Курс: *Архитектура компьютера и ОС*

Лекция 5: Управление процессами и представление данных

Лектор: Евгений Соколов

Дата: 06.10.2025

Содержание

1 Управление процессами: группы и сигналы

Продолжая изучение процессов как единиц параллелизма с изолированными адресными пространствами, мы рассмотрим механизмы их организации и взаимодействия. Для эффективного управления множеством связанных процессов операционные системы, включая Linux, предоставляют инструменты для их группировки и асинхронного уведомления.

1.1 Группы процессов (Process Groups)

Для упрощения управления несколькими процессами одновременно они могут быть объединены в группы. Каждый процесс в системе принадлежит определённой группе.

Определение: Группа процессов

PGID (Process Group ID) — это числовой идентификатор, общий для нескольких процессов. Он позволяет применять операции, такие как отправка сигналов, ко всей группе сразу, а не к каждому процессу по отдельности. Каждый процесс также принадлежит SID (Session ID), которая объединяет группы процессов.

Для работы с PGID (Process Group ID) существуют системные вызовы:

- `getpgid(pid_t pid)`: получает PGID (Process Group ID) процесса с указанным `pid`. Вызов `getpgid(0)` вернёт PGID (Process Group ID) текущего процесса.
- `setpgid(pid_t pid, pid_t pgid)`: устанавливает PGID (Process Group ID) для процесса. Чтобы создать новую группу, обычно процесс вызывает `setpgid(0, 0)`, что создаёт новую группу с PGID (Process Group ID), равным `PID` этого процесса.

1.2 Сигналы (Signals)

Определение: Сигнал

сигнал — это простой механизм IPC (Inter-Process Communication), представляющий собой асинхронное уведомление, которое может быть отправлено процессу операционной системой или другим процессом. В отличие от пайпов, для отправки сигнала не требуется наличие родственной связи между процессами.

Сигналы, как и `PID`, являются просто числами. При получении сигнала процесс может отреагировать одним из нескольких способов:

- **Term (Termination):** Завершение процесса. Это действие по умолчанию для большинства сигналов.
- **Ign (Ignore):** Игнорирование сигнала.
- **Core:** Завершение процесса с генерацией core dump. Это файл, содержащий полный снимок адресного пространства процесса в момент сбоя, что позволяет проводить посмертную отладку (post-mortem debugging) с помощью таких инструментов, как GDB.
- **Stop:** Приостановка выполнения процесса.
- **Cont (Continue):** Возобновление выполнения приостановленного процесса.

Процесс может переопределить стандартную реакцию на большинство сигналов, установив собственный обработчик.

Таблица 1 – Некоторые распространённые сигналы и их действия по умолчанию

Сигнал	Номер	Действие	Комментарий
`SIGINT`	2	Term	Отправляется при нажатии `Ctrl+C` в терминале.
`SIGQUIT`	3	Core	Отправляется при нажатии `Ctrl+\`.
`SIGKILL`	9	Term	Гарантированно завершает процесс. Этот сигнал нельзя перехватить или проигнорировать.
`SIGSEGV`	11	Core	Segmentation Fault. Отправляется при попытке доступа к неразрешённой области памяти.
`SIGTSTP`	20	Stop	Отправляется при нажатии `Ctrl+Z` в терминале, приостанавливая процесс.

1.3 Отправка сигналов и ожидание процессов

Для отправки сигналов используется системный вызов `kill`. Несмотря на название, он может отправлять любой сигнал, а не только те, что завершают процесс.

```
#include <signal.h>
int kill(pid_t pid, int sig);
```

Листинг 1 – Сигнатура системного вызова `kill`

Аргумент `pid` интерпретируется следующим образом:

- `pid > 0`: Сигнал `sig` отправляется процессу с `PID`, равным `pid`.
- `pid < -1`: Сигнал отправляется всем процессам в группе с PGID (Process Group ID), равным `-pid`.
- `pid == 0`: Сигнал отправляется всем процессам в группе текущего процесса.
- `pid == -1`: Сигнал отправляется всем процессам, которым текущий пользователь имеет право отправлять сигналы (за исключением некоторых системных процессов).

Механизмы ожидания дочерних процессов, такие как `wait` и `waitpid`, позволяют не только дождаться завершения, но и получить информацию о причине.

- `WIFEXITED(status)`: Возвращает `true`, если процесс завершился штатно через вызов `exit()`. Код возврата можно получить с помощью `WEXITSTATUS(status)`.
- `WIFSIGNALED(status)`: Возвращает `true`, если процесс был завершён сигналом. Номер сигнала можно получить через `WTERMSIG(status)`.

Вызов `waitpid` также интегрирован с группами процессов:

- `waitpid(-1, ...)`: Ждёт любого дочернего процесса (стандартное поведение).
- `waitpid(-pgid, ...)`: Ждёт завершения любого дочернего процесса из группы с PGID (Process Group ID).

1.4 Управление памятью при `fork()`

Ранее мы говорили, что `fork()` создаёт полную копию адресного пространства родителя для дочернего процесса. Для современных процессов, занимающих гигабайты памяти, полное копирование было бы крайне неэффективным.

Определение: Copy-on-Write (COW)

Copy-on-Write (копирование при записи) — это оптимизация, применяемая при вызове `fork()`. Вместо реального копирования всех страниц памяти, операционная система создаёт для дочернего процесса новые таблицы страниц, которые указывают на те же физические страницы, что и у родителя. Все эти страницы помечаются как доступные только для чтения. При попытке записи в такую страницу (и родителем, и ребёнком) происходит аппаратное прерывание. ОС перехватывает его, создаёт реальную копию только этой конкретной страницы, и уже в неё производится запись. Это позволяет копировать только те данные, которые действительно изменяются, экономя время и память.

1.4.1 Общая память через `mmap`

Хотя Copy-on-Write (копирование при записи) обеспечивает изоляцию, иногда процессам нужна общая область памяти для эффективного взаимодействия. Этого можно достичь с помощью системного вызова `mmap` с флагом `MAP_SHARED`.

Примечание

Если участок памяти был создан с флагом `MAP_PRIVATE`, то при `fork()` к нему применяется Copy-on-Write (копирование при записи). Если же был использован флаг `MAP_SHARED`, то этот участок памяти будет общим для родителя и ребёнка после `fork()`: изменения, сделанные одним процессом, будут видны другому.

Важно помнить, что при работе с общей памятью возникает проблема синхронизации. Нет гарантий относительно порядка выполнения инструкций в родителе и ребёнке. Чтобы корректно читать данные, записанные другим процессом, необходимо использовать механизмы синхронизации, например, пайпы или сигналы, чтобы уведомить читающий процесс о готовности данных.

Итоги раздела

- **Группы процессов** (PGID (Process Group ID)) упрощают управление множеством процессов, позволяя применять операции ко всей группе сразу.
- **Сигналы** — это механизм асинхронных уведомлений для межпроцессного взаимодействия.
- Вызов `kill` позволяет отправлять сигналы процессам и группам процессов.
- `waitpid` может ожидать завершения процессов из определённой группы.
- **Copy-on-Write** (Copy-on-Write (копирование при записи)) оптимизирует `fork()`, откладывая реальное копирование страниц памяти до первой операции записи.
- `mmap` с флагом `MAP_SHARED` создаёт общую область памяти для родителя и дочерних процессов, но требует явной синхронизации доступа.

2 Представление данных

Любые данные в компьютерных системах — будь то файлы, сетевые пакеты или потоки ввода-вывода — в конечном счёте представляются в виде последовательности байт. Способ преобразования структурированных данных в байты и обратно определяет формат данных.

2.1 Текстовые и бинарные форматы

Форматы данных условно делятся на две большие категории.

- **Текстовые форматы** (JSON, YAML, XML, TXT) оптимизированы для чтения и редактирования человеком. Они, как правило, избыточны и требуют больше ресурсов для парсинга (синтаксического анализа).
- **Бинарные форматы** (исполняемые файлы ELF, архивы ZIP, форматы сериализации BSON, Protocol Buffers) оптимизированы для машинной обработки, компактности или скорости. Они нечитаемы для человека без специальных инструментов, но обрабатываются программами гораздо эффективнее.

Некоторые бинарные форматы, например, исполняемые файлы ELF (Executable and Linkable Format), спроектированы так, что их можно загрузить в память простым отображением файла с помощью `mmap`, без дополнительной обработки.

2.1.1 Тонкости бинарных форматов: порядок байтов

При работе с бинарными данными, содержащими числа размером более одного байта, возникает проблема порядок байтов (endianness).

Определение: Порядок байтов (Endianness)

порядок байтов (endianness) определяет, как байты многобайтового числа располагаются в оперативной памяти.

- **Little-endian**: Младший байт числа хранится по младшему адресу памяти. Этот порядок используется в большинстве современных архитектур, включая x86-64.
- **Big-endian**: Старший байт числа хранится по младшему адресу. Этот порядок часто используется в сетевых протоколах (network byte order).

Проблема возникает при передаче бинарных данных между системами с разным порядком байтов. Если данные записываются и читаются на одной и той же машине, беспокоиться о порядке байтов не нужно.

2.2 Кодировки текста: от ASCII до Unicode

Представление текста — одна из фундаментальных задач. Исторически первой и наиболее влиятельной кодировкой стала ASCII (American Standard Code for Information Interchange).

Определение: ASCII

ASCII (American Standard Code for Information Interchange) — стандарт, определяющий соответствие между числами (кодами от 0 до 127) и символами. Он использует только 7 бит, восьмой бит всегда равен нулю. ASCII включает в себя символы английского алфавита, цифры, знаки препинания и управляющие символы.

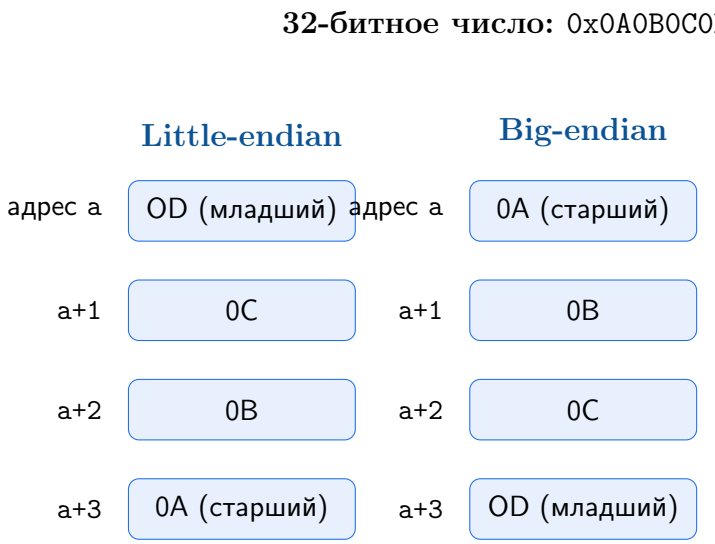

Рис. 1 – Расположение байтов числа `0x0A0B0C0D` в памяти при разном порядке байтов

Среди управляющих символов ASCII есть:

- `0x0A (\n)` — перевод строки (Line Feed).
- `0x0D (\r)` — возврат каретки (Carriage Return).
- `0x1B (ESC)` — Escape, используется для начала управляющих последовательностей, например, для управления цветом текста в терминале.

Примечание

Исторически сложилось два основных способа кодирования конца строки в текстовых файлах:

- **Unix/Linux**: используется один символ `\n`.
- **DOS/Windows**: используется последовательность из двух символов `\r\n` (возврат каретки и перевод строки).

Это различие может вызывать проблемы, например, при запуске скриптов с Windows-окончаниями строк в Linux, так как интерпретатор может неверно обработать `\r` в конце строки shebang.

Очевидным недостатком ASCII (American Standard Code for Information Interchange) является отсутствие поддержки символов других языков. Это привело к появлению множества несовместимых 8-битных кодировок (семейства ISO-8859, CP1251, KOI8-R и др.), которые использовали старший бит для кодирования национальных алфавитов. Проблема усугублялась в языках с иероглифической письменностью, где требовались многобайтовые кодировки со сложной логикой переключения режимов (например, EUC-JP).

2.3 Unicode и его кодировки

Для решения проблемы хаоса кодировок был создан стандарт Unicode.

Определение: Unicode

Unicode — это стандарт, который сопоставляет каждому символу из большинства мировых письменностей (включая мёртвые языки и эмодзи) уникальное целое число, называемое **Unicode**. Всего стандарт определяет более миллиона кодовых позиций (от `U+0000` до `U+10FFFF`).

Ключевые свойства Unicode:

- Первые 128 кодовых позиций (`U+0000` – `U+007F`) полностью совпадают с ASCII (American Standard Code for Information Interchange), обеспечивая частичную совместимость.
- Unicode сам по себе не является кодировкой. Он лишь определяет соответствие «символ ↔ число». Для представления этих чисел в виде байтов используются специальные кодировки (encodings).
- Стандарт имеет свои сложности: один и тот же видимый символ (глиф) может быть представлен либо одним кодпоинтом, либо комбинацией из базового символа и модифицирующего знака (например, диакритики). Например, символ 'ё' можно представить как `U+0451` или как комбинацию 'е' (`U+0435`) и диерезиса (`U+0308`).

2.3.1 Кодировка UTF-8

Существует несколько способов кодирования кодпоинтов Unicode в байты: UCS-2, UCS-4, UTF-16. Наиболее популярным и де-факто стандартом в вебе и современных ОС стал UTF-8 (Unicode Transformation Format, 8-bit).

Определение: UTF-8

UTF-8 (Unicode Transformation Format, 8-bit) — это кодировка Unicode с переменной длиной символа. Она кодирует кодпоинты в последовательности от 1 до 4 байт.

Основные преимущества UTF-8:

- **Совместимость с ASCII:** Любой текст в кодировке ASCII является корректным текстом в UTF-8, так как кодпоинты до `U+007F` кодируются одним байтом, полностью совпадающим с их ASCII-представлением.
- **Эффективность:** Маленькие значения кодпоинтов кодируются меньшим числом байт. Это экономит место для текстов на латинице.
- **Надёжность:** Нулевой байт (`0x00`) используется только для кодирования нулевого кодпоинта (`U+0000`). Это позволяет использовать стандартные C-функции для работы со строками, оканчивающимися нулём.
- **Самосинхронизация:** По значению любого байта можно определить, является ли он началом символа или частью многобайтовой последовательности. Это позволяет легко находить границы символов в потоке байт.

Принцип кодирования в UTF-8 основан на использовании старших битов первого байта для указания общей длины последовательности (см. Table 2).

2.4 Работа с Unicode в C/C++

Для поддержки Unicode в языках C и C++ существует тип `wchar_t` («широкий символ»).

Таблица 2 – Схема кодирования символов в UTF-8

Длина	Диапазон кодпоинтов	Схема байтов (x — биты кодпоинта)
1 байт	U+0000 – U+007F	`0xxxxxxx`
2 байта	U+0080 – U+07FF	`110xxxxx 10xxxxxx`
3 байта	U+0800 – U+FFFF	`1110xxxx 10xxxxxx 10xxxxxx`
4 байта	U+10000 – U+10FFFF	`11110xxx 10xxxxxx 10xxxxxx 10xxxxxx`

- `wchar_t`: Тип для хранения одного кодпоинта. Его размер зависит от платформы, но часто составляет 4 байта (32 бита), что достаточно для любого символа Unicode (аналогично кодировке UCS-4 в памяти).
- `L'a'`: Литерал типа `wchar_t`.
- `L"строка"`: Широкий строковый литерал (массив `const wchar_t`).
- Стандартная библиотека предоставляет аналоги для работы с широкими строками: `std::wstring`, `std::wcin`, `std::wcout` в C++ и функции `wprintf`, `wscanf` в C.

2.4.1 Локали

Чтобы стандартная библиотека знала, как преобразовывать широкие символы (`wchar_t`) в байтовые последовательности (например, в UTF-8) при вводе-выводе, ей необходимо сообщить текущие региональные настройки.

Определение: Локаль

локаль — это набор параметров, описывающих языковые и культурные особенности пользователя. локаль определяет кодировку текста (`LC_CTYPE`), формат чисел, валюты, даты и времени. Она обычно задаётся через переменные окружения, такие как `LANG` или `LC_ALL`.

Перед началом работы с широким вводом-выводом в программе на C++ необходимо установить глобальную локаль, чтобы она была унаследована от настроек операционной системы.

```cpp
#include <locale>
#include <iostream>
#include <string>

int main() {
    // Set the global locale based on the system's environment variables.
    // An empty string "" means "take from environment".
    std::locale::global(std::locale(""));

    // Now std::wcin and std::wcout will work correctly
    // with the encoding specified in the locale (e.g., UTF-8).
    std::wstring s;
    std::wcout << L"input text: ";
    std::wcin >> s;
    std::wcout << L"you input: " << s << L", len: " << s.size() << L" symb" << std
        ::endl;

    return 0;
```

```
18 }
```

Листинг 2 – Установка локали для корректной работы с Unicode в C++

Примечание

Смешивать обычные потоки (`std::cout`) и широкие (`std::wcout`) в одной программе не рекомендуется. Такое смешивание может привести к непредсказуемому поведению и некорректному выводу, так как внутреннее состояние потоков может быть нарушено.

Итоги раздела

- Данные могут быть представлены в **текстовом** (человекочитаемом) или **бинарном** (машиночитаемом) формате.
- При работе с бинарными данными важно учитывать **порядок байтов** (порядок байтов (endianness)), особенно при передаче данных между разными системами.
- **ASCII** — исторически важная, но ограниченная 7-битная кодировка.
- **Unicode** является универсальным стандартом, присваивающим уникальный номер (Unicode) каждому символу.
- **UTF-8** — самая популярная кодировка для Unicode, эффективная и обратно совместимая с ASCII.
- В C/C++ для работы с Unicode используется тип `wchar_t` и связанные с ним строковые классы и функции ввода-вывода.
- Для корректного ввода-вывода Unicode-текста необходимо настроить **локаль** программы, чтобы она соответствовала системным настройкам.