Московский Авиационный Институт

(Национальный Исследовательский Университет)

Институт №8 “Компьютерные науки и прикладная математика”

Кафедра №806 “Вычислительная математика и программирование”

Лабораторная работа №5 по курсу

«Операционные системы»

Группа: М8О-209БВ-24

Студент: Филь Н. А.

Преподаватель: Миронов Е.С.

Оценка: _____

Дата: 20.12.25

Москва, 2025

Постановка задачи

Цель работы:

Приобретение практических навыков диагностики работы программного обеспечения.

Задание:

Продемонстрировать ключевые системные вызовы, используемые в выполненных лабораторных работах (№1–4), с помощью утилиты strace. Подтвердить, что их использование соответствует вариантам заданий. Провести сводный анализ механизмов взаимодействия процессов, потоков и работы с памятью

Общий метод и алгоритм решения:

Strace - утилита командной строки для трассировки системных вызовов и сигналов в операционных системах Linux и других Unix-подобных системах. Её основное назначение - перехват и запись взаимодействия между пользовательским процессом и ядром операционной системы в реальном времени что полезно для отладки, анализа производительности и диагностики ошибок. Strace работает, используя механизм ядра ptrace (process trace), который позволяет одному процессу наблюдать и контролировать выполнение другого. При запуске с целевой программой strace перехватывает каждый системный вызов (например, открытие файла, запись в сеть, выделение памяти) на границе перехода из пользовательского пространства в пространство ядра и обратно. Это позволяет видеть не только факт вызова, но и его аргументы, возвращаемое значение и код ошибки (errno), если вызов завершился неудачно.

Основные флаги:

-p PID - присоединиться к уже работающему процессу с указанным идентификатором.

-c - подсчитать статистику по системным вызовам (время, вызовы, ошибки) и вывести сводку по завершении.

-f - трассировать также все дочерние процессы, созданные с помощью fork().

-e - фильтрация вывода

-o файл - вывести вывод трассировки в указанный файл вместо stderr.

-s размер - увеличить максимальную длину выводимых строк аргументов (по умолчанию часто 32 символа).

-t - выводить время в формате ЧЧ:ММ:СС при каждом вызове.

-T - показывать время, затраченное на каждый системный вызов.

-y - подробно выводить информацию о файловых дескрипторах (пути к файлам, сокетам).

-v - более подробный (verbose) вывод для некоторых вызовов.

-h - вывести справку по использованию.

Протокол работы программы

Анализ Лабораторной работы №1:

Тема: Процессы и межпроцессное взаимодействие (Pipe).

Вариант: Родительский процесс передает дочернему имя файла и числа для суммирования через неименованный канал .

Диагностика системных вызовов: В ходе анализа вывода strace были выделены следующие ключевые вызовы:

1. pipe2([3, 4], 0): Создание однонаправленного канала данных. Дескрипторы 3 и 4 используются для чтения и записи соответственно. Это подтверждает использование механизма pipe вместо других средств IPC.
2. clone(..., SIGCHLD, ...): Создание дочернего процесса (аналог fork). Флаг SIGCHLD указывает на то, что это отдельный процесс, а не поток.
3. write(4, ...) и read(0, ...): Родительский процесс записывает данные в канал (fd 4), а дочерний процесс, благодаря перенаправлению ввода (dup2), читает их как со стандартного ввода.
4. wait4(...): Родительский процесс ожидает завершения дочернего, что гарантирует синхронность выполнения задачи.

Вывод: Трассировка подтверждает реализацию передачи данных «Родитель -> Потомок» через pipe.

Анализ Лабораторной работы №2:

Тема: Потоки и синхронизация.

Вариант: Реализация метода k-средних (k-means) с использованием пула потоков.

Диагностика системных вызовов: Работа многопоточного приложения принципиально отличается на уровне системных вызовов:

1. clone(..., CLONE_VM|CLONE_THREAD|...): В отличие от ЛР1, здесь используется флаг CLONE_VM (общая виртуальная память) и CLONE_THREAD. Это доказывает, что создаются именно легковесные потоки (threads), а не изолированные процессы, что необходимо для обработки общего массива точек .
2. futex(..., FUTEX_WAIT, ...): Системный вызов, лежащий в основе работы мьютексов (pthread_mutex) и семафоров (sem_wait). Наличие этих вызовов в логе подтверждает работу механизмов синхронизации доступа к общим данным и ограничения количества активных потоков.
3. exit(0): Потоки завершаются стандартным образом после выполнения своей части вычислений.

Вывод: Анализ подтверждает использование общей памяти и примитивов синхронизации POSIX (семафоры, мьютексы) для параллельных вычислений.

Анализ Лабораторной работы №3:

Тема: Взаимодействие через отображаемые файлы (Memory Mapped Files).

Вариант: Передача массива чисел через файл, отображенный в память (mmap) .

Диагностика системных вызовов: Механизм IPC через файлы диагностируется следующей цепочкой:

1. openat(..., O_RDWR|O_CREAT|O_TRUNC, ...): Создание файла данных data.bin.
2. ftruncate(3, 40804): Установка размера файла, соответствующего размеру передаваемой структуры данных.
3. mmap(..., MAP_SHARED, ...): Ключевой вызов. Флаг MAP_SHARED указывает системе, что изменения в памяти должны быть немедленно отражены в файле и видны другим процессам. Это доказывает реализацию задания именно через shared memory, а не через копирование данных.
4. munmap(...) и unlinkat(...): Корректное освобождение памяти и удаление временного файла после завершения обмена данными.

Вывод: Использование MAP_SHARED подтверждает корректную реализацию обмена данными через отображаемые файлы.

Анализ Лабораторной работы №4:

Тема: Динамические библиотеки.

Вариант: Сравнение статической линковки и динамической загрузки (dlopen).

Диагностика системных вызовов: Сравнение трассировок двух программ показало различие в моментах загрузки кода:

1. Статическая линковка (prog_static): Библиотека libimpl1.so загружается загрузчиком ОС автоматически при старте программы, сразу после системных библиотек (libc). Вызовы openat и mmap для библиотеки происходят до передачи управления в main.
2. Динамическая загрузка (prog_dynamic):

- При старте загружается только libdl.so (интерфейс для работы с динамикой).
- Библиотека реализации (libimpl2.so) загружается только после явного действия пользователя (ввод команды, фиксируемый вызовом read). В логе strace видно, что вызов openat("./libimpl2.so") происходит в середине выполнения программы.

Вывод: strace наглядно демонстрирует разницу между автоматической загрузкой зависимостей при старте и загрузкой «по требованию» (on-demand) с помощью dlopen.

Вывод

В ходе выполнения лабораторной работы №5 были проанализированы результаты диагностики программного обеспечения, разработанного в рамках курса. Использование утилиты strace позволило:

1. Увидеть, как высокоуровневые функции C/C++ (fopen, pthread_create, dlopen) транслируются в системные вызовы ядра Linux (openat, clone, mmap).
2. Подтвердить корректность реализации вариантов заданий:

- Разделение памяти между потоками (CLONE_VM) в ЛР2.
- Использование разделяемых файлов (MAP_SHARED) в ЛР3.
- Динамическое связывание в ЛР4.

Полученные навыки диагностики позволяют эффективно отлаживать взаимодействие процессов и управление ресурсами в операционной системе.