Лекция 29
Замещение тела процесса

Замещение тела процесса

- Замещение тела процесса — запуск на выполнение другого исполняемого файла в рамках текущего процесса
- Для замещения тела процесса используется семейство exec*: сист. вызов execve и функции execv, execvp, execl, execlp, execle
  - «v» - передается массив параметров
  - «l» - передается переменное число параметров
  - «e» - передается окружение
  - «p» - выполняется поиск по PATH

Системный вызов execve

```
int execve(const char *path, char *const argv[],
           char *const envp[]);
```

- path — путь к исполняемому файлу
- argv — массив аргументов командной строки, заканчивается элементом NULL
- envp — массив переменных окружения, заканчивается элементом NULL
- Аргументы командной строки и переменные окружения помещаются на стек процесса
- При успехе системный вызов не возвращается

execve

- argv[0] обычно совпадает с path (но не обязано)
- Переменная 'environ' — текущее окружение процесса
- 'environ' должна быть объявлена явно, если требуется
- «Shebang» конструкции (#! /bin/python3) обрабатываются ядром

Сохранение атрибутов процесса

- Сохраняются все атрибуты, **за исключением**
  - Атрибутов, связанных с адресным пространством процесса
  - Сигналов, ожидающие доставки
  - Таймеров

Функция execlp

int execlp(const char *file, const char *arg, ...);

- Выполняется поиск исполняемого файла file по каталогам, перечисленным в переменной окружения PATH
- Аргументы запускаемого процесса передаются в качестве параметров функции execlp
- Последним аргументом функции должен быть NULL

Схема fork/exec

- Для запуска программ в отдельных процессах применяется комбинация fork/exec

- Системный вызов fork создает новый процесс
- В сыновнем процессе системными вызовами настраиваются параметры процесса (например, текущий рабочий каталог, перенаправления стандартных потоков и пр.)
- Вызовом exec* запускается требуемый исполняемый файл

Копирование файловых дескрипторов

- Функции семейства dup копируют ф. д.

```
int dup(int oldfd); // новый ф.д. неявный
int dup2(int oldfd, int newfd);
// Linux-only
int dup3(int oldfd, int newfd, int flags);
```

- У нового ф. д. сбрасывается O_CLOEXEC (при fork() O_CLOEXEC не сбрасывается!)
- Если новый ф.д. был открыт до dup, он предварительно будет закрыт
- И старый, и новый ф.д. ссылаются на одну и ту же struct file

Пример

```
int main(void)
{
  int pid, status, fd;
  pid = fork();
  if (!pid) {
    chdir("/usr/bin");
    fd = open("/tmp/log.txt", O_WRONLY|O_CREAT|O_TRUNC, 0600);
    dup2(fd, 1); close(fd);
    execlp("/bin/ls", "/bin/ls", "-l", NULL);
    fprintf(stderr, "Exec failed\n");
    _exit(1);
  }
  wait(&status);
  return 0;
}
```

Подготовка аргументов командной строки

- Часто необходимо запустить программу, если передана строка состоящая из имени программы и аргументов

int system(const char *command);

Например: res = system("ls -l");

Реализация с помощью execl:
execl("/bin/sh", "sh", "-c", command, NULL);

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
- **В современных версиях Linux (~2014): запись/чтение данных и изменение значения текущей позиции в совокупности атомарны (ранее нет)**

There are almost zero interesting applications that are "standard conforming", so if we used that as an excuse to not fix kernel issues, we'd never have to do any work. And we'd have no users ;)

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