Курс: *Архитектура компьютера и ОС*

Лекция 4: Работа с памятью и процессами

Лектор: Имя Фамилия

Дата: 05.10.2025

Содержание

1 Углублённая работа с памятью

На прошлой лекции мы познакомились с концепцией виртуальная память и системным вызовом `mmap`, который управляет отображением виртуальных адресов на физическая память. Однако модель, в которой `mmap` немедленно выделяет реальные физические страницы, является упрощением. На практике современные операционная система (ОС) используют более сложный и эффективный механизм.

1.1 Механизм Page Fault и ленивое выделение памяти

При попытке программы обратиться по виртуальному адресу, который не сопоставлен ни одной физической странице, процессор генерирует специальное прерывание.

Определение: Страничная ошибка (Page Fault)

страничная ошибка — это прерывание, которое генерируется аппаратно (процессором) при попытке доступа к странице виртуальная память, не имеющей корректного отображения в физическая память. При возникновении страничная ошибка исполнение текущего кода программы приостанавливается, и управление передаётся обработчику в ОС.

ОС анализирует причину страничная ошибка. Если обращение было к некорректному адресу (например, разыменование нулевого указателя), ОС принудительно завершает программу, как правило, с ошибкой **Segmentation Fault**.

Однако этот же механизм используется для реализации **ленивого выделения памяти** (постраничная подкачка по требованию). Когда программа вызывает `mmap`, ОС на самом деле не выделяет физические страницы. Она лишь запоминает, что данный диапазон виртуальных адресов теперь является валидным для процесса. Реальное выделение физической страницы происходит только при **первом обращении** к ней. Это обращение вызывает страничная ошибка, который ОС обрабатывает:

1. Находит свободную физическую страницу.
2. Устанавливает отображение между виртуальной страницей, вызвавшей прерывание, и новой физической страницей.
3. Возобновляет исполнение программы с прерванной инструкции.

Для программы этот процесс прозрачен, за исключением небольшой задержки.

Примечание

Плюсы и минусы ленивого выделения:

- **Плюс:** Эффективное использование ресурсов. Программы часто запрашивают больше памяти, чем реально используют. Ленивый подход позволяет системе выделять только ту физическую память, которая действительно нужна, и поддерживать так называемый *memory overcommitment* (когда суммарный объем запрошенной памяти превышает имеющуюся физическую).
- **Минус:** Усложнение обработки ошибок нехватки памяти. Вместо проверки кода возврата `mmap`, программа может быть внезапно "убита"ОС в произвольный момент при обращении к памяти, если свободные физические страницы закончились.
- **Минус:** Непредсказуемые задержки. Обращение к "новой"странице памяти вызывает страничная ошибка, что приводит к задержке, так как управление передается

ОС. Это может быть критично для приложений реального времени.

1.2 Структура таблиц страниц (Page Tables)

Для трансляции виртуальных адресов в физические ОС и процессор используют таблица страниц. Хранить простое линейное отображение для всего 64-битного адресного пространства (даже с учётом реальных ограничений современных процессоров в 256 ТБ) неэффективно.

В архитектуре x86-64 используется **четырехуровневая древовидная структура** таблиц страниц. Виртуальный адрес делится на несколько частей:

- **Смещение (offset):** Младшие 12 бит, указывающие на байт внутри страницы ($2^{12} = 4096$ байт).
- **Индексы в таблицах:** Четыре группы по 9 бит каждая, которые используются для последовательного обхода четырехуровневого дерева таблиц (L3, L2, L1, L0).

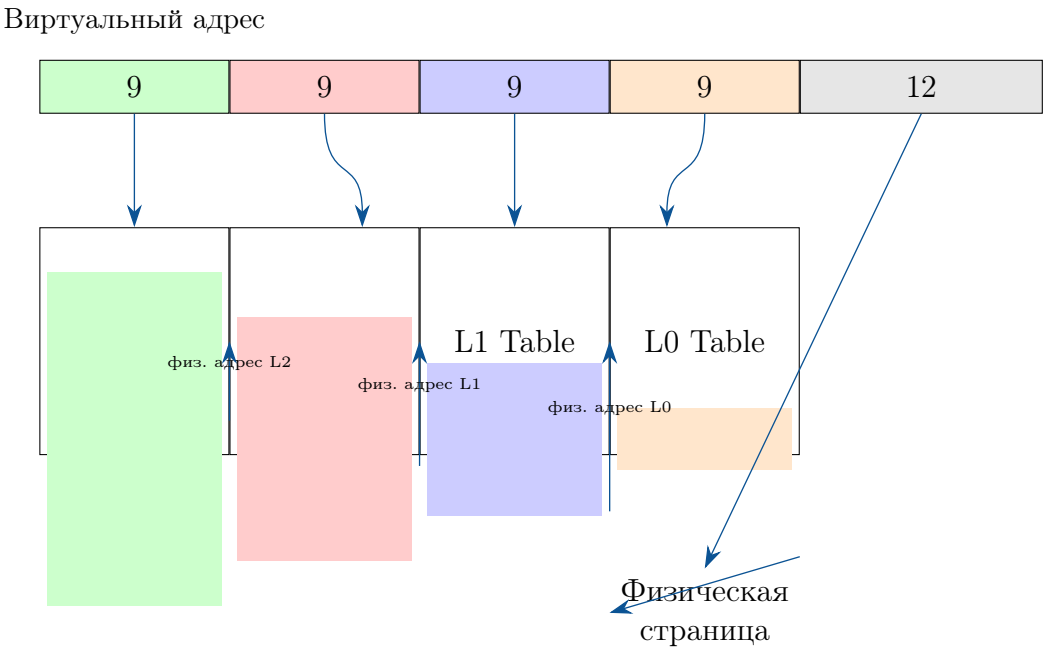

Рис. 1 – Трансляция виртуального адреса в физический через четырехуровневые таблицы страниц.

Процессор аппаратно выполняет обход этой структуры при каждом доступе к памяти: использует 9 бит адреса как индекс в таблице L3, находит там физический адрес таблицы L2, затем следующие 9 бит — как индекс в L2, и так далее, пока не дойдет до таблицы L0, где хранится адрес искомой физической страницы.

1.3 Дополнительные системные вызовы для работы с памятью

`mprotect(addr, size, prot)` изменяет права доступа (чтение, запись, исполнение) для уже выделенного диапазона виртуальной памяти `[addr, addr+size)`.

`mremap(old_addr, old_size, new_size, flags, ...)` позволяет изменять размер существующего отображения, а также перемещать его на новое место в виртуальном адресном пространстве.

`mlock(addr, size)` "закрепляет"указанный диапазон страниц в физической памяти,

запрещая ОС выгружать их в своп (swap). Это важно для приложений, работающих с чувствительными данными (пароли, ключи шифрования) или требующих предсказуемых задержек. `munlock` отменяет это действие.

Особый режим `mremap` позволяет создать "копию"участка памяти, где два разных диапазона виртуальных адресов указывают на **одни и те же физические страницы**. Любая запись в один диапазон немедленно видна в другом.

```cpp
// Allocate original mapping
void* from = mmap(nullptr, PAGE_SIZE, PROT_READ | PROT_WRITE,
                  MAP_ANONYMOUS | MAP_SHARED, -1, 0);

// Reserve space for the "copy"
void* to_placeholder = mmap(nullptr, PAGE_SIZE, PROT_NONE,
                            MAP_ANONYMOUS | MAP_PRIVATE, -1, 0);

// Create the shared mapping (remap `from` onto `to_placeholder`)
// MREMAP_FIXED tells mremap to use the address we provide.
// old_size = 0 is a special value for this copy operation.
void* to = mremap(from, 0 /* old_size */, PAGE_SIZE,
                  MREMAP_MAYMOVE | MREMAP_FIXED, to_placeholder);

// Now, `from` and `to` point to the same physical page.
volatile int* p_from = static_cast<volatile int*>(from);
volatile int* p_to = static_cast<volatile int*>(to);

*p_from = 123;
// Reading from p_to will now yield 123.
printf("Value at 'to': %d\n", *p_to); // Prints 123
```

Листинг 1 – Пример использования mremap для создания разделяемого отображения.

Примечание

В листинг 1 используется ключевое слово `volatile`. Оно сообщает компилятору, что значение в памяти, на которую указывает указатель, может измениться в любой момент без его ведома (например, через другой указатель, как в нашем случае). Это запрещает компилятору кэшировать значение переменной в регистре и заставляет его каждый раз честно читать значение из памяти, предотвращая неверные оптимизации.

1.4 Swap (своп) и его проблемы

Когда физическая память заканчивается, ОС может использовать своп (swap): выгрузить содержимое некоторых "неактивных"физических страниц на жесткий диск, чтобы освободить место для более актуальных данных. Когда программа обратится к такой выгруженной странице, произойдет страничная ошибка, и ОС загрузит её обратно с диска.

Проблемы свопинга:

- **Производительность:** Диск значительно медленнее оперативной памяти, что приводит к большим задержкам.
- **Безопасность:** Секретные данные (ключи, пароли) могут оказаться на диске в незашифрованном виде и остаться там даже после выключения питания, создавая уязви-

мость.

Итоги раздела

- Обращение к неотмеченной в таблица страниц странице вызывает страничная ошибка.
- ОС использует страничная ошибка для реализации ленивого выделения памяти, что экономит физическую память.
- Трансляция адресов в x86-64 реализована через многоуровневые таблицы страниц.
- Системные вызовы `mprotect`, `mremap`, `mlock` предоставляют тонкий контроль над отображениями памяти.
- своп (swap) помогает при нехватке памяти, но ценой производительности и потенциальных рисков безопасности.

2 Управление процессами

До сих пор мы рассматривали работу в рамках одного процесса. Теперь изучим, как создавать новые процессы и управлять ими.

Определение: Процесс

Процесс — это экземпляр запущенной программы. Каждый процесс является изолированной сущностью и обладает собственными ресурсами:

- Уникальным идентификатор процесса (Process ID) (PID).
- Отдельным виртуальным адресным пространством.
- Собственной таблицей файловых дескрипторов.

Процессы могут выполняться параллельно на многоядерных системах.

2.1 Подмена процесса: семейство `exec`

Системные вызовы семейства `exec` (`execlp`, `execvpe` и др.) **не создают** новый процесс. Они полностью **заменяют** текущий процесс новым, загружая и запуская указанный исполняемый файл.

```
#include <unistd.h>
#include <cstdio>

int main() {
    printf("Before exec...\n");

    // Replace the current process with "ls -l"
    // The first argument is the command,
    // subsequent args are for its argv.
    // The list must be terminated by a NULL pointer.
    execlp("ls", "ls", "-l", nullptr);

    // This line will never be reached if execlp succeeds.
    perror("execlp failed");
    return 1;
}
```

Листинг 2 – Запуск утилиты ls с помощью execlp.

При успешном вызове `execlp` код после него никогда не выполняется. Новый процесс (в данном случае, `ls`) наследует некоторые атрибуты старого, например, таблицу файловых дескрипторов, но получает новое адресное пространство.

Примечание

При запуске сторонних программ важно избегать утечки файловых дескрипторов. Если библиотека внутри вашего кода открыла файл, он останется открытым и в запущенном через `exec` процессе. Стандартное решение — открывать все файловые дескрипторы с флагом `O_CLOEXEC`, который предписывает ядру автоматически закрыть этот дескриптор при вызове `exec`.

2.2 Создание процесса: `fork`

Для создания нового процесса используется системный вызов fork.

Определение: Системный вызов fork

`fork()` создаёт точную копию текущего процесса. Уникальность fork в том, что он **вызывается один раз, а возвращается дважды**:

- В **родительском** процессе `fork()` возвращает PID нового (дочернего) процесса.
- В **дочернем** процессе `fork()` возвращает **0**.
- В случае ошибки возвращается -1.

Дочерний процесс является почти полной копией родителя: он получает копию адресного пространства, стека вызовов и таблицы файловых дескрипторов. Исполнение в обоих процессах продолжается с точки сразу после вызова `fork`.

```cpp
#include <unistd.h>
#include <sys/wait.h>
#include <cstdio>

int main() {
    pid_t child_pid = fork();

    if (child_pid == -1) {
        perror("fork failed");
        return 1;
    } else if (child_pid == 0) {
        // We are in the child process
        printf("I am the child! My PID is %d\n", getpid());
    } else {
        // We are in the parent process
        printf("I am the parent! My child's PID is %d\n", child_pid);
        wait(nullptr); // Wait for the child to finish
        printf("Parent knows child has finished.\n");
    }

    return 0;
}
```

Листинг 3 – Базовое использование fork.

2.3 Жизненный цикл процесса

2.3.1 Завершение процесса: `exit` vs `_Exit`

- `std::exit(code)` — функция стандартной библиотеки. Она не только завершает процесс с кодом `code`, но и выполняет ряд "очищающих"действий: сбрасывает буферы потоков ввода-вывода (например, `cout`), вызывает обработчики, зарегистрированные через `atexit`, и т.д..
- `std::_Exit(code)` (или системный вызов `_exit`) — немедленно завершает процесс без какой-либо очистки. В дочерних процессах после `fork` предпочтительнее использовать именно `_Exit`, чтобы избежать нежелательных побочных эффектов, например, двойного сброса буферов, которые были скопированы от родителя.

2.3.2 Ожидание дочерних процессов: `wait` и `waitpid`

Родительский процесс обязан "собирать"информацию о завершении своих дочерних процессов с помощью `wait()` или `waitpid()`. Эти вызовы блокируют родителя до тех пор, пока один из его детей не завершится, и позволяют получить его код завершения.

Определение: Процесс-зомби

процесс-зомби — это процесс, который уже завершил своё выполнение, но запись о нём (PID, код завершения) всё ещё остаётся в таблице процессов ядра. Он находится в этом состоянии до тех пор, пока родитель не "прочитает"его статус с помощью `wait`. Если родитель не делает `wait`, зомби накапливаются и "утекают"системные ресурсы (в частности, PID).

Определение: Процесс-сирота

процесс-сирота — это процесс, родитель которого завершился раньше него. Такие процессы не остаются "бесхозными"— их "усыновляет"специальный системный процесс `init` (с PID 1), который периодически вызывает `wait` и очищает зомби.

2.4 Паттерн `fork-exec`

Комбинация `fork` и `exec` — это стандартный способ в Unix-системах запустить новую программу, не прекращая работу текущей.

1. Родительский процесс вызывает `fork()`, создавая свою копию.
2. В дочернем процессе (где `fork()` вернул 0) выполняются необходимые настройки (например, перенаправление ввода-вывода с помощью `dup2`).
3. Дочерний процесс вызывает один из вызовов семейства `exec`, заменяя себя новой программой.
4. Родительский процесс (где `fork()` вернул PID > 0) может продолжить свою работу или дождаться завершения дочернего с помощью `waitpid()`.

Итоги раздела

- Процесс — это изолированный экземпляр запущенной программы.
- exec **заменяет** текущий процесс, fork **создаёт** его копию.
- Паттерн `fork-exec` является основой для запуска программ в Unix-подобных системах.
- Родитель **обязан** дожидаться завершения дочерних процессов с помощью `wait` или `waitpid`, чтобы избежать появления процесс-зомби.

3 Межпроцессное взаимодействие: Pipelines

Одним из самых мощных механизмов в Unix является канал (pipe), который позволяет связать стандартный вывод одного процесса со стандартным вводом другого. Рассмотрим, как реализовать аналог команды `ps aux | grep zsh` программно.

Для этого нам понадобится системный вызов `pipe()`, который создаёт однонаправленный канал данных и возвращает два файловых дескриптора: `pipefd[0]` для чтения и `pipefd[1]` для записи.

Алгоритм реализации пайплайна `cmd1 | cmd2`:

1. Создать канал (pipe) с помощью `pipe(pipefd)`.
2. Вызвать `fork()` для создания первого дочернего процесса (`child1` для `cmd1`).
3. **В `child1`:**
   - Закрыть ненужный конец канала: `close(pipefd[0])`.
   - Перенаправить стандартный вывод на пишущий конец канала: `dup2(pipefd[1], STDOUT_FILENO)`.
   - Закрыть оригинальный дескриптор: `close(pipefd[1])`.
   - Вызвать `exec` для запуска `cmd1`.
4. Вызвать `fork()` для создания второго дочернего процесса (`child2` для `cmd2`).
5. **В `child2`:**
   - Закрыть ненужный конец канала: `close(pipefd[1])`.
   - Перенаправить стандартный ввод на читающий конец канала: `dup2(pipefd[0], STDIN_FILENO)`.
   - Закрыть оригинальный дескриптор: `close(pipefd[0])`.
   - Вызвать `exec` для запуска `cmd2`.
6. **В родительском процессе:**
   - **Критически важно:** закрыть **оба** конца канала: `close(pipefd[0])` и `close(pipefd[1])`. Если этого не сделать, читающий процесс никогда не получит EOF и зависнет.
   - Дождаться завершения обоих дочерних процессов с помощью `waitpid()`.

```
#include <unistd.h>
#include <sys/wait.h>
#include <cstdio>
#include <cstdlib>

int main() {
    int pipefd[2];
    if (pipe(pipefd) == -1) {
        perror("pipe failed");
        return 1;
    }

    pid_t child1 = fork();
    if (child1 == 0) { // First child: ps aux
        close(pipefd[0]); // Close read end
```

```
        dup2(pipefd[1], STDOUT_FILENO);
        close(pipefd[1]); // Close original write end
        execlp("ps", "ps", "aux", nullptr);
        _exit(1); // exit if exec fails
    }

    pid_t child2 = fork();
    if (child2 == 0) { // Second child: grep zsh
        close(pipefd[1]); // Close write end
        dup2(pipefd[0], STDIN_FILENO);
        close(pipefd[0]); // Close original read end
        execlp("grep", "grep", "zsh", nullptr);
        _exit(1); // exit if exec fails
    }

    // Parent process
    close(pipefd[0]); // ESSENTIAL: close both pipe ends in parent
    close(pipefd[1]);

    waitpid(child1, nullptr, 0);
    waitpid(child2, nullptr, 0);

    return 0;
}
```

Листинг 4 – Программная реализация пайплайна ps aux | grep zsh.

Примечание

Что происходит, если читатель завершается раньше писателя? (например, в `ps aux | head -n 5`) Когда все читающие концы канала закрываются, а писатель пытается в него что-то записать, ОС посылает писателю сигнал `SIGPIPE`. По умолчанию, действие для этого сигнала — аварийное завершение процесса. Это элегантно решает проблему "бесконечной"работы процессов в начале пайплайна, если их вывод больше никому не нужен.

Итоги раздела

- канал (pipe) создаёт однонаправленный канал для данных между процессами.
- Комбинация `pipe`, `fork`, `dup2` и `exec` позволяет строить сложные конвейеры обработки данных.
- В родительском процессе необходимо закрывать оба конца канала, чтобы избежать взаимоблокировок.
- Сигнал `SIGPIPE` автоматически завершает процессы, которые пытаются писать в "сломанный"канал (без читателей).