УО «Белорусский государственный университет информатики и радиоэлектроники»

Кафедра ПОИТ

Отчет по лабораторной работе № 7

по предмету «Архитектура компьютерной техники и операционных систем»

Вариант 12

Выполнил:

Крутько А.А.

гр. 251004

Проверил:

Леванцевич В.А.

Минск 2023

Задание № 2

Написать программу синхронизации двух каталогов, например, ***Dir1*** и ***Dir2***. Пользователь задаёт имена ***Dir1*** и ***Dir2***в качестве первого и второго аргумента командной строки. В результате работы программы файлы, имеющиеся в ***Dir1***, но отсутствующие в ***Dir2***, должны скопироваться в ***Dir2*** вместе с правами доступа. Процедуры копирования должны запускаться в отдельном процессе для каждого копируемого файла с использованием функций ***read ()*** и ***write ()***. Каждый процесс выводит на экран свой ***pid,*** полный путь, имя копируемого файла и число скопированных байт. Число запущенных процессов в любой момент времени не должно превышать ***N*** (вводится пользователем).

* Лб. 7.1 посвящена созданию и управлению процессами.
* Лб. 7.2 посвящена созданию и управлению потоками.

Код программы 7.1:

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <dirent.h>

#include <sys/stat.h>

#include <sys/types.h>

#include <fcntl.h>

#include <unistd.h>

#include <stdbool.h>

#include <sys/wait.h>

#include <errno.h>

#include <limits.h>

typedef struct dirent DIRENT;

typedef struct stat STAT;

int copyFile(const char\* sourcePath, const char\* destPath, mode\_t ch) {

int sourceFd, destFd;

char buffer[PATH\_MAX];

int bytesRead, bytesWritten;

sourceFd = open(sourcePath, O\_RDONLY);

if (sourceFd == -1) {

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

return 0;

}

destFd = open(destPath, O\_WRONLY | O\_CREAT, ch);

if (destFd == -1) {

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

close(sourceFd);

return 0;

}

while ((bytesRead = read(sourceFd, buffer, sizeof(buffer))) > 0) {

bytesWritten = write(destFd, buffer, bytesRead);

}

close(sourceFd);

close(destFd);

return bytesWritten;

}

void syncDirectories(const char\* sourceDir, const char\* destDir, int maxProcesses, int\* currentProc) {

DIR \*dir1 = opendir(sourceDir);

DIR \*dir2 = opendir(destDir);

if (dir1 == NULL || dir2 == NULL) {

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

return;

}

DIRENT \*entry;

while ((entry = readdir(dir1)) != NULL) {

if (strcmp(entry->d\_name, ".") != 0 && strcmp(entry->d\_name, "..") != 0) {

bool wasFound = false;

rewinddir(dir2);

DIRENT \*end;

while ((end = readdir(dir2)) != NULL) {

if (entry->d\_ino == end->d\_ino) {

wasFound = true;

break;

}

}

if (!wasFound) {

char sourcePath[PATH\_MAX];

snprintf(sourcePath, PATH\_MAX, "%s/%s", sourceDir, entry->d\_name);

STAT buf1;

stat(sourcePath, &buf1);

if (S\_ISDIR(buf1.st\_mode)) {

syncDirectories(sourcePath, destDir, maxProcesses, currentProc);

} else if (S\_ISREG(buf1.st\_mode)) {

pid\_t childPid = fork();

if (childPid == -1) {

fprintf(stderr, "Error creating child process\n");

return;

}

if (childPid == 0) {

// Child process scope

char destPath[PATH\_MAX];

snprintf(destPath, PATH\_MAX, "%s/%s", destDir, entry->d\_name);

int bytes = copyFile(sourcePath, destPath, buf1.st\_mode);

fprintf(stdout, "PID: %d, PPID: %d, Path: %s, File: %s, Bytes: %d\n\n", getpid(), getppid(), sourcePath, entry->d\_name, bytes);

exit(0);

} else {

// Parent process scope

(\*currentProc)++;

fprintf(stdout, "I AM PROCESS OF THE %d AND MY SON IS %d AND MY PARENT IS %d\n\n", getpid(),

childPid, getppid());

if ((\*currentProc) >= maxProcesses) {

waitpid(-1, NULL, 0);

(\*currentProc)--;

}

}

}

}

}

}

closedir(dir1);

closedir(dir2);

while ((\*currentProc) > 0) {

waitpid(-1, NULL, 0);

(\*currentProc)--;

}

}

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

if (argc != 4) {

fprintf(stderr, "Usage: %s <directory\_1> <directory\_2> <count\_of\_processes>\n", argv[0]);

return 1;

}

const char\* catalogPath1 = argv[1];

const char\* catalogPath2 = argv[2];

int count = atoi(argv[3]);

int current = 0;

syncDirectories(catalogPath1, catalogPath2, count, &current);

return 0;

}

Код программы 7.2:

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <dirent.h>

#include <sys/stat.h>

#include <pthread.h>

#include <fcntl.h>

#include <unistd.h>

#include <stdbool.h>

#include <sys/wait.h>

#include <errno.h>

#include <limits.h>

typedef struct dirent DIRENT;

typedef struct stat STAT;

typedef struct {

const char\* sourcePath;

const char\* name;

mode\_t mode;

} ThreadArgs;

int copyFile(const char\* sourcePath, const char\* destPath, mode\_t ch) {

int sourceFd, destFd;

char buffer[PATH\_MAX];

int bytesRead, bytesWritten;

sourceFd = open(sourcePath, O\_RDONLY);

if (sourceFd == -1) {

printf("Error opening source file");

return 0;

}

destFd = open(destPath, O\_WRONLY | O\_CREAT, ch);

if (destFd == -1) {

printf("Error opening destination file");

close(sourceFd);

return 0;

}

while ((bytesRead = read(sourceFd, buffer, sizeof(buffer))) > 0) {

bytesWritten = write(destFd, buffer, bytesRead);

}

close(sourceFd);

close(destFd);

return bytesWritten;

}

void\* threadCopyFile(void\* args) {

ThreadArgs\* threadArgs = (ThreadArgs\*)args;

const char\* sourcePath = threadArgs->sourcePath;

const char\* name = threadArgs->name;

mode\_t mode = threadArgs->mode;

printf("Thread ID: %ld, Path: %s, File: %s, Bytes: %d\n\n", pthread\_self(), sourcePath, name, copyFile(sourcePath, name, mode));

free(args);

return NULL;

}

void syncDirectories(const char\* sourceDir, const char\* destDir, int maxThreads, pthread\_t\* threadIDs, int\* threadCount) {

DIR \*dir1 = opendir(sourceDir);

DIR \*dir2 = opendir(destDir);

if (dir1 == NULL || dir2 == NULL) {

printf("Error opening directories");

return;

}

DIRENT \*entry;

while ((entry = readdir(dir1)) != NULL) {

if (strcmp(entry->d\_name, ".") != 0 && strcmp(entry->d\_name, "..") != 0) {

bool wasFound = false;

rewinddir(dir2);

DIRENT \*end;

while ((end = readdir(dir2)) != NULL) {

if (entry->d\_ino == end->d\_ino) {

wasFound = true;

break;

}

}

if (!wasFound) {

char sourcePath[PATH\_MAX];

snprintf(sourcePath, PATH\_MAX, "%s/%s", sourceDir, entry->d\_name);

STAT buf1;

stat(sourcePath, &buf1);

if (S\_ISDIR(buf1.st\_mode)) {

syncDirectories(sourcePath, destDir, maxThreads, threadIDs, threadCount);

} else if (S\_ISREG(buf1.st\_mode)) {

char destPath[PATH\_MAX];

snprintf(destPath, PATH\_MAX, "%s/%s", destDir, entry->d\_name);

ThreadArgs\* threadArgs = (ThreadArgs\*)malloc(sizeof(ThreadArgs));

threadArgs->sourcePath = strdup(sourcePath);

threadArgs->name = strdup(destPath);

threadArgs->mode = buf1.st\_mode;

pthread\_create(&threadIDs[\*threadCount], NULL, threadCopyFile, (void\*)threadArgs);

(\*threadCount)++;

if (\*threadCount >= maxThreads) {

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

pthread\_join(threadIDs[i], NULL);

}

\*threadCount = 0;

}

}

}

}

}

closedir(dir1);

closedir(dir2);

for (int i = 0; i < \*threadCount; i++) {

pthread\_join(threadIDs[i], NULL);

}

}

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

if (argc != 4) {

fprintf(stderr, "Usage: %s <directory\_1> <directory\_2> <count\_of\_threads>\n", argv[0]);

return 1;

}

const char\* catalogPath1 = argv[1];

const char\* catalogPath2 = argv[2];

int count = atoi(argv[3]);

pthread\_t threadIDs[count];

int threadCount = 0;

syncDirectories(catalogPath1, catalogPath2, count, threadIDs, &threadCount);

return 0;

}

Контрольные вопросы:

1. Понятие процесса и потока
2. Понятие «образ процесса», системный регистровый и пользовательский контексты процесса
3. Диаграмма состояний процесса.
4. Особенности создания процесса с системным вызовом fork(). Механизм Copy-on-write
5. Процесс – это программа в выполнении. Процесс включает в себя исполняемый код, данные и системные ресурсы, такие как файлы и сетевые соединения. Каждый процесс имеет свое собственное адресное пространство памяти.

Поток (нить) – это легкий исполняемый элемент внутри процесса. Потоки в рамках одного процесса могут выполняться параллельно и имеют общее адресное пространство памяти с доступом к одним данным, что делает их более эффективными по сравнению с процессами.

1. Содержимое назначенного процессу виртуального адресного пространства, т.е. коды команд, исходящие и промежуточные данные, а также результаты вычислений, представляют собой **образ процесса.**

**Регистровый контекст процесса**:

1. Счетчик команд: Этот регистр указывает на виртуальный адрес следующей команды, которую процессор должен выполнить. Каждый процесс имеет свой собственный счетчик команд.

2. Регистр состояния процессора (PS): Содержит различные флаги условий, такие как нулевой, положительный или отрицательный результат последних вычислений. Также включает информацию о текущем уровне прерывания, режиме выполнения процесса (режим ядра/задачи) и другие аппаратные статусы.

3. Указатель вершины стека: Этот указатель указывает на следующий элемент стека, который используется для хранения временных данных и адресов возврата из подпрограмм. В зависимости от архитектуры, указатель вершины стека может указывать на свободное место в стеке или на последний использованный элемент.

4. Регистры общего назначения: Это регистры, в которых процесс хранит данные во время своего выполнения. Некоторые из них могут использоваться для передачи информации между процессом и ядром операционной системы.

**Системный контекст процесса**:

1. Запись в таблице процессов: Эта запись описывает состояние процесса и содержит управляющую информацию, к которой ядро операционной системы может обращаться для управления процессом.

2. Часть адресного пространства задачи: Это выделенная часть виртуального адресного пространства процесса, где хранится управляющая информация о процессе, доступная только в контексте процесса. Например, здесь могут храниться параметры, такие как приоритет процесса.

3. Записи частной таблицы областей процесса: Они описывают области виртуального адресного пространства процесса, включая области команд, данных и стека. Эти записи необходимы для преобразования виртуальных адресов в физические.

4. Стек ядра: Если процесс выполняется в режиме ядра, то здесь хранятся записи процедур ядра, такие как вызовы функций операционной системы. Каждый процесс имеет свой собственный стек ядра.

5. Динамическая часть системного контекста: Это стек уровней, который содержит информацию для восстановления предыдущего уровня. Эти уровни могут быть помещены в стек ядром или вытолкнуты из стека в зависимости от событий. Информация на каждом уровне включает в себя регистровый контекст предыдущего уровня.

**Пользовательский контекст процесса**:

1. Адресное пространство процесса: Включает код и данные, связанные с выполнением пользователя. Это включает инструкции программы, переменные и другие данные, сгенерированные самим процессом.

2. Регистры общего назначения: Хранят информацию, созданную процессом во время его выполнения. Эти регистры могут использоваться для передачи данных между различными частями пользовательского кода.

3. Стек пользовательского кода: Хранит временные данные и адреса возврата для подпрограмм, вызываемых пользовательским кодом. Каждый процесс имеет свой собственный стек, изолированный от стека ядра.

1. 
2. Создание процесса с fork():

* Дочерний процесс: Создается как почти точная копия родительского процесса, включая выполняемый код, данные и ресурсы.
* Выполнение одновременно: Родитель и дочерний процессы выполняются одновременно, начиная с оператора, следующего сразу за вызовом fork(). Оба процесса работают в разных адресных пространствах.

Copy-On-Write (CoW):

* Эффективное копирование ресурсов: После fork(), все ресурсы дочернего процесса являются копией ресурсов родителя.
* Отметка страниц как read-only: Все страницы памяти родителя помечаются как read-only и становятся доступными и для родителя, и для дочернего процесса.
* Ленивое копирование: Когда один из процессов изменяет данные на странице, происходит копирование страницы, и только тогда она становится доступной только этому процессу. Оригинал "отвязывается" от процесса.
* Оптимизация затрат на память: Такой механизм позволяет оптимизировать затраты на память, так как реальное копирование данных происходит только в случае изменения.