Enunciado do Projecto de Sistemas Operativos 2016-17
Serviço de banca paralelo - Exercício 1

LEIC-A / LEIC-T / LETI
IST

O projeto de Sistemas Operativos 2016/17 está organizado em 4 exercícios encadeados. Esta secção introdutória apresenta uma visão global do projecto e dos tópicos abordados em cada um dos exercícios.

O projeto consiste em desenvolver um serviço chamado `i-banco`, que oferece funcionalidades de gestão de contas bancárias. O `i-banco` está apto a explorar o paralelismo disponível nas arquiteturas modernas, sendo pensado para ser alojado na *nuvem*.

Os 4 exercícios que constituem o projeto abordam os tópicos seguintes (aqui apresentados de forma resumida) :

- Exercício 1 - Desenvolverá o programa base do `i-banco`, que suporta operações básicas sobre as contas do banco e simulações complexas sobre as mesmas.
- Exercício 2 - Dotará o `i-banco` de uma *pool* de tarefas, permitindo a paralelização completa das operações sobre contas.
- Exercício 3 - Estenderá o `i-banco` com novas operações sobre contas, recorrendo a mecanismos de sincronização avançados.
- Exercício 4 - Suportará o acesso ao `i-banco` através de múltiplos terminais remotos, a salvaguarda das operações em sistema de ficheiros e a redireção dos resultados.

As secções seguintes deste documento descrevem apenas o 1^o exercício. Serão disponibilidados documentos similares para cada um dos exercícios seguintes.

1 Versão sequencial do `i-banco`

O 1^o exercício parte de uma implementação sequencial do `i-banco`.

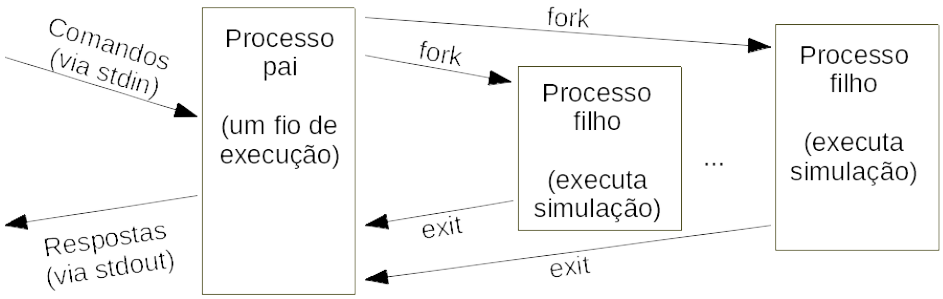

Figura 1: Representação dos processos e de algumas das chamadas sistema a utilizar.

Este programa inicial, cujo código está disponível na página da cadeira, gere um conjunto estático de n contas bancárias (n é parâmetro configurável em tempo de compilação).

Cada conta é identificada por um inteiro entre 1 e n. O `i-banco` associa a cada conta um saldo (por simplicidade, também um inteiro). Na versão fornecida do programa, os saldos das contas são mantidos num simples vector de inteiros.

O `i-banco` recebe comandos do *stdin* que permitem executar o seguinte conjunto de operações sobre as contas:

- `debitar idConta valor`

  Debita um determinado valor da conta identificada. Caso não exista saldo suficiente, nada é debitado e é apresentada mensagem de erro.

- `creditar idConta valor`

  Credita um determinado valor na conta identificada.

- `lerSaldo idConta`

  Imprime no *stdout* o saldo atual da conta identificada.

- `sair`

  Termina a aplicação.

2 `i-banco` com simulações em *background*

2.1 Simulações

Como primeiro exercício, pretende-se estender o `i-banco` com um novo comando:

`simular numAnos`

Este comando simula o saldo de cada conta ao longo de um número de anos (indicado em argumento) tendo em conta os juros (somados ao saldo da conta) e os custos de manutenção de conta (subtraídos à conta) em cada ano.

Mais precisamente, a simulação deve considerar que, a cada ano que passa, o saldo de uma determinada conta é atualizado da seguinte forma:

$$saldo_{novo} = max(saldo_{anterior} * (1 + taxaJuros) - custoManutencao; 0)$$

em que *taxaJuros* e *custoManutencao* são definidos como constantes do programa. Como valores referência, deve assumir-se uma taxa de juros de 10% e um custo de manutenção de 1.

Como saldo é um inteiro, o resultado da fórmula acima deve ser arredondado por baixo.

O resultado da simulação deve ser apresentado no *stdout* para cada ano solicitado seguindo o formato ilustrado por este exemplo:

```
SIMULACAO: Ano 0
================
Conta 1, Saldo 1000
Conta 2, Saldo 2000
Conta 3, Saldo 0

SIMULACAO: Ano 1
================
Conta 1, Saldo 1099
```

```
Conta 2, Saldo 2199
Conta 3, Saldo 0

SIMULACAO: Ano 2
================
Conta 1, Saldo 1207
Conta 2, Saldo 2417
Conta 3, Saldo 0

[...]
```

O Ano 0 acima significa o estado atual, ainda sem aplicação da fórmula acima.

A implementação do comando `simular` deve ser o mais simples possível e, sempre que possível, reutilizar as funções já existentes para aceder às contas.

2.2 Processamento em *background*

Como as simulações podem ser demoradas, pretende-se que sejam executadas em *background* num processo filho. Esta alteração permitirá que, enquanto uma simulação longa está em execução, outras operações simples sobre contas possam decorrer em paralelo. Apesar dessa possível concorrência (alterações ao saldo de contas enquanto a simulação decorre), a simulação deve basear-se apenas no estado que as contas tinham no instante em que a simulação iniciou.

2.3 Terminação coordenada

O comando `sair` só deve terminar o processo principal depois de todos os processos filho (em simulações) terem terminado.

Nesse momento, o processo pai deve imprimir no *stdout*, para cada filho: o respetivo *pid* e uma indicação da forma como terminou (terminação normal com *exit* ou terminação abrupta por *signal*).

O seguinte exemplo (em que 2 processos simulações foram lançadas) ilustra o formato que deve ser usado:

```
i-banco vai terminar.
--
FILHO TERMINADO (PID=2987; terminou normalmente)
FILHO TERMINADO (PID=2945; terminou abruptamente)
--
i-banco terminou.
```

Deve também ser implementada uma variante `sair agora` que permite terminar a aplicação mais cedo. Mais precisamente, ao receber este comando, o processo pai deve enviar um *signal* a todos os processos filho. Cada processo filho que ainda esteja ativo (a meio de uma simulação) e receba o *signal* deve: i) concluir a simulação das contas no ano em a simulação se encontra no instante em que chega o *signal*; ii) imprimir `Simulacao terminada por signal`; e iii) terminar o processo (sem continuar os anos que ainda estejam por simular).

2.4 Detalhes de implementação

Devem ser usadas as funções da API do Unix/Linux estudadas na teoria para programação com processos (*fork, exit, wait, kill, signal*). A solução não deve recorrer a funções de gestão de grupos de processos.

Por simplificação, o projeto pode assumir que o número de processos filho lançados por uma execução do `i-banco` é limitado por uma constante pré-conhecida (e.g., 20).

A solução deverá também incluir uma *makefile* com um alvo chamado `i-banco`, que gera um ficheiro executável da solução, com o mesmo nome. A *makefile* deve usar as opções `-g -Wall -pedantic` para compilar com o `gcc`. Soluções que não incluam *makefile* cumprindo estes requisitos não serão avaliadas.

3 Experimente

Para testar o programa, sugerimos compor ficheiros de texto com sequências de comandos. Por exemplo:

```
creditar 1 1000
creditar 2 2000
lerSaldo 1
lerSaldo 2
simular 5
debitar 2 50
lerSaldo 2
sair
```

Para correr estes lotes de comandos, basta depois lançar na linha de comandos: `i-banco < input.txt` (em que input.txt é um ficheiro com uma sequência de comandos tal como a apresentada acima).

Assim que tiverem implementado o comando de simulação em *background*, experimentar a sua execução seguida de operações que alteram o estado das contas. Confirmar que as operações se executam realmente em paralelo. Confirmar também se as alterações feitas sobre as contas influenciam ou não os resultados da simulação.

4 Entrega e avaliação

Os alunos devem submeter um ficheiro no formato zip com o código fonte e o ficheiro *makefile*. O exercício deve obrigatoriamente compilar e executar nos computadores dos laboratórios.

A data limite para a entrega do primeiro exercício é 14/10/2016 até às 23h59m. A submissão é feita através do Fénix.

Após a entrega, o corpo docente disponibilizará a codificação da respetiva solução, que pode ser usada pelos alunos para desenvolverem os exercícios seguintes.

5 Cooperação entre Grupos

Os alunos são livres de discutir com outros colegas soluções alternativas para o exercício. No entanto, *em caso algum* os alunos podem copiar ou deixar copiar o código do exercício. Caso duas soluções sejam cópias, ambos os grupos reprovarão à disciplina.