**Пакет: инструменты**

Содержание компонентов:

1) Журнал.

2) Таймер.

3) Генератор случайных значений.

1) Журнал.

Цель: Вывод и обработка временных замеров.

Как: Сохранять в два (первичный и вторичный) огромных статических массива длинные целые числа.

Проверка: Вывод в консоль + файл, тестовых значений.

Код:

unsigned long primaryIndex;

unsigned long secondaryIndex;

unsigned long primaryMeasuring[LOGGER\_SIZE];

unsigned long secondaryMeasuring[LOGGER\_SIZE];

void reset() {

primaryIndex = 0;

secondaryIndex = 0;

}

void logPrimary(unsigned long time) {

if (LOGGER\_SIZE < primaryIndex) { printf("%s", "Primary logger is out of range!\n"); return; }

primaryMeasuring[primaryIndex++] = time;

}

void logSecondary(unsigned long time) {

if (LOGGER\_SIZE < secondaryIndex) { printf("%s", "Secondary logger is out of range!\n"); return; }

secondaryMeasuring[secondaryIndex++] = time;

}

void printArray(unsigned long\* array, long length) {

long index = 0;

for (; index < length; index++) { printf("%lu ", array[index]); }

}

unsigned long sum(unsigned long\* array, long length) {

unsigned long result = 0;

long index = 0;

for (; index < length; index++)

{ result += array[index]; }

return result;

}

unsigned long min(unsigned long\* array, long length) {

unsigned long result = array[0];

long index = 0;

for (; index < length; index++)

if (array[index] < result)

{ result = array[index]; }

return result;

}

unsigned long max(unsigned long\* array,

long length) {

unsigned long result = array[0];

long index = 0;

for (; index < length; index++)

if (result < array[index])

{ result = array[index]; }

return result;

}

void fileLoggerOutput(char\* name, unsigned long\* array, unsigned long length, char\* measuringName,

unsigned long maxValue, unsigned long averageValue, unsigned long minValue) {

FILE\* file;

#define fileNameLength 80

char fileName[fileNameLength];

sprintf(fileName, "%s [%s] (%lu, %lu, %lu) %lu.txt", name, measuringName, minValue, averageValue,

maxValue, length);

file = fopen(fileName, "wb");

if (NULL == file) {

printf("%s:%s\n", "Can't create file ", fileName);

return;

}

int index = 0;

for (; index < length; index++) {fprintf(file, "%lu ", array[index]); }

fclose(file);

}

void (\*showMeasure)(char\* name, unsigned long\* array, unsigned long length, char\* measuringName,

unsigned long maxValue, unsigned long averageValue, unsigned long minValue) = &fileLoggerOutput;

void show(char\* name) {

#define calcAndShow(name, array, length, measuringName) showMeasure(name, array, length, measuringName, max(array, length), sum(array, length) / (0 == length ? 1 : length), min(array, length))

calcAndShow(name, primaryMeasuring, primaryIndex, "Primary time: ");

calcAndShow(name, secondaryMeasuring, secondaryIndex, "Secondary time: ");

}

2) Таймер.

Цель: Замер временных промежутков.

Как: Получить текущее нано время, и обработать.

Проверка: Циклический замер математического выражения.

Код:

#define BILLON 1000000000

unsigned long getULongNano() {

struct timespec currentTime;

clock\_gettime(TIMER\_ABSTIME, &currentTime);

return (unsigned long) currentTime.tv\_nsec;

}

unsigned int getUIntNano() {

return (unsigned int) getULongNano();

}

unsigned char getUCharNano() {

return (unsigned char) getULongNano();

}

unsigned long calcOffsetULong(unsigned long start) {

unsigned long end = getULongNano();

return start < end ? end - start : end + BILLON - start;

}

3) Генератор случайных значений

Цель: Получить случайное значение

Как: На основании текущего нановремени, и регистровых сдвигов.

Проверка: Циклическая генерация значений. Проверка не выхода за границы.

Код:

#define BIT\_IN\_BYTE 8

char int\_sz = sizeof(int);

char short\_sz = sizeof(short);

unsigned char generateChar() {

return getUCharNano();

}

//return generateChar() << 24 | generateChar() << 16 | generateChar() << 8 | generateChar();

unsigned short generateShort() {

unsigned int result = 0;

char index = short\_sz;

while (index--)

{ result |= generateChar() << (BIT\_IN\_BYTE \* index); }

return result;

}

unsigned int generateInt() {

unsigned int result = 0;

char index = int\_sz;

while (index--)

{ result |= generateChar() << (BIT\_IN\_BYTE \* index); }

return result;

}

unsigned int generateIntByWidth(unsigned int width) {

return width == 0 ? 0 : generateInt() % width;

}

unsigned int generateIntByRange(unsigned int start, unsigned int end) {

if (end < start) {

unsigned int swap = start;

start = end;

end = swap;

}

return generateIntByWidth(end - start) + start;

}

**Пакет: анализатор памяти**

Содержание компонентов:

1) Генератор размеров.

2) Цикличный список.

3) Многопоточный обработчик.

4) Анализатор памяти.

1) Генератор размеров

Цель: Получить следующее значение, в зависимости от выбранной стратегии выделения размеров переменных.

Как: Сгенерировать индекс массива, для фиксированных значений. Сгенерировать из диапазона, для областного выделения.

Проверка: Проверка не выхода за границы.

Код:

//Блоки выделяются любого размера в пределах от 16 байт до 16Кбайт случайным образом при каждом выделении

int getRange() {

return generateIntByRange(MIN\_RANGE, MAX\_RANGE);

}

int fixSizes[] = FIX\_SIZES

//Блоки выделяются одного из перечисленных размеров случайным образом при каждом выделении: 16 байт, 64 байта, 256 байт, 1 Кбайт, 4 КБайта, 16 Кбайт.

int getFix() {

return fixSizes[generateIntByWidth(

FIX\_SIZES\_LENGTH)];

}

2) Циклический список

Цель: Выделенные переменных требуется хранить здесь.

Как: Циклическое получение и заполнение массива по средством указателя для получения и заполнения.

Проверка: Пограничные проверки – выделить больше возможного, получить больше чем положено.

Код:

long putIndex = 0;

long getIndex = 0;

char\* list[MAX\_VARIABLES\_COUNT];

#define validation(variable) if (MAX\_VARIABLES\_COUNT <= ++variable) variable = 0;

void put(char\* variable) {

validation(putIndex);

list[putIndex] = variable;

}

char\* next() {

validation(getIndex);

return list[getIndex];

}

3) Многопоточный обработчик

Цель: Запуск двух потоков – выделения и освобождения памяти.

Как: На основании входных параметров – указатель на основное, вторичное, пред и послеитеративное действия, а так же количество этих итераций и сессий, формируется аспекто-ориентированный модуль. Основное и вторичное действие синхронизируется с помощью защелок, т.к. они в двух потоках.

Проверка: Создание и вызов ложных действий (mock), проверка их запуска.

Код:

typedef pthread\_mutex\_t Mutex;

Mutex locka;

Mutex lockb;

void initMutex(Mutex\* mutex) {

int result = pthread\_mutex\_init(mutex, NULL);

if (0 != result) { printf("%s%d\n", "Mutex init error: ", result); }

}

void unlockMutex(Mutex\* mutex) {

int result = pthread\_mutex\_unlock(mutex);

if (0 != result) { printf("%s%d\n", "Mutex unlock error: ", result); }

}

void lockMutex(Mutex\* mutex) {

int result = pthread\_mutex\_lock(mutex);

if (0 != result) { printf("%s%d\n", "Mutex lock error: ", result); }

}

void initThread(pthread\_t\* thread, void\* (\*method)()) {

int err = pthread\_create(thread, NULL, method, NULL);

if (err != 0) { printf("%s[%s]\n", "Can't create thread: ", strerror(err)); }

}

int (\*mainAction)();

void (\*additionalAction)();

long seesionCount;

long iterationCount;

void (\*preIterationAction)();

void (\*postIterationAction)();

char isCrash = NO\_CRASH;

void threadTaskA() {

while (CRASH != isCrash && seesionCount--) {

preIterationAction();

long index = iterationCount;

while (CRASH != isCrash && index--) {

lockMutex(&lockb);

isCrash = mainAction();

unlockMutex(&locka);

}

postIterationAction();

}

}

void threadTaskB() {

while (CRASH != isCrash ) {

lockMutex(&locka);

additionalAction();

unlockMutex(&lockb);

}

}

void tasksRun(int (\*\_mainAction)(),

void (\*\_additionalAction)(),

void (\*\_preIterationAction)(),

void (\*\_postIterationAction)(),

long \_seesionCount, long \_iterationCount) {

mainAction = \_mainAction;

additionalAction = \_additionalAction;

preIterationAction = \_preIterationAction;

postIterationAction = \_postIterationAction;

seesionCount = \_seesionCount;

iterationCount = \_iterationCount;

//if variable isCrash contain value CRASH, that's bad.

isCrash = NO\_CRASH;

initMutex(&locka);

initMutex(&lockb);

lockMutex(&locka);

pthread\_t threadA;

pthread\_t threadB;

initThread(&threadA, (void\*) &threadTaskA);

initThread(&threadB, (void\*) &threadTaskB);

pthread\_join(threadA, NULL);

pthread\_join(threadB, NULL);

}

4) Анализатор памяти

Цель: Зауск анализа по двум стратегиям выделения и обработка/вывод результата.

Как: Композиция предыдущих модулей. Формирование основных, вторичных итд действий.

Проверка: Запуск программы.

Код:

int (\*generate)();

int parent() {

int size = generate();

unsigned long startTime = getULongNano();

char\* allocateVariable = allocate(size);

unsigned long time = calcOffsetULong(startTime);

int result = CRASH;

if (NULL != allocateVariable) {

logPrimary(time);

put(allocateVariable);

result = NO\_CRASH;

}

return result;

}

void child() {

if (generateIntByWidth(REMOVE\_MAX\_CHANCE) < REMOVE\_CHANCE) {

char\* toRemove = next();

int result = 0;

unsigned long startTime = getULongNano();

result = removeVar(toRemove);

unsigned long time = calcOffsetULong(startTime);

if (VARIABLE\_REMOVE\_SUCCESS == result) { logSecondary(time); }

else { printf("%s", "Remove issue occurs\n"); }

}

}

void preIteration() {

reset();

}

char\* analysisName;

void postIteration() {

show(analysisName);

}

void startAnalyze(int (\*generateMethod)(), char\* msg) {

analysisName = msg;

int variablesSize[] = ANALYZE\_VARIABLES\_SIZE;

int variablesCount[] = ANALYZE\_VARIABLES\_COUNT;

char subheapCount = ANALYZE\_SUBHEAPS\_COUNT;

int memoryManagerInitResult = init(variablesSize, variablesCount, subheapCount);

if (INITIAL\_SUCCESS != memoryManagerInitResult) {

printf("%s%d\n", "\nIncorrect initial code: ", memoryManagerInitResult);

return;

}

generate = generateMethod;

tasksRun(&parent, &child, &preIteration, &postIteration, SESSION\_COUNT, ITERATION\_COUNT);

}

void start() {

setFileLoggerOutput();

startAnalyze(&getFix, "Fix");

startAnalyze(&getRange, "Range ");

}

Результат работы анализатора:

При шансе удаления 97.5 процента.

Флагах O3 – макс оптимизация компилятора.

#define ANALYZE\_VARIABLES\_SIZE {16, 64, 256, 1024, 4096, 16384}

#define ANALYZE\_VARIABLES\_COUNT {480, 480, 480, 480, 480, 480}

#define ANALYZE\_SUBHEAPS\_COUNT 6

|  |  |  |
| --- | --- | --- |
| (Мин/Сред/Макс) Количество | Фиксированный | Область |
| Выделение | (87, 866, 332480) 100000  (139, 835, 8672) 3414 | (251, 1450, 3067) 715 |
| Освобождение | (60, 151, 27876) 97405  (67, 114, 305) 3318 | (110, 135, 278) 701 |

**Пакет: коллекции**

Содержание компонентов:

0) Обработчик ключа.

1) Хэш таблица.

2) АВЛ дерево.

3) Двоичное дерево.

Компонент КЛЮЧ

struct Key {

unsigned int ip;

unsigned short port;

} \_\_attribute\_\_((packed));

typedef struct Key Key;

Компоненты реализуют интерфейс КОЛЛЕКЦИЯ.

struct Collection {

char (\*find)(Key\* key);

char (\*insert)(Key\* key);

char (\*alter)(Key\* source, Key\* target);

char (\*initCollection)(int size);

};

typedef struct Collection Collection;

Функциональность:

-Инициализация коллекции (Ключи и узлы храняться в менеджере памяти)

-Вставка ключа

-Поиск ключа

-Замена ключа

0) Обработчик ключа.

Цель: Сравнение и вычисление хэш значения для ключа.

Как: На основании полей структуры

Проверка: Пограничная проверка. Матрица выходных значений(DDT).

Код:

void charArrayToKey(Key\* key, char\* array) {

unsigned int\* ip = (unsigned int\*) array;

unsigned short\* port = (unsigned short\*) &array[4];

key->ip = \*ip;

key->port = \*port;

}

void keyToCharArray(Key\* key, char\* array) {

unsigned int\* ip = (unsigned int\*) array;

unsigned short\* port = (unsigned short\*)

&array[4];

\*ip = key->ip;

\*port = key->port;

}

// -1 : key1 < key2

// 0 : key1 eq key2

// 1 : key1 > key2

char compareKeys(Key\* key1, Key\* key2) {

if (key1 == key2) { return 0; }

else if (NULL == key1) { return -1; }

else if (NULL == key2) { return 1; }

//Compare ip: if key1.ip less that key2.ip than -1 etc

char result = (key1->ip < key2->ip ? -1 : (key1->ip == key2->ip ? 0 : 1) );

//Compare port, only if ips eq

result = (0 == result ? (key1->port < key2->port ? -1 : key1->port == key2->port ? 0 : 1) : result);

return result;

}

unsigned int hashCode(Key\* key) {

return NULL == key ? 0 : key->ip;

}

**1) Хэш таблица**

**Служебная библиотека:**

char addLinkedListNode(LinkedListNode\* root, LinkedListNode\* next) {

if (NULL == root)

{ return LINKED\_LIST\_NODE\_ROOT\_IS\_NULL; }

LinkedListNode\* currentNode = root;

while (NULL != currentNode->next)

{ currentNode = currentNode->next; }

currentNode->next = next;

return LINKED\_LIST\_NODE\_SUCCESS\_OP;

}

char removeLinkedListNode(LinkedListNode\* root, LinkedListNode\* target) {

if (target == root)

{ return LINKED\_LIST\_CANT\_REMOVE\_ROOT; }

LinkedListNode\* currentNode = root;

while (NULL != currentNode) {

if (target == currentNode->next) {

currentNode->next = target->next;

return LINKED\_LIST\_NODE\_SUCCESS\_OP;

}

currentNode = currentNode->next;

}

return LINKED\_LIST\_NODE\_NOT\_FOUND;

}

LinkedListNode\* findLinkedListNodeByKey(LinkedListNode\* root, Key\* key) {

if (NULL == key) { return NULL; }

LinkedListNode\* current = root;

while (NULL != current) {

if (0 == compareKeys(key, current->key))

{ return current; }

current = current->next;

}

return NULL;

}

**Верхний слой:**

LinkedListNode\*\* hashTable;

int hashTableSize;

int getIndexByHashKey(Key\* key) { return hashCode(key) % hashTableSize; }

LinkedListNode\* getLinkedListNode(Key\* key) {

int hashTableIndex = getIndexByHashKey(key);

LinkedListNode\* root = hashTable[hashTableIndex];

return findLinkedListNodeByKey(root, key);

}

char insertHashTable(Key\* key) {

LinkedListNode\* newNode = (LinkedListNode\*) allocate(sizeof(LinkedListNode));

Key\* copyKey = (Key\*) allocate(sizeof(Key));

if (NULL == copyKey || NULL == newNode)

{ return INSERT\_FAIL; }

copyKey->port = key->port;

copyKey->ip = key->ip;

newNode->key = copyKey;

int hashTableIndex = getIndexByHashKey(copyKey);

LinkedListNode\* oldRoot = hashTable[hashTableIndex];

hashTable[hashTableIndex] = newNode;

newNode->next = oldRoot;

return INSERT\_SUCCESS;

}

char findHashTable(Key\* key) {

if (NULL == key) { return FIND\_FAIL; }

LinkedListNode\* result = getLinkedListNode(key);

return NULL == result ? FIND\_FAIL : FIND\_SUCCESS;

}

char alterHashTable(Key\* source, Key\* target) {

LinkedListNode\* sourceNode;

if (NULL == source || NULL == (sourceNode = getLinkedListNode(source))) {

return ALTER\_FAIL;

}

int hashTableIndex = getIndexByHashKey(source);

LinkedListNode\* root = hashTable[hashTableIndex];

char removeStatus = removeLinkedListNode(root, sourceNode);

switch (removeStatus) {

case LINKED\_LIST\_CANT\_REMOVE\_ROOT:

hashTable[hashTableIndex] = sourceNode->next;

break;

case LINKED\_LIST\_NODE\_NOT\_FOUND:

printf("%s", "Key and node not found!\n");

return ALTER\_FAIL;

break;

}

if (NULL == target) {

printf("%s", "Remove in hash table\n");

removeVar((char\*) sourceNode->key);

removeVar((char\*) sourceNode);

return REMOVE\_SUCCESS;

}

Key\* keyContainInSource = sourceNode->key;

keyContainInSource->ip = target->ip;

keyContainInSource->port = target->port;

hashTableIndex = getIndexByHashKey(target);

root = hashTable[hashTableIndex];

hashTable[hashTableIndex] = sourceNode;

sourceNode->next = root;

return ALTER\_SUCCESS;

}

char initHashTable(int size) {

int variablesSize[3] = {sizeof(Key), sizeof(LinkedListNode), size \* sizeof(LinkedListNode\*)};

int variablesCount[3] = {size, size, 1};

char subheapCount = 3;

if (INITIAL\_SUCCESS != init(variablesSize, variablesCount, subheapCount)) {

printf("%s", "Memory manager can't init\n");

exit(-1);

}

hashTableSize = size;

hashTable = (LinkedListNode\*\*) allocate(size \* sizeof(LinkedListNode\*));

return (NULL == hashTable) ? HASH\_TABLE\_INIT\_FAIL : HASH\_TABLE\_INIT\_SUCCESS;

}

**1) АВЛ дерево**

**Служебная библиотека:**

char height(AVLNode\* root) {

return NULL == root ? 0 : root->height;

}

char bfactor(AVLNode\* root) {

return height(root->right) - height(root->left);

}

void fixHeight(AVLNode\* root) {

char hl = height(root->left);

char hr = height(root->right);

root->height = (hr < hl ? hl : hr) + 1;

}

AVLNode\* rotateRight(AVLNode\* root) {

AVLNode\* q = root->left;

root->left = q->right;

q->right = root;

fixHeight(root);

fixHeight(q);

return q;

}

AVLNode\* rotateLeft(AVLNode\* q) {

AVLNode\* root = q->right;

q->right = root->left;

root->left = q;

fixHeight(q);

fixHeight(root);

return root;

}

AVLNode\* balance(AVLNode\* root) {

fixHeight(root);

if ( bfactor(root) == 2 ) {

if ( bfactor(root->right) < 0 )

{ root->right = rotateRight(root->right); }

return rotateLeft(root);

}

if ( bfactor(root) == -2 ) {

if ( bfactor(root->left) > 0 )

{ root->left = rotateLeft(root->left); }

return rotateRight(root);

}

return root;

}

AVLNode\* insertAVLByNode(AVLNode\* root, AVLNode\* target) {

if (NULL == root) { return target; }

if (1 == compareKeys(root->key, target->key))

{ root->left = insertAVLByNode(root->left, target); }

else

{ root->right = insertAVLByNode(root->right, target); }

return balance(root);

}

AVLNode\* findMin(AVLNode\* root) {

return NULL != root->left ? findMin(root->left) : root;

}

AVLNode\* removeMin(AVLNode\* root) {

if (NULL == root->left)

{ return root->right; }

root->left = removeMin(root->left);

return balance(root);

}

AVLNode\* removeAVLNode(AVLNode\* root, Key\* key, void (\*deleteOp)(AVLNode\*)) {

if (NULL == root) { return NULL; }

if (NULL == key) { return root; }

switch (compareKeys(root->key, key)) {

case 1: root->left = removeAVLNode(root->left, key, deleteOp); break;

case -1: root->right = removeAVLNode(root->right, key, deleteOp); break;

case 0:

AVLNode\* left = root->left;

AVLNode\* right = root->right;

deleteOp(root);

if (NULL == right) { return left; }

AVLNode\* min = findMin(right);

min->right = removeMin(right);

min->left = left;

return balance(min);

break;

}

return balance(root);

}

AVLNode\* findAVLNode(AVLNode\* root, Key\* key) {

if (NULL == root) { return NULL; }

AVLNode\* result;

switch (compareKeys(root->key, key)) {

case 1: result = findAVLNode(root->left, key); break;

case -1: result = findAVLNode(root->right, key); break;

case 0: result = root; break;

}

return result;

}

**Верхний слой:**

AVLNode\* root;

char insertAVLTree(Key\* key) {

if (NULL == key)

{ return INSERT\_FAIL; }

Key\* copyKey = (Key\*) allocate(sizeof(Key));

AVLNode\* newNode = (AVLNode\*) allocate(sizeof(

AVLNode));

if (NULL == copyKey || NULL == newNode)

{ return INSERT\_FAIL; }

copyKey->port = key->port;

copyKey->ip = key->ip;

newNode->key = copyKey;

root = insertAVLByNode(root, newNode);

return INSERT\_SUCCESS;

}

char findAVLTree(Key\* key) {

if (NULL == key) { return FIND\_FAIL; }

AVLNode\* result = findAVLNode(root, key);

return NULL == result ? FIND\_FAIL : FIND\_SUCCESS;

}

char alterAVLTree(Key\* source, Key\* target) {

if (NULL == source || NULL == root)

{ return ALTER\_FAIL; }

AVLNode\* srcNode;

void deleteOp(AVLNode \* toDelete) {

srcNode = toDelete;

}

root = removeAVLNode(root, source, &deleteOp);

if (NULL == target) {

removeVar((char\*) srcNode->key);

removeVar((char\*) srcNode);

return REMOVE\_SUCCESS;

}

srcNode->key->ip = target->ip;

srcNode->key->port = target->port;

srcNode->left = NULL;

srcNode->right = NULL;

root = insertAVLByNode(root, srcNode);

return ALTER\_SUCCESS;

}

char initAVLTree(int size) {

root = NULL;

int variablesSize[2] = {sizeof(Key), sizeof(AVLNode)};

int variablesCount[2] = {size, size};

char subheapCount = 2;

char result = init(variablesSize, variablesCount, subheapCount);

return (INITIAL\_SUCCESS == result) ? HASH\_TABLE\_INIT\_SUCCESS :

HASH\_TABLE\_INIT\_FAIL;

}

**3) Двоичное дерево**

**Служебная библиотека:**

BNode\* findBNodeInsideRoot(BNode\* root, Key\* key) {

if (NULL == key) { return NULL; }

BNode\* currentNode = root;

while (NULL != currentNode) {

switch (compareKeys(currentNode->key, key)) {

case 1:

currentNode = currentNode->left;

break;

case -1:

currentNode = currentNode->right;

break;

case 0:

return currentNode;

break;

}

}

return NULL;

}

char insertBNodeInsideRoot(BNode\* root, BNode\* target) {

BNode\* currentNode = root;

while (NULL != currentNode) {

switch (compareKeys(currentNode->key, target->key)) {

case 1:

if (NULL == currentNode->left) {

currentNode->left = target;

return TREE\_INSERT\_OK;

}

currentNode = currentNode->left;

break;

case -1:

if (NULL == currentNode->right) {

currentNode->right = target;

return TREE\_INSERT\_OK;

}

currentNode = currentNode->right;

break;

case 0:

return TREE\_INSERT\_NOT\_UNIQ;

break;

}

}

return -5;

}

BNode\*\* getBNodeParentAnFindTarget(BNode\* root, Key\* key) {

BNode\*\* currentParentPointer = NULL;

BNode\* currentChild = NULL;

//Return if root == target

switch (compareKeys(root->key, key)) {

case 1:

currentParentPointer = &root->left;

break;

case -1:

currentParentPointer = &root->right;

break;

case 0:

return NULL;

break;

default :

printf("%s", "Compare error #10\n");

}

currentChild = \*currentParentPointer;

while (NULL != (currentChild = \*currentParentPointer)) {

switch (compareKeys(currentChild->key, key)) {

case 1:

currentParentPointer = &currentChild->left;

break;

case -1:

currentParentPointer = &currentChild->right;

break;

case 0:

return currentParentPointer;

break;

default:

printf("%s", "Compare error #11\n");

}

}

return NULL;

}

BNode\*\* getBNodeParentOfLefter(BNode\* root) {

BNode\*\* currentParentPointer = NULL;

BNode\* currentChild = NULL;

if ((NULL == root)

|| (NULL == (currentParentPointer = &root->left))

|| (NULL == (currentChild = \*currentParentPointer))) { return NULL; }

while (NULL != currentChild->left) {

currentParentPointer = &currentChild->left;

currentChild = \*currentParentPointer;

}

return currentParentPointer;

}

BNode\* deleteBNodeFromRoot(BNode\* root, Key\* key, void (\*deleteOp)(BNode\*), char\* status) {

if (NULL == root || NULL == key) { return root; }

BNode\* target = root;

BNode\*\* parentPointerTarget;

if (0 == compareKeys(target->key, key)) {

if ((NULL == target->left) && (NULL == target->right)) {

root = NULL;

} else if (NULL == target->left) {

root = target->right;

} else if (NULL == target->right) {

root = target->left;

} else {

BNode\* successor = NULL;

BNode\*\* successorParentPointer = getBNodeParentOfLefter(target->right);

if (NULL == successorParentPointer) {

successor = target->right;

successor->left = target->left;

root = successor;

} else {

successor = \*successorParentPointer;

\*successorParentPointer = successor->right;

successor->right = target->right;

successor->left = target->left;

root = successor;

}

}

} else {

parentPointerTarget = getBNodeParentAnFindTarget(root, key);

if (NULL == parentPointerTarget) {

printf("%s", "Not found\n");

\*status = B\_TREE\_NOT\_FOUND;

return root;

}

target = \*parentPointerTarget;

if ((NULL == target->left) && (NULL == target->right)) {

\*parentPointerTarget = NULL;

} else if (NULL == target->left) {

\*parentPointerTarget = target->right;

} else if (NULL == target->right) {

\*parentPointerTarget = target->left;

} else {

BNode\* successor = NULL;

BNode\*\* successorParentPointer = getBNodeParentOfLefter(target->right);

if (NULL == successorParentPointer) {

successor = target->right;

} else {

successor = \*successorParentPointer;

\*successorParentPointer = successor->right;

successor->right = target->right;

}

successor->left = target->left;

\*parentPointerTarget = successor;

}

}

\*status = B\_TREE\_REMOVE\_SUCCESS;

deleteOp(target);

return root;

}

**Верхний слой:**

BNode\* root;

char insertBTree(Key\* key) {

if (NULL == key)

{ return INSERT\_FAIL; }

Key\* copyKey = (Key\*) allocate(sizeof(Key));

BNode\* newNode = (BNode\*) allocate(sizeof(BNode));

if (NULL == copyKey || NULL == newNode)

{ return INSERT\_FAIL; }

copyKey->port = key->port;

copyKey->ip = key->ip;

newNode->key = copyKey;

if (NULL == root) {

root = newNode;

} else {

char result = insertBNodeInsideRoot(root, newNode);

if (TREE\_INSERT\_OK != result) {

printf("%s", "B tree insert error\n");

removeVar((char\*) copyKey);

removeVar((char\*) newNode);

return INSERT\_FAIL;

}

}

return INSERT\_SUCCESS;

}

char findBTree(Key\* key) {

if (NULL == key) { return FIND\_FAIL; }

BNode\* result = findBNodeInsideRoot(root, key);

return NULL == result ? FIND\_FAIL : FIND\_SUCCESS;

}

char alterBTree(Key\* source, Key\* target) {

if (NULL == source || NULL == root)

{ return ALTER\_FAIL; }

BNode\* srcNode;

void deleteOp(BNode \* toDelete) {

srcNode = toDelete;

}

char removeStatus;

root = deleteBNodeFromRoot(root, source, &deleteOp, &removeStatus);

if (NULL == target) {

removeVar((char\*) srcNode->key);

removeVar((char\*) srcNode);

return REMOVE\_SUCCESS;

}

srcNode->key->ip = target->ip;

srcNode->key->port = target->port;

srcNode->left = NULL;

srcNode->right = NULL;

char insertResult = insertBNodeInsideRoot(root, srcNode);

return TREE\_INSERT\_OK == insertResult ? ALTER\_SUCCESS : ALTER\_FAIL;

}

char initBTree(int size) {

root = NULL;

int variablesSize[2] = {sizeof(Key), sizeof(BNode)};

int variablesCount[2] = {size, size};

char subheapCount = 2;

char result = init(variablesSize, variablesCount, subheapCount);

return (INITIAL\_SUCCESS == result) ? HASH\_TABLE\_INIT\_SUCCESS :

HASH\_TABLE\_INIT\_FAIL;

}

**Пакет: анализатор сетевого шлюза**

Содержание компонентов:

1) Генератор ключей.

2) Ложные передатчики.

3) Обработчик сообщений.

4) Клиент-сервер

5) Анализатор сетевого шлюза.

1) Генератор ключей

Цель: Генерирование уникальных ключей

Как: Генерирование 4-ех и 2-ух байтовых переменных, проверка на уникальность массива.

Проверка: Пограничный проверки уникальных и не уникальных массивов.

Код:

char isUniq(Key\* key, Key\* keys, int length) {

while (0 < --length)

if (0 == compareKeys(key, &keys[length]))

{ return 1; }

return 0;

}

void generateKey(Key\* key) {

key->ip = generateInt();

key->port = generateShort();

}

void generateUniqKey(Key\* key, Key\* keys,

int length) {

Key bufferKey;

do {

generateKey(&bufferKey);

} while (1 == isUniq(&bufferKey, keys, length));

key->port = bufferKey.port;

key->ip = bufferKey.ip;

}

void generateUniqKeys(Key\* keys, int length) {

generateKey(&keys[0]);

int count = 0;

while (++count < length) {

generateUniqKey(&keys[count], keys, count);

}

}

2) Ложные передатчики

Цель: Эмулировать сообщения от передатчиков

Как: Инициализация уникальных передатчиков, упаковка в байтовый массив, на основнии ключа = передатчика.

Проверка: Проверка генерации на основании известнвх передатчиков. (методы setTransmitters - служебный)

Код:

Key staticTransmitters[TRANSMITTERS\_COUNT];

Key\* transmitters = staticTransmitters;

unsigned int size = TRANSMITTERS\_COUNT;

Key\* getTransmitters() { return transmitters; }

void setTransmitters(Key\* newTransmitters,

unsigned int newSize) {

transmitters = newTransmitters;

size = newSize;

}

/\* Сгенерировать сообщение (на клиенте)

\* 1) установить тип сообщения, как обычное (измерить время поиска)

\* 2) получить случайный передатчик

\* 3) установить данные передатчика в сообщение

\* 4) если входной параметр (итерация) 50-ая, то изменить данные передатчика

\* и записать измененые данные в сообщение

\* \*/

void generateMessage(char\* message, unsigned long value) {

message[0] = MESSAGE\_TYPE\_REGULAR;

Key\* currentKey = &transmitters[generateIntByWidth(size)];

keyToCharArray(currentKey, &message[1]);

if (0 == value % TRANSMITTER\_ALTER\_THRESHOLD\_VALUE) {

message[0] = MESSAGE\_TYPE\_ALTER;

generateUniqKey(currentKey, transmitters, size);

keyToCharArray(currentKey, &message[7]);

}

}

void initTransmitters() {

generateUniqKeys(transmitters, size);

}

3) Обработчик сообщений

Цель: Обработка сообщения полученного от “Ложные передатчики”

Как: Распаковка сообщение и передача результата с замером времени коллекции.

Проверка: Проверка выполнения всех ветвей выполнения.

Код:

Collection collection;

void initMessageHandlerCollection(Collection\* \_collection) {

collection = \*\_collection;

}

char getMessageStatus(char\* message) {

return \*message;

}

void getPrimaryKey(Key\* key, char\* message) {

charArrayToKey(key, &message[1]);

}

void getSecondKey(Key\* key, char\* message) {

charArrayToKey(key, &message[7]);

}

char handleMessage(char\* message) {

char collectionStatus;

unsigned long startTime;

unsigned long measureTime;

Key primaryKey;

Key secondKey;

getPrimaryKey(&primaryKey, message);

switch (getMessageStatus(message)) {

case MESSAGE\_TYPE\_REGULAR: {

startTime = getULongNano();

collectionStatus = collection.find(&primaryKey);

measureTime = calcOffsetULong(startTime);

if (FIND\_SUCCESS == collectionStatus) logPrimary(measureTime);

else kLog("Unsuccess find", &primaryKey);

}

break;

case MESSAGE\_TYPE\_ALTER: {

getSecondKey(&secondKey, message);

startTime = getULongNano();

collectionStatus = collection.alter(&primaryKey,

&secondKey);

measureTime = calcOffsetULong(startTime);

if (ALTER\_SUCCESS != collectionStatus)

dkLog("Unsuccess alter", &primaryKey, &secondKey);

else logSecondary(measureTime);

}

break;

default:

printf("%s", "Invalid message type\n");

return MESSAGE\_INVALID;

}

return MESSAGE\_VALID;

}

4) Клиент-сервер

Цель: Создание TCP подключения, отправка/примека сообщений.

Как: Многопоточная инициализация. Прослушка гнезда. Отправка и прием байтового массива.

Проверка: Отправка и принятия заранее известных данных.

Код:

void error(const char\* msg) {

perror(msg);

exit(1);

}

/////////SERVER//////////////

#define MESSAGE\_TYPE\_BYTE\_COUNT 1

int maxMessageSize = MESSAGE\_SIZE;

int serverSocket;

int serverConnectClientSocket;

char startServer(unsigned int port) {

maxMessageSize = MESSAGE\_TYPE\_BYTE\_COUNT + 2 \* sizeof(Key);

socklen\_t clilen;

struct sockaddr\_in serv\_addr, cli\_addr;

serverSocket = socket(AF\_INET, SOCK\_STREAM, 0);

if (serverSocket < 0)

{ error("ERROR opening socket"); }

int on = 1;

int status = setsockopt(serverSocket, SOL\_SOCKET,

SO\_REUSEADDR,

(const char\*) &on, sizeof(on));

if (-1 == status)

{ perror("setsockopt(...,SO\_REUSEADDR,...)"); }

bzero((char\*) &serv\_addr, sizeof(serv\_addr));

serv\_addr.sin\_family = AF\_INET;

serv\_addr.sin\_addr.s\_addr = INADDR\_ANY;

serv\_addr.sin\_port = htons(port);

if (bind(serverSocket, (struct sockaddr\*) &serv\_addr, sizeof(serv\_addr)) < 0)

{ error("ERROR on binding"); }

listen(serverSocket, 1);

clilen = sizeof(cli\_addr);

serverConnectClientSocket = accept(serverSocket, (struct sockaddr\*) &cli\_addr, &clilen);

if (serverConnectClientSocket < 0)

{ error("ERROR on accept"); }

return 0;

}

void readMessage(char\* message) {

bzero(message, maxMessageSize);

int readStatus = read(serverConnectClientSocket,

message, maxMessageSize);

if (readStatus < 0)

{ error("ERROR reading from socket"); }

}

void closeServer() {

close(serverConnectClientSocket);

close(serverSocket);

}

/////////CLIENT//////////////

#define LOCALHOST "localhost"

int clientSocket;

char startClient(unsigned int port) {

struct sockaddr\_in serv\_addr;

struct hostent\* server;

clientSocket = socket(AF\_INET, SOCK\_STREAM, 0);

if (clientSocket < 0)

{ error("ERROR opening socket"); }

server = gethostbyname(LOCALHOST);

if (server == NULL) {

fprintf(stderr, "ERROR, no such host\n");

exit(0);

}

bzero((char\*) &serv\_addr, sizeof(serv\_addr));

serv\_addr.sin\_family = AF\_INET;

bcopy((char\*)server->h\_addr,

(char\*)&serv\_addr.sin\_addr.s\_addr,

server->h\_length);

serv\_addr.sin\_port = htons(port);

if (connect(clientSocket,

(struct sockaddr\*) &serv\_addr,

sizeof(serv\_addr)) < 0)

{ error("ERROR connecting"); }

return 0;

}

void sendMessage(char\* message) {

int sendStatus = write(clientSocket, message,

maxMessageSize);

if (sendStatus < 0)

{ error("ERROR writing to socket"); }

}

void closeClient() {

close(clientSocket);

}

5) Анализатор сетевого шлюза.

Цель: Анализ и обработка/вывод результата работы коллекции и сетевого шлюза.

Как: Композиция компонентов выше.

Проверка: Запуск программы.

Код:

void initThread(pthread\_t\* thread, void\* (\*method)()) {

int err = pthread\_create(thread, NULL, method, NULL);

if (err != 0) { printf("%s[%s]\n", "Can't create thread: ", strerror(err)); }

}

void initClientServer(int port) {

void stClient() {

startClient(port);

}

void stServer() {

startServer(port);

}

pthread\_t threadA;

pthread\_t threadB;

initThread(&threadB, (void\*) &stServer);

initThread(&threadA, (void\*) &stClient);

pthread\_join(threadB, NULL);

pthread\_join(threadA, NULL);

printf("%s%d\n",

"Client and server start on port: ", port);

}

#define SENDER\_IS\_DONE 1

#define SENDER\_ISNT\_DONE -1

unsigned char isDone = SENDER\_ISNT\_DONE;

unsigned long receiveCount = 0;

unsigned long sendCount = 0;

void threadSender() {

isDone = SENDER\_ISNT\_DONE;

sendCount = 0;

char messageToSend[MESSAGE\_SIZE];

//Отправить MESSAGE\_COUNT\_TO\_SEND раз

while (sendCount < MESSAGE\_COUNT\_TO\_SEND) {

//Сформировать сообщение и отправить сообщение

generateMessage(messageToSend, sendCount + 1);

sendMessage(messageToSend);

sendCount++;

}

printf("sendCount: %lu\n", sendCount);

isDone = SENDER\_IS\_DONE;

}

void threadReceiver() {

char messageToReceive[MESSAGE\_SIZE];

receiveCount = 0;

//Пока клиент не закончил отправлять пакеты ИЛИ мы не все приняли, обрабатываем

while (receiveCount < sendCount || SENDER\_IS\_DONE != isDone) {

readMessage(messageToReceive);

handleMessage(messageToReceive);

receiveCount++;

}

printf("receiveCount: %lu\n", receiveCount);

}

void startAnalyzer() {

pthread\_t threadA;

pthread\_t threadB;

initThread(&threadB, (void\*) &threadReceiver);

initThread(&threadA, (void\*) &threadSender);

pthread\_join(threadA, NULL);

pthread\_join(threadB, NULL);

closeClient();

closeServer();

}

void startForCollection(char (\*find)(Key\* key), char (\*insert)(Key\* key), char (\*alter)(Key\* source,

Key\* target), char (\*initCollection)(int size), char\* collectionName) {

Collection collection = {find, insert, alter, initCollection};

if (HASH\_TABLE\_INIT\_FAIL == collection.initCollection(TRANSMITTERS\_COUNT)) { printf("%s", "Can't init collection\n"); exit(-1);}

printf("%s", "<<<<Reinit block>>>>\n");

reset();

initTransmitters();

Key\* transmitters = getTransmitters();

int index = 0;

for (; index < TRANSMITTERS\_COUNT; index++) {

if (INSERT\_FAIL == collection.insert(&transmitters[index])) { printf("%s", "\nInit insert error!\n"); }

}

initClientServer(2500);

initMessageHandlerCollection(&collection);

startAnalyzer();

show(collectionName);

}

void start() {

setFileLoggerOutput();

printf("%s", "\n<<<<<HASH TABLE>>>>>\n");

startForCollection(&findHashTable, &insertHashTable, &alterHashTable, &initHashTable, "Hash");

printf("%s", "\n<<<<<AVL TREE>>>>>\n");

startForCollection(&findAVLTree, &insertAVLTree, &alterAVLTree, &initAVLTree, "AVL");

printf("%s", "\n<<<<<B TREE>>>>>\n");

startForCollection(&findBTree, &insertBTree, &alterBTree, &initBTree, "BI");

}

Результат работы анализатора:

Флагах O3 – макс оптимизация компилятора.

#define TRANSMITTERS\_COUNT 10000

#define MESSAGE\_SIZE 13

#define MESSAGE\_COUNT\_TO\_SEND 300000

#define TRANSMITTER\_ALTER\_THRESHOLD\_VALUE 50

#define MESSAGE\_TYPE\_REGULAR 1

#define MESSAGE\_TYPE\_ALTER 2

|  |  |  |
| --- | --- | --- |
| (Мин/Сред/Макс) Количество | Поиск | Изменение |
| Флаг оптимизация уровено 0 | | |
| Хэш таблица | (64, 120, 11058) 294000 | (93, 160, 2261) 6000 |
| АВЛ дерево | (62, 407, 47041) 294000 | (1055, 2146, 15969) 6000 |
| Двоичное дерево | (71, 380, 38613) 294000 | (514, 1035, 8599) 6000 |
| Флаг оптимизация уровено 3 | | |
| Хэш таблица | (41, 89, 31927) 294000 | (54, 110, 1579) 6000 |
| АВЛ дерево | (44, 214, 62081) 294000 | (473, 844, 11904) 6000 |
| Двоичное дерево | (45, 246, 22959) 294000 | (398, 845, 11869) 6000 |

Потребление памяти

Хэш таблица:

TRANSMITTERS\_COUNT \* sizeof(Key) = 60000

TRANSMITTERS\_COUNT \* sizeof(LinkedListNode) = 80000

TRANSMITTERS\_COUNT \* (размер указателя) = 80000

Итог: 220000 байт

АВЛ дерево:

TRANSMITTERS\_COUNT \* sizeof(Key) = 60000

TRANSMITTERS\_COUNT \* sizeof(AVLNode) = 130000

Итог: 190000 байт

АВЛ дерево:

TRANSMITTERS\_COUNT \* sizeof(Key) = 60000

TRANSMITTERS\_COUNT \* sizeof(BNode) = 120000

Итог: 180000 байт