LICENCIATURA EM ENGENHARIA BIOMÉDICA

Sistemas Operativos

Trabalho prático[1]

6 de Dezembro de 2005 Duração: Dezembro

Simulação de um Parque de Estacionamento

Pretende-se simular a operação habitual de um parque de estacionamento. Os automóveis chegam ao parque, se não houver lugar livre esperam (passivamente!) e então avançam. Se existir pelo menos um lugar livre, o automóvel avança e irá à procura de lugar. Note que este enunciado permite que vários processos circulem dentro do parque à procura de um lugar vazio, criando a possibilidade de acidente... O número de automóveis é indeterminado mas assume-se que a certa altura será superior à capacidade do parque.

Este trabalho tem dois objectivos[2]: testar/desenvolver a capacidade de analisar e especificar problemas, e partir daí para a sua solução por via informática, e em segundo lugar testar/desenvolver a capacidade de concepção de algoritmos concorrentes. Para atingir o primeiro objectivo deverá começar-se por especificar o problema, dando especial atenção à protocolo que os automobilistas deverão seguir para entrar e sair do parque. Pretende-se que especifique a "API" correspondente, ou seja, as funções, argumentos, resultados e sua semântica. Os automobilistas limitar-se-ão a invocar estas funções pela ordem correcta.

Por outro lado, este trabalho prático visa consolidar conhecimentos de *Programação Concorrente*, incluindo a capacidade de visualização da execução de várias tarefas em simultâneo e das suas interações. Idealmente seria concretizado numa plataforma concorrente — por exemplo, C e Linux, ou java —, mas neste curso bastará a apresentação de um relatório detalhado que analise os problemas, hipóteses de solução, mecanismos de comunicação e sincronização, e finalmente apresente o pseudo-código da solução proposta. Note-se que, embora se facilite a apresentação de pseudo-código em Português, as estruturas de dados usadas e as primitivas da plataforma concorrente deverão ser usadas correctamente. Por outras palavras, deverá ter-se atenção aos argumentos e resultados das primitivas de comunicação e sincronização.

A execução paralela tem aspectos positivos (maior desempenho) e negativos (possibilidade de conflito no acesso a recursos partilhados, *deadlocks*, etc.), pelo que tem de ser controlada através de mecanismos apropriados. Neste trabalho propõem-se 2 mecanismos, representativos dos modelos conhecidos por *shared-memory* e *message-passing*. Em qualquer dos casos, a criação de actividades concorrentes

[1] Cotação — 30% nota final

[2] De facto há um terceiro objectivo, que é o de testar a capacidade de procura "pro-activa" de soluções nos livros, internet, etc., uma vez que este exercício não corresponde à aplicação directa de conhecimentos passados nas aulas.

poderá ser feita quer recorrendo à primitiva `fork()` do Unix/Linux quer instanciando várias cópias dos programas directamente a partir da *shell*[3].

No primeiro caso existe a possibilidade de partilha de memória (ou, mais correctamente, há recursos que podem ser partilhados/usados simultaneamente por várias actividades; pode ser memória, ficheiros, impressoras, etc.). Para facilitar, poderá considerar-se que existe uma declaração "mágica" que faz com que as estruturas de dados assinaladas como GLOBAIS sejam imediatamente visíveis a todos os processos[4]. Para coordenar os acessos concorrentes a estruturas de dados partilhados e para sincronizar a execução dos vários processos deverá usar as primitivas `P()` e `V()` sobre semáforos[5].

No modelo de *message-passing* não há partilha directa de memória. Assim, contrariamente ao que se passa no modelo *shared-memory*, os processos não podem consultar nenhuma tabela para saber se há lugar livre ou não. Por conseguinte, deverá estruturar a sua solução criando um processo "gestor", o único que tem a estrutura de dados que descreve o estado do parque e autoriza ou não a entrada de automóveis. Admita a existência de operações do tipo `send (pid, message)` e `receive(message)`, sendo que um `um receive()` antes de alguém ter enviado vai bloquear o processo até à recepção da mensagem.

[3]Por exemplo: `$ p0 & p1 & p1` cria 3 processos que vão executar, respectivamente, o programa `p0` e duas instâncias do programa `p1`

[4]Na realidade, poderia usar as primitivas `shmops()` do Unix V para obter tal efeito.

[5]Iniciais de *Proberen te verlangen* [P] e *Verhogen* [V]