INSTITUTO SUPERIOR TÉCNICO
Departamento de Engenharia Informática
Enunciado do Projecto de Sistemas Operativos – Parte I
LEIC/LERC 2012-2013

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, entrega e avaliação

Calendário proposto

- 12/10 a 19/10 (1 semana e meia):
  - Estudar o problema I.A e desenvolver solução em pseudo-código;
- 20/10 a 26/10 (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;
- 27/10 a 15/11 (2 semanas e meia)
  - Testar solução do problema I.A sobre sthreads.
  - Desenhar e implementar novo algoritmo de escalonamento (I.B);
  - Integrar ambas as soluções e testar.

Entrega da Parte I: 15 de Novembro às 23:59

Os alunos devem entregar uma implementação completa dos problemas I.A e I.B da da Parte I do projecto por via electrónica através do sistema Fénix. O formato do ficheiro de entrega com o código será indicado oportunamente.

Avaliação

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

A avaliação da Parte I vale **60%** da nota final do projecto (enquanto que a Parte II contará 40%).

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

Problema I.A – Sistema de reservas de cinema

Problema

Pretende-se implementar um sistema para gerir reservas de lugares de uma sala de cinema. Cada lugar é caracterizado pelas suas coordenadas (nomeadamente, número de linha e número de cadeira nesta linha) e um estado que pode tomar um dos seguintes três valores: *disponível* (codificado pelo valor 0), *reservado* (codificado pelo valor 1), e *reservado temporariamente* (codificado pelo valor 2).

O sistema de reservas corre num processo único constituído por múltiplas tarefas. Entre as tarefas distinguem-se tarefas *servidoras* e *clientes.*

As tarefas servidoras gerem o estado das reservas e servem pedidos gerados por tarefas clientes (Figura 1).

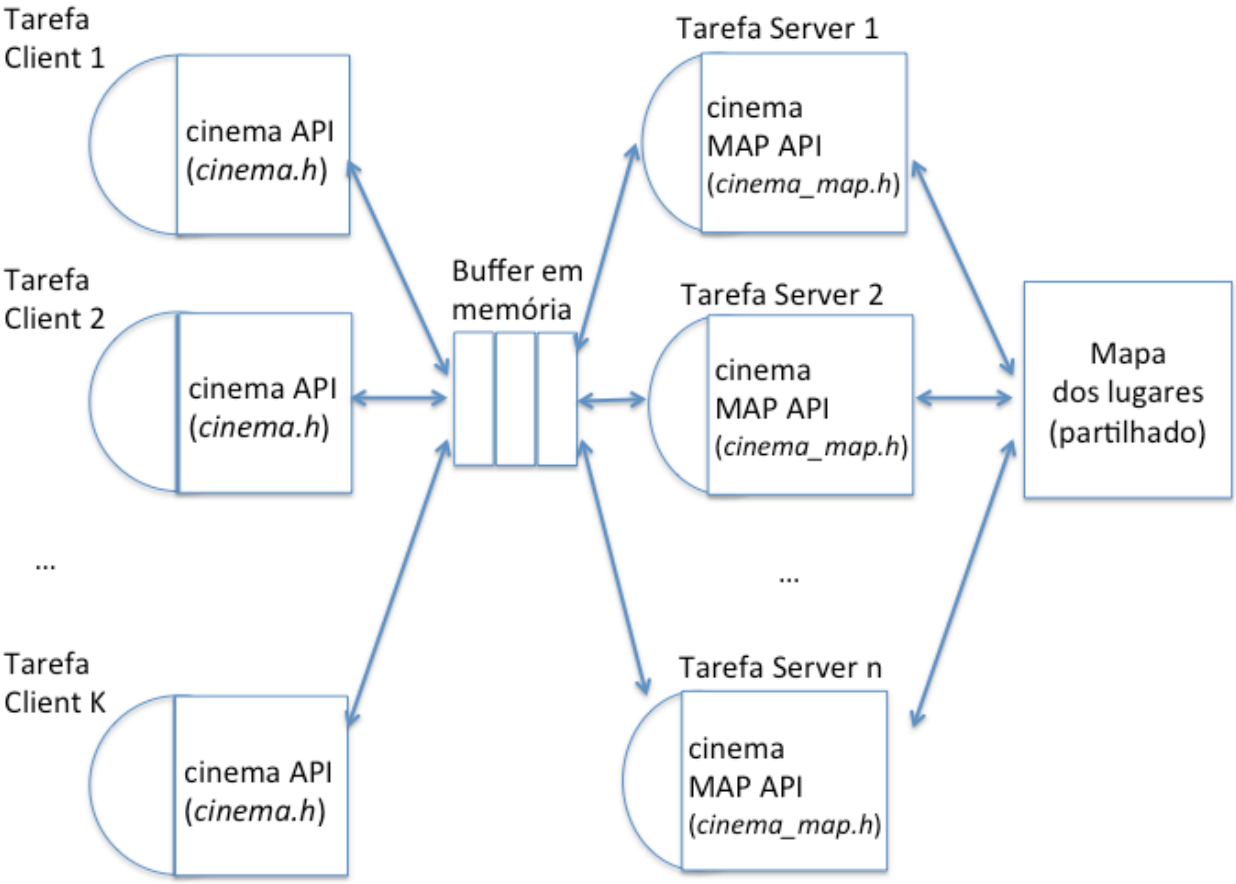

Figura 1. Arquitectura geral da solução proposta

As tarefas clientes não acedem diretamente ao estado das reservas. Em vez disso, estas tarefas devem interagir com as tarefas *server* usando as funções especificadas no ficheiro `cinema.h`, as quais suportam as seguintes funcionalidades:

- **inicialização das tarefas servidoras:** esta funcionalidade é oferecida pela função `int inicializa_servidor(int num_servers)`. Esta função toma em input o numero de tarefas servidoras que deverão ser utilizadas e devolve:
  - 0, se a inicialização foi bem sucedida;

    - -1, se houve algum erro na fase de inicialização ou se as tarefas servidoras já tiverem sido inicializadas.
- **reserva de lugares**: esta funcionalidade é oferecida pela função `int reserva(Coords_t* lugares, int dim)`. Esta função reserva os lugares desejados e devolve um inteiro maior do que 0 que identifica univocamente a reserva. Caso não seja possível reservar os lugares pretendidos (por exemplo, porque alguns deles já foram reservados), nenhum dos lugares pretendidos deve ser reservado e a função devolve -1.
- **pré-reservas temporárias**: esta funcionalidade é oferecida pela função `int prereserva(Coords_t* lugares, int dim, int segundos)`. Esta função reserva os lugares durante o número de segundos passado como *input*, e devolve um inteiro maior do que 0 que identifica univocamente a pre-reserva. Todas as pre-reservas que não sejam confirmadas a tempo deverão ser canceladas automaticamente por uma das tarefas *server*. Esta função devolve -1 caso não seja possível reservar os lugares pretendidos (por exemplo, porque alguns deles já foram reservados); nesse caso, nenhum dos lugares pretendidos deve ser pre-reservado.
- **confirmação de pre-reservas temporárias**: esta funcionalidade é oferecida pela função `int confirma_prereserva(int id)`. Esta função devolve -1 caso não exista uma pre-reserva válida com o id especificado em *input* (por exemplo porque a reserva expirou). Em caso contrário, devolve o valor 0.
- **cancelamento de reservas (ou pre-reservas temporárias)**: esta funcionalidade deverá ser oferecida pela função `int cancela(int id)`. Esta função altera o estado dos lugares associados à reserva ou pre-reserva para *disponível* e devolve o valor 0. Devolve -1 caso não exista nem uma reserva, nem uma pre-reserva válida com o id especificado em *input* (por exemplo porque a reserva expirou).
- **mostrar o estado dos lugares, ou seja o mapa das (pre)reservas do cinema**: esta funcionalidade deverá ser oferecida através das seguintes duas funções:
    - `void mostra_cinema()`: esta função encontra-se implementada no ficheiro *cinema/show.c* . Esta função não implementa nenhum esquema de sincronização, e pode portanto produzir um *output* que reflecte a execução parcial dos pedidos de (pre)reserva de lugares: por exemplo, no caso duma reserva para 10 lugares, poderá mostrar como reservados só 2 dos 10 lugares.

Desenho e implementação da solução

O desenho do mecanismo de sincronização para o problema descrito acima é deixado aos alunos, mas só serão consideradas válidas soluções em que as tarefas *clients* e *servers* comuniquem por um *buffer* circular em memória. **Soluções que utilizem outras alternativas para a comunicação entre tarefas *clients* e *servers* (por exemplo ficheiros, *sockets*, etc.) não serão avaliados.**

A solução desenvolvida deverá permitir o maior paralelismo possível entre as várias tarefas. Mais precisamente, a solução a desenvolver deverá minimizar as situações em que o processamento dos pedidos dos clientes seja atrasado/adiado devido a exigências de sincronização. A solução desenvolvida deverá também maximizar a eficiência das tarefas servidoras, as quais deverão ficar bloqueadas se não houver pedidos para processar.

Soluções que utilizem esquemas de sincronização triviais (por exemplo, um único trinco lógico para serializar a execução das tarefas servidores) serão fortemente penalizadas.

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

O estado de cada lugar do cinema deve ser manipulado (lido,escrito) exclusivamente pelas funções presentes no ficheiro *include/cinema_map.h*, cuja implementação é também fornecida no pacote base. Estes funções permitem simular atrasos devidos a acessos a repositórios externos (por exemplo um disco, ou um servidor remoto), e serão utilizadas pelos testes automáticos usados para avaliar o desempenho dos projectos. **Projectos que não cumpram integralmente este requisito 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 e semáforos. Como tal, é permitido recorrer a todas estas primitivas na solução a desenvolver.

Enquanto as *BFS-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.

Os ficheiros *cinema_map.h e cinema.h* encontram-se no pacote base e no Anexo B.

Problema I.B– Escalonador BFS-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.

Sendo o processador onde as tarefas se executam um recurso limitado, é desejável que o escalonador imponha alguma justiça no modo como distribui o tempo de execução entre as

várias tarefas. No entanto, se uma tarefa executar operações muito longas sem se bloquear enquanto outras se bloqueiam rapidamente, é fácil perceber que, com o escalonador actual das *sthreads*, a primeira tarefa verá processador dedicar-lhes mais tempo de execução.

Pretende-se modificar o escalonamento das *sthreads* por forma a assegurar uma distribuição mais justa do tempo de execução do processador entre as várias tarefas. Para isso, propõe-se o uso de uma variante simplificada do algoritmo de escalonamento *BFS*[1] chamada *BFS-sthreads.*

Algoritmo

No início da execução do processo, existe apenas uma tarefa, que executa a função main, com uma prioridade 5 e nice 19. A qualquer momento, novas tarefas podem ser criadas com prioridades entre 0 e 5 e nice entre -20 e 19.

O BFS-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 da tarefa;
- Tempo de execução acumulado pela tarefa;
- Valores *prioridade* e *nice* da tarefa (ver mais abaixo);
- Estado da tarefa (bloqueado, executável, em execução)
- Tempo acumulado pela tarefa em cada um dos diferentes estados da tarefa (ver abaixo).

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 0 (mais prioritária) e 5 (menos prioritária). O valor de *nice* de uma tarefa é inicialmente 0. Cada chamada à função *nice* pode modificá-lo para um valor entre -20 e 19.

Para além da prioridade e do nice, cada tarefa possui ainda dois outros valores: o virtual deadline (*VDL*) e o tempo restante (*remaining_time*). O VDL será objecto de descrição abaixo, quanto ao remaining_*time* corresponde a uma variável que é inicializada com o valor *do timeslice*. Em cada tick o *timeslice* da tarefa é reduzido do valor correspondente. Quando o *remaining_time* de uma tarefa chegar a zero a tarefa é retirada de execução e colocada na lista de executáveis, de acordo com a sua prioridade.

---

[1] http://en.wikipedia.org/wiki/Brain_Fuck_Scheduler

[2] Estas listas são designadas por listas de "tempo real" porque é essa a designação utilizada na especificação do BFS, mas não correspondem de facto a um escalonador de tempo real tal como

A lista de tarefas executáveis é, de facto, um conjunto de seis listas, uma para cada prioridade. O escalonador deverá escolher para execução uma tarefa da lista não vazia mais prioritária. Dentro de cada lista, a tarefa escolhida depende do escalonamento implementado em cada uma das listas.

As listas 0 a 3 são chamadas de listas de tempo real[2] e implementam um escalonamento FIFO. Nestas listas o *nice* não tem qualquer efeito, apenas a prioridade e a ordem em que elas foram introduzidas na lista das executáveis é relevante.

As listas 4 e 5 implementam um escalonamento diferente, que se descreve de seguida.
A lista com prioridade 4 é a lista das tarefas normais, i.e. todas as tarefas normais são executadas com prioridade 4. A tarefas com prioridade 5 são denominadas tarefas IDLE, i.e. só são executadas quando o processador estaria normalmente em estado ocioso (*idle*) caso não existissem.

Quando uma tarefa das listas 0 a 4 termina o seu timeslice, é de novo colocada no fim da lista da sua prioridade com o seu timeslice renovado, i.e.

remaining_time = TIMESLICE

Em que o TIMESLICE é uma constante do sistema que identifica o *timeslice* típico de uma tarefa. Quando uma tarefa das listas 5 e 6 termina o seu timeslice, é de novo colocada no fim da lista, actualizando o *timeslice* e o VDL de acordo com as seguintes fórmulas:

remaining_time = TIMESLICE
VDL = *currentime* + (nice+20)* TIMESLICE

O VDL representa assim o limite máximo de tempo que uma tarefa deve estar à espera na lista de executáveis. Note-se, no entanto, que este valor é virtual e nada garante que a tarefa seja escalonada até lá.
Quando uma tarefa é desbloqueada ou acordada, é colocada no fim da lista da sua prioridade sem qualquer actualização do *timeslice* ou VDL.

Nas listas FIFO a tarefa a escolher para execução é a primeira da lista, já nas listas 4 e 5 a tarefa a escolher é a que tiver menor VDL.

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 prioridade ou vdl inferior ao da

[2] Estas listas são designadas por listas de "tempo real" porque é essa a designação utilizada na especificação do BFS, mas não correspondem de facto a um escalonador de tempo real tal como é definido na disciplina.

tarefa actualmente em execução (e.g. na sequência de um assinalar efectuado sobre um semáforo).

O valor do RR_INTERVAL é um parâmetro do escalonador que deverá ser passado através da função `sthread_init(void *parameters)`. Inicialmente esse valor é igual a 30, mas poderá variar em alguns dos testes de funcionamento.

A biblioteca *BFS-sthreads* deverá oferecer suporte a trincos lógicos e semáforos, 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) implementado e pronto a usar;
- Suporte a trincos lógicos e semáforos implementados.

Caberá aos alunos estender a biblioteca base de forma usar o novo algoritmo de escalonamento, que suporta prioridades dinâmicas;

Naturalmente, cabe aos alunos decidir quais as estruturas de dados mais adequadas para manter o estado relativo às tarefas e clientes geridos pelo *BFS-sthreads*.

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

Existe uma função *void sthread_dump()* já implementada que imprime o estado das tarefas. Os dados necessários a essa impressão não estão no entanto preenchidos, devendo sê-lo pelos alunos, da forma que acharem mais adequada.

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. Contém também os ficheiros *cinema.h* e *cinema_map.h,* descritos no enunciado do problema I.A.
test-sthreads/	Contém vários testes para a biblioteca sthreads
cinema/	Contém a implementação de *cinema_map.h* e o ficheiro *show.c,* descritos no enunciado do problema I.A.

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_tnão 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".

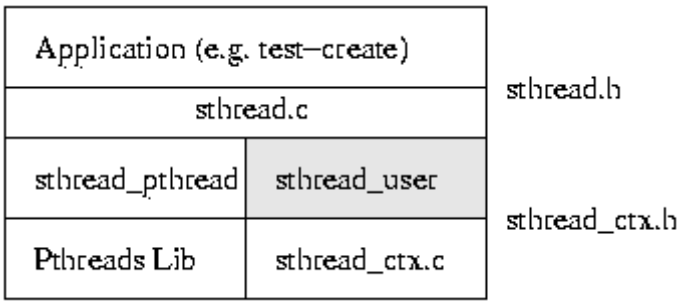

Fig. 2

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 lista. 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*.
-

Anexo B – Ficheiros *cinema.h* e *cinema_map.h*

```
/* cinema.h - API exposed to he client threads */

/* Struct used to identify the position of a seat in the cinema */
typedef struct Coords_t {
    int row;
    int column;
} Coords_t;

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

This function returns:
   *   0, if the initialization was successful;
   *  -1 if the initialization failed or had already taken place;
*/
int init_cinema(unsigned int num_rows, unsigned int num_cols, int
num_server_threads, int buf_size);

/* Submits a reservation request for the seats specified in the array
"lugares" having size "dim".
It returns -1 if not all seats are atomically reservable. Otherwise it
returns the reservation id, encoded as an integer larger than 0.*/
int reserva(Coords_t* lugares, int dim);

/* Submits a pre-reservation request for the seats specified in the array
"lugares" having size "dim". The duration in seconds of the pre-
reservation is specified by the parameter "segundos".
It returns -1 if not all seats are atomically reservable. Otherwise it
reserves the seats for the specified number of seconds, and returns the
reservation id, encoded as an integer larger than 0. */
int prereserva(Coords_t* lugares, int dim, int segundos);

/* Submits a confirmation request for a preservation having id
"reservation_id".
It returns -1 if the reservation does not currently exist; 0 if the pre-
booking was confirmed and updated into a booking. */
int confirma_prereserva(int reservation_id);

/* Submits a confirmation request for a preservation having id
"reservation_id".
It returns -1 if the reservation does not currently exist; 0 if the
reservation was removed and the correspondig seats made available */
int cancela(int reservation_id);

/* Submits a request for showing the current status of the cinema map and
the list of reservations and pre-reservations. The format of the output
produced by this function must be compliant with that specified in the
file cinema/show.c  */
void mostra_cinema();
```

```c
/* cinema_map.h - API to be used by the server threads to manipulate
(read and update) the state of the cinema seats map. The implementations
of these functions include artificial delays that are used to simulate
interactions with external devices (e.g. remote servers).

     *** THESE FUNCTIONS MUST NOT BE USED BY THE CLIENT THREADS ***
     *** THESE FUNCTIONS CAN BE USED ONLY BY THE SERVER THREADS ***

   *** THESE FUNCTIONS DO NOT IMPLEMENT ANY SYNCHRONIZATION SCHEME ***
    *** CONCURRENT ACCESSES TO THESE FUNCTIONS MUST BE REGULATED ***
              *** BY AN EXTERNAL SYNCHRONIZATION SCHEME ***
*/


/* struct encapsulating the reservation status of a seat */

typedef struct status_t {
    int state;       // 0=available; 1=pre-booked; 2=booked
    int id;          // if state != 0 it stores the reservation id
    time_t exp_time; // if seat is pre-booked it stores the expiration
time
} status_t;


/* Used to create a new cinema map. It takes as input parameters the
number of rows and columns of the map. */
void init_backend_cinema(unsigned int num_rows, unsigned int num_cols);

/* Mark a seat (at a given row and column) as booked by the reservation
having identifier id. Returns -1 if the coordinates passed as input are
invalid; 0, otherwise.*/
int backend_reserva(unsigned int row, unsigned int col, int id);

/* Mark a seat (at a given row and column) as pre-booked by the
reservation having identifier id and expiration time exp_time. Returns -1
if the coordinates passed as input are invalid; 0, otherwise.*/
int backend_prereserva(unsigned int row, unsigned int col, time_t
exp_time, int id);

/* returns -1 if the reservation or pre-reservation does not currently
exist; 0 if the reservation was removed and the correspondig seats made
available */
int backend_cancela(unsigned int row, unsigned int col);

/* returns the status of the seat identified by the coordinates passed as
input parameters; it returns a NULL pointer if the coordinates are
invalid*/
status_t* backend_get_state(unsigned int row, unsigned int col);
```