INSTITUTO SUPERIOR TÉCNICO

Departamento de Engenharia Informática

Enunciado de Projecto de Sistemas Operativos

LEIC/LERC
2010/2011

Parte I – Biblioteca sthreads

Objectivo

A biblioteca *sthreads* é uma biblioteca de pseudo-tarefas. É fornecida aos alunos uma versão desta biblioteca que suporta a criação de tarefas e o seu escalonamento, usando uma política de tempo partilhado e uma lista circular (*round-robin*). Pretende-se estender a biblioteca por forma a substituir a política de escalonamento fornecida por uma política de escalonamento semelhante à da versão 2.6.23 do núcleo do Linux.

Contexto

A partir da versão 2.6.23 do kernel o sistema operativo Linux passou a utilizar um algoritmo de escalonamento denominado *Completely Fair Schedule* (CFS). No CFS cada processo está ordenado pelo tempo de CPU que lhe foi atribuído (*vruntime*). Os processos com menos tempo de CPU são os primeiros a serem escalonados para o CPU. Quando um processo é criado é lhe atribuído um tempo de CPU fictício igual ao menor dos tempos de execução dos processos ainda em funcionamento. Deste modo, os processos recém-criados têm de imediato oportunidade de se executar, pois têm menos tempo de execução que todos os outros, no entanto, não têm a prioridade que teriam se lhes fosse atribuído o tempo de execução real (o tempo de execução de um processo recém criado é zero). Se tal acontecesse os processos de longa duração seriam sempre preteridos relativamente aos de curta duração. De cada vez que o escalonador se executa o *vruntime* do processo em execução é incrementado do valor em que

esteve em execução, e caso o novo valor de *vruntime* seja maior que o menor valor de *vruntime* dos outros processos, o processo é retirado de execução.

A novidade deste algoritmo é que só é relevante, para o escalonamento, o tempo que um processo está em execução; o tempo que ele esteve em espera bloqueado num recurso ou que esteve em espera na fila do escalonador são indistinguíveis. Como os processos interactivos vão estar frequentemente bloqueados em recursos o seu tempo de execução vai ser baixo pelo que, quando saírem do bloqueio, irão naturalmente ter prioridade sobre os outros.

O CFS usa apenas uma estrutura ordenada de processos por escalonar, mas de modo a evitar o problema de escalabilidade da gestão das filas da versão 2.4, o CFS usa uma árvore *red-black* para manter todos os processos ordenados por tempo de execução, garantindo que as operações de inserção e remoção de processos se conseguem efectuar em O(log(n)).

No CFS os processos mais prioritários são beneficiados na medida em que o tempo de execução adicionado à variável *vruntime* em cada execução do despacho é apenas uma fracção do tempo realmente executado. Por outro lado aos processos marcados com *nice* é adicionado a seu *vruntime* um valor superior ao valor de execução real (mais detalhes na secção seguinte).

O CFS não possui prioridades de "tempo real", i.e. processos que são sempre escalonados prioritariamente independentemente do seu tempo de execução. Os processos com prioridades deste tipo são geridos por outro escalonador. De facto, a partir da versão 2.6.23 o sistema operativo Linux passou a possuir um meta-escalonador que gere uma lista de escalonadores e que escolhe qual deles irá escolher o processo a executar. A política deste meta-escalonador é muito simples: os vários escalonadores são mantidos numa lista ordenada por prioridade, e os processos dos escalonadores de menor prioridade só são escolhidos se não existirem processos activos nos escalonadores de maior prioridade. Como o escalonador dos processos de "tempo real", possui uma prioridade superior ao CFS, o primeiro tem sempre prioridade sobre o segundo. Por seu lado o escalonador de "tempo real" é também muito simples escolhendo sempre os processos mais prioritários independentemente do seu tempo de execução. O meta-escalonador não é para ser implementado neste trabalho.

Implementação do Escalonador

Uma das primeiras tarefas a executar é criar uma RB-tree para colocar as tarefas activas, i.e. que podem ser executadas. A chave desta RB-Tree deve ser a variável *vruntime*, ou seja o tempo de execução da thread em causa.

Note que com esta solução a thread a escalonar é sempre a thread no nó mais à esquerda da árvore, pelo que do ponto de vista da eficiência, é importante manter uma variável a apontar para esse nó.

Um aspecto importante é o facto de a variável *vruntime* crescer indefinidamente, o que pode provocar *overflows*. Tenha este aspecto em consideração.

A cada *tick* de relógio é necessário adicionar ao *vruntime* da thread activa o valor gasto desde a última vez e verificar se é necessário que se seja *preempted*. Para evitar que uma thread fique muito pouco tempo em execução deve existir uma variável de configuração que estabelece o menor tempo possível que uma thread

tem que estar activa (*min_delay*). De notar que este conceito é diferente do conceito de *quantum*. Antes de *min_delay* uma tarefa não pode ser *preempted* por outra tarefa gerida pelo CFS (pode, no entanto, ser *preempted* por tarefas em escalonadores de maior prioridade, que neste trabalho não são implementados). Sugere-se que o min_delay seja igual a 5 *ticks*.

O valor adicionado a cada *tick* de relógio depende da prioridade da thread. De facto, o valor do tempo a adicionar deve ser multiplicado por prioridade da thread + nice: *vruntime* = *vruntime* + *tick*[1]*s**(prioridade+nice). Deste modo as tarefas de prioridade mais baixa têm muito mais tempo que as restantes. Sendo que o valor da prioridade varia entre 1 e 10 e o valor de nice entre 0 e 10.

Concretização

A biblioteca disponibilizada suporta comutação provocada explicitamente através da invocação da rotina sthread_yield() e efectua a comutação das tarefas de cada vez que ocorre um signal periódico que simula a interrupção de relógio. O material fornecido para esta parte do trabalho ajuda a compreender como gerar e tratar *signals*. Na biblioteca disponibilizada encontra-se ainda uma implementação de monitores.

Na concretização da implementação do escalonamento proposto terá que ter em conta os seguintes pontos:

- A função de criação de tarefas (sthread_create) terá que passar a receber o valor de prioridade inicial para além dos argumentos que já recebia. O protótipo da função passará a ser:
  sthread_t sthread_create(sthread_start_func_t start_routine, void *arg, int priority);
- Quando uma tarefa é criada deve ter o valor da prioridade igual a 1 e o valor do nice igual a 0.
- Implemente a função ***int sthread_nice(int nice)*** que modifica o valor do nice para a tarefa onde é chamada e retorna o valor da prioridade que a tarefa terá na próxima execução.
- Altere as funções correspondentes à implementação dos monitores para que estes continuem a funcionar com o novo escalonamento. De notar que a forma que as tarefas têm de se bloquear é através do uso dos monitores. Necessitará por isso desse mecanismo de sincronização para bloquear e desbloquear tarefas por forma a testar correctamente o escalonador;
- Implemente uma função cujo protótipo é void sthread_dump(). Esta função imprime o estado das listas de tarefas activas e bloqueadas. O output terá que ser obrigatoriamente o seguinte:

```
=== dump start ===
active thread
id: <id da tarefa onde foi chamado sthread_dump()> priority: <prioridade da
tarefa> vruntime: <valor da variável vruntime>
runtime: <tempo de execução real> sleeptime: <tempo em que esteve bloqueda>
waittime: <tempo de espera na RB-tree>

>>>> RB-Tree <<<<
id: <id da tarefa na B-Tree, ordenada por vruntime> priority: <prioridade da
tarefa> vruntime: < valor da variável vruntime >
```

[1] 1 *ticks* é o número de ticks de relógio que a tarefa esteve em execução.

```
runtime: <tempo de execução real> sleeptime: <tempo em que esteve bloqueda>
waittime: <tempo de espera na RB-tree>

id: <id da tarefa na B-Tree, ordenada por vruntime> priority: <prioridade da
tarefa> vruntime: < valor da variável vruntime >
runtime: <tempo de execução real> sleeptime: <tempo em que esteve bloqueda>
waittime: <tempo de espera na RB-tree>

>>>>SleepList<<<<

Igual por ordem de tempo por desbloquear


>>>>BlockedList<<<<

Igual por ordem de tempo de bloqueio e diferenciado por cada mutex/monitor
==== Dump End ====
```

Para mais detalhes, ver o Anexo A.

Parte II – Sistema de Ficheiros

Objectivo

Pretende-se melhorar o desenho e implementação de um sistema de ficheiros já existente, que se caracteriza pela presença de importantes limitações que afectam o seu desempenho e eficiência, quer no armazenamento, quer no tempo de resposta a pedidos de clientes, conforme se descreve de seguida. O objectivo desta 2ª parte do projecto consiste em incorporar no sistema de ficheiros, um conjunto de melhorias face a algumas dessas limitações.

Contexto

A arquitectura base do sistema de ficheiros é apresentada na Fig. 1. O dispositivo de memória secundária é controlado por um servidor de sistema de ficheiros. O sistema de ficheiros suporta múltiplos níveis de directorias. No entanto, o sistema não utiliza qualquer tipo de cache nem permite a remoção de ficheiros. Os detalhes do desenho do sistema de ficheiros devem ser consultados no Anexo B.

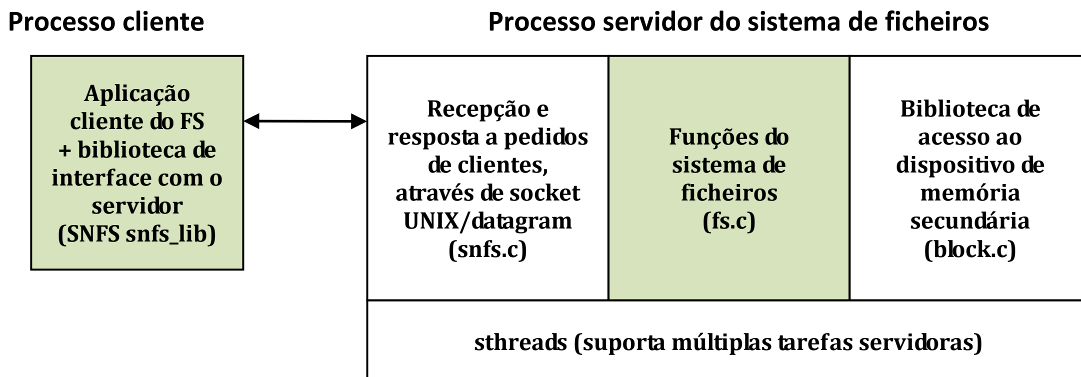

Fig. 1 A arquitectura base do sistema de ficheiros

O acesso ao dispositivo é feito através de uma biblioteca (*block.c*) que permite ler e escrever blocos de/para um dispositivo fictício. Os dispositivos também não incluem qualquer tipo de cache, pelo que a leitura e escrita de blocos de/para um disco são operações potencialmente demoradas e que são tratadas sequencialmente.

Os pedidos ao sistema de ficheiros chegam de aplicações cliente, cada uma a correr num processo independente na mesma máquina que o servidor. Os pedidos são enviados e respondidos através de um socket *datagram* UNIX do servidor, cujo nome é previamente conhecido.

Diversos clientes podem fazer pedidos ao mesmo servidor, que por isso pode receber múltiplos pedidos em paralelo. O sistema de ficheiros tira partido da biblioteca *sthreads* (a mesma implementada na primeira parte do projecto) para manter múltiplas tarefas, que são capazes de servir pedidos em paralelo.

Cada cliente consiste numa aplicação que, para aceder ao servidor do sistema de ficheiros, invoca funções da biblioteca *snfs_lib*. A biblioteca é, então, responsável

por converter tais pedidos para uma variante do protocolo SNFS (ver Anexo B) através do qual consegue interagir com o servidor.
Todos estes componentes são disponibilizados à partida. Os componentes que deverão ser modificados para a parte II estão indicados a sombreado na figura acima.

Requisitos

Claramente, o sistema de ficheiros é pouco interessante em vários aspectos, sendo que a interface de programação do sistema de ficheiros SNFS fornecida carece de importantes funcionalidades. Para a segunda parte do projecto, pretende-se a incorporação das seguintes melhorias:

1. Remoção de ficheiros

Pretende-se que se implemente uma função que apague um ficheiro do *file system* e liberte os blocos por ele usados, devendo estes blocos ficar novamente disponíveis para serem usados para outros ficheiros.

2. Cópia de ficheiros

Implementação do mecanismo de cópia de ficheiros ou directorias do *file system*.

3. Concatenação (append) de ficheiros

Realização dum mecanismo que implementa operação de concatenação (*append*) de ficheiros.

4. *Copy-on-write*

É habitual encontrar muitos conteúdos redundantes entre diferentes ficheiros em *file systems* comuns. Para minimizar o espaço ocupado por conteúdos redundantes, pretende-se que se implemente um esquema de *copy-on-write*. Sempre que há um pedido de cópia de um ficheiro, o novo ficheiro criado deve partilhar os mesmos blocos que o ficheiro original. Isto é, não devem ser alocados novos blocos para o novo ficheiro, já que todos os seus blocos têm inicialmente o mesmo conteúdo que o original.
Naturalmente, tal simetria pode quebrar-se à medida que algum(ns) dos blocos de um dos ficheiros (tanto o original como o novo) sejam alterados. Nessa altura, o bloco que antes era partilhado por 2 (ou mais) ficheiros deverá ser separado em 2 (ou mais) blocos, um com o conteúdo original (e referenciado pelos ficheiros que continuem com o conteúdo original) e outro com o novo conteúdo (referenciado pelo ficheiro sobre o qual a escrita foi feita).

5. Desfragmentação do sistema de ficheiros

Com a remoção e criação de ficheiros no sistema de ficheiros, o espaço de armazenamento que este ocupa fica fragmentado. Desta forma pretende-se que seja implementada uma função que percorra toda memória do sistema de ficheiros e a compacte; ou seja, os blocos ocupados devem ser sequenciais e os blocos dum ficheiro devem estar em posição contígua.

6. *Caching* de blocos

Implementar no servidor uma *cache* de blocos com política de substituição de blocos baseada no algoritmo do Não Usado Recentemente, NRU (*Not Recently Used*) e com política de escrita na *cache* baseada no método de escrita atrasada (*write-back*).

As entradas que sejam alteradas enquanto estão em *cache* não devem ser escritas imediatamente em disco (*flushed)*. A escrita (atrasada) é adiada até que: (i) a entrada seja seleccionada pela política de substituição NRU, ou (ii) depois de ter decorrido um intervalo de 10 segundos desde que a escrita em *cache* aconteceu.

Esta *cache* deve ser mantida em memória primária, usando uma estrutura de dados que deverá conter, obrigatoriamente, os seguintes campos:

- Bit V de validade da entrada na *cache*;
- Bits R e M do método de substituição NRU;
- Bit D, ou *dirty-bit*, do método de escrita atrasada (*write-back*);
- Nº do Bloco;
- Conteúdo do Bloco.

Os limites da *cache* de blocos devem ser facilmente parametrizáveis em tempo de compilação. Por omissão, a *cache* de blocos deve ter 8 entradas.

7. *Dumps*

Para que seja possível testar o sistema de ficheiros, assim como para uma melhor compreensão do mesmo pretende-se que sejam implementadas duas funções de *dump*:

- **Dump de blocos ocupados**: que mostra todos os blocos ocupados do disco com a indicação do nome dos ficheiros/directorias que estão a utilizar cada um desses blocos;
- **Dump da cache de blocos**: que mostra o conteúdo das suas entradas.

O formato dos *dumps* deve ser conforme os outputs mostrados na secção 3 do Anexo B.

8. Requisito transversal: sincronização eficiente

Transversalmente, todas as funções do sistema de ficheiros devem ser servidas da forma o mais paralela possível, para tirar partido de situações em que o servidor receba pedidos de múltiplos clientes em paralelo. A implementação da sincronização pode ser feita usando monitores e/ou trincos.

Entrega e Avaliação

Entrega intercalar (*Checkpoint*)

Data da entrega intercalar: 2 de Novembro às 12:00

Para a entrega intercalar, os alunos devem entregar uma implementação completa da Parte I do projecto. A entrega é por via electrónica através do sistema Fénix. O formato do ficheiro de entrega com o código será indicado oportunamente. A versão intercalar que cada grupo submete será avaliada na aula prática de cada grupo, na semana seguinte à entrega.

Entrega Final

Data da entrega final: 6 de Dezembro às 12:00

A entrega final consiste na submissão das Partes I e II, e implica obrigatoriamente duas fases:

1ª - Via electrónica

A entrega por via electrónica será efectuada no dia 6 de Dezembro até às 12h através do sistema Fénix. O formato do ficheiro de entrega com o código será indicado oportunamente.

2ª - Envelope

Para além da entrega electrónica deve ser entregue, até às 12h do mesmo dia, um envelope fechado identificado com o dia e hora do turno, número de grupo e o nome do docente das práticas.

O código deve ser impresso em 2 páginas por folha, frente e verso, sem linhas cortadas, indentado e devidamente comentado. Essa entrega é realizada nos seguintes locais:

- Pólo Alameda: Reprografia do DEI;
- Pólo Taguspark: Portaria do campus.

Visualizações do Projecto

As visualizações dos projectos realizam-se na semana **de 6 a 10 de Dezembro**.

Discussão do Projecto

As discussões dos projectos terão lugar **de 13 a 17 de Dezembro**. A modalidade de reserva dos horários das discussões será oportunamente anunciada na página da cadeira. De notar que nas discussões serão feitas perguntas sobre a implementação das rotinas que são fornecidas à partida, portanto não basta usá-las, há que entendê-las.

A informação sobre a entrega e avaliação aqui descrita está sujeita a alterações. Se tal suceder, serão afixados avisos na página da cadeira. Por este motivo recomenda-se a sua consulta periódica.

Anexo A – Pacote simplethreads

1. Material fornecido

O material dado consiste num pacote que permite criar tarefas que correm em modo utilizador. Esse pacote é o *sthreads.tgz* que se encontra na página da disciplina. Alguns ficheiros que destacamos neste pacote são:

Directório / Ficheiro	Conteúdo
sthread_lib/	Biblioteca de tarefas
sthread_lib/sthread_user.c	Onde vão ser implementadas as funções necessárias para a biblioteca de tarefas.
sthread_lib /sthread_ctx.{c,h}	Módulo para criar novas pilhas de execução e para comutar entre elas
sthread_lib /sthread_switch_i386.h	Funções *assembly* para comutar entre pilhas e para salvaguardar registos
sthread_lib/sthread_time_slice.{c,h}	Suporte para gerar *signals* e para controlá-los
include/	Contém *sthread.h* e *config.h* que são as interfaces públicas, a incluir em programas que usem ou comuniquem com os programas ou bibliotecas referidas.
test-sthreads/	Contém vários testes para a biblioteca sthreads

Tabela 1

As rotinas no ficheiro *sthread_ctx.h* realizam toda a manipulação da pilha, alterações ao PC (*program counter*), guardam registos e outras manipulações de baixo nível. O objectivo da Parte I do trabalho é a administração dos contextos de execução das tarefas, pelo que apenas é necessário implementar as funções que se encontram no sthread_user. Durante a implementação modificar apenas o ficheiro *sthread_user.c* e, eventualmente, os ficheiros *sthread.{h,c}*, enquanto que sthread_ctx_t não deve ser modificado directamente; para tal devem usar em vez disso as rotinas declaradas em *sthread_ctx.h*. Considerando o sistema em camadas (Fig. 2) apenas tem que implementar o "rectângulo a cinzento".

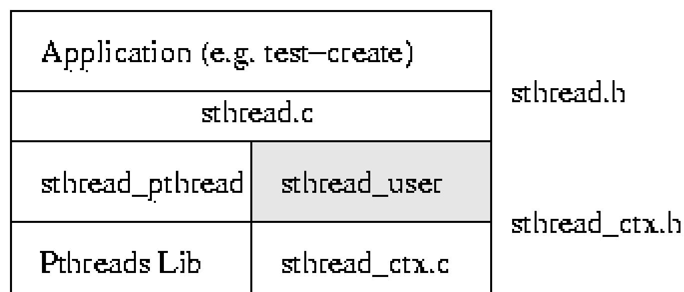

Fig. 2

Na camada superior encontra-se a aplicação que vai usar o pacote *sthreads* (através da API definida em `sthread.h`). `sthread.c` vai então chamar as rotinas implementadas neste trabalho em `sthread_user.c` ou as rotinas em `sthread_pthread.c` que fazem uso das *pthreads* (alterando a variável PTHREADS nas `Makefile`). O `sthread_user.c` por sua vez é construído em cima das rotinas do `sthread_ctx` (como descrito em `sthread_ctx.h`).
As aplicações (camada superior) não devem usar mais rotina nenhuma da biblioteca excepto as definidas em `sthread.h`. As aplicações não podem usar rotinas definidas noutros ficheiros nem podem "saber" como estão implementadas as tarefas. As aplicações apenas pedem para criar tarefas e podem requerer `yield` ou `exit`. Também não devem manter listas de tarefas em execução. Isso é a função do módulo marcado a cinzento na Fig. 2.
De igual forma, o rectângulo cinzento – `sthread_user.c` – não deve saber como `sthread_ctx` está implementado. Deve usar as rotinas definidas em `sthread_ctx.h`.
De seguida apresentam-se algumas notas sobre as várias partes da implementação da biblioteca de tarefas-utilizador.

2. Notas sobre as Funções da Biblioteca *sthreads*

Quanto à gestão de tarefas e escalonamento:

- sthread_create() cria uma nova tarefa que se irá executar quando for seleccionada pelo despacho. Qualquer tarefa só deve deixar de correr quando ela própria executar o sthread_yield() ou quando se bloqueia;
- Use as rotinas fornecidas em *sthread_ctx.h*. Não necessita escrever nenhum código *assembly*, nem manipular registos, nem entender detalhadamente como está organizada a pilha;
- A rotina que é passada para sthread_create() corresponde ao programa principal da tarefa pelo que se esta terminar, tem que se assegurar que os recursos são libertados (ou seja, sthread_exit() tem que ser chamado quer explicitamente pela rotina que é passada para sthread_create() quer implicitamente após a rotina terminar);
- Deve libertar todos os recursos quando a tarefa termina. Não deve no entanto libertar a pilha de uma tarefa que se encontre ainda em execução (nota: para libertar a pilha use sthread_free_ctx());
- A tarefa inicial deve ser tratada como qualquer outra tarefa. Por isso, caso se queira pará-la, deve ser criada um estrutura sthread_t para manter a informação do seu contexto de execução.
- Tenha cuidado com o uso de variáveis locais após chamar sthread_switch() já que os seus valores podem ser diferentes de anteriormente (é uma pilha de execução diferente);
- A função de inicialização da biblioteca sthreads, sthread_user_init, inicia o escalonador de tempo partilhado invocando sthread_time_slices_init, lançando assim um signal periódico, cujo função de tratamento inclui o algoritmo de despacho.

Quanto aos mutexes e monitores:

- Compreenda como bloquear uma tarefa, fazendo-a esperar numa fila. Como obtém a tarefa que se bloqueou? Como altera o contexto dela para passar para outra tarefa? Como volta a corrê-la?
- Quando se desbloqueia uma tarefa, não ocorre imediatamente uma mudança de tarefa. A tarefa fica executável e será executada quando seleccionada pelo despacho;
- Para colocar sincronização no sistema de tarefas existem na biblioteca duas primitivas de sincronização: atomic_test_and_set e atomic_clear que permitem realizar a exclusão mútua a baixo nível.

Outras Notas

- Se desactivar os *signals* não se esqueça de os activar em todos o caminhos possíveis do seu programa. Provavelmente vai querer desactivar os *signals* durante todo o tempo que estiver dentro de um yield e activá-las ao completar o sthread_switch. De notar que sthread_switch pode retornar para dois locais diferentes: para a linha a seguir a uma chamada a um sthread_switch ou para a sua função de começo da tarefa quando comuta para uma nova tarefa pela primeira vez. Active os *signals* nos dois locais;
- Não deve executar código da aplicação com os *signals* desligados;
- Verifique que o teste para *time-slices* funciona (*test-time-slices.c*) e verifique que todos os outros testes funcionam ainda.
- Uns bons valores para o período das interrupções são 10 milisegundos.
- Para comparar programas com e sem preempção comente a chamada à rotina sthread_time_slices_init().

Anexo B – Pacote API SNFS, Servidor Multi-Tarefas e Sistema de Ficheiros SNFS

1. Material fornecido

O material dado consiste num pacote que implementa os seguintes componentes:

- Cliente SNFS;
- Servidor Multi-Tarefa;
- Sistema de Ficheiros SNFS.

A **biblioteca SNFS** desenvolvida permite a uma aplicação cliente aceder ao sistema de ficheiros de um servidor remoto SNFS. O código das aplicações cliente liga-se com a biblioteca SNFS que, por sua vez, está dividida em duas camadas (ver Fig. 3):

- Uma camada de suporte à comunicação com o servidor (API SNFS).
- Uma camada de interface com a aplicação (Interface Sistema de Ficheiros).

A camada API SNFS consiste num conjunto de rotinas que:

- formata as mensagens SNFS;
- comunica com o servidor através de *sockets* domínio Unix sem ligação (SOCK_DGRAM). A utilização deste tipo de *sockets* possui a vantagem de delimitar automaticamente as mensagens e de enviar os dados nas mensagens sem necessidade de conversão.

Para esta parte é disponibilizado o material seguinte:

- Servidor SNFS funcional.
- Formato das mensagens do protocolo SNFS para todos os serviços, excepto as indicadas "por implementar" na Tabela 1 da secção 2 deste anexo.
- Esqueleto da biblioteca SNFS para os clientes.

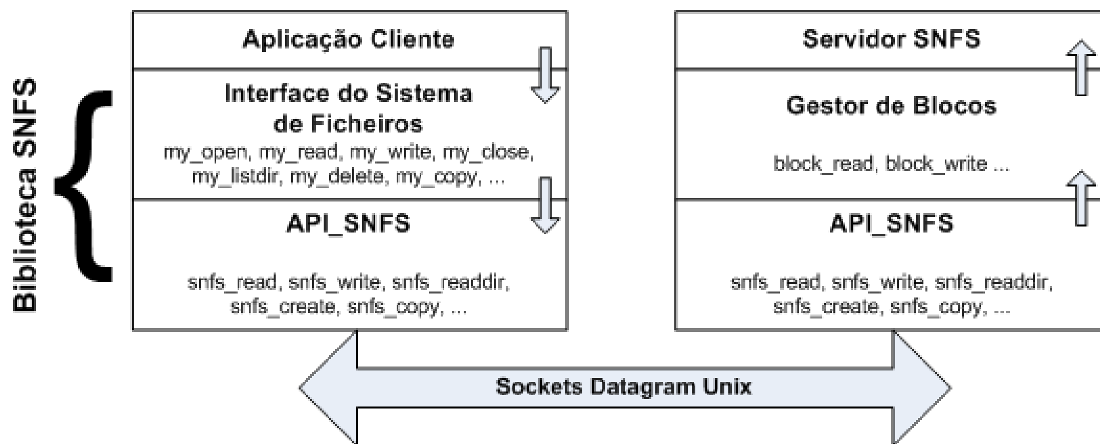

Fig. 3 Arquitectura do Cliente-Servidor SNFS

A camada Interface Sistema de Ficheiros oferece aos programadores uma interface de programação semelhante àquela que é disponibilizada pela biblioteca standard da linguagem C. O cliente utiliza a API SNFS sempre que necessita de comunicar com o servidor. Internamente, esta camada gere os ficheiros abertos pela aplicação cliente. No intuito de simplificar a complexidade do sistema, na implementação dos pacotes fornecidos não se consideraram dois aspectos que são imprescindíveis em situações reais:

- Acesso concorrente à biblioteca por várias tarefas do mesmo processo cliente.
- Acesso ao mesmo ficheiro por dois processos cliente independentes.

O **servidor SNFS** que é fornecido aos alunos funciona no modo multi-tarefa. A arquitectura multi-tarefa do servidor SNFS obedece ao seguinte desenho:

- Existe um número limitado de tarefas consumidoras (TC) que processam e respondem aos pedidos dos clientes.
- Existe uma única tarefa (TP) que descobre quando existem novos pedidos e que os distribui pelas tarefas consumidoras.

A tarefa TP sincroniza-se com as tarefas TC, de modo a que:

Quando chega um novo pedido, TP assegura que uma tarefa TC, e apenas uma, fica responsável por este pedido:

- Se todas as tarefas TC estiverem ocupadas, o mecanismo de sincronização assegura que o pedido não é esquecido.
- O número de tarefas TC é definido em tempo de compilação.

O **Sistema de Ficheiros** suportado pelo servidor de SNFS é muito simples, possuindo as seguintes características:

- O tamanho máximo dos ficheiros é de 10 blocos;
- Suporta a criação de subdirectórios no directório raíz;
- Tamanho dos blocos tem dimensão fixa de 512 bytes. O número de blocos é definido em tempo de compilação.

- A arquitectura do sistema de ficheiros é baseada numa simplificação dos *i-nodes* Unix , com 10 entradas directas para blocos de dados. O tamanho de um *i-node* é de 64 bytes. O número de *i-nodes* por omissão é de 64 (a tabela de *i-nodes* ocupa, por isso 8 blocos), mas este valor pode ser definido em tempo de compilação.
- Os directórios são ficheiros estruturados sob a forma de uma tabela de entradas. Cada entrada tem 16 bytes e é constituída por: nome do ficheiro/directório e número do *i-node*. Os nomes são limitados a 14 caracteres e o número do *i-node* a 2 bytes. Para simplificar a listagem dos directórios remotamente, considera-se que os directórios podem ocupar, no máximo 4 blocos, isto é suportam até 128 entradas.
- A disposição dos dados nos blocos é a seguinte:
- Bloco 0: reservado para o bitmap de blocos livres.
- Bloco 1: reservado para o bitmap de *i-nodes* livres.
- Blocos 2-9: reservados para a tabela de *i-nodes*.
- Blocos >= 10: para os dados dos ficheiros e directórios.

2. Protocolo SNFS

O protocolo SNFS define a comunicação entre clientes e servidores SNFS. O SNFS deve suportar o subconjunto dos serviços do protocolo NFS Versão 2 (ver versão completa em http://tools.ietf.org/html/rfc1094) descritos na Tabela 2.

Serviço	Argumentos	Resultado	Descrição
SNFS_LOOKUP	fhandle dir filename name	stat status: se STAT_OK fhandle file unsigned fsize	Obtém o identificador “fhandle” e respectivo tamanho “fsize” do ficheiro ou directório “name” fhandle file localizado no directório “dir”, se a resposta obtida for STAT_OK. Se “fhandle” for 0, refere-se ao directório raiz.
SNFS_READ	fhandle file unsigned offset unsigned count	stat status: se STAT_OK bytes data	Devolve até “count” bytes de “data” do ficheiro “file”, a partir do byte “offset” a contar do início do ficheiro. O primeiro byte do ficheiro corresponde ao offset 0.
SNFS_WRITE	fhandle file unsigned offset unsigned count byte data	stat status: se STAT_OK unsigned fsize	Escreve “data” a partir do byte “offset” a partir do início do ficheiro “file”. O primeiro byte do ficheiro está no offset 0. A operação de escrita é atómica. Os dados de uma escrita não serão misturados com os dados da escrita de outro cliente. Se for bem sucedida, a escrita devolve a dimensão actual do ficheiro em “fsize”
SNFS_CREATE	fhandle file filename name	stat status: se STAT_OK fhandle file	Os atributos iniciais do ficheiro são dados por “attributes”. Se o resultado é STAT_OK, o ficheiro foi criado com sucesso e “file” e “attributes” contém o “fhandle” e os atributos do ficheiro. Caso contrário a operação falhou e nenhum ficheiro foi criado.
SNFS_MKDIR	fhandle dir	stat status:	Cria o novo directório “name” no directório “dir”. A resposta STAT_OK indica que o

	filename name	se STAT_OK fhandle file	directório foi criado com sucesso e “file” contém o fhandle do directório. Caso contrário, a operação falhou e o directório não foi criado.
SNFS_READDIR	fhandle dir unsigned count	stat status: se STAT_OK entry* list unsigned count	Retorna um número variável de entradas do directório até “count” bytes do directório dado por “dir”. Se o valor retornado é STAT_OK, então segue-se de um número variável de “entry”s. Cada entry é uma estrutura com o nome da entrada (ficheiro ou directório), tamanho do nome e indicação se o nome se trata de ficheiro/directório.
SNFS_REMOVE (Por implementar)	fhandle dir fname name	stat status: se STAT_OK fhandle file	Remove o ficheiro"name" do directório “dir”. Se o resultado é STAT_OK, o ficheiro foi removido com sucesso e “file” contém o respectivo “fhandle”. Caso contrário a operação não teve sucesso.
SNFS_COPY (Por implementar)	fhandle dir1 fname name1 fhandle dir2 fname name2	stat status: se STAT_OK fhandle file	Copia o ficheiro “file1” da directoria “dir1” para um novo ficheiro “file2” na directoria “dir2”. Se o resultado é STAT_OK, o ficheiro foi criado com sucesso e “file” contém o “fhandle” do ficheiro. Caso contrário a operação falhou e nenhum ficheiro foi criado.
SNFS_APPEND (Por implementar)	fhandle dir1 fname name1 fhandle dir2 fname name2	stat status: se STAT_OK unsigned fsize	Concatena o conteúdo do ficheiro “name1” da directoria “dir1” com o conteúdo do ficheiro “name2” da directoria “dir2”. Se o resultado é STAT_OK, a concatenação teve sucesso e “fsize” contém o tamanho do ficheiro “name1”. Caso contrário a operação falhou.
SNFS_DEFRAG (Por implementar)		stat status	Efectua a operação de desfragmentação do disco do sistema de ficheiros. Se o resultado é: STAT_OK, a desfragmentação foi realizada com sucesso; STAT_BUSY se está em curso uma outra operação de desfragmentação; STAT_ERROR se não foi possível a desfragmentação por falta de espaço no sistema de ficheiros.
SNFS_DISKUSAGE (Por implementar)		stat status	Efectua o dump, do lado do servidor, dos blocos do sistema de ficheiros utilizados e correspondentes ficheiros/directorias que os ocupam. Se o resultado é STAT_OK a operação teve sucesso. Caso contrário a operação falhou.
SNFS_DUMPCACHE (Por implementar)		stat status	Efectua o dump, do lado do servidor, das entradas da cache de blocos do servidor. Se o resultado é STAT_OK a operação teve sucesso. Caso contrário a operação falhou.

Tabela 2

3. API do Cliente

A interface de programação do sistema de ficheiros SNFS é semelhante à oferecida pela biblioteca *standard* da linguagem C, **estando disponíveis**:

- `int my_init_lib();`
  Inicializa as estruturas internas da biblioteca de ficheiros.

- `int my_open(char * nome, int flags);`
  Devolve um inteiro que identifica o ficheiro em operações posteriores (handle). O argumento flags é usado para modificar o comportamento da função, sendo que apenas existe a flag com valor 1, que indica que o ficheiro deve ser criado caso já não exista.

- `int my_read(int fileID, char * buffer, unsigned numBytes);`
  Lê a partir de um ficheiro, para um buffer em memória, um número especificado de bytes. Devolve o número de bytes lidos, ou zero caso esteja no final do ficheiro.

- `int my_write(int fileID, char * buffer, unsigned numBytes);`
  Escreve para um ficheiro, o conteúdo de um buffer em memória, com o tamanho especificado.

- `void my_close(int fileID);`
  Fecha o ficheiro identificado pelo argumento *fileID*.

- `int my_listdir(char* path, char **filenames, int* numFiles);`
  Escreve para o ponteiro **filenames* o endereço de memória onde estão contidos os nomes dos ficheiros que se encontram no directório *path* do sistema de ficheiros, separados por '\0', e no inteiro *numberOfFiles* a quantidade dos mesmos ficheiros.

Adicionalmente, os alunos **devem acrescentar** à mencionada interface de programação do sistema as seguintes funções:

- ❒ `int my_remove(char *name);`
  Apaga o ficheiro ou directório identificado por *name*. Devolve 1 se teve sucesso, 0 caso contrário.

- ❒ `int my_copy(char *name1, char *name2);`
  Copia o ficheiro ou directório identificado por name1 para outro ficheiro ou directório identificado por name2. Devolve 1 se teve sucesso, 0 caso contrário.

- ❒ `int my_append(char *name1, char *name2);`
  Concatena ao conteúdo do ficheiro identificado por name1 o conteúdo do ficheiro identificado por name2. Devolve 1 se teve sucesso, 0 caso contrário. Em caso de sucesso, apenas o conteúdo do ficheiro name1 se altera.

- ❒ `int my_defrag();`
  Efectua do lado do servidor a operação de desfragmentação do disco associado ao sistema de ficheiro. Devolve 1 se teve sucesso, 0 caso esteja em curso uma operação de desfragmentação que ainda não terminou e -1 caso não haja disponibilidade de espaço em disco para a realização da desfragmentação.

❐ *void my_diskusage();*

Imprime, do lado do servidor, os blocos ocupados do disco com a indicação do nome dos ficheiros/directorias que estão a utilizar cada um desses blocos. O output desta função terá que ser obrigatoriamente o seguinte:

```
===== Dump: FileSystem Blocks =======================
blk_id: < nº do bloco>
file_name1: <pathname do 1º ficheiro/directoria que usa o
bloco>
file_name2: <pathname do 2º ficheiro/directoria que usa o
bloco>
                 . . . . . .
file_nameN: <pathname do Nº ficheiro/directoria que usa o
bloco>
*******************************************************
```

❐ *void my_dumpcache();*

Imprime, do lado do servidor, o conteúdo da cache de blocos no seguinte formato:

```
===== Dump: Cache of Blocks Entries =======================
Entry_N: <Nº da Entrada na cache>
V: < Bit V de validade> D: <Dirty-Bit de escrita atrasada>
R: <Bit R do método NRU>  M: <Bit M do método NRU>
Blk_Cnt: <Conteúdo do Bloco (em Hex?)>
************************************************************
```