Aula 19 de FSO

José A. Cardoso e Cunha
DI-FCT/UNL

Este texto resume o conteúdo da aula teórica.

1 Objectivo

Objectivo da aula: Sincronização por sinais assíncronos. O caso Unix.

2 Introdução

Em aulas anteriores já foram estudados alguns mecanismos de sincronização de processos concorrentes.

Em particular, estudámos os *semáforos* e vimos a sua aplicação em múltiplos problemas de controlo de processos concorrentes. Na altura, chamou-se a atenção para o facto de, no caso dos semáforos, a operação P() ter um carácter síncrono, já que é necessário que um processo a invoque, eventualmente bloqueando-se, até que lhe possa ser assinalada a execução de uma operação V(), por outro processo. Vimos que isto, em algumas aplicações, pode ser menos desejável, já que um processo, ao invocar P(), pode bloquear-se, sem ter tido a oportunidade prévia de saber se o valor do semáforo estava a zero ou não. Se este processo tivesse outras tarefas para executar, independentes das acções que devem aguardar a execução de uma operação V(), vimos que a única forma de programar, mantendo as definições originais de Dijkstra, era criar um processo filho, que fizesse P(), bloqueando-se enquanto o processo pai prosseguia.

Também vimos, ao estudar a comunicação por mensagens, que existe um modelo de entrega implícita e assíncrona de mensagens, baseado na declaração de procedimentos de tratamento (*handlers*) em cada processo, e num mecanismo que invoca esses procedimentos automaticamente, quando uma mensagem chega para esse processo. Evidentemente, isso origina uma interrupção da execução do programa do processo, no ponto em que ia, quando o procedimento de tratamento é invocado. Este mecanismo é parecido

com o mecanismo de interrupções de programa ao nível da arquitectura *hardware* do computador.

Nesta aula, vamos ver os aspectos essenciais de um mecanismo de notificação assíncrona de sinais, tal como existe nos SO Unix. Trata-se de uma forma simplificada do mecanismo de entrega assíncrona de mensagens, a principal diferenças sendo que um *sinal* não transporta um texto de uma mensagem, apenas veiculando ao destinatário um número inteiro que identifica um tipo de evento que ocorreu no sistema, durante a execução de um programa. A ideia chave é a de permitir ao processo notificado, alguma reacção em resposta à ocorrência do evento e, como em geral, esta é imprevisível, a única coisa que se pode fazer é declarar um procedimento, a ser invocado se e quando tal evento ocorrer.

3 Conceito básico

Um *sinal* é um mecanismo pelo qual um processo pode ser notificado da ocorrência de um evento, durante a sua execução.

Um sinal pode ser gerado devido à ocorrência de um evento provocado directamente pelo próprio processo que o recebe ou devido a eventos gerados por dispositivos hardware ou por outros processos.

Os sinais são gerados por falhas do hardware (e.g. erro no acesso à memória), por falhas que ocorrem na execução de um programa (e.g. violação de memória), por acções desencadeadas pelo utilizador (e.g. premir uma tecla de controlo do teclado) ou são explicitamente gerados por certas chamadas ao sistema (e.g. *exit* envia um sinal ao processo pai).

A figura 1 ilustra diversos momentos ligados ao processamento de um sinal.

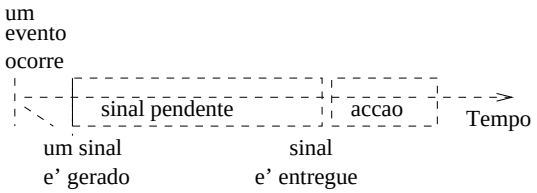

Figura 1: Sinais.

4 Alguns exemplos de sinais no Unix

No Unix existem diversos tipos de sinais. A cada tipo de sinal está associado um identificador inteiro (a que corresponde uma constante que, por convenção, começa pelo prefixo SIG). Para alguns sinais, o SO define certas acções específicas, enquanto que, para outros tipos de sinais, o SO deixa ao programador a liberdade de decidir de que forma reagir.

A seguir dão-se alguns exemplos, a título meramente ilustrativo, de alguns dos sinais mais importantes (veja o manual do Unix para uma descrição mais completa).

- Sinais gerados por falhas no hardware ou na execução do programa:
  - instruções ilegais (SIGILL)
  - falha no *bus* (SIGBUS)
  - falha na unidade de vírgula flutuante (SIGFPE)
  - falha no acesso à memória (SIGSEGV)
  - chamada ao SO inválida (SIGSYS)
- Sinais gerados pela tecla de controlo *interrupt* ou 'Control-C' (dependendo de configuração das rotinas de SO que controlam o teclado) (SIGINT)
- Sinal gerado pela chamada ao SO *alarm* (SIGALRM)
- Sinal enviado para destruir um processo (SIGKILL)
- Sinal enviado a um processo com canais de escrita abertos para um *pipe*, quando o último canal de leitura é fechado (SIGPIPE)
- Sinal enviado a um processo pai por um seu processo filho, ao terminar (SIGCHLD)

5 Acções no tratamento de um sinal

Quando um sinal é entregue a um processo, as seguintes acções são possíveis:

- **ignorar** o sinal: nada acontece, como se o sinal não tivesse sido gerado;
- **tratar** o sinal: uma função (*handler*), previamente especificada no programa do processo, é invocada, interrompendo a execução do processo, que é depois retomada, após a execução daquela função;

- **acção pré-definida pelo SO**: corresponde normalmente a desencadear a terminação forçada do processo e é adoptada pelo SO, se o programa não indicar outra acção; mas há excepções, por exemplo, os tipos de sinais SIGUSR1 e SIGUSR2, têm, pré-definida, a acção de ignorar sinal.

A figura 2 ilustra o caso de tratamento de um sinal.

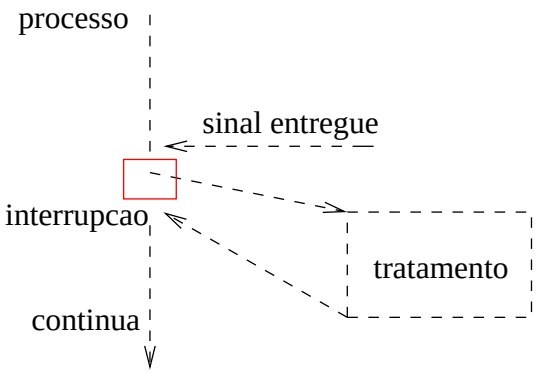

Figura 2: Tratamento de um sinal.

6 Declaração da acção de tratamento de um sinal

Historicamente no Unix, a declaração da função de tratamento de um sinal faz-se pela seguinte chamada ao SO:

```
typedef void (*sighandler_t)(int);

sighandler_t signal(int sig, sighandler_t handler);
```

em que:

- sig: indica o tipo de sinal
- handler: define a acção a efectuar:

  - função declarada no programa, indica o 'nome' da função *handler*, ou seja, um apontador para uma função, de tipo void, com um argumento inteiro (o tipo de sinal será passado como argumento à função invocada)
  - SIG_IGN (ignore)
  - SIG_DFL (default) acção pré-definida para sinais de tipo sig

- signal devolve um apontador para o anterior *handler*, uma função de tipo void, com argumento inteiro

Para declarar que se pretende ignorar sinais de tipo sig, invoca-se
signal(sig, SIG_IGN) no programa.

Para declarar que se pretende repor a acção pré-definida relativamente a sinais de tipo sig, invoca-se
signal(sig, SIG_DFL)
no programa.

Para declarar um *handler* que trate um sinal de tipo SIGINT (gerado pelo premir da tecla CONTROL-C), invoca-se
signal(SIGINT, func_int)
em que *func_int* é a função que se pretende seja invocada.

Por exemplo, para tratar os sinais devidos ao CONTROL-C:

```
int func_int()
{
  printf(``Interrupt \n'');
  ...
}

main()
{ ...
  signal(SIGINT, func_int);
  printf(``func_int instalado para SIGINT \n'');
  ...
}
```

No caso de ser premida a tecla CONTROL-C, o SO envia (dependendo das configurações das rotinas de controlo do teclado) um sinal SIGINT a todos os processos interactivos associados‘ àquele terminal, sendo que o processo *shell* está preparado para ignorar o sinal, mas os processos que

o não estejam, ao ser-lhes entregue este sinal, vêem activada a acção pré-definida pelo SO, a qual no caso de SIGINT, é terminar os processos à força.

7 Exemplo de utilização de sinais de alarme

A chamada ao SO *int alarm(int nsegundos);* desencadeia a geração de um sinal de tipo SIGALRM, que é entregue ao próprio processo, após decorridos *nsegundos*.

Assim, se o programa definiu uma função de tratamento, através da chamada ao SO *signal(SIGALRM, tratar-alarme);*, esta é invocada ao fim daquele tempo. Isto permite 'avisar' o processo de que passou aquele período de tempo e o processo pode mudar o seu curso de acção, por exemplo, se o processo estava bloqueado num *read* de um *pipe* vazio, ou numa chamada *msgrcv*, a entrega do sinal de alarme permite desbloqueá-lo.

8 Entrega de sinais, quando um processo está bloqueado

A figura 3 ilustra o caso em que é gerado um sinal destinado a um processo, que invocou uma chamada ao SO e ainda não retornou ao programa utilizador.

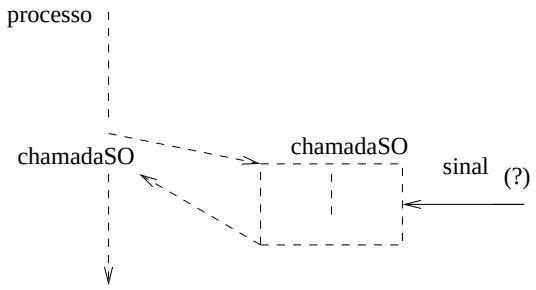

Figura 3: Efeito de sinais durante as chamadas ao SO.

Em geral, a execução da chamada ao SO é completada e o sinal só entregue no fim, isto é, logo que se dá o retorno. No entanto, se a chamada ao SO for 'lenta' ou 'bloqueante', o comportamento é diferente.

Exemplos de chamadas ao SO, caracterizadas como 'lentas' ou 'bloqueantes' são, pelas razões óbvias, as seguintes: *read, write, open, wait, pause*. Nota: *pause* é uma chamada ao SO que bloqueia o processo até que lhe seja entregue um sinal, seja qual for o seu tipo.

No caso destes tipos de chamadas, o comportamento, na ocorrência de um sinal é o seguinte:

1. o processo, ainda que bloqueado, é interrompido e a acção definida para o tratamento do sinal é efectuada;
2. em vez de retornar ao estado bloqueado, na chamada onde estava, esta é interrompida, dando-se o seu retorno forçado, com um código de erro (-1) e a variável global (em C) *errno* é afectada a EINTR, uma constante definida pelo sistema, que indica que houve um sinal.

Assim, por exemplo, o programa pode prever a ocorrência de um sinal do seguinte modo:

```
r = chamadaSObloqueante(...);

if (r == -1) && (errno == EINTR)
    { ... trata a ocorrência do sinal ...}
else
    {... trata o caso normal ...}
```

O tratamento, no caso de interrupção da chamada ao SO pode ser o de repetir a chamada, voltando potencialmente a bloquear-se ou, então, o de prosseguir a execução por outro caminho. Há versões do Unix que permitem ao programador indicar opções de comportamento a seguir automaticamente, no caso acima apontado, decidindo por exemplo, repetir a chamada automaticamente.

9 Exercício

Dadas as duas secções anteriores, programe um mecanismo de *time-out*, o qual, uma vez activado logo antes de uma chamada ao SO potencialmente bloqueante, lhe permita limitar o tempo de espera nessa chamada, a um valor máximo de *nsegundos*.

10 Envio explícito de sinais a um processo

A chamada ao SO *int kill(pid_ t pid, int sig);* permite enviar um sinal de tipo *sig* ao processo cujo *pid] é indicado, desde que os processos envolvidos tenham os mesmos identificadores de utilizador (*user-id, ou então desde que o processo invocador tenha *user-id* igual a zero (*root*).

11 Fiabilidade dos sinais no Unix

O mecanismo de sinais no Unix, nas suas primeiras versões (anteriores às versões 4.2 BSD e SVR3) não eram fiáveis.

- De cada vez que um sinal de um certo tipo era entregue a um processo, a acção de tratamento passava a ser sempre a acção pré-definida pelo SO, isto é, era preciso, 're-instalar' a função *handler* de cada vez que ocorria um sinal desse tipo. Isso dava origem a falhas durante a execução dos programas, cujas ocorrências dependiam do tempo.

- Também não era possível bloquear (inibir) a entrega de sinais de tipos determinados pelo programa, o que originava grandes problemas de programação, por exemplo, as funções de tratamento podiam, elas próprias, serem interrompidas em zonas críticas de código.

A partir das versões acima mencionadas, o Unix passou a garantir, segundo a norma POSIX, que aquelas anomalias não se verificam mais. Agora, a acção definida para tratamento de um sinal de um certo tipo mantém-se definida até que o programa a mude explicitamente. Também se tornou possível bloquear ou mascarar a entrega de sinais, de tipos definidos pelo programa, durante zonas críticas do programa, bem como durante a execução das funções de tratamento.

Para ver como isto se pode fazer, consulte as referências e o manual do Unix.

Faz-se, contudo, notar que um aspecto se manteve inalterado a nível da implementação de sinais no Unix: se ocorrerem dois ou mais sinais de um mesmo tipo, sucessivamente, antes de ter sido possível ao SO entregar o primeiro deles, não há garantia alguma de que todos os sinais sejam entregues. Aliás, é de esperar que, naquele caso, apenas um dos sinais seja entregue ao processo respectivo, sendo as outras ocorrências perdidas... porque o núcleo do SO não gere uma fila de sinais de cada tipo.

Assim, nas aplicações em que este comportamento possa originar situações de erro, deve-se complementar o envio/geração de sinais (se tal for possível), com o envio de outro tipo de informação por um meio de comunicação que tenha memória e seja confiável. Por exemplo, enviar uma mensagem por cada sinal gerado, e obrigar a função de tratamento, antes do seu retorno, a verificar se há mensagens pendentes por tratar.

12 Efeito das chamadas *fork* e *exec* sobre os sinais

Após uma chamada *exec*, todos os sinais que tivessem antes uma função de tratamento definida pelo programa, ficam 'configurados' com a acção pré-definida pelo SO; todos os outros sinais ficam com a mesma acção antes definida. Justifique este comportamento.

Após uma chamada *fork*, o processo filho herda as definições dos sinais do pai, excepto a do sinal de tipo SIGALRM.

13 Conclusão

O mecanismo de sinais do Unix, apesar do problema acima mencionado, é um bom exemplo de um mecanismo de notificação assíncrona de eventos. A sua utilização permite a um processo preparar funções de tratamento que são invocadas automaticamente pelo sistema, permitindo assim modelar comportamentos reactivos, em que o processo reage a certos eventos cujos momentos de ocorrência não se podem prever.