Московский Авиационный Институт

(Национальный Исследовательский Университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

**Лабораторная работа №6-8 по курсу**

**«Операционные системы»**

Студент: Садаков А.А.

Группа: М8О–206Б–19

Вариант: 33

Преподаватель: Соколов Андрей Алексеевич

Оценка: \_\_\_\_\_\_\_\_\_\_

Дата: \_\_\_\_\_\_\_\_\_\_

Подпись: \_\_\_\_\_\_\_\_\_\_

Москва, 2020.

**Постановка задачи**

**Цель работы**

Целью является приобретение практических навыков в:

* Управлении серверами сообщений (№6)
* Применение отложенных вычислений (№7)
* Интеграция программных систем друг с другом (№8)

**Задание**

Реализовать распределенную систему по асинхронной обработке запросов. В данной распределенной системе должно существовать 2 вида узлов: «управляющий» и «вычислительный». Необходимо объединить данные узлы в соответствии с той топологией, которая определена вариантом. Связь между узлами необходимо осуществить при помощи технологии очередей сообщений. Также в данной системе необходимо предусмотреть проверку доступности узлов в соответствии с вариантом. При убийстве («kill -9») любого вычислительного узла система должна пытаться максимально сохранять свою работоспособность, а именно все дочерние узлы убитого узла могут стать недоступными, но родительские узлы должны сохранить свою работоспособность.

Топология: дерево общего вида

Набор команд: локальный таймер

Тип проверки доступности узлов: ping id

**Общие сведения о программе**

Для работы с очередями используется ZMQ, программа собирается при помощи CMake. Управляющий узел – server, вычислительные узлы – client. В программе используются следующие системные вызовы:

1. **kill** – убивает процесс с pid – первый аргумент и посылает сигнал – второй аргумент.
2. **zmq\_ctx\_new –** создает ZMQ контекст.
3. **zmq\_socket –** создает ZMQ сокет.
4. **zmq\_send** – отправляет сообщение на socket.
5. **zmq\_recv** – получает сообщение на socket.
6. **zmq\_bind** – принимает соединие к сокету.
7. **fork** – создает копию процесса.

**Общий метод и алгоритм решения**.

Клиент отправляет на сервер запрос. Если требуется создать новый вычислительный узел, то сервер создаёт дочерний процесс вызовом fork и заносит его pid в дерево общего вида. Если требуется передать информацию вычислительному узлу, то осуществляется проход по дереву и отправляется сообщение в узел. Исполняющий узел получает сообщение выполняет команду и отправляет ответ серверу, а сервер отправляет клиенту.

**Основные файлы программы**

**CMakeLists.txt:**

**cmake\_minimum\_required(VERSION 3.0.2)**

**project(lab6-8 VERSION 1.0 LANGUAGES C)**

**set(SERVER server)**

**set(CLIENT client)**

**set(SOURCE\_SERVER server.c)**

**set(SOURCE\_CLIENT client.c)**

**add\_library(Mess OBJECT Mess.h Mess.c)**

**add\_library(Tree OBJECT Tree.h Tree.c)**

**target\_link\_libraries(Tree PRIVATE pthread)**

**add\_executable(${SERVER} ${SOURCE\_SERVER} Mess.h Mess.c Tree.h Tree.c)**

**add\_executable(${CLIENT} ${SOURCE\_CLIENT} Mess.h Mess.c)**

**target\_link\_libraries(${SERVER} ${Mess} ${Tree} zmq pthread)**

**target\_link\_libraries(${CLIENT} ${Mess} zmq)**

**add\_definitions(-std=c99 -Wextra -Wall -Werror -Wno-sign-compare -Wno-unused-result -lzmq -lpthread -D\_POSIX\_C\_SOURCE)**

**server.c:**

**#include <string.h>**

**#include <stdio.h>**

**#include <unistd.h>**

**#include <stdbool.h>**

**#include <sys/wait.h>**

**#include <sys/time.h>**

**#include <signal.h>**

**#include "zmq.h"**

**#include "Mess.h"**

**#include "Tree.h"**

**Tree\* tree = NULL;**

**void termination (int code) {**

**if (tree) {**

**Destroy(tree);**

**}**

**exit(code);**

**}**

**typedef struct Pipes {**

**int pipe1[2];**

**int pipe2[2];**

**} Pipes;**

**long long CurrentTime() {**

**struct timeval te;**

**if (gettimeofday(&te, NULL) != 0) {**

**fprintf(stderr, "time error\n");**

**exit(-1);**

**}**

**long long milliseconds = te.tv\_sec \* 1000LL + te.tv\_usec / 1000;**

**return milliseconds;**

**}**

**void Timer (Pipes p) {**

**long long begin = 0, end = 0;**

**long long timer;**

**while (1) {**

**PARAM\_TYPE a;**

**read(p.pipe1[0], &a, sizeof(PARAM\_TYPE));**

**switch (a) {**

**case START:**

**begin = CurrentTime();**

**end = begin;**

**break;**

**case STOP:**

**end = CurrentTime();**

**break;**

**case TIME:**

**timer = end - begin;**

**write(p.pipe2[1], &timer, sizeof(long long));**

**break;**

**case CLOSE:**

**return;**

**default:**

**fprintf(stderr, "incorrect command, try again\n");**

**break;**

**}**

**}**

**}**

**int main () {**

**signal(SIGINT, termination);**

**signal(SIGSEGV, termination);**

**printf("Starting server...\n");**

**void\* context = zmq\_ctx\_new();**

**if (!context) {**

**fprintf(stderr, "zmq\_ctx\_new error\n");**

**exit(-1);**

**}**

**void\* respond = zmq\_socket(context, ZMQ\_PAIR);**

**if (!respond) {**

**fprintf(stderr, "zmq\_socket error\n");**

**exit(-1);**

**}**

**zmq\_bind(respond, "tcp://\*:4040");**

**Init(&tree, -1, 0, -1, NULL, NULL);**

**long long timer;**

**ERROR\_TYPE result;**

**while (1) {**

**Message mess;**

**zmq\_recv(respond, &mess, sizeof(Message), 0);**

**Pipes arg;**

**pid\_t pid;**

**int ping ;**

**Tree\* tmp;**

**switch (mess.command) {**

**case CREATE:**

**if (pipe(arg.pipe1) == -1) {**

**fprintf(stderr, "pipe1 error");**

**exit(-1);**

**}**

**if (pipe(arg.pipe2) == -1) {**

**fprintf(stderr, "pipe2 error");**

**exit(-1);**

**}**

**pid = fork();**

**switch (pid) {**

**case -1:**

**fprintf(stderr, "fork error\n");**

**exit(-1);**

**case 0:**

**Timer(arg);**

**return 0;**

**default:**

**break;**

**}**

**result = Add(tree, mess.parent, mess.id, pid, arg.pipe1, arg.pipe2);**

**if (result == SUCCESS) {**

**printf("created new process with id = %d and pid = %d\n", mess.id, pid);**

**}**

**break;**

**case REMOVE:**

**result = DeleteNode(tree, mess.id);**

**if (result == SUCCESS) {**

**printf("deleted node with id = %d and pid = %d\n", mess.id, pid);**

**}**

**break;**

**case EXEC:**

**tmp = Find(tree, mess.id);**

**if (!tmp) {**

**result = NODE\_NOT\_FOUND;**

**} else if (waitpid(tmp->pid, NULL, WNOHANG) != 0) {**

**result = NODE\_IS\_UNAVAILABLE;**

**} else {**

**result = SUCCESS;**

**write(tmp->pipe1[1], &mess.param, sizeof(PARAM\_TYPE));**

**if (mess.command == EXEC && mess.param == TIME) {**

**if (read(tmp->pipe2[0], &timer, sizeof(long long)) != sizeof(long long)) {**

**result = READ\_ERROR;**

**}**

**}**

**}**

**break;**

**case PING:**

**tmp = Find(tree, mess.id);**

**result = SUCCESS;**

**if (!tmp) {**

**result = NODE\_NOT\_FOUND;**

**} else if ((ping = waitpid(tmp->pid, NULL, WNOHANG)) != 0) {**

**ping = 0;**

**} else {**

**ping = 1;**

**}**

**break;**

**case EXIT:**

**break;**

**case UNKNOWN\_COMM:**

**//fprintf(stderr, "unknown command, try again\n");**

**break;**

**}**

**if (mess.command == EXIT) {break;}**

**zmq\_send(respond, (void\*)&result, sizeof(ERROR\_TYPE), 0);**

**if (result == SUCCESS) {**

**switch (mess.command) {**

**case CREATE:**

**zmq\_send(respond, &pid, sizeof(int),0);**

**break;**

**case REMOVE:**

**break;**

**case EXEC:**

**if (mess.param == TIME) {**

**zmq\_send(respond, &timer, sizeof(long long),0);**

**}**

**break;**

**case PING:**

**zmq\_send(respond, &ping, sizeof(int),0);**

**break;**

**default:**

**break;**

**}**

**}**

**}**

**printf("Closing...\n");**

**Destroy(tree);**

**zmq\_close(respond);**

**zmq\_ctx\_destroy(context);**

**return 0;**

**}**

**client.c:**

**#include <string.h>**

**#include <stdio.h>**

**#include <stdlib.h>**

**#include <stdbool.h>**

**#include "Mess.h"**

**#include "zmq.h"**

**void help () {**

**printf("\tcreate id [parent] -- создать узел\n");**

**printf("\tremove id -- удалить узел\n");**

**printf("\texec id [param] -- выполнить узел\n");**

**printf("\tping [id] -- проверка узла на доступность\n");**

**printf("\texit -- выход\n");**

**}**

**int main () {**

**printf("Client Starting…\n");**

**void\* context = zmq\_ctx\_new();**

**if (!context) {**

**fprintf(stderr, "zmq\_ctx\_new error\n");**

**exit(-1);**

**}**

**void\* request = zmq\_socket(context, ZMQ\_PAIR);**

**if (!request) {**

**fprintf(stderr, "zmq\_socket error\n");**

**exit(-2);**

**}**

**zmq\_connect(request, "tcp://localhost:4040");**

**help();**

**long long timer = 0;**

**ERROR\_TYPE result;**

**int pid, ping;**

**while (1) {**

**int id = 0, parent = 0;**

**char command[100] = {'\0'}, param[100] = {'\0'};**

**Message mess;**

**scanf("%s",command);**

**mess.command = CreateCommand(command);**

**switch (mess.command) {**

**case CREATE:**

**scanf("%d %d", &id, &parent);**

**break;**

**case REMOVE:**

**scanf("%d", &id);**

**break;**

**case EXEC:**

**scanf("%d %s", &id, param);**

**break;**

**case PING:**

**scanf("%d", &id);**

**break;**

**case EXIT:**

**break;**

**default:**

**fgets(command, 100, stdin);**

**break;**

**}**

**mess.param = CreateParameter(param);**

**mess.id = id;**

**mess.parent = parent;**

**zmq\_send(request, (void\*)&mess, sizeof(Message), 0);**

**if (mess.command == EXIT) {**

**break;**

**}**

**zmq\_recv(request, &result, sizeof(ERROR\_TYPE), 0);**

**if (result == SUCCESS) {**

**printf("OK");**

**switch (mess.command) {**

**case CREATE:**

**zmq\_recv(request, &pid, sizeof(int),0);**

**printf(": %d", pid);**

**break;**

**case EXEC:**

**printf(":%d", mess.id);**

**if (mess.param == TIME) {**

**zmq\_recv(request, &timer, sizeof(long long),0);**

**printf(":%lld.%lld", timer / 1000, timer % 1000);**

**}**

**break;**

**case PING:**

**zmq\_recv(request, &ping, sizeof(int),0);**

**printf(" %d", ping);**

**break;**

**default:**

**break;**

**}**

**printf("\n");**

**} else {**

**printf("ERROR:%d: ", mess.id);**

**switch (result) {**

**case ALREADY\_EXIST:**

**printf("node already exist\n");**

**break;**

**case PARENT\_NOT\_FOUND:**

**printf("parent not found\n");**

**break;**

**case PARENT\_IS\_UNAVAILABLE:**

**printf("parent is unavailable\n");**

**break;**

**case NODE\_NOT\_FOUND:**

**printf("node not found\n");**

**break;**

**case NODE\_IS\_UNAVAILABLE:**

**printf("node is unavailable\n");**

**break;**

**case READ\_ERROR:**

**printf("can't read from node\n");**

**break;**

**default:**

**break;**

**}**

**}**

**}**

**zmq\_close(request);**

**zmq\_ctx\_destroy(context);**

**return 0;**

**}**

**Tree.c:**

**#include "Tree.h"**

**void Init (Tree\*\* node, int parent\_id, int node\_id, pid\_t pid, int pipe1[2], int pipe2[2]) {**

**(\*node) = malloc(sizeof(Tree));**

**if (!(\*node)) {**

**fprintf(stderr, "malloc error\n");**

**exit(-1);**

**}**

**(\*node)->parent\_id = parent\_id;**

**(\*node)->id = node\_id;**

**(\*node)->pid = pid;**

**if (pipe1 && pipe2) {**

**memcpy((\*node)->pipe1, pipe1, sizeof(int) \* 2);**

**memcpy((\*node)->pipe2, pipe2, sizeof(int) \* 2);**

**}**

**(\*node)->son = NULL;**

**(\*node)->brother = NULL;**

**}**

**Tree\* Find (Tree\* root, int id) {**

**if (!root) {**

**return NULL;**

**}**

**if (root->id == id) {**

**return root;**

**}**

**root = root->son;**

**while (root) {**

**Tree\* tmp = root;**

**tmp = Find(tmp, id);**

**if (tmp) {**

**return tmp;**

**} else {**

**root = root->brother;**

**}**

**}**

**return NULL;**

**}**

**ERROR\_TYPE Add (Tree\* root, int parent\_id, int child\_id, pid\_t pid, int pipe1[2], int pipe2[2]) {**

**if (Find(root, child\_id)) {**

**return ALREADY\_EXIST;**

**}**

**Tree\* parent = Find(root, parent\_id);**

**if (!parent) {**

**return PARENT\_NOT\_FOUND;**

**}**

**Tree\* newEl;**

**Init(&newEl, parent\_id, child\_id, pid, pipe1, pipe2);**

**Tree\* son = parent->son;**

**if (!son) {**

**parent->son = newEl;**

**} else {**

**while (son->brother) {**

**son = son->brother;**

**}**

**son->brother = newEl;**

**}**

**return SUCCESS;**

**}**

**pid\_t GetPid (Tree\* root, int id) {**

**pid\_t pid = 0;**

**Tree\* tmp = Find(root, id);**

**if (tmp) {**

**pid = tmp->pid;**

**}**

**return pid;**

**}**

**ERROR\_TYPE DeleteNode (Tree\* root, int id) {**

**Tree\* tmp = Find(root, id);**

**if (!tmp) {**

**return NODE\_NOT\_FOUND;**

**}**

**PARAM\_TYPE param = CLOSE;**

**if (tmp->pid != -1) {**

**write(tmp->pipe1[1], &param, sizeof(PARAM\_TYPE));**

**waitpid(tmp->pid, NULL, 0);**

**close(tmp->pipe1[0]);**

**close(tmp->pipe1[1]);**

**close(tmp->pipe2[0]);**

**close(tmp->pipe2[1]);**

**}**

**if (!tmp->son && !tmp->brother) {**

**Tree\* parent = Find(root, tmp->parent\_id);**

**if (parent->son == tmp) {**

**parent->son = NULL;**

**} else {**

**parent = parent->son;**

**while (parent->brother != tmp) {**

**parent = parent->brother;**

**}**

**parent->brother = NULL;**

**}**

**free(tmp);**

**} else {**

**Tree\* last = tmp;**

**Tree\* parent = NULL;**

**int rrr = 0;**

**while (last->brother) {**

**parent = last;**

**last = last->brother;**

**}**

**while (last->son) {**

**parent = last;**

**last = last->son;**

**rrr = 1;**

**}**

**\*tmp = \*last;**

**free(last);**

**if (!rrr) {**

**parent->brother = NULL;**

**} else {**

**parent->son = NULL;**

**}**

**}**

**return SUCCESS;**

**}**

**void Destroy (Tree\* root) {**

**if (!root) {**

**return;**

**}**

**Destroy(root->brother);**

**Destroy(root->son);**

**PARAM\_TYPE param = CLOSE;**

**if (root->pid != -1) {**

**write(root->pipe1[1], &param, sizeof(PARAM\_TYPE));**

**waitpid(root->pid, NULL, 0);**

**close(root->pipe1[0]);**

**close(root->pipe1[1]);**

**close(root->pipe2[0]);**

**close(root->pipe2[1]);**

**}**

**free(root);**

**}**

**Tree.h:**

**#ifndef TREE\_H**

**#define TREE\_H**

**#include <stdio.h>**

**#include <stdlib.h>**

**#include <stdbool.h>**

**#include <unistd.h>**

**#include <string.h>**

**#include <pthread.h>**

**#include <signal.h>**

**#include <sys/types.h>**

**#include <sys/wait.h>**

**#include "Mess.h"**

**typedef struct Tree {**

**int parent\_id, id;**

**pid\_t pid;**

**int pipe1[2], pipe2[2];**

**struct Tree\* son;**

**struct Tree\* brother;**

**} Tree;**

**void Init (Tree\*\* node, int parent\_id, int node\_id, pid\_t pid, int pipe1[2], int pipe2[2]);**

**Tree\* Find (Tree\* root, int id);**

**ERROR\_TYPE Add (Tree\* root, int id\_parent, int id\_child, pid\_t pid, int pipe1[2], int pipe2[2]);**

**int GetTid (Tree\* root, int id);**

**ERROR\_TYPE DeleteNode (Tree\* root, int id);**

**void Destroy (Tree\* root);**

**#endif**

**Mess.c:**

**#include "Mess.h"**

**COMM\_TYPE CreateCommand (char\* command) {**

**if (strcmp(command, "create") == 0) {**

**return CREATE;**

**} else if (strcmp(command, "remove") == 0) {**

**return REMOVE;**

**} else if (strcmp(command, "exec") == 0) {**

**return EXEC;**

**} else if (strcmp(command, "ping") == 0) {**

**return PING;**

**} else if (strcmp(command, "exit") == 0) {**

**return EXIT;**

**} else {**

**return UNKNOWN\_COMM;**

**}**

**}**

**PARAM\_TYPE CreateParameter (char\* param) {**

**if (strcmp(param, "start") == 0) {**

**return START;**

**} else if (strcmp(param, "stop") == 0) {**

**return STOP;**

**} else if (strcmp(param, "time") == 0) {**

**return TIME;**

**} else {**

**return UNKNOWN\_PARAM;**

**}**

**}**

**Mess.h:**

**#ifndef MESS\_H**

**#define MESS\_H**

**#include <string.h>**

**typedef enum ERROR\_TYPE {**

**SUCCESS = 0,**

**ALREADY\_EXIST,**

**PARENT\_NOT\_FOUND,**

**PARENT\_IS\_UNAVAILABLE,**

**NODE\_NOT\_FOUND,**

**NODE\_IS\_UNAVAILABLE,**

**READ\_ERROR,**

**} ERROR\_TYPE;**

**typedef enum COMM\_TYPE {**

**CREATE = 0,**

**REMOVE,**

**EXEC,**

**PING,**

**EXIT,**

**UNKNOWN\_COMM**

**} COMM\_TYPE;**

**typedef enum PARAM\_TYPE {**

**START = 0,**

**STOP,**

**TIME,**

**CLOSE,**

**UNKNOWN\_PARAM**

**} PARAM\_TYPE;**

**typedef struct Message {**

**COMM\_TYPE command;**

**PARAM\_TYPE param;**

**int id, parent;**

**} Message;**

**COMM\_TYPE CreateCommand (char\* command);**

**PARAM\_TYPE CreateParameter (char\* param);**

**#endif**

**Пример работы**

**aleksandr201@LAPTOP-M1VJISD6:~/OS/lab6-8/src$ ./client**

**Client Starting…**

**create id [parent] -- создать узел**

**remove id -- удалить узел**

**exec id [param] -- выполнить узел**

**ping [id] -- проверка узла на доступность**

**exit -- выход**

**create 1 0**

**OK: 428**

**exec 1 start**

**OK:1**

**exec 1 stop**

**OK:1**

**exec 1 time**

**OK:1:3.504**

**create 2 1**

**OK: 429**

**remove 1**

**OK**

**exec 1 time**

**ERROR:1: node not found**

**exec 2 time**

**OK:2:0.0**

**exit**

**Вывод**

Для общение в архитектуре клиент-сервер существуют очереди сообщений, при помощи них можно достаточно не сложно организовать обмен информацией. ZMQ является быстрой и простой библиотекой для обмена сообщениями между сокетами. Такие структуры, как деревья хорошо подходят для хранения информации о клиентах и сервере. Так же нужно уметь проверять доступны ли сейчас вычислительные узлы, потому что они могут быть уничтожены внешними программами.