Projeto de Sistemas Operativos 2023-24
Enunciado da 1ª parte do projeto
LEIC-A/LEIC-T/LETI

O objetivo deste projeto é desenvolver o IST "Event Management System" (IST-EMS), um sistema de gestão de eventos que permite a criação, reserva e verificação de disponibilidade de bilhetes para eventos, como concertos e espetáculos teatrais.

O IST-EMS explora técnicas de paralelização baseadas em múltiplos processos e múltiplas tarefas de forma a acelerar o processamento de pedidos. Ao desenvolver o IST-SEM os alunos aprenderão também como implementar mecanismos de sincronização escaláveis entre tarefas bem como mecanismos de comunicação entre processos (FIFOs e signals). O IST-EMS irá também interagir com o sistema de ficheiros oferecendo portanto a possibilidade de aprender a utilizar as interfaces de programação de sistemas de ficheiros POSIX.

Código base

O código base fornecido disponibiliza uma implementação sequencial que aceita os seguintes comandos:

1. **CREATE <event_id> <num_rows> <num_columns>**
    - Este comando é usado para criar um novo evento com uma sala onde '*event_id'* é um identificador único para o evento, '*num_rows'* o número de filas e '*num_columns'* o número de colunas da sala.
    - Este evento é representado através de uma matriz em que cada posição codifica o estado do lugar:
        - **0 indica lugar vazio;**
        - **res_id > 0 indica lugar reservado com o identificador da reserva res_id**.
    - Sintaxe de uso: **CREATE 1 10 20**
        - Cria um evento com identificador *1* com uma sala de *10* filas e *20* colunas.
2. **RESERVE <event_id> [(<x1>,<y1>) (<x2>,<y2>) ...]**
    - Permite reservar um ou mais lugares numa sala de um evento existente. '*event_id'* identifica o evento e cada par de coordenadas *(x,y)* especifica um lugar a reservar.
    - Cada reserva é identificada por um identificador inteiro estritamente positivo **(res_id > 0)**.
    - Sintaxe de uso: **RESERVE 1 [(1,1) (1,2) (1,3)]**
        - Reserva os lugares *(1,1), (1,2), (1,3)* no evento *1.*
3. **SHOW <event_id>**

- Imprime o estado atual de todos os lugares de um evento. Os lugares disponíveis são marcados com *'0'* e os reservados são marcados com o identificador da reserva que os reservou.
- Sintaxe de uso: **SHOW 1**
    - Exibe o estado atual dos lugares para o evento *1.*

4. **LIST**
    - Este comando lista todos os eventos criados pelo seu identificador.
    - Sintaxe de uso: **LIST**
5. **WAIT <delay_ms> [thread_id]**
    - Introduz um delay na execução dos comandos, útil para testar o comportamento do sistema sob condições de carga.
    - O parâmetro [thread_id] é apenas introduzido no exercício 3, sendo que até ao mesmo, esta deve adicionar um delay à única tarefa existente.
    - Sintaxe de uso: **WAIT 2000**
        - Adiciona um delay do próximo comando por 2000 milissegundos (2 segundos).
6. **BARRIER**
    - Apenas aplicável a partir do exercício 3, porém, o parsing do comando já existe no código base.
7. **HELP**
    - Fornece informações sobre os comandos disponíveis e como usá-los.

Comentários no Input:
Linhas iniciadas com o caractere **'#'** são consideradas comentários e são ignoradas pelo processador de comandos (uteis para os testes).

- Exemplo: '*# Isto é um comentário e será ignorado'.*

1ª parte do projeto

A primeira parte do projeto consiste em 3 exercícios.

Exercício 1. Interação com o sistema de ficheiros

O código base recebe pedidos apenas através do terminal (*std-input*). Nesse exercício pretende-se alterar o código base de forma que passe a processar pedidos em "batch" obtidos a partir de ficheiros.

Para este efeito o IST-EMS deve passar a receber como argumento na linha de comando o percurso para uma diretoria "*JOBS*", onde se encontram armazenados os ficheiros de comandos.

O IST-EMS deverá obter a lista de ficheiros com extensão “.jobs” contidos na diretoria “JOB”. Estes ficheiros contêm sequências de comandos que respeitam a mesma sintaxe aceite pelo código base.

O IST-EMS processa todos os comandos em cada um dos ficheiros “.jobs”, criando um correspondente ficheiro de output com o mesmo nome e extensão “.out” que reporta o estado de cada evento.

O acesso e a manipulação de ficheiros deverão ser efetuados através da interface POSIX baseada em descritores de ficheiros, e não usando a biblioteca *stdio.h* e a abstração de *FILE stream.*

Exemplo de output do ficheiro de teste */jobs/test.jobs:*

```
1 0 2
0 1 0
0 0 0
```

Exercício 2. Paralelização usando múltiplos processos

Após terem realizado o Exercício 1, os alunos devem estender o código criado de forma que cada ficheiro “.job” venha a ser processado por um processo filho em paralelo.

O programa deverá garantir que o número máximo de processos filhos ativos em paralelo seja limitado por uma constante, **MAX_PROC**, que deverá ser passada por linha de comando ao arranque do programa.

Para garantir a correção desta solução os ficheiros “.jobs” deverão conter pedidos relativos a eventos distintos, isto é, dois ficheiros “.jobs” não podem conter pedidos relativos ao mesmo evento. Os alunos, por simplicidade, não precisam de garantir nem verificar que esta condição seja respeitada (podem assumir que será sempre respeitada nos testes realizados em fase de avaliação).

O processo pai deverá aguardar a conclusão de cada processo filho e imprimir pelo *std-output* o estado de terminação correspondente.

Exercício 3. Paralelização usando múltiplas tarefas

Neste exercício pretende-se tirar partido da possibilidade de paralelizar o processamento de cada ficheiro .job usando múltiplas tarefas.

O número de tarefas a utilizar para o processamento de cada ficheiro “.job”, **MAX_THREADS**, deverá ser especificado por linha de comando no arranque do programa. Serão valorizadas soluções de sincronização no acesso ao estado dos eventos que maximizem o grau de paralelismo atingível pelo sistema. Contudo, a solução de sincronização desenvolvida deverá garantir que qualquer operação seja executada de forma

“atómica” (isto é, “tudo ou nada”). Por exemplo, deverá ser evitado que, ao executar uma operação “SHOW” para um evento, possam ser observadas reservas parcialmente executadas, ou seja, reservas para as quais apenas um subconjunto de todos os lugares pretendidos tenham sido atribuídos.

Pretende-se também estender o conjunto de comandos aceites pelo sistema com estes dois comandos adicionais:

- **WAIT <delay_ms> [thread_id]**
  Este comando injecta uma espera da duração especificada pelo primeiro parâmetro em todas as tarefas antes de processar o próximo comando, caso o parâmetro opcional *thread_id* não seja utilizado. Caso este parâmetro seja utilizado, o atraso é injetado apenas na tarefa com identificador “thread_id”.

  Exemplos de utilização:
  - **WAIT 2000**
    - Todas as tarefas devem aguardar 2 segundos antes de executarem o próximo comando.
  - **WAIT 3000 5**
    - A tarefa com *thread_id* = 5, ou seja a 5ª tarefa a ser ativada, aguarda 3 segundos antes de executar o próximo comando.

- **BARRIER**
  Obriga todas as tarefas a aguardarem a finalização dos comandos anteriores à **BARRIER** antes de retomarem a execução dos comandos seguintes.

  Para implementar esta funcionalidade, as tarefas, ao encontrarem o comando **BARRIER**, deverão retornar da função executada pela *pthread_create* devolvendo um valor de retorno *ad hoc* (p.e., o valor 1) de forma a indicar que encontraram o comando **BARRIER** e que não acabaram de processar o ficheiro de comandos (nesse caso as tarefas deveriam devolver um valor de retorno diferente, p.e., 0).

  A tarefa *main*, ou seja a tarefa que arranca as tarefas “trabalhadoras” usando *pthread_create()* deverá observar o valor de retorno devolvido pelas tarefas trabalhadoras usando *pthread_join* e, caso detecte que o comando **BARRIER** foi encontrado, arranca uma nova ronda de processamento paralelo que deverá retomar a seguir ao comando **BARRIER**.

  Exemplos de utilização:
  - **BARRIER**
    - Todas as tarefas devem chegar a este ponto antes de prosseguirem com os seus próximos comandos.

Este exercício deveria ser realizado idealmente a partir do código obtido após a resolução do exercício 2. Neste caso o grau de paralelismo atingível será **MAX_PROC * MAX_THREADS**. Contudo, não serão aplicadas penalizações se a solução deste exercício for realizada a partir da solução do exercício 1.

Submissão e avaliação

A submissão é feita através do Fénix **até ao dia 15/12/2023 às 23h59**.

Os alunos devem submeter um ficheiro no formato *zip* com o código fonte e o ficheiro *Makefile*. O arquivo submetido não deve incluir outros ficheiros (tais como binários). Além disso, o comando *make clean* deve limpar todos os ficheiros resultantes da compilação do projeto.

Recomendamos que os alunos se assegurem que o projeto compila/corre corretamente no cluster *sigma*. Ao avaliar os projetos submetidos, em caso de dúvida sobre o funcionamento do código submetido, os docentes usarão o cluster sigma para fazer a validação final.

O uso de outros ambientes para o desenvolvimento/teste do projeto (e.g., macOS, Windows/WSL) é permitido, mas o corpo docente não dará apoio técnico a dúvidas relacionadas especificamente com esses ambientes.

A avaliação será feita de acordo com o método de avaliação descrito no site da cadeira.

Os alunos não podem partilhar código e ou soluções com outros grupos. O código submetido tem de ser o resultado do trabalho original de cada grupo. A submissão de código com grande grau de semelhança com outros grupos ou realizado recorrendo a entidades externas ao grupo levará à reprovação dos grupos envolvidos e ao reporte da situação à coordenação da LEIC e ao Conselho Pedagógico do IST.