INSTITUTO SUPERIOR TÉCNICO
Departamento de Engenharia Informática
Enunciado do Projeto de Sistemas Operativos – Parte I
LEIC/LETI 2013-2014 – 1º Semestre

1. KOS: Key value store of the OS course

O objectivo deste projeto é desenvolver um sistema para o armazenamento de dados chamado KOS (*Key-value store of the Operating Systems course*). O KOS utiliza um modelo lógico de organização de dados chamado chave-valor (*key-value*), em que cada dado é associado com o par seguinte: uma "chave" que permite identificar univocamente o dado, e um "valor". Este tipo de sistemas de armazenamento de dados são usados em vários contextos, como por exemplo no *registry* do Windows para guardar informações de configuração e meta-dados do sistema operativo.

A organização lógica do KOS é ilustrada na Figura 1. Os dados guardados no KOS são organizados em partições ("shards"), identificadas por um identificador numérico inteiro (shardId). O "sharding" é uma técnica usada para optimizar o desempenho das primitivas de acesso, porque permite paralelizar o acesso aos dados ao subdividi-los em partições (shards) e atribuir partições a servidores dedicados.

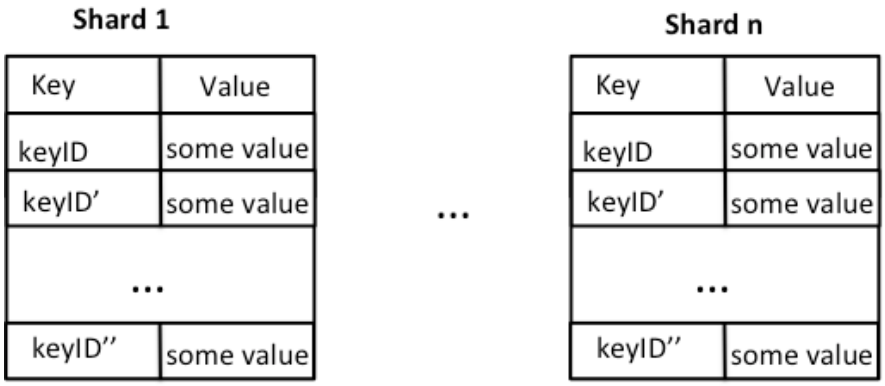

Figura 1 Modelo conceptual de organização de dados no KOS

O KSO suporta a seguinte API:

- char* get(int clientId, int shardId, char* key): retorna o "Value" associado à chave ("Key") passada como parâmetro de entrada, caso esta exista no KOS, NULL em caso contrário.
- char*put(int clientId, int shardId, char* key, char* value): insere um par "<Key,Value>" e devolve NULL se a chave "Key" não existe, ou, no caso contrário, o valor anterior associado a essa chave.
- char* remove(int clientId, int shardId, char* key): apaga a chave identificada pela string Key, devolvendo o valor da chave, caso esta existia, e NULL em caso contrário.

- KV_t* kos_getAllKeys(int clientId, int shardId, int* sizeKVarray): devolve um apontador para um vector de pares "<Key,Value>" (sendo cada par guardado numa estrutura de dados do tipo KV_t, cuja definição é especificada no Anexo A). O tamanho do vector devolvido é escrito no parâmetro sizeKVarray.

Por simplicidade, assume-se que tanto a chave como o valor são "strings" de tamanho fixo (definido pela constante KV_SIZE).

Internamente, cada partição do KOS é suportada por uma tabela de dispersão (*hashmap)*, que terá que ser concretizado pelos alunos. Mais indicações acerca da concretização da tabela de dispersão serão fornecidas na Secção 3.

O KOS executa-se num único processo constituído por múltiplas tarefas. Entre as tarefas distinguem-se tarefas *servidoras* e *clientes.*
As tarefas servidoras gerem o estado do KOS e servem pedidos gerados por tarefas cliente (Figura 2).

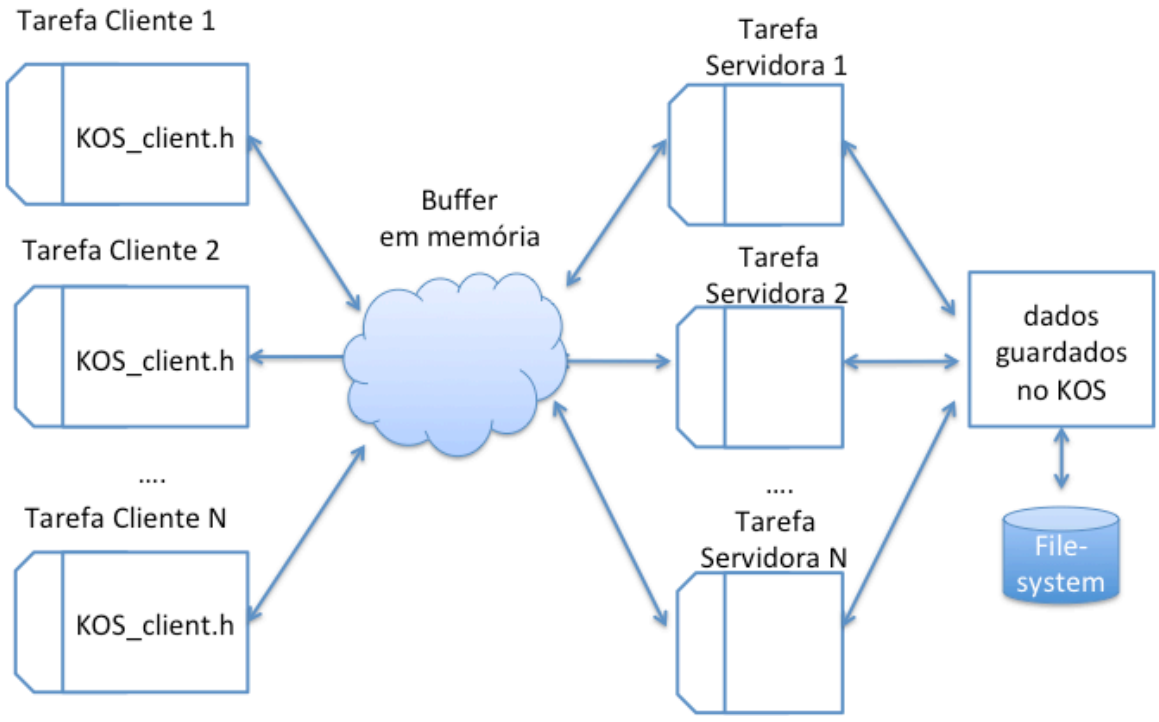

Figura 2 Arquitetura geral da solução proposta

As tarefas clientes não acedem diretamente ao estado do KOS. Em vez disso, estas tarefas devem interagir com as tarefas servidoras usando as funções especificadas no ficheiro `kos_client.h` (este ficheiro é incluído no pacote software e encontra-se também no Anexo A), as quais inserem um pedido num buffer em memória partilhado com as tarefas servidoras. Este buffer partilhado suporta a interação entre estes dois conjuntos de tarefas. As tarefas clientes colocam no buffer antes referido os seus pedidos que, por sua vez, as tarefas

servidoras executam; este mesmo buffer deve ser usado para retornar as respostas respectivas às tarefas clientes.

2. Estrutura e organização do projeto

O projeto é estruturado em duas partes:

- a primeira parte do projeto tem como alvo o desenvolvimento da arquitetura base do KOS. Em particular, durante a primeira fase será preciso:
  - conceber e concretizar um esquema básico de sincronização entre tarefas clientes e servidoras. Nesta fase, assume-se que o número de tarefas clientes e servidoras é o mesmo e é conhecido à partida, permitindo soluções em que cada tarefa cliente seja estaticamente associada a uma dada tarefa servidora.
  - conceber e concretizar um esquema básico de sincronização entre tarefas servidoras, em que o acesso ao KOS é protegido por um único trinco (ou semáforo).
  - concretizar a tabela de dispersão usada para guardar os dados do KOS em memória volátil (RAM), ou seja nesta fase **não será necessário** desenvolver mecanismos para persistir o estado do KOS em disco.
- a segunda parte do projeto será focada em dois componentes:
  - tornar os esquemas de sincronização entre tarefas mais eficiente, perseguindo o máximo nível de paralelismo possível (quer no acesso ao buffer quer no acesso aos dados persistentes do KOS)
  - desenvolver esquemas para garantir a persistência dos dados guardados no KOS através das API de sistema de ficheiros UNIX

3. Desenho e concretização da solução da primeira parte do projeto

Como mencionado anteriormente, na primeira parte do projeto será necessário concretizar a arquitectura base do projeto, sem incluir ainda os mecanismos para a persistência e implementando soluções de sincronização básicas. A seguir são fornecidas informações adicionais sobre as estruturas de dados usadas para guardar o estado do KOS em memória volátil, e sobre os mecanismos de sincronização entre tarefas.

a. Tabelas de dispersão usadas para guardar os dados do KOS

A Figura 3 mostra o esquema da tabela de dispersão usada para guardar (em memória volátil) cada partição de dados do KOS. Cada tabela é concretizada através de um vector de listas ligadas. O vector tem um número de elementos fixo (especificado através da constante HT_SIZE). Um par <chave, valor> que pertence à partição associada à tabela de dispersão é

inserido na lista na posição i do vector, onde i é determinado através de uma função dispersão (cujo código é fornecido no Anexo B). Esta função converte cada carácter da chave num inteiro, e soma cada um destes valores numa variável acumuladora, chamada *i*. A posição do vector de listas para aceder é determinada como o resultado de: i % HT_SIZE.

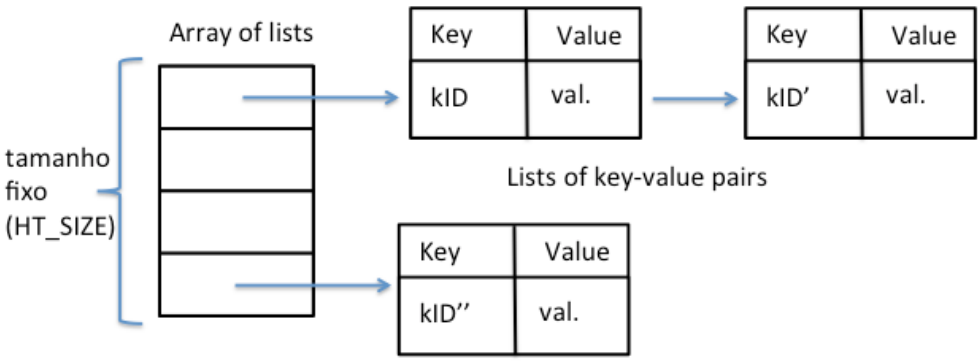

Figura 3. Tabela de dispersão usada para guardar cada partição de dados.

b. Buffer para a comunicação entre tarefas clientes e servidoras

Para a comunicação entre tarefas clientes e servidoras só serão consideradas válidas soluções em que as tarefas comuniquem indirectamente por um *buffer* partilhado em memória. **Soluções que utilizem outras alternativas para a comunicação entre tarefas (por exemplo ficheiros, *sockets*, etc.) não serão avaliados.**

Nesta primeira fase, será assumido que há o mesmo número de tarefas clientes e servidoras, e que este valor é conhecido à partida especificado pela constante NUM_THREADS.

O buffer em memória deverá ser materializado através de um vector de tamanho NUM_THREADS, onde cada elemento do vector é estaticamente associado a um par <tarefa cliente i, tarefa servidora i>. Isto quer dizer que os pedidos duma dada tarefa cliente sempre serão processados pela mesma tarefa servidora. Cada elemento do vector deverá conter informações suficientes para permitir a sincronização entre as tarefas cliente e servidora correspondente.

O diagrama em Figura 4 ilustra a solução descrita em cima. Para sincronizar as atividades duma tarefa cliente e da correspondente tarefa servidora, a solução a desenvolver deverá maximizar a eficiência da sincronização entre cada par de tarefas cliente/servidora. As tarefas servidoras deverão ficar bloqueadas se não houver pedidos para processar, e as tarefas clientes deverão bloquear-se em espera da resposta a os próprios pedidos; portanto, nenhuma tarefa deverá ficar em espera ativa.

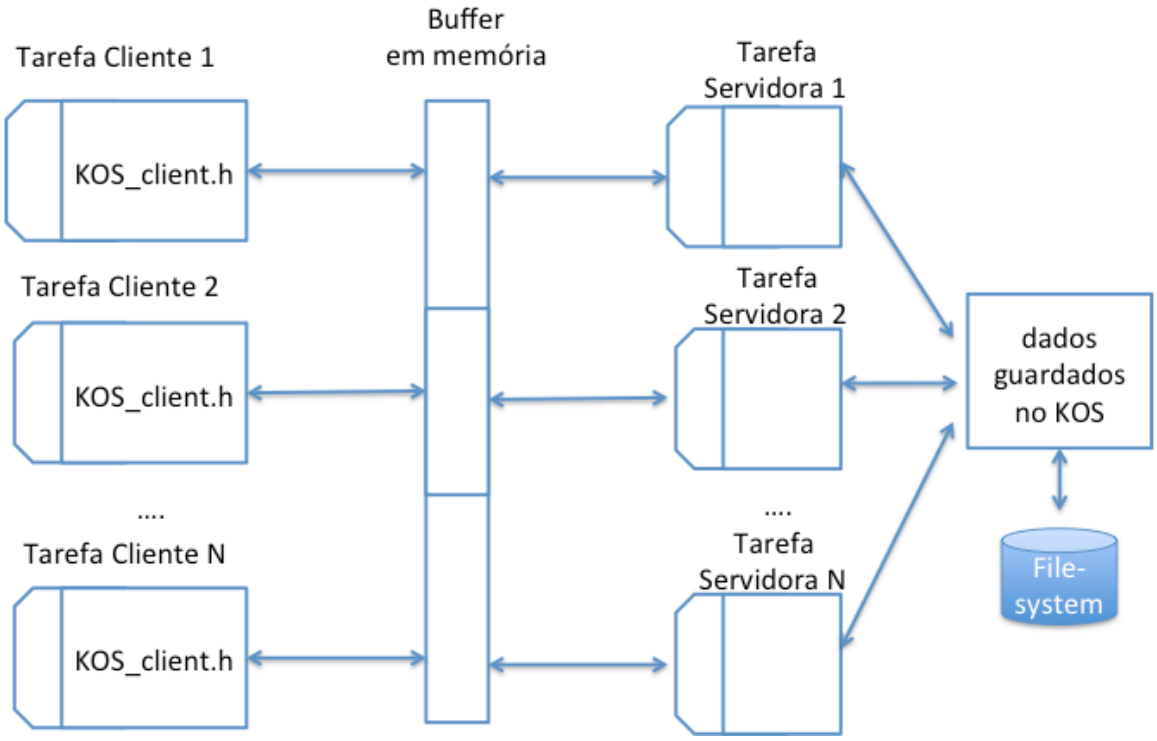

Figura 4. Diagrama que ilustra a organização do buffer em memória usado pela sincronização entre tarefas clientes e servidoras pretendido na primeira parte do projeto.

c. Sincronização entre tarefas servidoras para o acesso aos dados guardados no KOS

Nesta primeira fase de projeto será suficiente desenvolver soluções simples (por exemplo baseadas no uso de um único trinco/semáforo) para sincronizar o acesso das tarefas servidoras aos dados guardados no KOS. Será alvo da segunda parte do projeto melhorar o desempenho do sistema integrando esquemas de sincronização que permitam atingir níveis de paralelismo mais elevados.

Antes de começar a processar qualquer pedido do buffer, cada tarefa servidora deverá executar uma chamada à função delay(), fornecida no pacote do projeto e definida pelos ficheiros: include/delay.h e kos/delay.c. Esta função injeta um atraso (para simular, por exemplo, o indeterminismo do sistema operativo), suspendendo a execução da tarefa por um período de tempo (configurável no ficheiro delay.c) e permitindo exercitar os mecanismos de sincronização entre as tarefas servidoras.

4. Secção 4 - Calendário proposto, entrega e avaliação

a. Calendário proposto

- 3/10 a 10/10 (1 semana e meia):
  - Estudar o problema e desenvolver solução em pseudo-código;
- 11/10 a 18/10 (1 semana)
  - Desenvolver e testar o código da tabela de dispersão apenas com uma tarefa servidora;
- 19/10 a 3/11 (2 semanas)
  - Concretizar o esquema de sincronização entre tarefas clientes e servidoras, e entre tarefas servidoras para o acesso aos dados guardados no KOS
  - Desenvolver testes adicionais para verificar a correção da solução.

b. Entrega da Parte I (checkpoint): 3 de Novembro às 23:59

Os alunos devem entregar uma concretização completa da Parte I do projeto por via electrónica através do sistema Fénix. O formato do ficheiro de entrega com o código será indicado oportunamente. Posteriormente, os alunos podem ainda alterar o seu projeto pois, como se indica de seguida, ambas as fases são avaliadas apenas no final do semestre. No entanto, sugere-se veementemente que, para os alunos não se atrasarem, sigam as indicações do calendário proposto.

c. Avaliação

A avaliação da Parte I será feita numa discussão no fim do semestre (onde também se avaliará a Parte II do projeto) na qual terão que estar presentes todos os elementos do grupo e onde será atribuída uma nota individual a cada elemento. Na atribuição dessa nota, para além da qualidade do programa apresentado (Parte I e Parte II), serão levados em conta factores como o desempenho individual na discussão do projeto, a participação nas aulas de laboratório e o acompanhamento do progresso do projeto feito pelos docentes das aulas práticas.

O peso da Parte I vale 40% da nota final do projeto, enquanto que a Parte II contará 60%.

A informação sobre a entrega e avaliação aqui descrita está sujeita a alterações que, caso ocorram, serão afixadas na página da cadeira.

Anexo A – Ficheiro *kos_client.h*

```
#ifndef KOS_H
#define KOS_H 1

/* The keys and values of KOS are strings of fixed size, specified by the
KV_SIZE parameter */
#define KV_SIZE 20


/* Struct used to return the current state of an entire shard by
kos_getAllKeys function. */
typedef struct KV_t {
    char key[KV_SIZE];
    char value[KV_SIZE];
} KV_t;


/* Functions exposed by the KOS system to the Clients */

/* Used to initialize the server side of the application. It takes as
input parameter the number of threads to be concurrently activated on the
server side,  the size of the internal buffer used to enqueue user client
requests, and the number of existing shards.

This function returns:
   *   0, if the initialization was successful;
   *  -1 if the initialization failed;   */
int kos_init(int num_server_threads, int buf_size, int num_shards);


/*NOTE:
 All the operations specified below takes, among others, the following
parameters:
- the clientId: which is meant to be used to allow mapping the client to
a specific server thread (needed by the first part of the project).
- the shardId: which identifies the shard in which the key should be
removed.

In case an invalid clientId/shardId is passed as input parameter, all
functions should return NULL.

- The keys maintained by KOS must be not NULL, but the values (of
key/value pairs) can be NULL.

*/

/* Returns NULL if key is not present. Otherwise, it returns the value
associated with the key passed as input.
*/

char* kos_get(int clientId, int shardId, char* key);
```

```
/* Inserts the <key, value> pair passed as input parameter in KOS,
returning: NULL, if the key was not previously present in KOS; the
previous value associated with the key, otherwise.
*/
char* kos_put(int clientId, int shardId, char* key, char* value);

/* Removes the <key, value> pair passed as input parameter in KOS,
returning: NULL, if the key was not present in KOS; the value associated
with the key whose removal was requested, if the key was  previously
present in KOS.
*/

char* kos_remove(int clientId, int shardId, char* key);


/* returns an array of KV_t containing all the key value pairs stored in
the specified shard of KOS (NULL in case the shard is empty). It stores
in sizeKVarray the number of key/value pairs present in the specified
shard. If the clientId or shardId are invalid (e.g. too large with
respect to the value specified upon inizialitization of KOS), this
function assigns -1 to the parameter sizeKVarray.   */

KV_t* kos_getAllKeys(int clientId, int shardId, int* sizeKVArray);


#endif
```

Anexo B – Código da função de dispersão para implementar a hashmap

A função em baixo recebe como argumento um ponteiro para o identificador duma chave, e devolve o índice do array de listas (usado para implementar a hashmap) onde esta chave deve ser guardada.

Esta função encontra-se também no ficheiro *kos/hash.c* no pacote do projeto.

```
int hash(char* key) {

     int i=0;

     if (key == NULL)
      return -1;

     while (*key != '\0') {
      i+=(int) *key;
      key++;
     }

     i=i % HT_SIZE;

     return i;
}
```