Departamento de Informática, Faculdade de Ciências da Universidade de Lisboa

Sistemas Operativos
2021/22

Guião de aula
Sinais e tempo

1. Introdução

Um sinal é uma notificação enviada a um processo informando-o da ocorrência de dado evento. Os sinais são muitas vezes designados por *interrupções de software*, designação que advém do facto de um sinal *interromper* o fluxo normal de execução de um programa. Neste guião vamos estudar sinais. Como estudo de caso do uso de sinais apresentamos também um conjunto de funções de manipulação do tempo e da sua utilização na criação de alarmes para controlo de um processo.

1.1. Sinais

Se tiver as permissões adequadas, um processo pode enviar um sinal a outro processo, ou até a si próprio. O mais comum, no entanto, é ser o *kernel* do Sistema Operativo a enviar sinais a um processo. Os principais eventos que podem despoletar o envio de um sinal pelo *kernel* a um processo são os seguintes:

1) O *hardware* detetou um erro e notificou o sistema operativo, o qual depois envia um sinal aos processos que tomam conta da ocorrência. Exemplos incluem uma divisão por zero ou referenciar uma zona de memória inacessível.
2) O utilizador carregou em caracteres que geram sinais, como o `Ctrl-C` (*interrupt*) ou o `Ctrl-Z` (*suspend*).
3) Ocorreu um evento de *software*. Por exemplo, os dados de um ficheiro ficaram disponíveis, um alarme disparou, ou um processo filho terminou.

Um sinal é *gerado* por um evento e é *entregue* a um processo, que depois executa uma ação em resposta ao sinal. Quando um processo recebe um sinal, uma de várias ações podem ocorrer, dependendo do sinal recebido. O sinal:

1) pode ser ignorado pelo processo;
2) pode levar à terminação do processo;
3) pode levar à terminação do processo e à criação de um ficheiro *core dump*[1];
4) pode levar à suspensão da execução do processo.
5) pode levar à continuação da execução de um processo previamente suspenso.

Às ações executadas quando um sinal é entregue a um processo chama-se o *comportamento* perante o sinal. Para cada sinal o processo tem um comportamento por omissão, mas este pode ser alterado pelo programador.

Há vários tipos de comportamento para lidar com um sinal:

1) Correr a ação pré-definida por omissão.
2) Ignorar o sinal. Esta ação é muitas vezes útil no caso de sinais cujo comportamento por omissão é terminar o processo.
3) Executar um *handler*. Um *handler* é uma função definida pelo programador que executa uma dada tarefa como resposta à receção de um dado sinal. O *handler* de um sinal é automaticamente invocado quando o processo recebe esse sinal.

[1] Este ficheiro contém uma imagem da memória virtual do processo que pode ser usada para *debugging*.

1.2. Exemplo de sinais

Como exemplo do tipo de sinais existentes em sistemas UNIX, deixamos agora uma breve explicação e descrição do comportamento por omissão de cinco sinais disponíveis através do módulo *signal* do Python (ver man 7 signal):

- `signal.SIGALRM`
  O *kernel* envia este sinal quando um alarme dispara. Por omissão, o processo é terminado.

- `signal.SIGCHLD`
  O *kernel* envia este sinal a um processo pai quando um dos seus filhos termina. Por omissão este sinal é ignorado.

- `signal.SIGINT`
  Quando o utilizador carrega no caractere *interrupt* (normalmente, *`Ctrl-C`*), este sinal é enviado ao processo. Por omissão, o processo é terminado.

- `signal.SIGSTOP (python 2) ou signal.SIGTSTP (python 3)`
  Quando o utilizador carrega no caractere *suspend* (normalmente, *`Ctrl-Z`*), este sinal é enviado ao processo. Por omissão, o processo é suspenso.

- `signal.SIGCONT`
  Enviado para avisar a um processo suspenso que ele deve retomar a sua execução.

- `signal.SIGQUIT`
  Quando o utilizador carrega no caractere *quit* (normalmente, *`Ctrl-\`*), este sinal é enviado ao processo. Por omissão, o processo é terminado e é criado um ficheiro *core dump*.

1.3. Envio de sinais aos processos

Os sinais podem ser enviados aos processos através do teclado, da linha de comandos ou utilizando chamadas ao sistema operativo.

1.3.1. Teclado

Exemplos: sequências de teclas `Ctrl-C, Ctrl-Z, Ctrl-\`

1.3.2. Linha de comandos

O comando *kill* permite enviar sinais aos processos: kill -<sinal> <pid>. Se não for especificado nenhum sinal, por omissão, o comando *kill* envia o sinal TERM (15) ao processo.
O comando *fg* envia o sinal CONT ao processo (este comando usa o job id do processo, que pode ser obtido através do comando *jobs*). O tratamento por omissão associado a este sinal faz com que o processo retome a sua execução.

1.3.3. Chamadas ao sistema operativo

```
os.kill(pid, sig)
```

A chamada ao sistema operativo *kill* é utilizada para enviar um sinal a um processo a partir de outro processo.

1.3.4. Exemplo

No exemplo que apresentamos a seguir vamos criar um processo filho para imprimir 6 números, suspender a sua execução por 5 segundos e depois retomá-la e terminar.

```
from multiprocessing import Process
import time, os, signal

def sleepy():
   print("Filho: eu vou imprimir 6 números")
   for i in range(6):
      print(i+1)
      time.sleep(0.5) #para dar tempo ao pai de suspender o filho
   print ("Filho: terminei!")

filho = Process(target=sleepy)
filho.start()
time.sleep(1) # para dar tempo ao filho para começar a imprimir

print("Pai: vou suspender o filho por 5 segundos!")
os.kill(filho.pid,signal.SIGTSTP)

time.sleep(5)

print("Pai: vou acordar o filho")
os.kill(filho.pid,signal.SIGCONT)

filho.join()
```

Um exemplo da execução deste programa é apresentado a seguir.

```
Filho: eu vou imprimir 6 números
1
2
Pai: vou suspender o filho por 5 segundos!
Pai: vou acordar o filho
3
4
5
6
Filho: terminei!
```

1.4. Alterar comportamento por omissão

Alguns sinais não podem ser capturados nem ignorados:

- O sinal KILL (sinal 9) - quando este sinal é enviado a um processo, o processo termina a sua execução.
- O sinal STOP - quando este sinal é enviado a um processo, a execução do processo é suspensa.

Umas das formas possíveis de alterar o comportamento por omissão quando é recebido um sinal é através da função `signal()` do módulo *signal*.

```
signal.signal(signalnum, handler)
```

O argumento *signalnum* indica o sinal cujo comportamento por omissão se quer alterar. O segundo é o *handler*, isto é, a função que vai ser chamada quando este sinal for recebido pelo processo. O *handler* é chamado com dois argumentos: o número do sinal e um segundo argumento com um objeto do tipo *frame* (que podemos deixar como `NULL`).
Em vez de especificarmos um *handler* como segundo argumento da função `signal()` podemos especificar um destes dois valores:

`signal.SIG_DFL`
: Esta opção permite retornar ao comportamento por omissão. Esta opção é útil para desfazer o efeito de uma chamada anterior à função signal() que tenha alterado o comportamento por omissão de um sinal.

`signal.SIG_IGN`
: Esta opção permite ignorar o sinal.

1.5. Handlers

A entrega de um sinal a um processo pode interromper o seu fluxo normal de execução em qualquer momento. A sequência de operações executadas a seguir à receção de um sinal é ilustrada na figura seguinte. Quando o sinal é enviado ao processo (ponto 1) o *kernel* invoca o *handler* (ponto 2). Esta acção interrompe o fluxo normal de execução do programa (na figura, isto acontece entre as instruções *n* e *n+1*). O *handler* é executado (ponto 3), e quando esta função retorna, a execução do programa continua (ponto 4) de onde tinha sido interrompida.

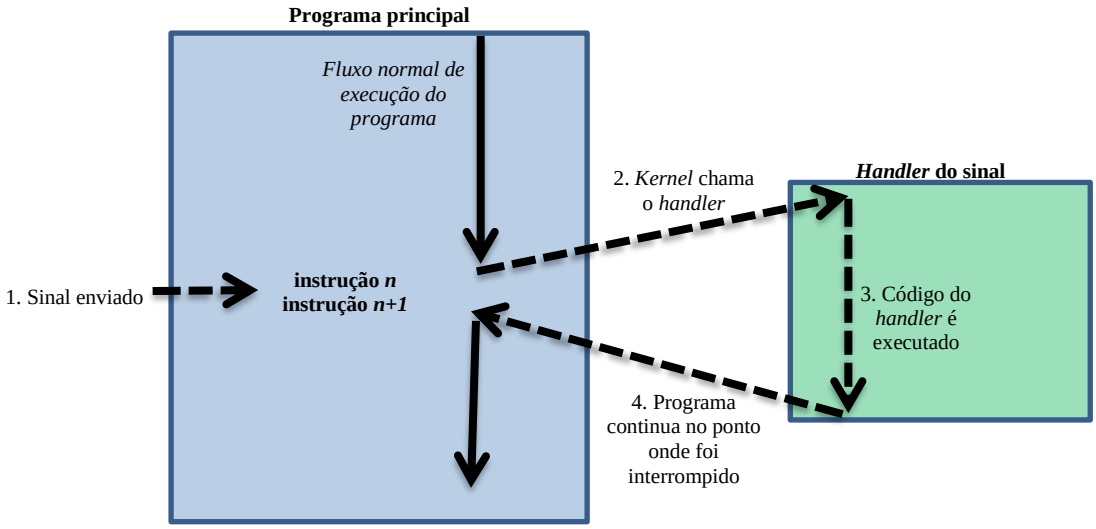

Figura 1. Envio de um sinal e execução do *handler* (baseado em [2])

1.5.1. Exemplo

No exemplo a seguir vamos modificar o comportamento por omissão de dois sinais: o `signal.SIGINT` e o `signal.SIGQUIT`. Em vez de o programa terminar, sempre que o utilizador carregar *`Control-C`* o programa vai incrementar um contador e imprimir o número de vezes que estas teclas já foram pressionadas. Quando o utilizador carregar *`Control-\`* o programa termina, depois de ser enviada uma mensagem ao utilizador.

```
import signal, time, sys

counter = 0

def controlC(sig,NULL):
      global counter
      counter += 1
      print ("carregou em ctrl+c ", str(counter), " vezes!")

def controlQuit(sig,NULL):
      print ("ok, sai ", str(counter), " ctrl+c depois!")
      sys.exit() #termina o programa

signal.signal(signal.SIGINT, controlC)
signal.signal(signal.SIGQUIT, controlQuit)

while True:
      time.sleep(1)
```

Um exemplo da execução deste programa é apresentado a seguir.

```
^Ccarregou em ctrl+c 1 vezes!
^Ccarregou em ctrl+c 2 vezes!
^Ccarregou em ctrl+c 3 vezes!
^Ccarregou em ctrl+c 4 vezes!
^Ccarregou em ctrl+c 5 vezes!
^\ok, sai 5 ctrl+c depois!
```

1.6. Tempo

Os sistemas UNIX representam internamente o tempo como o número de segundos desde a meia-noite do dia 1 de Janeiro de 1970, UTC.[2] A este "início dos tempos" chama-se *Epoch*.[3] Esta é a data aproximada do surgimento dos sistemas UNIX.

Os sistemas UNIX oferecem várias chamadas ao sistema relacionadas com tempo. O módulo time utiliza várias destas chamadas ao sistema para oferecer um leque alargado de funções de manipulação de tempo e de conversão de formato. Como ponto de partida para o estudo destes assuntos, nesta aula vamos analisar um conjunto (necessariamente) reduzido destas funções.

A função `time.time()` retorna o tempo passado, em segundos, desde a *epoch* (número real). Se correrrmos esta função antes e depois de um pedaço de código, podemos facilmente obter o tempo de execução desse código, o que por vezes é útil.

A função `time.gmtime([secs])`[4] converte o tempo expresso em segundos desde a *epoch* (por exemplo obtido através da função `time.time()`) numa estrutura do tipo `struct_time`. Por omissão, é retornado o tempo atual. Esta estrutura é um objeto que pode ser acedido através dos seus atributos. Por exemplo, o código seguinte:

```
import time

t = time.gmtime()
print ( t )
print ("===")
print (t.tm_year)
print (t.tm_mon)
print (t.tm_mday)
print (t.tm_hour)
print (t.tm_min)
```

retorna para o ecrã:

```
time.struct_time(tm_year=2020, tm_mon=11, tm_mday=10, tm_hour=16, tm_min=26,
tm_sec=32, tm_wday=1, tm_yday=317, tm_isdst=0)
===
2020
11
10
16
26
```

Mais informação sobre o objeto `struct_time` em https://docs.python.org/3/library/time.html#time.struct_time

[2] O UTC (*Coordinated Universal Time*) é o fuso horário de referência a partir do qual se calculam todas as outras zonas horárias do mundo.
[3] Em Português a melhor tradução talvez seja a palavra "era".
[4] Ver diferença entre time.localtime e time.gmtime

Uma outra função que pode ser útil é a `time.strftime(format)`, que converte um tuplo ou um objeto do tipo `struct_time` numa *`string`*, tal como especificada pelo argumento format. Por exemplo:

```
import time

print (time.strftime("%d %B %Y, %H:%M"))
```

retorna para o ecrã:

`22 November 2021, 16:29`

Mais informação sobre a formatação da *`string`* em https://docs.python.org/3/library/time.html#time.strftime

1.7. Alarmes

Um alarme permite a um processo agendar uma notificação para si próprio. Essa notificação é enviada ao processo sob a forma de um sinal – especificamente, um `signal.SIGALRM` – ao fim de um dado intervalo de tempo.

A função `signal.alarm(time)` permite requisitar um sinal `SIGALRM` para ser entregue ao processo dentro de `time` segundos. Qualquer alarme que tenha sido ligado anteriormente é cancelado (só podemos ter um alarme ligado por vez).

Por omissão, quando o alarme expira o processo é terminado. Se não for esta a ação pretendida, então é necessário criar um *handler* para correr quando o sinal `SIGALRM` for recebido pelo processo. Para isso, é necessário:

1) Definir o *handler*.
2) Chamar a função `signal.signal()` para indicar qual o *handler* a ser executado quando se receber o sinal;

Se desejarmos armar um alarme que dispara periodicamente, então podemos usar a função `signal.setitimer(which, sec [, interval])`. O argumento `which` define o tipo de alarme (nesta aula vamos considerar apenas o alarme `signal.ITIMER_REAL`). O alarme dispara depois de `sec` segundos (esta função aceita números reais, e não só inteiros, ao contrário da função `alarm()`). Depois desse primeiro alarme, este volta a disparar a cada `interval` segundos. Quando um alarme deste tipo dispara, um sinal `SIGALRM` é enviado ao processo.

1.8. Exemplo

No programa que apresentamos a seguir tentamos ilustrar alguns dos conceitos apresentados. Este programa corre durante 10 segundos, começando por "dormir" meio segundo (através de uma chamada à função `sleep()`). O programa inclui ainda o armamento de um alarme que dispara de 1 em 1 segundo, e que ao fim de 10 segundos termina o programa.

```
import signal, time, sys

counter = 0

def handler(sig, NULL):
      global counter
      counter += 1
      print (str(counter) + " segundos")
      if counter == 10:
            sys.exit()

signal.signal(signal.SIGALRM, handler)
signal.setitimer(signal.ITIMER_REAL, 1, 1)

t1 = time.time()
time.sleep(0.5)
t2 = time.time()

print ("tempo aprox. do sleep: " + str(t2-t1))

while True:
      time.sleep(1)
```

O resultado da execução deste programa é o seguinte:

```
tempo aprox. do sleep: 0.5051839351654053
1 segundos
2 segundos
3 segundos
4 segundos
5 segundos
6 segundos
7 segundos
8 segundos
9 segundos
10 segundos
```

A observação do código **a vermelho** permite verificar a configuração do alarme. O alarme dispara passado 1 segundo, e depois a sua periodicidade também é de 1 segundo. A chamada à função `signal()`, **a verde**, permite definir qual a função (o *handler*) que deve ser invocada cada vez que o alarme dispara (isto é, cada vez que o processo recebe o sinal SIGALRM). O *handler* (neste caso, a função com esse nome, `handler()`) imprime uma *string* com informação e termina o programa depois de passarem 10 segundos.

2. Exercícios fundamentais

1. Escreva um programa que arme um alarme que dispara uma vez por segundo, informando ao utilizador do tempo que já passou. Além disso:

a) Nos primeiros 10 segundos o programa deve contar o número de vezes que o utilizador carrega nas teclas *Ctrl-C*.
b) Ao fim de 10 segundos deve passar a ignorar o sinal recebido quando o utilizador carrega nessas mesmas teclas.
c) Ao fim de 15 segundos o programa deve terminar.

2. Desenvolva um programa que espere que o utilizador insira 20 números (um por linha) o mais rapidamente possível. De 5 em 5 segundos a função deve avisar quanto tempo já passou, e no final deve imprimir o tempo que o utilizador demorou a inserir os 20 números.

3. Faça um programa que realiza um ciclo em que é gerado um número aleatório entre 1 e 1000. Se o número gerado for 1 então terminar o ciclo e sair. Uma vez que o tempo de execução do processo é incerto, capture o *Ctrl-C* confirmando com o utilizador se quer mesmo sair e capture o *Ctrl-Z* confirmando com o utilizador se quer mesmo suspender o processo.

3. Bibliografia e outro material de apoio

[1] https://docs.python.org/3/library/signal.html
[2] https://docs.python.org/3/library/time.html