Departamento de Informática, Faculdade de Ciências da Universidade de Lisboa

Sistemas Operativos

2021/22

Guião de aula
Comunicação entre processos

---

1. Introdução

Na aula de processos estudámos a criação, execução e terminação de processos. Em particular, vimos como a chamada ao sistema `fork()` permite criar um novo processo-filho completamente independente do processo-pai. Isto resulta do facto de o pai e o filho não partilharem nenhuma parte da memória (pilha, *heap* e dados). Uma consequência deste facto é os processos não poderem comunicar entre si sem mecanismos adicionais. Nesta aula vamos estudar dois mecanismos – a memória partilhada e a troca de mensagens – que permitem que esta comunicação possa ocorrer.

1.1. Comunicação entre processos

Os mecanismos de comunicação entre processos permitem que estes possam partilhar dados e informação. Há dois modelos fundamentais de comunicação entre processos:

1. **Troca de mensagens**. Neste modelo a comunicação entre processos é conseguida através da troca de mensagens. Há duas operações básicas – um processo pode *enviar* uma mensagem e pode *receber* uma mensagem. Não há partilha de memória, e o *kernel* serve normalmente de intermediário no processo. A Figura 1. a) ilustra este modelo.
2. **Memória partilhada**. Neste modelo os processos trocam informação entre si através da partilha de uma região de memória. A Figura 1. b) ilustra este conceito. Reparem como o processo A e o processo B partilham uma região específica da memória ("*shared*"). Na figura, o processo A faz uma escrita para essa região (passo 1 na figura) e o processo B faz uma leitura do que foi escrito (passo 2).

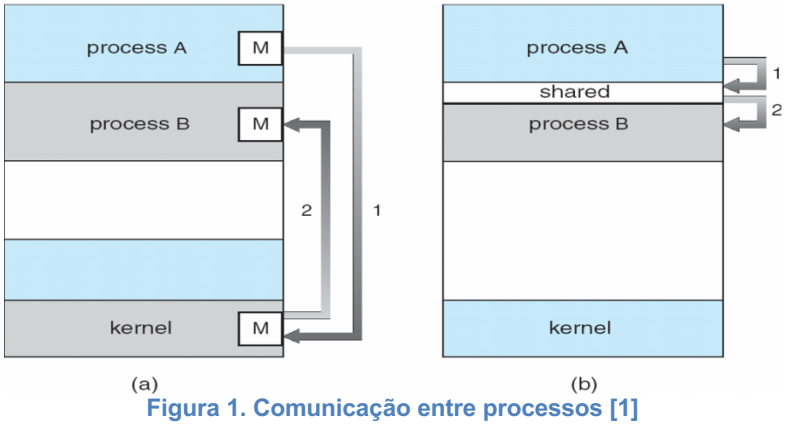


Figura 1. Comunicação entre processos [1]

A principal vantagem da memória partilhada é a velocidade da comunicação. No modelo de troca de mensagens é muitas vezes necessário fazer chamadas ao sistema sempre que se quer escrever (enviar) ou ler (receber). No caso da memória partilhada só são precisas chamadas ao sistema para criar a região partilhada.

Depois disto a leitura e escrita são simples acessos à memória, o que é muito mais rápido. A principal desvantagem da memória partilhada é ter de se assegurar que os processos não escrevem naquela região de memória simultaneamente. É necessário por isso implementar mecanismos de sincronização para acesso à memória partilhada. Este será o tópico da próxima aula.

1.2. Comunicação por memória partilhada em Python

O mecanismo de memória partilhada permite que vários processos partilhem uma mesma região de memória. Quando um processo altera essa região de memória, essa alteração é imediatamente visível para todos os outros processos que a partilham.

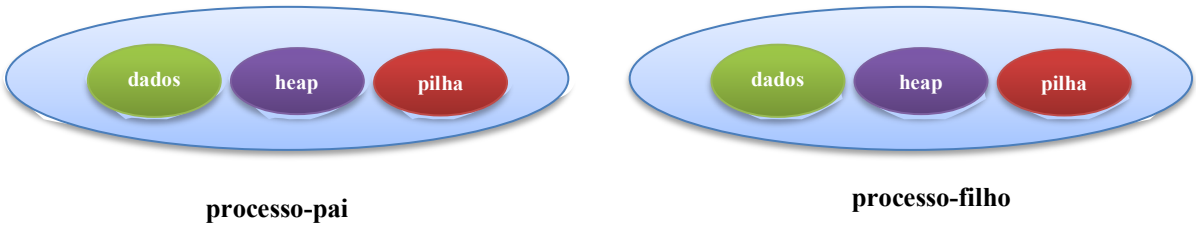
Para se perceber a utilidade da memória partilhada, considere-se o exemplo seguinte.[1]

```
import os
from multiprocessing import Process

myVar = 100

def funcao_filho():
    global myVar
    print ("Filho (PID=", str(os.getpid()), "). Mensagem do pai:", str(myVar) )
    myVar = 500
    print ("Filho (PID=", str(os.getpid()), "). Mensagem para pai:", str(myVar) )


print ("Pai (PID=", str(os.getpid()), "). Mensagem para filho: ", str(myVar) )

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

print ("Pai (PID=", str(os.getpid()), "). Mensagem do filho: ", str(myVar) )
print ("Pai (PID=", str(os.getpid()), "). Nao consegui comunicar com o filho!" )
```

Resumidamente, o programa cria um novo processo-filho que vai alterar a variável global `myVar`. O resultado da execução deste programa é o seguinte:

```
Pai (PID=3845). Mensagem para filho: 100
Filho (PID=3846). Mensagem do pai: 100
Filho (PID=3846). Mensagem para pai: 500
Pai (PID=3845). Mensagem do filho: 100
Pai (PID=3845). Nao consegui comunicar com o filho!
```

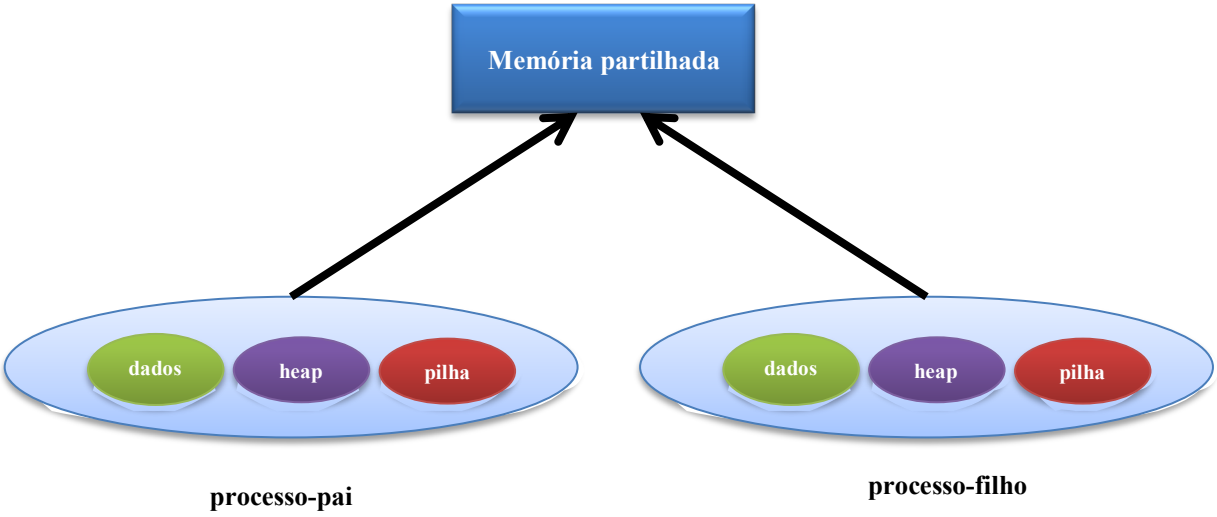
Como seria de esperar, a alteração da variável global `myVar` no processo filho não é visível no pai. Como vimos na aula de processos, isto resulta do facto de o pai e o filho não partilharem a secção de dados, como é ilustrado na figura seguinte. Assim fica impossibilitada a comunicação entre processos.



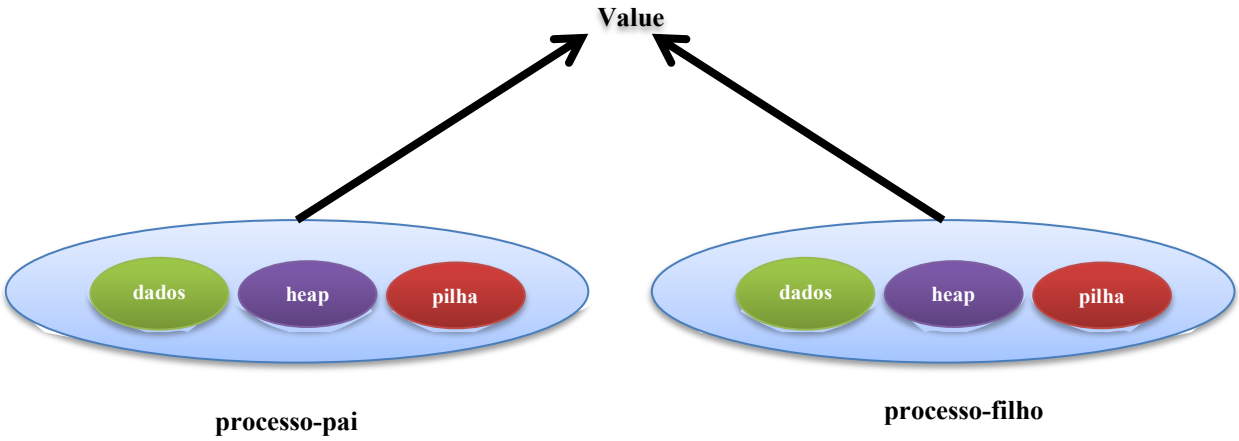
processo-pai **processo-filho**

[1] Em algumas instalações do Python 3 é preciso colocar o código do programa principal dentro de um if – "if __name__ == '__main__':" – para garantir que o processo seja corretamente iniciado antes de criar outros processos.

Como ilustrado na figura seguinte, é possível que múltiplos processos partilhem dados desde que estes sejam armazenados numa região de memória partilhada.



O módulo `multiprocessing` permite criar uma região desse tipo através das funções `Value()` ou `Array()`. A função `Value()` cria um objeto único em memória partilhada.



Múltiplos processos podem, assim, comunicar escrevendo e lendo dados deste objeto. O código seguinte apresenta o exemplo anterior, mas alterado para permitir comunicação entre os processos pai e filho. A vermelho assinala-se as alterações necessárias ao original.

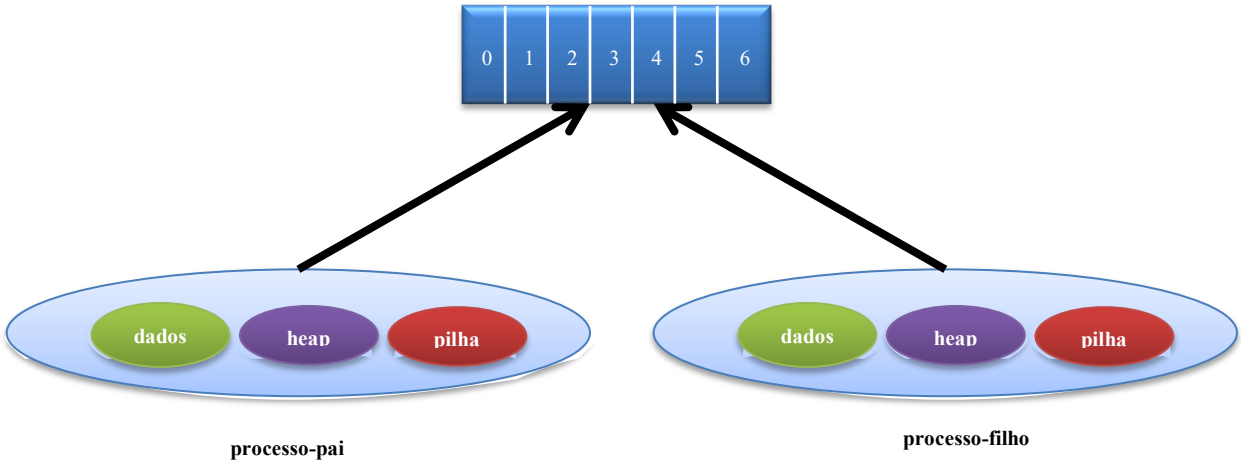
```
import os
from multiprocessing import Process, Value

myVar = Value("i",100)

def funcao_filho():
    print ("Filho (PID=", str(os.getpid()), "). Mensagem do pai: ", str(myVar.value))
    myVar.value = 500
    print ("Filho (PID=", str(os.getpid()), "). Mensagem para pai: ", str(myVar.value))


print ("Pai (PID=", str(os.getpid()), "). Mensagem para filho: ", str(myVar.value))

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

print ("Pai (PID=", str(os.getpid()), "). Mensagem do filho: ", str(myVar.value))
print ("Pai (PID=", str(os.getpid()), "). Consegui comunicar com o filho!")
```

O resultado da execução deste código é apresentado a seguir.

```
Pai (PID=3928). Mensagem para filho: 100
Filho (PID=3929). Mensagem do pai: 100
Filho (PID=3929). Mensagem para pai: 500
Pai (PID=3928). Mensagem do filho: 500
Pai (PID=3928). Ja consegui comunicar com o filho!
```

Como se pode verificar, o processo-pai e o processo-filho já conseguem comunicar. O filho alterou a variável partilhada, e essa alteração ficou imediatamente visível no processo-pai.

Uma limitação desta solução é permitir partilhar apenas um objeto. Para partilhar mais objetos podemos usar a classe `Array` do mesmo módulo. Isto permite partilhar um *array* de objetos, tal como representado na imagem seguinte (apresenta-se um *array* com espaço para 7 elementos).



No código seguinte ilustramos o uso da função `Array()`.

```
import os
from multiprocessing import Process, Array

myArray = Array("i",10)

def funcao_filho(myArray):

    print ("Filho (PID=", str(os.getpid()), "): Deixa ver array do pai -> " , end="")
    for i in range(10):
        print(myArray[i], " ", end="")
    print("")

    print ("Filho (PID=", str(os.getpid()), "): Vou mandar esta mensagem ao pai -> ",
end="")
    for i in range(10):
        myArray[i] = i+1
        print (myArray[i], " ", end="")
    print("")


print ("Pai (PID=", str(os.getpid()), "): Vou mandar este array ao filho -> ", end="")
for i in range(10):
    myArray[i] = 0
    print (myArray[i], " ", end="")
print("")

newT = Process(target=funcao_filho, args=(myArray,))
newT.start()
newT.join()

print ("Pai (PID=", str(os.getpid()), "): Deixa ver mensagem do filho -> ", end="")
for i in range(10):
    print (myArray[i], " ", end="")
print("")
```

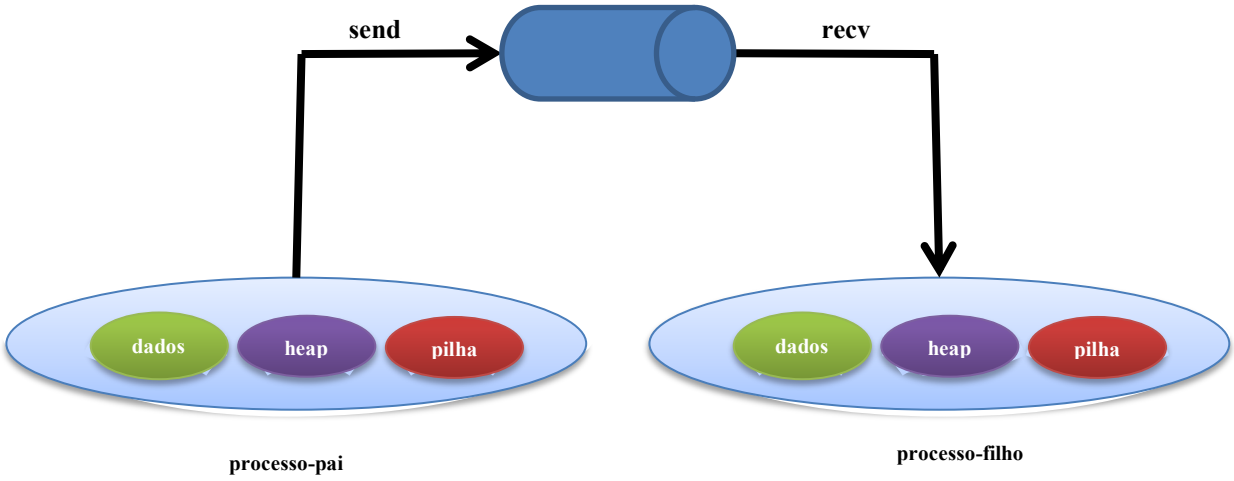
O resultado da execução deste código é apresentado a seguir.

```
Pai (PID=3952): Vou mandar este array ao filho ->  0 0 0 0 0 0 0 0 0 0
Filho (PID=3953): Deixa ver array do pai ->  0 0 0 0 0 0 0 0 0 0
Filho (PID=3953): Vou mandar esta mensagem ao pai ->  1 2 3 4 5 6 7 8 9 10
Pai (PID=3952): Deixa ver mensagem do filho ->  1 2 3 4 5 6 7 8 9 10
```

Como se pode verificar, o pai e o filho partilham a estrutura `myArray`.

1.3. Comunicação por troca de mensagens em Python

O módulo `multiprocessing` inclui dois canais de comunicação entre processos: a função `Pipe()` e a classe `Queue`. Assim, dois processos podem comunicar através de troca de mensagens. Nesta secção vamos apresentar a solução para o último exemplo apresentado usando estes canais. Começamos pela função `Pipe()`, abstração que é representada na figura seguinte.



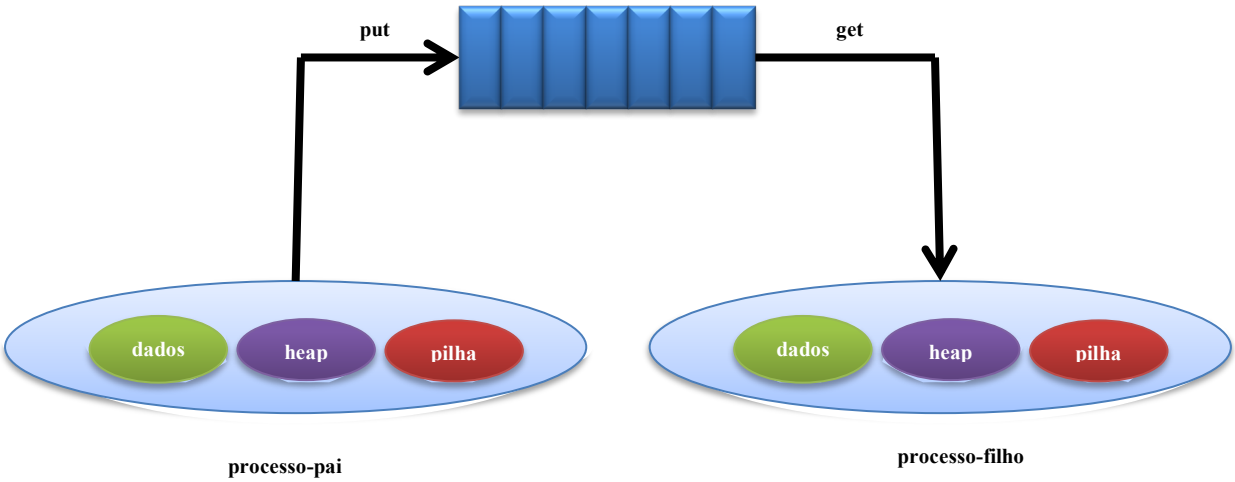
A função `Pipe()` retorna um par de objetos ligados por um *pipe* (um "cano") que permite comunicação bidirecional. Os dois objetos retornados na chamada à função representam cada um dos lados do *pipe*. Os métodos fundamentais destes objetos são o `send()` e o `recv()`.

O código apresentado na próxima página ilustra a utilização de *pipes*. Mais uma vez, as alterações necessárias ao código são apresentadas a vermelho. Optou-se por usar uma lista para trocar a informação entre o processo pai e o processo filho.

```
import os
from multiprocessing import Process, Pipe

pipe_pai, pipe_filho = Pipe()

def funcao_filho(pipe_filho):
    lista = pipe_filho.recv()
    print ("Filho (PID=", str(os.getpid()), "): Deixa ver lista do pai -> ", end="")
    for i in range(10):
        print (lista[i], " ", end="")
    print("")

    print ("Filho (PID=", str(os.getpid()), "): Vou mandar esta mensagem ao pai -> ",
end="")
    for i in range(10):
        lista[i] = i+1
        print (lista[i], " ", end="")
    print("")

    pipe_filho.send(lista)

lista=[]
print ("Pai (PID=", str(os.getpid()), "): Vou mandar esta lista ao filho: ", end="")
for i in range(10):
    lista.append(0)
    print (str(lista[i]), " ", end="")
print("")

newT = Process(target=funcao_filho, args=(pipe_filho,))
newT.start()

pipe_pai.send(lista)
lista = pipe_pai.recv()

print ("Pai (PID=", str(os.getpid()), "): Deixa ver mensagem do filho -> ", end="")
for i in range(10):
    print (lista[i], " ", end="")
print("")

newT.join()
```

Há uma diferença importante neste exemplo, que é necessário desde já esclarecer. Quando se utilizava a memória partilhada para comunicação entre processos, o processo-filho já podia ter terminado, mas o processo-pai ainda podia aceder à memória partilhada. No caso da comunicação por troca de mensagens **ambos os processos têm de estar ativos** para ser possível estabelecer comunicação. Isto significa que esta comunicação tem acoplamento temporal, e.g., os intervenientes têm de estar ativos ao mesmo tempo para se comunicarem. Esta é uma diferença muito importante entre os dois métodos.

A segunda abstração de comunicação usando troca de mensagens que apresentamos é a fila de espera. Para tal, usamos a classe `Queue` do módulo `multiprocessing`. Na figura seguinte representamos uma fila de espera com 7 posições livres. Os métodos fundamentais são o `put`, que permite colocar um novo elemento na fila, e o `get`, que permite retirar o elemento que está na frente da fila.



O código seguinte ilustra o uso de uma fila de espera para atingir os mesmos objetivos do programa anterior.

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

q = Queue()

def funcao_filho(q):
    lista = q.get()
    print ("Filho (PID=" + str(os.getpid()) + "): Deixa ver lista do pai -> ", end="")
    for i in range(10):
        print (lista[i], " ", end="")
    print("")

    print ("Filho (PID=" + str(os.getpid()) + "): Vou mandar esta mensagem ao pai -> ",
end="")
    for i in range(10):
        lista[i] = i+1
        print (lista[i], " ", end="")
    print("")

    q.put(lista)

lista=[]
print ("Pai (PID=" + str(os.getpid()) + "): Vou mandar esta lista ao filho: ", end="")
for i in range(10):
    lista.append(0)
    print (str(lista[i]), " ", end="")
print("")

newT = Process(target=funcao_filho, args=(q,))
newT.start()

q.put(lista)
time.sleep(1) #para obrigar a comutar para o filho
lista = q.get()

print ("Pai (PID=" + str(os.getpid()) + "): Deixa ver mensagem do filho -> ", end="")
for i in range(10):
    print (lista[i], " ", end="")
print("")

newT.join()
```

Nota: Neste exemplo foi necessário adicionar um `sleep()` *para "obrigar" o CPU a passar a execução para o filho. Esta é uma forma de ultrapassar os problemas de sincronização que existiriam neste exemplo e que serão tema de discussão das próximas aulas. (Experimentem remover o* `sleep()` *para ver o que acontece, e meditem sobre o resultado.)*

O resultado da execução dos dois últimos programas é, como esperado, exatamente o mesmo:

```
Pai (PID=3988): Vou mandar esta lista ao filho ->  0 0 0 0 0 0 0 0 0 0
Filho (PID=3989): Deixa ver lista do pai ->  0 0 0 0 0 0 0 0 0 0
Filho (PID=3989): Vou mandar esta mensagem ao pai ->  1 2 3 4 5 6 7 8 9 10
Pai (PID=3988): Deixa ver mensagem do filho ->  1 2 3 4 5 6 7 8 9 10
```

2. Exercícios propostos

2.1. Desenvolva um programa que:

a) peça ao utilizador que insira 5 números

b) os coloque numa lista

c) calcule a sua soma.

2.2. Altere o programa anterior de modo que a soma seja calculada num processo filho. Esse novo processo deve enviar o resultado, usando memória partilhada, ao processo pai, que o deve apresentar no ecrã.

Obs: como saber se foi criado um processo filho? – fazer print do pid no processo pai e no processo filho – estes valores têm de ser diferentes.

2.3. Faça um programa que imprima a tabuada de um número introduzido pelo utilizador. Para tal, o processo pai deve pedir um número ao utilizador, e um processo filho deve colocar o resultado da tabuada num array partilhado com o processo pai. Um possível resultado da execução do programa é apresentado a seguir:

```
Insira numero: 3
3x1=3
3x2=6
3x3=9
...
3x10=30
```

2.4. Resolva o exercício 2.3 usando:

a) um pipe;

b) uma queue.

3. Exercícios complementares

Os alunos que terminem com sucesso os exercícios anteriores são convidados agora a resolver um novo exercício relacionado com memória partilhada.

3.1. Desenvolva um programa que inicializa um valor inteiro numa zona de memória partilhada a zero, e que depois cria 500 processos que vão incrementar esse número. O processo pai deve esperar pela conclusão de todos os processos criados e imprimir o número final. Corra o programa várias vezes.

Qual o resultado que espera retornar da execução deste programa? E qual o resultado que efetivamente obtém? Comente.

4. Exercícios extra-aula

Os alunos que terminem com sucesso os exercícios anteriores devem tentar resolver os mesmos exercícios usando threads em vez de processos.

5. Bibliografia e outro material de apoio

[1] "*Operating System Concepts*". Abraham Silberschatz, Peter B. Galvin, Greg Gagne, 9th edition, 2014.