INSTITUTO SUPERIOR TÉCNICO

Departamento de Engenharia Informática

Enunciado do Projecto
Sistemas Operativos

LEIC/LERC

2008/09

Biblioteca Multitarefa em Modo Utilizador.

Sistema de Ficheiros Distribuído Simple NFS.

Índice

Introdução 3
Parte I – Biblioteca *sthreads* 3
Objectivos 3
Concretização 4
Escalonamento com prioridades 4
Mecanismos de sincronização 4
Parte II – Cliente SNFS 6
Objectivos 6
Concretização 6
Parte III – Servidor multi-tarefa 7
Objectivos 7
Concretização 7
Parte IV – Funcionalidades do Sistema de Ficheiros 7
Objectivos 7
Concretização 8
Entrega e Avaliação 9
Entregas intercalares 9
Entrega Final 9
Visualização e Discussão do Projecto 10
Anexo A – Pacote simplethreads+SNFS 11
1. Material fornecido 11
2. Notas sobre as Funções da Biblioteca Sthreads 12
Outras Notas 13

Introdução

Este trabalho tem por objectivo familiarizar os alunos com os seguintes aspectos do desenvolvimento de sistemas operativos:

- Gestão e escalonamento de actividades pseudo-concorrentes
- Sincronização de tarefas
- Comunicação entre processos
- Arquitectura do sistema de ficheiros

O projecto consiste na extensão de uma biblioteca de pseudo-tarefas, na definição da arquitectura e programação de uma versão simplificada de um sistema de ficheiros distribuído inspirada na arquitectura do sistema de ficheiros distribuído SUN NFS – *Network File System*. O sistema *Simple NFS* permite que vários processos (designados por clientes) partilhem ficheiros que são armazenados por um processo servidor.

O projecto será desenvolvido de forma modular, com várias etapas faseadas no tempo. Nomeadamente, os alunos deverão realizar os seguintes módulos:

- Estender uma biblioteca de gestão de tarefas utilizador, denominada *sthreads*, de modo a suportar diversas funcionalidades frequentes em sistemas multi-tarefas e úteis para a realização do resto do projecto.
- Analisar e programar o processo servidor de ficheiros.
- Analisar e programar o suporte para a comunicação entre os processos cliente e o processo servidor.
- Definir e programar uma biblioteca de funções (*API – Application Programming Interface*) através da qual as aplicações acedem ao sistema de ficheiros e cuja interface esconde das aplicações (tanto quanto possível) o facto de o sistema ser distribuído.

Os alunos não terão que desenvolver todos os módulos de raiz. Pretende-se que estendam o código que será fornecido pelo corpo docente. Os alunos devem desenvolver o projecto de acordo com o faseamento proposto neste enunciado e que se relaciona com as etapas de avaliação.

O trabalho está estruturado em quatro partes, que devem ser realizadas de forma sequencial.

Parte I – Biblioteca *sthreads*

Objectivos

A biblioteca *sthreads* é uma biblioteca de gestão de tarefas que se executam em modo utilizador, no contexto de um processo. 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 com as seguintes funcionalidades, que deverão ser implementadas pela seguinte ordem:

- Primitivas de sincronização entre tarefas: mutexes e monitores.
- Escalonamento de tempo partilhado com base em prioridades.

Concretização

O conjunto de funcionalidades a adicionar é apresentado em dois grupos. No Anexo A encontram-se informações complementares para a concretização de cada um destes grupos de funcionalidades.

Mecanismos de sincronização

Devem ser acrescentados os mecanismos de sincronização mutex e monitor à biblioteca utilizador. O esqueleto das funções encontram-se no ficheiro *sthread_user.c*. Nesta parte do trabalho tem que:

- Implementar as estruturas para os mutexes (struct _sthread_mutex_t);
- Implementar as operações dos mutexes (sthread_mutex_init(), sthread_mutex_free(), sthread_user_mutex_lock(), sthread_user_mutex_unlock()).

Os mutexes devem ter a semântica habitual que garante as propriedades de uma secção critica. Quando nenhuma tarefa se executa na secção critica a primeira que fizer lock, fecha o respectivo trinco fazendo com que invocações posteriores a lock bloqueiem as respectivas tarefas. Quando unlock é invocado, se existir uma tarefa à espera, é desbloqueada mantendo-se o trinco fechado, caso contrário o trinco é aberto.

- Implementar as estruturas para os monitores (struct _sthread_monitor_t);
- Implementar as operações dos monitores (sthread_monitor_init(), sthread_mon_free(), sthread_monitor_enter(), sthread_monitor_exit(), sthread_monitor_wait(), sthread_monitor_signal()).

Os monitores são definidos no código pelas primitivas sthread_user_monitor_enter e sthread_user_monitor_exit, e garantem que o bloco de código assim definido é executado em secção critica. No interior do monitor, os processos podem sincronizar-se explicitamente através de uma fila de sincronização. A primitiva sthread_user_monitor_wait (que só pode ser invocada entre um enter e um exit) permite a uma tarefa bloquear-se numa fila de sincronização, à espera que outra tarefa faça signal, libertando o monitor e a correspondente secção crítica. A primitiva sthread_user_monitor_signal (que também só pode ser invocada entre um enter e um exit) irá desbloquear uma das tarefas que estejam bloqueadas nesse monitor, se existir alguma, caso contrário, não tem qualquer efeito. As filas de sincronização dos monitores não são semáforos. A primitiva sthread_user_monitor_signal não tem memória pelo

que não acumula "créditos" no monitor, limita-se a desbloquear uma tarefa que esteja bloqueada nesse momento.

É importante perceber a relação entre a exclusão mútua e a sincronização explícita na fila de sincronização. Uma tarefa que é desbloqueada depois de um sthread_user_monitor_wait só tem acesso à secção crítica do monitor onde se bloqueou depois de a tarefa que a desbloqueou dela ter saído, pois de outra forma a secção crítica não seria garantida.

Para realizar esta parte do trabalho deve usar o seguinte material do pacote *sthreads*:

- Estruturas de dados que representam o estado das tarefas;
- O esqueleto destas rotinas de sincronização providenciadas no ficheiro *sthread_user.c*.

Para a realização desta parte, só é necessário alterar os ficheiros *sthread_user.c* e eventualmente os ficheiros *queue.h* e *queue.c*.

Escalonamento com prioridades

Pretende-se estender o escalonamento inicial da biblioteca *sthreads* que apenas implementa uma política de tempo partilhado com base numa lista circular para que passe a escalonar as tarefas com um algoritmo multilista da seguinte forma:

- Cada tarefa é criada com uma nível de prioridade entre 1 e 4, em que 1 é o nível das tarefas mais prioritárias.
- As tarefas executam um *quantum* de tempo até ao fim a menos que se bloqueiem ou cedam explicitamente o processador através da função sthread_yield().
- Quando uma tarefa perde o processador por se ter bloqueado ou chamado sthread_yield(), sobe 2 níveis de prioridade (sem ultrapassar o nível 1), e desce 1 nível de prioridade caso tenha concluído o *quantum*.
- O escalonador escolhe, para se executar a seguir, a tarefa da lista mais prioritária que esteja executável.

A biblioteca disponibilizada suporta comutação provocada explicitamente através da invocação da rotina sthread_yield() e efectua a comutação de cada vez que ocorre um *signal* periódico que simula a interrupção de relógio. O material fornecido para esta parte do trabalho ajuda nos seguintes aspectos:

- Como gerar e tratar *signals*;
- Primitivas para ligar e desligar a rotina de tratamento do *signal* de forma a garantir exclusão mútua necessária para a correcta codificação das funções de gestão de tarefas.

Terá de ser alterada a função de criação de threads, sthread_create, de forma a incluir um argumento indicativo da prioridade inicial da thread que está a ser criada. Deve também ser desenvolvida uma função que liste a identificação das tarefas existentes, o seu nível de prioridade e outra informação que os alunos achem relevante para a depuração do código. Essa função deverá ter a assinatura dump_stats(char * name) e deverá escrever no ficheiro name,

uma tabela com colunas separadas por virgula em que caad linha descreve uma das tarefas existentes sendo os dois primeiros campos o identificador da tarefa e a prioridade da tarefa.

Parte II – Cliente SNFS

Objectivos

Pretende-se desenvolver a biblioteca SNFS que permite a uma aplicação cliente aceder ao sistema de ficheiros de um servidor remoto SNFS. O código das aplicações cliente deve ser ligado com a biblioteca SNFS que está dividida em duas camadas (ver Anexo A-3):

- Uma camada de suporte à comunicação com o servidor (API SNFS).
- Uma camada de interface com a aplicação (interface do sistema de ficheiros).

Concretização

Para permitir o teste destas funcionalidades será fornecida uma concretização funcional (mas não completa) do servidor. O servidor será lançado da linha de comandos com dois argumentos: o primeiro é o caminho para o *socket* Unix do servidor, o segundo é opcional e indica o caminho do ficheiro com uma imagem com que inicializa o sistema de ficheiros do servidor.

1. API SNFS

Esta camada consiste num conjunto de rotinas que i) formata as mensagens SNFS e ii) comunica com o servidor através de *sockets* domínio Unix sem ligação (`SOCK_DGRAM`).

Para esta parte é disponibilizado o material seguinte:

- Servidor SNFS funcional,
- Formato das mensagens do protocolo SNFS para todos os serviços (Anexo A-4),
- Esqueleto da biblioteca SNFS para os clientes com comunicação baseada em *sockets* sem ligação.

2. Interface Sistema de Ficheiros

Esta camada deverá oferecer aos programadores uma interface de programação semelhante àquela que é disponibilizada pela biblioteca standard da linguagem C. Por sua vez, deverá utilizar a API SNFS sempre que necessitar de comunicar com o servidor. Internamente, esta camada deve gerir os ficheiros abertos pela aplicação cliente.

Para simplificar, a solução não precisa de considerar as seguintes situações:

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

Parte III – Servidor multi-tarefa

Objectivos

O servidor SNFS que é fornecido funciona com uma única tarefa. Os alunos devem alterar o servidor para que funcione em modo multi-tarefa.

Concretização

Há uma tarefa gestora que faz a leitura dos pedidos dos clientes e os entrega explicitamente a uma das tarefas trabalhadoras.

Existe um conjunto fixo de tarefas trabalhadoras que aguardam que a tarefa gestora lhes entregue um pedido directamente, fazem o processamento do pedido e respondem ao cliente. A tarefa gestora escolhe, de entre as tarefas trabalhadoras, aquela à qual vai entregar um dado pedido, colocando o pedido num *buffer* de 1 posição específico dessa tarefa.

A tarefa gestora deve ser criada com um prioridade inicial superior à prioridade inicial das tarefas trabalhadoras.

A tarefa gestora tem de se sincronizar com as tarefas trabalhadoras usando os mecanismos de sincronização desenvolvidos na primeira parte do projecto. Se todas as tarefas trabalhadoras estiverem ocupadas, o mecanismo de sincronização deve assegurar-se que o pedido não é esquecido (existem várias soluções, os alunos devem escolher e defender a que lhes parece mais interessante). Sempre que não estejam processando um pedido, as tarefas trabalhadoras deverão ficar bloqueadas aguardando o próximo pedido.

O número de tarefas trabalhadoras deve ser definido em tempo de compilação.

Parte IV – Funcionalidades do Sistema de Ficheiros

Objectivos

O servidor de SNFS fornecido aos alunos é muito simples. Para além da conversão num servidor multi-tarefa deverão ser desenvolvidas as seguintes funcionalidade:

- O SNFS suporta apenas um único directório raiz. O sistema de ficheiros deverá passar a suportar a criação de sub-directórios desse directório.
- Modificar o sistema de ficheiros de modo a que ficheiros com dimensão até 40 bytes (dimensão livre no *inode*) não usem blocos. Os dados ficam armazenados no próprio *inode* do ficheiro. De forma idêntica para directórios com um máximo de 2 entradas. Caso esta funcionalidade seja implementada, para que o comando disk_usage funcione, deverá ser adicionado um `#define OPTIMIZE_INODE` no ficheiro *fs.h*.
- Implementar a função `optimize(int FileID, int optimize_free_space)` que optimiza a *organização* do disco. Caso o `FileID` se refira a uma directório a operação de optimização deverá ser

realizada sobre os blocos do directório em si, bem como todos os sub-directórios e ficheiros aí presentes. O processo de optimização deverá ter as seguintes características:

  - Primeiro, a optimização deverá consistir na reorganização o ficheiro de modo a que os blocos que o constituem fiquem contíguos.
  - Um segundo aspecto da optimização é o de eliminar a fragmentação do disco, ou seja, tornar contíguos todos os blocos livres do disco. A execução desta optimização é controlada pelo parâmetro optimize_free_space.
- O processo de optimização deve conter todos os mecanismos de sincronização necessários mas deverá bloquear o sistema de ficheiros o mínimo tempo possível.
- O servidor deverá permitir o acesso para leitura enquanto o disco estiver a ser optimizado. O acesso para escrita é opcional.
- Deverão ser adicionados os comandos disk_usage e optimize no cliente. Esta função imprime no terminal do servidor de ficheiros a distribuição dos blocos usados no sistema de ficheiros. A implementação do serviço será fornecida.
- O directório de raiz do sistema operativo deve chamar-se "root".

Concretização

O sistema de ficheiros fornecido possui as seguintes características:

- Os blocos têm dimensão fixa de 512 bytes. O número de blocos é definido em tempo de compilação (um valor razoável para testes é de 64 Kblocos que ocupa 32 MB em memória).
- A arquitectura do sistema de ficheiros é baseada numa simplificação dos *inodes* Unix, com 10 entradas directas para blocos de dados. O tamanho de um *inode* é de 64 bytes. O número de *inodes* por omissão é de 64 (a tabela de *inodes* 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 *inode*. Os nomes são limitados a 14 caracteres e o número do *inode* 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 *inodes* livres.
  - Blocos 2-9: reservados para a tabela de *inodes*.
  - Blocos ≥ 10: para os dados dos ficheiros e directórios.

A realização da parte IV do projecto implica alterar as estruturas de dados e a implementação dos serviços do servidor SNFS disponibilizado.

Entrega e Avaliação

A entrega do trabalho será feita em três fases: duas entregas intercalares e a entrega final.

Entregas intercalares

As entregas intercalares serão verificadas nas aulas práticas correspondentes de cada grupo:

- 1ª entrega intercalar: 26 de Outubro com visualização na aula prática seguinte da parte I do trabalho;
- 2ª entrega intercalar: 28 de Novembro com visualização na aula prática seguinte das partes II e III do trabalho.

As entregas intercalares podem contar para subir a nota de acordo com a seguinte fórmula:

Nota final do projecto = MAX (Nota original do projecto; 0.9*Nota original do projecto + 0.1*Nota das entregas intercalares),
onde a "Nota original do projecto" decorre da avaliação normal do projecto, e a nota das entregas intercalares será uma avaliação generosa do esforço despendido para completar o trabalho a tempo das entregas intercalares (por exemplo, na nota das entregas intercalares seremos bastante benevolentes em relação à qualidade do desenho e da implementação do sistema).

Entrega Final

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

1 – Via electrónica

A entrega por via electrónica será efectuada no dia 10 de Dezembro até às 12h através da página WWW da cadeira. O formato do ficheiro de entrega com o código será indicado oportunamente.

2 – Envelope

Para além da entrega electrónica deve ser entregue até às 15h do mesmo dia um envelope fechado identificado com o dia, hora, turno e número de grupo:

O código 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: Gabinete de Apoio às Licenciaturas.

Discussão do Projecto

Após a entrega dos projectos será afixado no sítio WWW da cadeira um horário de discussões dos projectos. As discussões dos projectos realizam-se nas semanas de 15 a 20 de Dezembro e de 6 a 7 de Janeiro. 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 disciplina. Por este motivo recomenda-se a sua consulta periódica.

Anexo A – Pacote simplethreads+SNFS

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 nesse pacote são:

Directório / Ficheiro	Conteúdo
README	Informação detalhada de como fazer um programa que usa as sthreads (hello_sthreads).
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 Ficheiros fornecidos no início do projecto

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 Partes 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.c. Durante a implementação modificar apenas o ficheiro *sthread_user.c*. Não modificar sthread_ctx_t directamente, usando em vez disso as rotinas declaradas em *sthread_ctx.h*. Considerando o sistema em camadas (Figura 1) apenas têm que implementar o rectângulo marcado a cinzento.

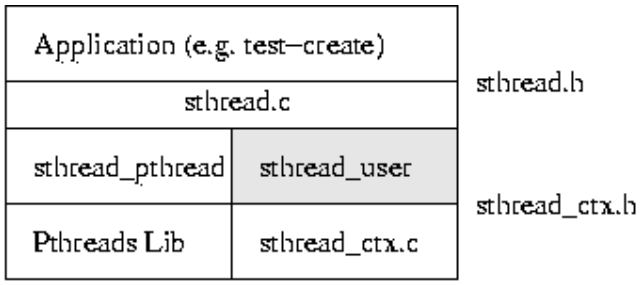

Figura 1

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 Figura 1.

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();
- 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);
- Apesar do uso de variáveis globais não ser recomendado, nalguns casos terão mesmo de ser usadas.
- 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().

3. Arquitectura do Cliente-Servidor SNFS

O sistema de ficheiros será implementado de acordo com a seguinte arquitectura.

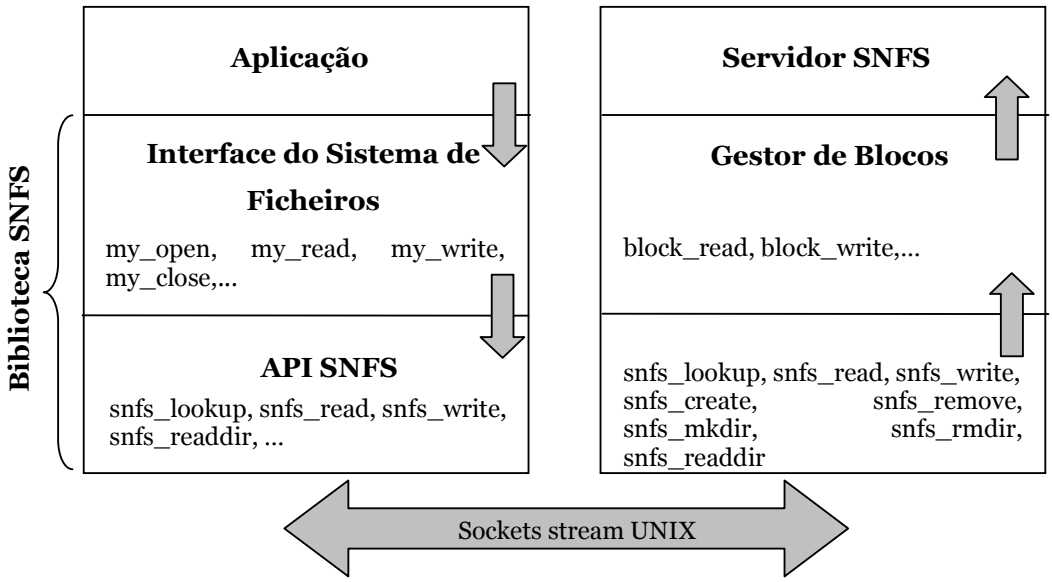

Figura 2 Arquitectura do cliente servidor SNFS

Notas:

- O servidor SNFS não mantém estado entre os pedidos. Logo, cada pedido é acompanhado de toda a informação necessária para que possa ser atendido.

4. 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 descritos na Tabela 1. A lista completa de serviços do protocolo NFS Versão 2 encontra-se do RFC 1094 (http://tools.ietf.org/html/rfc1094).

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" 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 byte 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 dir filename name	stat status se STAT_OK fhandle file	Cria o ficheiro "name" no directório dado por "dir". 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 filename name	stat status se STAT_OK fhandle file	Cria o novo directório "name" no directório "dir". A resposta STAT_OK indica que o 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.

Serviço	Argumentos	Resultado	Descrição
SNFS_REMOVE	filename name		Remove o ficheiro/directório "name". Só remove directórios se estiverem vazios.
SNFS_OPTIMIZE	filename name unsigned ofs	stat status	Optimiza o ficheiro ou directório "name" (caso o nome não seja vazio). Caso "ofs" (optimize free space) seja 1, optimiza o espaço livre em disco.
SNFS_USAGE		stat status	Imprime um mapa do sistema de ficheiros.

Tabela 2 Serviços do protocolo SNFS

5. API do Cliente

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

- `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_remove(char * name);`

  Remove o ficheiro ou directório indentificado por `name`. Devolve 1 se teve sucesso, 0 caso contrário.

- `void my_optimize(char * name, int optimizeFreeSpace);`

  Optimiza (tornando os seus blocos contíguos), o ficheiro identificado por `name`. Se `optimizeFreeSpace` for 1, deve também tornar contíguos os blocos livres em disco. Neste caso, name pode ser `NULL`, indicando que só se pretende realizar a optimização do espaço livre.

- `void my_diskusage();`

  Imprime, do lado do servidor, um mapa do estado do espaço de blocos do sistema de ficheiros.

- `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 `numFiles` a quantidade dos mesmos ficheiros.

  O espaço de memória onde são escritos os nomes deve ser alocado pela função `my_listdir` e libertado pela função que a chama, assim que não precisar mais dos nomes.

  Da mesma forma que as funções mencionadas anteriormente, estas funções devem devolver um código de erro negativo caso encontrem situações anómalas.