



INSTITUTO SUPERIOR TÉCNICO

Departamento de Engenharia Informática

Enunciado do Projecto Sistemas Operativos

LEIC/LERC
2009/10

Parte I – Biblioteca *sthreads*

Objectivos

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 2.6.8.1 do núcleo do Linux. Pretende-se ainda implementar o problema do “Atendimento no Supermercado” usando a biblioteca *sthreads* com a nova política de escalonamento.

Contexto

Um dos aspectos principais da evolução do escalonamento do Linux foi a tentativa de evitar um algoritmo cujo tempo de execução fosse linear em função do número de tarefas activas. Em termos formais isto significa passar de um funcionamento com complexidade $O(n)$ para um com $O(1)$. Esta optimização foi conseguida através da introdução do conceito de época e da utilização de duas estruturas de dados (multi-listas e vector de prioridades). Este escalonador foi implementado na versão 2.6.8.1 do núcleo do Linux.

Conceito de época

O tempo de execução do sistema divide-se em épocas consecutivas, e cada época termina quando todas as tarefas executáveis tiverem terminado a utilização do respectivo quantum ou prescindido de o fazer. Nem todas as tarefas utilizam o quantum até ao fim pois podem estar bloqueados ou bloquear-se no decorrer da época. O quantum e a prioridade atribuídos a cada tarefa no início de uma época (medido em tiques do relógio) varia consoante o tempo que essa tarefa deixou por utilizar na época anterior. São favorecidas as tarefas que não esgotaram o seu quantum, tendo assim mais quantum na época seguinte.

Note-se que as tarefas E/S intensivas têm primazia neste algoritmo por terem maior hipótese de arrancar a época com um quantum e prioridade maiores.

A escalabilidade deste algoritmo é melhorada em relação ao escalonamento anterior (versões 2.4 e anteriores) pelo facto de as durações de cada quantum serem recalculadas apenas uma vez por época, e a duração da época aumentar à medida que aumenta o número de tarefas em execução simultânea.

Estruturas de dados

A versão 2.6.8.1 do núcleo do Linux recorre a uma gestão multi-lista com prioridades dinâmicas. Existem duas runqueues, uma com as filas das tarefas executáveis da época actual (filas de tarefas activas) e outra com as filas das tarefas executáveis que esgotaram o seu quantum disponível para a época actual (filas de tarefas expiradas). Sempre que uma tarefa esgota o seu quantum é transferida para as filas de expirados, sendo calculados a prioridade e o quantum com que a tarefa iniciará a próxima época.

A época termina quando as filas activas estiverem todas vazias. Nessa altura, comutam-se as duas runqueues passando as filas de expirados para as filas de activos. Ou seja, esta comutação resume-se a uma troca de ponteiros.

Na Figura 1 descreve-se a estrutura de dados. As 100 primeiras posições de cada runqueue são reservadas a tarefas com prioridade fixa (o seu quantum e a sua prioridade não são alteradas) e as restantes 40 posições de cada runqueue são reservadas para as tarefas com prioridade dinâmica cuja prioridade e quantum variam como descrito acima.

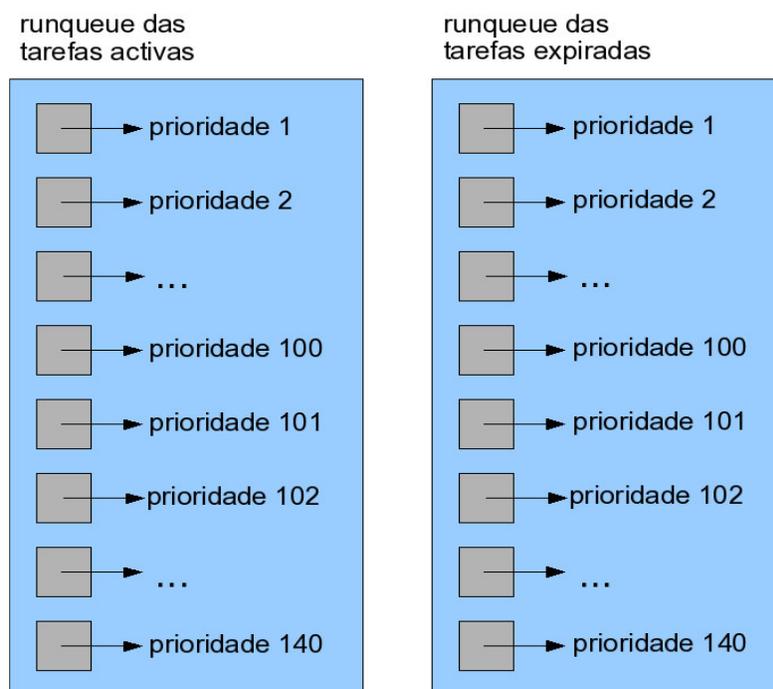


Figura 1: Runqueues na versão 2.6.8.1 do núcleo do Linux

Implementação do escalonamento

No trabalho pretende-se a implementação de um escalonamento do tipo Linux 2.6.8.1 usando a biblioteca sthreads, simplificando o número de níveis de prioridades mas mantendo os algoritmos. Terá que implementar as seguintes estruturas de dados:

2 runqueues:

- uma runqueue com as filas das tarefas activas para cada valor de prioridade;
- uma runqueue com as filas das tarefas expiradas para cada valor de prioridade;

Cada runqueue tem 15 posições correspondentes a 5 prioridades estáticas e 10 prioridades dinâmicas.

Conceptualmente, cada runqueue tem, para cada nível de prioridade, uma fila de tarefas. A escolha da estrutura de dados para implementar as filas fica ao critério dos alunos, devendo ser dado ênfase ao bom desempenho da biblioteca.

As prioridades seguem o modelo do inicial do UNIX um valor menor representa uma prioridade mais elevas (0 é o nível de maior prioridade e 14 o menor).

O cálculo do quantum e da prioridade de uma tarefa, com prioridade dinâmica, na próxima época é dado por:

Quantum Inicial = Quantum Base + Quantum por Usar / 2;

Em que Quantum por Usar representa o valor do quantum que a tarefa não utilizou na época anterior, e Quantum Base representa um valor atribuído pelo escalonador que depende da frequência das interrupções do relógio.

Prioridade = Prioridade Base - Quantum por Usar + Nice;

Em que Quantum por Usar representa o quantum ainda disponível para a tarefa nesta época, a Prioridade Base é dada aquando da criação da tarefa e o Nice é um valor que permite dar prioridade a outras tarefas ao diminuir a da tarefa que tem um valor de Nice maior.

Considere que Quantum Base é igual a 5.

Existem dois tipos de tarefas:

- Tarefas com prioridade fixa. Estas tarefas têm uma prioridade entre 0 e 4. A sua prioridade e o seu quantum nunca são alterados;
- Tarefas com prioridade dinâmica. Estas tarefas têm uma prioridade entre 5 e 14. A sua prioridade e o seu quantum são alterados de acordo com as expressões anteriores respeitando os limites de prioridade.

O procedimento do escalonador/despacho é o seguinte:

1. Se não existem tarefas nas filas de tarefas activas, trocar as duas runqueues;
2. Retirar da runqueue de tarefas activas a tarefa com prioridade mais elevada:
 1. Verificar qual a fila correspondente a um valor menor de prioridade que é não vazia;
 2. Dessa fila retirar o 1º elemento;
3. Colocar a tarefa em execução;
4. Quando a tarefa acabar a sua execução por o seu quantum ter sido excedido:
 1. Se for uma tarefa de prioridade dinâmica calcular a sua nova prioridade e o seu novo quantum;
 2. Colocá-la na posição correspondente à sua prioridade na runqueue de tarefas expiradas;
 3. o procedimento do escalonador/despacho é iniciado.

5. Quando a tarefa se bloquear:

1. O seu novo quantum e a sua nova prioridade são calculados;
2. A tarefa é colocada na fila de tarefas bloqueadas do objecto de sincronização onde se bloqueou;
3. o procedimento é iniciado.

Quando uma tarefa se desbloqueia ela é sempre inserida na runqueue de tarefas activas na fila correspondente à nova prioridade.

O escalonador tem preempção. Ou seja, se a tarefa que se desbloqueou tem maior prioridade (valor de prioridade mais baixo) do que a tarefa que se estava a executar, então ela deve-se executar imediatamente e a tarefa que se estava a executar deve continuar na runqueue de tarefas activas. De notar que não é levado em conta se a época em que a tarefa se desbloqueia é a mesma ou não em que a tarefa se bloqueou.

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);`
- 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 época caso esgote o seu quantum. O valor do `nice` pode estar entre 0 e 10. Note que a alteração do valor da prioridade apenas será reflectida na próxima época;
- Altere as funções correspondentes à implementação dos monitores para que os monitores 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, expiradas 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>
quantum: <quantum ainda disponível>
active runqueue
[0] <id,quantum das tarefas que têm prioridade 0 e estão na lista das activas separados por espaço>
[1] <id,quantum das tarefas que têm prioridade 1 e estão na lista das activas separados por espaço>
[2] <id,quantum das tarefas que têm prioridade 2 e estão na lista das activas separados por
```

```

espaço>
...
[14] <id,quantum das tarefas que têm prioridade 14 e estão na lista das activas separados
por espaço>
expired runqueue
[0] <id,quantum das tarefas que têm prioridade 0 e estão na lista das expiradas separados
por espaço>
[1] <id,quantum das tarefas que têm prioridade 1 e estão na lista das expiradas separados
por espaço>
[2] <id,quantum das tarefas que têm prioridade 2 e estão na lista das expiradas separados
por espaço>
...
[14] <id,quantum das tarefas que têm prioridade 14 e estão na lista das expiradas separados
por espaço>
blocked list
<id das tarefas que estão bloqueadas separados por espaço>
=== dump end ===

```

Exemplo de um output possível do `sthread_dump`:

```

=== dump start ===
active thread
id: 4
priority: 8
quantum: 2
active runqueue
[0]
[1]
[2]
[3]
[4]
[5]
[6]
[7]
[8] 1,6
[9]
[10]
[11] 3,5 8,5
[12]
[13] 7,5
[14]
expired runqueue
[0] 0,5
[1]
[2]
[3]
[4]
[5] 2,5 5,5
[6]
[7] 9,5
[8] 3,5
[9]
[10]
[11]
[12]
[13]
[14]
blocked list
6,6 8,7
=== dump end ===

```

No exemplo anterior, temos o seguinte:

- A tarefa activa é a com identificador 4, a sua prioridade actual é 8 e o seu quantum por usar é 2;
- Na runqueue das tarefas activas temos:
 - uma tarefa com prioridade 8 que tem como identificador o valor 1 e quantum o valor 6;
 - uma tarefa com prioridade 11 que tem como identificador o valor 3 e quantum o valor 5;
 - uma tarefa com prioridade 11 que tem como identificador o valor 8 e quantum o valor 5;
 - uma tarefa com prioridade 13 que tem como identificador o valor 7 e quantum o valor 5;
- Na runqueue das tarefas expiradas temos:
 - uma tarefa com prioridade 0 que tem como identificador o valor 0 e quantum o valor 5;
 - uma tarefa com prioridade 5 que tem como identificador o valor 2 e quantum o valor 5;
 - uma tarefa com prioridade 5 que tem como identificador o valor 5 e quantum o valor 5;
 - uma tarefa com prioridade 7 que tem como identificador o valor 9 e quantum o valor 5;
 - uma tarefa com prioridade 8 que tem como identificador o valor 3 e quantum o valor 5;
- Nas listas de tarefas bloqueadas temos:
 - uma tarefa que tem como identificador o valor 6 e quantum o valor 6;
 - uma tarefa que tem como identificador o valor 8 e quantum o valor 7;Aplicação do mecanismos de sincronização monitores

Nesta parte do trabalho deve exercitar a programação concorrente implementando o exemplo do Supermercado que foi dado nas aulas práticas.

Problema do Supermercado

O problema do Supermercado consiste em admitir que existe um supermercado com N caixas de pagamento e com um funcionário em cada caixa.

A tarefa de atendimento consiste no seguinte:

- Enquanto houver clientes na sua fila o funcionário atende-os;
- Se não estiver nenhum cliente para ser atendido na sua fila, o funcionário pode atender um cliente da outra fila;
- Se não existir ninguém para atender em nenhuma das filas, o funcionário bloqueia-se à espera de clientes.

A tarefa do cliente consiste no seguinte:

- Quando o cliente chega, espera na fila que tiver menos clientes;
- Uma vez escolhida a fila, o cliente não pode trocar, excepto para ser atendido conforme descrito anteriormente no procedimento da tarefa de atendimento;
- O número de clientes por fila é ilimitado.

Mecanismos de sincronização

Foram fornecidos os mecanismos de sincronização monitor com a biblioteca pthreads:

- A estrutura para declarar um monitor (struct pthread_monitor_t);
- As operações dos monitores
 - pthread_user_monitor_init();
 - pthread_user_monitor_enter();
 - pthread_user_monitor_exit();
 - pthread_user_monitor_wait();
 - pthread_user_monitor_signal().

Os monitores são explicitamente delimitados nos programas pelas primitivas pthread_user_monitor_enter e pthread_user_monitor_exit que garantem que o bloco de código assim definido é executado em secção crítica. No interior do monitor, as tarefas podem sincronizar-se explicitamente através de uma fila de sincronização.

A primitiva pthread_user_monitor_wait (que só pode ser invocada dentro do monitor) permite a uma tarefa bloquear-se numa fila de sincronização libertando o monitor e a correspondente secção crítica.

A primitiva pthread_user_monitor_signal (que também só pode ser invocada dentro de um monitor) irá desbloquear uma das tarefas que estejam bloqueadas nesse monitor se existir alguma, caso contrário, não tem qualquer efeito. As filas de sincronização não são semáforos: a primitiva pthread_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 pthread_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

Implementação com base em monitores

Os mecanismos de sincronização fornecidos com a biblioteca pthreads estão implementados para um escalonamento com uma política round-robin. Como referido anteriormente, pretende-se que modifique a biblioteca por forma a que as funções que operam sobre monitores funcionem com a implementação do novo escalonamento.

Com base nessas funções programe da maneira mais eficiente o problema do Supermercado. Considere que existem duas rotinas – Atender() e SerAtendido() – que são chamadas respectivamente quando o empregado está a atender o cliente e quando o cliente está a ser atendido pelo empregado. Estas rotinas devem bloquear as tarefas durante um intervalo de tempo enquanto o cliente está a ser atendido e o empregado está a atender. Depois do cliente ser atendido, deve ainda esperar algum tempo antes de ser novamente atendido. Em pseudo-código temos:

```
Cliente (int idCliente){
    EscolherFila ( TempoAtendimento)
    SerAtendido(TempoAtendimento);
}
}
```

```

Empregado (int fila) {
    while (TRUE){
        TempoAtendimento = ProximoCliente(fila);
        Atender(TempoAtendimento);
    }
}

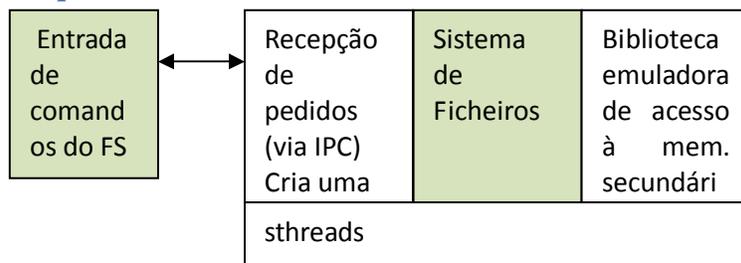
```

Parte II – Sistema de Ficheiros

Objectivos

Melhorar o desenho e implementação de um sistema de ficheiros já existente. A versão actual do sistema de ficheiros é extremamente limitada: não suporta directorias, utiliza o espaço em disco de uma forma pouco eficiente e tem fraco desempenho por várias razões. Pretende-se incorporar no sistema de ficheiros um número de melhoramentos que colmatem as limitações referidas.

Arquitectura Base



A arquitectura do sistema de ficheiros é a apresentada na figura acima. O acesso à memória secundária é feito através de uma biblioteca que permite ler e escrever blocos. O sistema de ficheiros corre num processo *daemon*, que utiliza a biblioteca *stthreads* (a mesma implementada na primeira parte do projecto) para servir múltiplos pedidos em paralelo. Os pedidos ao sistema de ficheiros são enviados e respondidos através de um canal de comunicação com um processo que é responsável por interagir com os clientes do sistema de ficheiros.

Todos estes componentes são disponibilizados à partida. Os componentes que deverão ser modificados estão indicados a sombreado.

Requisitos

Os melhoramentos a incorporar são os seguintes:

- Suporte a directorias multi-nível;
- Caching;
- Mecanismo de *copy-on-write*, que permita poupar espaço quando existem múltiplas cópias do mesmo ficheiro original no disco;
- Sincronização eficiente nas rotinas do sistema de ficheiros, para maior paralelismo;
- Suporte a replicação de discos, para melhor desempenho (pedidos enviados a múltiplos sistemas de ficheiros replicados, sendo que a resposta ao cliente é retornada assim que o sistema de ficheiros mais rápido a retornar).

A parte II só deverá ser resolvida após a parte I estar completada. Um enunciado detalhado da parte II será publicado em breve.

Entrega e Avaliação

A entrega do trabalho será feita em duas fases: uma entrega intercalar e a entrega final.

Entrega intercalar

Data da entrega intercalar: 9 de Novembro às 12:00

Para a entrega intercalar, **espera-se que os alunos entreguem uma implementação completa da Parte I do projecto.**

A entrega é por via electrónica através da página WWW da cadeira. O formato do ficheiro de entrega com o código será indicado oportunamente.

A versão intercalar que cada grupo entregue será avaliada **na aula prática de cada grupo, na semana seguinte à entrega.**

A entrega intercalar pode contar para subir a nota de acordo com a seguinte fórmula:

Nota final do projecto = MAX (Nota original do projecto; $0.9 \times$ Nota original do projecto + $0.1 \times$ Nota da entrega intercalar),

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

Data da entrega final: 9 de Dezembro às 12:00

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

1 - Via electrónica

A entrega por via electrónica será efectuada no dia 9 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 14 a 18 de Dezembro e de 4 a 6 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

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
<i>README</i>	Informação detalhada de como fazer um programa que usa as <i>sthreads</i> (<i>hello_sthreads</i>).
<i>thread_lib/</i>	Biblioteca de tarefas
<i>thread_lib/thread_user.c</i>	Onde vão ser implementadas as funções necessárias para a biblioteca de tarefas.
<i>thread_lib /thread_ctx.{c,h}</i>	Módulo para criar novas pilhas de execução e para comutar entre elas
<i>thread_lib /thread_switch_i386.h</i>	Funções <i>assembly</i> para comutar entre pilhas e para salvar registos
<i>thread_lib/thread_time_slice.{c,h}</i>	Suporte para gerar <i>signals</i> e para controlá-los
<i>include/</i>	Contém <i>thread.h</i> e <i>config.h</i> que são as interfaces públicas, a incluir em programas que usem ou comuniquem com os programas ou bibliotecas referidas.
<i>test-sthreads/</i>	Contém vários testes para a biblioteca <i>sthreads</i>

- Tabela 1 Ficheiros fornecidos no início do projecto

As rotinas no ficheiro *thread_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 *thread_user*. Durante a implementação modificar apenas o ficheiro *thread_user.c*. Não modificar *thread_ctx_t* directamente, usando em vez disso as rotinas declaradas em *thread_ctx.h*. Considerando o sistema em camadas (Figura 1) apenas tem que implementar o “rectângulo a cinzento”.

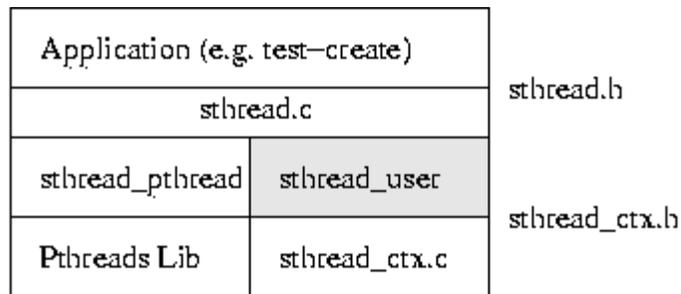


Figura 1

Na camada superior encontra-se a aplicação que vai usar o pacote *threads* (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 *pthread*s (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 *threads*

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 `pthread_t` para manter a informação do seu contexto de execução.
- Tenha cuidado com o uso de variáveis locais após chamar `pthread_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 `threads`, `pthread_user_init`, inicia o escalonador de tempo partilhado invocando `pthread_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 `pthread_switch`. De notar que `pthread_switch` pode retornar para dois locais diferentes: para a linha a seguir a uma chamada a um `pthread_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 `pthread_time_slices_init()`.