Projeto de Sistemas Operativos 2025-26
Enunciado da 2ª parte
LEIC-A/LEIC-T/LETI

A segunda parte do projeto consiste em 2 exercícios que visam:

i) Desenvolver um servidor de jogos para o PacmanIST que suporte jogos paralelos iniciados por múltiplos processos cliente ligando ao PacmanIST através de *named pipes;*

ii) Desenvolver um *signal handler* que permita gerar um *log* para ficheiro que descreve o estado de todos os tabuleiros de jogo atualmente existentes no servidor.

Código base

O código base fornecido para esta 2ª parte contém uma implementação do PacmanIST que corresponde a uma possível solução da primeira parte do projeto. Também contém uma implementação da API de cliente vazia. Considera-se também que, por simplicidade, a funcionalidade de gravação do estado (comando "G") implementada na primeira parte deve ser desativada na segunda parte.

Exercício 1. Interação com processos clientes por *named pipes*

O PacmanIST deve passar a ser um processo servidor autónomo, lançado da seguinte forma:

```
PacmanIST levels_dir max_games nome_do_FIFO_de_registo
```

Quando se inicia, o servidor deve criar um *named pipe* cujo nome (*pathname*) é o indicado como 3º argumento no comando acima. É através deste *named pipe* que os processos clientes se poderão ligar ao servidor e enviar pedidos de início de sessão. O 1º argumento é idêntico à primeira parte do projecto e indica a directoria onde o servidor deverá ler os níveis de jogo de comandos a executar disponíveis. O 2º argumento é o número máximo de instâncias de jogos paralelos geridos pelo servidor. O cliente deverá ser lançado, com:

```
client id_do_cliente nome_do_FIFO_de_registo ficheiro_pacman
```

onde os parâmetros especificam, respectivamente, o identificador do cliente (codificado através de um inteiro), o nome do *FIFO* de registo do servidor e, opcionalmente, o nome de um ficheiro com os comandos do Pacman (caso o último parâmetro não seja especificado, o cliente lê os comandos a partir do *stdin*).

Qualquer processo cliente pode ligar-se ao *pipe* do servidor e enviar-lhe uma mensagem a solicitar o início de uma sessão. Esse pedido contém os nomes de dois *named pipes*, que o cliente previamente criou para a nova sessão. É através destes *named pipes* que o cliente

enviará futuros comandos para o servidor e receberá as atualizações do tabuleiro de jogo do servidor no âmbito da nova sessão.
O cliente deve criar o FIFO de pedidos e o FIFO das atualizações adicionando ao nome dos pipes o id único recebido como argumento, como está no código base.

O servidor aceita no máximo max_games (clientes ou sessões) em simultâneo. Isto implica que o servidor, quando recebe um novo pedido de início de sessão e tem max_games sessões ativas, deve bloquear, esperando que uma sessão termine para que possa criar a nova.

Durante uma sessão, um cliente pode enviar comandos para alterar a posição do seu Pacman, usando o FIFOs de pedidos. Periodicamente (com um ritmo definido pelo parâmetro “TEMPO” do nível de jogo) os clientes recebem o estado atualizado do tabuleiro de jogo, que reflete os movimentos realizados pelo Pacman e pelos monstros no último ciclo de jogo. Com base nesta informação, o cliente do Pacman redesenha o tabuleiro de jogo usando a biblioteca ncurses.

Uma sessão dura até ao momento em que o servidor detecta que o cliente está indisponível, ou o jogo tenha terminado. Nas subsecções seguintes descrevemos a API cliente do PacmanIST em maior detalhe, assim como o conteúdo das mensagens de pedido e resposta trocadas entre clientes e servidor.

API cliente do PacmanIST

Para permitir que os processos clientes possam interagir com o processo servidor do PacmanIST, existe uma interface de programação (API), em C, a qual designamos por API cliente do PacmanIST.

As seguintes operações permitem que o cliente estabeleça e termine uma sessão com o servidor:

- `int pacman_connect(char const *req_pipe_path, char const* notif_pipe_path, char const *server_pipe_path)`
  Estabelece uma sessão usando os *named pipes* criados.
  Os *named pipes* usados pela troca de pedidos e notificações (isto é, após o estabelecimento da sessão) devem ser criados pelo cliente (chamando *mkfifo)*. O *named pipe* do servidor deve já estar previamente criado pelo servidor, e o correspondente nome é passado como argumento inicial do programa.
  Retorna 0 em caso de sucesso, 1 em caso de erro.

- `int pacman_disconnect()`
  Termina uma sessão ativa, envia o pedido de desconexão para o FIFO de pedido, fechando os *named pipes* que o cliente tinha aberto quando a sessão foi estabelecida (sem esperar por respostas pelo servidor) e apagando os *named pipes* do cliente. Deve ter o efeito no servidor de eliminar todos os recursos usados para gerir a sessão de jogo deste cliente.
  Retorna 0 em caso de sucesso, 1 em caso de erro.

Após de ter estabelecido a ligação com o servidor (`pacman_connect),`o cliente (ver Figura 1):

1) deve começar a ler do seu FIFO de notificações para receber atualizações periódicas do tabuleiro. A lógica necessária para a receção deverá ser implementada na função:

- `int receive_board_updates(char* tabuleiro)`

  que deverá atualizar o tabuleiro de jogo com base na mensagem de atualização enviada pelo servidor que, como já mencionado, envia periodicamente o conteúdo de cada uma das posições do tabuleiro.

2) deve ler comandos pelo *stdin* ou pelo ficheiro de *input*, usando uma tarefa dedicada. Esta tarefa é responsável por enviar os movimentos do Pacman ao servidor através do FIFO de pedidos. A lógica de comunicação com o servidor deverá ser implementada na função:

- `int pacman_play(char command)`
  Esta função não espera por respostas do servidor. Os efeitos deste comando aparecerão incorporados na próxima atualização periódica do tabuleiro.

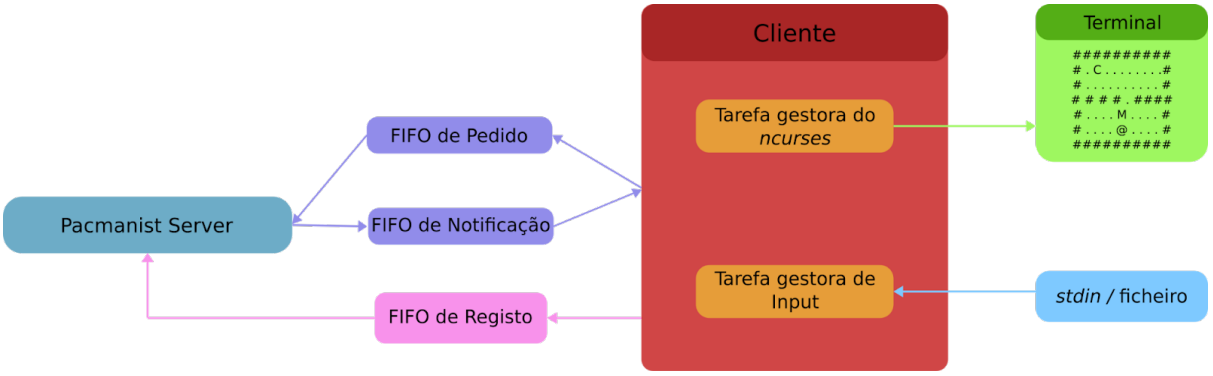

Figura 1: Arquitetura de um cliente, e a sua interação com I/O

Diferentes processos clientes podem existir concorrentemente, todos eles invocando a API acima indicada (concorrentemente entre si). Por simplificação, devem ser assumidos estes pressupostos:

- Cada processo cliente deverá usar duas tarefas. A tarefa principal deverá estabelecer a ligação com o servidor e receber as atualizações do estado do jogo pelo servidor, atualizando o estado do tabuleiro no processo cliente. Uma segunda tarefa lê os comandos do pacman do *stdin* ou a partir de ficheiro e envia-os ao servidor.
- Os processos cliente são corretos, ou seja cumprem a especificação que é descrita no resto deste documento. Contudo, podem desconectar-se de forma imprevista e isto não deve pôr em risco a correção do processo servidor.
- Quando um cliente fecha os *pipes* que o ligam ao servidor e o servidor detecta esse evento, o servidor deve eliminar toda a informação relativa a esse cliente.

Protocolo de comunicação

O conteúdo de cada mensagem (de pedido e resposta) da API cliente deve seguir o seguinte formato:

`int pacman_connect(char const *req_pipe_path, char const* notif_pipe_path, char const *server_pipe_path)`
Estabelece uma conexão com o servidor. Retorna quando receber uma mensagem de confirmação pelo fifo das notificações.

Mensagens de pedido
(char) OP_CODE=1 | (char[40]) nome do pipe do cliente (para pedidos) | (char[40]) nome do pipe do cliente (para notificações)
Mensagens de resposta (recebida pelo pipe de notificações)
(char) OP_CODE=1 | (char) result

`int pacman_disconnect(void)`
Mensagens de pedido
(char) OP_CODE=2
Mensagens de resposta
Nenhuma resposta é esperada, pois o cliente fecha todos os FIFOs que o ligam ao servidor.

`int pacman_play(char command)`
Mensagens de pedido.
(char) OP_CODE=3 | (char) command
Nenhuma mensagem de resposta é esperada. A função retorna -1 se ocorreu algum erro, ou 0 se não houver exceções.

`int receive_board_updates(char* tabuleiro)`
`Mensagens de atualização.`
(char) OP_CODE=4 | (int) width | (int) height | (int) tempo | (int) victory | (int) game_over | (int) accumulated_points| (char[width * height]) board_data

Onde:

- O símbolo | denota a concatenação de elementos numa mensagem. Por exemplo, a mensagem de resposta associada à função `pacman_connect` consiste num *byte* (`char`) seguido de um outro *byte* (`char`).
- Todas as mensagens de pedido são iniciadas por um código que identifica a operação solicitada (`OP_CODE`).
- As *strings* são de tamanho fixo (40 caracteres). No caso de nomes de tamanho inferior, os caracteres adicionais devem ser preenchidos com '`\0`'.
- O caso no qual o *byte* (`char`) *result* tem valor igual a 0 indica sucesso, qualquer outro valor indica um problema na execução de `pacman_connect`.

Implementação em duas etapas

A solução deste exercício será mais simples se for desenvolvida de forma gradual, em 2 etapas que descrevemos de seguida.

Etapa 1.1: Servidor PacmanIST com sessão única

Nesta fase (ver Figura 2), devem ser assumidas as seguintes simplificações (que serão eliminadas na próxima etapa). O servidor processa os ficheiros com os níveis como na primeira parte do projecto mas usa uma única tarefa adicional para atender apenas um cliente remoto de cada vez. O servidor só aceita uma sessão de cada vez (`max_games=1`).

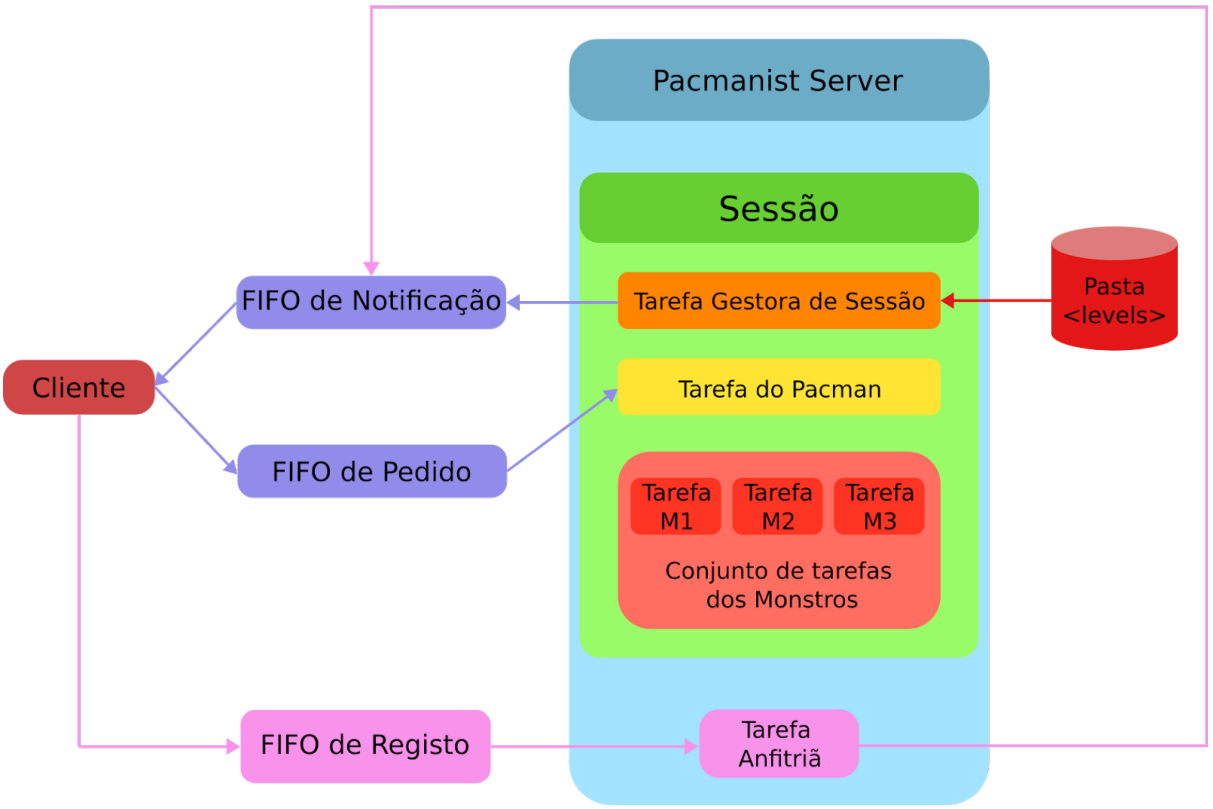

Figura 2: Arquitetura do servidor do Pacmanist, quando apenas suporta uma única sessão. A tarefa gestora da sessão não atualiza o tabuleiro de jogo, somente o lê periodicamente e envia-o para o cliente. Quem atualiza o tabuleiro do lado do servidor são as tarefas dos monstros e do Pacman. A tarefa do Pacman recebe as jogadas do cliente e atualiza o tabuleiro de jogo.

Etapa 1.2: Suporte a múltiplas sessões concorrentes

Nesta etapa (ver Figura 3), a solução composta até ao momento deve ser estendida para suportar os seguintes aspetos mais avançados.

Por um lado, o servidor deve passar a suportar múltiplas sessões ativas em simultâneo (ou seja, `max_games>1`).
Por outro lado, o servidor deve ser capaz de tratar pedidos de sessões distintas (ou seja, de clientes distintos) em paralelo, usando múltiplas tarefas (*threads)*, entre as quais:

- Uma das tarefas do servidor deve ficar responsável por receber os pedidos de conexão que chegam ao servidor através do seu *FIFO de registo*, sendo por isso chamada a *tarefa anfitriã*.
- Existem também max_games tarefas do Pacman que gerem os pedidos de jogadas, cada uma associada a um cliente e dedicada a servir os pedidos do cliente correspondente a esta sessão. As tarefas gestoras devem ser criadas aquando da inicialização do servidor. Note-se que estas tarefas são independentes das necessárias para executar os movimentos dos monstros nos tabuleiros de jogo.

A tarefa anfitriã coordena-se com as tarefas que gerem a interacção com os clientes da seguinte forma:

- Quando a tarefa anfitriã recebe um pedido de estabelecimento de sessão por um cliente, a tarefa anfitriã insere o pedido num buffer produtor-consumidor. As tarefas gestoras extraem pedidos deste *buffer* e comunicam com o respectivo cliente

através dos *named pipes* que o cliente terá previamente criado e comunicado junto ao pedido de estabelecimento da sessão. A sincronização do *buffer* produtor-consumidor deve basear-se em semáforos (além de *mutexes*).

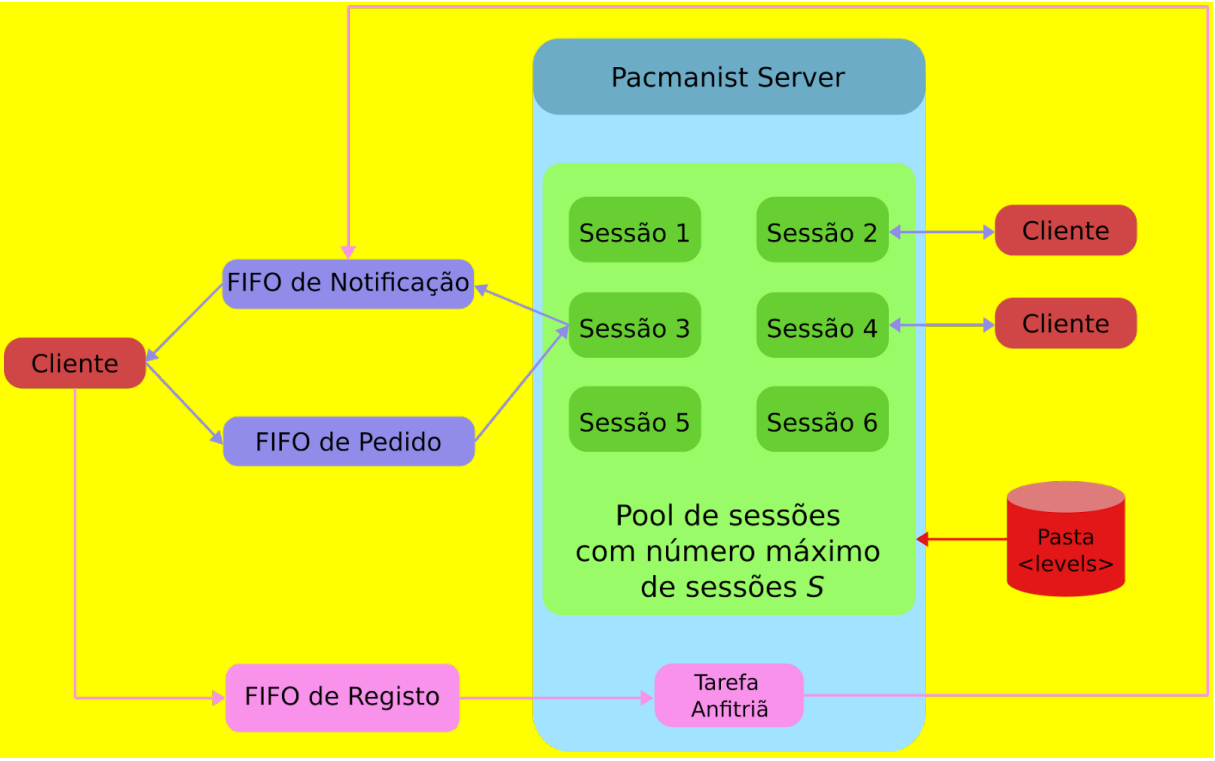

Figura 3: Arquitetura do servidor do PacmanIST, quando suporta múltiplas sessões.

Experimente:
Experimente correr os testes cliente-servidor que compôs anteriormente, mas agora lançando-os concorrentemente por 2 ou mais processos cliente.

Exercício 2. Terminação das ligações aos clientes usando sinais

Estender o PacmanIST de forma que no servidor seja redefinida a rotina de tratamento do sinal SIGUSR1. Ao receber este sinal, o (servidor) PacmanIST deve memorizar (na rotina de tratamento do sinal) que recebeu um `SIGUSR1`.

Apenas a tarefa anfitriã que recebe as ligações de registo de clientes deve escutar o `SIGUSR1`. Dado que só a tarefa anfitriã recebe o sinal, todas as tarefas de atendimento de um cliente específico devem usar a função `pthread_sigmask` para inibirem (com a opção `SIG_BLOCK`) a recepção do `SIGUSR1`.

Tendo recebido um SIGUSR1, a tarefa anfitriã deverá gerar um ficheiro com a lista dos 5 clientes atualmente ligados (identificados pelo próprio id) com a maior pontuação.

Ponto de partida

Para resolver a 2ª parte do projeto, os grupos podem optar por usar como base a sua solução da 1ª parte do projeto ou aceder ao novo código base. Caso se opte por usar a solução da 1ª parte do projeto como ponto de partida, poder-se-á aproveitar a lógica de sincronização entre tarefas.

Submissão e avaliação

A submissão é feita através do Fénix **até ao dia 9 de Janeiro 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.