Programação de Sistemas

Sinais

Modelo de eventos (1)

- Os processos de nível utilizador interagem com o núcleo através de chamadas de sistema.
- Nos sistemas computacionais, os acontecimentos esporádicos, designados por **eventos** (por serem atómicos) ou **sinais**, levam à necessidade do núcleo interagir com o utilizador.
  Exemplos de sinais:
  - Excepções (ex: tentativa de acesso ilegal a memória, divisão por 0, …)
  - Gerados pelo utilizador (ex: abortar processo)
  - Gerados externamente (ex: E/S de disco)
  - Gerados por chamadas a sistema (ex: terminação de um processo, esgotamento do temporizador,...)

Modelo de eventos (2)

- Existem várias abordagens do núcleo contactar os processos de utilizador:

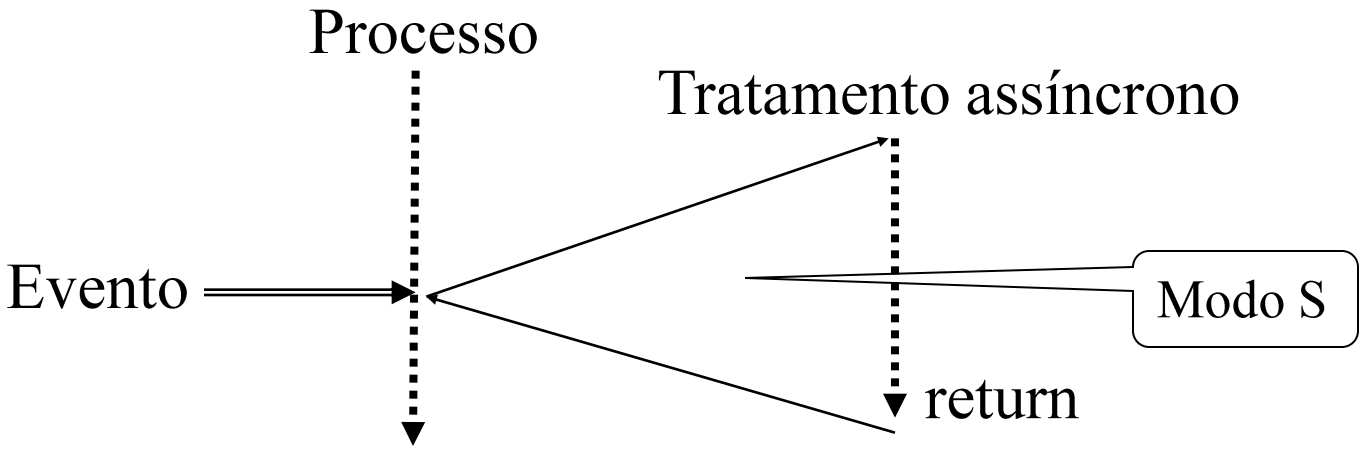
1. Obrigar todos os programas a verificar periodicamente a ocorrência dessas situações.
   Esta abordagem é incorrecta, porque
   - obriga projectistas a ter de codificar essa verificação
   - degrada o desempenho do programa
2. O sistema operativo multiprogramação tem uma unidade (processo ou fio de execução) à espera da ocorrência do evento.
   No entanto, esta abordagem é penalizante porque o elevado número de eventos obriga ao lançamento e gestão de elevado número de unidades funcionais.

Modelo de eventos (3)

Ideia: Associar uma rotina para tratar ("handle") o evento.

i. Quando o evento ocorre, passa-se de modo U (utilizador) para S (sistema). A execução do processo é interrompida e guardados os registos. Depois, a rotina é executada em modo U.

ii. Quando a rotina termina, regressa-se ao modo S para restabelecer os registos. Por fim, o processo retoma a execução na instrução seguinte em modo U.



Modelo de eventos (4)

- A resposta a eventos pode ser feita de 3 formas:
  A. Ignorar o evento.
  B. Tratar o evento por rotina do utilizador.
  C. O sistema operativo trata o evento. Nesta caso, afirma-se que o evento é tratado por omissão.
     Os eventos `SIGKILL` e `SIGSTOP` são sempre tratados por omissão! Porquê? o Linux não admite estes eventos sejam tratados pelo utilizador, para que o administrador tenha sempre a possibilidade de terminar um processo.

Definição de sinais (1)

- O Unix define códigos para um número fixo de sinais (31 no Linux), de tipo `int`.
  Nota: No Linux, os sinais `SIGIOT` e `SIGPOLL` possuem o mesmo código (29).
- Os identificadores, todos de prefixo SIG, podem ser consultados em diversos locais
  - Cabeçalho `/usr/include/asm/signal.h`
  - Comando `kill -l`
  - Manual `man 7 signal`
- O utilizador não pode definir sinais. Para resolver esta limitação, o Linux disponibiliza `SIGUSR1` e `SIGUSR2` para o utilizador usar como bem entender.

Definição de sinais (2)

- Alguns sinais definidos no POSIX (código para Linux, pode ser distinto noutros sistemas operativos)

Sinal	Código	Acção por omissão	Causa
SIGHUP	1	Termina	Terminal ou processo desconectado
SIGINT	2	Termina	Interrupção no teclado
SIGILL	3	Termina e gera core	Hardware (instrução ilegal)
SIGABRT	6	Termina e gera core	Gerado por instrução ABORT
SIGKILL	9	Termina	Força terminação do processo
SIGUSR1	10	Termina	Definido pelo utilizador
SIGSEGV	11	Termina e gera core	Hardware (referência inválida a memória)
SIGALRM	14	Termina	Esgotamento do temporizador
SIGCHLD	17	Ignora	Processo filho termina
SIGSTOP	19	Suspende	Suspender processo
SIGSYS	31	Termina e gera core	Chamada inválida a função de sistema

Tratados apenas pelo núcleo

Figura 10.1, Advanced Programming in the UNIX Environment

Tratamento de sinal (1)

- Função de tratamento de um sinal tem a seguinte assinatura
  `void (*sighandler_t)(int);`

A. O programador associa uma função ao tratamento de um sinal pela chamada

```
#include <signal.h>
sighandler_t signal(int, sighandler_t);
```

   - O 1º parâmetro identifica o código do sinal a tratar
   - O 2º parâmetro é o endereço da função a tratar o sinal, a constante `SIG_IGN` ou `SIG_DFL`
   - A função retorna o endereço da anterior função que tratava o sinal.

Nota: depois de tratado o sinal, como é tratado um novo sinal que seja gerado depois? Nalguns casos mantém-se a função de tratamento (ex: MacOS), noutros regressa-se ao tratamento de omissão (ex: RedHat Fedora).

Tratamento de sinal (2)

B. A função de tratamento pode ser identificada, ou alterada, por

```
#include <signal.h>
int sigaction(int, const struct sigaction *,
    struct sigaction *);
```

- O 1º parâmetro identifica o código do sinal a tratar
- O 2º parâmetro indica a nova função a tratar do sinal: se NULL, a anterior função mantém-se.
- O 3º parâmetro indica localização onde é salvo anterior tratamento (pode ser NULL).

```
struct sigaction {
        void (*sa_handler)();     /* tratamento sinal */
        sigset_t sa_mask;         /* sinal adicional a bloquear*/
        int sa_flags;             /* opções: usar 0 */
        void (*sa_restorer)();    /* não será usado! */ };
```

Envio de um sinal (1)

A. Envio de um sinal, em C, a um processo

POSIX:
```
#include <sys/types.h>
#include <signal.h>
int kill(pid_t, int);
```

– O 1º parâmetro identifica o processo alvo (i.e, para onde o sinal é enviado)
– O valor retornado é 0 (-1) em caso de sucesso (insucesso).

Nota: o facto de se designar por kill (matar), não significa que o processo alvo termine.

Envio de um sinal (2)

O efeito do sinal depende do 2º parâmetro indicado na chamada a signal()

- Se igual a `SIG_IGN`, o sinal não tem efeito
- Se igual a `SIG_DFL`, é executada a acção por omissão
- Se igual à referência de uma função de tratamento, ela é executada.

- Se um processo pretender enviar um sinal para si próprio, executar a instrução

ANSI-C:
```
#include <signal.h>
int raise(int);
```

Nota: equivalente a `kill(getpid(),int);`

Envio de um sinal (3)

B. Envio no interpretador de comandos
   Executar comando `kill [signal] PID`
   - O sinal é indicado em maiúsculas sem o prefixo SIG. Por omissão, o sinal é KILL

- Existem combinações de teclas que enviam sinais específicos para o processo em primeiro plano ("foreground")
  - CTRL+C : `SIGINT` (interrompe processo)
  - CTRL+\ : `SIGQUIT` (sai do teclado)
  - CTRL+Z : `SIGSTOP` (suspende processo)
    - `bg [%job]` : processo suspenso retorma no fundo ("background")
    - `fg [%job]` : processo suspenso retorma em primeiro plano

  Nota: por omissão, comandos bg,fg referem último processo suspenso.

Envio de um sinal (4)

- Um processo pode enviar sinais a outro processo, apenas se tiver autorização para o fazer.
  - Um processo com UID de `root` pode enviar sinais a qualquer outro processo.
  - Um processo com UID distinto de `root` apenas pode enviar sinais a outro processo se o UID real (ou efectivo) do processo for igual ao UID real (ou efectivo) do processo destino.
- Um processo pode suspender-se até receber um sinal

POSIX:
```
#include <unistd.h>
int pause();
```

C. Envio de um sinal, em C, a um fio de execução

POSIX:
```
#include <signal.h>
int pthread_kill(pthread_t, int);
```

Envio de um sinal-exemplos (1)

Exemplo 1: lançar processo, que envia em cada segundo o seu PID para o terminal. O lançador elimina o processo lançado ao fim de 5 segundos.

```
#include <sys/types.h>
#include <unistd.h>
#include <signal.h>
#include <stdio.h>
main() {
  pid_t pid=fork();
  if (pid==0) {
    for (;;) {
       printf("pid=%ld\n",getpid());
       sleep(1); }
  }
  else {
    sleep(5);
    kill(pid,SIGKILL); exit(0); }
}
```

```
[rgc@asterix Sinais]$ Segundo
pid=13704
pid=13704
pid=13704
pid=13704
pid=13704
[rgc@asterix Sinais]$
```

Envio de um sinal-exemplos (2)

Exemplo 2: programa que trata sinais de utilizador.

```
#include <sys/types.h>
#include <signal.h>
#include <stdio.h>

void tratamento(int sigNumb) {
   if (sigNumb==SIGUSR1) printf("Gerado SIGUSR1\n");
   else if (sigNumb==SIGUSR2) printf("Gerado SIGUSR2\n");
   else printf("Gerado %d\n",sigNumb); }

int main() {
   if (signal(SIGUSR1,tratamento)==SIG_ERR)
      printf("Erro na ligacao USER1\n");
   if (signal(SIGUSR2,tratamento)==SIG_ERR)
      printf("Erro na ligacao USER2\n");
   for(;;) pause(); }
```

Envio de um sinal-exemplos (3)

```
[rgc@asterix Sinais]$ Trata &
[1] 19215
[rgc@asterix Sinais]$ kill -USR1 19215
Gerado SIGUSR1
[rgc@asterix Sinais]$ kill -USR2 19215
Gerado SIGUSR2
[rgc@asterix Sinais]$ kill 19215
[rgc@asterix Sinais]$
[1]+  Terminated              Trata
[rgc@asterix Sinais]$
```

Lança o processo em fundo ("backgound")

O SIGKILL, sinal enviado por omissão, termina sempre o processo

Nota: a mensagem do Linux a indicar que o processo de fundo terminou só é escrita no terminal depois do utilizador fazer Enter no processo de primeiro plano.

Tratamento sinais após fork/exec

- Após o `fork`
  - O processo descendente herda o tratamento.
  - O processo descendente é livre de alterar o tratamento.
- Após o `exec`
  - Os sinais que estavam a ser tratados, passam a ter o tratamento por omissão.
  - Para os restantes sinais, o tratamento (omissão ou ignorar) mantém-se inalterado.

Alarmes (1)

- O temporizador é iniciado pela função

  POSIX:
  ```
  #include <unistd.h>
  unsigned alarm(unsigned);
  ```

  – O parâmetro indica os segundos até lançamento do alarme. Se for 0, cancela o alarme pendente.

  – O valor retornado, se diferente de 0, indica os segundos que faltam até terminar o temporizador pendente (lançado anteriormente).

- O temporizador lança o sinal `SIGALRM` quando chegar a 0.

Alarmes (2)

- Um fio de execução pode ser suspenso durante um intervalo de tempo. As funções podem retornar mais cedo se for gerado um sinal
  - Intervalo em segundos

    POSIX:
    ```
    #include <unistd.h>
    int sleep(int);
    ```
    - O valor retornado indica os segundos que faltam até o temporizador esgotar.
  - Intervalo em segundos

    POSIX:
    ```
    #include <time.h>
    int nanosleep(timespec*,timespec*);
    ```
    - O 1º parâmetro determina o tempo de suspensão.
    - O 2º parâmetro identifica o tempo que falta passar, caso entretanto tenha sido gerada outra interrupção.
    - `timespec` é estrutura com 2 campos, segundos e nanosegundos.

Máscaras de sinais (1)

- Um processo pode bloquear temporariamente um sinal, por forma que impeça a sua entrega para tratamento.
- Uma **máscara** de sinais contém o conjunto de sinais bloqueados.

Nota: não confundir bloquear com ignorar um sinal.

- Um sinal ignorado é entregue para tratamento e o "handler" não faz nada com ele.
- Um sinal bloqueado apenas será tratado mais tarde.
  No entanto, outras ocorrências do sinal bloqueado serão ignoradas até ser tratado!

Máscaras de sinais (2)

- A máscara de sinais bloqueados é definida pelo tipo de dados `sigset_t`: é uma tabela de bits, cada um referindo um sinal.

SigInt	SigQuit	SigKill	...	SigCont	SigAbrt
0	0	1	...	1	0

Nota: cada processo contém uma única máscara de sinais bloqueados.

Máscaras de sinais (3)

A. Limpeza na máscara de todos os sinais bloqueados

   POSIX:
```
#include <signal.h>
int sigemptyset(sigset_t *);
```

B. Preencher a máscara com todos os sinais bloqueados

   POSIX:
```
#include <signal.h>
int sigfillset(sigset_t *);
```

C. Passar um sinal a bloqueado

   POSIX:
```
#include <signal.h>
int sigaddset(sigset_t *,int);
```

Máscaras de sinais (4)

D. Sinal deixa de estar bloqueado

POSIX:
```
#include <signal.h>
int sigdelset(sigset_t *,int);
```

E. Testar se sinal se encontra bloqueado

POSIX:
```
#include <signal.h>
int sigismember(sigset_t *,int);
```

Máscaras de sinais (5)

F. Alteração da máscara de sinais

POSIX:
```
#include <signal.h>
int sigprogmask(int,
const sigset_t *, sigset_t *);
```

– O 1º parâmetro indica a acção a executar:
SIG_SETMASK: define nova máscara para o fio de execução
SIG_BLOCK: os sinais indicadas pela máscara são bloqueados
SIG_UNBLOCK : os sinais bloquados são removidos da máscara corrente.

– O 2º parâmetro indica a nova máscara: se for NULL, a máscara mantem-se inalterada.

– O 3º parâmetro indica o local onde máscara antiga é armazenada.

Máscaras de sinais (6)

- Exemplo de bloqueio do Alarme durante 10 seg

```
#include <signal.h>
#include <unistd.h>
#include <stdio.h>
#include <time.h>

time_t start, finish;
struct sigaction sact;
sigset_t new_set, old_set;
double diff;

void catcher( int sig ) {
   printf( "Dentro da funcao de tratamento\n" ); }
```

Máscaras de sinais (7)

```
int main( int argc,char *argv[] ) {

   sigemptyset( &sact.sa_mask );

   sact.sa_flags = 0;
   sact.sa_handler = catcher;
   sigaction( SIGALRM,&sact,NULL );

   sigemptyset( &new_set );
   sigaddset( &new_set,SIGALRM );
   sigprocmask( SIG_BLOCK,&new_set,&old_set );

   time( &start );
   printf( "SIGALRM bloqueado em %s\n",ctime(&start) );

   alarm( 1 ); /* SIGALRM gerado daqui a 1 segundo */
```

Máscaras de sinais (8)

```
do {
      time( &finish );
      diff = difftime( finish,start );
   } while (diff < 10);

   sigprocmask( SIG_SETMASK,&old_set,NULL );
   printf( "SIGALRM desbloqueado em %s\n",ctime(&finish) );
   return( 0 ); }
```

```
[rgc@asterix AlarmeBloqueado]$ Bloqueio
SIGALRM bloqueado em Sun Feb 22 17:11:55 2009

Dentro da funcao de tratamento
SIGALRM desbloqueado em Sun Feb 22 17:12:05 2009

[rgc@asterix AlarmeBloqueado]$
```

Sinais e fios de execução (1)

- Na presença de vários fios de execução, a estratégia de tratamento adoptada pelo Linux depende do tipo do sinal:

A. Sinais síncronos
   - Exemplos: SIGFPE (e.g., divisão por 0), SIGSEGV (acesso a memória protegida)
   - O sinal é entregue ao fio de execução que causou o erro.

B. Sinais assíncronos
   - Dirigidos: por exemplo `pthread_kill()`, enviados a um fio de execução específico.
   - Livres:
     - Cada fio de execução pode ter a sua máscara.
     - A distribuição de sinais livres pelas *threads* disponíveis depende da implementação do Linux.

Sinais e fios de execução (2)

A. A alteração da máscara dos sinais bloqueados para o fio de execução é semelhante a `sigprocmask()`

   POSIX:
   ```
   #include <signal.h>
   int pthread_sigmask(int,
   const sigset_t *, sigset_t *);
   ```

Nota: Na criação de fios de execução, o Linux

- O novo fio de execução herda a máscara de sinais.
- Os sinais pendentes não são herdados.

B. Uma thread fica bloqueada à espera do sinal pela função

   POSIX:
   ```
   #include <signal.h>
   int sigwait(sigset_t *,int *)
   ```

4º EXERCÍCIO TEORICO-PRATICO

- Implemente um programa simulador de uma jornada de trabalho.
  - A jornada de trabalho dura 12 segundos.
  - Um item é produzido em cada segundo.
  - Em cada 5 segundos há uma pausa para se tomar um café.