Міністерство освіти і науки України
Національний технічний університет України
«Київський політехнічний інститут»
Факультет електроніки
Кафедра фізичної та біомедичної електроніки

Ю.В.Вунтесмері

Цифрові технології в мікроелектроніці

Методичні вказівки до виконання лабораторних робіт

для студентів спеціалізації

7.05080102, 8.05080102 - Фізична та біомедична електроніка

Рекомендовано Вченою радою факультету електроніки НТУУ «КПІ»

Київ – 2012

Вунтесмері Ю.В. Цифрові технології в мікроелектроніці: методичні вказівки до виконання лабораторних робіт для студентів спеціалізації 7.05080102, 8.05080102 - Фізична та біомедична електроніка / Ю.В.Вунтесмері. – К. : НТУУ «КПІ», 2012. – с.

Гриф надано Вченою радою ФЕЛ НТУУ «КПІ»
(протокол № 08/12 від 30 серпня 2012 р.)

Затверджено на засіданні
кафедри фізичної та біомедичної електроніки
(протокол № 1 від 28 серпня 2012 р.)

Навчально-методичне видання
ЦИФРОВІ ТЕХНОЛОГІЇ В МІКРОЕЛЕКТРОНІЦІ.
МЕТОДИЧНІ ВКАЗІВКИ ДО ВИКОНАННЯ ЛАБОРАТОРНИХ РОБІТ ДЛЯ СТУДЕНТІВ СПЕЦІАЛІЗАЦІЇ
09.0804 - ФІЗИЧНА ТА БІОМЕДИЧНА ЕЛЕКТРОНІКА

Укладачі:

Вунтесмері Юрій Володимирович, к.т.н.,
доцент кафедри фізичної та біомедичної електроніки

Відповідальний редактор — *В. І. Тимофєєв*, д. т. н., проф.

Рецензент: *О. В. Борисов*, к. т. н., проф.

За редакцією укладачів

Зміст

Вступ

Цикл лабораторних робіт з дисципліни "Цифрові технології в мікроелектроніці" призначений для набуття студентами практичних навичок програмування на рівні системних викликів операційних систем для вирішення задач управління процесами, потоками, пам'яттю, файловою системою, мережевими сокетами та периферійними пристроями комп'ютерних систем.

Цикл складається з чотирьох компьютерних практикумів, кожен з яких мітить дві частини та передбачає розробку та відлагодження однієї чи кількох компьютерних програм. Результатом виконання кожної роботи є написаний та відлагоджений функціонал згідно завдання та розроблені засоби його тестування, придатні для презентації.

Програмування проводиться мовою Сі стандарту ANSI-C99 у програмному інтерфейсі сумісному з POSIX. Для розробки використовується компілятор GCC версії не нижче 4.2 під операційною системою GNU Linux або FreeBSD.

Загальні відомості.

Системне програмування починається із системних викликів. Системні виклики — це звертання до функцій, які робляться з користувацького простору — текстового редактора, вашої улюбленої гри і т.д. — у ядро (головний внутрішній орган системи) для того, щоб запросити в операційної системи яку-небудь службу або ресурс. Системні виклики можуть бути як поширеними та часто використовуваними, такими як read та write, так і екзотичними, як get_thread_area() або set_tid_address().

В Linux реалізоване набагато менше системних викликів, ніж в ядрах більшості інших операційних систем. Наприклад, число системних викликів в архітектурі i386 наближається до 300, а в Microsoft Windows, , їх тисячі. У ядрі Linux для кожної з архитектур (таких, як Alpha, i386 або Powerpc) реалізований власний список доступних системних викликів. Отже, системні виклики, наявні в одній архітектурі, можуть відрізнятися від списку системних викликів в іншій архітектурі. Проте дуже великий набір системних викликів — більше 90 % — реалізований однаково у всіх архітектурах.

Неможливо прямо зв'язати процес користувацького простору із простором ядра. Із причин, що відносяться до безпеки й надійності, процеси користувацького простору не можуть безпосередньо виконувати код ядра або маніпулювати даними ядра. Замість цього надається механізм, за допомогою якого процес користувацького простору може «сигналізувати» ядру, що йому необхідно виконати системний виклик. Процес може відправити переривання ядру і виконати тільки той код, до якого йому дозволяє звертатися ядро. Конкретна реалізація механізму варіюється залежно від архітектури. В архітектурі i386, на-приклад, процес користувацького простору виконує інструкцію програмного переривання int 0x80. Ця інструкція породжує перехід у простір ядра — захищену область ядра, де ядро виконує оброблювач програмного переривання. Процес повідомляє ядро, який системний виклик потрібно виконати і з якими параметрами, використовуючи

апаратні регістри (machine register). Системні виклики позначаються номерами починаючи з нуля (0).

В архітектурі i386, для того щоб запросити системний виклик 5 (тобто виклик open), процес користувацького простору поміщує значення 5 у регістр eax і тільки після цього виконує інструкцію int. Передача параметрів обробляється схожим образом. В архітектурі i386, наприклад, для кожного з можливих параметрів використовується окремий регістр: регістри ebx, ecx, edx, esi і edi містять, у зазначеному порядку, перші п'ять параметрів. У рідких випадках, коли системний виклик містить у собі більше п'яти параметрів, використовується один регістр, який вказує на буфер у користувацькому просторі, де зберігаються всі параметри виклику. Звичайно ж, більшість системних викликів виконуються всього лише з парою параметрів.

В інших архітектурах, виконання системних викликів обробляється по-іншому, хоча загальний дух зберігається. Як системному програмістові, звичайно вам не потрібно знати, яким способом ядро обробляє виконання системних викликів. Це знання закодоване в стандартних угодах про виклики для кожної конкретної архітектури й автоматично обробляється компілятором і бібліотекою C.

Бібліотека Сі (libc) перебуває в самім серці процесів Unix. Навіть коли ви програмуєте на іншій мові, бібліотека Сі, найімовірніше, однаково бере участь у роботі, обгорнена більш высокорівневими бібліотеками, і надає кореневі служби, спрощуючи виконання системних викликів.

Лабораторна робота № 1.
Управління файловою системою та методи мультиплексованого вводу-виводу.

Теоретичні відомості

Перед тем, как считать файл или записать что-нибудь в него, файл необходимо открыть. Ядро ведет списки открытых файлов для всех процессов. Эти списки называются таблицами файлов (file table), которые индексируются при помощи неотрицательных целых значений, известных как дескрипторы. Каждая запись в списке содержит информацию об открытом файле, включая указатель на находящуюся в памяти копию inode файла и связанные метаданные, такие, как позиция в файле и режимы доступа. И пользовательское пространство, и пространство ядра используют файловые дескрипторы в качестве уникальных для каждого процесса маркеров. При открытии файла, возвращается файловый дескриптор, а последующие операции (считывание, запись и т. и.) принимают файловый дескриптор как основной аргумент.

Файловые дескрипторы представляются типом C int. У каждого процесса есть максимальное число файлов, которые он может открыть. Нумерация файловых дескрипторов начинается с нуля и продолжается до значения, на единицу меньшего максимального числа открытых файлов. По умолчанию максимальное значение равно 1024, но его можно увеличить до 1 048 576. Так как отрицательные значения не являются допустимыми файловыми дескрипторами, 1 часто используется в качестве возвращаемого значения ошибки в функциях, которые в противном случае возвращают допустимый файловый дескриптор. Если только процесс явно не закроет их, у каждого процесса по определению есть по крайней мере три открытых файловых дескриптора: 0, 1 и 2. Файловый дескриптор 0 относится к стандартному вводу (standard in, stdin), файловый дескриптор 1 — это стандартный вывод (standard out, stdout), а файловый дескриптор 2 — стандартная ошибка (standard error, stderr).

Обратите внимание, что файловые дескрипторы могут ссылаться не только на обычные файлы. Они используются для доступа к файлам устройств и конвейерам, каталогам и фьютексам, конвейерам FIFO и сокетам — согласно философии «все является файлом», практически ко всему, что можно прочитать или записать, обращение идет через файловый дескриптор.

Системные вызовы open() и creat().

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

int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
int creat(const char *pathname, mode_t mode);
```

Вызов open() используется, чтобы преобразовать путь к файлу в описатель файла (небольшое неотрицательно целое число, которое используется с вызовами read, write и т.п. при последующем вводе-выводе). Если системный вызов завершается успешно, возвращенный файловый описатель является наименьшим описателем, который еще не открыт процессом. В результате этого вызова появляется новый открытый файл. Указатель устанавливается в начале файла. Параметр flags - это флаги O_RDONLY, O_WRONLY или O_RDWR, открывающие файлы "только для чтения", "только для записи" и для чтения и записи соответственно, которые собираются с помощью побитовой операции OR из таких значений, как:

O_CREAT

(если файл не существует, то он будет создан. Владелец (идентификатор пользователя) файла устанавливается в значение эффективного идентификатора пользователя процесса. Группа (идентификатор группы) устанавливается либо в значение эффективного идентификатора группы процесса, либо в значение идентификатора группы родительского каталога (зависит от типа файловой системы, параметров подсоединения (mount) и режима родительского каталога);

O_EXCL

(Если он используется совместно с O_CREAT, то при наличии уже созданного файла вызов open завершится с ошибкой. В этом состоянии, при существующей символьной ссылке не обращается внимание, на что она указывает.);

O_EXCL

(Оно не работает в файловых системах NFS, а в программах, использующих этот флаг для блокировки, возникнет "race condition". Решение для атомарной блокировки файла: создать файл с уникальным именем в той же самой файловой системе (это имя может содержать, например, имя машины и идентификатор процесса), используя link , чтобы создать ссылку на файл блокировки. Если link() возвращает значение 0, значит, блокировка была успешной. В противном случае используйте stat , чтобы убедиться, что количество ссылок на уникальный файл возросло до двух. Это также означает, что блокировка была успешной);

O_NOCTTY

(если pathname указывает на терминальное устройство, то оно не станет терминалом управления процесса, даже если процесс такового не имеет);

O_TRUNC

(если файл уже существует, он является обычным файлом и режим позволяет записывать в этот файл (т.е. установлено O_RDWR или O_WRONLY), то его длина будет урезана до нуля. Если файл является каналом FIFO или терминальным устройством, то этот флаг игнорируется. Иначе действие флага O_TRUNC не определено.

O_APPEND

(Файл открывается в режиме добавления. Перед каждой операцией write файловый указатель будет устанавливаться в конце файла, как если бы использовался lseek); O_APPEND (может привести к повреждению файлов в системе NFS, если несколько процессов одновременно добавляют данные в один файл. Это происходит из-за того, что NFS не поддерживает добавление в файл данных, поэтому ядро на машине-клиенте должно эмулировать эту поддержку);

O_NONBLOCK или O_NDELAY

(если возможно, то файл открывается в режиме non-blocking. Ни open, ни другие последующие операции над возвращаемым описателем файла не заставляют вызывающий процесс ждать. Для работы с каналами FIFO. Этот режим не оказывает никакого действия на не-FIFO файлы.);

O_SYNC

(Файл открывается в режиме синхронного ввода-вывода. Все вызовы write для соответствующего описателя файла блокируют вызывающий процесс до тех пор, пока данные не будут физически записаны. Однако, Вам необходимо прочитать раздел ОГРАНИЧЕНИЯ);

O_NOFOLLOW

(если pathname - это символьная ссылка, то open содержит код ошибки. Это расширение FreeBSD, которое было добавлено в Linux версии 2.1.126. Все прочие символьные ссылки в имени будут обработаны как обычно. Заголовочные файлы из glibc версии 2.0.100 (и более поздних) содержат определение этого флага; ядра версий, более ранних, чем 2.1.126, игнорируют этот флаг);

O_DIRECTORY

(Если pathname не является каталогом, то open укажет на ошибку. Этот флаг используется только в Linux и был добавлен к ядру 2.1.126, чтобы избежать проблем с "отказом от обслуживания", если opendir был вызван для канала FIFO или ленточного устройства. Этот флаг не следует использовать вне реализации opendir);

O_LARGEFILE

(На 32-битных системах, поддерживающих файловые системы (Large), этот флаг позволяет открывать файлы, длина которых больше 31-ого бита).

Некоторые из вышеописанных флагов могут быть изменены с помощью fctnl после открытия файла. Аргумент mode задает права доступа, которые используются в случае создания нового файла. Они модифицируются обычным способом, с помощью umask процесса; права доступа созданного файла равны (mode & ~umask). Обратите внимание, что этот режим применяется только к правам создаваемого файла; open создает файл только для чтения, но может вернуть дескриптор с установленными флагами для чтения и записи.

mode всегда должен быть указан при использовании O_CREAT; во всех остальных случаях этот параметр игнорируется. creat эквивалентен open с flags, которые равны O_CREAT | O_WRONLY | O_TRUNC.

open и creat возвращают новый описатель файла или -1 в случае ошибки (в этом случае значение переменной errno устанавливается должным образом).

Если создается файл, то его время последнего доступа, создания и модификации устанавливаются в значение текущего времени, а также устанавливаются поля времени модификации и создания родительского каталога. Иначе, если файл изменяется с флагом O_TRUNC, то его время создания и время изменения устанавливаются в значение текущего времени.

Системные вызовы read() и write().

```
#include <unistd.h>

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

read() пытается записать count байтов файлового описателя fd в буфер, адрес которого начинается с buf.

Если количество count равно нулю, то read() возвращает это нулевое значение и завершает свою работу. Если count больше, чем SSIZE_MAX, то результат не может быть определен.

При успешном завершении вызова возвращается количество байтов, которые были считаны (нулевое значение означает конец файла), а позиция файла увеличивается на это значение. Если количество прочитанных байтов меньше, чем количество запрошенных, то это не считается ошибкой: например, данные могли быть почти в конце файла, в канале, на терминале, или read() был прерван сигналом. В случае ошибки возвращаемое значение равно -1, а переменной errno присваивается номер ошибки. В этом случае позиция файла не определена.

write() записывает до count байтов из буфера buf в файл, на который ссылается файловый описатель fd. POSIX указывает на то, что вызов write(), произошедший после вызова read() возвращает уже новое значение. Заметьте, что не все файловые системы соответствуют стандарту POSIX.

В случае успешного завершения возвращается количество байтов, которые были записаны (ноль означает, что не было записано ни одного байта). В случае ошибки возвращается -1, а переменной errno присваивается соответствующее значение. Если count равен нулю, а файловый описатель ссылается на обычный файл, то будет возвращен ноль и больше не будет произведено никаких действий. Для специальных файлов результаты не могут быть перенесены на другую платформу.

Мультиплексированный ввод-вывод и системный вызов select()

Мультиплексированный ввод-вывод (multiplexed I/O) позволяет

приложению одновременно фиксироваться на нескольких файловых дескрипторах и получать уведомления, когда один из них становится доступным для чтения или записи без блокировки. Таким образом, мультиплексированный ввод-вывод становится центральной точкой для приложений, сконструированных на основе приблизительно такого алгоритма:

1) сообщить, когда любой из списка дескрипторов файла будет готов для ввода-вывода;

2) заснуть до тех пор, пока один или несколько файловых дескрипторов не

будут готовы;

3) проснуться и указать, какие дескрипторы готовы;

4) обработать все готовые к вводу-выводу дескрипторы файлов без блокировки;

5) вернуться к шагу 1 и начать сначала.

В POSIX-совместимых ОС предусмотрено три решения для мультиплексированного ввода-вывода: интерфейсы select, poll и epoll.

```
/* В соответствие с POSIX 1003.1-2001 */
#include <sys/select.h>

/* В соответствие с более ранними стандартами */
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>

int select(int n, fd_set *readfds, fd_set *writefds, fd_set
*exceptfds, struct timeval *timeout);

FD_CLR(int fd, fd_set *set);
FD_ISSET(int fd, fd_set *set);
FD_SET(int fd, fd_set *set);
FD_ZERO(fd_set *set);
```

Функция select ждет изменения статуса нескольких файловых описателей.

Отслеживаются 3 независимых набора описателей. Те, что перечислены в readfds, будут отслеживаться для того, чтобы обнаружить появление символов, доступных для чтения (говоря более точно, чтобы узнать, не будет ли блокировано чтение; описатель файла также будет указывать на конец файла); те описатели, которые указаны в writefds, будут отслеживаться для того, чтобы узнать, не заблокирован ли процесс записи; те же, что указаны в параметре exceptfds, будут отслеживаться для обнаружения исключительных ситуаций. При возврате из функции наборы описателей модифицируются, чтобы показать, какие описатели фактически изменили свой статус.

Для манипуляций наборами существуют четыре макроса: FD_ZERO, очищающий набор; FD_SET и FD_CLR добавляют заданный описатель к набору или удаляют его из набора; FD_ISSET проверяет, является ли описатель частью набора; этот макрос полезен после возврата из функции select

n - на единицу больше самого большого номера описателей из всех наборов.

timeout - это верхняя граница времени, которое пройдет перед возвратом из select. Можно использовать нулевое значение, и при этом select завершится

немедленно. Если timeout равен NULL (нет времени ожидания), то select будет ожидать изменений неопределенное время.

```
struct timeval {
    long    tv_sec;        /* seconds */
    long    tv_usec;      /* microseconds */
};
```

Иногда select вызывается с пустыми наборами (всеми тремя), n равным нулю и непустым timeout для переносимой реализации перехода в режим ожидания (sleep) на периоды с точностью более секунды.

В Linux функция select изменяет timeout для отражения времени, проведенного не в режиме ожидания; большая часть других реализаций этого не делают. Это вызывает проблемы как при переносе кода Linux, читающего timeout, на другие операционные системы, так и при переносе на Linux кода, использующего struct timeval для многократного вызова select в цикле без его переинициализации. Во избежание этого следует считать, что timeout не определен после возврата из select.

При успешном завершении select возвращает количество описателей, находящихся в наборах, причем это количество может быть равным нулю, если время ожидания на исходе, а интересующие нас события так и не произошли. При ошибке возвращаемое значение равно -1, а переменной errno присваивается номер ошибки; наборы описателей и значение timeout становятся неопределенными, поэтому при ошибке нельзя полагаться на их значение.

Завдання

Частина 1: Функції послідовного вводу-виводу*. Скласти програму, яка:*

1. Отримує аргументами командної строки два імені файлів
2. Перший файл відкриває для читання, другий до перезапису, або створює, якщо його не існує (для створення використати атрибути 0644)
3. Послідовно читає дані з першого файлу буферами фіксованого об'єму (наприклад 512 байт) та записує у другий.
4. Перед записом із вмістом буферу проводить таке перетворення – усі рядкові літери латинського алфавіту перетворює на відповідні прописні. Для цього розрахувати відповідний зсув за таблицею ASCII (див. man ascii).
5. В процесі перезапису проводить підрахунок перезаписаних байт, та наприкінці виводить у вихідний потік сумарний обсяг переписаних даних.

Частина 2: Функції мультиплексованого вводу-виводу*. Скласти програму, яка:*

1. Отримує аргументом командної строки довільний строковий ідентифікатор.
2. Налаштовує системний виклик *select* для очікування читання у вхідному потоці (файловий дескріптор STDIN_FILENO) з таймаутом 5 секунд.
3. При отриманні можливості читання, прочитати з потоку буфер довжиною не більше 1024 байта та вивести його у вихідний потік з поміткою у вигляді ідентифікатора п.1.
4. При спливанні таймауту, вивести у потік помилок повідомлення про це з поміткою у вигляді ідентифікатора п.1 та знову налаштувати системний виклик *select* (п.2).
5. Протестувати роботу програми отримуючи через вхідний потік результати вводу з клавіатури.

Рекомендації до виконання.

- Для перезапису файлів використовувати виключно системні виклики. Для виводу у вихідний потік припустимо використання функцій <*stdio*>
- Обов'язково діагностувати помилки, що виникають під час відкриття файлів, читання та запису.
- Для діагностики помилок використовувати *errno* та *strerror*
- Для налаштування системного виклику *select* використати макроси FD_ZERO, FD_SET та структуру timeval.
- **Додаткове завдання:** Повторити функціонал програми ч.2. з використанням системного виклику *poll.*

Посилання

- Рекомендована література: №№ 1-3.
- Офіційна документація стандартної бібліотеки Сі 2-й розділ, статті: open, creat, close, read, write, fsync, fseek, select, poll, errno, strerror.

Лабораторна робота № 2.
Управління процесами, відокремлення процесів.

Теоретичні відомості.

Каждый процесс представляется уникальным идентификатором процесса (process ID, название которого обычно сокращается просто до pid). Гарантируется, что в любой конкретный момент времени значение pid уникально. Это означает, что, хотя в момент времени t_0 может существовать только один процесс с pid 555 (или ни одного), нет никакой гарантии, что в момент времени t_1 не будет существовать совершенно другой процесс с pid 555. На практике обычно предполагается, что ядро не спешит повторно использовать и переназначать идентификаторы процессов, но, как вы совсем скоро узнаете, такое предположение небезопасно.

У процесса бездействия (idle process), который ядро «выполняет», когда нет никаких других доступных для выполнения процессов, идентификатор pid всегда равен 0. Первый процесс, который ядро выполняет после загрузки системы, называется процессом init — процессом инициализации, и значение pid для него равно 1. Обычно процесс init в Linux принадлежит программе init.

По умолчанию ядро накладывает ограничение на максимальное значение идентификатора процесса, равное 32768. Это необходимо для совместимости с более старыми системами Unix, в которых для идентификаторов процесса применялись меньшие 16-разрядные типы.

Ядро выделяет процессам идентификаторы строго линейно. Если в определенный момент времени максимальное значение выделенного идентификатора pid равно 17, то следующим будет выделен идентификатор pid, равный 18, даже если на момент запуска нового процесса процесс с последним назначенным идентификатором pid 17 уже не будет выполняться. Ядро не начинает повторно использовать значения идентификаторов процесса до тех пор, пока нумерация не возвращается к началу

Процесс, запускающий новый процесс, называется родительским процессом или предком (parent); новый процесс называется дочерним процессом или потомком (child). Каждый процесс запускается каким-то другим процессом (за исключением, конечно же, процесса init). Таким образом, у каждого потомка есть предок. Это взаимоотношение фиксируется при помощи идентификатора предка каждого процесса (parent process ID, ppid), значение которого для каждого потомка равно pid родительского процесса.

Каждым процессом владеют пользователь (user) и группа (group). Информация о владении используется для управления правами доступа к ресурсам. Каждый дочерний процесс наследует владельцев предка, то есть им владеют тот же пользователь и группа, что и его предком.

Каждый процесс также является частью группы процессов (process group), что просто выражает его взаимоотношения с другими процессами, — не путайте эту группу с вышеупомянутой концепцией пользователей и групп. Потомки обычно принадлежат той же группе процессов, что и их родители. Помимо этого, когда оболочка запускает конвейер (то есть когда происходит перенаправление стандартного потока вівода одноо процесса на стандартній вход другого), все команды в конвейере становятся членами одной группы процессов.

Понятие группы процессов упрощает отправку сигналов и получение информации обо всем конвейере, а также обо всех потомках процессов в конвейере. С точки зрения пользователя, группа процессов тесно связана с понятием задания (job).

В Unix акт загрузки в память и исполнения образа программы отделен от акта создания нового процесса. Один системный вызов (в действительности, один вызов из семейства вызовов) загружает двоичную программу в память, заменяя ею предыдущее содержимое адресного пространства, и начинает выполнение новой программы. Это называется исполнением (executing) новой программы, и функциональность обеспечивается семейством вызовов exec.

Другой системный вызов используется для создания нового процесса, который первоначально представляет собой практически копию своего родительского

процесса. Часто новый процесс немедленно начинает выполнять новую программу. Акт создания нового процесса называется ветвлением (forking), и эту функциональность предоставляет системный вызов fork(). Таким образом, чтобы выполнить новый образ программы в новом процессе, необходимы два акта — сначала ветвление, для создания нового процесса, а затем исполнение, для загрузки нового образа в этот процесс.

Системные вызовы getpid(), getppid().

```
#include <sys/types.h>
#include <unistd.h>
pid_t getpid(void);
pid_t getppid(void);
```

getpid возвращает идентификатор ID текущего процесса. (Это часто используется функциями, которые генерируют уникальные имена временных файлов)

getppid возвращает идентификатор ID родительского процесса.

Системные вызовы getuid(), getgid().

```
#include <unistd.h>
#include <sys/types.h>
uid_t getuid(void);
uid_t geteuid(void);
gid_t getgid(void);
gid_t getegid(void);
```

getuid возвращает фактический идентификатор ID пользователя в текущем процессе.

geteuid возвращает эффективный идентификатор ID пользователя в текущем процессе.

Фактический ID соответствует ID пользователя, который вызвал процесс. Эффективный ID соответствует установленному setuid биту на исполняемом файле.

getgid возвращает действительный идентификатор группы текущего процесса.

getegid возвращает еффективный идентификатор группы текущего процесса. Действительный идентификатор соответствует идентификатору вызывающего процесса. Эффективный идентификатор соответствует биту setuid на исполняемом файле.

Системные вызовы gettid(), getsid().

```
#include <sys/types.h>
#include <linux/unistd.h>
_syscall0(pid_t,gettid)
pid_t gettid(void);

#include <unistd.h>
pid_t getsid(pid_t pid);
```

gettid возвращает идентифкатор треда текущего процесса. Это эквивалентно идентифкатору процесса (который возвращает getpid), за исключением того, что данный процесс является частью группы тредов (созданной через флаг CLONE_THREAD в системном вызове clone). Все процессы в той же группе тредов имеют одинаковый идентифкатор процесса PID, но каждый из них имеет уникальный идентификатор треда процесса TID.

getsid(0) возвращает идентификатор (ID) сессии, вызвавшего процесса. getsid(p) возвращает идентификатор сессии процесса с номером p. (Идентификатор сессии процесса - это идентификатор группы процесса, который является лидером сессии). В случае ошибки, (pid_t) будет возвращено значение -1 и значение errno будет установлено соответствующим образом.

Системный вызов fork().

```
#include <sys/types.h>
#include <unistd.h>
pid_t fork(void);
```

fork создает процесс-потомок, который отличается от родительского только значениями PID (идентификатор процесса) и PPID (идентификатор

родительского процесса), а также тем фактом, что счетчики использования ресурсов установлены в 0. Блокировки файлов и сигналы, ожидающие обработки, не наследуются.

Под Linux fork реализован с помощью "копирования страниц при записи" (copy-on-write, COW), поэтому расходы на fork сводятся к копирования таблицы страниц родителя и созданию уникальной структуры, описывающей задачу.

Системный вызов wait().

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
```

Функция wait приостанавливает выполнение текущего процесса до тех пор, пока дочерний процесс не завершится, или до появления сигнала, который либо завершает текущий процесс, либо требует вызвать функцию-обработчик. Если дочерний процесс к моменту вызова функции уже завершился (так называемый "зомби" ("zombie")), то функция немедленно возвращается. Системные ресурсы, связанные с дочерним процессом, освобождаются. Функция waitpid приостанавливает выполнение текущего процесса до тех пор, пока дочерний процесс, указанный в параметре pid, не завершит выполнение, или пока не появится сигнал, который либо завершает текущий процесс либо требует вызвать функцию-обработчик. Если указанный дочерний процесс к моменту вызова функции уже завершился (так называемый "зомби"), то функция немедленно возвращается. Системные ресурсы, связанные с дочерним процессом, освобождаются. Параметр pid может принимать несколько значений:

< -1 означает, что нужно ждать любого дочернего процесса, идентификатор группы процессов которого равен абсолютному значению pid.

-1 означает ожидание любого дочернего процесса; функция wait ведет себя точно так же.

0 означает ожидание любого дочернего процесса, идентификатор группы процессов которого равен идентификатору текущего процесса.

> 0 означает ожидание дочернего процесса, чей идентификатор равен pid.

Значение options создается путем логического сложения нескольких следующих констант:

WNOHANG означает немедленное возвращение управления, если ни один дочерний процесс не завершил выполнение.

WUNTRACED означает возврат управления и для остановленных (но не отслеживаемых) дочерних процессов, о статусе которых еще не было сообщено. Статус для отслеживаемых остановленных подпроцессов также обеспечивается без этой опции.

Если status не равен NULL, то функции wait и waitpid сохраняют информацию о статусе в переменной, на которую указывает status. Этот статус можно проверить с помощью нижеследующих макросов (они принимают в качестве аргумента буфер (типа int), --- а не указатель на буфер!):

WIFEXITED(status) не равно нулю, если дочерний процесс успешно завершился.

WEXITSTATUS(status) возвращает восемь младших битов значения, которое вернул завершившийся дочерний процесс. Эти биты могли быть установлены в аргументе функции exit() или в аргументе оператора return функции main(). Этот макрос можно использовать, только если WIFEXITED вернул ненулевое значение.

WIFSIGNALED(status) возвращает истинное значение, если дочерний процесс завершился из-за необработанного сигнала.

WTERMSIG(status) возвращает номер сигнала, который привел к завершению дочернего процесса. Этот макрос можно использовать, только если WIFSIGNALED вернул ненулевое значение.

WIFSTOPPED(status) возвращает истинное значение, если дочерний процесс, из-за которого функция вернула управление, в настоящий момент остановлен; это возможно, только если использовался флаг WUNTRACED или когда подпроцесс отслеживается.

WSTOPSIG(status) возвращает номер сигнала, из-за которого дочерний процесс был остановлен. Этот макрос можно использовать, только если WIFSTOPPED вернул ненулевое значение.

Системный вызов execve()

```
#include <unistd.h>
int execve(const char *filename, char *const argv [], char *const envp[]);
```

execve() выполняет программу, заданную параметром filename. Программа должна быть или двоичным исполняемым файлом, или скриптом, начинающимся со строки вида "#! интерпретатор [аргументы]". В последнем случае интерпретатор -- это правильный путь к исполняемому файлу, который не является скриптом; этот файл будет выполнен как интерпретатор [arg] filename.

argv -- это массив строк, аргументов новой программы. envp -- это массив строк в формате key=value, которые передаются новой программе в качестве окружения (environment). Как argv, так и envp завершаются нулевым указателем. К массиву аргументов и к окружению можно обратиться из функции main(), которая объявлена как int main(int argc, char *argv[], char *envp[]).

execve() не возвращает управление при успешном выполнении, а код, данные, bss и стек вызвавшего процесса перезаписываются кодом, данными и стеком загруженной программы. Новая программа также наследует от вызвавшего

процесса его идентификатор и открытые файловые дескрипторы, на которых не было флага закрыть-при-exec (close-on-exec, COE). Сигналы, ожидающие обработки, удаляются. Переопределённые обработчики сигналов возвращаются в значение по умолчанию. Обработчик сигнала SIGCHLD (когда установлен в SIG_IGN) может быть сброшен или не сброшен в SIG_DFL.

Если текущая программа выполнялась под управлением ptrace, то после успешного execve() ей посылается сигнал SIGTRAP.

При успешном завершении execve() не возвращает управление, при ошибке возвращается -1, а значение errno устанавливается должным образом.

Системный вызов setsid().

```
#include <unistd.h>
pid_t setsid(void);
```

setsid() создает новый сеанс, если вызывающий процесс не создает группу. Вызывающий процесс становится ведущим в группе, ведущим процессом нового сеанса и не имеет контролирующего терминала. Идентификаторы группы процессов и сеанса при установке будут равными идентификатору вызывающего процесса. Вызывающий процесс будет единственным в этой группе и сеансе. Возвращает идентификатор сеанса вызывающего процесса.

При ошибке возвращаемое значение равно -1. Единственная ошибка, которая может произойти, - это EPERM. Она происходит, когда идентификатор группы процессов любого процесса равен идентификатору вызывающего процесса. В этом случае функция setsid не может быть выполнене, так как процесс уже является ведущим в группе.

Создающий группу процесс ("лидер") - это процесс, идентификатор группы процессов которого равен идентификатору самого процесса. Для того, чтобы удостовериться, что функция setsid выполнена, создайте дочерний процесс при помощи команды fork и выйдите из процесса, затем в дочернем процессе сделайте вызов setsid().

Завдання

Частина 1. Запуск, завершення та параметри процесу. *Скласти програму, яка:*

1. Отримує та друкує інформацію про параметри свого процесу за допомогою системних викликів getpid(), getgid(), getsid() тощо.
2. Виконує розгалудження процесу за допомогою системного виклику fork().
3. Для процесу-батька та процесу-нащадка окремо роздруковує їхні ідентифікатори у циклі.
4. Виконати очікування процесом-батьком завершення нащадка.
5. Повідомляє про завершення процесів — батька та нащадка.
6. Пояснити отримані результати

Частина 2. Демонізація процесу. *Скласти програму, яка:*

1. Відкриває на запис текстовий файл логу та робить у нього запис про старт програми.
2. Виконує демонізацію свого функціоналу за таким алгоритмом:
    - Виконує fork()
    - Для процесу-батька робить запис у лог про породження нащадка та закриває себе викликом exit().
    - Для поцесу-нащадка:
        - робить setsid()
        - змінює поточний каталог на “/”
        - закриває усі відкриті батьком дескриптори
        - відкриває “/dev/null” на запис для трьох стандартних потоків
3. Відкриває лог та робить у нього запис про параметри демонізованого процесу.
4. Виконує витримку часу у нескінченному циклі.
5. Пояснити отримані результати

Рекомендації до виконання.

1. Розділення функціоналів батька та нащадка виконати за результатом виконання fork().
2. Для очікування завершення нащадка, використовувати системний виклик wait().
3. Для стеження за останніми записами у логу використовуйте tail -f
4. Для завершення роботи демона використовуйте kill -9

Посилання.

- Рекомендована література: №№ 1-3.
- Офіційна документація стандартної бібліотеки Сі 2-й розділ, статті: getpid, getgid, getuid, getgid, fork,.wait, exit, setsid.

Лабораторна робота № 3.
Сигнали. Розділення пам`яті між процесами.

Теоретичні відомості.

Сигналы — это программные прерывания, предоставляющие механизм для обработки асинхронных событий. Такие события могут приходить из-за пределов системы — например, при вводе пользователем символа прерывания, или возникать вследствие действий в программе или ядре, например, когда процесс исполняет код, в котором выполняется деление на ноль. В виде примитивной формы взаимодействия между процессами (interprocess communication, IPC) один процесс также может отправлять сигналы другому процессу.

Самое главное в сигналах — это не только то, что события происходят асинхронно, например пользователь может нажать Ctrl+C в любой момент выполнения программы, но и то, что программа асинхронно обрабатывает сигналы.

Функции обработки сигналов регистрируются в ядре, которое асинхронно вызывает функции из оставшейся части программы, как только сигналы доставляются.

У сигналов очень точный жизненный цикл. Сначала сигнал поднимается (raise), также мы иногда говорим, что он отправляется (send) или генерируется (generate)). После этого ядро хранит (store) сигнал до тех пор, пока у него не появляется шанс доставить его. Наконец, как только такая возможность предоставляется, ядро соответствующим образом обрабатывает (handle) сигнал. Ядро может выполнить одно из трех действий, в зависимости от того, что у него запросил процесс:

Игнорировать сигнал

Никакое действие не предпринимается. Существуют два сигнала, которые не могут быть проигнорированы: SIGKILL и SIGSTOP. Причина этого заключается в том, что системным администраторам нужно иметь возможность убивать и

останавливать процессы, и было бы нарушением такого права, если бы процесс мог просто игнорировать SIGKILL (что делало бы его неубиваемым) или SIGSTOP (что делало бы его неостанавливаемым).

Захватить и обработать сигнал

Ядро приостанавливает исполнение текущего пути кода процесса и переходит к ранее зарегистрированной функции. Затем процесс исполняет эту функцию. После того как процесс возвращается из этой функции, он перепрыгивает обратно в то место, где находился на тот момент, когда был захвачен сигнал. SIGINT и SIGTERM — это два сигнала, которые захватываются чаще всего. Процессы захватывают SIGINT, чтобы обработать ситуацию, когда пользователь вводит символ прерывания, — например, терминал может захватить его и вернуться к главной строке приглашения. Процессы захватывают SIGTERM для выполнения необходимой уборки, например отсоединения от сети или удаления временных файлов перед завершением. Сигналы SIGKILL и SIGSTOP захватить невозможно.

Выполнить действие по умолчанию

Это действие зависит от того, какой сигнал отправляется. Действием по умолчанию часто бывает завершение процесса. Например, именно так обрабатывается сигнал SIGKILL. Однако многие сигналы предоставляются для специфических целей, которые беспокоят программистов лишь в определенных ситуациях, и эти сигналы по умолчанию игнорируются, так как многие программы просто в них не заинтересованы.

Номера сигналов начинаются с единицы (обычно единице соответствует SIGHUP) и линейно продолжаются далее. Всего сигналов около 30, но в большинстве программ регулярно используется лишь несколько из них. Сигнала со значением 0 нет — это специальное значение, называемое нулевым сигналом (null signal).

У каждого сигнала есть символическое имя, начинающееся с префикса SIG. Например, SIGINT — это сигнал, который отправляется, когда пользователь нажимает клавишное сочетание Ctrl+C, SIGABRT — это сигнал, отправляемый,

когда процесс вызывает функцию abortO, а SIGKILL — сигнал, отправляемый, когда процесс принудительно завершается.

Простейший и самый старый интерфейс для управления сигналами — это системный вызов signal().

Регистрация обработчика сигнала.

```
#include <signal.h>
typedef void (*sighandler_t)(int);
sighandler_t signal (int signo, sighandler_t handler);
```

Успешный вызов signal удаляет текущее действие, предпринимаемое при получении сигнала signo, и вместо этого обрабатывает сигнал обработчиком, указанным при помощи аргумента handler. Значением аргумента signo может быть одно из названий сигналов, например SIGINT или SIGUSR1. Вспомните, что процесс не может захватывать ни SIGKILL, ни SIGSTOP, поэтому установка обработчика для любого из этих сигналов не имеет никакого смысла.

Функция handler должна возвращать значение void, что имеет смысл, так как (в отличие от обычных функций) в программе нет стандартного места, в котором данная функция должна была бы возвращать результат. Она принимает один аргумент — целочисленное значение, представляющее собой идентификатор обрабатываемого сигнала (например, SIGUSR2). Это позволяет одной функции обрабатывать несколько сигналов. Ее прототип имеет такую форму:

```
void my_handler (int signo);
```

Определены два обработчика, позволяющие при помощи вызова signal выполнить специальные операции по отношению к сигналам:

SIG_DFL - Восстановить поведение по умолчанию для сигнала, указанного при помощи аргумента signo. Например, в случае сигнала SIGPIPE процесс будет завершаться.

SIG_IGN - Игнорировать сигнал, указанный при помощи параметра signo.

Ожидание сигнала.

```
#include <unistd.h>
int pause(void);
```

После вызова функции pause вызывающий процесс (или подзадача) приостанавливается до тех пор, пока не получит сигнал. Данный сигнал либо остановит процесс, либо заставит его вызвать функцию обработки этого сигнала.

Функция pause возвращается только тогда, когда сигнал был перехвачен и произошел возврат из функции обработки сигнала. В этом случае она возвращает -1, а значение переменной errno становится равным EINTR.

Отправка сигнала другому процессу.

```
#include <sys/types.h>
#include <signal.h>
int kill(pid_t pid, int sig);
int killpg(int pgrp, int sig);
```

Системный вызов *kill* может быть использован для посылки какого-либо сигнала какому-либо процессу или группе процесса.

Если значение pid является положительным, сигнал sig посылается процессу с идентификатором pid.

Если pid равен 0, то sig посылается каждому процессу, который входит в группу текущего процесса.

Если pid равен -1, то sig посылается каждому процессу, за исключением процесса с номером 1 (init), но есть нюансы, которые описываются ниже.

Если pid меньше чем -1, то sig посылается каждому процессу, который входит в группу процесса -pid.

Если sig равен 0, то никакой сигнал не посылается, а только выполняется проверка на ошибку.

В случае успеха, возвращается ноль. При ошибке, возвращается -1 и значение errno устанавливается соответствующим образом.

Невозможно послать сигнал процессу с номером 1, т.е. процессу init, для которого не устанавливается обработчик сигналов. Так сделано, чтобы быть

уверенным, что в случае какой-либо нештатной ситуации, рабата системы не будет завершена аварийно.

POSIX 1003.1-2001 требует, чтобы kill(-1,sig) посылал sig всем процессам, которым текущий процесс может послать сигналы, за исключением возможно нескольких, определяемых реализацией системы процессов. Linux разрешает любому процессу посылать сигнал себе, но в Linux системный вызов kill(-1,sig) не посылает сигнал текущему процессу.

Killpg отправляет сигнал sig группе процессов pgrp. Если значение pgrp равно 0, то killpg отправляет сигнал текущей группе процессов. Процессы группы и процесс, посылающий сигнал, должны иметь один и тот же эффективный идентификатор пользователя, или процесс-отправитель должен иметь права суперпользователя. Единственное исключение из этого - сигнал продолжения SIGCONT может быть отправлен любому процессу - потомку текущего процесса.

При удачном завершении возвращается 0. При ошибке возвращается -1, а переменной errno присваивается соответствующее значение.

Расширенная обработка сигналов.

```
#include <signal.h>
int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact);
int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);
int sigpending(sigset_t *set);
int sigsuspend(const sigset_t *mask);
```

Системный вызов sigaction используется для изменения действий процесса при получении соответствующего сигнала.

Параметр signum задает номер сигнала и может быть равен любому номеру, кроме SIGKILL и SIGSTOP.

Если параметр act не равен нулю, то новое действие, связянное с сигналом signum, устанавливается соответственно act. Если oldact не равен нулю, то предыдущее действие записывается в oldact.

Структура sigaction имеет следующий формат:

```
struct sigaction {
 void (*sa_handler)(int);
 void (*sa_sigaction)(int, siginfo_t *, void *);
 sigset_t sa_mask;
 int sa_flags;
 void (*sa_restorer)(void);
}
```

В некоторых архитектурах используется объединение элементов, но не используйте sa_handler и sa_sigaction вместе.

sa_handler задает тип действий процесса, связанный с сигналом signum, и может быть равен: SIG_DFL для выполнения стандартных действий, SIG_IGN для игнорирования сигнала,- или быть указателем на функцию обработки сигнала.

sa_mask задает маску сигналов, которые должны блокироваться при обработке сигнала. Также будет блокироваться и сигнал, вызвавший запуск функции, если только не были использованы флаги SA_NODEFER или SA_NOMASK.

sa_flags содержит набор флагов, которые могут влиять на поведение процесса при обработке сигнала. Он состоит из следующих флагов:

SA_NOCLDSTOP Если signum равен SIGCHLD, то уведомление об остановке дочернего процесса не будет получено (т.е., в тех случаях, когда дочерний процесс получает сигнал SIGSTOP, SIGTSTP, SIGTTIN или SIGTTOU).

SA_ONESHOT или SA_RESETHAND Восстановить поведение сигнала после одного вызова обработчика.

SA_ONSTACK Вызвать обработчик сигнала в дополнительном стеке сигналов, предоставленном sigaltstack . Если дополнительный стек недоступен, то будет использован стек по умолчанию.

SA_RESTART Поведение должно соответствовать семантике сигналов BSD и позволять некоторым системным вызовам работать, в то время как идет обработка сигналов.

SA_NOMASK or SA_NODEFER Не препятствовать получению сигнала при его обработке.

SA_SIGINFO Обработчик сигнала требует 3-х аргументов, а не одного. В этом случае надо использовать параметр sa_sigaction вместо sa_handler. Параметр siginfo_t поля sa_sigaction является структурой, состоящей из следующих элементов:

```
siginfo_t {
   int      si_signo;  /* Номер сигнала */
   int      si_errno;  /* Значение errno */
   int      si_code;   /* Код сигнала */
   pid_t    si_pid;    /* Идентификатор процесса, пославшего сигнал */
   uid_t    si_uid;    /* Реальный идентификатор пользователя
процесса, пославшего сигнал */
   int      si_status; /* Выходное значение или номер сигнала */
   clock_t  si_utime;  /* Занятое пользователем время */
   clock_t  si_stime;  /* Использованное системное время */
   sigval_t si_value;  /* Значение сигнала */
   int      si_int;    /* Сигнал  POSIX.1b */
   void *   si_ptr;    /* Сигнал  POSIX.1b */
   void *   si_addr;   /* Адрес в памяти, приводящий к ошибке */
   int      si_band;   /* Общее событие */
   int      si_fd;     /* Описатель файла */
}
```

Поля si_signo, si_errno и si_code определены для всех сигналов. Остальная часть структуры может быть объединением, поэтому

Следует работать только с теми полями, которые имеют смысл для конкретного сигнала. Вызов kill, сигналы POSIX.1b и SIGCHLD заполняют поля si_pid и si_uid. SIGCHLD также заполняет поля si_status, si_utime и si_stime. Поля si_int и si_ptr задаются процессом, пославшим сигнал POSIX.1b. Сигналы SIGILL, SIGFPE, SIGSEGV и SIGBUS заполняют поле si_addr адресом в памяти, который привел к ошибке. Сигнал IGPOLL заполняет si_band и si_fd. si_code указывает на причину отправки сигнала. Это значение, а не битовая маска.

Системный вызов sigprocmask используется для того, чтобы изменить список блокированных в данный момент сигналов. Работа этой функции зависит от значения параметра how следующим образом:

SIG_BLOCK Набор блокируемых сигналов - объединение текущего набора и аргумента set.

SIG_UNBLOCK Сигналы, устанавливаемое значение битов которых равно set, удаляются из списка блокируемых сигналов. Допускается разблокировать незаблокированные сигналы.

SIG_SETMASK Набор блокируемых сигналов приравнивается к аргументу set.

Если значение поля oldset не равно нулю, то предыдущее значение маски сигналов записывается в oldset.

Системный вызов sigpending позволяет определить наличие ожидающих сигналов (полученных заблокированных сигналов). Маска ожидающих сигналов помещается в set.

Системный вызов sigsuspend временно изменяет значение маски блокировки сигналов процесса на указанное в mask, и затем приостанавливает работу процесса до получения соответствующего сигнала.

Функции sigaction, sigprocmask и sigpending возвращают 0 при удачном завершении работы функции и -1 при ошибке. Функция sigsuspend всегда возвращает -1, обычно с кодом ошибки EINTR.

Отправка сигнала с "прицепом".

```
#include <signal.h>
int sigqueue(pid_t pid, int sig, const union sigval value);
```

sigqueue() отправляет сигнал, указанный в sig процессу с идентификатором PID, определенном pid. Требуются определенные права для отправки сигнала, такие же как и для kill. Как и в случае с kill, пустой сигнал (null, 0) может использоваться для проверки того, что заданный PID вообще существует.

Аргумент value используется для указания сопутствующих элементов или данных (либо целых либо указателей), отправляемых сигналу, и имеет следующий тип:

```
union sigval {
    int   sival_int;
    void *sival_ptr;
};
```

Если процесс, принимающий сигнал, имеет для него обработчик, используя флаг SA_SIGINFO для sigaction, то он может получить данные через поле si_value структуры siginfo_t передаваемой как второй аргумент для обработчика. Далее, поле si_code этой структуры будет установлено в SI_QUEUE.

При нормальном завершении работы sigqueue() возвращает 0, показывая что сигнал был успешно отправлен получающему процессу. В других случаях будет возвращаться -1 и переменная errno будет установлена соответственно ошибке.

Выделение объектов разделенной памяти.

```
#include <sys/types.h>
#include <sys/mman.h>
int shm_open(const char *name, int oflag, mode_t mode);
int shm_unlink(const char *name);
```

shm_open создает и открывает новый (или открывает уже существующий) объект разделяемой памяти POSIX. Объект разделяемой памяти POSIX - это обработчик, используемый несвязанными процессами для исполнения mmap на одну область разделяемой памяти. Функция shm_unlink выполняет обратную операцию, удаляя объект, предварительно созданный с помощью shm_open.

Операция shm_open аналогична open . name определяет собственно создаваемый объект разделяемой памяти для создания или открытия. Для использования в портируемых программах name должно иметь в начале косую черту (/) и больше не содержать их внутри имени.

oflag является маской битов, созданной через логическое сложение OR одного из флагов O_RDONLY или O_RWDR и любых других флагов, перечисленных далее:

O_RDONLY Открыть объект для чтения. Объект разделяемой памяти, открытый таким образом, может быть обработан mmap только для чтения (PROT_READ).

O_RDWR Открывает объект для чтения и записи.

O_CREAT Создает объект разделяемой памяти, если он еще не существует. Владелец и группа объекта устанавливаются как для open , а биты прав объекта устанавливаются в соответствии с 9 битами mode младшего порядка, за исключением того, что эти биты, установленные во время обработки маски создания режимов файла очищаются у новых объектов.

Новый объект разделяемой памяти изначально имеет нулевую длину, размер объекта можно установить, используя ftruncate. (Новые, только что распределенные байты объекта разделяемой памяти автоматически инициализируются в 0.)

O_EXCL Если также было указано O_CREAT и объект разделяемой памяти с заданным name уже существует, то возвращается ошибка. Проверка наличия объекта и его создание (если он не существует) исполняется автоматически.

O_TRUNC Если объект разделяемой памяти уже существует, то обрезать его до 0 байтов.

При успешном выполнении shm_open возвращает новый описатель файла, ссылающийся на объект разделяемой памяти. Этот описатель файла гарантированно будет описателем файла с самым маленьким номером (не среди предварительно открытых) внутри процесса. Флаг FD_CLOEXEC устанавливается для описателя файла.

Описатель файла обычно используется в последовательных вызовах ftruncate (для вновь созданных объектов) и mmap . После вызова к mmap описатель файла может быть закрыт без влияния на распределение памяти.

Операции shm_unlink аналогичны unlink : удаляется имя объекта разделяемой памяти и, как только все процессы завершили работу с объектом и отменили его распределение, очищают пространство и уничтожают связанную с ним область памяти. После успешного выполнения shm_unlink, попытка исполнить shm_open для

объекта с тем же именем name выдаст ошибку (если только не было указано O_CREAT, в этом случае создается новый, уже другой объект).

При нормальном завершении работы shm_open возвращает неотрицательный описатель файла. При ошибках shm_open возвращает -1. shm_unlink возвращает 0 при нормальном завершении работы и -1 при ошибках.

Отображение объектов разделяемой памяти.

```
#include <unistd.h>
#include <sys/mman.h>
#ifdef _POSIX_MAPPED_FILES
void * mmap(void *start, size_t length, int prot , int flags, int fd, off_t offset);
int munmap(void *start, size_t length);
#endif
```

Функция mmap отражает length байтов, начиная со смещения offset файла (или другого объекта), определенного файловым описателем fd, в память, начиная с адреса start. Последний параметр (адрес) необязателен, и обычно бывает равен 0. Настоящее местоположение отраженных данных возвращается самой функцией mmap, и никогда не бывает равным 0.

Аргумент prot описывает желаемый режим защиты памяти (он не должен конфликтовать с режимом открытия файла). Оно является либо PROT_NONE либо побитовым ИЛИ одного или нескольких флагов PROT_*.

PROT_EXEC (данные в страницах могут исполняться);

PROT_READ (данные можно читать);

PROT_WRITE (в эту область можно записывать информацию);

PROT_NONE (доступ к этой области памяти запрещен).

Параметр flags задает тип отражаемого объекта, опции отражения и указывает, принадлежат ли отраженные данные только этому процессу или их могут читать другие. Он состоит из комбинации следующих битов:

MAP_FIXED Не использовать другой адрес, если адрес задан в параметрах функции. Если заданный адрес не может быть использован, то функция mmap вернет сообщение об ошибке. Если используется MAP_FIXED, то start

должен быть пропорционален размеру страницы. Использование этой опции не рекомендуется.

MAP_SHARED Разделить использование этого отражения с другими процессами, отражающими тот же объект. Запись информации в эту область памяти будет эквивалентна записи в файл. Файл может не обновляться до вызова функций msync или munmap.

MAP_PRIVATE Создать неразделяемое отражение с механизмом copy-on-write. Запись в эту область памяти не влияет на файл. Не определено, являются или нет изменения в файле после вызова mmap видимыми в отраженном диапазоне.

Должно быть задано либо MAP_SHARED, либо MAP_PRIVATE.

offset должен быть пропорционален размеру страницы, получаемому при помощи функции getpagesize .

Memory mapped by mmap is preserved across fork , with the same attributes.

A file is mapped in multiples of the page size. For a file that is not a multiple of the page size, the remaining memory is zeroed when mapped, and writes to that region are not written out to the file. The effect of changing the size of the underlying file of a mapping on the pages that correspond to added or removed regions of the file is unspecified. Системный вызов munmap удаляет все отражения из заданной области памяти, после чего все ссылки на данную область будут вызывать ошибку "неправильное обращение к памяти" (invalid memory reference). Отражение удаляется автоматически при завершении процесса. С другой стороны, закрытие файла не приведет к снятию отражения.

Адрес start должно быть кратен размеру страницы. Все страницы, содержащие часть указанного диапазона, не отображены, и последующие ссылки на эти страницы будут генерировать SIGSEGV. Это не будет являться ошибкой, если указанный диапазон не содержит отображенных страниц. Для отображений 'файл-бэкэнд' поле st_atime отображаемого файла может быть обновлено в любой момент между mmap() и соответствующим снятием отображения; первое обращение к отображенной странице обновит поле, если оно до этого уже не было обновлено.

Поля st_ctime и st_mtime файла, отображенного по PROT_WRITE и MAP_SHARED, будут обновлены после записи в отображенний диапазон, и до вызова последующего msync() с флагом MS_SYNC или MS_ASYNC, если такой случится.

При удачном выполнении mmap возвращает указатель на область с отраженными данными. При ошибке возвращается значение MAP_FAILED (-1), а переменная errno приобретает соответствующее значение. При удачном выполнении munmap возвращаемое значение равно нулю. При ошибке возвращается -1, а переменная errno приобретает соответствующее значение. (Вероятнее всего, это будет EINVAL).

Завдання.

Частина 1. Обробка сигналів*. Скласти програму, яка:*

1. Описує глобальний дескриптор файла логу.
2. Описує функцію-обробник сигналів, прототипу

```
void signal_handler( int signo, siginfo_t *si, void * ucontext );
```

3. У функції-обробнику виконати запис у файлі логу з докладним розкриттям структури `siginfo_t`, яка подана на вхід.
4. Відкриває файл логу на запис.
5. Відмічає в ньому факт власного запуску та свій pid.
6. Описує структуру sigaction, у якій вказує на функцію обробник.
7. Реєструє обробник для сигналу SIGHUP із збереженням попереднього обробника.
8. Переходить до нескінченного циклу із засинанням на кілька секунд та відмітками у файлі логу.
9. Протестувати отриману програму, посилаючи до неї сигнали утілітою kill, та спостерігаючи результат у файлі логу.
10. Пояснити отримані результати

***Частина 2. Розподілена пам`ять**. Скласти програму, яка:*

1. Описує структуру датума, яка містить ціле значення для ідентифікатора процесу, ціле значення для мітки часу та строку фіксованої довжини.
2. Реєструє обїект розподіленої пам`яті через виклик shm_open
3. Приводить його до розміру кратного розміру структури датуму
4. Відображае отриманий об`єкт у пам`ять через показчик на структуру датуму та виклик mmap
5. Переходить до нескінченного циклу у якому:
    a. Запитує строку з клвіатури
    b. Вичитує та презентує вміст структури датуму
    c. Записує у структуру натомість свій ідентификатор процесу, поточний час та отриману строку
6. Протестувати отриману програму, запустивши два її примірники у різних сесїях.

Рекомендації до виконання.

1. Аргумент ucontext у обробнику сигналів може бути проігнорований
2. До параметру sa_flags структури sigaction додати флаг SA_SIGINFO
3. Для приведення розміру розподіленого сегменту використовуйте ftruncate
4. Перед читанням структури датуму з розподіленого сегменту, використовуйте msync
5. Для стеження за останніми записами у логу використовуйте tail -f

Посилання.

- Рекомендована література: №№ 1-3.
- Офіційна документація стандартної бібліотеки Сі 2-й розділ, статті: signal, kill, raise, sigaction, sigqueue, mmap, shm_open, shm_unlink.

Лабораторна робота № 4.
Мережеві сокети. Елементарний сервер масового обслуговування.

Теоретичні відомості.

Сокеты предоставляют мощный и гибкий механизм межпроцессного взаимодействия (IPC). Они могут использоваться для организации взаимодействия программ на одном компьютере, по локальной сети или через Internet, что позволяет вам создавать распределённые приложения различной сложности. Кроме того, с их помощью можно организовать взаимодействие с программами, работающими под управлением других операционных систем. Например, под Windows существует интерфейс Window Sockets, спроектированный на основе socket API.

Сокеты поддерживают многие стандартные сетевые протоколы (конкретный их список зависит от реализации) и предоставляют унифицированный интерфейс для работы с ними. Наиболее часто сокеты используются для работы в IP-сетях. В этом случае их можно использовать для взаимодействия приложений не только по специально разработанным, но и по стандартным протоколам - HTTP, FTP, Telnet и т. д. Например, вы можете написать собственный сервер, способный обслуживать одновременно множество клиентов.

Сокет (socket) - это конечная точка сетевых коммуникаций. Он является чем-то вроде "портала", через которое можно отправлять байты во внешний мир. Приложение просто пишет данные в сокет; их дальнейшая буферизация, отправка и транспортировка осуществляется используемым стеком протоколов и сетевой аппаратурой. Чтение данных из сокета происходит аналогичным образом.

В программе сокет идентифицируется дескриптором - это просто переменная типа int. Программа получает дескриптор от операционной системы при создании сокета, а затем передаёт его сервисам socket API для указания сокета, над которым необходимо выполнить то или иное действие.

Создание сокета.

```
#include <sys/types.h>
#include <sys/socket.h>
int socket(int domain, int type, int protocol);
```

Socket создает конечную точку соединения и возвращает ее описатель.

Параметр domain задает домен соединения: выбирает набор протоколов, которые будут использоваться для создания соединения. Такие наборы описаны в <sys/socket.h>. В настоящее время распознаются такие форматы:

PF_UNIX, PF_LOCAL	Локальное соединение
PF_INET	IPv4 протоколы Интернет
PF_INET6	IPv6 протоколы Интернет
PF_IPX	IPX - протоколы Novell
PF_NETLINK	Устройство для взаимодействия с ядром
PF_X25	Протокол ITU-T X.25 / ISO-8208
PF_AX25	Протокол AX.25 - любительское радио
PF_ATMPVC	ATM - доступ к низкоуровневым PVC
PF_APPLETALK	Appletalk
PF_PACKET	Низкоуровневый пакетный интерфейс

Сокет имеет тип type, задающий семантику коммуникации. В настоящее время определены следующие типы:

SOCK_STREAM Обеспечивает создание двусторонних надежных и последовательных потоков байтов , поддерживающих соединения. Может также поддерживаться механизмвнепоточных данных.

SOCK_DGRAM Поддерживает датаграммы (ненадежные сообщения с ограниченной длиной и не поддерживающие соединения).

SOCK_SEQPACKET Обеспечивает работу последовательного двустороннего канала для передачи датаграмм с поддержкой соединений; датаграммы имеют ограниченную длину; от получателя требуется за один раз прочитать целый пакет.

SOCK_RAW Обеспечивает доступ к низкоуровневому сетевому протоколу.

SOCK_RDM Обеспечивает надежную доставку датаграмм без гарантии, что они будут расположены по порядку.

SOCK_PACKET Устарел и не должен использоваться в новых программах.

Некоторые типы сокетов могут быть не включены в определенные наборы протоколов; например, SOCK_SEQPACKET не включен в набор AF_INET.

Параметр protocol задает конкретный протокол, который работает с сокетом. Обычно существует только один протокол, задающий конкретный тип сокета в определенном семействе протоколов, в этом случае protocol может быть определено, как 0. Однако, возможно существование нескольких таких протоколов (в этом случае и используется данный параметр). Номер протокола зависит от используемого ``домена коммуникации".

Сокеты типа SOCK_STREAM являются соединениями полнодуплексных байтовых потоков, похожими на каналы. Они не сохраняют границы записей. Потоковый сокет должен быть в состоянии соединения перед тем, как из него можно будет отсылать данные или принимать их в нем. Соединение с другим сокетом создается с помощью системного вызова connect . После соединения данные можно передавать, с помощью системных вызовов read , write или одного из вариантов следующих системных вызовов: send , recv . Когда сеанс закончен, выполняется команда close . Внепоточные данные могут передаваться, как описано в send , а приниматься, как описано в recv .

Коммуникационные протоколы, которые реализуют SOCK_STREAM, следят, чтобы данные не были потеряны или дублированы. Если часть данных, для которых имеется место в буфере протокола, не может быть передана за определенное время, соединение считается "мертвым". Когда в сокете включен флаг SO_KEEPALIVE, протокол каким-либо способом проверяет, не отключена ли еще другая сторона. Сигнал SIGPIPE появляется, если процесс посылает или принимает данные, пользуясь "разорванным" потоком; это приводит к тому, что процессы, не

обрабатывающие сигнал, завершаются. Сокеты SOCK_SEQPACKET используют те же самые системные вызовы, что и сокеты SOCK_STREAM. Единственное отличие в том, что вызовы read возвращают только запрошенное количество данных, а остальные данные пришедшего пакета не будут учитываться. Границы сообщений во входящих датаграммах сохраняются.

Сокеты SOCK_DGRAM и SOCK_RAW позволяют посылать датаграммы принимающей стороне, заданной при вызове send . Датаграммы обычно принимаются с помощью вызова recvfrom , который возвращает следующую датаграмму с соответствующим обратным адресом.

SOCK_PACKET - это устаревший тип сокета, позволявший получать необработанные пакеты прямо от драйвера устройства. Используйте вместо него packet .

Системный вызов fcntl с аргументом F_SETOWN может использоваться для задания группы процессов, которая будет получать сигнал SIGURG, когда прибывают внепоточные данные; или сигнал SIGPIPE, когда соединение типа SOCK_STREAM неожиданно обрывается. Этот вызов также можно использовать, чтобы задать процесс или группу процессов, которые получают асинхронные уведомления о событиях ввода-вывода с помощью SIGIO. Использование F_SETOWN эквивалентно использованию вызова ioctl с аргументом FIOSETOWN или SIOCSPGRP.

Когда сеть сообщает модулю протокола об ошибке (например, в случае IP, используя ICMP-сообщение), то для сокета устанавливается флаг ожидающей ошибки. Следующая операция этого сокета вернет код ожидающей ошибки. Некоторые протоколы позволяют очереди ошибок в сокете получить детальную информацию об ошибке.

В случае ошибки возвращается -1; в противном случае возвращается описатель, ссылающийся на сокет.

Операции сокетов контролируются их параметрами. Эти параметры описаны в <sys/socket.h>. Функции setsockopt и getsockopt используются, чтобы установить и получить необходимые параметры соответственно.

Манипуляции файловым дескриптором.

```
#include <unistd.h>
#include <fcntl.h>
int fcntl(int fd, int cmd);
int fcntl(int fd, int cmd, long arg);
int fcntl(int fd, int cmd, struct flock *lock);
```

fcntl выполняет одну из различных дополнительных операций над файловым дескриптором fd. Эта операция определяется содержимым аргумента cmd.

Управление close-on-exec

F_DUPFD Ищет наименьший доступный номер файлового дескриптора, который больше arg и делает его копией дескриптора fd. Фактически это другая форма вызова dup2 которая используется с явным указанием файлового дескриптора. Старый и новый дескрипторы могут использоваться равнозначно. Они разделяют одни и те же блокировки, указатели на позиции в файле и флаги; например, если позция в файле изменяется с помощью lseek для одного из дескрипторов, то эта же позиция также будет изменена и для другого. Однако, данные два дескриптора не разделяют флаг close-on-exec. Флаг close-on-exec в копии выключен. Это означает, что дескриптор не будет закрыт в случае вызова exec. При успешном выполнении этой операции, возвращается новый файловый дескриптор.

F_GETFD Читает флаг close-on-exec. Если бит FD_CLOEXEC установлен в 0, то файл будет оставлен открытым при вызове exec, в противном случае он будет закрыт.

F_SETFD Устанавливает флаг close-on-exec в значение, заданное битом FD_CLOEXEC аргумента arg.

Флаги состояния файла

Любой файловый дескриптор имеет несколько связанных с ним флагов, которые инициализируются вызовом open и, возможно, изменяются затем вызовом fcntl . Эти флаги разделяются между копиями (сделанными с помощью dup , fork , и других вызовов) этого же файлового дескриптора.

Эти флаги и их смысл описываются на странице руководства open .

F_GETFL Читает флаги файлового дескриптора.

F_SETFL Устанавливает часть флагов, относящихся к состоянию файла, согласно значению, указанному в аргументе arg. Оставшиеся биты (режим доступа, флаги создания файла) в значении arg игнорируются. В Linux данная команда может изменять только флаги O_APPEND, O_NONBLOCK, O_ASYNC и O_DIRECT.

Совместная (advisory) блокировка

F_GETLK, F_SETLK и F_SETLKW используются для установки, снятия и тестирования существующих блокировок записи (также известных как блокировки сегмента файла или области файла). Третий аргумент lock является указателем на структуру, которая имеет по крайней мере следующие поля (в произвольном порядке).

```
struct flock {
    ...
    short l_type;    /* Тип блокировки: F_RDLCK,
                        F_WRLCK, F_UNLCK */
    short l_whence;  /* Как интерпретировать l_start:
                        SEEK_SET, SEEK_CUR, SEEK_END */
    off_t l_start;   /* Начальное смещение для блокировки */
    off_t l_len;     /* Количество байт для блокировки */
    pid_t l_pid;     /* PID процесса блокирующего нашу блокировку
                        (F_GETLK only) */
    ...
};
```

Поля l_whence, l_start и l_len этой структуры задают диапазон байт, который мы хотим заблокировать. l_start - это начальное смещение для блокировки, которое интерпретируется как: начало файла (если значение l_whence установлено в SEEK_SET); текущая позиция в файле (если значение l_whence установлено в SEEK_CUR); или конец файла (если значение l_whence установлено в SEEK_END). В последних двух случаях, l_start может иметь отрицательное значение, предоставляя смещение, которого не может быть перед началом файла. l_len -это неотрицательное целое число, которое задаёт количество байт, которые будут

заблокированы. Байты следующие после конца файла могут быть заблокированы, но это нельзя сделать для байтов, которые находятся перед началом файла. Значение 0 для l_len имеет специальное назначение: блокировка всех байтов, начиная от позиции, заданной l_whence и l_start до конца файла, не зависимо от того, насколько велик файл. Поле l_type может быть использовано для указания типа блокировки: чтение (F_RDLCK) или запись (F_WDLCK). Блокировку на чтение (разделяемая блокировка) на область файла может удерживать любое количество процессов, но только один процесс может удерживать блокировку на запись (эксклюзивная блокировка). Любая эксклюзивная блокировка исключает все другие блокировки, как разделяемые так и эксклюзивные. Один процесс может удерживать только один тип блокировки области файла; если происходит новая блокировка на уже заблокированную область, то существующая блокировка преобразуется в новый тип блокировки. (Такие преобразования могут привести к разбиению, уменьшению или срастанию с существующей блокировкой, если диапазон байт, заданный для новой блокировки неточно совпадает с диапазоном существующей блокировки.)

F_SETLK Установить блокировку (когда l_type установлено в значение F_RDLCK или F_WRLCK) или снять блокировку (когда l_type установлено в значение F_UNLCK) или байты, заданны с помощью полей l_whence, l_start и l_len структуры lock. Если конфликтующая блокировка удерживается другим процессом, то данный вызов вернёт -1 и установит значение errno в EACCES или EAGAIN.

F_SETLKW Если устнаовлен F_SETLK, но для файла удерживается конфликтующая блокировка, то происходит ожидание снятия блокировки. Если во время ожидания поступил сигнал, то данный вызов прерывается и (после возврата из обработчика сигнала) из него происходит немедленный возрат (где возвращаемое значение установлено в -1, а errno установлено в значение EINTR).

F_GETLK При входе в этот вызов, lock описывает блокировку, которую мы должны бы установить на файл. Если такая блокировка не может быть установлена, fcntl() по факту не устанавливает её, но возвращает F_UNLCK в поле l_type структуры lock и оставляет другие поля структуры неизменёнными. Если одна

или более несовместимых блокировок мешают установке нашей блокировки, то fcntl() возвращает подробности об одной из этих блокировок в полях l_type, l_whence, l_start и l_len структуры lock и устанавливает l_pid в значение PID того процесса, который удерживает блокировку. Для того, чтобы установить блокировку на чтение, fd должен быть открыт на чтение. Для того, чтобы установить блокировку на запись, fd должен быть открыт на запись. Чтобы установить оба типа блокировки, дескриптор должен быть открыт на запись и на чтение. Также, как и при снятии блокировки через явное указание F_UNLCK, блокировка автоматически снимается, когда процесс завершается или если он закрывает любой файловый дескриптор, ссылающийся на файл, на котором удерживается блокировка. Это плохо: это означает, что процесс может потерять блокировки на файлах типа /etc/passwd или /etc/mtab, когда по какой-либо причине библиотечная функция производит их открытие, чтение и закрытие. Блокировки не наследуются процессом-потомком, созданным через fork , но сохраняются при вызове execve . Поскольку буферизация, выполняется через библиотеку stdio , использование блокировок с функциями в этом пакете нужно избегать; вместо этих функций используйте read и write .

Обязательная (mandatory) блокировка

Совместные блокировки не являются принудительными и полезны только между сотрудничающими процессами. Обязательные блокировки являются принудительными для всех процессов.

Управление сигналами

F_GETOWN, F_SETOWN, F_GETSIG и F_SETSIG используются для управления сигналами доступности ввода/вывода:

F_GETOWN Получить идентификатор процесса или группу процесса, которые в текущий момент принимают сигналы SIGIO и SIGURG для событий на файловом дескрипторе fd. Группы процессов возвращаются в виде отрицательных значений.

F_SETOWN Установить идентификатор процесса или группу процесса, которые будут принимать сигналы SIGIO и SIGURG для событий на

файловом дескрипторе fd. Группы процессов задаются в виде отрицательных значений. (F_SETSIG может использоваться, чтобы задать другой сигнал вместо SIGIO). Если вы установили на файловый дескриптор флаг состояния O_ASYNC (или через предоставление этого флага в вызове open , или используя команду F_SETFL вызова fcntl), то сигнал SIGIO посылается всякий раз, когда для данного файлового дескриптора становится возможным ввод или вывод. Процесс или группа процесса, для приёма сигнала могут быть выбраны, используя команду F_SETOWN в вызове fcntl. Если файловым дескриптором является сокет, то для него также выбирается получатель сигналов SIGURG, которые доставляются, когда на сокет поступает данных больше, чем его пропускная способность. (SIGURG посылается во всех ситуациях, когда вызов select говорит, что сокет находится в состоянии "исключительной ситуации"). Если файловый дескриптор соответствует терминальному устройству, то группе процессов, которые работают с терминалом не в фоновом режиме будут посылаться сигналы SIGIO.

F_GETSIG Получить сигнал, который будет послан, когда станет возможен ввод или вывод. Значение 0 означает сигнал SIGIO. Любое другое значение (включая SIGIO) является другим сигналам, посылаемым вместо SIGIO и в это случае, для обработчика сигнала доступна дополнительная информация, если он был установлен с SA_SIGINFO.

F_SETSIG Установить сигнал, который будет послан, когда станет возможен ввод или вывол. Значение 0 означает сигнал SIGIO. Любое другое значение (включая SIGIO) является другим сигналам, посылаемым вместо SIGIO и в это случае, для обработчика сигнала доступна дополнительная информация, если он был установлен с SA_SIGINFO. В случае использования F_SETSIG с ненулевым значением и установкой SA_SIGINFO для обработчика сигнала обработчику передаётся дополнительная информация о событиях ввода/вывода в структуре siginfo_t. Если поле si_code показывает, что сигналом является SI_SIGIO, то поле si_fd содержит файловый дескриптор, ассоциированный с собрытием вывода/вывода. В противном случае, не существует никакого механизма, чтобы сообщить с каким файловым дескриптором связан полученный сигнал и вы должны

использовать (select , poll , read с установленным флагом O_NONBLOCK и т.д.) чтобы определить какой файловый дескриптор доступен для ввода/вывода. При выборе сигнала реального времени (value >= SIGRTMIN) как описано в POSIX.1b, в очередь будут добавляться несколько событий ввода вывода, с теми же номерами сигналов. (Размер очереди зависит от доступной памяти). Дополнительная информация будет доступна как описано выше, если для обработчика сигнала будет установлено SA_SIGINFO.

Используя эти механизмы, программа может реализовать полностью асинхронный ввода/вывод без использования в своей работе select или poll .

Использование O_ASYNC, F_GETOWN, F_SETOWN является специфичным для BSD и Linux. F_GETSIG и F_SETSIG являются специфичными для Linux. POSIX описывает асинхронный ввод/вывод и структуру aio_sigevent используемую для сходных вещей; которые также доступны в Linux как часть библиотеки GNU C (Glibc).

Аренда

F_SETLEASE и F_GETLEASE (в Linux 2.4 и выше) используются (соответственно) для установки и получения текущих настроек вызывающего процесса, арендующего файл, на который указывает fd. Аренда файла предоставляет такой механизм, когда процесс, удерживающий аренду ("держатель аренды") уведомляется (через доставку сигнала), о том, что другой процесс ("конкурент") пытается выполнить open или truncate для этого файла.

F_SETLEASE Устанавливает или удаляет аренду файла, в соответствии со целым значением, установленным в аргументе arg:

_RDLCK Установить аренду чтения. Это приведёт к генерации уведомления, когда другой процесс открывает указанный файл для записи или усекает его.

_WRLCK Установить аренду записи. Это приведёт к генерации уведомления, когда другой процесс открывает указанный файл (для чтения или записи) или усекает его. Аренда записи может быть установлена на файл, только если этот файл не открыт в этот момент каким-либо другим процессом.

_UNLCK Удалить аренду с указанного файла. Процесс может удерживать для файла только один тип аренды. Аренда может быть установлена только для обычных файлов. Непривелегированный процесс может установить аренту только для файла, у которого UID совпадает с UID файловой системы этого процесса.

F_GETLEASE Показывает какой тип аренды удерживается для файла, на который указывает fd и возвращает одно из значений F_RDLCK, F_WRLCK или F_UNLCK, соответственно показывающих, что вызывающий процесс удерживает аренду чтения, записи или что аренды нет. (Третий аргумент fcntl() при этому опускается.)

Когда конкурент выполняет вызов open() или truncate(), который конфликтует с арендой, установленной через F_SETLEASE, системный вызов блокируется ядром (за исключением случая, когда при вызове open() указывается флаг O_NONBLOCK; в этом случае немедленно возвращается ошибка EWOULDBLOCK). Затем ядро уведомляет держателя аренды, отправляя ему сигнал (по умолчанию SIGIO). Держатель аренты должен при получении этого сигнала выполнить все необходимые действия для подготовки этого файла к использованию другим процессом (например, сбросить буферы кэша) и затем удалить аренду, выполнив операцию F_SETLEASE и установив значение аргумента arg в F_UNLCK.

Если держатель аренды не освободит аренду в течении количества секунд, которое задаётся в файле /proc/sys/fs/lease-break-time, а системный вызов конкурента останется блокирующим (то, есть конкурент не установит флаг O_NONBLOCK для системного вызова open() и этот системный вызов не будет прерван обработчиком события), то ядро принудительно удалит аренду у процесса держателя аренды.

После того как аренда снята держателем аренды или принудительно удалена ядром, ядро снимает блокировку с системного вызова процесса конкуренту и разрешает продолжить его работу.

По умолчанию, для уведомления держателя аренды используется сигнал SIGIO, но его можно изменить, используя команду F_SETSIG для fcntl (). Если выполняется команда F_SETSIG (даже назначая сигнал SIGIO) и при этом

обработчик сигнала устанавливается с использованием SA_SIGINFO, то обработчик будет получит в качестве второго аргумента структуру siginfo_t, в которой поле si_fd будет содержать дескриптор файла, для которого установлена аренда и которому пытается получить доступ другой процесс. (Это полезно, если вызывающий процесс удерживает аренду на несколько файлов).

Уведомления об изменении файла и каталога

F_NOTIFY Предоставляет уведомление, когда изменяется каталог, на который указывает fd или файлы, которые в нём содержатся. События, о наступлении которых делается уведомление, задаются в аргументе arg, который является битовой маской, получаемой битовым сложением (OR) одной или более следующих масок:

DN_ACCESS

DN_MODIFY Файл был изменён (write, pwrite, writev, truncate, ftruncate)

DN_CREATE Файл был создан (open, creat, mknod, mkdir, link, symlink, rename)

DN_DELETE Файл был удалён (unlink, rename в другой каталог, rmdir)

DN_RENAME Файл был переименован внутри каталога (rename)

DN_ATTRIB У файла был изменён атрибут (chown, chmod, utime[s])

Уведомления об изменении состояния каталога обычно однократные и приложение должно перерегистрировать установку уведомлений, чтобы и дальше получать их. Однако, если в аргумент arg добавить маску DN_MULTISHOT, то уведомления будут происходить до тех пор, пока не будут явно отменены.

Серии запросов F_NOTIFY накапливаются, таким образом запросы на уведомления, указываемые в arg будут добавляться к тем, что уже установлены. Чтобы запретить уведомления для всех событий, выполните вызов F_NOTIFY, в котором значение arg установлено в 0.

Уведомления выполняются через доставку сигнала. По умолчанию - это SIGIO, но вы можете изменить его, используя команду F_SETSIG для вызова fcntl(). В последнем случае, обработчик сигнала получит в качестве второго аргумента

структуру siginfo_t (если обработчик был установлен с использованием SA_SIGINFO), а поле si_fd в этой структуре будет содержать дескриптор файла, для которого было сегенерировано уведомление (полезно, когда устанавливается уведомление для нескольких каталогов).

Кроме того, когда используется DN_MULTISHOT, для уведомлений должен бы быть использован сигнал реального времени POSIX.1b, так что множественные уведомления могут быть поставлены в очередь.

При успешном вызове fcntl, возвращаемое значение зависит от использованной операции:

F_DUPFD Новый файловый дескриптор.

F_GETFD Значение флага.

F_GETFL Значение флага.

F_GETOWN Значение, представляющее собой владельца дескриптора.

F_GETSIG Значение сигнала, посылаемого когда становится возможным чтение или запись или ноль для традиционного поведения SIGIO.

Все другие команды. Ноль.

В случае ошибки, возвращается -1 и значение errno устанавливается соответствующим образом.

Манипулирование параметрами сокетов.

```
#include <sys/types.h>
#include <sys/socket.h>
int getsockopt(int s, int level, int optname, void *optval, socklen_t *optlen);
int setsockopt(int s, int level, int optname, const void *optval, socklen_t optlen);
```

getsockopt и setsockopt манипулируют флагами, установленными на сокете. Флаги могут существовать на нескольких уровнях протоколов; они всегда присутствуют на самом верхнем из них.

При манипулировании флагами сокета должен быть указан уровень, на котором находится этот флаг, и имя этого флага. Для манипуляции флагами на

уровне сокета level задается как SOL_SOCKET. Для манипуляции флагами на любом другом уровне этим функциям передается номер соответствующего протокола, управляющего флагами. Например, для указания, что флаг должен интерпретироваться протоколом TCP, в параметре level должен передаваться номер протокола TCP; смотри описание getprotoent .

Параметры optval и optlen используются в функции setsockopt для доступа к значениям флагов. Для getsockopt они задают буфер, в который нужно поместить запрошенное значение. Для getsockopt параметр optlen передается по ссылке. При вызове он содержит размер буфера, на который указывает параметр optval, а после вызова -- реальный размер возвращенного значения. Если значение флага не используется, то параметр optval может быть NULL.

optname и все указанные флаги без изменений передаются для интерпретации соответствующему модулю протоколов. Файл <sys/socket.h> содержит определения флагов уровня сокета, описанные ниже. Флаги на других уровнях протоколов различаются по формату и по имени. Обращайтесь к соответствующим пунктам секции 4 руководства.

Большинство флагов уровня сокета используют тип int для параметра optval. Для функции setsockopt, параметр должен быть ненулевым, чтобы установить флаг логического типа, или нуль, чтобы сбросить этот флаг.

Описание доступных флагов сокетов находится в и соответствующих протоколам страницах руководства.

В случае успеха возвращается ноль. При ошибке возвращается -1, а значение errno устанавливается должным образом.

Привязывание (позиционирование) сокета.

```
#include <sys/types.h>
#include <sys/socket.h>
int bind(int sockfd, struct sockaddr *my_addr, socklen_t addrlen);
```

bind привязывает к сокету sockfd локальный адрес my_addr длиной addrlen. Традиционно, эта операция называется "присваивание сокету имени". Когда сокет

только что создан с помощью socket , он существует в пространстве имён (семействе адресов), но не имеет назначенного имени.

Обычно сокету типа SOCK_STREAM требуется назначить локальный адрес с помощью bind, перед тем, как он сможет принимать соединения.

В случае успеха возвращается ноль. При ошибке возвращается -1, а errno устанавливается должным образом.

Правила, используемые при привязке имён, разные в разных семействах адресов.

Для сокетов типа AF_INET (сетевых), методика следующая:

```
#include <sys/socket.h>
#include <netinet/in.h>
tcp_socket = socket(PF_INET, SOCK_STREAM, 0);
raw_socket = socket(PF_INET, SOCK_RAW, protocol);
udp_socket = socket(PF_INET, SOCK_DGRAM, protocol);
```

IP-сокет создается с помощью вызова функции socket в виде socket(PF_INET, socket_type, protocol). Возможными типами сокета являются SOCK_STREAM, чтобы открыть сокет tcp , SOCK_DGRAM чтобы открыть сокет udp , или SOCK_RAW, чтобы открыть сокет raw для прямого доступа к IP протоколу. протокол - это IP-протокол, указанный в IP-заголовке, который будет принят или отослан. Единственные возможные значения для параметра протокол - это 0 или IPPROTO_TCP для TCP сокетов, 0 или IPPROTO_UDP для UDP сокетов. Для SOCK_RAW можно указать зарегистрированный в IANA IP-протокол, один из тех, что описаны в RFC1700.

Если процесс хочет принимать новые входящие пакеты или соединения, то он должен связать сокет с адресом локального интерфейса, используя bind . Только один IP-сокет может быть связан с каждой заданной локальной парой (адрес, порт). Если при вызове bind указать INADDR_ANY, то сокет будет связан со всеми локальными интерфейсами. Если listen или connect вызываются для несвязанного сокета, то он будет автоматически привязан к выбранному наугад свободному порту, а в качестве локального адреса будет указан INADDR_ANY.

Адрес локального TCP-сокета, который был связан, будет недоступен в течение некоторого времени после его закрытия, если только не был установлен флаг SO_REUSEADDR. Следует проявлять осторожность при использовании этого флага, поскольку он делает TCP менее надежным.

Адрес IP сокета определяется как комбинация адреса IP интерфейса и номера порта. Сам по себе IP протокол не присваивает номера портов, они реализуются протоколами более высокого уровня, типа udp и tcp . У сокетов типа raw переменная sin_port содержит протокол IP.

```
struct sockaddr_in {
   sa_family_t    sin_family; /* семейство адресов: AF_INET */
   u_int16_t      sin_port;   /* порт сокета в сетевом порядке байт */
   struct in_addr  sin_addr;  /* адрес в интернете */
};
struct in_addr {
  u_int32_t      s_addr;     /* адрес сокета в сетевом порядке байт */
};
```

Значение переменной sin_family всегда равно AF_INET. Это обязательно; в Linux 2.2 большая часть сетевых функций возвращает код ошибки EINVAL, если это условие не выполняется. Переменная sin_port содержит порт сокета в сетевом порядке байт. Порты, номера которых меньше 1024, называются зарезервированными портами. Только процессы с фактическим идентификатором пользователя 0 или со способностью CAP_NET_BIND_SERVICE могут быть связаны с этими сокетами с помощью bind . Заметьте, что у чистого протокола IPv4, как такового, нет понятия порта, они реализуются только протоколами более высокого уровня, типа tcp и udp.

Значением переменной sin_addr является адрес IP-хоста. Переменная addr, являющаяся членом структуры struct in_addr, содержит адрес сокета в сетевом формате. Работать со структурой in_addr следует только посредством библиотечных функций inet_aton, inet_addr, inet_makeaddr или напрямую с помощью преобразователя имен. Адреса IPv4 делятся на одиночные, широковещательные и

групповые. Каждый одиночный адрес указывает на один интерфейс хоста, широковещательные адреса указывают на все хосты в сети, а групповые адреса соответствуют всем хостам в группе. Датаграммы могут посылаться по широковещательным адресам только если установлен флаг SO_BROADCAST. В текущей реализации сокетам, ориентированным на соединения, разрешено иметь только одиночные адреса.

Значения адреса и порта всегда хранятся в сетевом формате. В частности, это означает, что требуется вызывать htons для числа, обозначающего порт. Все функции из стандартной библиотеки, манипулирующие с адресами/портами, работают с сетевым форматом.

Есть несколько специальных адресов: INADDR_LOOPBACK (127.0.0.1) всегда приписывается локальному хосту через закольцовывающий интерфейс; INADDR_ANY (0.0.0.0) означает любой адрес для связывания; INADDR_BROADCAST (255.255.255.255) означает любой хост и по историческим причинам при связывании создает тот же эффект, что и INADDR_ANY.

IP поддерживает некоторые опции сокета, относящиеся к протоколу, которые могут быть установлены с помощью setsockopt , и прочитаны с помощью getsockopt . Параметр "уровень опции сокета" этих функций равен SOL_IP. Двоичный флаг со значением нуль означает "ложь", другие значения -- "истина".

IP_OPTIONS Устанавливает или возвращает те опции IP, которые посылаются с каждым пакетом из данного сокета. Аргументами являются указатель на область памяти, содержащую эти опции, и размер опции. Системный вызов setsockopt устанавливает опции IP, связанные с сокетом. Для IPv4 максимальный размер этой опции равен 40 байтам. Все возможные опции перечислены в RFC791. Если запрос, устанавливающий соединение с сокетом типа SOCK_STREAM, содержит опции IP, то такие же IP-опции (с инвертированными заголовками маршрутизации) будут использоваться в этом сокете. Входящие пакеты не могут изменять опции после того, как соединение установлено. По умолчанию обработка всех опций, связанных с маршрутизацией по отправителю, отключена, но ее можно включить, используя sysctl-значение accept_source_route. Другие опции, например

связанные с временными отметками, продолжают обрабатываться. Для датаграммных сокетов опции IP могут быть установлены только локальным пользователем. В результате вызова getsockopt с параметром IP_OPTIONS текущие опции IP, используемые при отправки пакетов, будут помещены в указанный буфер.

IP_PKTINFO Передает служебное сообщение IP_PKTINFO, содержащее структуру pktinfo, которая содержит некоторую информацию о входящем пакете. Эта опция используется только для сокетов, ориентированных на посылку датаграмм. Аргумент является флагом, который сообщает сокету, нужно ли посылать сообщение IP_PKTINFO или нет. Само сообщение может быть послано/получено только в виде контрольного сообщения с пакетом, используя recvmsg или sendmsg .

```
struct in_pktinfo {
      unsigned int   ipi_ifindex;  /* указатель на интерфейс */
      struct in_addr ipi_spec_dst; /* локальный адрес */
      struct in_addr ipi_addr;     /* адрес назначения из заголовка */
};
```

ipi_ifindex - это уникальный указатель на интерфейс, от которого был получен этот пакет. ipi_spec_dst это локальный адрес пакета, а ipi_addr это адрес назначения, указанный в заголовке пакета. Если опция IP_PKTINFO передана sendmsg , то исходящий пакет будет послан через интерфейс, указанный в ipi_ifindex, по адресу из ipi_spec_dst.

IP_RECVTOS Если включена, то вместе с исходящими пакетами передается вспомогательное сообщение IP_TOS. Оно содержит байт, который определяет поле Тип Сервиса/Приоритет в заголовке пакета. Ожидается логический целочисленный флаг.

IP_RECVTTL Если этот флаг установлен, то в поле Время Жизни (time to live) получаемого пакета, как байт, передается управляющее сообщение IP_RECVTTL. Не поддерживается сокетами типа SOCK_STREAM.

IP_RECVOPTS Передает пользователю все входящие опции IP, с помощью управляющего сообщения IP_OPTIONS. Заголовок маршрутизации и

другие опции уже установлены для локального хоста. Не поддерживается сокетами типа SOCK_STREAM.

IP_RETOPTS Идентична опции IP_RECVOPTS, но возвращает необработанные опции, причем временные отметки и записи о маршрутизации для этого хопа еще не заполнены.

IP_TOS Устанавливает или получает значение поля Тип-Сервиса (Type-Of-Service (TOS)), которое посылается с каждым IP-пакетом, который отсылается с этого сокета. Это поле используется, чтобы задавать сетевые приоритеты пакетов. TOS хранится в одном байте. Существует несколько стандартных значений флага TOS: IPTOS_LOWDELAY, чтобы минимизировать задержки для передаваемого трафика, IPTOS_THROUGHPUT, чтобы улучшить пропускную способность, IPTOS_RELIABILITY, чтобы увеличить надежность, IPTOS_MINCOST, следует использовать для "необязательных данных", которые можно пересылать на минимальной скорости. Может быть указано не более одного из этих значений TOS. Все другие биты являются недействительными и должны быть обнулены. По умолчанию Linux посылает датаграммы IPTOS_LOWDELAY первыми, но точное поведение зависит от сконфигурированного порядка очередности. Для установки некоторых высокоприоритетных типов сервиса фактический идентификатор пользователя должен быть равен 0, или же у процесса должна быть способность CAP_NET_ADMIN. Приоритеты также можно расставить не зависящим от типа протокола способом, через опции сокета (SOL_SOCKET, SO_PRIORITY).

IP_TTL Устанавливает или получает текущее значение поля Время Жизни (time to live), которое указывается в каждом пакете, который отсылается с этого сокета.

IP_HDRINCL Включение этого флага означает, что пользователь уже добавил заголовок IP в начало своих данных. Применяется только в сокетах типа SOCK_RAW. Если этот флаг включен, то значения, установленные опциями IP_OPTIONS, IP_TTL и IP_TOS, игнорируются.

IP_MTU Возвращает используемое в данный момент значение MTU маршрута текущего сокета. Эта опция используется только если сокет установил соединение. Возвращает целое число. Значение этой опции можно получить только через getsockopt .

IP_ROUTER_ALERT Передает этому сокету все пакеты, которые пересылаются с опцией IP Router Alert. Эта опция используется только в сокетах типа raw. Она может быть полезна, например, для демонов RSVP, запущенных на пользовательском уровне. Перехваченные пакеты не пересылаются ядром: ответственность за их повторную отсылку лежит на пользователе. Связывание сокета игнорируется, такие пакеты фильтруются только протоколом. В качестве аргумента использует целочисленный флаг.

Инициация исходящего соединения на сокете.

```
#include <sys/types.h>
#include <sys/socket.h>
int connect(int sockfd, const struct sockaddr *serv_addr, socklen_t addrlen);
```

Файловый дескриптор sockfd должен ссылаться на сокет. Если сокет имеет тип SOCK_DGRAM, значит, адрес serv_addr является адресом по умолчанию, куда посылаются датаграммы, и единственным адресом, откуда они принимаются. Если сокет имеет тип SOCK_STREAM или SOCK_SEQPACKET, то данный системный вызов попытается установить соединение с другим сокетом. Другой сокет задан параметром serv_addr, являющийся адресом длиной addrelen в пространстве коммуникации сокета. Каждое пространство коммуникации интерпретирует параметр serv_addr по-своему.

Обычно сокеты с протоколами, основанными на соединении, могут устанавливать соединение только один раз; сокеты с протоколами без соединения могут использовать connect многократно, чтобы изменить адрес назначения. Сокеты без поддержки соединения могут прекратить связь с другим сокетом, установив член sa_family структуры sockaddr в AF_UNSPEC.

Если соединение или привязка прошла успешно, возвращается нуль. При ошибке возвращается -1, а errno устанавливается должным образом.

Принятие входящего соединения на сокете.

```
#include <sys/types.h>
#include <sys/socket.h>
int accept(int s, struct sockaddr *addr, socklen_t *addrlen);
```

Функция accept используется с сокетами, ориентированными на устанавление соединения (SOCK_STREAM, SOCK_SEQPACKET и SOCK_RDM). Эта функция извлекает первый запрос на соединение из очереди ожидающих соединений, создаёт новый подключенный сокет почти с такими же параметрами, что и у s, и выделяет для сокета новый файловый дескриптор, который и возвращается. Новый сокет более не находится в слушающем состоянии. Исходный сокет s не изменяется при этом вызове. Заметим, что флаги файловых дескрипторов (те, что можно установить с помощью параметра F_SETFL функции fcntl, типа неблокированного состояния или асинхронного ввода-вывода) не наследуются новым файловым дескриптором после accept.

Аргумент s - это сокет, который был создан с помощью socket , привязан к локальному адресу с помощью bind , и слушает соединения после listen .

Аргумент addr - это указатель на структуру sockaddr. В эту структуру помещается адрес другой стороны, в том виде, в каком он известен на коммуникационном уровне. Точный формат адреса, передаваемого в параметре addr, определяется "семейством" сокета. Аргумент addrlen является параметром, передаваемым по ссылке: перед вызовом он содержит размер структуры, на которую ссылается addr, а после вызова~-- действительную длину адреса в байтах. Если addr равен NULL, он не заполняется.

Если в очереди нет запросов на соединение, и на сокет не установлен флаг, что он является неблокирующим, accept блокирует вызвавшую программу до появления соединения. Если сокет является неблокирующим, а в очереди нет запросов на соединение, то accept возвращает EAGAIN.

Для того, чтобы получать уведомления о входящих сооединениях на сокете, можно использовать select или poll . В этом случае, когда придёт запрос на новое соединение, будет доставлено событие "можно читать", и после этого вы можете вызвать accept, чтобы получить сокет для этого соединения. Можно также настроить сокет так, чтобы он посылал сигнал SIGIO, когда на нём происходит какая-либо активность.

Этот системный вызов возвращает -1 в случае ошибки. При успешном завершении возвращается неотрицательное целое, являющееся дескриптором сокета.

Завдання.

Частина 1. Побудова мережевого сервера. *Скласти програму, яка:*

1. Виконує демонізацію, подальший функціонал стосується демона.
2. Описує глобальний дескриптор файла логу.
3. Описує структуру *struct sockaddr_in* з параметрами: формат сокетів - PF_INET, адреса - будь яка, порт - 3200 + номер варіанта.
4. Формує сокет типу SOCK_STREAM формату PF_INET.
5. Налаштовує сокет на очікування запитів за допомогою *bind.*
6. Запускає нескінченний цикл обробки запитів від клієнтів.
7. На кожний запит виконує fork для породження процесу обробки.
8. Батьківський процес закриває сокет.
9. Процес обробки в нескінченному циклі отримує від клієнта строки за допомогою *recv*, додає до них свій префікс у вигляді поточного часу та власного *pid*, та повертає клієнту за допомогою *send*.
10. Процес обробки завершує обробку даних від клієнта отримавши строку "close".
1. Кожна дія сервера супроводжується відміткою у логу.

Частина 2. Побудова мережевого клієнта. *Скласти програму, яка:*

7. Описує структуру *struct sockaddr_in* з параметрами: формат сокетів - PF_INET, адреса - обчислена за викликом *htonl(INADDR_LOOPBACK)*, порт - 3200 + номер варіанта.
8. Формує сокет типу SOCK_STREAM формату PF_INET.
9. Налаштовує сокет на підключення до сервера за допомогою *connect.*
10. В нескінченному циклі запитує рядки від оператора, передає їх на сервер та друкує отримані відповіді.
11. Робота закінчується після вводу оператором рядка "close" та отримання на нього відповіді від сервера.

Частина 3. Дослідити роботу мережевого сервера.

Дослідження провести запустивши сервер та кілька примірників клієнта у різних сесіях. Проаналізувати лог сервера та зробити висновки щодо моменту запуску та закриття процесів обробки з'єднань.

Рекомендації до виконання.

1. Порти сокетів формувати за допомогою htons.
2. Адреси сокетів формувати за допомогою htonl.
3. Обробляти помилки при виклику socket, bind, accept, connect. Виводити діагностику.
4. Додатково вивчити принцип побудови серверів на неблокуючих сокетах (O_NONBLOCK) з використанням select.
5. Для стеження за останніми записами у логу використовуйте tail -f

Посилання.

- Рекомендована література: №№ 1-3.
- Офіційна документація стандартної бібліотеки Сі 2-й розділ, статті: socket, fcntl, bind, listen, connect, accept, send, recv.

Перелік рекомендованої літератури

1. Роберт Лав, "Linux. Системное программирование",- СПб:БХВ-Питер,2008
2. Керниган Б.В., Пайк Р. UNIX - универсальная среда программирования: пер. с англ.- М.: Финансы и статистика,1992.-304 с.
3. Кейт Хэвиленд, Дайна Грэй, Бен Салама " Системное программирование в UNIX. Руководство программиста по разработке ПО": Пер. с англ.- М, ДМК Пресс,2000.
4. Теренс Чан " Системное программирование на C++ для UNIX.":Пер. с англ.- K:BHV, 1999.
5. Андрей Робачевский "Операционная система UNIX".- СПб:БХВ-Санкт-Петербург, 2000.
6. Уильям Стивенс "UNIX: взаимодействие процессов" "- СПб:БХВ-Питер,2002
7. Бах Морис Дж. Архитектура операционной системы UNIX. //THE DESIGN OF THE UNIX OPERATING SYSTEM by Maurice J. Bach// Пер. с англ. к.т.н. Крюкова А.В. Copyright 1986 Корпорация Bell Telephone Laboratoties. Издано корпорацией Prentice-Hall. Отделение Simon & Schuster Энглвуд Клиффс, Нью-Джерси 07632. Серия книг по программному обеспечению издательства Prentice-Hall. Консультант Брайан В. Керниган.