Московский Авиационный Институт

(Национальный Исследовательский Университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

**Лабораторная работа №4 по курсу**

**«Операционные системы»**

**ОТОБРАЖЕНИЕ ФАЙЛОВ В ПАМЯТЬ**

Студент: Черненко Илья Денисович

Группа: М8О–206Б–18

Вариант: 20

Преподаватель: Соколов Андрей Алексеевич

Оценка: \_\_\_\_\_\_\_\_\_\_

Дата: \_\_\_\_\_\_\_\_\_\_

Подпись: \_\_\_\_\_\_\_\_\_\_

Москва, 2019.

**Содержание**

1. Постановка задачи
2. Общие сведения о программе
3. Общий метод и алгоритм решения
4. Основные файлы программы
5. Примеры работы
6. Вывод

**Постановка задачи**

Составить и отладить программу на языке Си, осуществляющую работу с процессами и взаимодействие между ними в одной из двух операционных систем. В результате работы программа (основной процесс) должен создать для решение задачи один или несколько дочерних процессов. Взаимодействие между процессами осуществляется через системные сигналы/события и/или через отображаемые файлы (memory-mapped files).Необходимо обрабатывать системные ошибки, которые могут возникнуть в результате работы.

Дочерний процесс представляет собой сервер по работе с деревом общего вида и принимает команды со стороны родительского процесса.

**Общие сведения о программе**

Программа компилируется из файла main.c. Также используется заголовочные файлы: stdio.h, stdbool.h, stdlib.h, string.h, sys/types.h, fcntl.h, sys/mman.h, sys/stat.h, semaphore.h, pthread.h, unistd.h, c\_queue.h. В программе используются следующие системные вызовы:

1. **mmap** – отражает указанное кол-во байтов , начиная с некоторого смещения файла(определенного файловым описателем).
2. **munmap** – удаляет все отражения на указанную область памяти.
3. **sem\_post** – увеличивает (разблокирует) семафор, на который указывает аргумент функции. Если значение семафора после этого становится больше нуля, то другой процесс или нить заблокированная в вызове sem\_wait, проснётся и заблокирует семафор
4. **sem\_open –** создаёт новый семафор POSIX или открывает существующий семафор. Семафору присваивается имя, указанное в его аргументах.
5. **unlink** – удаляет имя и файл, на который он ссылается.
6. **sem\_wait** – уменьшает (блокирует) семафор, на который указывает *sem*. Если значение семафор больше нуля, то выполняется уменьшение и функция сразу завершается. Если значение семафора равно нулю, то вызов блокируется до тех пор, пока не станет возможным выполнить уменьшение (т. е., значение семафора не станет больше нуля), или пока не вызовется обработчик сигнала.

**Общий метод и алгоритм решения**.

Для реализации поставленной задачи необходимо:

1. Используя системный вызов mmap, используя файловый дескриптор fd(который ранее определяется функцией create\_tmp(), данные отражаются в память.
2. Используя системный вызов fork создать дочерний процесс.
3. В родительском процессе считывать данные cо стандартного потока и правильно распарсить подаваемую команду.
4. Как в родительском процессе данные считались, необходимо записать их в отображаемую память, с помощью sprintf.
5. Как только родительский процесс записал данные процесс считывает их, производит вычисления с деревом общего вида.
6. Дочерний процесс выводит результат используя write.
7. **Основные файлы программы**

**main.cpp:**

#include <stdio.h>

#include <stdbool.h>

#include <stdlib.h>

#include <string.h>

#include <sys/types.h>

#include <fcntl.h>

#include <sys/mman.h>

#include <sys/stat.h>

#include <semaphore.h>

#include <pthread.h>

#include <unistd.h>

#include "c\_queue.h"

typedef struct ans ans;

struct ans {

int cmd;

int val;

char path[32];

};

typedef struct node node,\*pnode;

struct node {

pnode s;

pnode b;

int val;

bool is\_root;

};

pnode node\_create(int val) {

pnode new\_node = (pnode)malloc(sizeof(node));

if (new\_node) {

new\_node -> val = val;

new\_node -> s = NULL;

new\_node -> b = NULL;

new\_node -> is\_root = false;

}

return new\_node;

}

pnode\* search(pnode\* t, queue \*path) {

if (!(\*t) && !q\_is\_empty(path)) {

return NULL;

}

if (!q\_is\_empty(path)) {

char c = q\_front(path);

pop(path);

if (c == 's') {

return search(&(\*t) -> s, path);

} else if (c == 'b') {

return search(&(\*t) -> b, path);

}

return NULL;

}

return t;

}

bool add(pnode\* t, int val, queue \*path) {

if (!(\*t) && q\_is\_empty(path)) {

(\*t) = node\_create(val);

return true;

}

pnode\* pr = search(t, path);

if (!pr) {

return false;

}

pnode new\_node = node\_create(val);

if (!new\_node) {

return false;

}

new\_node -> b = (\*pr);

(\*pr) = new\_node;

return true;

}

void rmv(pnode\* t) {

while((\*t) -> s != NULL){

rmv(&((\*t) -> s));

}

pnode tmp = \*t;

\*t = (\*t) -> b;

free(tmp);

}

bool valid\_numb(char\* numb) {

if (numb == NULL) {

return false;

}

bool flag = true;

int i = 0;

if (numb[i] != '-' && !(numb[i] >= '0' && numb[i] <= '9')) {

flag = false;

}

i++;

while (i < 11) {

if (numb[i] == '\0') {

break;

}

if (!(numb[i] >= '0' && numb[i] <= '9')) {

flag = false;

break;

}

i++;

}

return flag;

}

bool valid\_path(char\* path) {

if (path == NULL) {

return false;

}

if (path[0] == '@' && path[1] == '\0') {

return true;

}

for (int i = 0; i < 32; i++) {

if (path[i] == '\0') {

break;

} else if (path[i] != 's' && path[i] != 'b') {

return false;

}

}

return true;

}

ans\* parser(char\* cmd) {

ans\* parsed = (ans\*)malloc(sizeof(ans));

char\* pch = strtok(cmd," \n");

while (pch != NULL) {

if (strcmp(pch, "prt") == 0) {

parsed->cmd = 0;

break;

} else if (strcmp(pch, "rmv") == 0) {

pch = strtok(NULL, " \n");

if (valid\_path(pch)) {

parsed->cmd = 1;

strcpy(parsed->path, pch);

if (parsed->path[0] == 'b') {

parsed->cmd = -1;

}

break;

} else {

parsed->cmd = -1;

break;

}

} else if (strcmp(pch, "add") == 0) {

pch = strtok(NULL, " \n");

if (valid\_path(pch)) {

strcpy(parsed->path, pch);

pch = strtok(NULL, " \n");

if (parsed->path[0] == 'b') {

parsed->cmd = -1;

break;

}

if (valid\_numb(pch)) {

parsed->cmd = 2;

parsed->val = atoi(pch);

break;

} else {

parsed->cmd = -2;

break;

}

} else {

parsed->cmd = -1;

break;

}

} else if (strcmp(pch, "ext") == 0) {

parsed->cmd = 3;

break;

} else {

parsed->cmd = -777;

break;

}

}

return parsed;

}

void tree\_print(pnode t, int depth) {

if (t) {

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

write(1, "\t", 1);

}

char numb[11] = {'\0'};

sprintf(numb, "%d", t->val);

int i = 0;

while (numb[i] != '\0') {

i++;

}

write(1, numb, i);

write(1,"\n", 1);

tree\_print(t -> s, depth + 1);

tree\_print(t -> b, depth);

}

}

int create\_tmp() {

char \*fn = strdup("/tmp/tmpf.XXXXXX");

int fd = mkstemp(fn);

unlink(fn);

free(fn);

write(fd, " ", 100);

return fd;

}

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

setvbuf(stdout, (char \*) NULL, \_IONBF, 0);

pnode test = NULL;

char cmd[100] = {'\0'};

ans \*parsed = (ans \*) malloc(sizeof(ans));

int fd = create\_tmp();

lseek(fd, 100, SEEK\_END);

write(fd, "", 1);

struct stat sb;

if (fstat(fd, &sb) == -1) {

perror("can't get file size\n");

}

int fsize = sb.st\_size;

char\* f\_in\_m = mmap(NULL, fsize, PROT\_READ | PROT\_WRITE, MAP\_SHARED, fd, 0);

sem\_t\* sem\_calc = sem\_open("/calc", O\_CREAT, 777, 0);

if (sem\_calc == SEM\_FAILED) {

perror("Semaphores doesn't create");

exit(1);

}

sem\_unlink("/calc");

pid\_t pr = -1;

pr = fork();

if (pr < 0) {

write(1, "Can't create process\n", 22);

} else if (pr > 0) {

while (read(0, cmd, 100)) {

parsed = parser(cmd);

sprintf(f\_in\_m, "%d %d %s", parsed->cmd, parsed->val, parsed->path);

if (parsed->cmd == 3) {

return 0;

}

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

cmd[i] = '\0';

}

sem\_post(sem\_calc);

}

sem\_post(sem\_calc);

close(fd);

} else {

while (1) {

sem\_wait(sem\_calc);

queue \*q = q\_create();

sscanf(f\_in\_m, "%d %d %32s", &parsed->cmd, &parsed->val, parsed->path);

int k = 0;

while (parsed->path[k] != '\0') {

push(q, parsed->path[k]);

k++;

}

if (q\_size(q) == 0) {

push(q, '\0');

}

if (parsed->cmd == 3) {

return 0;

} else if (parsed->cmd == 2) {

if (test == NULL) {

while (q\_size(q) != 0) {

pop(q);

}

test = node\_create(parsed->val);

test->is\_root = true;

} else {

add(&test, parsed->val, q);

}

} else if (parsed->cmd == 1) {

pnode\* f = search(&test, q);

if (test == NULL) {

write(1, "empty tree\n", 11);

} else if ((\*f) == NULL) {

write(1, "its root\n", 9);

rmv(&test);

} else {

rmv(f);

}

} else if (parsed->cmd == 0) {

if (test == NULL) {

write(1, "empty tree\n", 11);

} else {

tree\_print(test, 0);

}

} else if (parsed->cmd == -2){

write(1, "invalid value\n", 14);

} else if (parsed->cmd == -1) {

write(1, "invalid path\n", 13);

} else if (parsed->cmd == -777) {

write(1, "invalid command\n", 16);

}

q\_destroy(q);

lseek(fd, 0, SEEK\_SET);

write(fd, "", 100);

}

sem\_close(sem\_calc);

munmap(f\_in\_m, 100);

lseek(fd, 0, SEEK\_SET);

write(fd, "", 100);

close(fd);

}

return 0;

}

**Примеры работы**

**test1:**

/home/zebr/os\_lab\_4-master/src/cmake-build-debug/os\_lab\_4

add s 3434

prt

3434

add s 999

PRT

invalid command

prt

3434

999

ext

Process finished with exit code 0

**test2:**

/home/zebr/os\_lab\_4-master/src/cmake-build-debug/os\_lab\_4

add s 34

add s 22

add sb 9

prt

34

22

9

add ss 0

prt

34

22

0

9

rmv ssb

its root

prt

empty tree

ext

Process finished with exit code 0

**test3:**

/home/zebr/os\_lab\_4-master/src/cmake-build-debug/os\_lab\_4

add s 324

prt

324

add s 929

prt

324

929

add s 223

prt

324

223

929

add ssb 31

prt

324

223

929

add sb 0

prt

324

223

0

929

add sbs 34

prt

324

223

0

34

929

ext

Process finished with exit code 0

**Вывод**

Обрел навыки работы с mmaped-files, ознакомился с устройством работы семафоров. Ознакомился с принципами работы (), и применил этот системный вызов для передачи данных дочернему процессу. Изучил принципы работы при создании временных файлов в ОС Linux.