Aula 12 de FSO

José A. Cardoso e Cunha
DI-FCT/UNL

Este texto resume o conteúdo da aula teórica.

1 Objectivo

Objectivo da aula: problemas da programação concorrente. Tipos de interacção entre processos: competição e cooperação. Exemplos: produtor-consumidor, cliente-servidor, leitores-escritores, reserva dinâmica de recursos. Suportes físsicos para a comunicação entre processos: arquitecturas de memória partilhada e de memória distribuída.

2 Encadeamento sequencial de actividades

Como vimos anteriormente (no estudo dos sistemas de multiprogramação), se tivermos uma aplicação decomponível em três tarefas: ler dados, processar os dados, imprimir resultados, podemos executá-las sequencialmente. Contudo, isso origina uma maior intervalo de tempo para completar a execução da aplicação e pode originar também uma má utilização dos recursos (periféricos e CPU), caso esta seja a única aplicação presente no sistema.

Surge, assim, o interesse em modelar aplicações em termos de tarefas concorrentes:

- na multiprogramação, o SO encarrega-se de considerar cada aplicação, activada por utilizadores possivelmente independentes, como um processo autónomo. O SO controla a execução concorrente dos múltiplos processos, submetidos ao SO, através do *shell*, ou directamente através de chamadas ao SO, como *fork*;
- na programação de aplicações individuais, permite-se a sua decomposição em termos de processos concorrentes, seja isto especificado a nível de uma linguagem de programação estruturada, ou a nível das chamadas ao SO.

A figura 1 ilustra duas possíveis maneiras de especificar processos concorrentes: a nível de linguagem de alto nível ou a nível de SO, como no Unix.

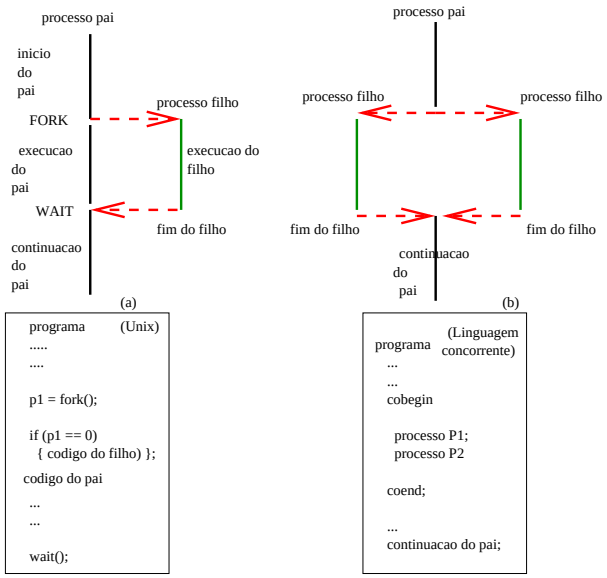

Figura 1: Especificação de processos concorrentes.

Enquanto que no caso (a) o processo pai continua, logicamente, a execução, concorrentemente com a do processo filho, criado por *fork*, no caso (b), o processo pai bloqueia-se durante a execução dos processos filhos, criados pelo comando *cobegin* (designação originada do termo *concurrent begin*). No caso (a), o pai, querendo, pode em qualquer ponto da sua execução aguardar explicitamente a terminação do filho, bloqueando-se em *wait*). No caso (b), o pai é automaticamente re-activado, pela implementação dos comandos *cobegin/coend*, sem que o programador tenha de o indicar explicitamente. No caso (b), a concorrência é bem explícita no texto do programa, tendo um único ponto de entrada (no *cobegin*) e um único ponto de saída (no *coend*).

3 Processos concorrentes

A nível do SO, o conceito de processo permite abstrair do facto de haver um ou múltiplos processadores reais a suportarem a execução dos programas.

Num sistema de multiprogramação, os processos concorrentes podem manifestar dois tipos principais de interacções:

- competição: os processos são independentes, por exemplo, de dois shells de dois utilizadores independentes, mas têm de partilhar os recursos comuns de um computador: tempo de CPU, espaço de memória, acesso aos periféricos;
- cooperação: os processos são interdependentes, fazendo parte de uma mesma aplicação, ou interagindo explicitamente uns com os outros, por exemplo, para pedirem 'serviços' uns aos outros, ou para efectuarem o processamento concorrente das entradas, tratamento e saída de dados, ou para executarem partes de um programa em paralelo, em processadores reais distintos.

Na figura 2 ilustram-se estes dois casos.

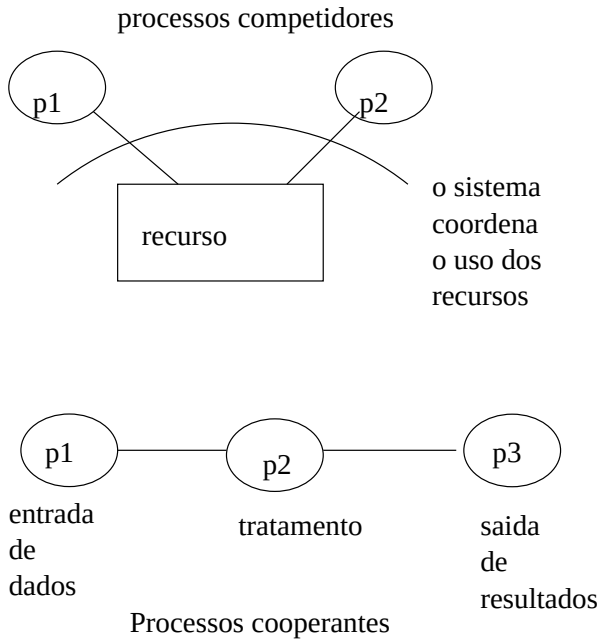

Figura 2: Processos competidores e processos cooperantes.

No caso da competição é o SO que se encarrega de todo o controlo. Ao criar um processo, o SO garante a sua execução numa 'cápsula' protegida, caracterizada por um mapa de memória protegido e privado, por um mecanismo que garante automaticamente que o processo dispõe de uma justa oportunidade para execução, mesmo que só haja um CPU, e pelo suporte de canais virtuais

de entrada e saída que lhe dão acesso a periféricos, apresentados sob a forma de 'ficheiros'.

No caso da cooperação, são os processos que têm de se coordenar mutuamente, seja para enviarem dados uns aos outros, seja para se sincronizarem no acesso a memória partilhada.

4 Estados de um processo

Para modelar os estados por que passa um processo, sujeito a execução num sistema onde há recursos partilhados com outros processos, que assim competem pelo acesso aos recursos, tem-se a figura 3.

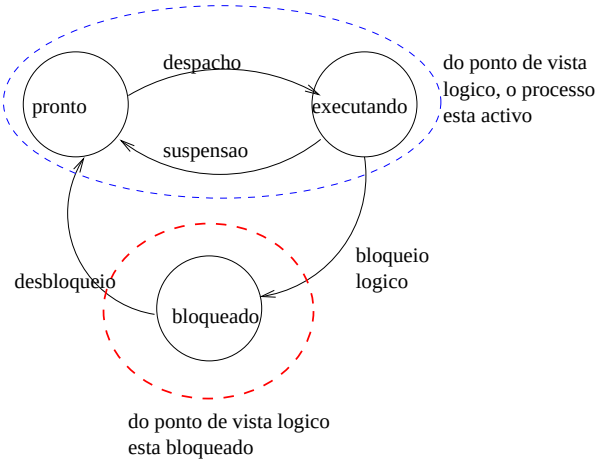

Figura 3: Estados de um processo.

Devido à competição por recursos partilhados, o SO vê-se obrigado a impor uma estratégia que dê uma oportunidade justa a cada processo. Por exemplo, se o recurso for o tempo de CPU, o SO dá uma fatia de tempo (TIME-SLICE) a cada processo, quando o activa para execução, ao fim da qual, se o processo ainda não tiver terminado ou não se tiver bloqueado a aguardar alguma condição lógica, o processo é obrigado a libertar o CPU para outro processo (passagem do estado 'executando' para o estado 'pronto').

Como se compreende, a competição por recursos partilhados fica totalmente ao cuidado do SO, não sendo uma preocupação do programador, ou seja, não exige programação explícita. Só se torna uma preocupação quando o número de processos competidores excede o limite aceitável para o número

de recursos disponíveis, mas então, a consequência é em geral apenas a de que o programa se executa mais lentamente ou exibe um tempo de resposta maior ao utilizador ao terminal.

A cooperação, pelo contrário, exige que o programador preveja as situações em que o programa deva aguardar pela satisfação de alguma condição, pelo que a passagem ao estado 'bloqueado' resulta da invocação de alguma primitiva no próprio programa, por exemplo, *read* de um *pipe* que esteja correntemente vazio.

5 Exemplos

Exemplos de interacção entre processos: vários processos reclamam o uso de um mesmo periférico. Se este for, por exemplo, uma impressora, este caso é habitualmente tratado delegando num processo de SPOOL de impressão, lançado após a inicialização do SO, a tarefa de tratar os pedidos que os processos utilizadores fazem para imprimir ficheiros. Estabelece-se, assim, uma relação cliente-servidor, ilustrada na figura 4

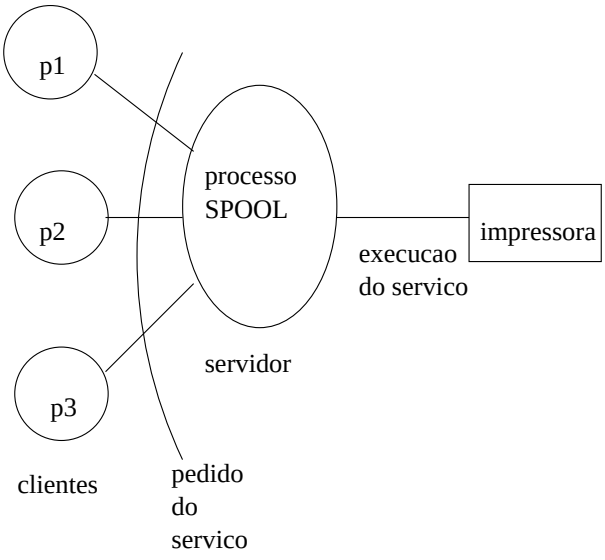

Figura 4: SPOOL.

Outro exemplo de interacção ocorre quando dois processos partilham um *buffer* em memória, no qual escrevem (processos chamados produtores) e lêem (processos chamados consumidores) elementos (figura **??**). Neste caso,

as condições de bloqueio potencial são: o acesso 'simultâneo' ao buffer, a situação de buffer cheio, quando se tenta escrever, e a de buffer vazio, quando se tenta ler.

Ainda outro exemplo de interacção ocorre quando múltiplos processos utilizadores concorrentes, acedem a uma base de dados, suportada fisicamente em disco e em memória partilhada. Os processos que só consultam - chamados *leitores* - podem aceder concorrentemente, sem restruções entre si, mas os processos que fazem actualizações - chamados *produtores* - precisam de garantir acesso exclusivo, durante a escrita.

Outros exemplos estão relacionados com situações em que certos conjuntos de recursos, por exemplo, buffers em zonas de memória, podem ser utilizados dinamicamente. Isto significa que, conforme os pedidos dos processos utilizadores, estes necessitam de guardar dados temporariamente em buffers. Por exemplo, se há, num dado momento uma torrente de dados vindo de um periférico, é necessário reservar espaço de memória para os ir guardando, até os poder processar. Para isso, pode gerir-se um 'reservatório' de buffers em memória, devendo agora os processos utilizadores coordenarem-se nos pedidos para *reservar* e *libertar* buffers, quando já não precisem deles. Estas primitivas podem esconder as interacções indirectas entre processos: por exemplo se um processo estiver bloqueado a aguardar que haja buffers disponíveis, este pode ser desbloqueado quando outro processo devolver buffers ao reservatório. Este problema pode ser modelado como uma relação cliente-servidor.

6 Sincronização

Se os processos concorrentes fossem sempre independentes uns dos outros, por exemplo, shells de utilizadores desconhecidos, as únicas interacções entre eles, correspondentes a competição pelos recursos do sistema (CPU, memória, periféricos, ficheiros), podiam ser geridas totalmente pelo SO, de forma automática e transparente, isto é, sem a intervenção do programador.

Contudo, por diversas razões, algumas das quais se ilustraram acima, os processos estabelecem interdependências que têm de ser geridas explicitamente pelo programa. Isto acontece sempre que se desenvolvem aplicações concorrentes, decompostas em múltiplos processos, ou quando um processo cliente precisa de fazer um pedido a um outro processo.

Para que os processos possam cooperar de forma harmoniosa, isto é, garantindo que os dados por eles modificados se mantenham num estado coerente, é necessário impor relações de *sincronização* entre eles. Entende-se por sincronização o conjunto de restrições que determinam as ordens pelas

quais se podem desenrolar os processos concorrentes, quando são executados sob o controlo de um SO multiprogramado, mesmo que só sobre um CPU, em *time-sharing*, ou quando são executados sobre um multiprocessador, com vários CPUs.

A sincronização permite a um processo activo bloquear-se explicitamente aguardando uma certa condição. Ou permite a um processo actuar sobre outro, desbloqueando-o, através de um envio de um sinal, ou através do envio de uma mensagem, ou simplesmente através da escrita num buffer partilhado. Assim, podemos distinguir mecanismos de sincronização de acção directa (um processo actua explicitamente sobre outro) ou de acção indirecta (um processo assinala uma condição que, indirectamente acaba por desbloquear ou bloquear outro).

Exemplo de acção directa: enviar uma mensagem directamente a um processo.

Exemplo de acção indirecta: colocar dados num pipe, no qual outro processo estava bloqueado num read.

7 Comunicação

Os processos cooperantes, para além de terem de se sincronizar entre si, esperando uns pelos outros, também necessitam, em geral, de comunicarem informação uns aos outros, seja através de um meio intermediário, por exemplo, memória partilhada, seja através de uma comunicação directa, por exemplo, o envio de uma mensagem a outro processo.

A comunicação = transmitir informação de um processo para outro, pode envolver também sincronização, ou não, como veremos mais adiante.

Para se compreenderem as alternativas de comunicação e sincronização entre processos, teremos de ver, ainda que brevemente, quais os dois principais modelos abstractos de arquitecturas de sistemas de computadores. Faz-se notar que, por necessidade de exposição, a síntese que se faz aqui é muito sumária (outras disciplinas de ASC irão desenvolver esta temática em várias direcções).