Fundamentos de Sistemas de Operação

Unix Windows NT Netware MacOS DOS/VS Vax/VMS
Linux Solaris HP/UX AIX Mach
Chorus

Programação Concorrente

Interacção por Sinais:

Parte I

Eventos e notificações assíncronas

- As notificações assíncronas - sinais - usam-se para assinalar acontecimentos “imprevisíveis”:
  - Podem ocorrer, ou não, no decurso da execução de um processo
  - Se ocorrerem, pode não ser possível prever quando
- Exemplos:
  - Violação do espaço de endereçamento – falha de segmentação;
  - Divisão por zero – falha aritmética;
  - Acções desencadeadas pelo utilizador para terminar o processo – e.g., interrupção CTRL-C no teclado
  - Notificações desencadeadas por outros processos

Sinais no UNIX

- Conceptualmente similares às interrupções hardware:
  - Mas oferecidas pelo SO (aos processos) ;
  - Permitem-lhes reagir à ocorrência de um evento.
- Identificados por um número ou tipo
- Enviados de forma assíncrona
  - Podem ser gerados/enviados de forma independente da execução do processo “receptor”.
- Recebidos de forma implícita
  - A sua entrega desencadeia, no receptor, a execução de uma função - pré-definida ou não – designada “signal handler” (por vezes omite-se a palavra “signal”, por ser óbvio).

Geração de sinais no UNIX

- Um sinal é gerado pelo SO como consequência de um evento:
  - Explicitamente desencadeado pelo próprio processo
    - alarm()
    - Terminação: normal – exit() – ou não – abort()
  - Detectado pelo hardware
    - Violação do espaço de endereçamento;
    - Erro aritmético – e.g., divisão por zero.
  - Desencadeado a pedido de um processo
    - Que o faz usando a chamada kill();

Envio de um sinal no UNIX

- **`int kill(pid_t pid, int signo)`**
  - *Envia o sinal de tipo (número)* **`signo`** *ao processo cujo identificador é* **`pid`** *(para um* **`pid`** *positivo; outros casos não serão aqui abordados)*
    - *Se o processo identificado por* **`pid`** *não existe, ou o emissor não tem permissões para lhe enviar o sinal, a chamada falha. Senão, o SO envia o sinal ao processo-alvo, que o irá receber e tratar.*

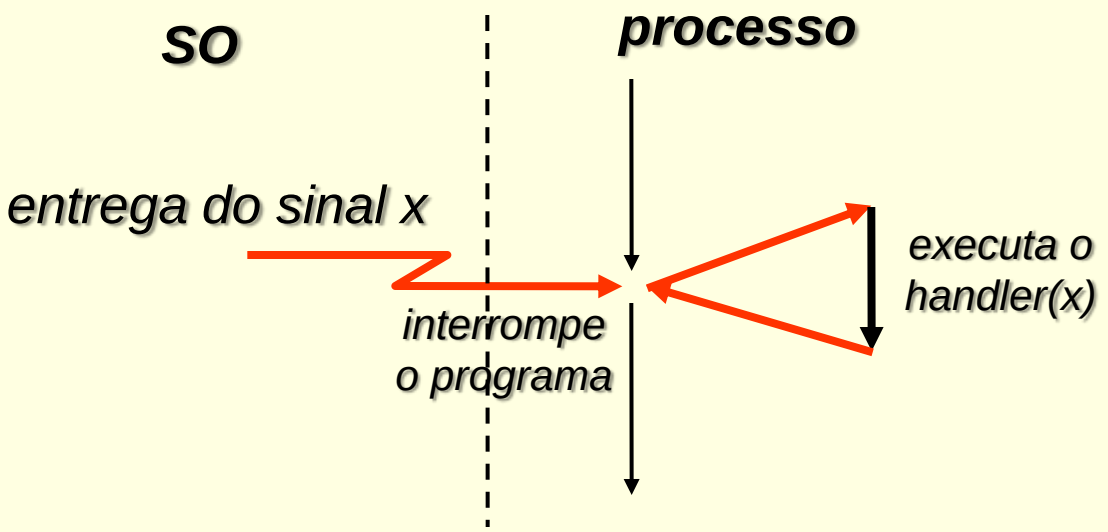
Exemplos de (tipos de) sinais

- **`#include <bits/signum.h>`**

```
SIGHUP          1          /* Hangup (POSIX).  */
SIGINT          2          /* Interrupt (ANSI).  */
SIGQUIT         3          /* Quit (POSIX).  */
...             ...        ...
SIGIOT          6          /* IOT trap (4.2 BSD).  */
SIGBUS          7          /* BUS error (4.2 BSD).  */
SIGFPE          8          /* Floating-point exception (ANSI).  */
SIGKILL         9          /* Kill, unblockable (POSIX).  */
...             ...        ...
SIGSEGV         11         /* Segmentation violation (ANSI).  */
...             ...        ...
SIGALRM         14         /* Alarm clock (POSIX).  */
...             ...        ...
SIGCHLD         17         /* Child status has changed (POSIX).  */
...             ...        ...
```

Recepção de um sinal no UNIX

- O SO gera o sinal, o processo recebe-o e "reage"…



Recepção e tratamento de sinais no UNIX (1)

- Um processo deixa o tratamento dos sinais
  - ao cuidado do SO e bibliotecas da linguagem, ao aceitar os handlers por omissão,
  - ou então declara o(s) seu(s) próprio(s) handler(s)

- Declaração de um handler
  - Um handler é uma função de argumento inteiro e resultado apontador:

```
void * fhandler( int sig );
```

  - Um handler é "instalado" no programa para "tratar" um sinal com a chamada:

```
signal (fhandler, signo);
```

Recepção e tratamento de sinais no UNIX (2)

- Tratamentos pré-definidos
  - Há dois tipos de tratamentos pré-definidos: registar, mas ignorar o sinal (**`SIG_IGN`**), ou registar e executar a acção por omissão para esse sinal (**`SIG_DFL`**).

- Ignorar um sinal
  - Como é evidente, se um sinal é ignorado então o receptor "não executa nenhuma acção" quando o recebe (embora "fique registado" que o recebeu – ver exemplo nos slides 11 e 12). Podemos fazer com que a recepção de um sinal seja ignorada com

```
signal(SIG_IGN, sinal);
```

Recepção e tratamento de sinais no UNIX (3)

- Tratamentos por omissão (default)
  - Há dois tipos de tratamentos por omissão: um consiste em ignorar o sinal; o outro, provocar a morte do processo.
- Exemplos:
  - Como exemplo de um sinal cujo tratamento por omissão é ser ignorado, menciona-se o sinal **`SIGCLD`**, enviado pelo SO ao pai de um processo quando este morre.
  - Para quase todos os sinais, o tratamento por omissão é provocar a morte do processo: **`SIGHUP, SIGINT, ..., SIGFPE, ...`**
- Repor o tratamento por omissão
  - Se alterarmos o tratamento de um sinal podemos posteriormente repor o tratamento por omissão com

```
signal(SIG_DFL, sinal);
```

Exemplo: tratamento de **SIGCHLD** (1)


```
int main() {

  int status;

  if ( fork() ) {
    printf("Pai %d\n", getpid());
    wait(&status);
    if (WIFSIGNALED(status))
      printf("Filho morto pelo sinal %d\n",WTERMSIG(status));
    return 0;
  } else {
    printf("Filho %d\n", getpid());
    sleep(30);
    printf("Vou terminar %d\n", getpid());
    exit(0);
  }
}
```

Exemplo: tratamento de **`SIGCHLD`** (2)

- *Comentários ao programa:*
  - *O sinal* **`SIGCHLD`** *é enviado ao processo pai; este, sai do* **`wait()`** *mas a variável* **`status`** *regista a recepção do sinal;*
  - *O teste à variável* **`status`**, *executado com a "macro" (***`#define`***)* **`WIFSIGNALED`** *indica que o processo (pai) recebeu um sinal.*
  - *A "macro"* **`WTERMSIG`** *é aplicada à variável* **`status`**, *para "extrair" o número do sinal.*

Semânticas de sinais

- Ao longo dos tempos, das versões e dos diferentes sabores (flavors) de UNIX (System V, BSD, SunOS, Solaris, Linux, …) a semântica dos sinais tem variado ☹
  - O handler é mantido depois de um sinal ser tratado, ou tem de ser "re-armado" com novo **`signal()`**?
  - Se um sinal está a ser tratado e entretanto chega outro, este último perde-se, ou é guardado e entregue mais tarde?
  - Se um processo está bloqueado numa chamada ao sistema e chega um sinal, a chamada é interrompida?
- A norma POSIX define uma nova chamada: **`sigaction()`**

API POSIX para sinais: **`sigaction()`**

- A estrutura

```
struct sigaction {
  void     (*sa_handler)(int);
  void     (*sa_sigaction)(int, siginfo_t *, void *);
  sigset_t sa_mask;
  int      sa_flags;
  void     (*sa_restorer)(void);
};
```

permite definir o handler tradicional, ao estilo System V (**`sa_handler`**), ou um "mais avançado" (em **`sa_sigaction`**) que não abordaremos; uma máscara (**`sa_mask`**) com os sinais a bloquear enquanto o handler se executa; e dois campos que não abordamos aqui, **`sa_flags`** e **`sa_restorer`**.

Exemplo: tratamento com **`sigaction`**


```
void sighandler(int signo) {
    ...
    return;
}

int main(int argc, char **argv) {
    struct sigaction action;

    memset(&action, 0, sizeof(action));
    action.sa_handler = sighandler;
    action.sa_flags = 0; // desnecessário

    sigaction(SIGIO, &action, NULL);
    ...
}
```