Лекция 16
Файловая система

Файловая система

- Формат хранения данных на носителе (ф.с. FAT32, NTFS, EXT4)
  - Каждая ф.с. имеет свои особенности: максимальный размер, ограничения на размеры файлов, длину имени и т. п.
- Компонент ядра ОС, отвечающий за доступ к файлам
  - API для работы с файлами и файловой системой
  - Драйвера конкретных файловых систем

Требования к файловым системам

- Постоянство (сохранение данных после окончания процесса и после остановки ОС)
- Поддержка данных огромного размера (одного файла и суммарного)
- Эффективность (скорость поиска файла и обращения к файлу)
- Поддержка разделения прав доступа и квотирования
- Устойчивость к программному сбою
- Устойчивость к аппаратному сбою

Задачи ядра ОС

- Предоставление стандартного интерфейса
  - POSIX API для работы с файлами и файловой системой: user space ↔ kernel space
  - VFS (virtual file system): kernel ↔ FS driver
- Управление ресурсами
  - Разграничение прав доступа к объектам ФС
  - Квотирование ресурсов ФС
  - Арбитраж параллельного доступа к ФС (блокировки, атомарность)

Основные абстракции

- Файл — именованный набор данных (для Unix-систем — регулярный файл)
- Или файл — запись в каталоге. В Unix-системах:
  - Регулярные файлы
  - Файлы-каталоги
  - Файлы-устройства (блочные и символьные)
  - Символические ссылки
  - Именованные каналы (FIFO)
  - Сокеты

Регулярный файл

- Регулярный файл содержит данные, находящиеся на устройстве (в блоках данных)
- Представляет собой поток байт
- Структурирование, поддержание корректности – задача программ пользовательского режима
- В некоторых случаях ядро требует файлы определенной структуры (например, загрузка ELF-файла на выполнение)

Идентификация файлов

- В загруженной и работающей системе – пути к файлам
- В файловой системе на уровне ядра ОС – номер индексного дескриптора
- Открытый файл на уровне процесса – файловый дескриптор

Файловый дескриптор

- Файловый дескриптор — идентификатор открытого файла в процессе
- Каждый процесс имеет свой набор файловых дескрипторов, независимый от других процессов
- Неотрицательное целое число
- Обычно при старте процесса:
  - 0 — stdin
  - 1 — stdout
  - 2 — stderr
- При выделении нового ф. д. всегда выбирается свободный с минимальным номером
- ф. д. - индекс в таблицу открытых файлов процесса

Открытие файла

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int open(const char *path, int flags, int mode);
```

- Возвращает файловый дескриптор при успехе и -1 при ошибке
- При ошибке код ошибки в errno (<errno.h>)
- path — путь к открываемому файлу

Флаги открытия

- Основные режимы открытия:
  - O_RDONLY — только чтение
  - O_WRONLY — только запись
  - O_RDWR — чтение-запись
- Флаги управления файловым дескриптором
  - O_CLOEXEC — файловый дескриптор закрывается автоматически при exec
- Модификаторы режима записи
  - O_APPEND — режим добавления в конец файла
  - O_TRUNC — очистить файл

Флаги открытия

- Модификаторы создания файла
  - O_CREAT — создание файла
  - O_EXCL — создание файла только в случае, если он еще не существует
- Типичные комбинации флагов
  - O_RDONLY
  - O_WRONLY | O_CREAT | O_TRUNC
  - O_WRONLY | O_CREAT | O_APPEND

Режим создания файла

- При указании флага O_CREAT используется параметр mode — права доступа на создаваемый файл
- Права доступа накладываются на параметр umask: mode & ~umask
- Например, mode == 0666, umask == 0022, права на создаваемый файл 0644
- Mode == 0700, umask == 0007, права 0700

umask

- Атрибут процесса
- Указывает, какие биты прав доступа должны быть сброшены в задаваемых правах (9 основных бит)
- Могут быть получены/изменены с помощью системного вызова

```
int umask(int newmask);
```

- Возвращается старое значение umask

Закрытие файла

int close(int fd);

- При успехе возвращается 0, при неудаче - -1.
- Причины неудачи:
  - EBADF — неправильный файловый дескриптор
  - EINTR — операция была прервана
  - EIO — ошибка записи
- В любом случае, ничего разумного при ошибке сделать нельзя!

Синхронизация с диском

int fsync(int fd);

- Для избежания потерь данных сохранение данных на диск не должно выполняться при закрытии
- В случае ошибки EIO вызова fsync ф. д. fd не закрыт и есть возможность ситуацию исправить

Позиционирование в файле

- Если открытый файл является файлом произвольного доступа, текущую позицию в файле (f_pos) можно произвольно изменять

off_t lseek(int fd, off_t offset, int whence);

- Whence:
  - SEEK_SET — относительно начала файла
  - SEEK_END — относительно конца файла
  - SEEK_CUR — относительно текущей позиции
- Возвращается новое положение в файле относительно начала или -1 в случае ошибки

Позиционирование в файле

- Для каналов, сокетов, символьных устройств, псевдотерминалов позиционирование невозможно!
- Позиционирование на позицию до начала файла невозможно (EINVAL)
- В файле, открытом O_RDONLY, позиционирование после текущего конца невозможно (EINVAL)
- В файле, открытом O_WRONLY или O_RDWR, позиционирование после текущего конца файла допускается. Последующая операция записи заполняет пропуск между старым концом файла и текущей позицией нулевыми байтами

32-битные системы

- off_t — знаковый, 32-битный, то есть lseek не может работать с файлами > 2G
- Чтобы работать с большими файлами:
  - -D_FILE_OFFSET_BITS=64 в командной строке gcc
  - Все смещения будут 64-битными знаковыми

Установка размера файла

```
int truncate(const char *path, off_t length);
int ftruncate(int fd, off_t length);
```

Эти системные вызовы позволяют задать новый размер файла (length). Он может быть как больше (файл дополняется нулевыми байтами), так и меньше текущего.

Чтение

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

- size_t — **беззнаковый** целый тип размера, достаточного для хранения размера любого объекта в C/C++ (обычно на Unix это unsigned long)
- ssize_t — **знаковый** тип такого же размера, как и size_t
- Если count > SSIZE_MAX, поведение read не определено
- На 32-битных системах count < 2G

Чтение

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

- Возвращает -1 при ошибке (EIO, EGAIN, …) - см. man 2 read
- Если count == 0, то выполняется проверка на ошибки и возвращается либо 0, либо -1

Чтение

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

- Обычный случай: count > 0, успешное завершение (возвр. значение >= 0)
- 0 — признак конца файла (то есть данных больше нет и не будет)
- Иначе не более чем count байт считано в буфер buf и количество байт возвращено

Чтение

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

- Если готовых к чтению данных нет, read переведет процесс в состояние ожидания до появления данных
  - Но в режиме O_NONBLOCK read вернет EAGAIN немедленно!
- Если есть хоть один байт готовых к чтению данных read возвращает их немедленно
- Никогда не ждет полного заполнения буфера до размера count

Чтение

- Как правило, при работе с регулярными файлами на обычных файловых системах, если нет данных доступных немедленно, процесс переводится в состояние «uninterruptible sleep» (D-state) до получения данных
- Как правило, при этом возвращается столько данных, сколько запрошено
- НО ПОЛАГАТЬСЯ НА ЭТО НЕЛЬЗЯ!

Запись

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

- Возвращает -1 при ошибке
- Если count > SSIZE_MAX, поведение read не определено
- Если count == 0 и файл регулярный, выполняется проверка на ошибки и возвращается либо 0, либо -1
- Если count > 0, возвращается количество записанных байт

Запись

- Как правило, при работе с регулярными файлами на обычных файловых системах write записывает все за один раз и возвращает count
- НО ПОЛАГАТЬСЯ НА ЭТО НЕЛЬЗЯ!