Справочник Linux по курсу “Основы программирования для Linux”

Подготовлено к зачету по МДК 01.04 Системное программирование 03.05.2024

1. Введение в разработку Linux
   1. Основной инструментарий разработчика Linux

Начнем с того что необходимо установить на компьютере, чтобы вести разработку под Linux

* Linux Distribution: Вам понадобится установленная система Linux. Вы можете выбрать дистрибутив, который вам больше всего подходит, такой как Ubuntu, Fedora или Debian.
* Компилятор GCC: Это основной компилятор для Linux, который поддерживает несколько языков программирования, включая C и C++.
* Текстовый редактор или IDE: Вам понадобится текстовый редактор или интегрированная среда разработки (IDE) для написания кода. Некоторые популярные варианты включают Visual Studio Code, Atom, Sublime Text, или более сложные IDE, такие как CLion или Eclipse для C/C++.
* Утилиты командной строки: Набор базовых утилит командной строки, таких как grep, sed, awk, и других, будет очень полезен для работы с текстовыми файлами и скриптования.
* Git: Git - это система контроля версий, которая позволит вам отслеживать изменения в вашем коде и сотрудничать с другими разработчиками.
* Утилиты для отладки: gdb - это основной отладчик для Linux, который может быть очень полезен для поиска и исправления проблем в вашем коде.
* Make и CMake: Это инструменты для автоматизации процесса сборки вашего кода. Они могут автоматически обрабатывать зависимости и компилировать ваш код в правильном порядке.
* Valgrind: Это инструмент для поиска утечек памяти и других проблем с памятью в вашем коде.

Одна из основных команд для работы **- lsb\_release –a**

Команда lsb\_release -a в Linux используется для вывода информации о дистрибутиве Linux, на котором вы работаете. LSB означает Linux Standard Base - объединенный стандарт, который пытается стандартизировать внутреннюю структуру всех дистрибутивов Linux. Флаг -a означает "all", что говорит команде показать всю доступную информацию.

Вы можете получить вывод, который выглядит примерно так:

No LSB modules are available.

Distributor ID: Ubuntu

Description: Ubuntu 20.04.1 LTS

Release: 20.04

Codename: focal

В этом примере вывод говорит нам, что мы работаем на дистрибутиве Ubuntu, версии 20.04.1 LTS, и его кодовое имя - "focal".

Обратите внимание, что команда lsb\_release может не быть установлена в некоторых дистрибутивах Linux по умолчанию. В таком случае ее можно установить с помощью менеджера пакетов вашего дистрибутива.

1.2 Динамические библиотеки и линковка

Деманглинг с использованием C++filt

c++filt - это утилита командной строки в Unix/Linux, которая деманглирует (восстанавливает) имена функций и переменных, "замангленные" компиляторами C++ и Java.

Замангливание - это процесс преобразования идентификаторов программы в уникальные строки для линковщика, который ничего не знает об объектной системе C++.

Деманглинг - это обратный процесс, позволяющий восстановить исходное имя из замангленного.

Вот как вы можете использовать c++filt. Допустим, у вас есть замангленное имя \_Zolhellov.

echo "\_Zolhellov" | c++filt

Вы получите на выходе hello, которое является исходным именем функции.

Это может быть полезно при отладке, когда вы видите в стеке вызовов или в списке символов замангленные имена. Вы можете использовать c++filt, чтобы восстановить исходные имена и лучше понять, что делает ваша программа.

Использование nm для вывода символических имен

nm - это инструмент командной строки в Unix/Linux, который выводит символы из объектного файла. Это полезно для отладки, чтобы убедиться, что программа содержит правильные символы.

Вот пример использования команды nm:

nm my\_program

Эта команда выведет список всех символов в объектном файле my\_program.

Сборка динамической библиотеки с помощью GCC

GCC - это компилятор, который может использоваться для создания статических и динамических библиотек в Linux. Для создания динамической библиотеки нужно использовать опцию --shared.

Вот пример использования GCC для сборки динамической библиотеки:

gcc -shared -o libmylib.so mylib.c

В этом примере mylib.c - это исходный файл C, который вы хотите скомпилировать в динамическую библиотеку. Результатом будет файл libmylib.so, который является динамической библиотекой.

## Прототипы функций и закодированные имена в C++

Прототип функции в C++ - это объявление функции, которое указывает компилятору о имени, возвращаемом типе и параметрах функции. Прототипы функций очень важны, так как они позволяют компилятору правильно обрабатывать вызовы функций.

Вот пример прототипа функции:

int add(int a, int b); // Прототип функции

Закодированные (или "замангленные") имена - это способ, которым компиляторы C++ кодируют дополнительную информацию о функциях в их именах, включая количество, типы и порядок параметров. Это позволяет компилятору поддерживать перегрузку функций, где одно и то же имя функции может быть использовано для разных функций, отличающихся только типами и/или количеством параметров.

Например, функция add выше может быть заманглена как \_Z3addii в зависимости от компилятора. Здесь Z3addii означает функцию add, принимающую два целочисленных параметра.

Важно отметить, что разные компиляторы могут использовать разные схемы манглинга, и манглированные имена не предназначены для чтения или понимания людьми. Вместо этого они могут быть деманглированы обратно в исходные имена с помощью утилиты, такой как c++filt, обсуждаемой выше.

Теория создания и использования динамических библиотек в C

У нас есть задача: Вам необходимо создать разделяемую динамическую библиотеку **libsolution.so** в которой реализовать функцию со следующим прототипом:

int stringStat(const char \*string, size\_t multiplier, int \*count);

Функция возвращает значение длины строки string, умноженное на multiplier, и увеличивает на 1 значение, на которое указывает count.

Представление решения

Решение предоставляется в виде двух файлов solution.c и Makefile, в последнем предполагается цель по умолчанию, которая приводит к сборке Вашей библиотеки.

Вывод

Программа (функция в библиотеке) ничего не выводит на консоль

Решение:

В файле solution.c, реализуется функция stringStat, которая принимает указатель на строку, множитель и указатель на счетчик.

#include <string.h>

int stringStat(const char \*string, size\_t multiplier, int \*count) {

if (string == NULL || count == NULL) {

return -1; // Error: invalid input

}

size\_t length = strlen(string);

\*count += 1;

return length \* multiplier;

}

Здесь, функция stringStat сначала проверяет, являются ли входные данные валидными. Если указатель на строку или указатель на счетчик равен NULL, функция возвращает -1 как код ошибки.

Затем, функция вычисляет длину строки с помощью функции strlen, увеличивает значение счетчика на 1 и возвращает произведение длины строки и множителя.

Makefile:

Makefile - это файл, который содержит набор инструкций (или "правил"), который make использует для сборки проекта.

CC = gcc

CFLAGS = -Wall -Werror -fpic

LIBRARY = libsolution.so

SOURCES = solution.c

OBJECTS = $(SOURCES:.c=.o)

all: $(LIBRARY)

$(LIBRARY): $(OBJECTS)

$(CC) -shared -o $@ $^

%.o: %.c

$(CC) $(CFLAGS) -c -o $@ $<

clean:

rm -f $(LIBRARY) $(OBJECTS)

В этом Makefile, CC обозначает компилятор, который будет использоваться для сборки проекта, в данном случае это gcc. CFLAGS - это флаги компиляции, которые будут использоваться.

LIBRARY обозначает имя создаваемой библиотеки, а SOURCES обозначает исходные файлы, которые должны быть скомпилированы. OBJECTS - это промежуточные объектные файлы, которые создаются при компиляции исходных файлов.

Цель all указывает make на то, что должно быть создано в конечном итоге. Зависимости этого правила указывают на то, что для создания библиотеки сначала нужно создать объектные файлы.

Правило с $(LIBRARY): $(OBJECTS) говорит make, как создать библиотеку из объектных файлов.

Правило %.o: %.c обозначает, как из .c файлов получить .o файлы.

Правило clean используется для удаления всех созданных файлов, чтобы можно было начать сборку заново.

1.3 Runtime линковка. Libdl

Компиляция и линковка в GCC

GCC - это компилятор GNU для C и C++ и других языков программирования. Эта команда используется для компиляции и линковки программы.

Разберем команду gcc main.c -fPIC -ldl -o loader:

gcc: это сам компилятор.

main.c: это исходный файл, который вы хотите скомпилировать.

-fPIC: этот флаг говорит компилятору создавать только позиционно-независимый код, или PIC. Позиционно-независимый код важен, когда вы создаете библиотеки, которые могут быть загружены в любое место в памяти, и он должен быть использован при создании динамических библиотек.

-ldl: этот флаг линковщика говорит компилятору, что ваша программа будет использовать динамическую библиотеку загрузки (dl), которая предоставляет функции для динамической загрузки библиотек во время выполнения.

-o loader: флаг -o следует за именем файла, который будет создан после компиляции и линковки. В данном случае, результатом будет исполняемый файл с именем loader.

В итоге, эта команда компилирует исходный файл main.c с опцией для создания позиционно-независимого кода, линкует его с библиотекой dl для динамической загрузки и создает исполняемый файл с именем loader.

Использование extern "C" для отключения мэнглинга имен в C++

В языке C++, когда компилятор встречает функцию, он изменяет её имя в объектном файле для поддержки перегрузки функций - этот процесс называется мэнглингом имен. extern "C" указывает компилятору C++, что следующие функции должны иметь ссылочное связывание по стилю C, отключая тем самым мэнглинг имен.

extern "C" void myFunction(int param) {

// тело функции

}

Приведение указателей при использовании dlsym

dlsym - это функция, которая используется для загрузки символа (обычно функции или переменной) из динамической библиотеки. Она возвращает void\*, поэтому программисту нужно вручную привести этот указатель к типу вызываемой функции.

typedef void (\*functionType)(int);

functionType myFunction = (functionType)dlsym(handle, "myFunctionName");

Использование LD\_LIBRARY\_PATH для указания места поиска библиотек

LD\_LIBRARY\_PATH - это переменная среды, используемая в Unix-подобных операционных системах. Она указывает, где dlopen и другие функции должны искать динамические библиотеки.

export LD\_LIBRARY\_PATH=/path/to/my/library:$LD\_LIBRARY\_PATH

Эта команда добавляет путь к вашей библиотеке в начало LD\_LIBRARY\_PATH, что означает, что dlopen будет сначала искать библиотеку в этом месте.

1. Файлы и файловые системы
   1. Базовые функции работы с файлами

Системный интерфейс доступа к файлам в Linux

В системе Linux есть множество функций, которые предоставляют прямой интерфейс доступа к файловым системам. Некоторые из них включают:

* open(): Функция open() используется для открытия существующего файла или создания нового файла. Она возвращает файловый дескриптор, который можно использовать для последующих операций с файлом.

int fd = open("file.txt", O\_RDONLY);

* creat(): Функция creat() используется для создания нового файла. Если файл с таким именем уже существует, он будет усечен до нулевой длины. Функция возвращает файловый дескриптор для нового файла.

int fd = creat("newfile.txt", 0644);

* read(): Функция read() используется для чтения данных из файла. Она возвращает количество прочитанных байтов.

char buffer[256];

ssize\_t bytes\_read = read(fd, buffer, sizeof(buffer));

* write(): Функция write() используется для записи данных в файл. Она возвращает количество записанных байтов.

char \*data = "Hello, World!";

ssize\_t bytes\_written = write(fd, data, strlen(data));

* close(): Функция close() используется для закрытия файлового дескриптора, освобождая его для последующего использования.

close(fd);

* lseek(): Функция lseek() используется для изменения позиции чтения/записи в файле.

off\_t new\_pos = lseek(fd, 0, SEEK\_SET); // Перемещает позицию чтения/записи в начало файла

* stat(): Функция stat() используется для получения информации о файле, такой как размер файла, время последнего доступа и так далее.

struct stat file\_info;

stat("file.txt", &file\_info);

Флаги открытия файла в Linux

Флаги открытия файла в Linux определяют режим, в котором файл будет открыт. Они используются в функциях open() и fcntl(). Вот некоторые из наиболее распространенных флагов:

* O\_RDONLY: Файл открывается только для чтения.
* O\_WRONLY: Файл открывается только для записи.
* O\_RDWR: Файл открывается для чтения и записи.
* O\_CREAT: Если файл не существует, он будет создан.
* O\_EXCL: Используется с O\_CREAT, и вызов будет неудачным, если файл уже существует.
* O\_TRUNC: Если файл уже существует и успешно открыт для записи, его длина будет усечена до нуля.
* O\_APPEND: Файл открывается в режиме добавления.

Примеры использования:

// Открыть файл только для чтения

int fd = open("file.txt", O\_RDONLY);

// Создать новый файл для записи

int fd = open("newfile.txt", O\_CREAT | O\_WRONLY, 0644);

// Открыть файл для добавления данных в конец

int fd = open("file.txt", O\_WRONLY | O\_APPEND);

// Создать новый файл, но выйти, если файл уже существует

int fd = open("newfile.txt", O\_CREAT | O\_EXCL | O\_WRONLY, 0644);

2.2 Работа с каталогами и путями

В Linux есть несколько функций и команд для работы с каталогами и путями. Некоторые из них включают:

* pwd: Это команда оболочки, которая выводит текущий рабочий каталог. Например:

$ pwd

/home/user

* cd: Это команда оболочки, которая используется для изменения текущего рабочего каталога. Например, чтобы перейти в домашний каталог, вы можете использовать:

$ cd ~

* mkdir: Это команда оболочки, которая используется для создания нового каталога. Например:

$ mkdir new\_directory

* rmdir: Это команда оболочки, которая используется для удаления каталога. Например:

$ rmdir new\_directory

* ls: Это команда оболочки, которая выводит содержимое каталога. Например:

$ ls /home/user

* chdir(): Это системный вызов в Linux, который используется для изменения текущего рабочего каталога программы. Например:

chdir("/home/user");

* getcwd(): Это системный вызов в Linux, который используется для получения текущего рабочего каталога программы. Например:

char cwd[1024];

getcwd(cwd, sizeof(cwd));

2.3 Ссылки

В Linux системе файлы и ссылки на них играют важную роль. Основные концепции включают жесткие ссылки, символические ссылки и работу с файлами.

1. **Работа с файлами:** Команда cat используется для вывода содержимого файла. Если вы попытаетесь использовать cat на файле, который не существует, например libMylib.so.2.3.1, вы получите ошибку "No such file or directory".
2. **Жесткие ссылки:** Жесткая ссылка - это просто другое имя для существующего файла. В вашем случае, Makefile является жесткой ссылкой на makefile. Это означает, что Makefile и makefile на самом деле являются одним и тем же файлом, просто с разными именами.
3. **Символические ссылки:** Символическая, или мягкая ссылка, это файл, который служит ссылкой на другой файл или каталог. В вашем примере, file2 является символической ссылкой на file1. Это означает, что при доступе к file2, вы фактически получите доступ к содержимому file1.

Важно помнить, что если исходный файл, на который указывает символическая ссылка, будет удален, то символическая ссылка останется, но будет указывать на несуществующий файл (так называемая "мертвая" ссылка). Если исходный файл, на который указывает жесткая ссылка, будет удален, то файл на самом деле не будет удален до тех пор, пока не будет удалена последняя жесткая ссылка на него.

2.4 Файловая система /proc. Построение дерева процессов

В Linux файловая система /proc представляет собой виртуальную файловую систему, которая предоставляет информацию о состоянии операционной системы, включая данные о процессах. Это является основой для построения дерева процессов.

**Файловая система /proc**

Все каталоги в /proc с числовыми именами представляют активные процессы в системе. Номер каталога соответствует идентификатору процесса (PID) в системе. Внутри каждого из этих каталогов есть файлы и подкаталоги, отображающие различные аспекты состояния процесса.

Например, следующая команда отобразит все процессы в системе:

$ ls /proc | grep '^[0-9]'

**Построение дерева процессов**

Дерево процессов можно построить, используя информацию из /proc. В частности, файл /proc/PID/status содержит информацию о родительском процессе (PPID) каждого процесса.

Команда pstree может быть использована для отображения дерева процессов:

$ pstree

Эта команда отобразит иерархию процессов в формате дерева, начиная с init процесса (PID 1).

Важно помнить, что файловая система /proc является виртуальной и не занимает никакого места на диске. Она создается и обновляется динамически ядром при запуске системы и во время работы системы.

2.5 Отладка программ в Linux

Отладка - это важный аспект разработки программного обеспечения. Это процесс поиска и устранения ошибок или сбоев в программе. В ОС Linux для этого доступны различные инструменты.

GDB - GNU Debugger

GDB - это мощный отладчик для программ, написанных на C, C++, и других языках. Он позволяет следить за выполнением программы, останавливать её при достижении определенных точек (брейкпоинтов), изучать и изменять значения переменных.

Пример использования GDB:

$ gcc -g myprog.c -o myprog # Компиляция с флагом -g для добавления отладочной информации

$ gdb myprog # Запуск GDB

(gdb) run # Запуск программы внутри GDB

Valgrind

Valgrind - это инструмент для поиска утечек памяти, ошибок работы с памятью, а также для профилирования программ. Он оценивает использование памяти и может помочь выявить места, где память выделяется, но не освобождается.

Пример использования Valgrind:

$ gcc myprog.c -o myprog # Компиляция программы

$ valgrind ./myprog # Запуск программы с помощью Valgrind

Strace

strace - это утилита для отслеживания системных вызовов и сигналов в Linux. Она может помочь определить, как программа взаимодействует с операционной системой.

Пример использования strace:

$ strace ls # Запуск команды ls с отслеживанием системных вызовов

3. Процессы и потоки

3.1 Жизненный цикл процесса в Linux

В операционной системе Linux, процесс проходит через несколько этапов от момента его создания до завершения. Это общеизвестно как жизненный цикл процесса.

Создание (Forking)

Жизненный цикл процесса начинается с этапа создания. Каждый процесс в Linux создается с помощью системного вызова fork(). Этот вызов создает новый процесс, который является копией родительского процесса.

pid\_t pid = fork();

if(pid == 0)

{

// Этот код будет выполнен дочерним процессом

}

else if(pid > 0)

{

// Этот код будет выполнен родительским процессом

}

else

{

// Fork не удался

}

Исполнение (Executing)

После создания процесса, он начинает исполнять код. Это может быть код, который был унаследован от родительского процесса, или новый код может быть загружен в процесс с помощью системного вызова exec().

Пример:

if(pid == 0)

{

execl("/bin/ls", "ls", "-l", NULL);

}

Завершение (Termination)

В конце своего жизненного цикла процесс завершается. Это может произойти либо нормальным образом, когда процесс закончил исполнение своего кода, либо может быть вызвано внешним сигналом, таким как команда kill.

Пример:

if(pid > 0)

{

int status;

waitpid(pid, &status, 0); // Ожидание завершения дочернего процесса

}

Зомби и Сироты

В некоторых случаях, процесс может войти в состояние "зомби", когда он завершен, но родительский процесс еще не получил его статус завершения.

С другой стороны, если родительский процесс завершается до своего дочернего процесса, дочерний процесс становится "сиротой" и его родительским процессом становится init (процесс с PID 1).

Важно отметить, что жизненный цикл процесса может слегка варьироваться в зависимости от конкретной операционной системы и среды выполнения.

3.2 Создание процессов fork/exec

В Linux для создания новых процессов обычно используются два системных вызова: fork() и exec().

Fork

fork() - это системный вызов, который создает новый процесс, делая его точной копией текущего процесса. Новый процесс называется "дочерним", а исходный - "родительским". Весь контекст родительского процесса, включая значения переменных и регистров процессора, дублируется в дочернем процессе.

После вызова fork(), оба процесса (родительский и дочерний) начинают выполняться параллельно.

Пример использования fork():

#include <unistd.h>

#include <stdio.h>

int main() {

pid\_t pid = fork();

if(pid == 0) {

printf("I am the child process!\n");

} else if(pid > 0) {

printf("I am the parent process!\n");

} else {

printf("Fork failed!\n");

}

return 0;

}

Exec

exec() - это группа системных вызовов, которые заменяют текущий образ процесса новым образом процесса. Это значит, что после успешного вызова exec(), старая программа полностью заменяется новой программой. Новая программа начинает выполняться с функции main(), и никакого кода, следующего за вызовом exec(), не будет выполнено.

Пример использования exec():

#include <unistd.h>

int main() {

char \*args[] = {"/bin/ls", "-l", NULL};

execv(args[0], args);

// Этот код не будет выполнен, если execv() успешно

return 0;

}

Важно помнить, что fork() и exec() часто используются вместе: сначала создается дочерний процесс с помощью fork(), а затем в дочернем процессе вызывается exec(), чтобы заменить его образ новой программой.

3.3 Атрибуты создаваемого процесса. Системный вызов clone

Атрибуты создаваемого процесса

Когда в Linux создается процесс, он наследует некоторые атрибуты от своего родительского процесса. Эти атрибуты включают:

* Идентификатор пользователя (User ID) и идентификатор группы (Group ID)
* Контролирующий терминал
* Набор открытых файлов
* Текущий рабочий каталог
* Окружение (Environment variables)
* Приоритет выполнения (Nice value)

Системный вызов clone

В Linux системный вызов clone() используется для создания нового процесса. Он более гибкий, чем fork(), так как позволяет точно указать, какие ресурсы будут разделены между родительским и дочерним процессом, а какие будут скопированы.

Системный вызов clone() принимает два аргумента: указатель на функцию, которую должен выполнить новый поток, и указатель на область памяти, которая будет использоваться в качестве стека для нового потока.

Пример использования clone():

#define \_GNU\_SOURCE

#include <sched.h>

#include <stdio.h>

#include <stdlib.h>

int child\_func(void \*arg) {

printf("Child process\n");

return 0;

}

int main() {

const int STACK\_SIZE = 65536;

char \*stack = malloc(STACK\_SIZE);

if (!stack) {

perror("Malloc failed");

exit(1);

}

printf("Parent process\n");

clone(child\_func, stack + STACK\_SIZE, SIGCHLD, NULL);

return 0;}

В этом примере, сначала создается новый стек, затем вызывается clone(), который создает новый процесс. В новом процессе выполняется функция child\_func(). Заметьте, что новый процесс будет иметь доступ к тому же адресному пространству, что и родительский процесс, если не указаны другие флаги clone(). Это отличает clone() от fork(), который всегда создает копию адресного пространства.

3.4 Процессы-демоны

В Linux демонами называются процессы, которые работают в фоновом режиме, а не под управлением прямого пользователя.

Демоны обычно запускаются при загрузке системы и продолжают работать до ее выключения. Они используются для выполнения различных служб и обработки системных событий. Например, httpd - это демон, который обрабатывает HTTP-запросы и служит веб-страницы.

Создание процесса-демона

Чтобы процесс стал демоном, он должен отделиться от терминала, который его запустил, и перейти в фоновый режим. Это обычно делается с помощью следующего кода:

if (fork() > 0) exit(0); // Создание дочернего процесса и завершение работы родительского

setsid(); // Создание нового сеанса, в котором дочерний процесс становится лидером

Стандартные демоны Linux

В Linux есть множество стандартных демонов, которые обрабатывают различные задачи. Например:

* init: первый процесс в системе, который запускает все остальные процессы.
* cron: демон, который запускает запланированные задачи в заданное время.
* syslogd: демон, который обрабатывает системные журналы.
* sshd: демон, который обрабатывает входящие SSH-соединения.

Обычно имена демонов в Linux заканчиваются на "d", что означает "демон".

3.5 Задачи на программирование

Задача 1: Разработать программу solution, которая осуществляет поиск родительского PID текущего процесса в файловой системе proc и выводит найденное значение на консоль. Результат, возвращаемый функций должен быть выведен отдельной строкой(должен оканчиваться символом перевода строки \n) в stdio.

Пример вызова

./solution

13221

Решение:

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <string.h>

#define BUFFER\_SIZE 1024

int main() {

pid\_t pid = getpid();

char path[BUFFER\_SIZE];

snprintf(path, BUFFER\_SIZE, "/proc/%d/stat", pid);

FILE \*file = fopen(path, "r");

if (file == NULL) {

fprintf(stderr, "Error opening file\n");

return 1;

}

int ppid;

fscanf(file, "%\*d %\*s %\*c %d", &ppid);

printf("%d\n", ppid);

fclose(file);

return 0;

}

Давайте разберем решение

1. Использование функции getpid(): Ваш код начинается с вызова функции getpid(), которая возвращает идентификатор процесса (PID) текущего процесса. Поскольку каждый процесс в системе имеет уникальный PID, это позволяет нам однозначно идентифицировать процесс.
2. Формирование пути до файла stat: Далее ваш код формирует путь до файла stat текущего процесса в файловой системе proc. Этот путь выглядит как /proc/{PID}/stat, где {PID} - это PID текущего процесса. Файловая система proc - это виртуальная файловая система, которая предоставляет информацию о процессах и других аспектах системы в виде файлов. Файл stat для каждого процесса содержит различную информацию о процессе, включая его PPID, статус, использование ЦП и другое.
3. Открытие файла stat: Далее ваш код открывает файл stat для чтения, используя функцию fopen(). Если файл не может быть открыт, ваш код выводит сообщение об ошибке и завершает выполнение программы.
4. Чтение PPID из файла stat: Затем ваш код читает PPID из файла stat, используя функцию fscanf(). Функция fscanf() используется для чтения форматированных данных из файла. В данном случае, вы используете ее для чтения четвертого поля из файла stat, которое является PPID. Здесь используется \* в спецификаторе формата для пропуска ненужных полей.
5. Вывод PPID: После извлечения PPID, ваш код выводит его в стандартный поток вывода с помощью функции printf(). Отметим, что вывод оканчивается символом новой строки (\n), что важно для корректного отображения результата.
6. Закрытие файла: Наконец, после того как PPID был прочитан и выведен, ваш код закрывает файл stat, используя функцию fclose(). Это важно для освобождения ресурсов, которые были заняты при открытии файла.

В результате, программа успешно извлекает и выводит PPID текущего процесса, используя информацию, доступную в файловой системе proc Linux.

Задача 2:

Разработать программу solution, которая по заданному pid, осуществляет поиск пути в дереве процессов до процесса с идентификатором 1 (init). Для каждого найденного процесса печатается в отдельной строке его идентификатор.

Пример вызова

./solution 1766

1766

1299

465

1

Решение:

#include <stdio.h>

#include <stdlib.h>

int print\_pid(int pid) {

char path[1024];

FILE \*fp;

int ppid;

// Печатаем текущий PID

printf("%d\n", pid);

if (pid == 1) {

return 0;

}

// Открываем файл stat

sprintf(path, "/proc/%d/stat", pid);

fp = fopen(path, "r");

if (fp == NULL) {

fprintf(stderr, "Failed to open /proc/%d/stat\n", pid);

return 1;

}

// Считываем информацию о процессе

fscanf(fp, "%\*d %\*s %\*c %d", &ppid);

fclose(fp);

// Рекурсивно печатаем PPID

return print\_pid(ppid);

}

int main(int argc, char \*argv[]) {

if (argc != 2) {

fprintf(stderr, "Usage: %s <pid>\n", argv[0]);

return 1;

}

return print\_pid(atoi(argv[1]));

}

Данная программа решает задачу поиска пути в дереве процессов от заданного процесса до процесса с идентификатором 1 (init) в операционной системе Linux. Она использует функционал файловой системы /proc для получения информации о процессах. Давайте разберем каждый шаг программы:

1. Функция print\_pid: Эта функция реализует основную логику программы. Она принимает идентификатор процесса (PID), печатает его, а затем рекурсивно вызывает себя для родительского процесса (PPID).
2. Открытие файла stat: Функция формирует путь до файла stat для заданного PID, затем открывает этот файл для чтения. Если файл не может быть открыт (например, если процесса с таким PID не существует), функция выводит сообщение об ошибке и завершает выполнение с кодом ошибки 1.
3. Чтение PPID из файла stat: Функция считывает PPID из файла stat, затем закрывает файл. PPID это идентификатор родительского процесса.
4. Рекурсивный вызов функции для PPID: Функция рекурсивно вызывает себя для PPID, что позволяет ей пройти вверх по дереву процессов.
5. Основная функция main: Основная функция проверяет аргументы командной строки, а затем вызывает функцию print\_pid для заданного PID. Если количество аргументов командной строки не равно двум (имя программы и PID), функция выводит сообщение об ошибке и завершает выполнение с кодом ошибки 1

В результате, при запуске программы с PID в качестве аргумента, она распечатает путь в дереве процессов от заданного процесса до процесса с PID 1.

Задача 3: Разработать программу solution, которая по заданному pid, осуществляет поиск всех потомков процесса с этим идентификатором и выводит их количество (включая процесс с заданным pid).

Решение

#include <stdio.h>

#include <dirent.h>

#include <string.h>

#include <stdlib.h>

static int child\_count(const char \*pid) {

char path[1024];

DIR \*dir;

struct dirent \*entry;

int count = 1;

sprintf(path, "/proc/%s/task", pid);

if (!(dir = opendir(path))) return 0;

while ((entry = readdir(dir)) != NULL) {

FILE \*fp;

char line[1024];

if (entry->d\_name[0] == '.') continue;

sprintf(path, "/proc/%s/task/%s/children", pid, entry->d\_name);

if (!(fp = fopen(path, "r"))) continue;

while (fscanf(fp, "%s", line) == 1) {

count += child\_count(line);

}

fclose(fp);

}

closedir(dir);

return count;

}

int main(int argc, char \*argv[]) {

printf("%d\n", child\_count(argv[1]));

return 0;

}

Эта программа решает задачу подсчета всех потомков процесса с заданным идентификатором процесса (PID) в операционной системе Linux. Она использует функционал файловой системы /proc для получения информации о процессах. Давайте разберем каждый шаг программы:

1. Функция child\_count: Эта функция реализует основную логику программы. Она принимает PID процесса, затем рекурсивно обходит все дочерние процессы и считает их, включая сам процесс.
2. Открытие директории процесса: Функция открывает директорию /proc/{PID}/task, содержащую информацию о всех потоках (задачах) в процессе. Если директория не может быть открыта, функция возвращает 0.
3. Обход директории процесса: Функция обходит все файлы в директории процесса. Для каждого потока она открывает файл children, который содержит список дочерних процессов этого потока.
4. Подсчет дочерних процессов: Функция считывает PID каждого дочернего процесса из файла children и рекурсивно вызывает себя для каждого из них. Результат каждого вызова добавляется к общему количеству процессов.
5. Основная функция main: Основная функция вызывает функцию child\_count для заданного PID и выводит полученное количество процессов. Если количество аргументов командной строки не равно двум (имя программы и PID), функция выводит сообщение об ошибке и завершает выполнение с кодом ошибки 1.

В результате, при запуске программы с PID в качестве аргумента, она подсчитывает и выводит количество всех потомков заданного процесса, включая сам процесс.

1. Взаимодействия процессов

4.1 Виды межпроцессного взаимодействия в Linux

Межпроцессное взаимодействие (Inter-Process Communication, IPC) в Linux обеспечивает способы обмена данными между процессами. Это важно для координации действий и обмена информацией между процессами. Ниже приведены различные виды межпроцессного взаимодействия в Linux:

* Каналы (Pipes): Это самый базовый способ межпроцессного взаимодействия. Каналы обеспечивают однонаправленную связь между родительским и дочерним процессами. Данные, записанные в канал на одном конце, могут быть прочитаны на другом конце.

Пример использования канала:

int fd[2];

pipe(fd);

if (fork() == 0) {

close(fd[1]); // Закрыть записывающий конец в дочернем процессе

dup2(fd[0], STDIN\_FILENO); // Перенаправить STDIN на чтение из канала

// Дочерний процесс может теперь читать данные из канала

} else {

close(fd[0]); // Закрыть читающий конец в родительском процессе

write(fd[1], "Hello, Child!", 14); // Записать данные в канал

}

* Очереди сообщений (Message Queues): Очереди сообщений позволяют процессам обмениваться дискретными сообщениями. Каждое сообщение имеет свой тип, который может быть использован для приоритизации сообщений.
* Семафоры (Semaphores): Семафоры - это средство синхронизации, которое позволяет контролировать доступ к общим ресурсам. Процесс может "захватить" семафор перед доступом к ресурсу и "освободить" его после. Если семафор уже захвачен, другие процессы будут ждать, пока он не освободится.
* Разделяемая память (Shared Memory): Это самый быстрый способ межпроцессного взаимодействия. Процессы могут совместно использовать область памяти, что позволяет им обмениваться данными без необходимости копирования.
* Сокеты (Sockets): Сокеты позволяют обмениваться данными между процессами, даже если они работают на разных машинах. Сокеты поддерживают как однонаправленную (UDP), так и двунаправленную (TCP) связь.

Эти механизмы IPC позволяют процессам в Linux эффективно взаимодействовать друг с другом, обмениваясь данными и синхронизируя свои действия.

4.2 Каналы

Каналы (Pipes) в Linux – это форма межпроцессного взаимодействия, позволяющая одному процессу передавать данные другому процессу. Они особенно полезны в ситуациях, когда один процесс генерирует данные, которые должны быть обработаны другим процессом.

В Linux существуют два типа каналов:

1. Анонимные каналы (Unnamed pipes): Это самый простой тип каналов, который обеспечивает однонаправленную связь между родительским и дочерним процессами. Родительский процесс создает канал, а затем порождает дочерний процесс. Один процесс пишет в канал, а другой читает из него.

Пример использования анонимного канала:

int pipefd[2];

pipe(pipefd);

if (fork() == 0) {

close(pipefd[1]); // Закрыть записывающий конец в дочернем процессе

// Дочерний процесс может теперь читать данные из канала

} else {

close(pipefd[0]); // Закрыть читающий конец в родительском процессе

write(pipefd[1], "Hello, Child!", 14); // Записать данные в канал

}

1. Именованные каналы (Named pipes или FIFOs): Именованные каналы позволяют обеспечивать связь между не родственными процессами. Они представляют собой специальные файлы в файловой системе, которые обеспечивают канал связи между процессами.

Пример использования именованного канала:

mkfifo("/tmp/myfifo", 0666); // Создать именованный канал с именем myfifo

char buf[256];

int fd = open("/tmp/myfifo", O\_RDONLY);

read(fd, buf, sizeof(buf)); // Прочитать данные из именованного канала

close(fd);

Основное отличие между этими двумя типами каналов заключается в том, что анонимные каналы существуют только в памяти и исчезают после завершения общающихся процессов, в то время как именованные каналы существуют в файловой системе и не исчезают после завершения процессов.