Курс: *Архитектура компьютера и ОС*

Лекция 3: Управление памятью

Лектор: Евгений Соколов

Дата: 06.10.2025

Содержание

1 Дополнительные инструменты для работы с файловой системой

На прошлом занятии мы рассмотрели основы работы с файловой системой. Сегодня мы завершим эту тему, изучив несколько оставшихся, но важных инструментов, которые могут пригодиться в практических задачах.

1.1 Новые флаги для системного вызова `open`

Системный вызов `open` имеет несколько полезных флагов, которые мы не обсуждали ранее.

- `O_TRUNC`: Этот флаг позволяет при открытии файла немедленно обрезать его размер до нуля. Это удобная альтернатива последовательному вызову `open` и `ftruncate`, если содержимое файла нужно полностью перезаписать.
- `O_PATH`: Позволяет получить файловый дескриптор, который ссылается не на сам файл, а на его путь в файловой системе. Такой дескриптор имеет ограниченное применение (например, из него нельзя читать или в него писать), но он полезен для передачи в другие системные вызовы, такие как `fstat`, для получения информации об объекте файловой системы (включая директории), не открывая его для операций ввода-вывода.
- `O_NOFOLLOW`: Если путь, передаваемый в `open`, является символической ссылкой, то с этим флагом вызов не будет переходить по ней, а вернёт ошибку. Это важно для безопасности, чтобы избежать работы с непредусмотренным файлом.

1.2 Получение метаданных о файлах: семейство `stat`

Для получения подробной информации о файле или директории используется семейство системных вызовов `stat`. Они заполняют структуру `struct stat`, содержащую метаданные об объекте.

```c
#include <sys/stat.h>

int stat(const char* path, struct stat* statbuf);
int lstat(const char* path, struct stat* statbuf);
int fstat(int fd, struct stat* statbuf);
```

Листинг 1 – Сигнатуры функций семейства `stat`

Ключевые различия между вызовами:

- `stat`: Принимает путь к файлу. Если путь указывает на символическую ссылку, `stat` переходит по ней и возвращает информацию о файле, на который она указывает.
- `lstat`: Аналогичен `stat`, но **не** переходит по символическим ссылкам. Вместо этого он возвращает информацию о самой ссылке.
- `fstat`: Принимает файловый дескриптор, полученный ранее через `open`.

Структура `struct stat` содержит множество полезных полей:

- `st_mode`: Тип файла (обычный файл, директория, символическая ссылка и т.д.) и права доступа к нему (чтение, запись, исполнение для владельца, группы и остальных).
- `st_uid` и `st_gid`: ID пользователя и группы-владельца файла.
- `st_size`: Размер файла в байтах.

- `st_blocks`: Количество дисковых блоков, занимаемых файлом.
- `st_atim`, `st_mtim`, `st_ctim`: Временные метки последнего доступа, последней модификации содержимого и последней модификации метаданных соответственно.

```c
#include <fcntl.h>
#include <sys/stat.h>
#include <unistd.h>

// ...

int fd = open(path, O_RDONLY | O_PATH | O_NOFOLLOW);
if (fd == -1) { /* handle error */ }

struct stat stats;
if (fstat(fd, &stats) == -1) { /* handle error */ }

close(fd);

if (S_ISDIR(stats.st_mode)) {
    //
} else if (S_ISLNK(stats.st_mode)) {
    //
} else if (S_ISREG(stats.st_mode)) {
    //
}
```

Листинг 2 – Пример использования `fstat` для определения типа объекта

В этом примере используется флаг `O_PATH`, чтобы безопасно получить дескриптор для проверки типа объекта, не открывая его для полноценной работы.

2 Управление памятью: виртуальная адресация

2.1 Проблема модели линейной памяти

Мы привыкли думать о памяти как о большом непрерывном массиве байтов. Однако эта модель не соответствует действительности. Проведём простой эксперимент: создадим две переменные — одну в куча (через `new`), а другую на стек (локальная переменная) — и выведем их адреса.

```c
#include <iostream>

int main() {
    int* heap_var = new int(10);
    int stack_var = 20;

    std::cout << "Heap address: " << (void*)heap_var << std::endl;
    std::cout << "Stack address: " << (void*)&stack_var << std::endl;

    long long diff = (long long)&stack_var - (long long)heap_var;
    std::cout << "Difference (bytes): " << diff << std::endl;
    //  64-

    delete heap_var;
```

```
15     return 0;
16 }
```

Листинг 3 – Сравнение адресов в стеке и куче

Разница между этими адресами может составлять десятки терабайт, что очевидно превышает объём физической оперативной памяти любого современного компьютера. Это наблюдение доказывает, что адреса, с которыми мы работаем в программе, не являются прямыми физическими адресами.

2.2 Виртуальная и физическая память

Для решения проблемы изоляции и безопасности процессов операционные системы вводят абстракцию — виртуальная память.

Определение: Виртуальная и физическая память

- **физическая память** — это реальные микросхемы оперативной памяти (RAM) в компьютере. Её адреса последовательны и ограничены её физическим объёмом.
- **виртуальная память** — это логическое адресное пространство, которое ОС предоставляет каждому процессу. Каждый процесс «видит» свой собственный, изолированный массив памяти, начинающийся с нуля. Адреса в этом пространстве называются **виртуальными**.

Процессор с помощью специального модуля (MMU — Memory Management Unit) и при содействии операционной системы преобразует виртуальные адреса в физические при каждом обращении к памяти. Это преобразование прозрачно для программиста.

2.3 Страничная организация памяти

Преобразование адресов происходит не для каждого байта в отдельности, а для блоков памяти фиксированного размера, называемых **страница памяти**.

Примечание

На большинстве современных систем (x86-64) размер страницы составляет 4 килобайта (4096 bytes, или $0x1000$ в шестнадцатеричной системе). Узнать точный размер страницы в системе можно с помощью вызова `sysconf(_SC_PAGESIZE)`.

Операционная система поддерживает для каждого процесса таблицу страниц, которая устанавливает соответствие между страницами виртуальной и физической памяти.

При обращении к адресу, например, `0x2345`:

1. Процессор разделяет его на номер страницы и смещение. Для страниц размером $0x1000$ адрес `0x2345` — это смещение `0x345` внутри страницы `2`.
2. С помощью таблицы страниц находится физический фрейм, соответствующий виртуальной странице `2` (на рис. **??** это фрейм A).
3. Процессор обращается к физической памяти по адресу, равному начальному адресу фрейма A плюс смещение `0x345`.

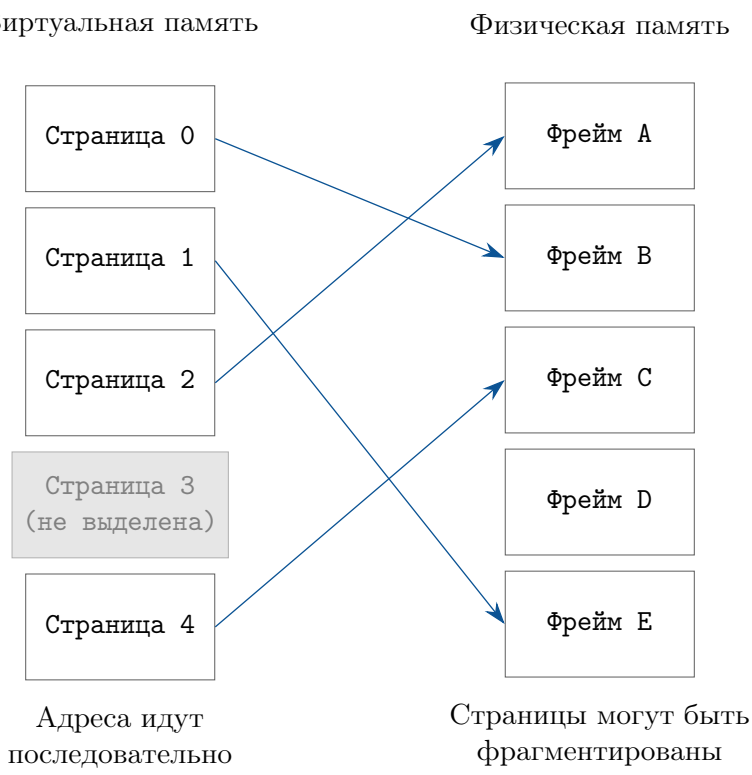

Рис. 1 – Схема отображения виртуальных страниц на физические фреймы памяти. Две соседние виртуальные страницы не обязательно отображаются в соседние физические.

Итоги раздела

Виртуальная память обеспечивает изоляцию процессов, позволяет программам работать с большим адресным пространством, чем доступно физической памяти, и упрощает управление памятью для ОС. Это достигается за счёт постраничного отображения виртуальных адресов на физические.

3 Системные вызовы для управления памятью: mmap

Для управления виртуальным адресным пространством процесса в POSIX-системах используется системный вызов `mmap` и его пара `munmap`.

```c
#include <sys/mman.h>

void *mmap(void *addr, size_t length, int prot, int flags,
           int fd, off_t offset);

int munmap(void *addr, size_t length);
```

Листинг 4 – Сигнатуры `mmap` и `munmap`

`mmap` — это мощный, но сложный инструмент, который выполняет две основные функции:

1. **Анонимное отображение**: выделение новых страниц оперативной памяти для процесса.
2. **Файловое отображение**: отображение содержимого файла (или его части) в виртуальное адресное пространство процесса.

3.1 Аргументы и флаги mmap

Рассмотрим ключевые параметры mmap:

- addr: Желаемый стартовый адрес для отображения. Обычно передаётся nullptr, чтобы ОС сама выбрала подходящий адрес.
- length: Размер отображаемой области в байтах.
- prot (protection): Права доступа к памяти.
  - PROT_READ: память можно читать.
  - PROT_WRITE: в память можно писать.
  - PROT_EXEC: содержимое памяти можно исполнять как код.
  - PROT_NONE: к памяти нет доступа.
- flags: Определяют тип и поведение отображения.
  - MAP_SHARED или MAP_PRIVATE: Один из этих флагов обязателен. MAP_SHARED означает, что изменения, сделанные в памяти, будут видны другим процессам, отображающим тот же объект, и (в случае файла) будут записаны обратно в файл. MAP_PRIVATE создаёт copy-on-write отображение: изменения видны только текущему процессу и не затрагивают исходный файл.
  - MAP_ANONYMOUS: Создаёт анонимное отображение. Память инициализируется нулями и не связана ни с каким файлом. При использовании этого флага аргумент fd должен быть -1.
  - MAP_FIXED: Требует от ОС использовать точно адрес, указанный в addr. Это опасный флаг, так как он может без предупреждения перезаписать существующие отображения.
- fd, offset: Файловый дескриптор и смещение от начала файла для файловых отображений.

В случае успеха mmap возвращает указатель на начало выделенной области. В случае ошибки — MAP_FAILED.

3.2 Примеры использования mmap

3.2.1 Анонимное отображение

Это основной способ, которым аллокаторы (malloc, new) запрашивают большие блоки памяти у операционной системы.

```
#include <sys/mman.h>
#include <unistd.h> //  sysconf

// ...

//
size_t page_size = sysconf(_SC_PAGESIZE);
void* raw_mem = mmap(nullptr, page_size,
                     PROT_READ | PROT_WRITE,
                     MAP_PRIVATE | MAP_ANONYMOUS,
                     -1, 0);

```

```
if (raw_mem == MAP_FAILED) {
    //
}

char* data = static_cast<char*>(raw_mem);
//   data
data[0] = 'H';
data[1] = 'i';

//
munmap(raw_mem, page_size);
```

Листинг 5 – Выделение одной страницы памяти с помощью `mmap`

3.2.2 Отображение файла в память

Отображение файла позволяет работать с его содержимым как с обычным массивом в памяти, что может быть эффективнее, чем многократные вызовы `read` и `write`, особенно при произвольном доступе.

```
#include <sys/mman.h>
#include <fcntl.h>
#include <unistd.h>

const size_t FILE_SIZE = 128;
int fd = open("storage.bin", O_RDWR | O_CREAT, 0644);
ftruncate(fd, FILE_SIZE); //

void* raw_mem = mmap(nullptr, FILE_SIZE,
                     PROT_READ | PROT_WRITE,
                     MAP_SHARED, //
                     fd, 0);

close(fd); //  mmap

if (raw_mem == MAP_FAILED) { /* ... */ }

char* data = static_cast<char*>(raw_mem);
for (size_t i = 0; i < FILE_SIZE; ++i) {
    data[i] = static_cast<char>(i);
}

//
// (  )

munmap(raw_mem, FILE_SIZE);
```

Листинг 6 – Работа с файлом через `mmap`

3.3 Освобождение памяти: `munmap`

Вызов `munmap` удаляет отображение для указанного диапазона виртуальных адресов. Крайне важно освобождать память, выделенную через `mmap`, чтобы избежать утечек ресурсов. Аналогично паре `new`/`delete`, каждому успешному вызову `mmap` должен соответствовать вызов `munmap`.

4 Аргументы командной строки и переменные окружения

Кроме ввода-вывода, программа может получать информацию извне при запуске. Рассмотрим два основных механизма: аргументы командной строки и переменные окружения.

4.1 Аргументы командной строки

При запуске программы из терминала можно передать ей параметры. Они доступны в функции `main` через её аргументы.

```
int main(int argc, char* argv[]) {
    // ...
}
```

Листинг 7 – Интерфейс функции `main`

- `argc` (argument count): количество переданных аргументов.
- `argv` (argument vector): массив указателей на C-строки.

Важно помнить, что `argv[0]` — это всегда имя самой запущенной программы. Реальные аргументы начинаются с `argv[1]`. Например, для команды `./myprog hello world` будет:

- `argc` = 3
- `argv[0]` = `"./myprog"`
- `argv[1]` = `"hello"`
- `argv[2]` = `"world"`

4.2 Переменные окружения

Переменные окружения — это набор пар "ключ-значение которые наследуются дочерними процессами от родительских. Они используются для передачи контекста и настроек программам (например, `PATH` для поиска исполняемых файлов, `HOME` для пути к домашней директории).

В Linux переменные окружения физически располагаются в памяти процесса сразу после массива `argv`, отделённые от него указателем `nullptr`.

Для безопасного доступа к ним из C++ используется функция `getenv`.

```
#include <iostream>
#include <cstdlib> //  getenv

int main() {
    const char* user = std::getenv("USER");
    if (user != nullptr) {
        std::cout << "Hello, " << user << "!" << std::endl;
    } else {
        std::cout << "USER environment variable is not set." << std::endl;
    }
    return 0;
}
```

Листинг 8 – Чтение переменной окружения

Примечание

Переменные окружения часто используются для передачи конфиденциальной информации (ключей API, паролей), так как они, в отличие от аргументов командной строки, не видны другим пользователям системы через команды типа `ps`.

Итоги раздела

- Аргументы командной строки (`argc`, `argv`) позволяют передавать простые параметры при запуске.
- Переменные окружения — это наследуемые пары "ключ-значение"для передачи настроек и контекста.
- Для доступа к переменным окружения следует использовать `getenv`, что является более безопасным и портируемым способом.