Лекция 4
Файловая система

Напоминание

- Материалы семинарских занятий на github — выложены подготовительные материалы к семинару 4
- Проверка стиля кодирования при сдаче программ:
  - Символ TAB не допускается (настраивайте свой редактор)
  - Отступы — 4 пробела.
  - В редакторе vim:
    - Set tabstop=4
    - Set expandtab
  - В редакторе nano
    - Nano -E

Создание файла

int fd = open(path, O_CREAT | …, 0666/* mode*/);

- При указании флага O_CREAT используется параметр mode — права доступа на создаваемый файл (только 9 основных бит)
- 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 файлового дескриптора

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

Копирование ф. д.

```
int dup(int oldfd); // берем первый свободный
int dup2(int oldfd, int newfd);
int dup3(int oldfd, int newfd, int flags);
```

- Если newfd был открыт, он закрывается, ошибки игнорируются
- Новый ф. д. разделяет доступ к файлу со старым файловым дескриптором
- Флаг O_CLOEXEC сбрасывается (для dup, dup2), либо может быть задан явно (dup3)

Разделение открытого файла

- Все ф. д. - копии разделяют (имеют общую) следующую информацию:
  - Режим открытия файла
  - Текущую позицию в файле
- Открытый файл закрывается, когда закрывается последний ф. д.-копия
- Каждый ф. д. - копия имеет свое значение флага O_CLOEXEC

struct file

- Состояние открытого файла
- Находится в исходном коде ядра Linux в include/linux/fs.h

```
struct file
{
    atomic_long_t  f_count; // счетчик ссылок
    unsigned int   f_flags; // флаги open
    fmode_t        f_mode;  // внутр. флаги
    loff_t         f_pos;   // текущее смещение
    // много всего еще
};
```

Подсчет ссылок

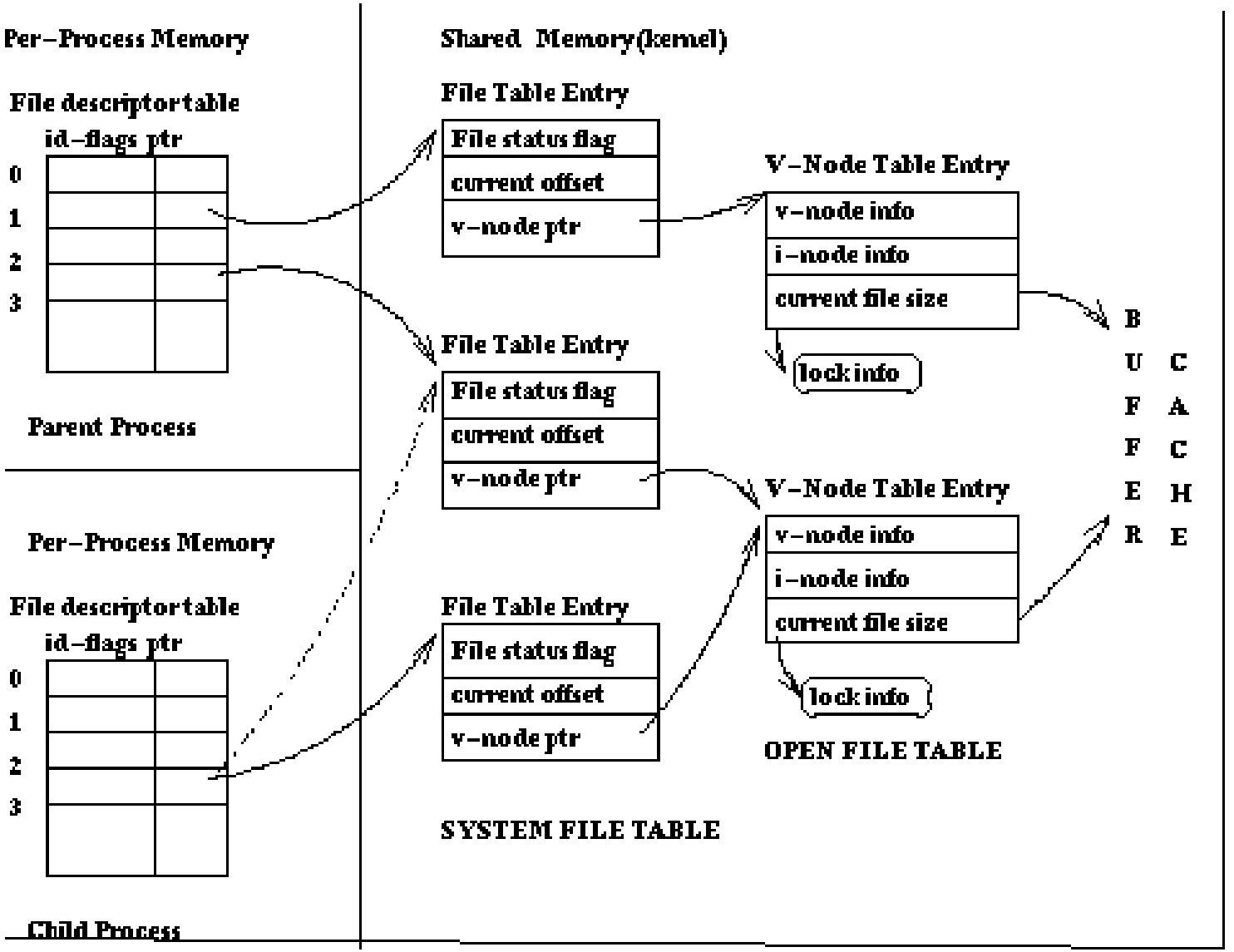
- При выполнении open (создание первого ф. д.): f_count = 1
- При копировании ф. д. (dup* или fork):
  ++f_count
- При закрытии ф. д. (close)
  ```
  if (--f_count == 0) {
    // реально закрыть файл:
    // сохранить несохраненные данные,
    // освободить ресурсы ядра
  }
  ```
- Подсчет ссылок — эффективный способ управления ресурсами в случае ациклических графов

Таблица файловых дескрипторов

- Хранится для каждого процесса
- Находится в include/linux/fdtable.h

```
struct fdtable
{
    unsigned int max_fds;
    struct file **fd;                  // file pointer
    unsigned long *close_on_exec; //CLOEXEC bitset
    unsigned long *open_fds;       // opened bitset
};
```

Структуры ядра



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 дописывает в конец файла необходимое количество нулей

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);
```

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!

Атомарность

- Атомарность (относительно класса наблюдателей) — свойство операции
- Наблюдатель не может «поймать момент», когда атомарная операция будет в середине выполнения в промежуточном состоянии
- Для наблюдателя атомарная операция или не началась, или уже закончилась

Не атомарность 1

- Процесс 1

```
fd=open("A", O_RDONLY, 0);
if(fd<0 && errno=ENOENT) {
  fd=open("A", O_WRONLY
  |O_CREAT, 0600);
}
```

- Между операцией проверки на существование и создания файла может вклиниться другой процесс и создать свой файл

- Процесс 2

```
fd=open("A", O_CREAT|
O_WRONLY|O_TRUNC, 0666);
```

- В старых Unix это был один из способов для получения прав root в случае ошибки в привилегированном ПО

Атомарность

int fd = open("A", O_CREAT|O_EXCL|O_WRONLY|O_TRUNC, 0600);

- Возвращает -1 (errno == EEXIST) если файл уже существует
- Если файл не существует создает его
- Эти две операции выполняются атомарно, т. е. другой процесс не может вклиниться между проверкой на существование и созданием

Одновременная работа с файлами

- В Unix если одновременно несколько процессов работают с копиями одного и того же файлового дескриптора или с одним и тем же файлом, и операции чтения, и операции записи разрешены без ограничений
- Процессы должны сами согласовать свое поведение, чтобы избежать порчи данных
- Варианты: флаг O_APPEND, рекомендательные блокировки, обязательные блокировки

Атомарность чтения/записи

- При работе с каналами и сокетами если count < PIPE_BUF (на Linux — 4KiB), операции чтения и записи атомарны, т. е. Данные не перемешиваются и записываются последовательно

Чтение/запись с каналами

- Процесс 1

  write(fd, "123\n", 4);

- Процесс 2

  write(fd, "456\n", 4);

- Два возможных результата:

  123
  456

- Или

  456
  123

Атомарность с файлами

- POSIX не гарантирует атомарности чтения/записи при работе с файлами
- Реально Linux записывает/считывает данные небольшого (зависит от типа ФС, около 1KiB) размера атомарно, то есть при записи данные двух процессов не перемешаются
- **НО! Запись/чтение данных и изменение значения текущей позиции в совокупности не атомарны!**

Чтение/запись с файлами

- Процесс 1

  write(fd, "123\n", 4);

- Процесс 2

  write(fd, "456\n", 4);

- Четыре возможных результата:

  123
  456

- Или

  456
  123

- Или

  123

- Или

  456

Флаг O_APPEND

- Если при открытии файла задан флаг O_APPEND,
- При каждой записи в файл указатель текущей позиции сначала перемещается в конец файла
- Затем выполняется запись в файл
- Эти два действия - атомарны

Чтение/запись с O_APPEND

- Процесс 1

  write(fd, "123\n", 4);

- Процесс 2

  write(fd, "456\n", 4);

- Два возможных результата:

  123
  456

- Или

  456
  123