****Московский Авиационный Институт

(Национальный Исследовательский Университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

**Лабораторная работ №6 по курсу**

**«Операционные системы»**

Группа: М80 – 207Б – 18

Студент: Син Денис Дмитриевич

Преподаватель: Миронов Евгений Сергеевич

Оценка: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Дата: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Москва, 2019.

**Содержание**

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

**Постановка задачи**.

Реализовать распределенную систему по асинхронной̆ обработке запросов. В данной распределённой̆ 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. Управляющий узел отвечает за ввод команд от пользователя и отправку этих команд на вычислительные узлы. Список основных поддерживаемых команд:

1. Создание нового вычислительного узла create id parent
2. Удаление существующего вычислительного узла remove id
3. Исполнение команды на вычислительном узле exec id [params]
4. Проверка доступности узлов

Вариант 52

1. Тип топологии: дерево общего вида
2. Тип вычислительных команд: подсчет суммы n чисел exec id n
3. Тип проверки доступности узлов: ping id

**Общие сведения о программе**

Проект состоит из двух исполняемых файлов calc\_node и app. Исполняемый файл app является главным, при его запуске создается управляющий узел, который принимает команды от пользователя. Исполняемый файл calc\_node используется как аргумент для системного вызова execl. Данный файл является реализацией вычислительного узла. Исполняемый файл app компилируется из файла main.cpp, а исполняемый файл calc\_node компилируется из файла calc\_node.cpp.

Для удобства были реализованы классы ManageNode и CalcNode, которые являются наследниками класса BaseNode. Данные классы являются абстракциями над узлами и их реализации находятся в файле node.h.

Чтобы легче и понятнее было работать с данными, которые перемещаются между узлами были реализован класс Message, который инкапсулирует в себя сереализацию объектов, прием и отправку сообщений.

**Общий метод и алгоритм решения**.

Реализацию можно разделить на две части: выбор архитектуры взаимодействия узлов и написание четырех команд, которые может вводить пользователь. В программе все команды выполняются асинхронно, другими словами, пользователь может заранее послать на выполнение сразу несколько команд на любые узлы, это достигается с помощью много поточности и очередей сообщений. В управляющем узле работают 4 потока

1. Поток, отвечающий за ввод пользователя, парсинг ввода и передачу команд в поток, отвечающий за обработку сообщений
2. Поток, отвечающий за получение команды из очереди и обработки этой команды, после получения результата команды, он отправляет результат в поток, отвечающий за вывод.
3. Поток, отвечающий за вывод.
4. Поток, отвечающий за регистрацию дочерних узлов.

Потоки 1, 2 и 3 работают с помощью сокетов zmq по следующей схеме:

push 🡪 pull 🡪 handle 🡪 push 🡪 pull. Все остальные вычислительные узлы работают на сокетах типа req/rep. И в каждом вычислительном узле находится по 2 reply сокета, которые прослушиваются в отдельных потоках, первый сокет отвечает за регистрацию дочерних узлов, а второй за прием команд и их обработку. При абсолютно асинхронных запросах важно обработать ситуации, когда следующий запрос зависит от завершения предыдущего, такое может быть при команде create. В этом случае необходимо синхронизировать потоки, которые отвечают за регистрацию дочерних узлов и за прем следующий команды из очереди. Это делается с помощью условной переменной и mutex.

**Демонстрация работы программы.**

~/CSinDenis/university/sem3/OperatingSystems/lab06/cmake-build-debug(master\*) » cat test4

create 1 -1

create 2 -1

create 3 -1

create 4 -1

create 5 -1

create 6 -1

create 7 -1

create 8 -1

create 9 -1

create 10 -1

create 11 1

create 111 11

create 1111 111

remove 1111

remove 111

create 81 8

create 82 8

create 83 8

ping 83

ping 82

exec 82 10 1 2 3 4 5 6 7 8 9 10

ping 1

ping 2

ping 3

ping 4

ping 5

ping 6

ping 7

ping 8

ping 9

ping 10

exec 1 5 1 2 3 4 5

exec 2 5 1 2 3 4 5

exec 3 5 1 2 3 4 5

exec 4 5 1 2 3 4 5

exec 5 5 1 2 3 4 5

exec 6 5 1 2 3 4 5

exec 7 5 1 2 3 4 5

exec 8 5 1 2 3 4 5

exec 9 5 1 2 3 4 5

exec 10 5 1 2 3 4 5

quit

~/CSinDenis/university/sem3/OperatingSystems/lab06/cmake-build-debug(master\*) »./app < test4

Ok: 6959

Ok: 6960

Ok: 6961

Ok: 6962

Ok: 6963

Ok: 6964

Ok: 6965

Ok: 6966

Ok: 6967

Ok: 6968

Ok: 6969

Ok: 6970

Ok: 6971

Ok

Ok

Ok: 6972

Ok: 6973

Ok: 6974

Ok: 1

Ok: 1

Ok: 82: 55

Ok: 1

Ok: 1

Ok: 1

Ok: 1

Ok: 1

Ok: 1

Ok: 1

Ok: 1

Ok: 1

Ok: 1

Ok: 1: 15

Ok: 2: 15

Ok: 3: 15

Ok: 4: 15

Ok: 5: 15

Ok: 6: 15

Ok: 7: 15

Ok: 8: 15

Ok: 9: 15

Ok: 10: 15

Delete all nodes

All nodes is killed

**Вывод**

Данная лабораторная работа довольно сложная и объемная. При выполнении заметны две основные проблемы. Первая заключается в том, что сразу не получается продумать все и поэтому в какой-то момент необходимо переделывать все по новой. В моём случае переписывать код пришлось около 3 раз. Вторая проблема заключается в реализации команд, в случае с топологией дерева общего вида в виду того, что мышление идет в терминах процессов не сразу удается понять, что на самом деле пишутся рекурсивные алгоритмы. И уже в самом конце я столкнулся с проблемой синхронизации потоков, но она решалась несложно, необходимо было просто применить один из примитивов синхронизации.

Хоть лабораторная работа была действительно сложной (скорее всего это самая сложная лабораторная работа в семестре из всех предметов), но здесь сразу можно применить знания, полученные в ходе выполнения предыдущих лабораторных работ, так как здесь и многопоточность, и межпроцессорное взаимодействие, основанное на очередях сообщений, и синхронизация потоков, а также асинхронность. И ко всему этому необходимо разбираться с дополнительной библиотекой (zmq). Также стоит отметить, что так как работа выполнялась на c++, то пригодились знания из курса ООП, так как там тоже была лабораторная работа на асинхронность и многопоточное программирование.

**Трейсы dtruss.**

CALL COUNT

access 1 sendto 448

bind 1 recvfrom 569

connect\_nocancel 1 kevent 1206

csrctl 1 poll 114923

getentropy 1

getrlimit 1

issetugid 1

kevent\_qos 1

listen 1

open\_nocancel 1

pread 1

read\_nocancel 1

shared\_region\_check\_np 1

shm\_open 1

sigaction 1

sysctlbyname 1

\_\_disable\_threadsignal 2

\_\_pthread\_sigmask 2

bsdthread\_ctl 2

bsdthread\_terminate 2

fcntl\_nocancel 2

sysctl 2

fstat64 3

ioctl 3

kevent\_id 3

psynch\_mutexdrop 3

psynch\_mutexwait 3

mmap 4

open 4

proc\_info 4

bsdthread\_create 5

mprotect 8

ulock\_wait 9

ulock\_wake 9

workq\_kernreturn 9

accept 10

execve 10

fork 10

psynch\_cvsignal 10

psynch\_cvwait 10

\_\_mac\_syscall 11

bsdthread\_register 11

csops 12

getpid 12

write\_nocancel 12

thread\_selfid 17

getsockopt 32

connect 33

getsockname 34

sendmsg\_nocancel 35

socket 35

kqueue 37

getpeername 42

stat64 43

socketpair 67

select\_nocancel 69

sendto\_nocancel 69

close 92

recvfrom\_nocancel 102

close\_nocancel 104

setsockopt 115

\_\_semwait\_signal 260

fcntl 370