Курс: *Архитектура компьютера и ОС*

Лекция 1: Введение в ОС и системные вызовы

Лектор: Олег

Содержание

1 Введение и организационные моменты **2**
- 1.1 Формула оценки . . . 2
- 1.2 Работа с домашними заданиями . . . 2

2 Зачем нужна операционная система? **2**
- 2.1 Проблема прямого доступа к оборудованию . . . 3
- 2.2 Решение: операционная система как абстракция . . . 3

3 Работа с памятью в C++: краткое повторение **5**
- 3.1 Линейно адресуемая память и указатели . . . 5
- 3.2 Динамическая память . . . 5
- 3.3 Разделение аллокации и конструирования . . . 5
- 3.4 Арифметика указателей . . . 6

4 Взаимодействие с ОС: системные вызовы **6**
- 4.1 Системный вызов read . . . 6
- 4.2 Системный вызов write . . . 7
- 4.3 Обработка ошибок и частичных операций . . . 9

5 Работа с файлами **9**
- 5.1 Системные вызовы open и close . . . 10
- 5.2 Пример чтения из файла . . . 10

Словарь терминов **12**

1 Введение и организационные моменты

Этот курс посвящён изучению пользовательской части Операционная система (ОС) и архитектуры компьютера. Основная цель — понять, как программы выполняют свои действия на низком уровне, «под капотом» стандартных библиотечных функций. Курс рассчитан на один семестр и является обязательным, с возможностью выбрать продолжение во втором семестре.

1.1 Формула оценки

Итоговая оценка за курс формируется по следующей формуле:

$$\text{Оценка} = \min\left(10, 0.6 \cdot O_{\text{дз}} + 0.2 \cdot O_{\text{кр}} + 0.2 \cdot O_{\text{экз}} + 0.1 \cdot O_{\text{сем}}\right) \tag{1.1}$$

где:

- $O_{\text{дз}}$ — оценка за домашние задания.
- $O_{\text{кр}}$ — оценка за контрольные работы.
- $O_{\text{экз}}$ — оценка за экзамен.
- $O_{\text{сем}}$ — оценка за работу на семинарах.

Примечание

Сумма весовых коэффициентов в формуле (1.1) равна 1.1. Это означает, что с учётом бонусов за домашние задания можно набрать более 10 баллов, но итоговая оценка ограничивается 10 баллами.

1.2 Работа с домашними заданиями

Домашние задания будут выдаваться примерно раз в неделю со сроком выполнения 1–2 недели. Дедлайны «мягкие»: баллы за задание начинают убывать постепенно и достигают 10–20% от первоначальной стоимости через 3 недели после выдачи. Это сделано для того, чтобы студенты не жертвовали сном ради сдачи заданий в последний момент. Все задания будут выполняться в среде GEDLab.

2 Зачем нужна операционная система?

Рассмотрим простейшую программу на C++, которая считывает два числа и выводит их сумму (листинг 1).

```cpp
#include <iostream>

int main() {
    int a, b;
    std::cin >> a >> b;
    std::cout << a + b;
    return 0;
}
```

Листинг 1 – Программа для сложения двух чисел

На первый взгляд, все операции ввода-вывода выполняются благодаря библиотеке `iostream`. Однако в самом языке C++ нет встроенных механизмов для прямого взаимодействия с устройствами, такими как экран или клавиатура. Как же тогда текст появляется на мониторе?

2.1 Проблема прямого доступа к оборудованию

Представим модель, в которой программа напрямую взаимодействует с аппаратными компонентами компьютера: Центральный процессор (CPU), Оперативная память (RAM), жестким диском, сетевой картой и т.д. (рис. 1).

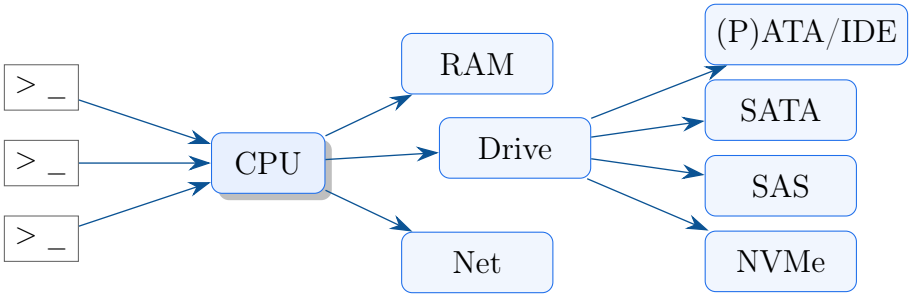

Рис. 1 – Модель прямого взаимодействия программы с оборудованием

Такая модель порождает две ключевые проблемы:

1. **Сложность и непереносимость.** Существует множество протоколов для взаимодействия с одним и тем же типом устройств. Например, для работы с дисками программа должна была бы поддерживать интерфейсы PATA, SATA, SAS, NVMe и другие. Аналогично, каждый производитель сетевых карт может предлагать свой уникальный протокол. Чтобы программа работала на разных компьютерах, ей пришлось бы реализовывать поддержку всех этих интерфейсов, что практически невозможно.

2. **Разделение ресурсов.** В современных системах одновременно запущены сотни и тысячи программ, в то время как количество ядер CPU ограничено единицами или десятками. Необходимо эффективно распределять процессорное время и другие ресурсы (память, доступ к дискам) между всеми программами. При прямом доступе программы к оборудованию сделать это было бы крайне затруднительно.

2.2 Решение: операционная система как абстракция

Для решения этих проблем была придумана ОС.

Определение: Операционная система

ОС — это программный слой, который выступает посредником между пользовательскими программами и аппаратным обеспечением компьютера.

ОС решает обе проблемы:

- Она **предоставляет унифицированный интерфейс** для работы с оборудованием. Программа работает не с конкретным жестким диском, а с абстракцией «файловой системы». ОС сама берёт на себя реализацию всех низкоуровневых протоколов.
- Она **управляет ресурсами**. ОС решает, какой программе и на какое время предоставить CPU, распределяет память, организует доступ к устройствам, предотвращая конфликты.

Эта модель показана на рис. 2.

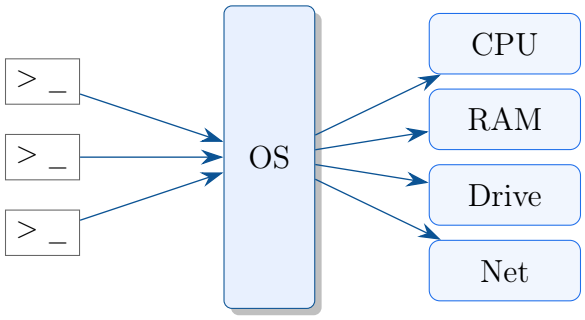

Рис. 2 – Операционная система как посредник

Итоги раздела

- Прямое взаимодействие программ с оборудованием сложно, непереносимо и не позволяет эффективно разделять ресурсы.
- ОС решает эти проблемы, предоставляя программам абстракции (файлы, сокеты) и управляя доступом к аппаратуре.

3 Работа с памятью в C++: краткое повторение

Взаимодействие с ОС в значительной степени происходит через память. Программа записывает данные в свою область памяти и затем просит ОС что-то с этими данными сделать. Поэтому важно освежить знания о модели памяти в C++.

3.1 Линейно адресуемая память и указатели

Память можно представить как большой массив байтов, где у каждого байта есть уникальный числовой адрес. Это называется моделью линейно адресуемой памяти.

Для работы с памятью используются **указатели** — переменные, которые хранят адрес. Разыменование указателя (`*ptr`) означает обращение к данным, лежащим по этому адресу. Тип указателя определяет, сколько байт будет прочитано и как они будут интерпретированы. Например, указатель типа `int64_t*` при разыменовании прочитает 8 байт и представит их как 64-битное целое число.

3.2 Динамическая память

Иногда память нужно выделить так, чтобы она «пережила» функцию, в которой была создана. Для этого используется динамическое выделение памяти в «куче» (heap).

```
// Allocating a single object
// Operator `new` allocates sizeof(T) bytes and constructs an object
T* ptr = new T{};

// Deleting an object and freeing memory
delete ptr;

// Allocating an array of 10 objects
T* arr = new T[10];

// Deleting all objects in the array and freeing memory
delete[] arr;
```

Листинг 2 – Работа с динамической памятью

Примечание

Важно соблюдать парность операторов: память, выделенную через `new`, нужно освобождать через `delete`. Память, выделенную через `new[]`, — через `delete[]`. Нарушение этого правила приводит к неопределённому поведению.

3.3 Разделение аллокации и конструирования

Оператор `new T` на самом деле выполняет две операции:

1. Выделение «сырой» (неинициализированной) памяти нужного размера.
2. Конструирование объекта типа `T` в этой памяти.

Эти шаги можно выполнить раздельно.

```
// 1. Allocate sizeof(T) raw bytes. operator new returns void*
void* raw_ptr = operator new(sizeof(T));

// 2. Construct an object of type T at the given address (placement new)
T* ptr = new (raw_ptr) T{};
```

```

// --- object `ptr` is ready to use ---

// 3. Explicitly call the destructor to destroy the object
ptr->~T();

// 4. Free the raw memory
operator delete(raw_ptr);
```

Листинг 3 – Явное управление памятью и объектами

Такой подход даёт больше контроля, но требует аккуратного ручного управления временем жизни объекта и памяти.

3.4 Арифметика указателей

В C++ арифметика указателей типизирована. Прибавление к указателю `ptr` единицы (`ptr + 1`) сдвигает его адрес не на 1 байт, а на `sizeof(*ptr)` байт, то есть к адресу следующего элемента в массиве.

- `ptr[i]` эквивалентно `*(ptr + i)`.
- Разность двух указателей одного типа `ptr1 - ptr2` даёт количество элементов (а не байт) между ними.

4 Взаимодействие с ОС: системные вызовы

Теперь, когда мы освежили знания о памяти, перейдём к основному механизму взаимодействия программы с ОС.

Определение: Системный вызов

Системный вызов — это основной интерфейс между пользовательскими программами и ядром ОС. Программа использует Системный вызов, чтобы попросить ядро выполнить действие, которое она не может выполнить сама (например, работать с файлом или сетью).

Рассмотрим два базовых системных вызова для ввода-вывода: `read` и `write`.

4.1 Системный вызов read

Функция `read` читает данные из источника, идентифицируемого Файловый дескриптор, в буфер.

```
#include <unistd.h>

ssize_t read(int fd, void *buf, size_t count);

// Example
char buf[10];
// Read from stdin (fd=0) into buf, at most 9 bytes
ssize_t bytes_read = read(0, buf, 9);

if (bytes_read == -1) {
    // Error occurred
} else if (bytes_read == 0) {
    // End of input (EOF)
```

```
} else {
    // Successfully read `bytes_read` bytes
}
```

Листинг 4 – Сигнатура и использование read

Аргументы:

- `int fd`: Файловый дескриптор источника данных. По соглашению, 0 — это Стандартный поток ввода.
- `void *buf`: Указатель на буфер, куда будут записаны данные.
- `size_t count`: Максимальное количество байт для чтения.

Возвращаемое значение (`ssize_t`):

- Положительное число: количество успешно прочитанных байт. **Важно:** `read` не гарантирует, что прочитает ровно `count` байт, даже если они доступны. Он может прочитать меньше.
- `0`: достигнут конец файла (EOF) или потока. Больше данных для чтения нет.
- `-1`: произошла ошибка. Код ошибки сохраняется в глобальной переменной errno.

4.2 Системный вызов write

Функция `write` записывает данные из буфера в приёмник, идентифицируемый Файловый дескриптор.

```
#include <unistd.h>

ssize_t write(int fd, const void *buf, size_t count);

// Example
const char msg[] = "Hello, world!\n";
// Write to stdout (fd=1), strlen(msg) bytes
// We use sizeof(msg) - 1 to exclude the terminating null byte ('\0')
ssize_t bytes_written = write(1, msg, sizeof(msg) - 1);

// Write the same message to stderr (fd=2)
write(2, msg, sizeof(msg) - 1);

if (bytes_written == -1) {
    // Error occurred
}
```

Листинг 5 – Сигнатура и использование write

Аргументы:

- `int fd`: Файловый дескриптор приёмника данных. 1 — Стандартный поток вывода, 2 — Стандартный поток ошибок.
- `const void *buf`: Указатель на буфер с данными для записи. Буфер константный, так как `write` его не изменяет.
- `size_t count`: Количество байт для записи.

Возвращаемое значение:

- Положительное число: количество успешно записанных байт. Как и `read`, `write` может записать меньше байт, чем было запрошено (например, если на диске закончилось место).
- `-1`: произошла ошибка, код которой записан в errno.

4.3 Обработка ошибок и частичных операций

Поскольку `read` и `write` могут обработать меньше данных, чем запрошено, для надёжной передачи всего объёма данных необходимо использовать циклы.

```cpp
// Writes exactly `count` bytes from `buf` to `fd`.
// Returns `true` on success, `false` on error.
bool WriteAll(int fd, const char* buf, size_t count) {
    size_t written = 0;
    while (written < count) {
        ssize_t res = write(fd, buf + written, count - written);
        if (res == -1) {
            return false; // An error occurred
        }
        written += res;
    }
    return true;
}
```

Листинг 6 – Надёжная функция для записи всех данных

Аналогичная функция `ReadAll` должна быть реализована для чтения, но с дополнительной проверкой на возврат 0 (EOF).

Для получения текстового описания ошибки по её коду из errno можно использовать функцию `strerror` из заголовка `<cstring>`.

```cpp
#include <cerrno>
#include <cstring>
#include <iostream>

// ... inside a function
if (!WriteAll(1, "Hello", 5)) {
    // errno is set by the last failed `write` call
    std::cerr << "Error writing data: " << strerror(errno) << std::endl;
    return 1; // Exit with error code
}
```

Листинг 7 – Обработка ошибок с выводом сообщения

Итоги раздела

- Системные вызовы — это API операционной системы.
- `read` и `write` — базовые вызовы для неформатированного ввода-вывода.
- Файловые дескрипторы 0, 1, 2 зарезервированы для стандартных потоков stdin, stdout, stderr.
- Всегда проверяйте возвращаемые значения системных вызовов на ошибки (-1) и обрабатывайте частичные операции.
- Для получения информации об ошибке используйте переменную errno.

5 Работа с файлами

Стандартные потоки — это лишь частный случай. Основное применение Файловый дескриптор — работа с файлами на диске.

5.1 Системные вызовы open и close

Чтобы работать с файлом, его сначала нужно открыть с помощью системного вызова `open`.

Определение: Файловый дескриптор

Файловый дескриптор — это неотрицательное целое число, которое ОС возвращает процессу при открытии файла. Процесс использует этот дескриптор во всех последующих операциях с файлом (`read`, `write`, `close`).

```
#include <fcntl.h> // For flags
#include <unistd.h>

int open(const char *pathname, int flags, ... /* mode_t mode */);
```

Листинг 8 – Сигнатура системного вызова open

Аргументы:

- `const char *pathname`: Путь к файлу.
- `int flags`: Флаги, определяющие режим доступа (например, `O_RDONLY` — только для чтения, `O_WRONLY` — только для записи, `O_RDWR` — для чтения и записи). Флаги можно комбинировать с помощью побитового ИЛИ (`|`).
- `mode_t mode`: (Опционально) Права доступа, которые устанавливаются, если файл создаётся с флагом `O_CREAT`.

В случае успеха `open` возвращает новый Файловый дескриптор (обычно наименьший из доступных). В случае ошибки возвращается -1, а errno устанавливается.

После завершения работы с файлом его дескриптор необходимо освободить с помощью системного вызова `close`.

```
#include <unistd.h>

int close(int fd);
```

Листинг 9 – Сигнатура системного вызова close

Если не закрывать файлы, это приведёт к утечке ресурсов (файловых дескрипторов), так как их количество для одного процесса ограничено.

5.2 Пример чтения из файла

В листинг 10 показан полный цикл работы: открытие файла, чтение из него, вывод содержимого в стандартный поток и закрытие.

```
#include <iostream>
#include <fcntl.h>
#include <unistd.h>
#include <cerrno>
#include <cstring>

int main() {
    const char* filename = "output.txt";
    int fd = open(filename, O_RDONLY);
```

```
    if (fd == -1) {
        std::cerr << "Failed to open file " << filename << ": "
                  << strerror(errno) << std::endl;
        return 1;
    }

    char buffer[1024];
    ssize_t bytes_read;

    // Read from file in a loop until EOF
    while ((bytes_read = read(fd, buffer, sizeof(buffer))) > 0) {
        // Write the read data to stdout
        if (!WriteAll(1, buffer, bytes_read)) {
            std::cerr << "Failed to write to stdout: "
                      << strerror(errno) << std::endl;
            close(fd);
            return 1;
        }
    }

    if (bytes_read == -1) {
        std::cerr << "Error reading from file: " << strerror(errno) << std::endl;
    }

    close(fd); // Don't forget to close the file!
    return 0;
}
// Assume WriteAll is defined as in listing 4.4
```

Листинг 10 – Чтение из файла и вывод в stdout

Итоги раздела

- Работа с файлом начинается с его открытия вызовом `open`, который возвращает Файловый дескриптор.
- Полученный Файловый дескриптор используется в вызовах `read` и `write` для взаимодействия с файлом.
- После окончания работы файл необходимо закрыть вызовом `close`, чтобы освободить ресурсы.
- Каждый `open` должен иметь парный `close`, подобно паре `new`/`delete`.

Словарь терминов

errno

Глобальная переменная в C/C++, в которую системные вызовы записывают код последней произошедшей ошибки..

Файловый дескриптор

Неотрицательное целое число, служащее идентификатором для доступа к файлу или другому ресурсу ввода-вывода в рамках одного процесса..

Системный вызов

Обращение пользовательской программы к ядру операционной системы для выполнения какой-либо привилегированной операции..

Стандартный поток ошибок

Отдельный поток для вывода сообщений об ошибках, обычно связанный с файловым дескриптором 2..

Стандартный поток ввода

Поток ввода данных по умолчанию, обычно связанный с файловым дескриптором 0..

Стандартный поток вывода

Поток вывода данных по умолчанию, обычно связанный с файловым дескриптором 1..