МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
федеральное государственное автономное образовательное учреждение
высшего профессионального образования
«Северный (Арктический) федеральный университет имени М.В. Ломоносова»

Институт математики и компьютерных наук

Рудалёв А.В.

Операционные системы

г. Архангельск
2012 г.

Версия документа

Версия	Дата	Комментарий
0.1	01.02.2012	План
0.2	07.02.12	Лабораторная

Оглавление

Организация взаимодействия процессов через pipe и FIFO в UNIX

Вопрос «умеет ли компьютер думать» имеет не больше смысла,
чем вопрос «умеет ли подводная лодка плавать».
Эдсгер Вибе Дейкстра

В лабораторной работе рассматриваются процессы в ОС UNIX, получение сведений о них, через системные вызовы.

Теория

В качестве теоретической базы по процессам в Unix можно почитать:

- К.А. Коньков, В.Е. Карпов, «Основы операционных систем» (http://www.intuit.ru/department/os/osintro/).
- К.А. Коньков, В.Е. Карпов, «Основы операционных систем. Практикум» (http://www.intuit.ru/department/os/osintropractice/).
- А. Боровский, Linux API – «Введение в межпроцессное взаимодействие» (http://citforum.ru/programming/unix/ipc_intro/)

Практическая работа

На основе 3-й практической работы курса К.А. Коньков, В.Е. Карпов, «Основы операционных систем. Практикум» (http://www.intuit.ru/department/os/osintropractice/).

Работа с файлами

В операционной системе UNIX можно упрощенно полагать, что информация о файлах, с которыми процесс осуществляет операции потокового обмена, наряду с информацией о потоковых линиях связи, соединяющих процесс с другими процессами и устройствами ввода-вывода, хранится в некотором массиве, получившем название таблицы открытых файлов или таблицы файловых дескрипторов. Индекс элемента этого массива, соответствующий определенному потоку ввода-вывода, получил название файлового дескриптора для этого потока. Таким образом, файловый дескриптор представляет собой небольшое целое неотрицательное число, которое для текущего процесса в данный момент времени однозначно определяет некоторый действующий канал ввода-вывода. Некоторые файловые дескрипторы на этапе старта любой программы ассоциируются со стандартными потоками ввода-вывода. Так, например, файловый дескриптор 0 соответствует стандартному потоку ввода (stdin), файловый дескриптор 1 – стандартному потоку вывода (stdout), файловый дескриптор 2 – стандартному потоку для вывода ошибок (stderr). В нормальном интерактивном режиме работы стандартный поток ввода связывает процесс с клавиатурой, а стандартные потоки вывода и вывода ошибок – с текущим терминалом.

Для совершения потоковых операций чтения/записи информации из потоков ввода/вывода применяются системные вызовы read() и write().

Прототипы системных вызовов read и write:

```
#include <sys/types.h>
#include <unistd.h>
size_t read(int fd, void *addr, size_t nbytes);
size_t write(int fd, void *addr, size_t nbytes);
```

Пример использования:

```
#include <sys/types.h>
#include <fcntl.h>
#include <stdio.h>
int main(){
    int fd;
    size_t size;
    char string[] = "Hello, world!";
    char filename[] = "myfile.txt";
    /* Обнуляем маску создания файлов текущего процесса для того,
    чтобы права доступа у создаваемого файла точно соответствовали
    параметру вызова open() */
    (void)umask(0);
    /* Попытаемся открыть файл с именем myfile в текущей директории
     только для операций вывода. Если файла не существует, попробуем
     его создать с правами доступа 0666, т. е. read-write для всех
     категорий пользователей */
    if((fd = open(filename, O_WRONLY | O_CREAT,
        0666)) < 0){
        /* Если файл открыть не удалось, печатаем об этом сообщение
        и прекращаем работу */
        printf("Can\'t open file\n");
        exit(-1);
    }
    /* Пробуем записать в файл 14 байт из нашего массива, т.е. всю
    строку "Hello, world!" вместе с признаком конца строки */
    size = write(fd, string, 14);
    if(size != 14){
        /* Если записалось меньшее количество байт, сообщаем об
        ошибке */
        printf("Can\'t write all string\n");
        exit(-1);
    }
    /* Закрываем файл */
    if(close(fd) < 0){
        printf("Can\'t close file\n");
    }
    return 0;
}
```

1. Создайте хранилище Меркуриал с именем os-03-pipe.
2. Напишите программу file-write.c на C создающую файл «myfile.txt» и заносящую в него текст «Hello, world».
3. Зафиксируйте изменения с комментарием «Example write file».
4. Напишите программу file-read.c на C считывающую файл «myfile.txt» и выводящую её на экран.
5. Зафиксируйте изменения с комментарием «Example read file».

Pipe

Наиболее простым способом для передачи информации с помощью потоковой модели между различными процессами или даже внутри одного процесса в операционной системе UNIX является pipe (канал, труба, конвейер).

Важное отличие pip'а от файла заключается в том, что прочитанная информация немедленно удаляется из него и не может быть прочитана повторно.

Pipe можно представить себе в виде трубы ограниченной емкости, расположенной внутри адресного пространства операционной системы, доступ к входному и выходному отверстию которой осуществляется с помощью системных вызовов. В действительности pipe представляет собой область памяти, недоступную пользовательским процессам напрямую, зачастую организованную в виде кольцевого буфера (хотя существуют и другие виды организации). По буферу при операциях чтения и записи перемещаются два указателя, соответствующие входному и выходному потокам. При этом выходной указатель никогда не может перегнать входной и наоборот. Для создания нового экземпляра такого кольцевого буфера внутри операционной системы используется системный вызов pipe()

Прототип системного вызова pipe():

```
#include <unistd.h>
int pipe(int *fd);
```

Пример программы для pipe в одном процессе:

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
int main(){
    int fd[2];
    size_t size;
    char string[] = "Hello, world!";
    char resstring[14];
    /* Попытаемся создать pipe */
    if(pipe(fd) < 0){
        /* Если создать pipe не удалось, печатаем об этом сообщение
        и прекращаем работу */
        printf("Can\'t create pipe\n");
        exit(-1);
    }
    /* Пробуем записать в pipe 14 байт из нашего массива, т.е. всю
    строку "Hello, world!" вместе с признаком конца строки */
    size = write(fd[1], string, 14);
    if(size != 14){
        /* Если записалось меньшее количество байт, сообщаем об
        ошибке */
        printf("Can\'t write all string\n");
        exit(-1);
    }
    /* Пробуем прочитать из pip'а 14 байт в другой массив, т.е. всю
    записанную строку */
    size = read(fd[0], resstring, 14);
    if(size < 0){
        /* Если прочитать не смогли, сообщаем об ошибке */
        printf("Can\'t read string\n");
        exit(-1);
    }
    /* Печатаем прочитанную строку */
    printf("%s\n",resstring);
```

```
    /* Закрываем входной поток*/
    if(close(fd[0]) < 0){
        printf("Can\'t close input stream\n");
    }
    /* Закрываем выходной поток*/
    if(close(fd[1]) < 0){
        printf("Can\'t close output stream\n");
    }
    return 0;
}
```

6. Напишите программу pipe.c свызовом pipe в одном процессе.
7. Зафиксируйте изменения с комментарием «Simple pipe».
8. Измените программу pipe.c, так что бы основной поток разделился на два с помощью fork(), и родительский поток отправил дочернему сообщение «Hello, world!».
9. Зафиксируйте изменения с комментарием «Forked pipe».

Pipe служит для организации однонаправленной или симплексной связи. Если бы в предыдущем примере мы попытались организовать через pipe двустороннюю связь, когда процесс-родитель пишет информацию в pipe, предполагая, что ее получит процесс-ребенок, а затем читает информацию из pip’a, предполагая, что ее записал порожденный процесс, то могла бы возникнуть ситуация, в которой процесс-родитель прочитал бы собственную информацию, а процесс-ребенок не получил бы ничего. Для использования одного pip’a в двух направлениях необходимы специальные средства синхронизации процессов, о которые будут рассмотрены на следующей лабораторной работе. Более простой способ организации двунаправленной связи между родственными процессами заключается в использовании двух pipe.

10. Измените программу pipe.c, так что бы основной поток разделился на два с помощью fork(), а родительский и дочерний потоки обменялись своими pid через pip-ы.
11. Зафиксируйте изменения с комментарием «2 pipes».

Именованные pipe (FIFO)

Как мы выяснили, доступ к информации о расположении pip'a в операционной системе и его состоянии может быть осуществлен только через таблицу открытых файлов процесса, создавшего pipe, и через унаследованные от него таблицы открытых файлов процессов-потомков. Поэтому изложенный выше механизм обмена информацией через pipe справедлив лишь для родственных процессов, имеющих общего прародителя, инициировавшего системный вызов pipe(), или для таких процессов и самого прародителя и не может использоваться для потокового общения с другими процессами. В операционной системе UNIX существует возможность использования pip'a для взаимодействия других процессов, но ее реализация достаточно сложна и лежит далеко за пределами наших занятий.

Для организации потокового взаимодействия любых процессов в операционной системе UNIX применяется средство связи, получившее название **FIFO** (от First Input First Output) или **именованный pipe**. FIFO во всем подобен pip’у, за одним исключением: данные о расположении FIFO в адресном пространстве ядра и его состоянии процессы могут получать не через родственные связи, а через файловую систему. Для этого при создании именованного pip’a на диске заводится файл специального типа, обращаясь к которому процессы могут получить интересующую их информацию. Для создания FIFO используется системный вызов mknod() или существующая в некоторых версиях UNIX функция mkfifo().

Следует отметить, что при их работе не происходит действительного выделения области

адресного пространства операционной системы под именованный pipe, а только заводится файл-метка, существование которой позволяет осуществить реальную организацию FIFO в памяти при его открытии с помощью уже известного нам системного вызова open().

После открытия именованный pipe ведет себя точно так же, как и неименованный. Для дальнейшей работы с ним применяются системные вызовы read(), write() и close(). Время существования FIFO в адресном пространстве ядра операционной системы, как и в случае с pip'ом, не может превышать время жизни последнего из использовавших его процессов. Когда все процессы, работающие с FIFO, закрывают все файловые дескрипторы, ассоциированные с ним, система освобождает ресурсы, выделенные под FIFO. Вся непрочитанная информация теряется. В то же время файл-метка остается на диске и может использоваться для новой реальной организации FIFO в дальнейшем.

Прототип системного вызова mknod():

```
#include <sys/stat.h>
#include <unistd.h>
int mknod(char *path, int mode, int dev);
```

Пример программы для работы с FIFO:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
int main(){
    int fd, result;
    size_t size;
    char resstring[14];
    char name[]="aaa.fifo";
    /* Обнуляем маску создания файлов текущего процесса для того,
    чтобы права доступа у создаваемого FIFO точно соответствовали
    параметру вызова mknod() */
    (void)umask(0);
    /* Попытаемся создать FIFO с именем aaa.fifo в текущей
    директории */
    if(mknod(name, S_IFIFO | 0666, 0) < 0){
        /* Если создать FIFO не удалось, печатаем об этом
        сообщение и прекращаем работу */
        printf("Can\'t create FIFO\n");
        exit(-1);
    }
    /* Порождаем новый процесс */
    if((result = fork()) < 0){
        /* Если создать процесс не удалось, сообщаем об этом и
        завершаем работу */
        printf("Can\'t fork child\n");
        exit(-1);
    } else if (result > 0) {
        /* Мы находимся в родительском процессе, который будет
        передавать информацию процессу-ребенку. В этом процессе
        открываем FIFO на запись.*/
        if((fd = open(name, O_WRONLY)) < 0){
            /* Если открыть FIFO не удалось, печатаем об этом
            сообщение и прекращаем работу */
            printf("Can\'t open FIFO for writing\n");
            exit(-1);
        }
```

```
        /* Пробуем записать в FIFO 14 байт, т.е. всю строку
        "Hello, world!" вместе с признаком конца строки */
        size = write(fd, "Hello, world!", 14);
        if(size != 14){
        /* Если записалось меньшее количество байт,то сообщаем
        об ошибке и завершаем работу */
            printf("Can\'t write all string to FIFO\n");
            exit(-1);
        }
        /* Закрываем входной поток данных и на этом родитель
        прекращает работу */
        close(fd);
        printf("Parent exit\n");
    } else {
        /* Мы находимся в порожденном процессе, который будет
        получать информацию от процесса-родителя. Открываем
        FIFO на чтение.*/
        if((fd = open(name, O_RDONLY)) < 0){
            /* Если открыть FIFO не удалось, печатаем об этом
            сообщение и прекращаем работу */
            printf("Can\'t open FIFO for reading\n");
            exit(-1);
        }
        /* Пробуем прочитать из FIFO 14 байт в массив, т.е.
        всю записанную строку */
        size = read(fd, resstring, 14);
        if(size < 0){
            /* Если прочитать не смогли, сообщаем об ошибке
            и завершаем работу */
            printf("Can\'t read string\n");
            exit(-1);
        }
        /* Печатаем прочитанную строку */
        printf("%s\n",resstring);
        /* Закрываем входной поток и завершаем работу */
        close(fd);
    }
    return 0;
}
```

12. Введите предыдущий код в файл fifo-example.c, скомпилируйте и проверьте его работу.
13. Зафиксируйте изменения с комментарием «Forked FIFO».
14. Напишите программы pipe-writer.c создающую именованный канал one.pipe в текущей директории и передающей туда сообщение «Hello, world!».
15. Скомпилируйте и запустите программу.
16. Что с ней произошло?
17. Запустите второй терминал.
18. Перейдите в рабочую директорию лабораторной работы.
19. Выполните:

```
~/work/os-03-pipe$ cat one.pipe
```

20. Что произошло в обоих терминалах?

21. Напишите программу pipe-reader.c для чтения сообщения, не более 255 символов из именованного канала one.pipe.
22. Скомпилируйте и проверьте одновременную работу двух программ.
23. Зафиксируйте изменения с комментарием «FIFO2».

Дополнительные задания

1. После каждого запуска программ из лабораторной работы остаются pipe-файлы, которые необходимо удалять вручную. Решите эту проблему.
2. Самостоятельно изучите функции popen() и pclose(). Используйте её для запуска и контроля работы программ pipe-writer и pipe-reader.