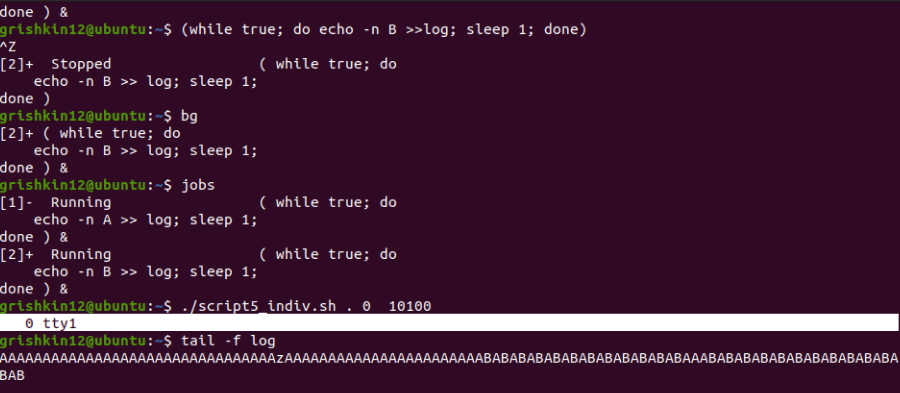
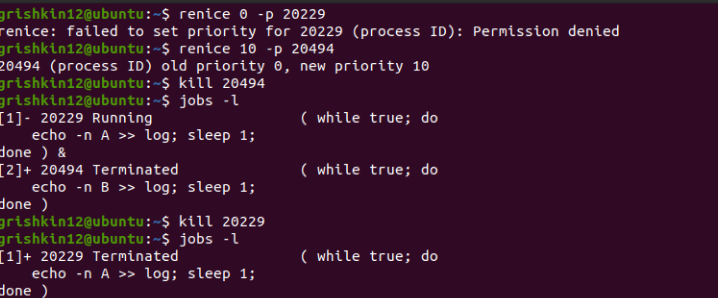


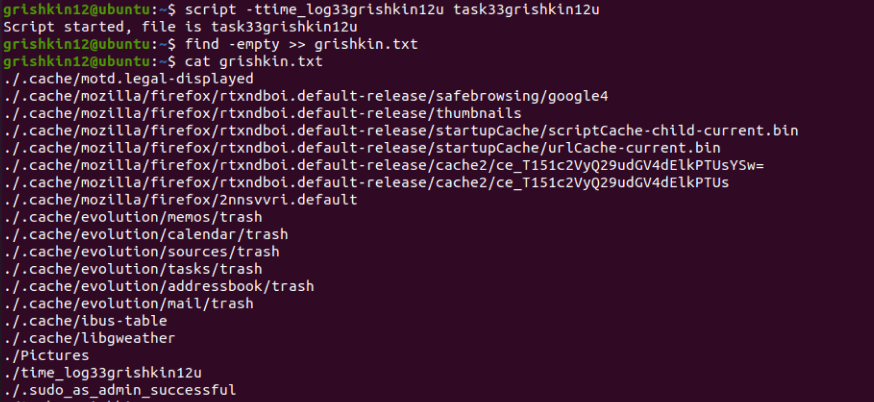
3.2

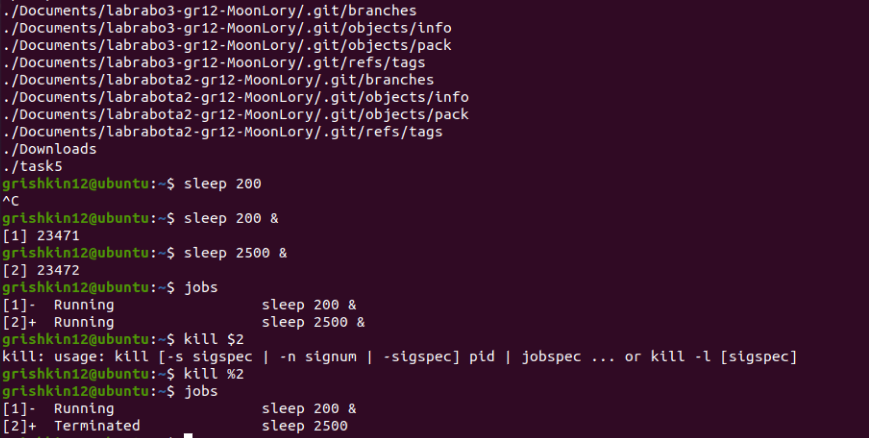


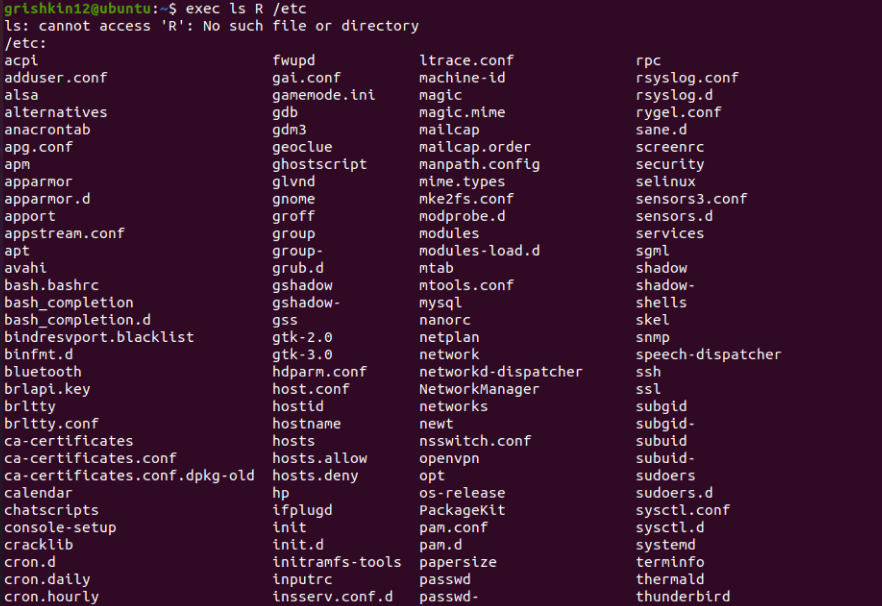




3.3



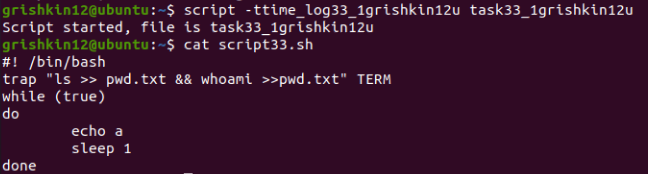


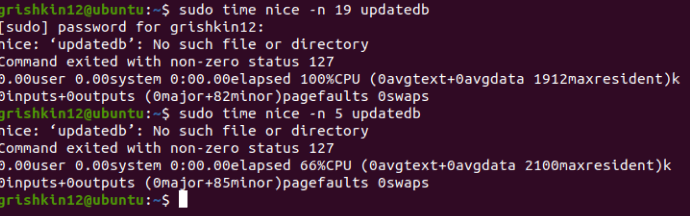


*4. TERM - игнорируется  
INT - очищает командную строку*

*HUP - завершает сеанс*

*5. Ничего не произойдёт*

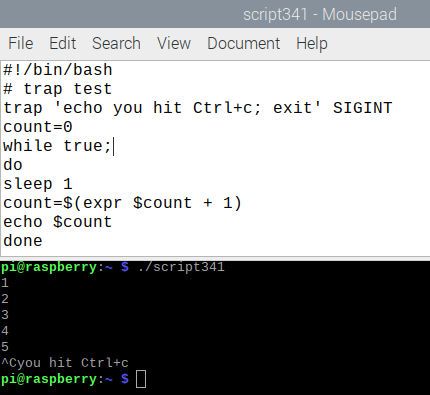
**

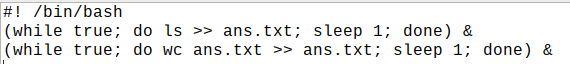
**

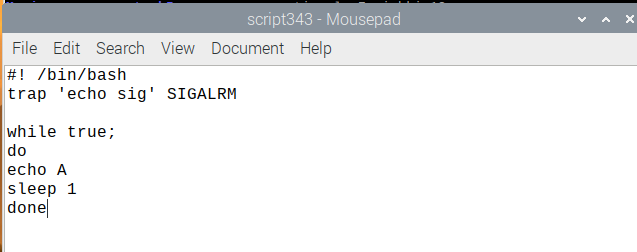
**

**

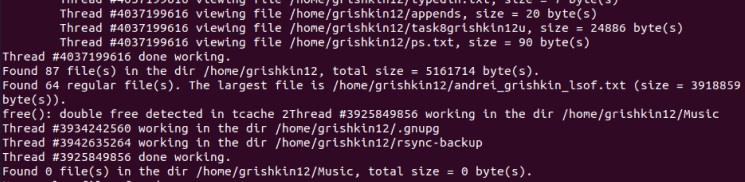
3.4







3.5 (код в самом конце)



Контрольные вопросы

1) *Объясните понятия процесса и ресурса. Какое их значение в организации вычислительного процесса в ОС Linux?*

Процесс – набор выполняемых заданий, инструкций. Ресурс – это то, что выделяется процессу для выполнения данных задач. Процесс представлен в памяти в виде дескриптора процесса, который включает: исполняемый текст программы и управляющая информация. Концепция процессов обеспечивает свойство многозадачности ОС, т.е. когда в активном состоянии могут находиться несколько задач, между которыми разделяется процессорное время.

2) *Какая информация содержится в описателях процессов? Как просмотреть их содержание в процессе работы с системой?*

Описатель процесса – это дескриптор. Дескриптор включает в себя исполняемый текст программы задачи, управляющую информацию (контекст процесса), необходимую для выделения и использования выделенных процессу ресурсов: приоритет, необходимое адресное пространство, вектор прерывания (слово состояния программы), величину кванта процессорного времени, признаки состояния процесса и пр.

Посмотреть: ps -l, ps.

3) *Какими способами можно организовать выполнение программ в фоновом режиме?*

Для запуска фонового (параллельного с другими потомками) процесса в командную строку необходимо и достаточно последним символом добавить знак &

4) *Какие особенности выполнения программ в фоновом режиме? Как избежать вывода фоновых сообщений на экран и прерывания выполнения фоновых программ при прекращении сеанса работы с системой?*

Фоновые процессы обладают некоторыми недостатками:

- не допускают ввода с клавиатуры;

- обеспечивают вывод на экран, но при этом нарушают целостность вывода диалогового процесса.

Общепринятый прием исключения влияния фонового вывода на интерактивную работу:

$<командная\_строка> > имя\_файла.out &

- <командная\_строка> планирует задание для фонового режима;

- перенаправляет вывод вместо экрана в указанный файл головного каталога пользователя.

особенности работы с фоновым режимом:

- выполняемая в фоновом режиме программа (команда), требующая стандартного ввода, должна читать его из файла с использованием перенаправленного ввода;

- программа, выполняемая в фоновом режиме, не может быть прервана <Ctrl\*C>, так как она отсоединяется от клавиатуры и может быть прекращена только с помощью команды kill или выходом из системы;

- выход из системы exit надо выполнять два раза: для завершения фонового процесса и завершения основного процесса shell.

5) *Как пользователь может повлиять на распределение ресурсов между активными процессами?*

С помощью команды nice.

Если при выполнении задания образуются несколько порожденных процессов, то все они имеют одинаковый приоритет равный родительскому. В этом случае все процессы получают ресурсы равными долями (простой режим разделения времени). При необходимости выделения наиболее важных родительских процессов порожденным второстепенным можно понизить приоритет с помощью команды.

6) *Как можно прервать выполнение активных процессов? Какая информация для этого необходима и откуда она извлекается?*

Завершение процессов – одна из функций управления процессами. Прекратить выполнение любого процесса можно с помощью команды:

kill [-опции] PID1 [PID2......] — передает сигнал процессу PID.

7) *Перечислите базовые средства взаимодействия процессов в Linux.*

Каналы, сигналы, разделяемая память, очереди сообщений, сокеты

8) *Поясните особенности работы с каналами в Linux.*

Канал - это средство связи стандартного вывода одного процесса со стандартным вводом другого. Эта особенность широко используется даже в командной строке UNIX (в shell-е). Когда процесс создает канал, ядро устанавливает два файловых дескриптора для пользования этим каналом. Один такой дескриптор используется, чтобы открыть путь ввода в канал (запись), в то время как другой применяется для получения данных из канала (чтение)

9) *Почему отложенные вызовы не обрабатываются непосредственно обработчиком прерывания таймера?*

Потому что функции отложенных вызовов выполняются в системном контексте, а не в контексте прерывания

КОД 3.5

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <stdbool.h>

#include <stdatomic.h>

#include <pthread.h>

#include <semaphore.h>

#include <dirent.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <unistd.h>

#define BUFFER\_SIZE 9964

typedef struct dir\_queue

{

char \*dir\_path;

struct dir\_queue \*next;

}

dir\_queue;

dir\_queue \*dir\_queue\_head, \*dir\_queue\_tail;

sem\_t dir\_queue\_mutex;

void init\_dir\_queue()

{

dir\_queue\_head = dir\_queue\_tail = NULL;

sem\_init(&dir\_queue\_mutex, 0, 1);

}

void enqueue\_dir(const char \*dir\_path)

{

dir\_queue \*element = malloc(sizeof(dir\_queue));

element->dir\_path = malloc(strlen(dir\_path) + 1);

strcpy(element->dir\_path, dir\_path);

element->next = NULL;

sem\_wait(&dir\_queue\_mutex);

if (!dir\_queue\_head)

dir\_queue\_head = dir\_queue\_tail = element;

else

{

dir\_queue\_tail->next = element;

dir\_queue\_tail = element;

}

sem\_post(&dir\_queue\_mutex);

}

char \* dequeue\_dir()

{

char \*result = NULL;

sem\_wait(&dir\_queue\_mutex);

if (!dir\_queue\_head)

{

sem\_post(&dir\_queue\_mutex);

return NULL;

}

dir\_queue \*temp = dir\_queue\_head;

dir\_queue\_head = dir\_queue\_head->next;

if (!dir\_queue\_head)

dir\_queue\_tail = NULL;

sem\_post(&dir\_queue\_mutex);

result = temp->dir\_path;

free(temp);

return result;

}

void destroy\_dir\_queue()

{

sem\_destroy(&dir\_queue\_mutex);

for (dir\_queue \*curr = dir\_queue\_head; curr;)

{

dir\_queue \*temp = curr;

curr = curr->next;

free(temp->dir\_path);

free(temp);

}

dir\_queue\_tail = NULL;

}

sem\_t console\_mutex, file\_mutex;

FILE \*output\_log\_file;

void init\_printing(const char \*file\_name)

{

output\_log\_file = fopen(file\_name, "w");

sem\_init(&console\_mutex, 0, 1);

sem\_init(&file\_mutex, 0, 1);

}

void print\_to\_file(FILE \*file, const char \*msg, sem\_t \*mutex)

{

sem\_wait(mutex);

fputs(msg, file);

fputc('\n', file);

sem\_post(mutex);

}

void print\_msg(const char \*msg)

{

print\_to\_file(output\_log\_file, msg, &file\_mutex);

print\_to\_file(stdout, msg, &console\_mutex);

}

void destroy\_printing()

{

fclose(output\_log\_file);

sem\_destroy(&console\_mutex);

sem\_destroy(&file\_mutex);

}

sem\_t dir\_thread\_semaphore;

void \* dir\_thread\_func(void \*dir)

{

char rel\_file\_name[2 \* BUFFER\_SIZE], full\_file\_name[2 \* BUFFER\_SIZE], message\_buffer[5 \* BUFFER\_SIZE];

char \*dir\_path = dir;

int dir\_path\_length = strlen(dir\_path);

strcpy(rel\_file\_name, dir\_path);

pthread\_t thread\_id = pthread\_self();

sprintf(message\_buffer, "Thread #%u working in the dir %s",

(unsigned)thread\_id, dir\_path);

print\_msg(message\_buffer);

DIR \*dir\_stream = opendir(dir\_path);

if (dir\_stream == NULL)

{

sprintf(message\_buffer, "Thread #%u can't open the dir %s!",

(unsigned)thread\_id, dir\_path);

print\_msg(message\_buffer);

free(dir\_path);

sem\_post(&dir\_thread\_semaphore);

pthread\_exit(NULL);

}

struct dirent \*dir\_entry;

struct stat dir\_entry\_stat;

int total\_files = 0, total\_viewed\_files = 0;

long long total\_file\_size = 0;

char largest\_file[2 \* BUFFER\_SIZE];

off\_t largest\_file\_size = -1;

while (dir\_entry = readdir(dir\_stream))

{

if (!strcmp(dir\_entry->d\_name, ".") ||

!strcmp(dir\_entry->d\_name, ".."))

continue;

++total\_files;

rel\_file\_name[dir\_path\_length] = '/';

rel\_file\_name[dir\_path\_length + 1] = '\0';

strcat(rel\_file\_name, dir\_entry->d\_name);

if (stat(rel\_file\_name, &dir\_entry\_stat) < 0)

continue;

total\_file\_size += dir\_entry\_stat.st\_size;

if ((dir\_entry\_stat.st\_mode & S\_IFMT) == S\_IFLNK)

continue;

else if ((dir\_entry\_stat.st\_mode & S\_IFMT) == S\_IFDIR)

enqueue\_dir(rel\_file\_name);

else if ((dir\_entry\_stat.st\_mode & S\_IFMT) == S\_IFREG)

{

++total\_viewed\_files;

realpath(rel\_file\_name, full\_file\_name);

sprintf(message\_buffer, "\tThread #%u viewing file %s, size = %li byte(s)",

(unsigned)thread\_id, full\_file\_name, dir\_entry\_stat.st\_size);

print\_msg(message\_buffer);

if (dir\_entry\_stat.st\_size > largest\_file\_size)

{

largest\_file\_size = dir\_entry\_stat.st\_size;

strcpy(largest\_file, full\_file\_name);

}

}

}

sprintf

(message\_buffer, "Thread #%u done working.\nFound %i file(s) in the dir %s, total size = %lli byte(s).\n",

(unsigned)thread\_id, total\_files, dir\_path, total\_file\_size);

if (total\_viewed\_files == 0)

sprintf(message\_buffer + strlen(message\_buffer), "No regular files found.");

else

{

sprintf(message\_buffer + strlen(message\_buffer),

"Found %i regular file(s). The largest file is %s (size = %li byte(s)).",

total\_viewed\_files, largest\_file, largest\_file\_size);

}

print\_msg(message\_buffer);

closedir(dir);

free(dir\_path);

sem\_post(&dir\_thread\_semaphore);

}

atomic\_bool stop\_traversal = false;

void \* daemon\_thread\_func(void \*number\_of\_traversers)

{

int traversers = \*((int \*)number\_of\_traversers);

sem\_init(&dir\_thread\_semaphore, 0, traversers);

while (!stop\_traversal)

{

while (dir\_queue\_head)

{

if (stop\_traversal)

break;

if (sem\_trywait(&dir\_thread\_semaphore) < 0)

continue;

char \*dir = dequeue\_dir();

pthread\_t thread\_id;

pthread\_create(&thread\_id, NULL, dir\_thread\_func, dir);

pthread\_detach(thread\_id);

}

}

int sem\_value;

do

{

sem\_getvalue(&dir\_thread\_semaphore, &sem\_value);

}

while (sem\_value < traversers);

sem\_destroy(&dir\_thread\_semaphore);

}

int main(int argc, char \*\*argv)

{

if (argc != 4)

{

puts("Incorrect number of parameters; must be exactly 3: the dir name, the log name, and the number of simultaneous threads");

return -1;

}

int traversers = strtol(argv[3], NULL, 10);

if (traversers <= 0)

{

puts("Incorrect number of threads; must be nonnegative");

return -2;

}

init\_dir\_queue();

init\_printing(argv[2]);

enqueue\_dir(argv[1]);

pthread\_t daemon\_thread\_id;

pthread\_create(&daemon\_thread\_id, NULL, daemon\_thread\_func, &traversers);

puts("Press any key to stop traversing...");

getchar();

stop\_traversal = true;

pthread\_join(daemon\_thread\_id, NULL);

destroy\_printing();

destroy\_dir\_queue();

return 0;

}