INSTITUTO SUPERIOR TÉCNICO
Departamento de Engenharia Informática

Enunciado do Projecto de
Sistemas Operativos – Parte I
LEIC/LERC 2011-2012

Introdução

A primeira parte do projecto de SO aborda a programação de aplicações paralelas usando o modelo de tarefas, sob dois pontos de vista: i) ponto de vista do programador da aplicação; e ii) ponto de vista do escalonador.
Esta parte divide-se, pois, em dois problemas a resolver (I.A e I.B), que devem ser desenvolvidos em paralelo. É esperado que, no final desta parte , as soluções dos dois problemas funcionem correctamente de forma integrada.

Calendário proposto

- 12/10 a 21/10 (1 semana e meia):
  - Estudar o problema I.A e desenvolver solução em pseudo-código;
- 24/10 a 28/10 (1 semana)
  - Implementar e testar solução de I.A sobre pthreads;
  - Em paralelo, tomar primeiro contacto com biblioteca *sthreads* (problema I.B) e implementar o suporte a semáforos;
- 31/10 a 11/11 (2 semanas)
  - Testar solução do problema I.A sobre sthreads.
  - Desenhar e implementar novo algoritmo de escalonamento (I.B);
  - Integrar ambas as soluções e testar.

Os detalhes de cada passo serão explicados de seguida.

Entrega e Avaliação

Entrega intercalar (*Checkpoint*): 11 de Novembro às 23:59

Para a entrega intercalar, os alunos devem entregar uma implementação completa dos problemas I.A e I.B da 1ª parte 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: 2 de Dezembro às 23:59

A entrega final consiste na submissão das Partes I e II (o enunciado da parte II será publicado até à data do checkpoint). Os detalhes sobre a entrega final serão publicados no 2º enunciado.
A informação sobre a entrega e avaliação aqui descrita está sujeita a alterações, que serão afixadas na página da cadeira.

Problema I.A – Leitores/escritores baseados em logs

Problema

Considere um sistema de tarefas *trabalhadoras* partilham um conjunto de *registos* partilhados. Durante a sua execução cada tarefa trabalhadora pode pedir para ler ou escrever integralmente o conteúdo de um dos registos partilhados. Assuma que o número de registos é fixo e dado por *num_reg,* sendo a dimensão de cada registo dada por *bloco_dim*.

Na prática, os registos são mantidos em dois arrays de *blocos* (em que cada bloco tem dimensão igual à de um registo, *bloco_dim*). A cada momento, um dos arrays é designado por array *activo* e o outro por array *inactivo*. Ambos têm *array_dim* blocos. Em cada momento, o conteúdo *actual* de cada registo está guardado num dado bloco de um dos arrays. Inicialmente, o registo 0 tem o seu conteúdo actual no bloco 0 do array activo, o registo 1 no bloco 1 do array activo, etc.

Sempre que um trabalhador solicita uma escrita a um dado registo, o bloco que contém o conteúdo actual do registo não é modificado. Em vez disso, é reservado o próximo bloco livre[1] no actual array activo e o novo conteúdo é escrito no novo bloco. Desta forma, a localização actual de cada registo muda à medida que o mesmo sofre escritas.
Como o bloco onde se encontra o conteúdo actual de um dado registo é dinâmico, é necessário manter uma *tabela de índices* que indica, para cada registo, qual o array e o índice nesse array onde se encontra o bloco com o conteúdo actual do registo. A tabela de índices é consultada por cada leitor antes de obter o bloco desejado e é modificada por cada escritor para reflectir a novo bloco actual do registo em causa.

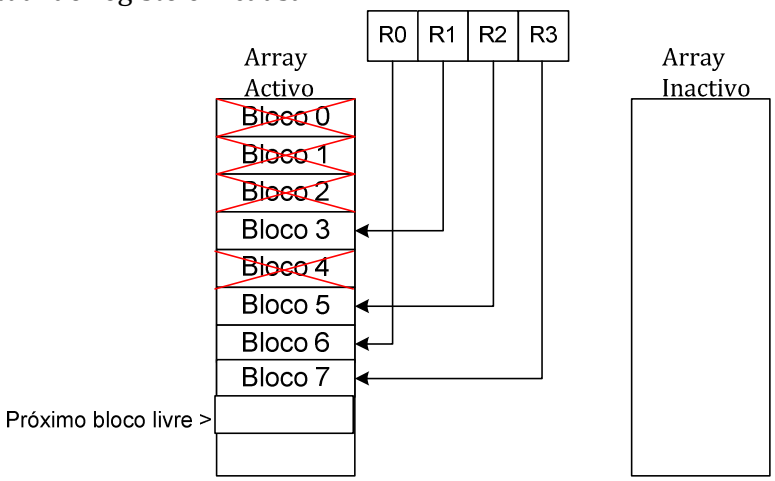

Fig. 1: Exemplo de estado possível dos arrays e do bloco de índices. Se neste momento houver uma leitura ao registo R3, o conteúdo será lido a partir do bloco 7 do array activo. Em paralelo, uma escrita de R3 será colocada no próximo bloco livre do array activo.

Esta solução introduz um desafio importante: à medida que as escritas ocorrem, são consumidos blocos, podendo o array actualmente activo esgotar ao fim de algumas escritas. Quando esse momento acontecer (ou seja, quando o último bloco livre no array activo é ocupado), ambos os arrays trocam de papéis: o array inactivo passa a ser activo, e vice-versa; a este momento chamamos o *flip* dos arrays. Qualquer escrita que ocorra após o flip utilizará um

[1] Isto é, se o último bloco alocado foi na posição j do array, o seguinte será alocado na posição j+1 do mesmo array, e assim sucessivamente.

bloco livre no novo array activo. Sempre que ocorre um *flip*, todos os blocos actuais (i.e., com conteúdo actual de um registo) do anterior array activo deverão ser copiados para blocos (livres) do novo array activo. Note-se que, idealmente, a cópia de blocos actuais do array inactivo para o activo não deverá impedir o progresso de leituras e escritas que sejam solicitadas após o flip (mesmo que a cópia ainda esteja em curso).

Desenho e implementação da solução

Deverá ser desenvolvida uma solução para o problema descrito acima que permita o maior paralelismo possível entre as várias tarefas. Mais precisamente, a solução a desenvolver deverá minimizar as situações em que uma leitura ou escrita seja atrasada/adiada devido a exigências de sincronização.

A solução deverá oferecer a API documentada no ficheiro *include/logrw.h* que é fornecido no pacote base, no site da cadeira. Projectos que não cumpram integralmente a API documentada nesse ficheiro não serão avaliados.

A solução deverá ser implementada usando a API de programação concorrente oferecida pela biblioteca *sthreads* (descrita em I.B), que inclui: suporte para tarefas, trincos lógicos, semáforos e trincos de leitura/escrita. Como tal, é permitido recorrer a todas estas primitivas na solução a desenvolver.

Enquanto as *CFQ-sthreads* não estiver completamente e correctamente implementada de acordo com o especificado no problema I.B, recomenda-se que os alunos usem a opção *USE_PTHREADS* na compilação da biblioteca *sthreads*. Esta opção permite que as solução para o problema I.A possam ser testadas sobre *pthreads*, mas mantendo o uso da API das *sthreads*. Desta forma, assim que o problema I.B esteja resolvido, a integração de ambas as soluções deverá ser directa.

Problema I.B – Escalonador CFQ-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*). Ou seja, a *sthreads* trata todas as tarefas do processo de igual forma.

Entre outros usos, a biblioteca *sthreads* permite que um processo *servidor* que receba concorrentemente pedidos vindos de diferentes *clientes* possa, para cada pedido, lançar uma tarefa para o servir. Por exemplo, os clientes podem ser outros processos, cada um a interagir com um utilizador diferente do serviço oferecido pelo processo servidor. Neste cenário, cada tarefa criada no servidor estará, na verdade, a executar trabalho em nome de um dado cliente.

Sendo o processador onde o servidor se executa um recurso limitado, é desejável que o escalonador imponha alguma justiça no modo como distribui o tempo de execução entre as várias tarefas (a correr em nome dos vários clientes). No entanto, se um dado cliente enviar vários pedidos em paralelo para o servidor, esse cliente terá, simultaneamente, muitas tarefas a executar-se em seu benefício. É fácil perceber que, com o escalonador actual das sthreads, um cliente que lance muitos pedidos simultaneamente irá perverter a justiça oferecida pelo

escalonamento seguido pelas *sthreads* pois, clientes com menos tarefas, verão o servidor dedicar-lhes menos tempo de execução.

Pretende-se modificar o escalonamento das *sthreads* por forma a assegurar uma distribuição mais justa do tempo de execução do servidor entre os vários clientes. Mais precisamente, pretende-se substituir a política actual de escalonamento por outra que tenha em conta a relação tarefa-cliente. Para isso, propõe-se o uso de uma variante simplificada do algoritmo de escalonamento *Completely Fair Queuing* (CFQ) chamada *CFQ-sthreads*[2]

Algoritmo

No início da execução do processo servidor, existe apenas uma tarefa, que executa a função main em nome do cliente 0 (cliente fictício). A qualquer momento, novas tarefas podem ser criadas em nome de qualquer cliente (identificado por um inteiro não negativo) no processo servidor.

O CFQ-sthreads deverá manter estado que lhe permita saber quais as tarefas actualmente activas (i.e., tarefa que foi criada e ainda não terminou) e, para cada uma dessas tarefas, a seguinte informação:

- Identificador do cliente em nome do qual a tarefa foi lançada;
- Identificador da tarefa;
- Tempo de execução acumulado pela tarefa (*vruntime*), calculado de acordo com a fórmula apresentada mais à frente;
- Valores *prio* e *nice* da tarefa (ver mais abaixo);
- Estado da tarefa (bloqueado, executável, em execução).

O CFQ-sthreads mantém também informação sobre quais os clientes que têm pelo menos uma tarefa activa. Para cada cliente, o CFQ-sthreads sabe qual o seu identificador (do tipo inteiro) e qual o número actual de tarefas activas desse cliente.

O escalonador executa-se periodicamente, interrompendo a execução da tarefa actualmente em execução. Ao momento em que o escalonador interrompe a execução chamamos um *tick*, sendo o período entre ticks parametrizável.

Cada tarefa tem também associada um valor *prio*, que determina a prioridade da tarefa e é especificado aquando da sua criação; e um valor *nice*, que permite dinamicamente compensar essa mesma prioridade. O valor de *prio* varia entre 1 (mais prioritária) e 10 (menos prioritária). O valor de *nice* de uma tarefa é inicialmente 0. Cada chamada à função *nice* pode modificá-lo para um valor entre 0 e 10.

De cada vez que o escalonador se executa (ou seja, quando ocorre um tick), adiciona ao *vruntime* da tarefa que se esteve a executar o valor dado por:

$$(prio+nice)* num\_tarefas\_deste\_cliente$$

Como se explicará em breve, o CBQ-sthreads seguirá o princípio de atribuir o CPU à tarefa que, em cada momento, tenha o menor vruntime. Pela expressão acima, é fácil concluir que o CBQ-

[2]Mera curiosidade: o CFQ é o algoritmo de escalonamento que, a partir da versão 2.6.6, é usado no núcleo do Linux para escalonar o acesso a I/O.

sthreads dará maior tempo de execução às tarefas de maior prioridade (valor menor de *prio*). Pela mesma expressão também se deduz que clientes com muitas tarefas simultaneamente activas não irão monopolizar a execução do processador, já que cada uma dessas tarefas será penalizada devido a esse factor, desta forma promovendo uma maior justiça entre clientes.

Quando uma dada tarefa é colocada em excução, essa tarefa executa-se, no mínimo, um determinado número de ticks (definido por *min_delay*) de modo a ter oportunidade de executar algum trabalho útil.

Sempre que o valor do vruntime de uma tarefa é actualizado, o seu novo valor de *vruntime* pode passar a ser maior que o valor de vruntime de outras tarefas executáveis. Nesse caso, caso a tarefa em execução já se tenha executado durante o tempo mínimo de ticks (min-delay), o CPU é atribuído à tarefa com menor *vruntime*.

O escalonador deve suportar preempção. Isto significa que a tarefa em execução pode perder o processador, o que poderá ocorrer no final de um qualquer tick , assim como noutros momentos em que possa surgir uma tarefa executável com vruntime inferior ao da tarefa actualmente em execução (e.g. na sequência de um assinalar efectuado sobre um semáforo). Esta preempção só deverá ocorrer caso a tarefa em execução já se tenha executado durante pelo menos min_delay ticks.

Caso uma tarefa perca o processador por preempção, o seu novo valor de *vruntime* deve contemplar apenas o número de ticks totalmente consumidos.

A biblioteca *CBQ-sthreads* deverá oferecer suporte a trincos lógicos, semáforos e trincos de leitura-escrita, sendo que parte desse suporte já se encontra pronto nas *sthreads* (ver abaixo).

Notas sobre a Implementação do Escalonador

A biblioteca base (sthreads) está disponível a partir do site da cadeira e já inclui:

- Escalonador sthreads (round-robin, sem noção de prioridade nem cliente) implementado e pronto a usar;
- Suporte a trincos lógicos e trincos de leitura-escrita implementado.

Caberá aos alunos estender a biblioteca base de forma a:

- Usar o novo algoritmo de escalonamento, que suporta prioridades dinâmicas e clientes;
- Implementar semáforos (que estão por implementar na *sthreads*)

Naturalmente, cabe aos alunos decidir quais as estruturas de dados mais adequadas para manter o estado relativo às tarefas e clientes geridos pelo *CBQ-sthreads*.
Um aspecto importante é o facto da variável *vruntime* crescer indefinidamente, o que pode provocar *overflows*. Tenha este aspecto em consideração.

A API da biblioteca base serve também para as CBQ-sthreads e não poderá ser alterada (projectos que modifiquem a API não passarão na avaliação).

É também obrigatório implementar uma função cujo protótipo é *void sthread_dump().* Esta função imprime o estado tarefas activas no momento em que é chamada. O output terá obrigatoriamente de respeitar o seguinte formato:

```
=== dump start ===
```

```
Active thread
id: <id do cliente onde foi chamado sthread_dump()> request: <identificador da sthread
correspondente ao pedido>: priority: <prioridade da tarefa + nice> vruntime: <valor da variável
vruntime>
runtime: <tempo de execução real> on state for: <há quanto tempo está no estado actual>
sleeptime: <tempo total em que esteve bloqueda> waittime: <tempo total de espera no
escalonador>

>>>> Scheduler <<<<
id: <id do cliente no escalonador ordenada por cliente e vruntime>: request: <identificador da
sthread>: priority: <prioridade da tarefa + nice> vruntime: <valor da variável vruntime>
runtime: <tempo de execução real>: on state for: <há quanto tempo está no estado actual>:
sleeptime: <tempo total em que esteve bloqueda> waittime: <tempo total de espera no
escalonador>

>>>>BlockedList<<<<
id: <id do cliente em sleep ordenado por ordem de desbloqueio>: request: <identificador da
sthread>: priority: <prioridade da tarefa + nice> vruntime: <valor da variável vruntime>
runtime: <tempo de execução real>: on state for: <há quanto tempo está no estado actual>:
sleeptime: <tempo total em que esteve bloqueda>: waittime: <tempo total de espera no
escalonador>

>>>>SleepList<<<<
id: <id dos clientes em sleep ordenada por tempo de desbloqueio>: request: <identificador da
sthread>: priority: <prioridade da tarefa + nice> vruntime: <valor da variável vruntime>
runtime: <tempo de execução real> on state for: <há quanto tempo está no estado actual>
sleeptime: <tempo total em que esteve bloqueda> waittime: <tempo total de espera no
escalonador>: Waketime: <Valor de relógio em que a sthread passará para executável>
==== Dump End ====
```

Para mais detalhes, ver o Anexo A.

Anexo A – Pacote simplethreads (sthreads)

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".

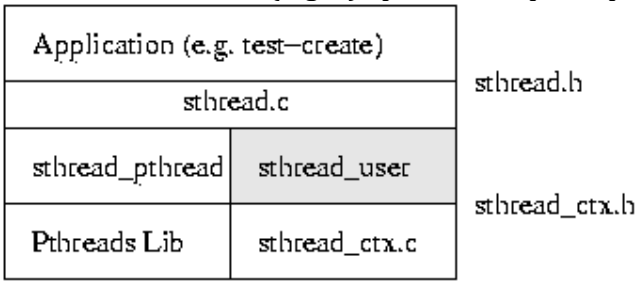

Fig. 2

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 uma 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, cuja função de tratamento inclui o algoritmo de despacho.

Quanto aos semáforos:

- Com base no código já disponibilizado para os trincos lógicos, 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, apenas deverá ocorrer comutação de tarefa se a tarefa desbloqueada tiver menor *vruntime* que a tarefa actualmente em execução. Caso contrário, a tarefa desbloqueada 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;
- O pacote base disponibilizado inclui um conjunto de testes que deverão funcionar correctamente na versão base. Sempre que fizer alterações, corra de novo os testes para confirmar que continuam a correr correctamente.
- 10 milisegundos é um bom valor para o período entre ticks.