Recomendações de teste da 2ª parte do Projecto

O pacote de testes utilizado no Projecto gera sequências de pares <key, value>, a partir dos Clientes, que permitem testar as quatro funções de acesso ao KOS – `get, put, remove, getAllKeys` – em diversas situações. Os testes são feitos por comparação dos pares lidos face ao padrão de pares previamente escritos/removidos.

Sendo testes de dados, no entanto também permitem exercitar a sincronização entre Clientes e Servidores, através dos Buffers, e exercitar a sincronização no acesso dos Servidores ao KOS, pressupondo que, se houver alguma falha nestes passos da execução ela se reflectirá na estabilidade das tarefas ou na consistência dos dados.

Os testes foram disponibilizados por forma a que possam ser ajustados pelos grupos no desenvolvimento dos seus projectos para testar outros padrões de acesso particulares.

Seguem-se recomendações de parametrização ou configuração de testes para tratar especificamente requisitos da 2ª parte do Projecto:

- Sincronização Clientes-Servidores através dos buffers
- Concorrência no acesso ao KOS
- Persistência

Sincronização Clientes-Servidores

Os testes assumem, de acordo com os requisitos da 1ª parte, números de clientes iguais a números de servidores (`test_kos_multithreaded.c`):

```
#define NUM_EL 600
#define NUM_SHARDS 5
#define NUM_CLIENT_THREADS 30
#define NUM_SERVER_THREADS 30
#define BUF_SIZE 30
```

Esta limitação terá conduzido, na 1ª parte, a soluções simplificadas para a comunicação entre cada Cliente e o seu Servidor.

Na 2ª parte pretende-se implementar um mecanismo mais flexível pelo que se sugere testar situações com Nº Clientes ≠ Nº Servidores, e também com Nº buffers ≠ Nº Servidores ≠ Nº Clientes. Por exemplo:

Nº Clientes > Nº Servidores, Nº Clientes > Nº buffers > Nº Servidores

```
#define NUM_CLIENT_THREADS 30
#define NUM_SERVER_THREADS 10
#define BUF_SIZE 15
```

Nº Clientes > Nº Servidores, Nº Clientes > Nº Servidores > Nº buffers

```
#define NUM_CLIENT_THREADS 30
#define NUM_SERVER_THREADS 10
#define BUF_SIZE 5
```

Teste com estes valores ou com outros valores razoáveis mas que mantenham as proporções relativas.

Em particular, para testar em carga a gestão da alocação do buffer considere também `BUF_SIZE 1`.

Concorrência no acesso ao KOS

A 2ª parte introduz o requisito dos acessos concorrentes entre listas e em leituras em cada lista da partição (leitores-escritores na lista).

Tal como na 1ª parte, deficiências na sincronização tenderão a reflectir-se na estabilidade das tarefas e na consistência dos dados.

Pode ser difícil evidenciar o grau de concorrência obtido na implementação da 2ª parte face à 1ª parte (eventualmente com sincronização – mutex – global ao KOS ou por partição). Pode até suceder que o desempenho, estimado pelo tempo de execução dos testes, agora aumente, devido à sobrecarga com a implementação dos novos modelos de sincronização. (Multitarefa *CPU bond* num monoprocessador.)

Sugere-se a introdução de um pequeno atraso no código dos servidores – `sleep ()` - que agrave artificialmente o seu tempo de execução mas não altere a sua funcionalidade e liberte o controlo do processador. O atraso deverá ser inserido na secção de código imediatamente após a obtenção do *lock* e antes de inciar o processamento na lista da partição.

Se pretender tomar como referência de comparação a 1ª parte do Projecto idêntica alteração deve ser introduzida na sua implementação.

Deve-se ter em conta que o tempo de execução dos testes aumenta significativamente com a introduição do `sleep ()` pelo que será então recomendável ajustar o número de elementos gerados no teste.

Persistência

A 2ª parte introduz o requisito de armazenamento em ficheiro do estado de cada partição. No entanto os Clientes continuam a ser "alimentados" a partir dos dados armazenados na partição em memória, isto é, as quatro funções – get, put, remove, getAllKeys – mantêm a funcionalidade da 1ª parte continuando a devolver aos Clientes dados resultantes do estado da partição.

Para verificar que os dados escritos nos ficheiros correspondem efectivamente ao conteúdo da partição em memória sugere-se realizar o teste em anexo . Este teste logo no início faz uma leitura e verifica se a partição já tem conteúdo, se não, executa os testes já incluídos em `test_kos_multi_threaded_all_getAll`, como na 1ª parte, se sim, quer dizer que, como ainda não houve escritas, a partição foi carregada a partir de um ficheiro e chama logo a função getAllKeys () que faz o *dump* da partição para o Cliente.

Sugere-se a realização dos seguintes passos:

1) Apagar ficheiros fShard eventualmente criados em testes anteriores;

2) Correr o teste – Como não há ficheiros o teste verifica os acessos ao KOS como na 1ª parte do projecto.

3) Verificar que foram criados ficheiros `fShard`;

4) Correr o teste – Agora já há ficheiros, o teste só executa o `getAllKeys ()`.

Compare também a dimensão dos ficheiros `fShard` gerados com o número de pares <key, value> inseridos e removidos no teste – proporcional a `NUM_EL` – para avaliar o grau de compactação dos ficheiros:

```
NUM_EL put() + NUM_EL/2 remove() + NUM_EL put()
```

Anexo - test_kos_multi_threaded_all_getAll_fromFile

```
#include <kos_client.h>
#include <pthread.h>
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define NUM_EL 1000
#define NUM_SHARDS 10
#define NUM_CLIENT_THREADS 10
#define NUM_SERVER_THREADS 10

#define KEY_SIZE 20

// #define DEBUG_PRINT_ENABLED 1  // uncomment to enable DEBUG statements
#if DEBUG_PRINT_ENABLED
#define DEBUG printf
#else
#define DEBUG(format, args...) ((void)0)
#endif


int lookup(char* key, char* value, KV_t* dump,int dim) {
      int i=0;
      for (;i<dim;i++) {
            if ( (strncmp(key,dump[i].key,KEY_SIZE)) &&
(strncmp(value,dump[i].value,KEY_SIZE) ) )
             return 0;
      }
      return -1;
}


void *client_thread(void *arg) {

      char key[KEY_SIZE], value[KEY_SIZE], value2[KEY_SIZE];
      char* v;
      int i,dim;
      int client_id=*( (int*)arg);
      KV_t* dump;


      // Check if shard is empty or already written. Check the first <key,value>
      usually inserted //
      i=NUM_EL-1;
      sprintf(key, "k-c%d-%d",client_id,i);
      sprintf(value, "val:%d",i);
      v=kos_get(client_id, client_id, key, value);

      if (v == NULL) {
            printf("Shard seems to be empty - Run all tests 1 to 6");
            // Otherwise goes directly to test 7 - the final getAllKeys //



            for (i=NUM_EL-1; i>=0; i--) {
                  sprintf(key, "k-c%d-%d",client_id,i);
                  sprintf(value, "val:%d",i);
                  v=kos_put(client_id, client_id, key,value);
```

```
                  DEBUG("C:%d  <%s,%s> inserted in shard %d. Prev Value=%s\n",
client_id, key, value, j, ( v==NULL ? "<missing>" : v ) );
            }

            printf("-------------------- %d:1/7 ENDED INSERTING
----------------------\n",client_id);

            for (i=0; i<NUM_EL; i++) {
                  sprintf(key, "k-c%d-%d",client_id,i);
                  sprintf(value, "val:%d",i);
                  v=kos_get(client_id, client_id, key);
                  if (strncmp(v,value,KEY_SIZE)!=0) {
                        printf("Error on key %s value should be %s and was
returned %s",key,value,v);
                        exit(1);
                  }
                  DEBUG("C:%d  %s %s found in shard %d: value=%s\n", client_id,
key, ( v==NULL ? "has not been" : "has been" ),client_id,
                                                ( v==NULL ? "<missing>" : v ) );
            }


            printf("------------------- %d:2/7 ENDED READING
---------------------\n",client_id);


            dump=kos_getAllKeys(client_id, client_id, &dim);
            if (dim!=NUM_EL) {
                  printf("TEST FAILED - SHOULD RETURN %d ELEMS AND HAS RETURNED
%d",NUM_EL,dim);
                  exit(-1);
            }

            for (i=0; i<NUM_EL; i++) {
                  sprintf(key, "k%d",i);
                  sprintf(value, "val:%d",i);
                  if (lookup(key,value,dump, dim)!=0) {
                        printf("TEST FAILED - Error on <%s,%s>, shard %d - not
returned in dump\n",key,value,client_id);
                        exit(-1);
                  }

            }


            printf("------------------ %d-3/7 ENDED GET ALL KEYS
------------------------\n",client_id);


            for (i=NUM_EL-1; i>=NUM_EL/2; i--) {
                  sprintf(key, "k-c%d-%d",client_id,i);
                  sprintf(value, "val:%d",i);
                  v=kos_remove(client_id, client_id, key);
                  if (strncmp(v,value,KEY_SIZE)!=0) {
                        printf("Error when removing key %s value should be %s
and was returned %s",key,value,v);
                        exit(1);
                  }
                  DEBUG("C:%d  %s %s removed from shard %d. value =%s\n",
client_id, key, ( v==NULL ? "has not been" : "has been" ),client_id,
                                                ( v==NULL ? "<missing>" : v ) );
```

```
            }


            printf("----------------- %d-4/7 ENDED REMOVING
------------------------\n",client_id);


            for (i=0; i<NUM_EL; i++) {
                  sprintf(key, "k-c%d-%d",client_id,i);
                  sprintf(value, "val:%d",i);
                  v=kos_get(client_id, client_id, key);
                  if (i>=NUM_EL/2 && v!=NULL) {
                        printf("Error when gettin key %s value should be NULL
and was returned %s",key,v);
                        exit(1);
                  }
                  if (i<NUM_EL/2 && strncmp(v,value,KEY_SIZE)!=0 ) {
                        printf("Error on key %s value should be %s and was
returned %s",key,value,v);
                        exit(1);
                  }
                  DEBUG("C:%d  %s %s found in shard %d. value=%s\n", client_id,
key, ( v==NULL ? "has not been" : "has been" ) ,client_id, ( v==NULL ?
"<missing>" : v ) );
      }

            printf("----------------- %d-5/7 ENDED CHECKING AFTER REMOVE
-----------------\n",client_id);


                  for (i=0; i<NUM_EL; i++) {
                        sprintf(key, "k-c%d-%d",client_id,i);
                        sprintf(value, "val:%d",i*10);
                        sprintf(value2, "val:%d",i*1);
                        v=kos_put(client_id, client_id, key,value);

                        if (i>=NUM_EL/2 && v!=NULL) {
                              printf("Error when getting key %s value should be
NULL and was returned %s",key,v);
                              exit(1);
                        }
                        if (i<NUM_EL/2 && strncmp(v,value2,KEY_SIZE)!=0 ) {
                              printf("Error on key %s value should be %s and was
returned %s",key,value2,v);
                              exit(1);
                        }


                        DEBUG("C:%d  <%s,%s> inserted in shard %d. Prev Value=
%s\n", client_id, key, value, clent_id, ( v==NULL ? "<missing>" : v ) );
                  }


            printf("----------------- %d-6/7 ENDED 2nd PUT WAVE
----------------\n",client_id);

      }

            dump=kos_getAllKeys(client_id, client_id, &dim);
            if (dim!=NUM_EL) {
```

```
                  printf("TEST FAILED - SHOULD RETURN %d ELEMS AND HAS RETURNED
%d",NUM_EL,dim);
                  exit(-1);
            }

            for (i=0; i<NUM_EL; i++) {
                  sprintf(key, "k%d",i);
                  sprintf(value, "val:%d",i*10);
                  if (lookup(key,value,dump, dim)!=0) {
                        printf("TEST FAILED - Error on <%s,%s>, shard %d - not
returned in dump\n",key,value,client_id);
                        exit(-1);
                  }

            }



      printf("----------------- %d-7/7 ENDED FINAL ALL KEYS CHECK
----------------------\n",client_id);

      return NULL;
}


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

      int i,s,ret;
      int* res;
      pthread_t*
threads=(pthread_t*)malloc(sizeof(pthread_t)*NUM_CLIENT_THREADS);
      int* ids=(int*) malloc(sizeof(int)*NUM_CLIENT_THREADS);

      // this test uses NUM_CLIENT_THREADS shards, as each client executes
commands in its own shard
      ret=kos_init(NUM_CLIENT_THREADS,NUM_SERVER_THREADS,NUM_CLIENT_THREADS);

      //printf("KoS inited");

      if (ret!=0)  {
                  printf("kos_init failed with code %d!\n",ret);
                  return -1;
            }

      for (i=0; i<NUM_CLIENT_THREADS; i++) {
            ids[i]=i;

            if ( (s=pthread_create(&threads[i], NULL, &client_thread, &(ids[i]))
) ) {
                  printf("pthread_create failed with code %d!\n",s);
                  return -1;
            }
      }

      for (i=0; i<NUM_CLIENT_THREADS; i++) {
                s = pthread_join(threads[i], (void**) &res);
                if (s != 0) {
                    printf("pthread_join failed with code %d",s);
                return -1;
            }
           }
```

```
	printf("\n--> TEST PASSED <--\n");

	return 0;
}
```