СОДЕРЖАНИЕ:

1. Создайте файл сценария оболочки с названием get\_open\_file\_flags.sh.
2. Определение флагов открытия файла с использованием файловой системы proc в Linux
3. Получение родительского идентификатора процесса (PPID) через файловую систему proc в Linux
4. Извлечение и вывод списка дочерних процессов определенного процесса в Linux
5. Просмотр версии ядра Linux с использованием файловой системы proc
6. Определение времени работы системы с использованием файловой системы proc в Linux
7. Создание динамической библиотеки из одного файла C
8. У вас есть два файла: functions1.c и functions2.c.

functions1.c содержит функцию для суммирования двух чисел, а functions2.c - функцию для умножения двух чисел.

Ваша задача - создать одну динамическую библиотеку из этих двух файлов.

1. Подсчет процессов в поддереве процессов в Linux
2. Подсчет активных процессов пользователя в Linux
3. Использование мягких ссылок (символических ссылок) в Linux
4. Использование мягких ссылок (символических ссылок) в Linux для работы с директориями
5. Использование каналов (pipes) для передачи чисел между процессами
6. Использование каналов (pipes) для общения между процессами
7. Создание Makefile для компиляции программы с несколькими файлами C
8. У вас есть программа на C, которая состоит из трех файлов: main.c, functions.c и functions.h. main.c и functions.c используют функции, определенные в functions.h. Ваша задача - создать Makefile, который будет компилировать эту программу
9. Использование функции fork для создания дочернего процесса
10. Использование функции fork для выполнения разных задач в родительском и дочернем процессах
11. Использование ключевого слова extern для обращения к глобальной переменной, определенной в другом исходном файле
12. Создание дочернего процесса с помощью clone, который выводит сообщение
13. Создание нового процесса с помощью clone, который исполняет внешнюю программу
14. Запуск внешней программы из C-программы.Exec.
15. Readdir
16. Создать жесткую ссылку для файла
17. Процессы-демоны
18. Отключение буферизации для стандартного вывода
19. Использование nm
20. Runtime линковка
21. Create
22. Подсчет числа процессов с заданным именем
23. Создайте файл сценария оболочки с названием get\_open\_file\_flags.sh.

Этот сценарий оболочки извлекает флаги открытия файла для каждого открытого файла определенного процесса и выводит их на экран.

2. Чтобы выполнить этот сценарий, откройте терминал Linux и выполните следующие команды:

chmod +x get\_open\_file\_flags.sh

./get\_open\_file\_flags.sh [PID]

Замените [PID] на идентификатор процесса, для которого вы хотите получить флаги открытия файла.

После выполнения этих команд вы должны увидеть флаги открытия файла для каждого открытого файла указанного процесса.

\*\*\*

Задача: Проверка состояния блокировки открытых файлов с использованием файловой системы proc в Linux

Создайте сценарий оболочки для Linux, который получает идентификатор процесса (PID) в качестве аргумента и выводит состояние блокировки каждого открытого файла этого процесса, используя файловую систему proc.

#!/bin/bash

# Проверяем, был ли предоставлен аргумент

if [ $# -ne 1 ]

then

echo "Пожалуйста, предоставьте PID в качестве аргумента."

exit 1

fi

# Получаем PID из аргумента

pid=$1

# Проверяем, существует ли каталог /proc/$pid/fd

if [ ! -d "/proc/$pid/fd" ]

then

echo "Процесс с PID $pid не существует."

exit 1

fi

echo "Состояние блокировки открытых файлов для процесса с PID $pid:"

for fd in /proc/$pid/fdinfo/\*

do

lock\_status=$(awk '/^lock:/ {print $2}' $fd)

echo "Файловый дескриптор: $(basename $fd), Состояние блокировки: $lock\_status"

done

1. Задача: Определение флагов открытия файла с использованием файловой системы proc в Linux

Создайте сценарий оболочки для Linux, который получает идентификатор процесса (PID) в качестве аргумента и выводит флаги открытия файла для каждого открытого файла этого процесса, используя файловую систему proc.

1. Создайте файл сценария оболочки с названием get\_file\_lock\_status.sh.

Этот сценарий оболочки извлекает состояние блокировки каждого открытого файла определенного процесса и выводит его на экран.

2. Чтобы выполнить этот сценарий, откройте терминал Linux и выполните следующие команды:

chmod +x get\_file\_lock\_status.sh

./get\_file\_lock\_status.sh [PID]

Замените [PID] на идентификатор процесса, для которого вы хотите получить состояние блокировки открытых файлов.

После выполнения этих команд вы должны увидеть состояние блокировки каждого открытого файла указанного процесса.

#!/bin/bash

# Проверяем, был ли предоставлен аргумент

if [ $# -ne 1 ]

then

echo "Пожалуйста, предоставьте PID в качестве аргумента."

exit 1

fi

# Получаем PID из аргумента

pid=$1

# Проверяем, существует ли каталог /proc/$pid/fd

if [ ! -d "/proc/$pid/fd" ]

then

echo "Процесс с PID $pid не существует."

exit 1

fi

echo "Флаги открытия файла для процесса с PID $pid:"

for fd in /proc/$pid/fd/\*

do

if [ -L $fd ]

then

flags=$(ls -l $fd | awk '{print $NF}')

echo "Файловый дескриптор: $(basename $fd), Флаги: $flags"

fi

done

1. Задача: Получение родительского идентификатора процесса (PPID) через файловую систему proc в Linux

Создайте сценарий оболочки для Linux, который получает идентификатор процесса (PID) в качестве аргумента и возвращает идентификатор родительского процесса (PPID), используя файловую систему proc.

1. Создайте файл сценария оболочки с названием get\_ppid.sh.

Этот сценарий оболочки считывает PPID из файла /proc/$pid/status и выводит его на экран.

2. Чтобы выполнить этот сценарий, откройте терминал Linux и выполните следующие команды:

chmod +x get\_ppid.sh

./get\_ppid.sh [PID]

Замените [PID] на идентификатор процесса, для которого вы хотите получить PPID.

После выполнения этих команд вы должны увидеть PPID указанного процесса.

#!/bin/bash

# Проверяем, был ли предоставлен аргумент

if [ $# -ne 1 ]

then

echo "Пожалуйста, предоставьте PID в качестве аргумента."

exit 1

fi

# Получаем PID из аргумента

pid=$1

# Извлекаем и выводим список дочерних процессов

echo "Дочерние процессы для PID $pid:"

for child\_pid in $(ps -o pid --ppid $pid --no-headers)

do

echo $child\_pid

done

1. Задача: Извлечение и вывод списка дочерних процессов определенного процесса в Linux

Создайте сценарий оболочки для Linux, который получает идентификатор процесса (PID) в качестве аргумента и выводит список всех дочерних процессов этого процесса,

используя файловую систему proc.

1. Создайте файл сценария оболочки с названием get\_child\_processes.sh.

Этот сценарий оболочки извлекает список дочерних процессов определенного процесса и выводит его на экран.

2. Чтобы выполнить этот сценарий, откройте терминал Linux и выполните следующие команды:

chmod +x get\_child\_processes.sh

./get\_child\_processes.sh [PID]

Замените [PID] на идентификатор процесса, для которого вы хотите получить список дочерних процессов.

После выполнения этих команд вы должны увидеть список дочерних процессов для указанного PID.

#!/bin/bash

# Проверяем, был ли предоставлен аргумент

if [ $# -ne 1 ]

then

echo "Пожалуйста, предоставьте PID в качестве аргумента."

exit 1

fi

# Получаем PID из аргумента

pid=$1

# Считываем PPID из /proc/$pid/status

ppid=$(awk '/PPid:/ {print $2}' /proc/$pid/status)

echo "PPID процесса $pid: $ppid"

1. Задача: Просмотр версии ядра Linux с использованием файловой системы proc

Создайте сценарий оболочки для Linux, который выводит версию ядра системы, используя файловую систему proc.

1. Создайте файл сценария оболочки с названием get\_uptime.sh.

Этот сценарий оболочки считывает время работы системы из файла /proc/uptime и выводит его в минутах.

2. Чтобы выполнить этот сценарий, откройте терминал Linux и выполните следующие команды:

chmod +x get\_uptime.sh

./get\_uptime.sh

После выполнения этих команд вы должны увидеть время работы вашей системы Linux в минутах.

#!/bin/bash

# Считываем время работы системы из /proc/uptime

uptime\_seconds=$(awk '{print $1}' /proc/uptime)

uptime\_minutes=$(awk 'BEGIN {print int('$uptime\_seconds' / 60)}')

echo "Время работы системы: $uptime\_minutes минут"

1. Задача: Определение времени работы системы с использованием файловой системы proc в Linux

Создайте сценарий оболочки для Linux, который выводит время работы системы в минутах, используя файловую систему proc.

1. Создайте файл сценария оболочки с названием get\_kernel\_version.sh.

Этот сценарий оболочки считывает версию ядра из файла /proc/version и выводит ее на экран.

2. Чтобы выполнить этот сценарий, откройте терминал Linux и выполните следующие команды:

chmod +x get\_kernel\_version.sh

./get\_kernel\_version.sh

После выполнения этих команд вы должны увидеть версию ядра вашей системы Linux.

#!/bin/bash

# Считываем версию ядра из /proc/version

kernel\_version=$(awk '{print $3}' /proc/version)

echo "Версия ядра Linux: $kernel\_version"

1. Создание динамической библиотеки из одного файла C

У вас есть файл с именем functions.c, который содержит функцию для суммирования двух чисел. Ваша задача - создать динамическую библиотеку из этого файла.

1. Скомпилируйте functions.c в объектный файл с помощью следующей команды:

gcc -c -fPIC functions.c -o functions.o

2. Сохраните объектный файл functions.o как динамическую библиотеку libfunctions.so следующей командой:

gcc -shared -o libfunctions.so functions.o

#include <stdio.h>

int multiply(int a, int b) {

return a \* b;

}

\*\*\*

#include <stdio.h>

int add(int a, int b) {

return a + b;

}

1. У вас есть два файла: functions1.c и functions2.c.

functions1.c содержит функцию для суммирования двух чисел, а functions2.c - функцию для умножения двух чисел.

Ваша задача - создать одну динамическую библиотеку из этих двух файлов.

1. Скомпилируйте functions1.c и functions2.c в объектные файлы с помощью следующих команд:

gcc -c -fPIC functions1.c -o functions1.o

gcc -c -fPIC functions2.c -o functions2.o

2. Сохраните объектные файлы functions1.o и functions2.o как динамическую библиотеку libfunctions.so следующей командой:

gcc -shared -o libfunctions.so functions1.o functions2.o

#include <stdio.h>

int add(int a, int b) {

return a + b;

}

1. Задача: Подсчет процессов в поддереве процессов в Linux

Создайте сценарий оболочки для Linux, который получает идентификатор процесса (PID) в качестве аргумента и подсчитывает количество процессов в поддереве этого процесса.

1. Создайте файл сценария оболочки с названием count\_processes.sh.

Этот сценарий оболочки определяет функцию count\_processes, которая рекурсивно подсчитывает количество процессов в поддереве заданного процесса.

2. Чтобы выполнить этот сценарий, откройте терминал Linux и выполните следующие команды:

chmod +x count\_processes.sh

./count\_processes.sh [PID]

Замените [PID] на идентификатор процесса, для которого вы хотите подсчитать подпроцессы.

После выполнения этих команд вы должны увидеть количество процессов в поддереве указанного процесса.

#!/bin/bash

# Функция для подсчета процессов в поддереве

count\_processes() {

local pid=$1

local count=0

# Получаем список дочерних процессов

local children=$(pgrep -P $pid)

# Подсчитываем каждый дочерний процесс и его подпроцессы

for child in $children

do

count=$((count + 1 + $(count\_processes $child)))

done

echo $count

}

# Проверяем, был ли предоставлен аргумент

if [ $# -ne 1 ]

then

echo "Пожалуйста, предоставьте PID в качестве аргумента."

exit 1

fi

# Подсчитываем и выводим количество процессов

echo $(count\_processes $1)

1. Задача: Подсчет активных процессов пользователя в Linux

Создайте сценарий оболочки для Linux, который получает имя пользователя в качестве аргумента и подсчитывает количество активных процессов, запущенных этим пользователем.

1. Создайте файл сценария оболочки с названием count\_user\_processes.sh.

Этот сценарий оболочки использует команды ps и wc для подсчета количества активных процессов, запущенных указанным пользователем.

2. Чтобы выполнить этот сценарий, откройте терминал Linux и выполните следующие команды:

chmod +x count\_user\_processes.sh

./count\_user\_processes.sh [username]

Замените [username] на имя пользователя, для которого вы хотите подсчитать активные процессы.

После выполнения этих команд вы должны увидеть количество активных процессов, запущенных указанным пользователем.

#!/bin/bash

# Проверяем, был ли предоставлен аргумент

if [ $# -ne 1 ]

then

echo "Пожалуйста, предоставьте имя пользователя в качестве аргумента."

exit 1

fi

# Получаем имя пользователя из аргумента

username=$1

# Используем команду ps для получения списка процессов пользователя

# и команду wc для подсчета этих процессов

process\_count=$(ps -U $username --no-headers | wc -l)

echo "Пользователь $username запустил $process\_count процессов."

1. Задача: Использование мягких ссылок (символических ссылок) в Linux

Создайте сценарий оболочки для Linux, который создает файл, а затем создает мягкую ссылку на этот файл.

Сценарий должен записать текст в файл через мягкую ссылку, а затем считать и вывести этот текст на экран.

1. Создайте файл сценария оболочки с названием soft\_link\_dir.sh.

Этот сценарий оболочки создает директорию mydir, затем создает мягкую ссылку mydir\_link на эту директорию.

Затем сценарий создает файл myfile.txt в директории через мягкую ссылку, записывает строку "Привет, мир!" в файл и выводит содержимое файла на экран.

2. Чтобы выполнить этот сценарий, откройте терминал Linux и выполните следующие команды:

chmod +x soft\_link\_dir.sh

./soft\_link\_dir.sh

После выполнения этих команд вы должны увидеть сообщение "Привет, мир!" в консоли.

#!/bin/bash

# Создаем файл

touch myfile.txt

# Создаем мягкую ссылку на файл

ln -s myfile.txt mylink

# Записываем текст в файл через мягкую ссылку

echo "Привет, мир!" > mylink

# Считываем и выводим текст из файла

cat myfile.txt

1. Задача: Использование мягких ссылок (символических ссылок) в Linux для работы с директориями

Создайте сценарий оболочки для Linux, который создает директорию, а затем создает мягкую ссылку на эту директорию.

Сценарий должен создать файл в директории через мягкую ссылку, записать текст в этот файл и затем считать и вывести этот текст на экран.

1. Создайте файл сценария оболочки с названием soft\_link.sh.

Этот сценарий оболочки создает файл myfile.txt, затем создает мягкую ссылку mylink на этот файл.

Затем сценарий записывает строку "Привет, мир!" в файл через мягкую ссылку и выводит содержимое файла на экран.

2. Чтобы выполнить этот сценарий, откройте терминал Linux и выполните следующие команды:

chmod +x soft\_link.sh

./soft\_link.sh

После выполнения этих команд вы должны увидеть сообщение "Привет, мир!" в консоли.

#!/bin/bash

# Создаем директорию

mkdir mydir

# Создаем мягкую ссылку на директорию

ln -s mydir mydir\_link

# Создаем файл в директории через мягкую ссылку

touch mydir\_link/myfile.txt

# Записываем текст в файл

echo "Привет, мир!" > mydir\_link/myfile.txt

# Считываем и выводим текст из файла

cat mydir/myfile.txt

1. Задача: Использование каналов (pipes) для передачи чисел между процессами

Создайте программу на языке C для Linux, которая создает дочерний процесс с помощью функции fork.

Родительский процесс должен передать дочернему процессу через канал все числа от 1 до 100.

Дочерний процесс должен принять эти числа, вывести их на экран и подсчитать их сумму.

Чтобы скомпилировать и выполнить эту программу, откройте терминал Linux и выполните следующие команды:

gcc pipe\_numbers.c -o pipe\_numbers

./pipe\_numbers

После выполнения этих команд вы должны увидеть все числа от 1 до 100, а также их сумму, в консоли.

#include <stdio.h>

#include <string.h>

#include <unistd.h>

#include <sys/types.h>

#define BUFFER\_SIZE 50

#define READ\_END 0

#define WRITE\_END 1

int main(void) {

char write\_msg[BUFFER\_SIZE] = "Привет, дочерний процесс!";

char read\_msg[BUFFER\_SIZE];

int fd[2];

pid\_t pid;

if (pipe(fd) == -1) {

fprintf(stderr, "Pipe failed");

return 1;

}

pid = fork();

if (pid < 0) {

fprintf(stderr, "Fork failed");

return 1;

}

if (pid > 0) {

close(fd[READ\_END]);

write(fd[WRITE\_END], write\_msg, strlen(write\_msg) + 1);

close(fd[WRITE\_END]);

} else {

close(fd[WRITE\_END]);

read(fd[READ\_END], read\_msg, BUFFER\_SIZE);

printf("Received string: %s", read\_msg);

close(fd[READ\_END]);

}

return 0;

}

1. Задача: Использование каналов (pipes) для общения между процессами

Создайте программу на языке C для Linux, которая создает дочерний процесс с помощью функции fork.

Родительский процесс должен передать строку "Привет, дочерний процесс!" дочернему процессу через канал.

Дочерний процесс должен принять эту строку и вывести ее на экран.

Чтобы скомпилировать и выполнить эту программу, откройте терминал Linux и выполните следующие команды:

gcc pipe\_communication.c -o pipe\_communication

./pipe\_communication

После выполнения этих команд вы должны увидеть сообщение "Received string: Привет, дочерний процесс!" в консоли.

#include <stdio.h>

#include <unistd.h>

#include <sys/types.h>

#define READ\_END 0

#define WRITE\_END 1

int main(void) {

int fd[2];

pid\_t pid;

if (pipe(fd) == -1) {

fprintf(stderr, "Pipe failed");

return 1;

}

pid = fork();

if (pid < 0) {

fprintf(stderr, "Fork failed");

return 1;

}

if (pid > 0) {

close(fd[READ\_END]);

for (int i = 1; i <= 100; i++) {

write(fd[WRITE\_END], &i, sizeof(i));

}

close(fd[WRITE\_END]);

} else {

close(fd[WRITE\_END]);

int sum = 0;

for (int i = 0; i < 100; i++) {

int num;

read(fd[READ\_END], &num, sizeof(num));

printf("%d ", num);

sum += num;

}

printf("\nСумма чисел: %d\n", sum);

close(fd[READ\_END]);

}

return 0;

}

1. **Задача 2: Создание Makefile для компиляции программы с несколькими файлами C**

У вас есть программа на C, которая состоит из трех файлов: main.c, functions.c и functions.h. main.c и functions.c используют функции, определенные в functions.h. Ваша задача - создать Makefile, который будет компилировать эту программу. Добавьте в ваш Makefile цель test, которая будет компилировать и запускать тесты для вашего проекта.

CC=gcc

CFLAGS=-I.

all: main

main: main.o functions.o

$(CC) -o main main.o functions.o

main.o: main.c functions.h

$(CC) -c main.c

functions.o: functions.c functions.h

$(CC) -c functions.c

test: main

./main

clean:

rm -f \*.o main

1. У вас есть программа на C, которая состоит из трех файлов: main.c, functions.c и functions.h. main.c и functions.c используют функции, определенные в functions.h. Ваша задача - создать Makefile, который будет компилировать эту программу.

CC=gcc

CFLAGS=-I.

all: main

main: main.o functions.o

$(CC) -o main main.o functions.o

main.o: main.c functions.h

$(CC) -c main.c

functions.o: functions.c functions.h

$(CC) -c functions.c

clean:

rm -f \*.o main

1. Задача: Использование функции fork для создания дочернего процесса

Напишите программу на языке C для Linux, которая создает дочерний процесс с помощью функции fork. Дочерний процесс должен вывести сообщение "Привет, я дочерний процесс!", а родительский процесс должен вывести сообщение "Привет, я родительский процесс!". Чтобы скомпилировать и выполнить эту программу, откройте терминал Linux и выполните следующие команды:

gcc fork\_hello.c -o fork\_hello

./fork\_hello

После выполнения этих команд вы должны увидеть сообщения "Привет, я дочерний процесс!" и "Привет, я родительский процесс!" в консоли. Обратите внимание, что порядок сообщений может отличаться в зависимости от того, как операционная система планирует процессы.

#include <stdio.h>

#include <unistd.h>

#include <sys/types.h>

int main() {

pid\_t pid = fork();

if (pid == 0) {

printf("Привет, я дочерний процесс!\n");

} else if (pid > 0) {

printf("Привет, я родительский процесс!\n");

} else {

printf("Ошибка создания процесса\n");

}

return 0;

}

1. Задача: Использование функции fork для выполнения разных задач в родительском и дочернем процессах

Создайте программу на языке C для Linux, которая создает дочерний процесс с помощью функции fork.

Родительский процесс должен выводить все четные числа от 1 до 10, а дочерний процесс должен выводить все нечетные числа от 1 до 10.

Чтобы скомпилировать и выполнить эту программу, откройте терминал Linux и выполните следующие команды:

gcc fork\_numbers.c -o fork\_numbers

./fork\_numbers

После выполнения этих команд вы должны увидеть два набора чисел в консоли: один для четных чисел и другой для нечетных чисел.

Обратите внимание, что порядок этих наборов может отличаться в зависимости от того, как операционная система планирует процессы.

#include <stdio.h>

#include <unistd.h>

#include <sys/types.h>

int main() {

pid\_t pid = fork();

if (pid == 0) {

printf("Нечетные числа от 1 до 10: \n");

for (int i = 1; i <= 10; i += 2) {

printf("%d ", i);

}

printf("\n");

} else if (pid > 0) {

printf("Четные числа от 1 до 10: \n");

for (int i = 2; i <= 10; i += 2) {

printf("%d ", i);

}

printf("\n");

} else {

printf("Ошибка создания процесса\n");

}

return 0;

}

1. Использование ключевого слова extern для обращения к глобальной переменной, определенной в другом исходном файле

Вам нужно создать два исходных файла на языке C. В первом файле (назовем его data.c) вам необходимо определить глобальную переменную с именем globalData и присвоить ей любое значение.

Во втором файле (назовем его main.c) вам нужно обратиться к этой глобальной переменной с использованием ключевого слова extern и вывести ее значение на экран.

Чтобы скомпилировать и выполнить эту программу, вам нужно выполнить следующие команды в терминале Linux:

gcc data.c main.c -o main

./main

После выполнения этих команд вы должны увидеть сообщение "Значение глобальной переменной globalData: 100" в консоли.

int globalData = 100;

#include <stdio.h>

extern int globalData;

int main() {

printf("Значение глобальной переменной globalData: %d\n", globalData);

return 0;

}

Main1

#include <stdio.h>

void sayHello() {

printf("Привет, мир!\n");

}

Чтобы скомпилировать и выполнить эту программу, вам нужно выполнить следующие команды в терминале Linux:

gcc util.c main.c -o main

./main

После выполнения этих команд вы должны увидеть сообщение "Привет, мир!" в консоли.

#include <stdio.h>

extern void sayHello();

int main() {

sayHello();

return 0;

}

1. Создание дочернего процесса с помощью clone, который выводит сообщение

Напишите программу на языке C для Linux, которая создает дочерний процесс с помощью системного вызова clone. Дочерний процесс должен выводить сообщение "Привет, я дочерний процесс!" в консоль.

Чтобы скомпилировать и выполнить эту программу, вам нужно открыть терминал Linux и выполнить следующие команды: gcc clone\_example1.c -o clone\_example1 ./clone\_example1

#define \_GNU\_SOURCE

#include <stdio.h>

#include <sched.h>

#include <unistd.h>

#include <sys/wait.h>

#define STACK\_SIZE (1024 \* 1024)

char child\_stack[STACK\_SIZE];

int child() {

printf("Привет, я дочерний процесс!\n");

return 0;

}

int main() {

pid\_t pid = clone(child, child\_stack + STACK\_SIZE, SIGCHLD, NULL);

waitpid(pid, NULL, 0);

return 0;

}

1. Создание нового процесса с помощью clone, который исполняет внешнюю программу

Вашей задачей является создание программы на языке программирования C для операционной системы Linux, которая будет создавать новый процесс с помощью системного вызова clone.

Этот новый процесс должен быть способен запустить внешнюю программу, например, команду ls, которая выводит список файлов и директорий в текущем каталоге.

Чтобы скомпилировать и выполнить эту программу, вам нужно открыть терминал Linux и выполнить следующие команды:

gcc clone\_example2.c -o clone\_example2

./clone\_example2

После выполнения этих команд вы должны увидеть список файлов и директорий в текущем каталоге, выводимый командой ls, которую запускает новый процесс, созданный командой clone.

#define \_GNU\_SOURCE

#include <sched.h>

#include <stdlib.h>

#include <sys/wait.h>

#include <unistd.h>

#define STACK\_SIZE (1024 \* 1024)

static char child\_stack[STACK\_SIZE];

static int child\_fn() {

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

execv(args[0], args);

return 0;

}

int main() {

pid\_t child\_pid = clone(child\_fn, child\_stack + STACK\_SIZE, CLONE\_NEWPID | SIGCHLD, NULL);

waitpid(child\_pid, NULL, 0);

return 0;

}

1. Задание 1: Запуск внешней программы из C-программы.Exec.

Описание задачи

Программа на языке C, которая использует системный вызов `execvp` для запуска внешней команды `echo` с передачей аргументов. Эта программа демонстрирует использование функций семейства `exec` для выполнения внешних команд из программ на C.

Инструкции

Программа компилируется с помощью Makefile и должна быть запущена на системе с Linux. При успешной работе программа выведет на экран текст "Hello, world!". В случае ошибки будет выведено сообщение об ошибке.

Тестирование

Тестирование программы можно провести, запустив скомпилированный исполняемый файл. Входные данные не требуются. Ожидаемый вывод — "Hello, world!".

#include <unistd.h>

#include <stdio.h>

#include <stdlib.h>

int main() {

char \*args[] = {"echo", "Hello, world!", NULL};

execvp("echo", args);

perror("execvp"); // Сюда программа вернется только в случае ошибки

exit(EXIT\_FAILURE);

}

makeFile:

all: exec\_example

exec\_example: exec\_example.c

@echo "Compiling exec\_example.c to produce exec\_example"

@gcc exec\_example.c -o exec\_example

clean:

@echo "Cleaning up..."

@rm -f exec\_example

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

1. Readdir

Задача: Чтение содержимого директории.

Программа на языке C, которая осуществляет чтение содержимого директории и выводит на экран имена всех файлов в данной директории.

Исходные данные: отсутствуют.

Результат выполнения программы: вывод на экран имен всех файлов в текущей директории.

#include <dirent.h>

#include <stdio.h>

#include <stdlib.h>

int main() {

DIR \*d;

struct dirent \*dir;

d = opendir(".");

if (d) {

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

printf("%s\n", dir->d\_name);

}

closedir(d);

} else {

perror("opendir");

exit(EXIT\_FAILURE);

}

return 0;

}

MakeFile:

all: readdir

readdir: readdir.c

@echo "Compiling readdir.c to produce readdir"

@gcc readdir.c -o readdir

clean:

@echo "Cleaning up..."

@rm -f readdir

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

1. Создать жесткую ссылку для файла

Описание: Программа на языке C, которая создает жесткую ссылку для файла. Жесткая ссылка — это дополнительное имя (путь), по которому можно обращаться к содержимому файла на файловой системе.

Инструкции: Программа компилируется с помощью Makefile и должна быть запущена на системе с поддержкой файловых систем, поддерживающих жесткие ссылки. При успешном выполнении программа выведет сообщение "Жесткая ссылка успешно создана." В случае ошибки будет выведено сообщение об ошибке.

Тестирование: Тестирование программы можно провести, запустив скомпилированный исполняемый файл. Входные данные не требуются. Ожидаемый результат — сообщение "Жесткая ссылка успешно создана.".

#include <unistd.h>

#include <stdio.h>

#include <stdlib.h>

int main() {

if (link("source.txt", "new\_link.txt") == -1) {

perror("link");

exit(EXIT\_FAILURE);

}

printf("Жесткая ссылка успешно создана.\n");

return 0;

}

MakeFile:

all: hard\_link

hard\_link: hard\_link.c

gcc hard\_link.c -o hard\_link

clean:

rm -f hard\_link

1. Процессы-демоны

Задача: Создать демон-процесс в Linux.

Описание: Программа на языке C, которая создает демон-процесс в операционной системе Linux.

Инструкции: Компилируется с помощью Makefile. После запуска программа создает демон-процесс, который работает в фоновом режиме.

// Процессы-демоны

// Задача: Создать демон-процесс в Linux.

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/stat.h>

int main() {

pid\_t pid;

pid = fork();

if (pid < 0) {

exit(EXIT\_FAILURE);

}

if (pid > 0) {

exit(EXIT\_SUCCESS);

}

umask(0);

if (setsid() < 0) {

exit(EXIT\_FAILURE);

}

printf("Демон запущен.\n");

while (1) {

sleep(60);

}

return 0;

}

MakeFile:

all: daemon\_process

daemon\_process: daemon\_process.c

gcc daemon\_process.c -o daemon\_process

clean:

rm -f daemon\_process

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

1. Отключение буферизации для стандартного вывода

Описание задачи: Необходимо создать программу на языке C, которая отключает буферизацию для стандартного вывода и немедленно выводит строку "Hello, world!".

Инструкции: Программа компилируется с помощью Makefile. После запуска она выводит строку "Hello, world!" без задержек.

Тестовые входные и выходные данные

Не требуется вводных данных. Ожидается, что программа выведет "Hello, world!" немедленно после запуска.

#include <stdio.h>

int main() {

setbuf(stdout, NULL);

printf("Hello, world!\n");

return 0;

}

MakeFile:

all: setbuf\_example

setbuf\_example: setbuf\_example.c

gcc setbuf\_example.c -o setbuf\_example

clean:

rm -f setbuf\_example

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

1. Использование nm

Задание: Напишите программу на языке C, содержащую функции и переменные. После компиляции в исполняемый файл используйте утилиту nm, чтобы вывести список всех символов, включая имена функций и переменных, с их адресами в памяти.

Инструкции: 1. Создайте программу на языке C.

2. Скомпилируйте программу в исполняемый файл.

3. Используйте утилиту `nm` для анализа символов в этом исполняемом файле.

#include <stdio.h>

void function1() {

printf("Function 1\n");

}

void function2() {

printf("Function 2\n");

}

int main() {

function1();

function2();

return 0;

}

MakeFile:

all: symbol\_analysis

symbol\_analysis: symbol\_analysis.c

gcc symbol\_analysis.c -o symbol\_analysis

clean:

rm -f symbol\_analysis

1. Runtime линковка

Описание задачи: В этом задании вы познакомитесь с понятием runtime (время выполнения) линковки в языке программирования C. Runtime линковка позволяет загружать и связывать библиотеки и модули программы во время ее выполнения, что дает большую гибкость и возможность динамически загружать код и изменять его поведение.

Задача:

1. Напишите программу на языке C, которая динамически загружает и использует функцию из внешней библиотеки во время выполнения.

2. Скомпилируйте программу в исполняемый файл и динамическую библиотеку.

3. В основной программе используйте утилиты `dlopen`, `dlsym` и `dlclose` для динамической загрузки, поиска символов и выгрузки библиотеки во время выполнения.

4. Протестируйте программу, убедившись, что она успешно загружает и использует функцию из внешней библиотеки.

#include <stdio.h>

#include <dlfcn.h>

int main() {

void \*handle;

int (\*add)(int, int);

handle = dlopen("./library.so", RTLD\_LAZY);

if (!handle) {

fprintf(stderr, "%s\n", dlerror());

return 1;

}

add = dlsym(handle, "add");

if (!add) {

fprintf(stderr, "%s\n", dlerror());

dlclose(handle);

return 1;

}

printf("Результат сложения: %d\n", add(5, 3));

dlclose(handle);

return 0;

}

Library:

#include <stdio.h>

int add(int a, int b) {

return a + b;

}

MakeFale:

all: runtime\_linkage library

runtime\_linkage: runtime\_linkage.c

gcc runtime\_linkage.c -o runtime\_linkage -ldl

library: library.c

gcc -shared -fPIC library.c -o library.so

clean:

rm -f runtime\_linkage library.so

1. Create

Описание задачи: Напишите программу на языке C, которая создает новый файл и записывает в него некоторый текст.

Задача:

1. Напишите программу на языке C, которая создает новый файл.

2. Запишите в этот файл некоторый текст.

3. Убедитесь, что файл успешно создан и содержит записанный текст.

#include <stdio.h>

int main() {

FILE \*fp;

fp = fopen("output.txt", "w");

if (fp == NULL) {

printf("Ошибка при открытии файла.\n");

return 1;

}

fprintf(fp, "Привет, мир!");

fclose(fp);

printf("Файл успешно создан и текст успешно записан.\n");

return 0;

}

MakeFile:  
all: create\_file

create\_file: create\_file.c

gcc create\_file.c -o create\_file

clean:

rm -f create\_file

1. Подсчет числа процессов с заданным именем

Цель задания:Научиться работать с системными вызовами Linux для получения информации о процессах и использовать базовые возможности языка программирования C для обработки этой информации.

Задача: Разработать программу на языке C, которая принимает имя процесса как входной параметр и выводит количество запущенных процессов с этим именем.

Инструкции по выполнению

1. Изучите системный вызов `popen()` и команду `pgrep -c`.

2. Напишите программу, которая принимает имя процесса в качестве аргумента командной строки.

3. Программа должна выполнять команду `pgrep -c` для подсчета количества процессов с заданным именем.

4. Выведите результат на экран.

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

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

if (argc != 2) {

fprintf(stderr, "Usage: %s process\_name\n", argv[0]);

return EXIT\_FAILURE;

}

char command[256];

sprintf(command, "pgrep -c %s", argv[1]);

FILE \*fp = popen(command, "r");

if (fp == NULL) {

perror("Failed to run command");

exit(EXIT\_FAILURE);

}

char num\_procs[10];

if (fgets(num\_procs, sizeof(num\_procs), fp) != NULL) {

printf("Number of processes named '%s': %s", argv[1], num\_procs);

}

pclose(fp);

return 0;

}

MakeFile:

CC=gcc

CFLAGS=-Wall

all: count\_procs

count\_procs: count\_procs.c

$(CC) $(CFLAGS) count\_procs.c -o count\_procs

clean:

rm -f count\_procs