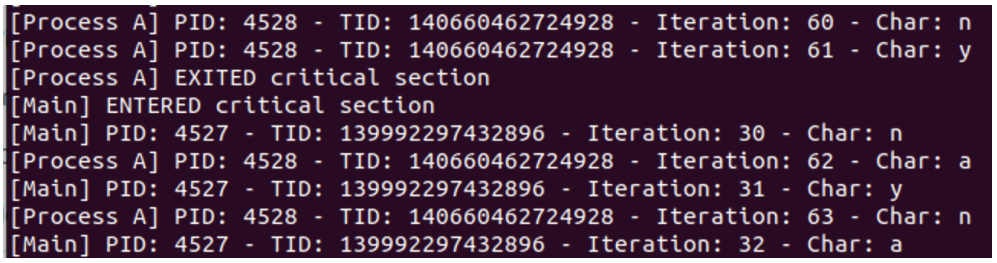

Рисунок 2.3 – Результат работы Lab-06b

Программа создается как консольное приложение, код которого представлен в Листинге Е. После запуска можно наблюдать, что выполнение потоков и процессов аналогично версии для Windows: все процессы выполняют свои циклы, а механизм синхронизации гарантирует корректный доступ к разделяемым ресурсам.

На рисунке 2.3 и 2.4 показан пример работы приложения Lab-06b.

Стоит отметить различия между реализацией в Windows и Linux. В Windows используется единый системный API: CreateThread для потоков и CreateProcess для процессов, работающий через дескрипторы объектов. Объекты синхронизации, такие как мьютексы, семафоры и события, интегрированы в ОС, могут быть именованными через пространство Global\ и управляются системой автоматически. Консоли процессов по умолчанию разделены, что упрощает визуальный вывод информации.

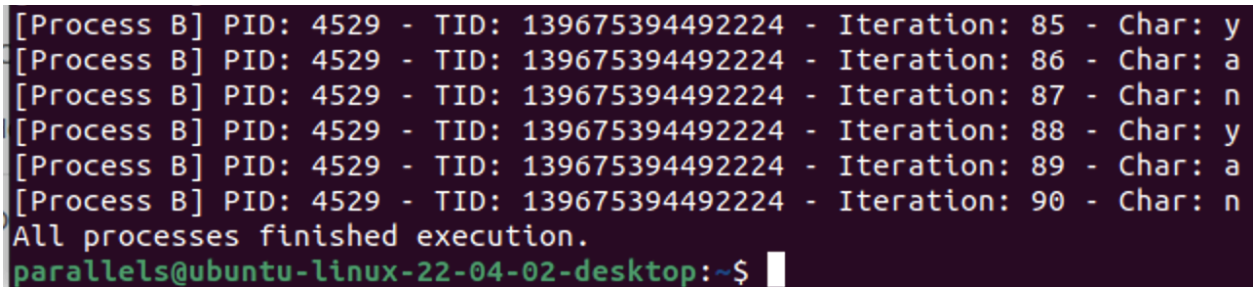

Рисунок 2.4 – Результат работы Lab-06b

В Linux применяются отдельные механизмы: для потоков используется библиотека pthread (pthread_create), а для процессов — классические fork/exec. Синхронизация осуществляется через POSIX-объекты (pthread_mutex_t, sem_t), которые требуют ручной инициализации и явного управления памятью. Именованные объекты размещаются в файловой системе (например, /dev/shm) и должны иметь имена, начинающиеся с символа слэша. Все процессы по умолчанию выводят данные в общую консоль, что создаёт необходимость дополнительной синхронизации вывода для предотвращения наложения сообщений.

Заключение

В ходе выполнения лабораторной работы №6 было получено расширенное представление о механизмах синхронизации, взаимодействия процессов и потоков, а также о различиях в реализации этих механизмов в операционных системах Windows и Linux.

На платформе Windows были реализованы приложения Lab-06a, Lab-06b, Lab-06c и Lab-06d, демонстрирующие различные подходы к синхронизации: от критической секции для защиты общего участка кода до межпроцессного взаимодействия через мьютексы, бинарные семафоры и объекты событий. Особое внимание было уделено управлению доступом к разделяемым ресурсам и координации выполнения между потоками и процессами.

На платформе Linux были разработаны аналогичные приложения, в которых синхронизация реализована с использованием POSIX-примитивов: pthread_mutex_t, pthread_cond_t, pthread_barrier_t и sem_t. В частности, критическая секция была заменена мьютексом, а события — механизмами ожидания и пробуждения потоков через условные переменные. Это позволило отработать навыки переноса многопоточных решений между платформами с учётом различий в системных API.

Сравнительный анализ показал, что при одинаковой логике работы различия между платформами проявляются в деталях реализации: Windows предлагает более высокоуровневые и интегрированные средства, тогда как Linux требует явного управления синхронизацией через POSIX-интерфейсы.

Приложение А – Листинг А.1

```
#include <windows.h>
#include <stdio.h>
#include <string.h>
#define USERNAME "yan"
#define ITERATIONS 90
#define DELAY_MS 100
#define CRITICAL_START 30
#define CRITICAL_END 60
CRITICAL_SECTION cs; // Critical section for synchronization
typedef struct {
    const char* threadName;  // Thread name
    char threadLetter;       // Letter for identification (A, B, M)
} ThreadData;
DWORD WINAPI ThreadFunction(LPVOID lpParam) {
    ThreadData* data = (ThreadData*)lpParam;
    DWORD threadId = GetCurrentThreadId();
    DWORD processId = GetCurrentProcessId();
    int nameLength = (int)strlen(USERNAME);
    printf("[%s] Thread started. TID: %lu, PID: %lu\n",
        data->threadName, threadId, processId);
    for (int i = 0; i < ITERATIONS; i++) {
        int currentIteration = i + 1;
        char currentChar = USERNAME[i % nameLength];
        if (currentIteration == CRITICAL_START) {
            EnterCriticalSection(&cs);
            printf("[%s] >>> Entering critical section (iteration %d)\n",
                data->threadName, currentIteration);
        }
        printf("[%s] iteration: %3d, PID: %lu, TID: %lu, Character: %c\n",
            data->threadName, currentIteration, processId, threadId,
currentChar);
        Sleep(DELAY_MS);
        if (currentIteration == CRITICAL_END) {
            printf("[%s] <<< Exiting critical section (iteration %d)\n",
                data->threadName, currentIteration);
            LeaveCriticalSection(&cs);
        }
    }
    printf("[%s] Thread finished. TID: %lu\n", data->threadName, threadId);
    return 0;
}
int main() {
    printf("=== LAB-06a APPLICATION: WINDOWS THREAD SYNCHRONIZATION ===\n");
    printf("Username: %s\n", USERNAME);
    printf("Number of iterations: %d\n", ITERATIONS);
    printf("Delay: %d ms\n", DELAY_MS);
    printf("Critical section: iterations %d-%d\n", CRITICAL_START,
CRITICAL_END);
    printf("============================================================\n\n");
    InitializeCriticalSection(&cs);
    ThreadData threadA = { "Thread A", 'A' };
    ThreadData threadB = { "Thread B", 'B' };
    ThreadData mainThread = { "Main   ", 'M' };
```

```
    HANDLE hThreadA = CreateThread(NULL, 0, ThreadFunction, &threadA, 0,
NULL);
    HANDLE hThreadB = CreateThread(NULL, 0, ThreadFunction, &threadB, 0,
NULL);
    if (hThreadA == NULL || hThreadB == NULL) {
        printf("ERROR: Failed to create threads!\n");
        DeleteCriticalSection(&cs);
        return 1;
    }
    printf("Threads created:\n");
    printf("  - Thread A (child)\n");
    printf("  - Thread B (child)\n\n");
    printf("Starting main thread...\n\n");
    ThreadFunction(&mainThread);
    printf("\nWaiting for child threads to finish...\n");
    WaitForSingleObject(hThreadA, INFINITE);
    WaitForSingleObject(hThreadB, INFINITE);
    CloseHandle(hThreadA);
    CloseHandle(hThreadB);
    DeleteCriticalSection(&cs);
    printf("\n=== EXECUTION COMPLETED ===\n");
    printf("All threads have finished successfully.\n");
    printf("Critical section ensured exclusive access\n");
    printf("to iterations %d-%d for each thread.\n", CRITICAL_START,
CRITICAL_END);
    return 0;
}
```

Приложение Б – Листинг Б.1

```
#include <windows.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define USERNAME "yan"
#define ITERATIONS 90
#define DELAY_MS 100
#define MUTEX_NAME L"Global\\Lab06bMutex"
void RunThread(const char* name) {
    DWORD pid = GetCurrentProcessId();
    DWORD tid = GetCurrentThreadId();
    size_t nameLen = strlen(USERNAME);
    // Wait until the mutex is available
    HANDLE hMutex = NULL;
    while ((hMutex = OpenMutexW(SYNCHRONIZE, FALSE, MUTEX_NAME)) == NULL) {
        Sleep(10);
    }
    BOOL inCritical = FALSE;
    for (int i = 0; i < ITERATIONS; ++i) {
        if (i + 1 == 30) {
            WaitForSingleObject(hMutex, INFINITE);
            printf("[%s] >>> Entering critical section\n", name);
            inCritical = TRUE;
        }
        char letter = USERNAME[i % nameLen];
        printf("[%s] PID: %lu - TID: %lu - Iteration %d - Character: %c\n",
            name, pid, tid, i + 1, letter);
        Sleep(DELAY_MS);
        if (i + 1 == 61 && inCritical) {
            printf("[%s] <<< Exiting critical section\n", name);
            ReleaseMutex(hMutex);
            inCritical = FALSE;
        }
    }
    system("pause");
    CloseHandle(hMutex);
}
int main(int argc, char* argv[]) {
    SetConsoleOutputCP(1251);
    SetConsoleCP(1251);
    // Child process handling
    if (argc == 2) {
        if (strcmp(argv[1], "childA") == 0) {
            RunThread("Process A");
            return 0;
        }
        if (strcmp(argv[1], "childB") == 0) {
            RunThread("Process B");
            return 0;
        }
    }
    // Parent process
    SECURITY_ATTRIBUTES sa;
```

```
    sa.nLength = sizeof(SECURITY_ATTRIBUTES);
    sa.lpSecurityDescriptor = NULL;
    sa.bInheritHandle = TRUE;
    HANDLE hMutex = CreateMutexW(&sa, FALSE, MUTEX_NAME);
    if (hMutex == NULL) {
        fprintf(stderr, "ERROR: Failed to create mutex\n");
        return 1;
    }
    STARTUPINFOW si;
    ZeroMemory(&si, sizeof(si));
    si.cb = sizeof(si);
    PROCESS_INFORMATION piA, piB;
    wchar_t cmdA[MAX_PATH + 20];
    wchar_t cmdB[MAX_PATH + 20];
    wchar_t exePath[MAX_PATH];
    GetModuleFileNameW(NULL, exePath, MAX_PATH);
    swprintf(cmdA, MAX_PATH + 20, L"\"%s\" childA", exePath);
    swprintf(cmdB, MAX_PATH + 20, L"\"%s\" childB", exePath);
    if (!CreateProcessW(NULL, cmdA, NULL, NULL, FALSE, CREATE_NEW_CONSOLE,
NULL, NULL, &si, &piA)) {
        fprintf(stderr, "ERROR: Failed to start Process A\n");
        CloseHandle(hMutex);
        return 1;
    }
    if (!CreateProcessW(NULL, cmdB, NULL, NULL, FALSE, CREATE_NEW_CONSOLE,
NULL, NULL, &si, &piB)) {
        fprintf(stderr, "ERROR: Failed to start Process B\n");
        CloseHandle(piA.hProcess);
        CloseHandle(piA.hThread);
        CloseHandle(hMutex);
        return 1;
    }
    // Run main thread
    RunThread("Main");
    // Wait for child processes to finish
    WaitForSingleObject(piA.hProcess, INFINITE);
    WaitForSingleObject(piB.hProcess, INFINITE);
    // Clean up handles
    CloseHandle(piA.hProcess);
    CloseHandle(piA.hThread);
    CloseHandle(piB.hProcess);
    CloseHandle(piB.hThread);
    CloseHandle(hMutex);
    printf("All processes have finished successfully.\n");
    // Delay to view results
    system("pause");
    return 0;
}
```

Приложение В – Листинг В.1

```
#include <windows.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define USERNAME "yan"
#define ITERATIONS 90
#define DELAY_MS 100
#define SEMAPHORE_NAME L"Global\\Lab06cSemaphore"
void RunThread(const char* name) {
    DWORD pid = GetCurrentProcessId();
    DWORD tid = GetCurrentThreadId();
    size_t nameLen = strlen(USERNAME);
    // Wait for the semaphore to become available
    HANDLE hSemaphore = NULL;
    while ((hSemaphore = OpenSemaphoreW(SYNCHRONIZE | SEMAPHORE_MODIFY_STATE,
FALSE, SEMAPHORE_NAME)) == NULL) {
        Sleep(10);
    }
    BOOL inCritical = FALSE;
    for (int i = 0; i < ITERATIONS; ++i) {
        if (i + 1 == 30) {
            WaitForSingleObject(hSemaphore, INFINITE); // Acquire semaphore
            printf("[%s] >>> Entering critical section\n", name);
            inCritical = TRUE;
        }
        char letter = USERNAME[i % nameLen];
        printf("[%s] PID: %lu - TID: %lu - Iteration %d - Character: %c\n",
            name, pid, tid, i + 1, letter);
        Sleep(DELAY_MS);
        if (i + 1 == 61 && inCritical) {
            printf("[%s] <<< Exiting critical section\n", name);
            ReleaseSemaphore(hSemaphore, 1, NULL); // Release semaphore
            inCritical = FALSE;
        }
    }
    // Pause to keep the console open
    printf("[%s] Execution finished. Press Enter to exit...\n", name);
    getchar();
    CloseHandle(hSemaphore);
}
int main(int argc, char* argv[]) {
    SetConsoleOutputCP(1251);
    SetConsoleCP(1251);
    // Child process handling
    if (argc == 2) {
        if (strcmp(argv[1], "childA") == 0) {
            RunThread("Process A");
            return 0;
        }
        if (strcmp(argv[1], "childB") == 0) {
            RunThread("Process B");
            return 0;
        }
```

```
    }
    // Parent process
    SECURITY_ATTRIBUTES sa;
    sa.nLength = sizeof(SECURITY_ATTRIBUTES);
    sa.lpSecurityDescriptor = NULL;
    sa.bInheritHandle = TRUE;
    // Create binary semaphore
    HANDLE hSemaphore = CreateSemaphoreW(&sa, 1, 1, SEMAPHORE_NAME);
    if (hSemaphore == NULL) {
        fprintf(stderr, "ERROR: Failed to create semaphore\n");
        return 1;
    }
    STARTUPINFOW si;
    ZeroMemory(&si, sizeof(si));
    si.cb = sizeof(si);
    PROCESS_INFORMATION piA, piB;
    wchar_t cmdA[MAX_PATH + 20];
    wchar_t cmdB[MAX_PATH + 20];
    wchar_t exePath[MAX_PATH];
    // Get path to the current executable
    GetModuleFileNameW(NULL, exePath, MAX_PATH);
    // Prepare command lines for child processes
    swprintf(cmdA, MAX_PATH + 20, L"\"%s\" childA", exePath);
    swprintf(cmdB, MAX_PATH + 20, L"\"%s\" childB", exePath);
    // Start child process A
    if (!CreateProcessW(NULL, cmdA, NULL, NULL, FALSE, CREATE_NEW_CONSOLE,
NULL, NULL, &si, &piA)) {
        fprintf(stderr, "ERROR: Failed to start Process A\n");
        CloseHandle(hSemaphore);
        return 1;
    }
    // Start child process B
    if (!CreateProcessW(NULL, cmdB, NULL, NULL, FALSE, CREATE_NEW_CONSOLE,
NULL, NULL, &si, &piB)) {
        fprintf(stderr, "ERROR: Failed to start Process B\n");
        CloseHandle(piA.hProcess);
        CloseHandle(piA.hThread);
        CloseHandle(hSemaphore);
        return 1;
    }
    // Run main thread
    RunThread("Main");
    // Wait for child processes to finish
    WaitForSingleObject(piA.hProcess, INFINITE);
    WaitForSingleObject(piB.hProcess, INFINITE);
    // Cleanup handles
    CloseHandle(piA.hProcess);
    CloseHandle(piA.hThread);
    CloseHandle(piB.hProcess);
    CloseHandle(piB.hThread);
    CloseHandle(hSemaphore);
    printf("All processes have finished successfully.\n");
    // Pause to view results
    printf("Press Enter to exit parent console...");
```

```
    getchar();
    return 0;
}
```

Приложение Г – Листинг Г.1

```
#include <windows.h>
#include <iostream>
#include <vector>
#include <thread>
#include <chrono>
#include <random>
#include <string>
#include <ctime>
using namespace std;
using namespace chrono;
CRITICAL_SECTION cs;
CONDITION_VARIABLE notFull;       // условная переменная: буфер не
полон
CONDITION_VARIABLE notEmpty;      // условная переменная: буфер не
пуст
vector<int> buffer; // кольцевой буфер
int maxBufferSize = 0;
int producedTotal = 0;            // всего произведено
int consumedTotal = 0;            // всего потреблено
int producedLast = 0;             // произведено за последние 5 сек
int consumedLast = 0;             // потреблено за последние 5 сек
int bufferSize = 0;
double producerTimeTotal = 0;     // общее время работы
производителя
double consumerTimeTotal = 0;     // общее время работы потребителя
steady_clock::time_point startTime;
// диапазоны задержек
int prodDelayMin = 0, prodDelayMax = 0;
int consDelayMin = 0, consDelayMax = 0;
bool running = true;
// Генерация случайной задержки в заданном диапазоне
int getRandomDelay(int minMs, int maxMs) {
    static random_device rd;
    static mt19937 gen(rd());
    uniform_int_distribution<> dis(minMs, maxMs);
    return dis(gen);
}
// Поток производителя
void producer() {
    while (running) {
        int delay = getRandomDelay(prodDelayMin, prodDelayMax);
        this_thread::sleep_for(milliseconds(delay));
        auto t1 = steady_clock::now();
        EnterCriticalSection(&cs);
        while (bufferSize == maxBufferSize) {
            cout << "[Producer] Буфер полон. Ожидание...\n";
            SleepConditionVariableCS(&notFull, &cs, INFINITE);
            cout << "[Producer] Пробуждение.\n";
```

```
        }
        buffer.push_back(++producedTotal);
        bufferSize++;
        producedLast++;
        cout << "[Producer] Добавлен элемент. Размер буфера: " <<
bufferSize << "\n";
        WakeConditionVariable(&notEmpty);   // будим потребителя
        LeaveCriticalSection(&cs);          // выход из критической
зоны
        auto t2 = steady_clock::now();
        producerTimeTotal += duration<double>(t2 - t1).count();
    }
}
// Поток потребителя
void consumer() {
    while (running) {
        int delay = getRandomDelay(consDelayMin, consDelayMax);
        this_thread::sleep_for(milliseconds(delay));
        auto t1 = steady_clock::now();
        EnterCriticalSection(&cs);
        while (bufferSize == 0) {
            cout << "[Consumer] Буфер пуст. Ожидание...\n";
            SleepConditionVariableCS(&notEmpty, &cs, INFINITE);
            cout << "[Consumer] Пробуждение.\n";
        }
        int item = buffer.back();
        buffer.pop_back();
        bufferSize--;
        consumedTotal++;
        consumedLast++;
        cout << "[Consumer] Извлечён элемент. Размер буфера: " <<
bufferSize << "\n";
        WakeConditionVariable(&notFull);
        LeaveCriticalSection(&cs);
        auto t2 = steady_clock::now();
        consumerTimeTotal += duration<double>(t2 - t1).count();
    }
}
void statistics() {
    while (running) {
        this_thread::sleep_for(seconds(5));
        auto now = steady_clock::now();
        double elapsed = duration<double>(now - startTime).count();
        EnterCriticalSection(&cs);
        // вычисление метрик
        double prodSpeed = producedLast / 5.0;
        double consSpeed = consumedLast / 5.0;
        double prodAvg = producedLast ? producerTimeTotal /
producedLast : 0;
```

```
        double consAvg = consumedLast ? consumerTimeTotal /
consumedLast : 0;
        double fillPercent = (double)bufferSize / maxBufferSize *
100.0;
        cout << "\n===== Статистика =====\n";
        cout << "Время работы: " << elapsed << " сек\n";
        cout << "Произведено всего: " << producedTotal << "\n";
        cout << "Потреблено всего: " << consumedTotal << "\n";
        cout << "Текущий размер буфера: " << bufferSize << "\n";
        cout << "Заполнение буфера: " << fillPercent << "%\n";
        cout << "Произведено за период: " << producedLast << "\n";
        cout << "Среднее время производства: " << prodAvg << "
сек\n";
        cout << "Скорость производства: " << prodSpeed << "
эл/сек\n";
        cout << "Общее время работы производителя: " <<
producerTimeTotal << " сек\n";
        cout << "Потреблено за период: " << consumedLast << "\n";
        cout << "Среднее время потребления: " << consAvg << "
сек\n";
        cout << "Скорость потребления: " << consSpeed << "
эл/сек\n";
        cout << "Общее время работы потребителя: " <<
consumerTimeTotal << " сек\n";
        cout << "======================\n\n";
        // сброс счетчиков
        producedLast = 0;
        consumedLast = 0;
        LeaveCriticalSection(&cs);
    }
}
int main(int argc, char* argv[]) {
    SetConsoleOutputCP(CP_UTF8);
    SetConsoleCP(CP_UTF8);
    setlocale(LC_ALL, "Russian");
    if (argc != 6) {
        cout << "Использование: Lab-06-conds [размер буфера]
[minProd] [maxProd] [minCons] [maxCons]\n";
        return 1;
    }
    maxBufferSize = atoi(argv[1]);
    prodDelayMin = atoi(argv[2]);
    prodDelayMax = atoi(argv[3]);
    consDelayMin = atoi(argv[4]);
    consDelayMax = atoi(argv[5]);
    char username[256];
    DWORD size = 256;
    GetUserNameA(username, &size);
    cout << "Пользователь: " << username << "\n";
```

```
    cout << "Буфер инициализирован. Размер: 0 / " << maxBufferSize
<< "\n";
    // инициализация синхронизации
    InitializeCriticalSection(&cs);
    InitializeConditionVariable(&notFull);
    InitializeConditionVariable(&notEmpty);
    startTime = steady_clock::now();
    thread t1(producer);
    thread t2(consumer);
    thread t3(statistics);
    cout << "Нажмите Enter для завершения...\n";
    cin.get();
    running = false;
    // пробуждение всех потоков
    WakeAllConditionVariable(&notFull);
    WakeAllConditionVariable(&notEmpty);
    t1.join();
    t2.join();
    t3.join();
    DeleteCriticalSection(&cs);
    return 0;
}
```

Приложение Д – Листинг Д.1

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <pthread.h>
#include <time.h>
#include <locale.h>

#define USERNAME "Daunil"
#define ITERATIONS 90
#define DELAY_MS 100

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

struct ThreadArgs {
    char name[20];
    pthread_t waitFor1;
    pthread_t waitFor2;
    int isMain;
};

void delay_ms(int milliseconds) {
    struct timespec ts;
    ts.tv_sec = milliseconds / 1000;
    ts.tv_nsec = (milliseconds % 1000) * 1000000L;
    nanosleep(&ts, NULL);
}

void* ThreadFunc(void* arg) {
    struct ThreadArgs* args = (struct ThreadArgs*)arg;
    pid_t pid = getpid();
    pthread_t tid = pthread_self();
    size_t nameLen = strlen(USERNAME);

    printf("[%s] Thread started. TID: %lu\n", args->name, (unsigned long)tid);

    int inCritical = 0;

    for (int i = 0; i < ITERATIONS; ++i) {
        if (i + 1 == 30) {
            pthread_mutex_lock(&mutex);
            printf("[%s] >>> Entering critical section\n", args->name);
            inCritical = 1;
        }

        char letter = USERNAME[i % nameLen];
        printf("[%s] PID: %d - TID: %lu - Iteration %d - Character: %c\n",
               args->name, pid, (unsigned long)tid, i + 1, letter);

        delay_ms(DELAY_MS);

        if (i + 1 == 61 && inCritical) {
            printf("[%s] <<< Leaving critical section\n", args->name);
```

```
            pthread_mutex_unlock(&mutex);
            inCritical = 0;
        }
    }

    if (args->isMain) {
        printf("[Main] TID: %lu finished iterations, waiting for child
threads...\n",
               (unsigned long)tid);

        pthread_join(args->waitFor1, NULL);
        pthread_join(args->waitFor2, NULL);
    }

    printf("[%s] Thread finished.\n", args->name);

    return NULL;
}

int main() {
    // Set locale for proper output (optional for English, needed for UTF-8)
    setlocale(LC_ALL, "en_US.UTF-8");

    struct ThreadArgs argsA, argsB;
    pthread_t threadA, threadB;

    // Initialize arguments for thread A
    strncpy(argsA.name, "Thread A", sizeof(argsA.name));
    argsA.waitFor1 = 0;
    argsA.waitFor2 = 0;
    argsA.isMain = 0;

    // Initialize arguments for thread B
    strncpy(argsB.name, "Thread B", sizeof(argsB.name));
    argsB.waitFor1 = 0;
    argsB.waitFor2 = 0;
    argsB.isMain = 0;

    // Create threads A and B
    if (pthread_create(&threadA, NULL, ThreadFunc, &argsA) != 0) {
        fprintf(stderr, "Error creating thread A\n");
        return 1;
    }

    if (pthread_create(&threadB, NULL, ThreadFunc, &argsB) != 0) {
        fprintf(stderr, "Error creating thread B\n");
        return 1;
    }

    // Initialize arguments for main thread
    struct ThreadArgs argsMain;
    strncpy(argsMain.name, "Main", sizeof(argsMain.name));
    argsMain.waitFor1 = threadA;
    argsMain.waitFor2 = threadB;
```

```
    argsMain.isMain = 1;

    pthread_t mainThread;

    // Create main thread
    if (pthread_create(&mainThread, NULL, ThreadFunc, &argsMain) != 0) {
        fprintf(stderr, "Error creating main thread\n");
        return 1;
    }

    // Wait for main thread to finish
    pthread_join(mainThread, NULL);

    // Destroy mutex
    pthread_mutex_destroy(&mutex);

    printf("All threads have finished. Program completed.\n");

    // Short pause to view output
    sleep(2);

    return 0;
}
```

Приложение Е – Листинг Е.1

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <pthread.h>
#include <semaphore.h>
#include <sys/wait.h>
#include <sys/types.h>
#include <fcntl.h>
#include <errno.h>
#include <time.h>
#include <locale.h>

#define USERNAME "User-c121f4dd"
#define ITERATIONS 90
#define DELAY_MS 100
#define SEMAPHORE_NAME "/lab06b_semaphore"

void delay_ms(int milliseconds) {
    struct timespec ts;
    ts.tv_sec = milliseconds / 1000;
    ts.tv_nsec = (milliseconds % 1000) * 1000000L;
    nanosleep(&ts, NULL);
}

void RunProcess(const char* name) {
    pid_t pid = getpid();
    pthread_t tid = pthread_self();
    size_t nameLen = strlen(USERNAME);

    // Open existing semaphore
    sem_t* sem = SEM_FAILED;
    while ((sem = sem_open(SEMAPHORE_NAME, 0)) == SEM_FAILED) {
        delay_ms(10);
    }

    int inCritical = 0;

    for (int i = 0; i < ITERATIONS; ++i) {
        if (i + 1 == 30) {
            sem_wait(sem); // Acquire semaphore
            printf("[%s] ENTERED critical section\n", name);
            inCritical = 1;
        }

        char letter = USERNAME[i % nameLen];
        printf("[%s] PID: %d - TID: %lu - Iteration: %d - Char: %c\n",
               name, pid, (unsigned long)tid, i + 1, letter);

        delay_ms(DELAY_MS);

        if (i + 1 == 61 && inCritical) {
            printf("[%s] EXITED critical section\n", name);
            sem_post(sem); // Release semaphore
            inCritical = 0;
        }
    }

    sem_close(sem);
}
```

```
int main(int argc, char* argv[]) {
    // Set locale for proper UTF-8 output
    setlocale(LC_ALL, "en_US.UTF-8");

    // Handle child process arguments
    if (argc == 2) {
        if (strcmp(argv[1], "childA") == 0) {
            RunProcess("Process A");
            return 0;
        } else if (strcmp(argv[1], "childB") == 0) {
            RunProcess("Process B");
            return 0;
        }
    }

    // Parent process
    sem_unlink(SEMAPHORE_NAME); // Remove old semaphore if exists

    sem_t* sem = sem_open(SEMAPHORE_NAME, O_CREAT | O_EXCL, 0666, 1);
    if (sem == SEM_FAILED) {
        fprintf(stderr, "Failed to create semaphore: %s\n", strerror(errno));
        return 1;
    }

    // Launch child process A
    pid_t pidA = fork();
    if (pidA == 0) {
        char* args[] = { argv[0], (char*)"childA", NULL };
        execvp(argv[0], args);
        perror("Failed to start process A");
        exit(1);
    } else if (pidA < 0) {
        perror("Failed to fork process A");
        sem_close(sem);
        sem_unlink(SEMAPHORE_NAME);
        return 1;
    }

    // Launch child process B
    pid_t pidB = fork();
    if (pidB == 0) {
        char* args[] = { argv[0], (char*)"childB", NULL };
        execvp(argv[0], args);
        perror("Failed to start process B");
        exit(1);
    } else if (pidB < 0) {
        perror("Failed to fork process B");
        kill(pidA, SIGTERM);
        sem_close(sem);
        sem_unlink(SEMAPHORE_NAME);
        return 1;
    }

    // Run main process
    RunProcess("Main");

    // Wait for child processes to finish
    waitpid(pidA, NULL, 0);
    waitpid(pidB, NULL, 0);

    // Cleanup
    sem_close(sem);
    sem_unlink(SEMAPHORE_NAME);
```

```
    printf("All processes finished execution.\n");

    // Short delay to view output
    sleep(2);

    return 0;
}
```