Учреждение образования

«БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ»

Операционные системы

Студент: Лопатнюк П.В.

ФИТ 3 курс 1 группа

Преподаватель: Бернацкий П.В.

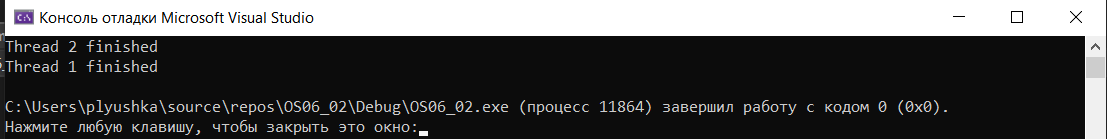
Минск 2024

**Лабораторная работа 06. Синхронизация**

**Задание 01. Windows**

1. Напишите ассемблерный код с применением команд BTS или BTR, демонстрирующий реализацию механизма синхронизации двух потоков одного процесса и поясните его работу.

|  |
| --- |
| #include <iostream>  #include <thread>  #include <atomic>  #include <chrono>  // Флаг синхронизации  std::atomic<int> sync\_flag(0);  void printMessage(const char\* message) {  printf("%s\n", message);  }  // Первый поток  void Thread1() {  while (true) {  bool bitSet;  \_\_asm {  mov eax, sync\_flag // Загрузить sync\_flag  bts eax, 0 // Установить бит 0  setc al // Установить AL, если бит уже был установлен  mov bitSet, al // Сохранить результат  }  if (!bitSet) break; // Если бит не был установлен, выходим из цикла  std::this\_thread::sleep\_for(std::chrono::milliseconds(100));  }  std::this\_thread::sleep\_for(std::chrono::milliseconds(5000)); // Работа потока  \_\_asm {  mov eax, sync\_flag // Загрузить sync\_flag  btr eax, 0 // Очистить бит 0  mov sync\_flag, eax // Сохранить результат  }  printMessage("Thread 1 finished");  }  // Второй поток  void Thread2() {  std::this\_thread::sleep\_for(std::chrono::milliseconds(1000)); // Работа потока  while (true) {  bool bitSet;  \_\_asm {  mov eax, sync\_flag  bts eax, 0  setc al  mov bitSet, al  }  if (!bitSet) break;  std::this\_thread::sleep\_for(std::chrono::milliseconds(100));  }  std::this\_thread::sleep\_for(std::chrono::milliseconds(1000)); // Работа потока  \_\_asm {  mov eax, sync\_flag // Загрузить sync\_flag  btr eax, 0 // Очистить бит 0  mov sync\_flag, eax // Сохранить результат  }  printMessage("Thread 2 finished");  }  int main() {  std::thread t1(Thread1);  std::thread t2(Thread2);  t1.join();  t2.join();  return 0;  } |



Поток 1 устанавливает флаг синхронизации и ожидает, пока другой поток не завершит свою работу. Поток 2 также ожидает флаг, выполняет свою работу, а затем сигнализирует потоку 1, что можно завершать выполнение.

Основная идея заключается в том, чтобы через флаг передавать информацию о текущем состоянии потока и синхронизировать их выполнение, избегая конфликтов и гонок.

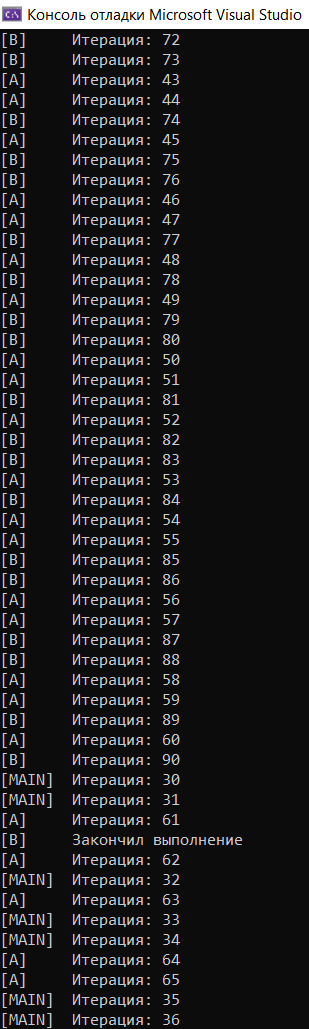
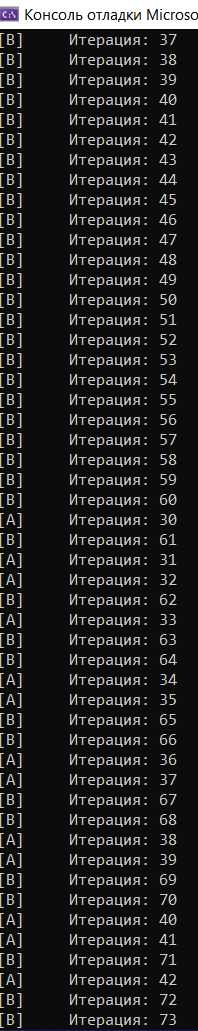
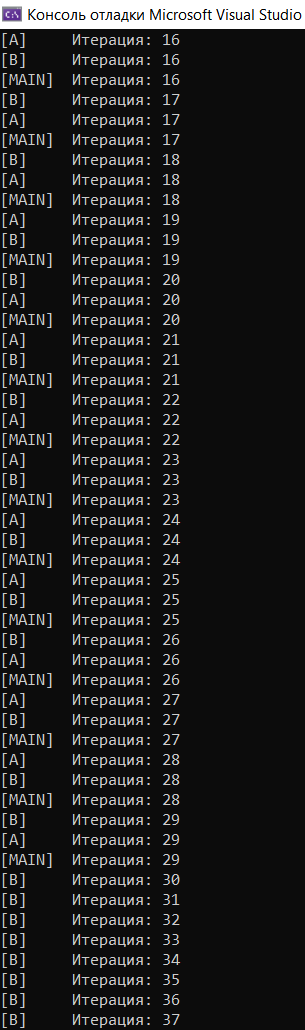
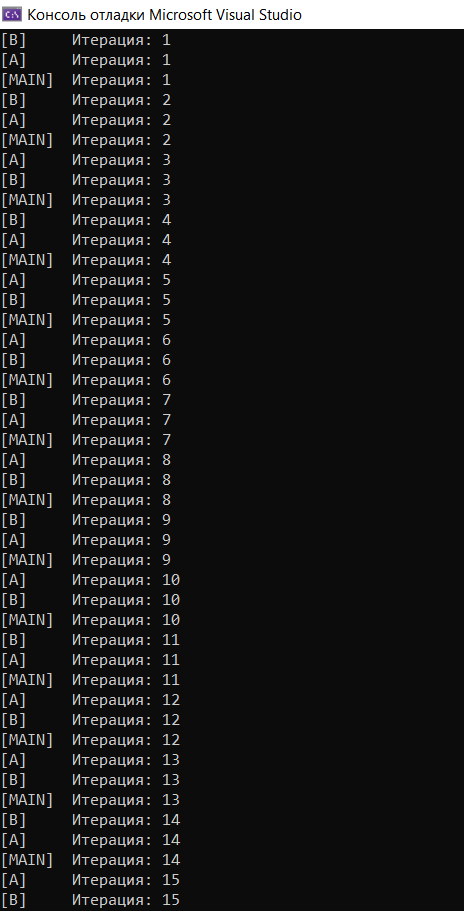
Потоки используют атомарную переменную sync\_flag, чтобы синхронизировать свои действия. Бит 0 синхронизирует выполнение потоков: первый поток устанавливает этот бит, чтобы второй поток знал, что он может завершиться.

Инструкции BTS (битовая установка) и BTR (битовая очистка) обеспечивают работу с этим флагом. Они выполняются на уровне процессора, что позволяет синхронизировать потоки без использования мьютексов или других сложных механизмов синхронизации.

**Задание 02. Windows**

1. Разработайте приложение **OS06\_02**, запускающее два дочерних потока **A** и **B**.
2. Все потоки выполняют циклы в 90 итераций, выводящие имена потоков и номера итерации с задержкой в 0.1 сек.
3. Приложение **OS06\_02** синхронизирует выполнение потоков **main**, **A** и **B** с помощью механизма **critical section.**
4. Синхронизация должна обеспечивать поочередное выполнение итераций цикла с 30 по 60 в каждом потоке.

|  |
| --- |
| #include <iostream>  #include <windows.h>  using namespace std;  CRITICAL\_SECTION critical\_section;  DWORD WINAPI loop(LPVOID param) {  char\* thread\_name = static\_cast<char\*>(param);  for (int i = 1; i <= 90; ++i) {  // Синхронизация в диапазоне итераций [30; 60]  if (i == 30) EnterCriticalSection(&critical\_section);  printf("[%s]\tИтерация: %d\n", thread\_name, i);  if (i == 60) LeaveCriticalSection(&critical\_section);  Sleep(100);  }  printf("[%s]\tЗакончил выполнение\n", thread\_name);  return 0;  }  int main() {  setlocale(LC\_ALL, "rus");  const int num\_threads = 2;  HANDLE threads[num\_threads];  // Инициализация критической секции  InitializeCriticalSection(&critical\_section);  threads[0] = CreateThread(NULL, 0, loop, (LPVOID)"A", 0, NULL);  if (threads[0] == NULL) {  cerr << "Ошибка при создании потока A\n";  return 1;  }  threads[1] = CreateThread(NULL, 0, loop, (LPVOID)"B", 0, NULL);  if (threads[1] == NULL) {  cerr << "Ошибка при создании потока B\n";  return 1;  }  // Основной поток выполняет свои итерации  for (int i = 1; i <= 90; ++i) {  if (i == 30) EnterCriticalSection(&critical\_section);  printf("[MAIN]\tИтерация: %d\n", i);  if (i == 60) LeaveCriticalSection(&critical\_section);  // Задержка между итерациями  Sleep(100);  }  printf("[MAIN]\tЗакончил выполнение\n");  WaitForMultipleObjects(num\_threads, threads, TRUE, INFINITE);  for (int i = 0; i < num\_threads; ++i) {  CloseHandle(threads[i]);  }  DeleteCriticalSection(&critical\_section);  return 0;  } |





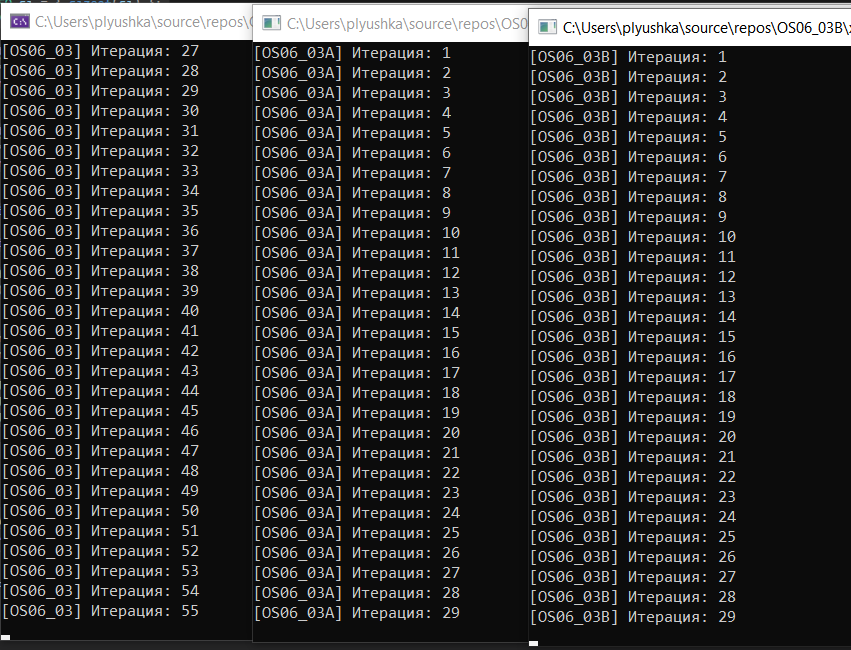
**Задание 03. Windows**

1. Разработайте приложение **OS06\_03**, запускающее два дочерних процесса **OS06\_03A** и **OS06\_03B** свыводом в отдельные консоли.
2. Все процессы выполняют циклы в 90 итераций, выводящие имена процессов и номера итерации с задержкой в 0.1 сек.
3. Приложение **OS06\_03** синхронизирует выполнение процессов **OS06\_03**, **OS06\_03A** и **OS06\_03B** с помощью механизма **mutex.**
4. Синхронизация должна обеспечивать поочередное выполнение итераций цикла с 30 по 60.

|  |
| --- |
| #include <iostream>  #include <Windows.h>  using namespace std;  int main() {  setlocale(LC\_ALL, "rus");  HANDLE mutex = CreateMutex(NULL, FALSE, L"OS06\_03");  if (mutex == NULL) {  cerr << "Ошибка при создании мьютекса\n";  return 1;  }  STARTUPINFO si = { sizeof(si) };  PROCESS\_INFORMATION piA, piB;  if (!CreateProcess(L"C:\\Users\\plyushka\\source\\repos\\OS06\_03A\\x64\\Debug\\OS06\_03A.exe", NULL, NULL, NULL, FALSE, CREATE\_NEW\_CONSOLE, NULL, NULL, &si, &piA)) {  cerr << "Ошибка при создании процесса A\n";  return 1;  }  if (!CreateProcess(L"C:\\Users\\plyushka\\source\\repos\\OS06\_03B\\x64\\Debug\\OS06\_03B.exe", NULL, NULL, NULL, FALSE, CREATE\_NEW\_CONSOLE, NULL, NULL, &si, &piB)) {  cerr << "Ошибка при создании процесса B\n";  return 1;  }  for (int i = 1; i <= 90; ++i) {  if (i == 30) WaitForSingleObject(mutex, INFINITE);  else if (i == 60) ReleaseMutex(mutex);  printf("[OS06\_03] Итерация: %d\n", i);  Sleep(100);  }  WaitForSingleObject(piA.hProcess, INFINITE);  WaitForSingleObject(piB.hProcess, INFINITE);  CloseHandle(piA.hProcess);  CloseHandle(piB.hProcess);  CloseHandle(mutex);  return 0;  } |

**OS03\_03A-OS03\_03B**

|  |
| --- |
| #include <iostream>  #include <Windows.h>  using namespace std;  int main() {  setlocale(LC\_ALL, "rus");  HANDLE mutex = OpenMutex(SYNCHRONIZE, FALSE, L"OS06\_03");  if (mutex == NULL) {  cerr << "Ошибка при открытии мьютекса\n";  return 1;  }  for (int i = 1; i <= 90; ++i) {  if (i == 30) WaitForSingleObject(mutex, INFINITE);  else if (i == 60) ReleaseMutex(mutex);  printf("[OS06\_03A] Итерация: %d\n", i);  Sleep(100);  }  CloseHandle(mutex);  return 0;  } |



Родительский процесс создает мьютекс (CreateMutex). Мьютекс получает уникальное имя OS06\_03, чтобы быть доступным для всех дочерних процессов.

Дочерние процессы (OS06\_03A и OS06\_03B) открывают существующий мьютекс с помощью OpenMutex, используя его имя.

В родительском и дочерних процессах, начиная с итерации 30, выполняется WaitForSingleObject, чтобы заблокировать мьютекс. Это означает, что процесс или поток, который первый заблокировал мьютекс, выполняет свою критическую секцию, а другие ждут.

После итерации 60 процесс освобождает мьютекс с помощью ReleaseMutex, позволяя другим процессам продолжить выполнение.

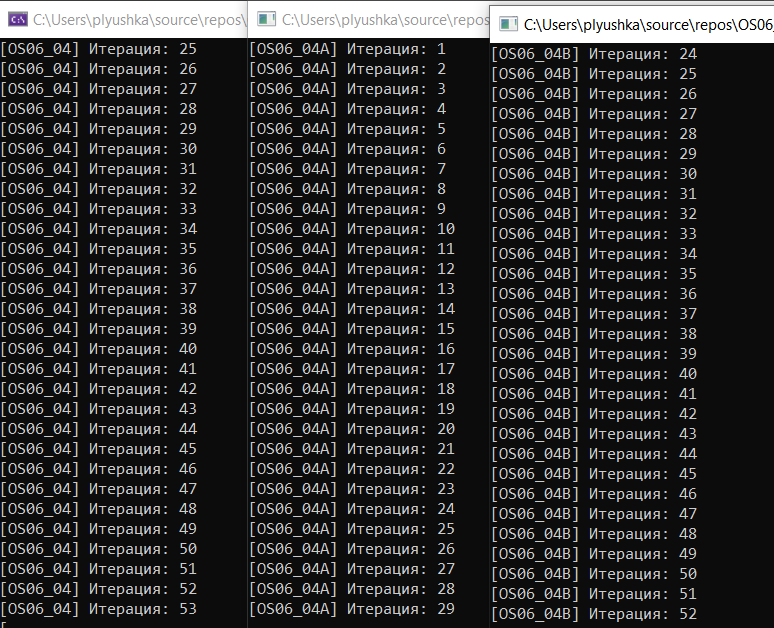
Мьютекс гарантирует, что доступ к критической секции кода всегда будет последовательным, предотвращая "гонки данных" или одновременное выполнение.

**Задание 04. Windows**

1. Разработайте приложение **OS06\_04**, запускающее два дочерних процесса **OS06\_04A** и **OS06\_04B** свыводом в отдельные консоли.
2. Все процессы выполняют циклы в 90 итераций, выводящие имена процессов и номера итерации с задержкой в 0.1 сек.
3. Приложение **OS06\_04** синхронизирует выполнение процессов **OS06\_04**, **OS06\_04A** и **OS06\_04B** с помощью механизма **semaphore.**
4. Синхронизация должна обеспечивать поочередное выполнение итераций цикла с 30 по 60 одного (любого) процесса и двух других процессов. Другими словами, итерации с 30 по 60 должны одновременно выполняться только в двух из трех процессов.

|  |
| --- |
| #include <iostream>  #include <Windows.h>  using namespace std;  int main() {  setlocale(LC\_ALL, "RUS");  const int size = 2;  HANDLE semaphore;  HANDLE processes[size] = { NULL, NULL };  STARTUPINFO startupInfoA, startupInfoB;  PROCESS\_INFORMATION processInfoA, processInfoB;  ZeroMemory(&startupInfoA, sizeof(STARTUPINFO));  startupInfoA.cb = sizeof(STARTUPINFO);  ZeroMemory(&startupInfoB, sizeof(STARTUPINFO));  startupInfoB.cb = sizeof(STARTUPINFO);  if (CreateProcessW(L"C:\\Users\\plyushka\\source\\repos\\OS06\_04A\\x64\\Debug\\OS06\_04A.exe", NULL, NULL, NULL, FALSE, CREATE\_NEW\_CONSOLE, NULL, NULL, &startupInfoA, &processInfoA)) {  cout << "Процесс A создан\n";  processes[0] = processInfoA.hProcess;  }  else {  cout << "Не удалось создать процесс A\n";  }  if (CreateProcessW(L"C:\\Users\\plyushka\\source\\repos\\OS06\_04B\\x64\\Debug\\OS06\_04B.exe", NULL, NULL, NULL, FALSE, CREATE\_NEW\_CONSOLE, NULL, NULL, &startupInfoB, &processInfoB)) {  cout << "Процесс B создан\n";  processes[1] = processInfoB.hProcess;  }  else {  cout << "Не удалось создать процесс B\n";  }  semaphore = CreateSemaphore(NULL, 2, 2, L"OS06\_04");  for (int i = 1; i <= 90; ++i) {  if (i == 30) {  WaitForSingleObject(semaphore, INFINITE); // Ожидание семафора  }  if (i == 60) {  ReleaseSemaphore(semaphore, 1, NULL); // Освобождение семафора  }  printf("[OS06\_04] Итерация: %d\n", i);  Sleep(100);  }  WaitForMultipleObjects(size, processes, TRUE, INFINITE);  CloseHandle(processInfoA.hProcess);  CloseHandle(processInfoA.hThread);  CloseHandle(processInfoB.hProcess);  CloseHandle(processInfoB.hThread);  if (semaphore) {  CloseHandle(semaphore);  }  cout << '\n';  system("pause");  return 0;  } |

|  |
| --- |
| #include <iostream>  #include <Windows.h>  int main() {  HANDLE semaphore = OpenSemaphore(SEMAPHORE\_ALL\_ACCESS, FALSE, L"OS06\_04");  for (int i = 1; i <= 90; ++i) {  if (i == 30) {  WaitForSingleObject(semaphore, INFINITE); // Ожидание семафора  }  if (i == 60) {  ReleaseSemaphore(semaphore, 1, NULL); // Освобождение семафора  }  printf("[OS06\_04A] Итерация: %d\n", i);  Sleep(100);  }  if (semaphore) {  CloseHandle(semaphore);  }  return 0;  } |

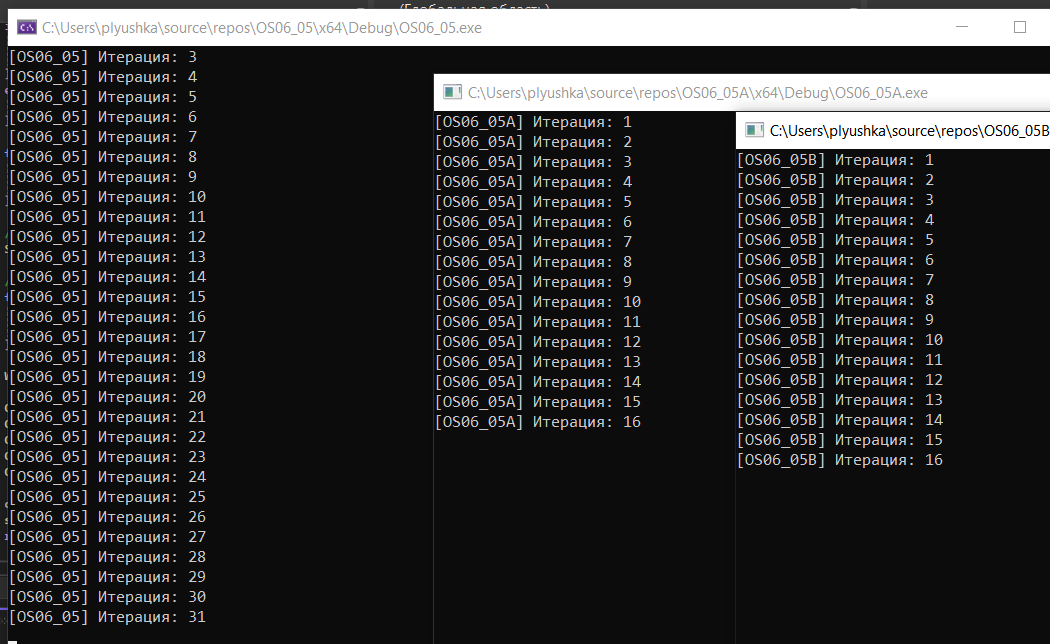


**Задание 05. Windows**

1. Разработайте приложение **OS06\_05**, запускающее два дочерних процесса **OS06\_05A** и **OS06\_05B** свыводом в отдельные консоли.
2. Все процессы выполняют циклы в 90 итераций, выводящие имена процессов и номера итерации с задержкой в 0.1 сек.
3. Приложение **OS06\_05** синхронизирует выполнение процессов **OS06\_05**, **OS06\_05A** и **OS06\_05B** с помощью механизма **event**.
4. Синхронизация должна обеспечивать выполнение приложения в два этапа:
5. выполнение итераций с 1 по 15 процесса **OS06\_05**;
6. одновременное выполнение всех трех процессов: **OS06\_05** – продолжает выполнение итераций; процессы **OS06\_05A** и **OS06\_05B** выполняются начиная с первой итерации.

|  |
| --- |
| #include <iostream>  #include <Windows.h>  using namespace std;  int main() {  setlocale(LC\_ALL, "RUS");  const int size = 2;  HANDLE event; // Событие для синхронизации  HANDLE processes[size] = { NULL, NULL };  STARTUPINFO startupInfoA, startupInfoB;  PROCESS\_INFORMATION processInfoA, processInfoB;  ZeroMemory(&startupInfoA, sizeof(STARTUPINFO));  startupInfoA.cb = sizeof(STARTUPINFO);  ZeroMemory(&startupInfoB, sizeof(STARTUPINFO));  startupInfoB.cb = sizeof(STARTUPINFO);  // Создание события  event = CreateEvent(NULL, TRUE, FALSE, L"OS06\_05\_Event");  if (CreateProcessW(L"C:\\Users\\plyushka\\source\\repos\\OS06\_05A\\x64\\Debug\\OS06\_05A.exe", NULL, NULL, NULL, FALSE, CREATE\_NEW\_CONSOLE, NULL, NULL, &startupInfoA, &processInfoA)) {  cout << "Процесс A создан\n";  processes[0] = processInfoA.hProcess;  }  else {  cout << "Не удалось создать процесс A\n";  }  if (CreateProcessW(L"C:\\Users\\plyushka\\source\\repos\\OS06\_05B\\x64\\Debug\\OS06\_05B.exe", NULL, NULL, NULL, FALSE, CREATE\_NEW\_CONSOLE, NULL, NULL, &startupInfoB, &processInfoB)) {  cout << "Процесс B создан\n";  processes[1] = processInfoB.hProcess;  }  else {  cout << "Не удалось создать процесс B\n";  }  for (int i = 1; i <= 15; ++i) {  printf("[OS06\_05] Итерация: %d\n", i);  Sleep(100);  }  // Сигнализация дочерним процессам о продолжении работы  SetEvent(event);  // Основной процесс продолжает выполнение итераций  for (int i = 16; i <= 90; ++i) {  printf("[OS06\_05] Итерация: %d\n", i);  Sleep(100);  }  WaitForMultipleObjects(size, processes, TRUE, INFINITE);  CloseHandle(processInfoA.hProcess);  CloseHandle(processInfoA.hThread);  CloseHandle(processInfoB.hProcess);  CloseHandle(processInfoB.hThread);  CloseHandle(event);  cout << '\n';  system("pause");  return 0;  } |

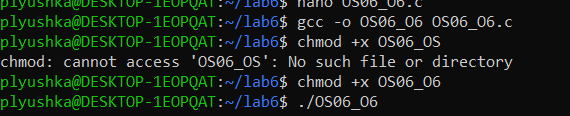
|  |
| --- |
| #include <iostream>  #include <Windows.h>  int main() {  setlocale(LC\_ALL, "RUS");  // Открытие события для синхронизации  HANDLE event = OpenEvent(EVENT\_ALL\_ACCESS, FALSE, L"OS06\_05\_Event");  // Ожидание сигнала от основного процесса  WaitForSingleObject(event, INFINITE);  for (int i = 1; i <= 90; ++i) {  printf("[OS06\_05A] Итерация: %d\n", i);  Sleep(100);  }  CloseHandle(event);  return 0;  } |

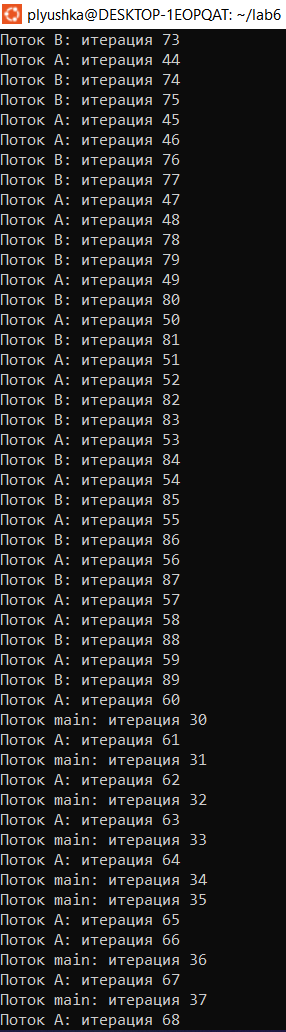
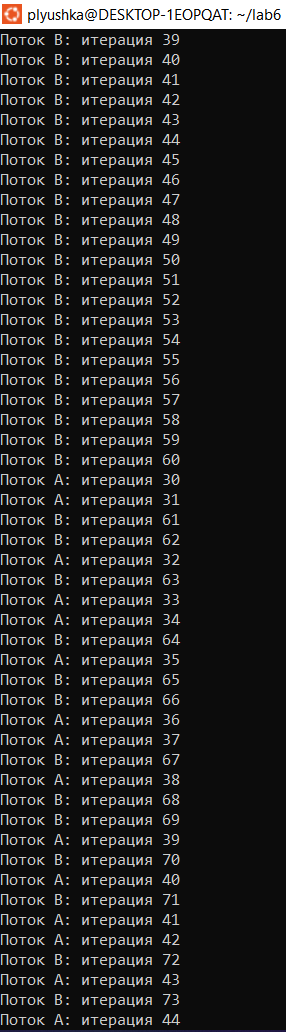
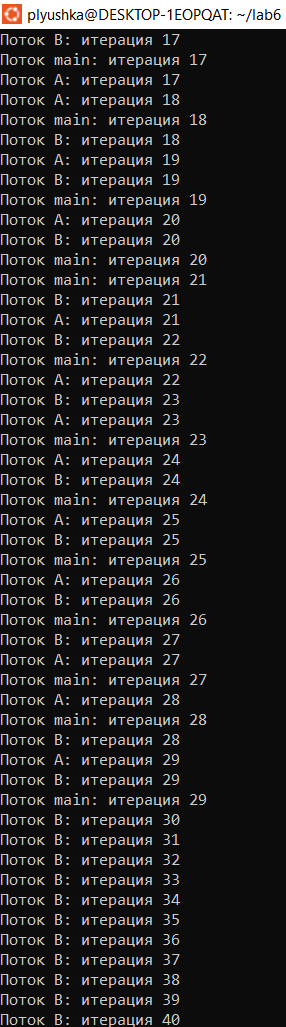
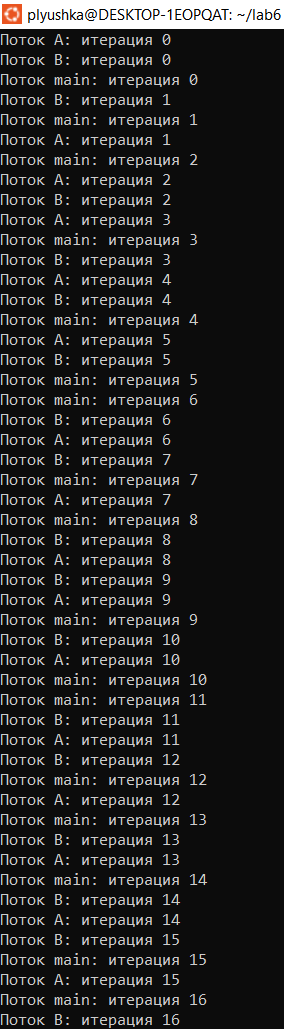


**Задание 06. Linux**

1. Разработайте приложение **OS06\_06**, запускающее два дочерних потока **A** и **B**.
2. Все потоки выполняют циклы в 90 итераций, выводящие имена потоков и номера итерации с задержкой в 0.1 сек.
3. Приложение **OS06\_06** синхронизирует выполнение потоков **main**, **A** и **B** с помощью механизма **mutex.**
4. Синхронизация должна обеспечивать поочередное выполнение итераций цикла с 30 по 60 в каждом потоке.

|  |
| --- |
| #include <pthread.h>  #include <stdio.h>  #include <unistd.h>  pthread\_mutex\_t mutex;  void\* thread\_function(void\* arg) {  char\* thread\_name = (char\*)arg;  for (int i = 0; i < 90; i++) {  if (i == 30) pthread\_mutex\_lock(&mutex); // Захватываем мьютекс  if (i == 60) pthread\_mutex\_unlock(&mutex); // Освобождаем мьютекс  printf("Поток %s: итерация %d\n", thread\_name, i);  usleep(100000); // Задержка в 0.1 сек  }  return NULL;  }  int main() {  pthread\_t thread\_a, thread\_b;  pthread\_mutex\_init(&mutex, NULL);  pthread\_create(&thread\_a, NULL, thread\_function, "A");  pthread\_create(&thread\_b, NULL, thread\_function, "B");  thread\_function("main");  pthread\_join(thread\_a, NULL);  pthread\_join(thread\_b, NULL);  pthread\_mutex\_destroy(&mutex);  return 0;  } |





**Задание 07.** Ответьте на следующие вопросы

1. **Дайте определение понятию «синхронизация потоков».**

**Синхронизация потоков** — это процесс координации выполнения нескольких потоков в многозадачной среде для обеспечения их правильного взаимодействия и предотвращения конфликтов при доступе к общим ресурсам, таким как память, файлы или устройства.

1. **Объясните понятие «взаимная блокировка».**

**Взаимная блокировка** (или deadlock) — это состояние в многопоточной программе, когда два или более потоков (или процессов) находятся в состоянии ожидания ресурсов, заблокированных друг другом, что приводит к бесконечному ожиданию и невозможности завершить работу.

1. **Перечислите механизмы синхронизации OS.**

**Механизмы синхронизации:** критические секции, мьютексы, семафоры, событие, барьер, атомарные операции, монитор.

Критические секции: Простейший механизм взаимного исключения. Только один поток может находиться в критической секции одновременно.

Мьютексы: Более развитый механизм взаимного исключения. Поток «захватывает» мьютекс перед доступом к разделяемому ресурсу и «освобождает» его после завершения.

Семафоры: Обобщение мьютексов. Целочисленная переменная, управляющая доступом к ресурсам. Операции P (wait) и V (signal) регулируют доступ.

События: Механизм сигнализации. Один поток устанавливает событие, а другой ожидает его наступления. Подходит для асинхронной синхронизации.

Барьеры: Синхронизируют множество потоков. Все потоки останавливаются на барьере, пока все не достигнут его.

Атомарные операции: Операции, выполняющиеся неразрывно, без перерывов от других потоков. Являются основой для других механизмов синхронизации.

Мониторы: Абстракция, объединяющая данные и операции над ними, гарантируя взаимное исключение доступа к данным из нескольких потоков.

1. **Поясните в чем разница между механизмом mutex и** **semaphore**.

**Мьютекс (Mutex):**

Взаимное исключение: Его основная и единственная функция — обеспечить, чтобы только один поток владел ресурсом одновременно.

Владение: Только поток, который захватил мьютекс, может его освободить. Это важно для предотвращения deadlock'ов.

Ограничение: Обычно используется только для синхронизации доступа к одному ресурсу, поэтому в этом сценарии он и семафор работают аналогично.

Принцип работы: Внутренний счетчик мьютекса имеет только два состояния: 0 (занят) и 1 (свободен).

**Семафор:**

Управление доступом: Может управлять доступом к нескольким ресурсам одновременно. Внутренний счетчик определяет количество доступных ресурсов.

Бинарный семафор: Действительно, является аналогом мьютекса, но с одним важным отличием: его может освободить любой поток.

Подсчитывающий семафор: Значение счетчика может быть больше 1. Разрешает одновременный доступ к ресурсу нескольким потокам до тех пор, пока не будет исчерпан лимит, заданный значением счетчика. Например, если семафор имеет значение 5, то 5 потоков могут одновременно использовать ресурс.

Более гибкий: Позволяет реализовать более сложные схемы синхронизации, такие как производитель-потребитель, управление пулом соединений и т.д.

Если нужно только обеспечить взаимное исключение, лучше использовать мьютекс. Если нужно управлять доступом к ресурсу, количество которого может быть больше одного, или нужны более сложные схемы синхронизации, лучще использовать семафор. Мьютекс – это частный случай семафора, но он более безопасен и понятен в контексте взаимного исключения.

1. **Почему mutex, semaphore, event создают объект ядра OS, а critical section нет.**

Мьютексы, семафоры и события — механизмы синхронизации, способные работать как внутри одного процесса, так и между разными процессами. Для межпроцессной коммуникации необходима поддержка со стороны ядра операционной системы. Поэтому для этих механизмов создаются **объекты ядра**, которые обеспечивают атомарность операций, защиту от гонок данных, и управление состоянием объектов синхронизации в масштабе всей системы.

Критические секции, напротив, предназначены только для синхронизации внутри одного процесса. Они используют более лёгкие механизмы, не требующие создания объектов ядра, например, аппаратные инструкции или библиотечные функции, обеспечивающие взаимное исключение на уровне процесса.