

**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»**

Выполнил: А. А. Устинов
Группа: М8О-207БВ-24
Преподаватель: Е. С. Миронов

Москва, 2025

Условие

Цель работы: Приобретение практических навыков диагностики работы программного обеспечения.

Задание: При выполнении лабораторных работ по курсу ОС необходимо продемонстрировать ключевые системные вызовы, которые в них используются и то, что их использование соответствует варианту ЛР.

Вариант: -

Метод решения

Для проверки корректности выполненных лабораторных работ, а также для подтверждения использования предписанных системных вызовов, все разработанные программы были проанализированы с применением утилиты `strace`. Данная утилита предоставляет возможность отслеживать выполнение системных вызовов в процессе работы программы, фиксируя как передаваемые параметры, так и возвращаемые результаты.

Для каждой лабораторной работы формировался отдельный файл трассировки посредством команды следующего вида:

`strace -o trace_labN.log ./labN_executable [аргументы],`

где `trace_labN.log` — файл с результатами трассировки, а `labN_executable` — исполняемый файл, реализующий задание лабораторной работы номер N.

Полученные логи подвергались детальному анализу, включающему ручную проверку и элементы автоматизированной обработки, с целью:

- идентификации системных вызовов, обязательных в рамках соответствующего задания;
- контроля корректности их применения, включая последовательность вызовов, корректную обработку ошибок и допустимость передаваемых аргументов;
- подтверждения соответствия фактического поведения программы предполагаемой логике выполнения, включая управление процессами, механизмы синхронизации и обмен данными.

Описание Программы

Анализ системных вызовов лабораторной работы 1

1. `execve("./parent ['./parent']", 0x7fff77b42d10) = 0`

Системный вызов выполняет загрузку и запуск исполняемого файла, полностью заменяя текущий образ процесса.

- `"./parent"` — путь к исполняемому файлу;
- `["./parent"]` — массив аргументов командной строки (`argv`);
- `0x7fff77b42d10` — указатель на массив переменных окружения (`envp`).

2. `pipe2([3, 4], 0) = 0`

Создаётся неименованный канал для межпроцессного взаимодействия.

- 3 — файловый дескриптор конца канала для чтения;
- 4 — файловый дескриптор конца канала для записи;
- 0 — отсутствие флагов (канал создаётся без дополнительных опций).

3. `write(1, "Filename:\n 10) = 10`

Выполняется вывод строки-приглашения пользователю.

- 1 — стандартный поток вывода (`stdout`);
- "Filename:\n" — выводимая строка;
- 10 — количество записываемых байт.

4. `read(0, "input.txt\n 1024) = 10`

Осуществляется чтение имени файла из стандартного потока ввода.

- 0 — стандартный поток ввода (`stdin`);
- буфер для сохранения считанных данных;
- 1024 — максимальное число байт для чтения.

5. `openat(AT_FDCWD, "input.txt O_RDONLY) = 5`

Открывается файл для последующего чтения.

- AT_FDCWD — текущий рабочий каталог;
- "input.txt" — имя файла;
- O_RDONLY — режим открытия «только для чтения».

Файлу присвоен дескриптор 5.

6. `clone(...)` = 6474

Создаётся дочерний процесс. Аргументы скрыты в трассировке, но включают флаги клонирования и параметры стека. Возвращаемое значение 6474 соответствует идентификатору дочернего процесса.

7. `read(3, "Sum: 3354.30\n 4096) = 13`

Родительский процесс считывает данные из канала.

- 3 — файловый дескриптор чтения из `pipe`;
- буфер для принимаемых данных;
- 4096 — максимально допустимый размер чтения.

8. `write(1, "Sum: 3354.30\n 13) = 13`

Результат вычислений выводится в стандартный поток вывода.

- 1 — стандартный вывод;
- строка с результатом;
- 13 — длина выводимой строки.

9. -- SIGCHLD --

Родительский процесс получает сигнал SIGCHLD, уведомляющий о завершении дочернего процесса.

10. wait4(-1, NULL, 0, NULL) = 6474

Выполняется ожидание завершения дочернего процесса и освобождение его ресурсов.

- -1 — ожидание любого дочернего процесса;
- NULL — статус завершения не сохраняется;
- 0 — отсутствие дополнительных флагов;
- NULL — информация о потреблённых ресурсах не требуется.

Возвращён идентификатор завершившегося процесса.

Анализ системных вызовов лабораторной работы 2

```
execve("./matrix_filters", [ "./matrix_filters", "1000", "1000", "5", "4"], ...)
```

Системный вызов execve запускает новый процесс.

- "./matrix_filters" — путь к исполняемому файлу.
- Массив аргументов:
 - 1000 — число строк матрицы M .
 - 1000 — число столбцов матрицы N .
 - 5 — количество повторов фильтрации K .
 - 4 — максимальное число потоков.
- ... — указатель на переменные окружения.

```
brk(NULL)
```

Вызов brk запрашивает текущее положение границы кучи процесса. Используется библиотекой glibc для управления динамической памятью.

```
mmap(NULL, size, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0)
```

Выделение области памяти.

- NULL — адрес выбирается ядром автоматически.
- size — размер выделяемой области.
- PROT_READ | PROT_WRITE — разрешено чтение и запись.
- MAP_PRIVATE — изменения видны только процессу.

- MAP_ANONYMOUS — память не связана с файлом.
- -1, 0 — файловый дескриптор не используется.

Данный вызов используется для размещения матриц и служебных структур.

```
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC)
```

Загрузка стандартной библиотеки C.

- AT_FDCWD — путь задан относительно текущего каталога.
- O_RDONLY — файл открыт только для чтения.
- O_CLOEXEC — файл будет закрыт при execve.

```
mmap(...)
```

Отображение библиотеки libc.so в адресное пространство процесса.

```
clone3({flags=CLONE_VM|CLONE_FS|CLONE_FILES|CLONE_SIGHAND|CLONE_THREAD|...})
```

Создание нового потока выполнения.

- CLONE_VM — общая виртуальная память.
- CLONE_FS — общая файловая система.
- CLONE_FILES — общая таблица файловых дескрипторов.
- CLONE_SIGHAND — общие обработчики сигналов.
- CLONE_THREAD — поток в рамках одного процесса.

Этот вызов соответствует использованию pthread_create в программе.

```
futex(0x..., FUTEX_WAIT_BITSET|FUTEX_CLOCK_REALTIME, tid, NULL, FUTEX_BITSET_MATCH_ANY)
```

Механизм ожидания завершения потоков. Используется библиотекой pthread для реализации pthread_join.

```
openat(AT_FDCWD, "results.csv", O_WRONLY|O_CREAT|O_TRUNC, 0666)
```

Создание файла результатов.

- O_WRONLY — запись в файл.
- O_CREAT — создать файл при отсутствии.
- O_TRUNC — очистить содержимое.
- 0666 — права доступа (rw-rw-rw-).

```
write(3, "threads,time\n1,0.372066\n...", 46)
```

Запись результатов измерений в CSV-файл.

- 3 — файловый дескриптор `results.csv`.
- Стока содержит число потоков и измеренное время.

```
write(1, "Потоков: 1, время: 0.372066 сек", 47)
```

Вывод информации в стандартный поток вывода (`stdout`).

```
close(3)
```

Закрытие файла `results.csv`.

```
exit_group(0)
```

Корректное завершение процесса и всех потоков с кодом возврата 0.

Анализ системных вызовов лабораторной работы 3

Для анализа работы программы был использован инструмент `strace`, позволяющий отследить системные вызовы, выполняемые процессом во время запуска и завершения. Ниже приведено пояснение ключевых вызовов, непосредственно связанных с выполнением программы и работой операционной системы.

```
execve("./ipc_program ['./ipc_program']", envp) = 0
```

Системный вызов `execve` загружает исполняемый файл программы `ipc_program` в адресное пространство текущего процесса.

- первый аргумент — путь к исполняемому файлу;
- второй аргумент — массив аргументов командной строки;
- третий аргумент — массив переменных окружения.

Возврат значения 0 означает успешный запуск программы.

```
brk(NULL) = 0x5ea37c54b000
```

Вызов `brk` возвращает текущее положение конца сегмента кучи (heap), который используется для динамического выделения памяти.

```
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0)
```

Выделяется анонимная область виртуальной памяти размером 8192 байта.

- NULL — адрес выбирается ядром;
- 8192 — размер отображаемой области;
- `PROT_READ | PROT_WRITE` — разрешено чтение и запись;

- MAP_PRIVATE | MAP_ANONYMOUS — приватное отображение, не связанное с файлом;
- -1 — файловый дескриптор отсутствует;
- 0 — смещение.

```
openat(..., "/lib/x86_64-linux-gnu/libc.so.6 O_RDONLY|O_CLOEXEC)
```

Открывается динамическая библиотека `libc.so.6`, необходимая для выполнения стандартных функций языка С.

```
mmap(..., PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_FIXED)
```

Библиотека отображается в виртуальную память процесса с правами на чтение и выполнение. Это стандартный этап динамической линковки.

```
mprotect(address, size, PROT_READ)
```

Системный вызов `mprotect` изменяет права доступа к уже отображённым страницам памяти, запрещая запись и повышая безопасность выполнения программы.

```
write(2, "...Использование: ./ipc_program input_file 53)
```

Выполняется вывод диагностического сообщения в стандартный поток ошибок.

- 2 — файловый дескриптор `stderr`;
- строка — сообщение об ошибке использования программы;
- 53 — количество записанных байт.

```
exit_group(1)
```

Завершение процесса и всех его потоков с кодом возврата 1, что указывает на ошибочное завершение программы.

Анализ системных вызовов лабораторной работы 4

Анализ вывода утилиты `strace` позволяет наглядно увидеть различия в механизмах загрузки динамических библиотек между Программой №1 (Ранняя линковка) и Программой №2 (Поздняя линковка).

Случай №1: Программа с Ранней Линковкой (`./ipc_program`)

Объяснение ключевых вызовов:

- `execve(...)` = 0: Начало выполнения программы. Возврат 0 означает успешное выполнение.
- `openat(..., "/etc/ld.so.cache")` = 3: Процесс открывает кэш динамического загрузчика, чтобы быстро найти пути к системным библиотекам.
- `openat(..., "/lib/x86_64-linux-gnu/libc.so.6")` = 3: Загрузчик открывает стандартную библиотеку С (обязательная зависимость для почти всех программ).
- `mmap(...)` = 0x76058ec00000: Библиотека `libc.so.6` проецируется (`mmap`) в адресное пространство процесса для использования.

- `write(2, ...)`: Программа выводит сообщение об ошибке на стандартный поток ошибок (2).
- `exit_group(1)`: Программа завершает работу с кодом ошибки 1 (вероятно, из-за отсутствия аргументов).

Случай №2: Программа с Поздней Линковкой (`./prog2`)

Вывод `strace` для `prog2` демонстрирует ключевое отличие: использование `dlopen` для загрузки пользовательской библиотеки, что подтверждает механизм поздней линковки.

Листинг 1: Фрагмент `strace` для `./prog2`

```

1 execve("./prog2", ["/etc/ld.so.cache"], ...) = 0
2 openat(AT_FDCWD, "/etc/ld.so.cache", ...) = 3
3 openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libc.so.6", ...) = 3
4 [...]
5 munmap(0x7ee779763000, 30255) = 0
6 write(1, "\320\232\320\276\320\274\320\260\320\275\320\264\321\213:\n", 16) = 16
7 openat(AT_FDCWD, "./liblib1.so", O_RDONLY|O_CLOEXEC) = 3
8 fstat(3, {st_mode=S_IFREG|0755, st_size=15208, ...}) = 0
9 mmap(NULL, 16400, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7ee779766000
10 mmap(0x7ee779767000, 4096, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE,
      3, 0x1000) = 0x7ee779767000
11 close(3) = 0
12 write(1, "\320\243\321\201\320\277\320\265\321\210\320\275\320\276
      \320\267\320\260\320\263\321\200\321\203\320\266\320\265\320\275\320"..., 58) = 58

```

Ключевые вызовы поздней линковки:

- `openat(AT_FDCWD, "./liblib1.so O_RDONLY|O_CLOEXEC) = 3`: Вызов соответствует вызову `dlopen("./liblib1.so ...")` в коде `main_dynamic.c`. Программа сама инициирует открытие файла библиотеки. Возврат 3 — это файловый дескриптор.
- `mmap(NULL, 16400, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7ee779766000`: Библиотека `liblib1.so` проецируется (`mmap`) в память процесса. Аргумент 3 здесь — это файловый дескриптор, полученный от предыдущего `openat`.
- `close(3) = 0`: Файловый дескриптор, открытый для библиотеки, немедленно закрывается после того, как ее содержимое проецировано в память.
- `write(1, "2043210120772065211020752076 206720602063....") = 58`: Вывод сообщения "Успешно загружена Реализация 1. которое генерируется кодом сразу после успешного вызова `dlopen`.

Результаты

Результат вывода `strace` по лабораторной работе 1

Трассировка `strace` показывает, что программа корректно создаёт дочерний процесс, организует межпроцессное взаимодействие через неименованный канал и выполняет синхронизацию с помощью ожидания завершения дочернего процесса. Все используемые системные вызовы применяются в правильной последовательности и соответствуют логике работы лабораторной программы.

Результат вывода strace по лабораторной работе 2

Вывод утилиты `strace` показывает, что программа запускается как один процесс, динамически выделяет память и создаёт несколько потоков внутри этого процесса для параллельных вычислений. Многопоточность реализуется через системные вызовы `clone3` и `futex`, а результаты работы корректно записываются в файл, что подтверждает правильное использование библиотеки `pthread`.

Результат вывода strace по лабораторной работе 3

Результаты трассировки с помощью `strace` показывают, что программа корректно загружается в адресное пространство процесса, инициализирует виртуальную память и стандартные библиотеки, после чего переходит к выполнению пользовательского кода. Завершение работы происходит штатно с выводом диагностического сообщения, что подтверждает корректную обработку ошибок и взаимодействие программы с операционной системой.

Результат вывода strace по лабораторной работе 4

Анализ системных вызовов `strace` подтвердил фундаментальное различие в механизмах загрузки: в то время как системные библиотеки (`libc.so.6`) загружаются системным загрузчиком автоматически, `prog2` демонстрирует **позднюю линковку**, программно инициируя вызовы `openat` и `mmap` для загрузки `liblib1.so` уже в процессе выполнения.

Выводы

В ходе выполнения цикла лабораторных работ были успешно освоены фундаментальные механизмы системного программирования и взаимодействия программ с операционной системой. Анализ системных вызовов `strace` подтвердил освоение различных форм параллельной обработки: **межпроцессное взаимодействие** (ЛР 1) было реализовано через системные вызовы `fork`, `pipe` и `wait` для создания процессов и синхронизации через каналы; в то время как **многопоточность** (ЛР 2) использовала низкоуровневые вызовы `clone3` и `futex` для управления потоками внутри единого адресного пространства. Одновременно были изучены методы управления кодом и его загрузкой (ЛР 4). Был продемонстрирован контраст между автоматической загрузкой системных библиотек (`libc.so.6`) системным загрузчиком и программным управлением **поздней линковкой**, где тестовая программа `prog2` инициировала вызовы `openat` и `mmap` для загрузки `liblib1.so` уже во время выполнения. Это подтверждает успешное освоение интерфейса `dlopen` для реализации модульной архитектуры и плагинов. Наконец, базовая трассировка (ЛР 3) показала корректный процесс инициализации программы, включая выделение виртуальной памяти и штатную обработку системных зависимостей. Таким образом, был освоен полный цикл взаимодействия программы с ОС: от базовой загрузки и выделения ресурсов до реализации сложных механизмов параллелизма и динамического управления кодом.

Исходная программа

Strace Лабораторной работы 1

```
execve("./parent", ["./parent"], 0x7ffe01b3e6a0 /* 27 vars */) = 0
```


Strace Лабораторной работы 2

Strace Лабораторной работы 3

```
execve("./ipc_program", ["./ipc_program"], 0x7ffc644f3670 /* 27 vars */) = 0
brk(NULL) = 0x5fef289e2000
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7a2e6e1b
access("/etc/ld.so.preload", R_OK) = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=30327, ...}) = 0
mmap(NULL, 30327, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7a2e6e1ae000
```

Strace Лабораторной работы 4

```
close(3) = 0
mmap(NULL, 12288, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x76058ef
arch_prctl(ARCH_SET_FS, 0x76058ef94740) = 0
set_tid_address(0x76058ef94a10) = 1482
set_robust_list(0x76058ef94a20, 24) = 0
rseq(0x76058ef95060, 0x20, 0, 0x53053053) = 0
mprotect(0x76058edff000, 16384, PROT_READ) = 0
mprotect(0x5ea33ed8e000, 4096, PROT_READ) = 0
mprotect(0x76058efd7000, 8192, PROT_READ) = 0
prlimit64(0, RLIMIT_STACK, NULL, {rlim_cur=8192*1024, rlim_max=RLIM64_INFINITY}) = 0
munmap(0x76058ef97000, 30255) = 0
write(2, "\320\230\321\201\320\277\320\276\320\273\321\214\320\267\320\276\320\262\320\261") = 53
exit_group(1) = ?
```

Для prog2

