Белорусский государственный технологический университет

Факультет информационных технологий

Кафедра программной инженерии

Лабораторная работа 3

По дисциплине «Операционные системы»

На тему «Управление процессами»

Выполнил:

Студент 3 курса 9 группы

Хуторцов Кирилл Владимирович

Преподаватель: Савельева М.Г.

Минск, 2025

# **Введение**

**Цель работы:** получение практических навыков в работе с процессами в OC Windows и Linux.

**Постановка задачи:**

1. Требуется разработать несколько приложений для ОС Windows.
2. Требуется разработать несколько приложений для ОС Linux.
3. Требуется реализовать создание и управление процессами, передачу параметров через аргументы командной строки и переменные окружения, межпроцессное взаимодействие через анонимные каналы, анализ процессов с помощью системных утилит, обработку ошибок и корректное завершение дочерних процессов.

**Используемые инструменты**:

* Oracle VirtualBox
* Командная оболочка cmd
* Process Explorer
* MinGW-w64 и g++

# **Постановка задачи для Windows**

## **Уровень 4-5**

**Приложение Lab-03х.**

Программа выполняет простейший цикл с заданным количеством итераций, при чём каждая итерация выполняется с задержкой 500 мс. Программа должна выводить PID на каждой итерации и перед стартом выводить количество итераций. Количество итераций передается в программу через аргументы командной строки. Если не задан аргумент командной строки, то количество итераций приложение пытается получить через переменную окружения ITER\_NUM. Если не заданы ни переменная окружения, ни аргумент командной строки приложение завершает свою работу с ошибкой (Использовать **ExitProcess**).

Создаем приложение файл с расширением cpp, затем пишем туда код, который представлен в листинге 1.1.

#include <windows.h>

#include <iostream>

#include <cstdlib>

int main(int argc, char\* argv[]) {

int iterations = 0;

if (argc > 1) {

iterations = std::atoi(argv[1]);

} else {

char\* envVar = std::getenv("ITER\_NUM");

if (envVar != nullptr) {

iterations = std::atoi(envVar);

} else {

std::cerr << "Error: no number of iterations is specified" << std::endl;

ExitProcess(1);

}

}

std::cout << "Number of iterations: " << iterations << std::endl;

DWORD pid = GetCurrentProcessId();

for (int i = 0; i < iterations; ++i) {

std::cout << "Iteration " << (i + 1) << ", PID: " << pid << std::endl;

Sleep(500);

}

return 0;

}

Листинг 1.1 – Приложение Lab-03x

После этого нам необходимо скомпилировать из cpp-файла – exe-файл. А затем запускаем наше приложение через командную строку с помощью указания параметров и с помощью переменной окружения.

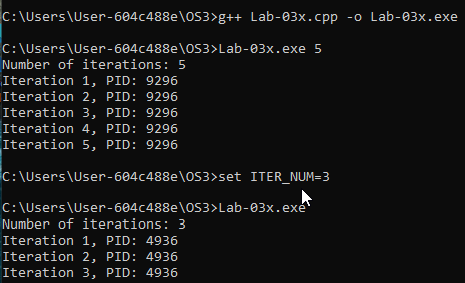


Рисунок 1.1 – Компиляция и запуск приложения Lab-03x

В результате выполнения ошибок не произошло. В случае если не задать переменную окружения и запустить приложение без переменных, то на экране появится соответствующая ошибка.

**Приложение Lab-03a:**

Программа создаёт три дочерних процесса с помощью **CreateProcess**. Все три процесса должны создаваться для программы **Lab-03x**, при чём для передачи имени исполняемого файла (в данном случае используется [полное имя файла](https://ru.wikipedia.org/wiki/%D0%98%D0%BC%D1%8F_%D1%84%D0%B0%D0%B9%D0%BB%D0%B0)) и параметра количества итераций должны быть выполнены следующие условия:

* При создании первого процесса используется только первый параметр функции **CreateProcess**. Количество итераций также требуется указать в первом параметре, так задумано (прим. «**Lab-03х.exe 50**»).
* При создании второго процесса используется только второй параметр функции **CreateProcess**. По сути взять строку из предыдущего пункта и передать во второй параметр.
* При создании третьего процесса для передачи имени программы используется – первый параметр, для количества итераций – второй. (при этом во втором параметре вам может понадобиться пробел в начале строки)

Родительский процесс должен дождаться завершения всех дочерних процессов (выполняются одновременно), перед тем как завершиться самому.

***Вопрос****: Все ли процессы созданы успешно? Если нет, почему?*

Также предусмотреть обработку ошибок и очистку ресурсов. Использовать Process Explorer, cmd и TaskManager для просмотра информации о созданных процессах. В Process Explorer проверить наличие дескрипторов дочерних процессов у родительского процесса.

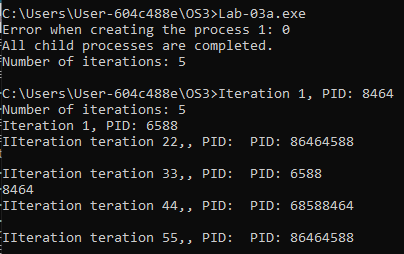
Установить глобальную переменную окружения ITER\_NUM для вашего пользователя.

Создаем файл Lab-03a.cpp и записываем туда код, представленный в листинге 1.2.

|  |
| --- |
| #include <windows.h>  #include <iostream>  #include <string>  int main() {  STARTUPINFO si = { sizeof(si) };  PROCESS\_INFORMATION pi[3];  BOOL success;  std::string exePath = "C:\\Users\\User-604c488e\\OS3\\Lab-03x.exe";  std::string arg = "5";  // 1. Всё в lpApplicationName  std::string full1 = exePath + " " + arg;  success = CreateProcess(  full1.c\_str(), // ← всё в первом параметре  NULL,  NULL, NULL, FALSE, 0, NULL, NULL,  &si, &pi[0]);  if (!success) {  std::cerr << "Error when creating the process 1: " << GetLastError() << std::endl;  }  // 2. Всё в lpCommandLine  std::string full2 = exePath + " " + arg;  success = CreateProcess(  NULL,  (LPSTR)full2.c\_str(), // ← всё во втором параметре  NULL, NULL, FALSE, 0, NULL, NULL,  &si, &pi[1]);  if (!success) {  std::cerr << "Error when creating the process 2: " << GetLastError() << std::endl;  }  // 3. Имя отдельно, аргументы отдельно  std::string arg3 = " " + arg; // ← пробел обязателен!  success = CreateProcess(  exePath.c\_str(), // ← только имя  (LPSTR)arg3.c\_str(), // ← только аргументы  NULL, NULL, FALSE, 0, NULL, NULL,  &si, &pi[2]);  if (!success) {  std::cerr << "Error when creating the process 3: " << GetLastError() << std::endl;  }  // Ожидание завершения всех процессов  HANDLE handles[3] = { pi[0].hProcess, pi[1].hProcess, pi[2].hProcess };  WaitForMultipleObjects(3, handles, TRUE, INFINITE);  // Очистка ресурсов  for (int i = 0; i < 3; ++i) {  CloseHandle(pi[i].hProcess);  CloseHandle(pi[i].hThread);  }  std::cout << "All child processes are completed." << std::endl;  return 0;  } |

Листинг 1.2 – Приложение Lab-03a

После этого компилируем наш cpp-файл в exe-файл и запускаем.

  
Рисунок 1.2 – Приложение Lab-03a

В результате не все процессы были созданы успешно. Первый процесс завершился с ошибкой, потому что в функцию CreateProcess был передан путь к программе вместе с аргументом ("Lab-03x.exe 5") через первый параметр lpApplicationName. Однако этот параметр должен содержать только путь к исполняемому файлу, без аргументов. Из-за этого система попыталась найти файл с именем "Lab-03x.exe 5", которого не существует, и вернула ошибку. Второй и третий процессы были созданы правильно, потому что аргументы были переданы через второй параметр lpCommandLine, как это и требуется по правилам работы CreateProcess.

Теперь выполним проверку на запущенные процессы через cmd.

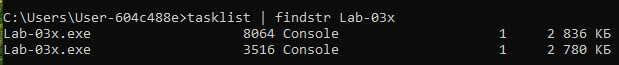


Рисунок 1.3 – Проверка процессов через cmd

Затем проверим процессы через Task Manager



Рисунок 1.4 – Проверка процессов через Task Manager

Затем проверяем наши процессы через Process Explorer

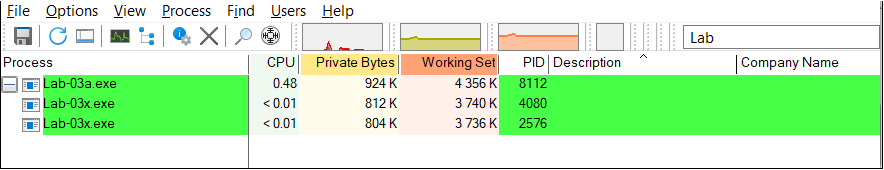


Рисунок 1.5 – Проверка процессов через Process Explorer

После всех совершенных проверок, мы убедились, что процесс был запущен 2 раза. Также видим, что корневое древо: Lab-03a является родительским элементом, а Lab-03x – дочерними.

**Приложение Lab-03b**

Создать программу идентичную программе Lab-03a, но:

* В первом вызове **CreateProcess** оставить только имя файла (т.е. убрать количество итераций, которое указывалось в Lab-03a).
* В третьем вызове передать NULL вместо второго параметра, и перед этим установить локальную переменную окружения ITER\_NUM. Значение должно отличаться от глобальной переменной.

***Вопрос:*** *Все ли процессы созданы успешно? Сколько итераций выполнил каждый процесс и почему?*

Создаем файл Lab-03b.cpp и пишем туда код, который представлен в листинге 1.3.

|  |
| --- |
| #include <windows.h>  #include <iostream>  #include <string>  int main() {  STARTUPINFO si = { sizeof(si) };  PROCESS\_INFORMATION pi[3];  BOOL success;  std::string exePath = "C:\\Users\\User-604c488e\\OS3\\Lab-03x.exe";  std::string arg = "5";  // 1. Только имя файла, без аргументов  success = CreateProcess(  exePath.c\_str(), // ← только путь  NULL, // ← аргументы не передаются  NULL, NULL, FALSE, 0, NULL, NULL,  &si, &pi[0]);  if (!success) {  std::cerr << "Error when creating the process 1: " << GetLastError() << std::endl;  }  // 2. Всё в lpCommandLine  std::string full2 = "\"" + exePath + "\" " + arg;  success = CreateProcess(  NULL,  (LPSTR)full2.c\_str(), // ← путь + аргумент  NULL, NULL, FALSE, 0, NULL, NULL,  &si, &pi[1]);  if (!success) {  std::cerr << "Error when creating the process 2: " << GetLastError() << std::endl;  }  // 3. NULL в lpCommandLine + локальная переменная окружения  SetEnvironmentVariableA("ITER\_NUM", "7"); // локальное значение  success = CreateProcess(  exePath.c\_str(), // ← только имя  NULL, // ← аргументы не передаются  NULL, NULL, FALSE, 0, NULL, NULL,  &si, &pi[2]);  if (!success) {  std::cerr << "Error when creating the process 3: " << GetLastError() << std::endl;  }  // Ожидание завершения всех процессов  HANDLE handles[3] = { pi[0].hProcess, pi[1].hProcess, pi[2].hProcess };  WaitForMultipleObjects(3, handles, TRUE, INFINITE);  // Очистка ресурсов  for (int i = 0; i < 3; ++i) {  CloseHandle(pi[i].hProcess);  CloseHandle(pi[i].hThread);  }  std::cout << "All child processes are completed." << std::endl;  return 0;  } |

Листинг 1.3 – Приложение Lab-03b

Далее компилируем из cpp-файла exe-файл и запускаем его.

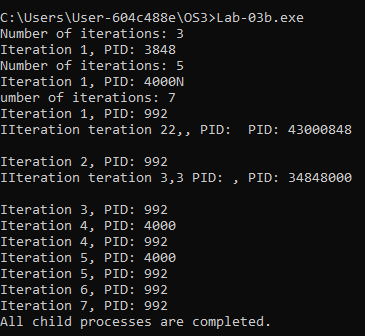


Рисунок 1.6 – Приложение Lab-03b

В результате, все процессы были запущены, однако первый выполнился 3 раза, так как глобальная переменная была установлена = 3. Второй процесс выполнился 5 раз, так как в коде был передан аргумент 5. Третий процесс выполнился 7 раз, так как локальная переменная была установлена = 7.

**Приложение Lab-03с**

Создать программу, которая выводит список запущенных в системе процессов. Для каждого процесса выводится его имя, идентификатор и родительский идентификатор. ([Tool Help Library](https://learn.microsoft.com/en-us/windows/win32/api/_toolhelp/))

Создаем файл cpp и записываем туда код, представленный в листинге 1.4.

|  |
| --- |
| #include <windows.h>  #include <tlhelp32.h>  #include <iostream>  int main() {  HANDLE hSnapshot = CreateToolhelp32Snapshot(TH32CS\_SNAPPROCESS, 0);  if (hSnapshot == INVALID\_HANDLE\_VALUE) {  std::cerr << "Failed to create a snapshot of processes. Error code: " << GetLastError() << std::endl;  return 1;  }  PROCESSENTRY32 pe32;  pe32.dwSize = sizeof(PROCESSENTRY32);  if (!Process32First(hSnapshot, &pe32)) {  std::cerr << "Failed to create a snapshot of processes. Error code: " << GetLastError() << std::endl;  CloseHandle(hSnapshot);  return 1;  }  std::cout << "List of running processes:\n";  std::cout << "---------------------------------------------\n";  do {  std::wcout << L"Process Name: " << pe32.szExeFile  << L", PID: " << pe32.th32ProcessID  << L", PPID: " << pe32.th32ParentProcessID << std::endl;  } while (Process32Next(hSnapshot, &pe32));  CloseHandle(hSnapshot);  return 0;  } |

Листинг 1.4 – Приложение Lab-03c

Затем компилируем из cpp-файла exe-файл. И запускаем.

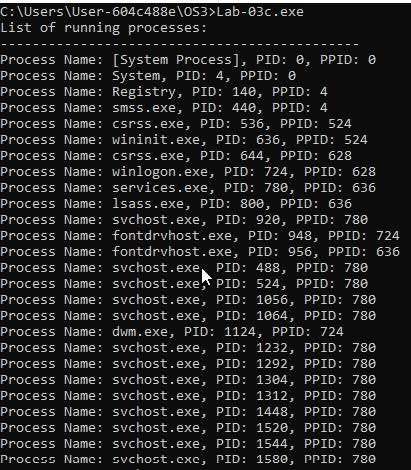


Рисунок 1.7 – Приложение Lab-03c

Как видно на рисунке 1.7, в результате мы получили список всех запущенных процессов. А именно: имя процесса, его id и id его родителя.

## **Уровень 7-8**

**Приложение Lab-03d-client**

Программа выполняет поиск простых чисел в заданном диапазоне. Диапазон задаётся через аргументы командной строки. Найденные простые числа сохраняются в промежуточном буфере, а затем после завершения работы цикла отправляются по анонимному каналу родительскому процессу.

Структура вызова данного приложения:

**Lab-03d-client** <Нижний порог диапазона> <Верхний порог диапазона> Предусмотреть обработку ошибок и корректную очистку всех ресурсов.

Для начала создаем файл Lab-03d-client.cpp. Записываем туда код, который представлен в листинге 1.5.

|  |
| --- |
| #include <windows.h>  #include <iostream>  #include <vector>  #include <string>  #include <sstream>  // Проверка, является ли число простым  bool isPrime(int n) {  if (n < 2) return false;  for (int i = 2; i \* i <= n; ++i)  if (n % i == 0) return false;  return true;  }  int main(int argc, char\* argv[]) {  if (argc != 3) {  // Пишем в stderr, так как stdout перенаправлен в пайп  std::cerr << "Usage: Lab-03d-client <lower> <upper>" << std::endl;  return 1;  }  int lower, upper;  try {  lower = std::stoi(argv[1]);  upper = std::stoi(argv[2]);  } catch (...) {  std::cerr << "Invalid arguments. Must be integers." << std::endl;  return 1;  }  if (lower > upper || lower < 0) {  std::cerr << "Invalid range: lower must be <= upper and >= 0." << std::endl;  return 1;  }  // НЕ используем GetStdHandle - просто пишем в std::cout  // Так как сервер перенаправил stdout в пайп    // Поиск простых чисел  std::ostringstream buffer;  for (int i = lower; i <= upper; ++i) {  if (isPrime(i)) {  buffer << i << " ";  }  }  std::string result = buffer.str();    // Просто выводим в stdout - сервер перехватит это через пайп  std::cout << result;    // Явно сбрасываем буфер  std::cout.flush();  return 0;  } |

Листинг 1.5 – Приложение Lab-03d-client

Затем компилируем из cpp-файла exe-файл. После чего запускаем приложение.



Рисунок 1.8 – Приложение Lab-03d-client

В результате выполнения в диапазоне 2 и 30 были перечислены все простые числа.

**Приложение Lab-03d-server**

Программа запускает заданное количество дочерних процессов для нахождения простых чисел. Также программа создаёт анонимный канал для работы с дочерними процессами. (Передача дескриптора через STARTUPINFO). При получении информации от дочернего процесса программа должна выводит её на консоль.

Структура вызова данного приложения:

**Lab-03d-server** <Количество создаваемых процессов> <Нижний порог диапазона> <Верхний порог диапазона>

Указанный диапазон должен быть разделён на равные части в количестве равном количеству создаваемых процессов. Программа не может быть завершена пока не завершены все дочерние процессы.

Предусмотреть обработку ошибок и очистку ресурсов.

Использовать Process Explorer и найти дескриптор канала в момент работы приложения.

***Примечание! Никакой синхронизации предусматривать на данном этапе не требуется!***

***Вопрос:*** *Какой тип дескриптора указан в Process Explorer для канала? Почему?*

Создаем файл Lab-03d-server.cpp. Записываем туда код, представленный в листинге 1.6.

|  |
| --- |
| #include <windows.h> #include <iostream> #include <string> #include <vector>  int main(int argc, char\* argv[]) {     if (argc != 4) {         std::cerr << "Usage: Lab-03d-server <num\_processes> <lower> <upper>" << std::endl;         return 1;     }  int numProcesses, globalLower, globalUpper;     try {         numProcesses = std::stoi(argv[1]);         globalLower = std::stoi(argv[2]);         globalUpper = std::stoi(argv[3]);     } catch (...) {  std::cerr << "Invalid arguments. Must be integers." << std::endl;         return 1;     }      if (numProcesses <= 0 || globalLower > globalUpper || globalLower < 0) {         std::cerr << "Invalid parameters." << std::endl;         return 1;     }      // Вычисляем шаг для разделения диапазона     int rangeSize = globalUpper - globalLower + 1;     int step = rangeSize / numProcesses;     int remainder = rangeSize % numProcesses;      std::vector<PROCESS\_INFORMATION> pi(numProcesses);     std::vector<HANDLE> hReadHandles(numProcesses);     std::vector<HANDLE> hWriteHandles(numProcesses);     std::vector<HANDLE> processHandles(numProcesses);      int currentLower = globalLower;     for (int i = 0; i < numProcesses; ++i) {         int partSize = step + (i < remainder ? 1 : 0);         int currentUpper = currentLower + partSize - 1;          // Проверка границ         if (currentUpper > globalUpper) currentUpper = globalUpper;          // Создаём анонимный пайп         SECURITY\_ATTRIBUTES sa = { sizeof(SECURITY\_ATTRIBUTES), NULL, TRUE };         if (!CreatePipe(&hReadHandles[i], &hWriteHandles[i], &sa, 0)) {             std::cerr << "Failed to create pipe for process " << i << std::endl;             return 1;         }          // Делаем write handle наследуемым         SetHandleInformation(hWriteHandles[i], HANDLE\_FLAG\_INHERIT, HANDLE\_FLAG\_INHERIT);          // Настраиваем STARTUPINFO         STARTUPINFO si = { sizeof(STARTUPINFO) };         si.hStdOutput = hWriteHandles[i];         si.hStdError = hWriteHandles[i];  // Перенаправляем и stderr         si.dwFlags = STARTF\_USESTDHANDLES;          // Команда для клиента         std::string cmdLine = "Lab-03d-client.exe " +                               std::to\_string(currentLower) + " " +                               std::to\_string(currentUpper);          // Создаём процесс         BOOL success = CreateProcess(             NULL,              cmdLine.data(),  // Убираем приведение типа             NULL, NULL,              TRUE,  // Наследуем handles             0, NULL, NULL,              &si,              &pi[i]         );          if (!success) {             std::cerr << "Failed to create process " << i << std::endl;             CloseHandle(hReadHandles[i]);             CloseHandle(hWriteHandles[i]);             continue;         }          // Закрываем ненужные handles         CloseHandle(hWriteHandles[i]);  // В родителе больше не нужен         hWriteHandles[i] = NULL;                  processHandles[i] = pi[i].hProcess;                  std::cout << "Created process " << i << " for range "                    << currentLower << " to " << currentUpper << std::endl;                  currentLower = currentUpper + 1;         if (currentLower > globalUpper) break;     }      // Ждём завершения всех процессов     DWORD waitResult = WaitForMultipleObjects(         numProcesses,          processHandles.data(),          TRUE,          INFINITE     );      if (waitResult == WAIT\_FAILED) {         std::cerr << "WaitForMultipleObjects failed" << std::endl;     } // Читаем данные из пайпов     std::cout << "\n=== RESULTS ===" << std::endl;     for (int i = 0; i < numProcesses; ++i) {         if (hReadHandles[i] && hReadHandles[i] != INVALID\_HANDLE\_VALUE) {             char buffer[4096];             DWORD bytesRead;             std::string output;                          // Читаем все доступные данные             while (ReadFile(hReadHandles[i], buffer, sizeof(buffer) - 1, &bytesRead, NULL) && bytesRead > 0) {                 buffer[bytesRead] = '\0';                 output += buffer;             }                          if (!output.empty()) {                 // Убираем лишние переносы строк                 if (output.back() == '\n') output.pop\_back();                 if (output.back() == '\r') output.pop\_back();                                  std::cout << "Process " << i << ": " << output << std::endl;             } else {                 std::cout << "Process " << i << ": No output" << std::endl;             }                          CloseHandle(hReadHandles[i]);         }                  // Закрываем handles процесса         if (pi[i].hThread) CloseHandle(pi[i].hThread);         if (pi[i].hProcess) CloseHandle(pi[i].hProcess);     }      std::cout << "All child processes completed." << std::endl;     return 0; }      return true;  }  int main(int argc, char\* argv[]) {      if (argc != 4) {          std::cerr << "Usage: Lab-03d-server <num\_processes> <lower> <upper>" << std::endl;          return 1;      }      int numProcesses, globalLower, globalUpper;      try {          numProcesses = std::stoi(argv[1]);          globalLower = std::stoi(argv[2]);          globalUpper = std::stoi(argv[3]);      } catch (...) {          std::cerr << "Invalid arguments. Must be integers." << std::endl;          return 1;      }      if (numProcesses <= 0 || globalLower > globalUpper || globalLower < 0) {          std::cerr << "Invalid parameters." << std::endl;          return 1;      }      // Вычисляем шаг для разделения диапазона      int rangeSize = globalUpper - globalLower + 1;      int step = rangeSize / numProcesses;      int remainder = rangeSize % numProcesses;      std::vector<PROCESS\_INFORMATION> pi(numProcesses);      std::vector<HANDLE> hReadHandles(numProcesses);      std::vector<HANDLE> hWriteHandles(numProcesses);      std::vector<HANDLE> processHandles(numProcesses);      int currentLower = globalLower;      for (int i = 0; i < numProcesses; ++i) {          int partSize = step + (i < remainder ? 1 : 0);          int currentUpper = currentLower + partSize - 1;          // Создаём анонимный пайп для этого процесса          SECURITY\_ATTRIBUTES sa = { sizeof(SECURITY\_ATTRIBUTES), NULL, TRUE };  // Наследуемый          BOOL pipeSuccess = CreatePipe(&hReadHandles[i], &hWriteHandles[i], &sa, 0);          if (!pipeSuccess) {              std::cerr << "Failed to create pipe for process " << i << ". Error: " << GetLastError() << std::endl;              // Очистка предыдущих              for (int j = 0; j < i; ++j) {                  CloseHandle(hReadHandles[j]);                  CloseHandle(hWriteHandles[j]);              }              return 1;          }          // Делаем write handle наследуемым для клиента          SetHandleInformation(hWriteHandles[i], HANDLE\_FLAG\_INHERIT, HANDLE\_FLAG\_INHERIT);          // Настраиваем STARTUPINFO для перенаправления stdout на write handle          STARTUPINFO si = { sizeof(STARTUPINFO) };          si.hStdOutput = hWriteHandles[i];          si.dwFlags = STARTF\_USESTDHANDLES;          // Команда для клиента          std::string cmdLine = "Lab-03d-client.exe " + std::to\_string(currentLower) + " " + std::to\_string(currentUpper);          BOOL createSuccess = CreateProcess(NULL, (LPSTR)cmdLine.c\_str(), NULL, NULL, TRUE, 0, NULL, NULL, &si, &pi[i]);          if (!createSuccess) {              std::cerr << "Failed to create process " << i << ". Error: " << GetLastError() << std::endl;              CloseHandle(hReadHandles[i]);              CloseHandle(hWriteHandles[i]);              continue;          }          processHandles[i] = pi[i].hProcess;          currentLower = currentUpper + 1;          std::cout << "Created process " << i << " for range " << (currentUpper - partSize + 1) << " to " << currentUpper << std::endl;      }      // Ждём завершения всех процессов      WaitForMultipleObjects(numProcesses, processHandles.data(), TRUE, INFINITE);      // Читаем и выводим данные из пайпов в отформатированном виде      for (int i = 0; i < numProcesses; ++i) {          if (hReadHandles[i]) {              char buffer[4096];              DWORD bytesRead;              std::string output;              do {                  BOOL readSuccess = ReadFile(hReadHandles[i], buffer, sizeof(buffer) - 1, &bytesRead, NULL);                  if (readSuccess && bytesRead > 0) {                      buffer[bytesRead] = '\0';                      output += buffer;                  }              } while (bytesRead > 0);              // Форматированный вывод              int processLower = globalLower + (i \* step) + (i < remainder ? i : remainder);              int processUpper = processLower + (i < remainder ? step : step - 1);              std::cout << "Process " << i << " [Range: " << processLower << "-" << processUpper << "]: " << output << std::endl;              CloseHandle(hReadHandles[i]);          }          CloseHandle(pi[i].hThread);          CloseHandle(pi[i].hProcess);      }      std::cout << "All child processes completed." << std::endl;      return 0;  } |

Листинг 1.6 – Приложение Lab-03d-server

Затем компилируем из cpp-файла exe-файл и запускаем его.

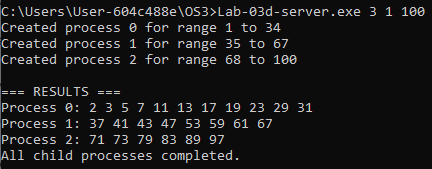


Рисунок 1.9 – Приложение Lab-03d-server

В результате было запущено 3 процесса, каждый из которых вернул список простых чисел. Затем запускаем Process Explorer для просмотра дескриптора.

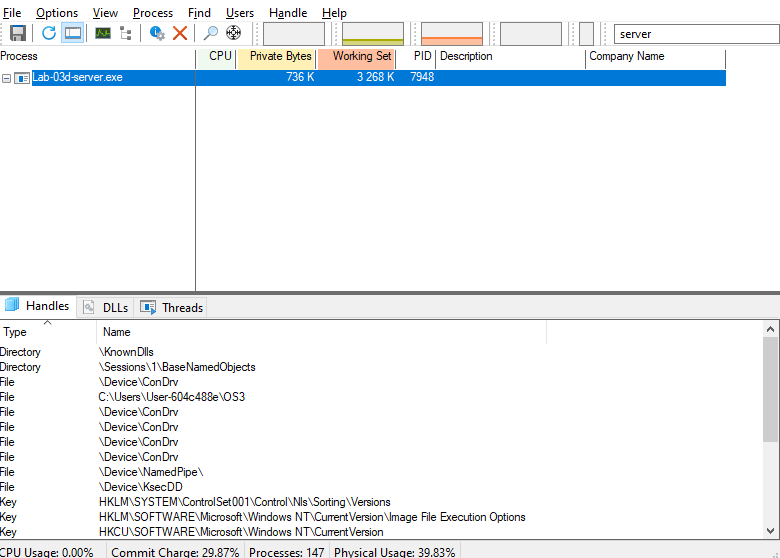


Рисунок 1.10 – Просмотр дескриптора

В Process Explorer для анонимных каналов, созданных через функцию CreatePipe, тип дескриптора указывается как File. Это происходит по фундаментальной архитектурной причине: в Windows все объекты ввода-вывода, включая каналы, сокеты и устройства, реализованы через единую файловую модель. Операционная система рассматривает анонимные каналы как специальный тип файловых объектов в системном пространстве имен \Device\NamedPipe.Такая архитектурная реализация позволяет Windows использовать унифицированные механизмы работы с различными ресурсами ввода-вывода. Когда вы создаете анонимный канал с помощью CreatePipe, система фактически создает два файловых дескриптора - один для чтения и один для записи, которые функционируют как виртуальные файлы в памяти. Именно поэтому Process Explorer, анализируя системные дескрипторы процесса, отображает их с типом File.

# **Постановка задачи для Linux**

## **2.1 Уровень 4-5**

**Приложение Lab-03x:**

Программа является полным аналогом программы из заданий для Windows.

Создаем файл Lab-03x.cpp и записываем туда код, который представлен в листинге 2.1.

#include <iostream>

#include <cstdlib>

#include <unistd.h> // для getpid() и usleep()

int main(int argc, char\* argv[]) {

int iterations = 0;

if (argc > 1) {

iterations = std::atoi(argv[1]);

} else {

const char\* envVar = std::getenv("ITER\_NUM");

if (envVar != nullptr) {

iterations = std::atoi(envVar);

} else {

std::cerr << "Error: no number of iterations is specified" << std::endl;

return 1;

}

}

std::cout << "Number of iterations: " << iterations << std::endl;

pid\_t pid = getpid();

for (int i = 0; i < iterations; ++i) {

std::cout << "Iteration " << (i + 1) << ", PID: " << pid << std::endl;

usleep(500000); // 500 ms

}

return 0;

}

Листинг 2.1 – Приложение Lab-03x

Затем компилируем cpp-файл в exe-файл. После чего запускаем.

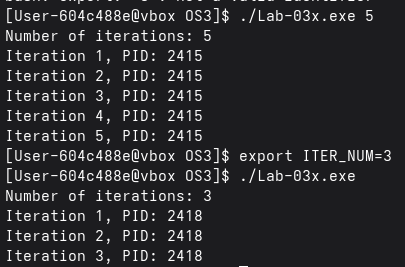


Рисунок 2.1 – Приложение Lab-03x

Проверили на выполнение наше приложение. Запустили с передачей параметров и с помощью глобальной переменной. Результат такой же как и у аналога на Windows.

**Приложение Lab-03a**

Программа создаёт два дочерних процесса и загружает в них исполняемый файл **Lab-03x**. При создании процессов должны быть выполнены следующие условия:

* В ходе запуска первого дочернего процесса параметр с количеством итераций должен быть передан как аргумент командной строки
* В ходе запуска второго дочернего процесса параметр с количеством итераций должен быть передан через переменную окружения ITER\_NUM. Переменная создается непосредственно в родительском процессе перед созданием второго дочернего процесса

Родительский процесс должен дождаться завершения всех дочерних процессов (выполняются одновременно).

Изучить информацию о созданных процессах через файловую систему (каталог /proc) и утилиту ps. Предусмотреть обработку ошибок и очистку ресурсов.

***Вопрос****: Все ли процессы созданы успешно? Если нет, почему?*

Для начала создаем файл Lab-03a.cpp и записываем туда код, который представлен в листинге 2.2.

#include <iostream>

#include <unistd.h>

#include <sys/wait.h>

#include <cstdlib>

int main() {

const char\* path = "./Lab-03x";

const char\* arg1 = "5";

pid\_t pid1 = fork();

if (pid1 == 0) {

// Первый процесс: аргумент командной строки

execl(path, path, arg1, NULL);

std::cerr << "Error starting the first process" << std::endl;

exit(1);

}

pid\_t pid2 = fork();

if (pid2 == 0) {

// Второй процесс: переменная окружения

setenv("ITER\_NUM", "7", 1);

execl(path, path, NULL);

std::cerr << "Error starting the second process" << std::endl;

exit(1);

}

// Родитель ждёт завершения обоих

waitpid(pid1, NULL, 0);

waitpid(pid2, NULL, 0);

std::cout << "Both child processes are completed." << std::endl;

return 0;

}

|  |
| --- |
| }      pid\_t pid2 = fork();      if (pid2 == 0) {          // Второй процесс: переменная окружения          setenv("ITER\_NUM", "7", 1);          execl(path, path, NULL);          std::cerr << "Error starting the second process" << std::endl;          exit(1);      }      // Родитель ждёт завершения обоих      waitpid(pid1, NULL, 0);      waitpid(pid2, NULL, 0);      std::cout << "Both child processes are completed." << std::endl;      return 0;  } |

Листинг 2.2 – Приложение Lab-03a

Затем компилируем из cpp-файла exe-файл. И запускаем его.

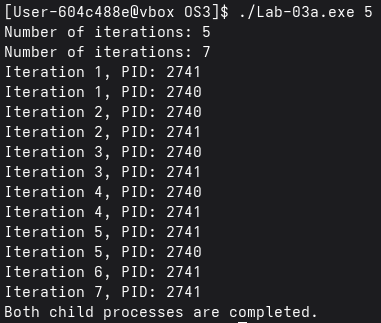


Рисунок 2.2 – Приложение Lab-03a

В результате все процессы были созданы успешно. Мы передали как параметр 5. Из-за этого у нас первый процесс выполнился 5 раз. А второй процесс выполнился 7 раз, из-за того, что в коде у нас была добавлена переменная окружения равная 7.

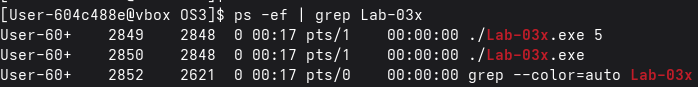


Рисунок 2.3 – Использование утилиты ps

Использовав утилиту ps, наши PID каждого дочернего элемента.

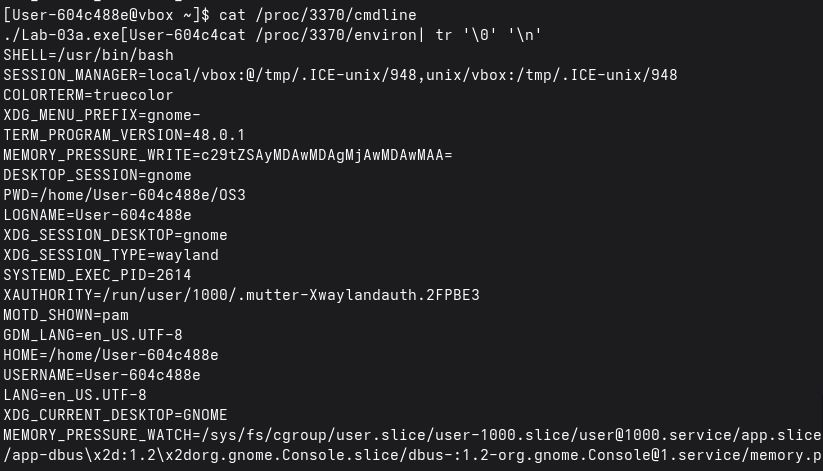


Рисунок 2.4 – Использование /proc

При помощи /proc просмотрели командную строку, переменные окружения, имя, состояние, PPID и память.

## **2.2 Уровень 7-8**

Написать аналог приложений **Lab-03d-\*** для ОС Linux. Предусмотреть обработку ошибок и очистку ресурсов.

Для начала создаем файл Lab-03d-client.cpp и записываем туда код, который представлен в листинге 2.3.

|  |
| --- |
| #include <iostream>  #include <cstdlib>  #include <unistd.h>  #include <vector>  #include <sstream>  bool isPrime(int n) {  if (n < 2) return false;  for (int i = 2; i \* i <= n; ++i)  if (n % i == 0) return false;  return true;  }  int main(int argc, char\* argv[]) {  if (argc != 3) {  std::cerr << "Usage: Lab-03d-client.exe <lower> <upper>" << std::endl;  return 1;  }  int lower = std::atoi(argv[1]);  int upper = std::atoi(argv[2]);  if (lower > upper || lower < 0) {  std::cerr << "Invalid range." << std::endl;  return 1;  }  std::ostringstream buffer;  for (int i = lower; i <= upper; ++i)  if (isPrime(i)) buffer << i << " ";  std::string result = buffer.str();  write(STDOUT\_FILENO, result.c\_str(), result.size());  return 0;  } |

Листинг 2.3 – Приложение Lab-03d-client

После чего создаем файл Lab-03d-server.cpp и записываем туда код, который представлен в листинге 2.4.

|  |
| --- |
| #include <iostream>  #include <unistd.h>  #include <sys/wait.h>  #include <vector>  #include <sstream>  #include <cstring>  int main(int argc, char\* argv[]) {      if (argc != 4) {          std::cerr << "Usage: Lab-03d-server.exe <num\_procs> <lower> <upper>" << std::endl;          return 1;      }      int numProcs = std::atoi(argv[1]);      int lower = std::atoi(argv[2]);      int upper = std::atoi(argv[3]);      if (numProcs <= 0 || lower > upper) {          std::cerr << "Invalid input." << std::endl;          return 1;      }      int rangeSize = (upper - lower + 1) / numProcs;      int remainder = (upper - lower + 1) % numProcs;      std::vector<pid\_t> children;      std::vector<int> readFDs;      for (int i = 0; i < numProcs; ++i) {          int start = lower + i \* rangeSize;          int end = start + rangeSize - 1;          if (i == numProcs - 1) end += remainder;          int pipefd[2];          if (pipe(pipefd) == -1) {              std::cerr << "Pipe creation failed." << std::endl;              continue;          }          pid\_t pid = fork();          if (pid < 0) {              std::cerr << "Fork failed." << std::endl;              close(pipefd[0]);              close(pipefd[1]);              continue;          }          if (pid == 0) {              // Child              close(pipefd[0]); // Close read end                      close(pipefd[1]);              std::ostringstream cmd;              cmd << "./Lab-03d-client.exe " << start << " " << end;              execl("/bin/sh", "sh", "-c", cmd.str().c\_str(), NULL);              std::cerr << "Exec failed." << std::endl;              exit(1);          } else {              // Parent              close(pipefd[1]); // Close write end              children.push\_back(pid);              readFDs.push\_back(pipefd[0]);          }      }      // Read from pipes      for (size\_t i = 0; i < readFDs.size(); ++i) {          char buffer[1024];          ssize\_t bytes;          std::cout << "Процесс #" << i + 1 << " вернул: ";          while ((bytes = read(readFDs[i], buffer, sizeof(buffer) - 1)) > 0) {              buffer[bytes] = '\0';              std::cout << buffer;          }          std::cout << std::endl;          close(readFDs[i]);      }      // Wait for children      for (pid\_t pid : children) {          waitpid(pid, NULL, 0);      }      std::cout << "Все дочерние процессы завершены." << std::endl;      return 0;  } |

Листинг 2.4 – Приложение Lab-03d-server

Затем компилируем оба этих файла из формата cpp в формат exe. И запускаем.

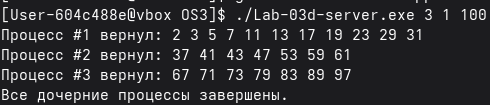


Рисунок 2.5 – Приложение Lab-03d-server

В результате мы видим вывод от каждого дочернего процесса с найденными простыми числами.

# **Заключение**

В ходе выполнения лабораторной работы было получено комплексное представление о механизмах создания, управления и анализа процессов в операционных системах Windows и Linux. На платформе Windows были реализованы приложения Lab-03x, Lab-03a, Lab-03b, Lab-03c, Lab-03d-client и Lab-03d-server, демонстрирующие работу с переменными окружения, аргументами командной строки, созданием дочерних процессов через CreateProcess, а также передачей данных по анонимным каналам. Также уделили внимание анализу дескрипторов через Process Explorer.

На платформе Linux были разработаны аналоги этих приложений с использованием системных вызовов fork, exec, pipe, waitpid, а также анализа процессов через файловую систему /proc и утилиту ps. Были реализованы сценарии передачи параметров через аргументы и переменные окружения, а также межпроцессная коммуникация с использованием перенаправления стандартного вывода. Приложения Lab-03x, Lab-03a, Lab-03d-client и Lab-03d-server позволили закрепить навыки работы с системными вызовами, обработкой ошибок и управлением ресурсами.

Сравнительный анализ поведения процессов в Windows и Linux, а также изучение структуры командной строки, переменных окружения и системных утилит, позволили углубить понимание архитектурных различий между платформами и сформировать устойчивые навыки системного программирования.