Departamento de Informática, Faculdade de Ciências da Universidade de Lisboa

Sistemas Operativos

2021/22

Guião de aula
Processos e *Threads*

1. Introdução

Nesta aula vamos estudar os processos e as *threads*, as abstrações básicas de computação dos sistemas operativos modernos. O uso destas abstrações nos nossos programas permite-nos executar tarefas de forma *concorrente* (ou mesmo *paralela*, se o sistema for multiprocessador).

1.1. Tratamentos de erros

O sistema operativo guarda a identificação do último erro que ocorreu na invocação de uma chamada ao sistema operativo, na variável `errno`.
A exceção `OSError` é gerada na invocação de várias chamadas ao sistema operativo, quando a sua execução dá origem a um erro. O seu atributo `errno` tem o código de erro e o atributo `strerror` tem a mensagem de erro correspondente.
As mensagens de erro devem ser escritas em `stderr`,
O tratamento de erros derivados de chamadas ao sistema operativo é exemplificado nos exemplos apresentados nas secções seguintes.

1.2. Processos

Um processo é uma instância de um programa em execução. Quando um programa é executado, o *kernel*[1]:

a) carrega o código do programa para memória;
b) reserva espaço em memória para as variáveis do programa; e
c) prepara as estruturas necessárias para armazenar informação sobre o processo, designadamente o bloco de controlo do processo, que contém a sua identificação (*Process ID*, ou PID), o seu estado, etc.

O processo `init` é o pai de todos os processos. Tem PID igual a 1. Todos os processos derivam desse.

Como se pode ver na figura ao lado, a memória usada por um processo é dividida em vários segmentos:

a) **texto:** contém as instruções do programa;
b) **dados:** contém as variáveis globais estáticas;
c) **heap:** área onde o programa pode reservar memória (variáveis globais) de uma forma dinâmica;

Figura 1. Organização de um processo em memória

[1] O *kernel* é a componente central do sistema operativo. É o *software* que gere os recursos de um computador.

d) **stack (ou pilha):** memória usada na chamada de funções, para passar parâmetros (de entrada e de saída) e para armazenar as variáveis temporárias usadas pelas funções. A pilha cresce e encolhe à medida que as funções são chamadas.

1.3. Criação de processos em Python

Para criar um processo usa-se a chamada ao sistema[2] `fork()`. Esta chamada ao sistema está acessível em Python através do módulo os:

```
import os

try:
  pid = os.fork()

except OSError as e:
  print >> sys.stderr, "fork failed ",e.errno, "-", e.strerror
  sys.exit(1)
```

Ao fazer-se esta chamada ao sistema é criado um novo processo e passam a existir dois processos – o **pai** e o **filho** – e a execução do programa continua, em ambos os processos, no ponto em que o `fork()` retorna. O processo chamador é o **pai**, e o novo processo é o **filho**. Caso não haja erros, o `fork()` retorna o PID do filho ao processo pai, e zero ao processo filho. Caso haja erros, é gerada a exceção `OSError`. O processo-filho é uma cópia do processo-pai[3]. Assim, qualquer um dos dois processos pode alterar as variáveis definidas em qualquer dos seus segmentos sem afetar o outro processo. Ou seja, o pai e o filho são ***independentes***. A figura seguinte ilustra esquematicamente estes conceitos.

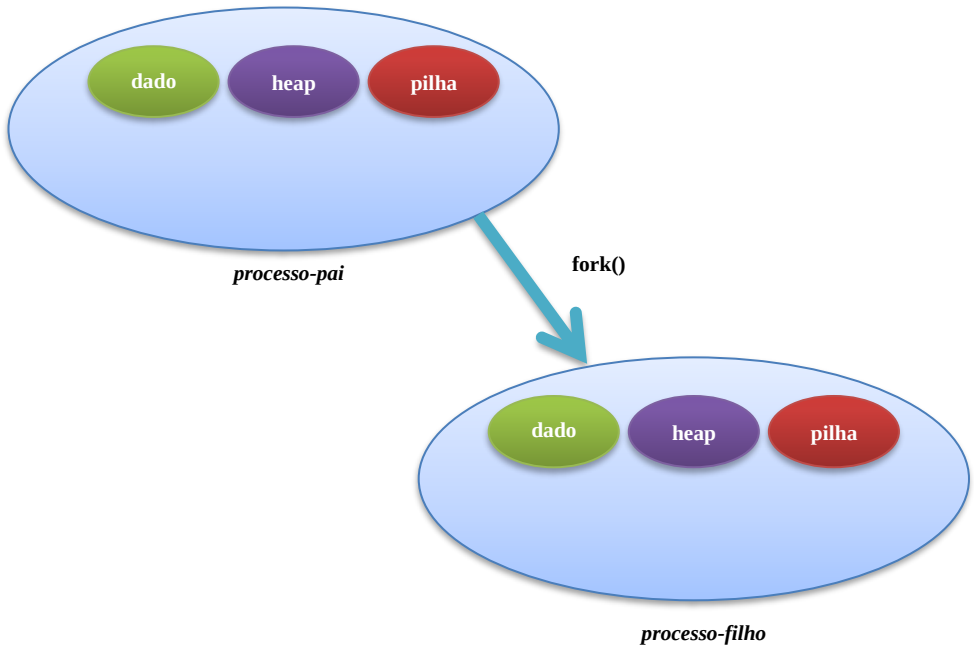

Figura 2. Chamada ao sistema `fork()`.

[2] Uma chamada ao sistema (ou *system call*) serve como ponto de entrada (controlado) para o *kernel*, permitindo ao processo pedir ao *kernel* que execute alguma ação. Para o programador, no entanto, uma chamada ao sistema é exatamente igual à invocação de uma qualquer função em Python. Entretanto, "atrás das cortinas" as coisas são um pouco diferentes…

[3] Esta cópia é muitas vezes concretizada de forma diferente em sistemas diferentes.

Reparem como os segmentos de memória do filho são uma cópia exata dos segmentos do pai, mas são totalmente independentes.

A chamada `sys.exit()` termina o processo devolvendo um valor inteiro. Por convenção, o valor 0 significa que o processo terminou normalmente, enquanto que um valor diferente de 0 significa que ocorreu um erro. Se em vez de um valor inteiro, for passado um objeto a `sys.exit()`, a função imprime o objeto em `stderr` e devolve o valor 1. Assim, passando uma `string` a `sys.exit()`, pode-se terminar um programa com uma mensagem de erro e devolver o valor convencionado para erro.

Para obter o PID de um processo e do seu pai podem usar-se as seguintes funções:

```
os.getpid()    → obtém PID do processo
os.getppid()   → obtém PID do seu pai
```

O próximo código em Python[4] ilustra o uso da chamada ao sistema `fork`[5].

```
import os, time

myVar = 10

def funcao():
      global myVar
      myVar = 1000
      time.sleep(5)

try:
  pid=os.fork()

  if pid == 0: #filho
      funcao()

  else: #pai
      time.sleep(5)

  print ("Sou o PID = " + str(os.getpid()) + ". myVar = " + str(myVar))

except OSError as e:
  print >> sys.stderr, "fork failed ",e.errno, "-", e.strerror
  sys.exit(1)
```

Nota: executem este programa em background e verifiquem que os processos são de facto criados com os PIDs esperados (os `sleep()` *introduzidos no código permitem que tenham tempo para os visualizar). Assumindo que gravam este código num ficheiro processos.py, basta correr estes comandos na* `shell`*:*

```
$ python processos.py &
$ ps
```

O resultado da execução deste programa é o seguinte (os números de PID serão diferentes em cada execução):

```
Sou o PID = 16000; myVar = 10
Sou o PID = 16001; myVar = 1000
```

[4] O código em Python apresentado vai ser sempre colocado dentro de uma caixa para se distinguir dos comandos a executar no terminal (e dos *shell-scripts*).

[5] Os exemplos apresentados incluem muitas vezes a função `sleep()` da biblioteca `time`. Esta função permite suspender um processo por um período especificado em segundos, e é por isso muito útil para dar tempo de se observarem os processos no sistema.

Reparem como o filho alterou a variável global `myVar` mas isso não afetou a variável do pai com o mesmo nome. Elas são globais em cada processo[6] mas são, de facto, *independentes*. Os filhos também podem, por sua vez, criar outros filhos, como no exemplo que se segue (o tratamento das exceções está omitido).

```
import os, time

pid_filho=os.fork()

if pid_filho == 0: #filho
      pid_neto=os.fork()

      if pid_neto == 0: #neto
            print ("Eu sou o neto. PID = " + str(os.getpid()))
            time.sleep(5)

      else: #filho
            print ("Eu sou o filho. PID = " + str(os.getpid()))
            time.sleep(5)

else: #pai
      print ("Eu sou o pai. PID = " + str(os.getpid()))
      time.sleep(5)
```

O resultado da execução deste programa é o seguinte:

```
Eu sou o pai. PID = 2562
Eu sou o filho. PID = 2563
Eu sou o neto. PID = 2564
```

A árvore dos processos gerados por este programa e o seu fio de execução são apresentados na figura a seguir.

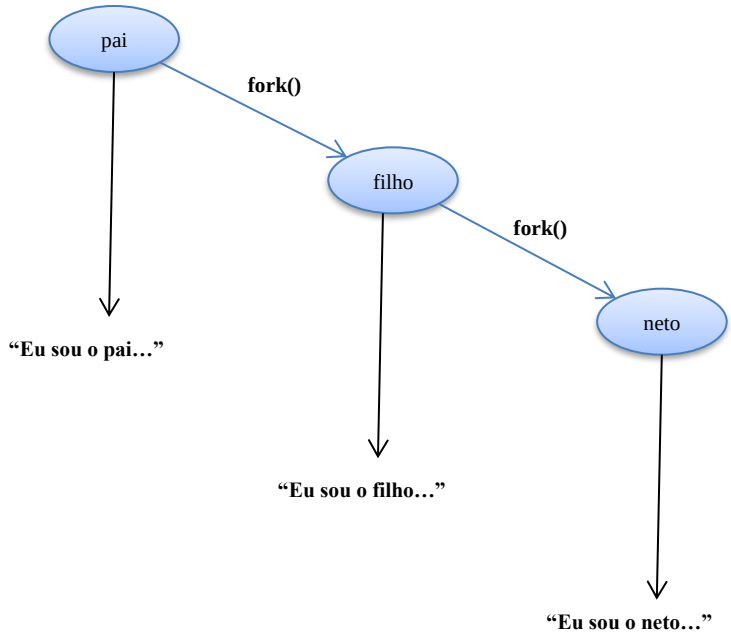

Figura 3. Árvore de processos.

[6] Quando se quer usar uma variável global numa função isso deve ser feito explicitamente, como na linha de código a vermelho

1.4. Execução de um novo programa

Como vimos nos exemplos anteriores, o processo-filho pode executar funções diferentes no mesmo código que partilha com o pai. No entanto, é muito mais comum o processo-filho executar um programa *completamente novo*. Para tal existe a família de chamadas ao sistema `exec()`. Há várias funções de biblioteca que chamam a `exec()`, tais como a `execlp()`, `execvp()`, entre outras. Estas diferem apenas pela forma como são passados os argumentos ao novo programa e pouco mais. Dada esta explicação, neste guião vamos usar apenas uma destas funções: a `execlp()`:

```
os.execlp(file, arg0, arg1, ...)
```

Nesta função, o `file` indica o ficheiro que é para ser executado. De seguida incluem-se todos os argumentos a passar ao novo programa, separados por vírgulas. Quando se corre esta função, o código do processo é substituído pelo novo programa. Há assim uma *separação radical* entre o pai e o filho.

Veja-se o exemplo que se segue, onde são criados 5 filhos que se vão separar radicalmente do pai correndo um novo programa, `filho.py`, passando-lhe o argumento `numFilho` (o tratamento das exceções está omitido).

```
import os, time

for i in range(5):
    numFilho = i+1

    pid = os.fork()

    if pid == 0: #filho
        os.execlp("python", "python", "filho.py", str(numFilho))
    else:
        print ("PID do filho " + str(numFilho) + " = " + str(pid))
```

O programa `filho.py` encontra-se ilustrado a seguir:

```
import os, sys

print ("Ola a partir do filho numero " + sys.argv[1] + ". PID = " +
str(os.getpid()))
```

Nota: `sys.argv[1]` contém o segundo argumento passado ao programa `filho.py`.

O resultado da execução deste programa é o seguinte:

```
PID do filho 1 = 16317
PID do filho 2 = 16318
PID do filho 3 = 16319
PID do filho 4 = 16320
PID do filho 5 = 16321
Ola a partir do filho numero 2. PID = 16318
Ola a partir do filho numero 1. PID = 16317
Ola a partir do filho numero 3. PID = 16319
Ola a partir do filho numero 4. PID = 16320
Ola a partir do filho numero 5. PID = 16321
```

Observe que a ordem em que os filhos executam é "aleatória", já que depende do escalonamento do CPU, algo sobre o qual o utilizador não tem controlo.

A chamada `os.wait()` suspende a execução do processo pai até que o processo filho faça `exit()`. Retorna o *`pid`* do filho e o status que inclui o valor de exit do filho.

O valor de status pode ser avaliado com as seguintes funções:

- `os.WIFEXITED(status)` – retorna verdadeiro se o processo filho terminou através da invocação da chamada exit.

- `WEXITSTATUS(status)` – avalia os 8 bits do status que correspondem ao valor utilizado no exit do processo filho (ou do return do main do processo filho).

O exemplo seguinte ilustra a utilização da chamada `os.wait()`:

```
import os
import sys
import random
try:
    pid = os.fork()
    if pid == 0:   #filho
        x = random.randrange(1,100)
        print ("o filho gerou: "+str(x))
        sys.exit(x)
    else:
        ipid, status = os.wait()
        if os.WIFEXITED(status):
              print ("Valor recebido no processo pai:
"+str(os.WEXITSTATUS(status))

except OSError as e:
    print >>sys.stderr, "fork failed "+str(e.errno)+"-"+e.strerror
    sys.exit(1)
```

1.5. Threads

Tal como os processos, as *threads* são um mecanismo que permite a uma aplicação executar várias tarefas simultaneamente (isto é, de forma *concorrente*). Quando um programa é iniciado, o processo consiste numa única *thread*, normalmente chamada de "*main thread*", mas cada processo pode depois ter múltiplas *threads* de execução (fios de execução). A *thread* corre o mesmo código do processo e **partilha** as variáveis globais (o segmento de dados e o *heap*[7]). Mas cada *thread* mantém a sua própria pilha. Ao contrário dos processos-filho, as *threads* podem assim comunicar entre si muito facilmente utilizando as variáveis globais partilhadas. Sendo partilhadas, a desvantagem é que é necessário *sincronizar* o acesso às mesmas. A figura seguinte ilustra a organização da memória de um processo após a criação de uma *thread*. Reparem nos segmentos partilhados – dados e *heap* – e nos segmentos que são locais a cada uma das threads – a pilha.

[7] Notem a diferença em relação à criação de processos-filho, onde estes dois segmentos eram *copiados,* passando a ser segmentos *independentes.*

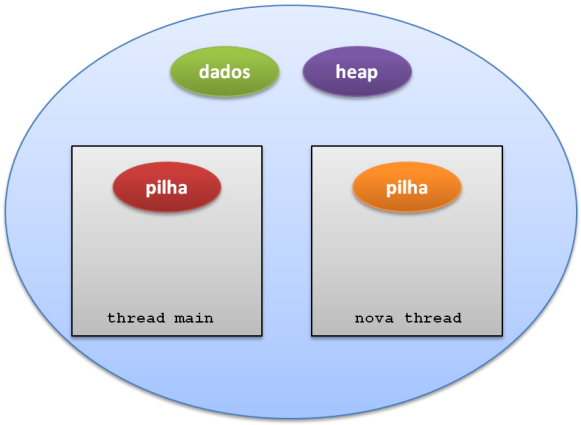

Figura 4. Criação de nova *thread*

A função construtor `Thread()` do módulo `threading` permite criar uma *thread* num processo.

```
import threading

newThread = threading.Thread(group, target, name, args, kwargs)
```

Quando a nova *thread* começar a ser executada vai correr a função definida em *`target`*, com os argumentos passados através do tuplo *`args`*. Os outros argumentos não serão explorados nesta aula. A execução de uma *thread* começa com:

```
newThread.start()
```

Notem que após a criação de uma nova *thread* o programa não tem qualquer garantia acerca de qual a próxima *thread* que vai utilizar o CPU. Portanto, o programador não tem nenhuma garantia relativamente à ordem em que vão ser executadas as instruções do programa.

O método `start()` gera uma exceção `RuntimeError` se for invocado mais do que uma vez sobre a mesma *thread*.

Quando a *thread* principal correr o método `join()` fica bloqueada à espera que a *thread* previamente lançada termine.

```
newThread.join()
```

O método `join` também pode gerar a exceção `RuntimeError` se uma *thread* fizer `join` a ela própria, ou se for feito `join` a uma *thread* antes de esta ter sido iniciada com `start()`.

O próximo código em Python ilustra o uso de *threads* num programa equivalente ao primeiro programa apresentado neste documento (omitindo o tratamento de exceções).

```
import time
from threading import Thread

myVar = 10

def funcao():
    global myVar
    myVar = 1000
    print ("Sou a nova thread. myVar= " + str(myVar))

newT = Thread(target = funcao)
newT.start()
newT.join()

print ("Sou a main thread. A nova terminou. myVar = " + str(myVar))
```

Apresenta-se agora o resultado da execução deste programa:

```
Sou a nova thread. myVar= 1000
Sou a main thread. A nova terminou. myVar = 1000
```

Como se pode confirmar, ao contrário do caso em que se criam processos novos com `fork()` (recordar primeiro exemplo apresentado neste guião), uma nova *thread* partilha as variáveis globais com a *thread* principal (`main`). Por isso, quando se altera a variável global, esta alteração é visível pela *main thread*.
Terminamos com um exemplo um pouco mais complexo do uso de *threads*, onde se tenta mostrar a concorrência entre várias *threads*.

```
import time
from threading import Thread

def somatorio(i):
    soma = 0
    for num in range(1, i+1):
        soma += num
    time.sleep(1) #simula computacao complexa
    print ("soma = " + str(soma) + "  ||  ")

def fact(i):
    prod = 1
    for num in range(1, i+1):
        prod *= num
    time.sleep(1) #simula computacao complexa
    print ("factorial = " + str(prod) + "  ||  ")

t = []

t.append(Thread(target = somatorio, args = (15,)))
t.append(Thread(target = fact, args = (15,)))

start_time = time.time()

t[0].start()
t[1].start()

for i in range(2):
    t[i].join()

elapsed_time = time.time() - start_time

print ("Threads terminaram em " + str(elapsed_time) + " segundos")
```

O resultado da execução deste programa é o seguinte:

```
soma = 120  ||
factorial = 1307674368000  ||
Threads terminaram em 1.00152802467 segundos
```

O código inclui *duas* chamadas à função `time.sleep(1)`. Seria, assim, de esperar que as *threads* terminassem em apenas 1 segundo, como aconteceu?

1.6. Módulo multiprocessing

Em Python existe um mecanismo de exclusão mútua, o *Global Interpreter Lock* (GIL), que obriga a que em determinadas situações apenas uma *thread* de cada vez possa executar instruções, impedindo que as aplicações com múltiplas *threads* possam tirar partido da existência de vários processadores para executar código em paralelo. Isto acontece porque a gestão de memória do interpretador de Python não é segura com *threads* e poderia gerar incoerências na informação relativa aos objetos do Python.

Para programas cuja execução gera muitas situações em que o GIL bloqueia a execução simultânea de *threads*, existe em alternativa ao módulo *threading* - o módulo *multiprocessing*, que ultrapassa esta limitação e permite paralelizar o programa. O módulo *multiprocessing* usa uma API praticamente igual à do módulo *threading*, apresentada na secção anterior, mas em vez de lançar novas *threads* lança em execução novos processos, permitindo que o programa corra, de facto, em paralelo num sistema multiprocessador.

O exemplo com que terminamos este documento obtém-se alterando o primeiro exemplo de *threads* apresentado na secção anterior, modificando o módulo utilizado: *multiprocessing* em vez de *threading*. Ou seja, o exemplo usa processos em vez de *threads*. Como se pode verificar, as alterações no código são mínimas (a vermelho).

```
import time
from multiprocessing import Process

myVar = 10

def funcao():
    global myVar
    myVar = 1000
    print ("Sou a nova thread. myVar= " + str(myVar))

newT = Process(target=funcao)
newT.start()
newT.join()

print ("Sou a main thread. A nova terminou. myVar = " + str(myVar))
```

Apresenta-se agora o resultado da execução deste programa:

```
Sou a nova thread. myVar= 1000
Sou a main thread. A nova terminou. myVar = 10
```

Como esperado, as variáveis globais de cada processo são agora completamente independentes.

2. Exercícios propostos

Todos os exercícios propostos são para ser realizados através da linha de comando existente no Linux.

2.1. Copie o código dos diferentes exemplos apresentados na introdução e teste-os.

2.1.1. Qual a diferença entre o funcionamento de um processo e uma *thread*?

2.2. Considere o seguinte programa:

```
import os, sys
n=10
try:
   os.fork()
   n+=1
   print ("hello, n=", n)
except OSError as e:
   print >>sys.stderr, "fork failed ", e.errno, "-", e.strerror
   sys.exit(1)
```

a) Indique quantos processos são criados durante a execução deste programa. Justifique.

b) Indique os resultados que são escritos para *stdout*.

c) Altere o programa de modo que o processo pai escreva para *stdout* o dobro da variável `var` e que o processo filho escreva metade.

2.3. Considere o seguinte programa, que foi executado numa diretoria onde não existe nenhum ficheiro cujo nome começa por *TMP*:

```
import os, sys

try:
   os.fork()
   os.fork()
   os.fork()
   file=open("TMP"+str(os.getpid()),'w')
   file.close()

except OSError as e:
   print >>sys.stderr, "fork failed ", e.errno, "-", e.strerror
   sys.exit(1)
```

a) Após ter terminado, quantos ficheiros começados por *TMP* passarão a existir? Justifique.

b) Diga que alterações deve fazer ao código mostrado para garantir que no fim da execução existam exatamente quatro ficheiros cujo nome comece por "*TMP*". **Só pode retirar coisas do programa, e não pode acrescentar mais nada.**

c) Alterando agora o código à sua vontade, mas utilizando a forma que achar mais fácil, garanta que no fim da execução existirão obrigatoriamente 8 ficheiros, sendo que cada ficheiro deverá ter sido criado por um processo diferente. No fim, os nomes dos ficheiros deverão ser:

TMP1, TMP2, TMP3, TMP4, TMP5, TMP6, TMP7, TMP8

2.4. Considere o seguinte programa:

```
import os, time

sum=0

def calc_sum(num):
    global sum
    for i in range(num+1):
        sum += i
    print ("soma na funcao: ", str(sum))

n=5

try:
    pid = os.fork()
    if pid == 0:
        calc_sum(n)
    else:
        os.wait()

    print ("PID = ", str(os.getpid()), ". Soma = ", str(sum))

except OSError as e:
    print >>sys.stderr, "fork failed ", e.errno, "-",
e.strerror
    sys.exit(1)
```

a) Comente o resultado da execução deste programa.

b) Esta versão do programa cria um novo processo para correr a função `calc_sum`. Altere este programa de forma que, em vez de criar um novo processo, o programa crie uma *thread* para executar a mesma função. Comente o resultado da execução deste novo programa.

c) Altere o programa desenvolvido em b) de forma a, novamente, criar um processo para correr a função `calc_sum`. Desta vez, use a API do módulo *multiprocessing* para o efeito. Comente também o resultado da execução deste programa.

2.5. Considere o seguinte *shell script*:

```
#!/bin/bash
ls | wc -w
```

a) Crie um programa em Python que crie um novo processo para executar este script.

2.6. Tendo um *array* de 1000 posições e preenchido com números de 1 a 1000 aleatórios e não repetidos, elabore um programa que crie 5 processos e:

- Dado um número, procurar esse número no *array*;
- Cada processo *filho*, procura em 200 posições do *array*;
- Um processo ao encontrar o número, deve imprimir a posição do *array* onde este se encontra. Também deve "retornar" como valor de saída o número do processo (1, 2, 3, 4, 5);
- Os processos que não encontrarem o número devem "retornar" como *valor de saída* o valor 0;
- O processo *pai* tem de esperar que todos os *filhos* terminem e imprimir o número do processo onde esse número foi encontrado (1, 2, 3, 4, 5) que encontrou o número.

2.7. Elabore o programa anterior com recurso a *threads*.