1. Conceitos e Fundamentos

Antes de entrarmos nos exemplos práticos, vamos revisar alguns conceitos importantes.

1.1 Fork e Criação de Processos

Como vimos em sala, o padrão definido pelo POSIX, utiliza a função `os.fork`, onde o processo atual é duplicado, gerando um processo pai e um filho. O retorno para o processo pai é o PID do filho, enquanto para o filho é 0. Essa separação permite que cada processo execute seu próprio fluxo de execução. **Os métodos utilizados no primeiro trabalho também fazem o fork, mas executam mais coisas na sequência, portanto, estamos vendo agora um API do Python de nível mais baixo.**

1.2 Comunicação entre Processos (IPC)

IPC (Comunicação entre Processos) é um conjunto de mecanismos que permite que processos independentes troquem dados entre si. As principais técnicas de IPC incluem:

- Pipes (canalizações)
- Queues (filas) com a biblioteca multiprocessing
- Sockets
- Memória compartilhada

Cada método possui suas características, vantagens e desafios, como sincronização, bloqueios e deadlocks. A escolha da técnica depende do tipo de dados a serem comunicados e da forma como os processos interagem. **Nesse primeiro momento, estaremos mais preocupados com os Pipes.**

2. Método `os.fork`

Cada exemplo a seguir mostra a criação de processos e a forma de diferenciar o que o pai e o filho executar, conforme os conceitos estudados.

2.1 Exemplo (Processo Básico com Fork)

Este exemplo cria um processo filho. O pai imprime uma mensagem diferente da do filho.

```
import os
import sys

pid = os.fork()
```

```
if pid == 0:
    print("Processo filho: executando sua parte.")
    sys.exit(0)
else:
    print("Processo pai: executando sua parte.")
```

O que acontece:

- O fork é executado e o código se ramifica em dois processos.
- No filho (pid == 0), é exibida uma mensagem e o processo finaliza.
- No pai (pid diferente de 0), é exibida outra mensagem.

Notem que dessa forma não fazemos nenhuma sincronização entre os processos, apenas os deixamos rodar em paralelo.

2.2 Exemplo (Fork com Sincronização (usando wait))

Neste exemplo, o pai espera o término do filho, verificando seu código de saída.

```
import os
import sys

pid = os.fork()
if pid == 0:
    print("Filho: iniciando execução.")
    sys.exit(42)
else:
    pid, status = os.wait()
    exit_code = os.WEXITSTATUS(status)
    print("Pai: o filho terminou com o código de saída", exit_code)
```

O que acontece:

- Após o fork, o filho imprime sua mensagem e encerra com o código 42.
- O pai usa os.wait() para esperar o término do filho, capturando seu status.
- Em seguida, o pai extrai o código de saída e o exibe.

Nesse exemplo, conseguimos fazer o processo pai esperar o processo filho terminar uma tarefa de computação, utilizando algoritmos de sleep/wake-up.

2.3 Exemplo (Cadeia de Processos (Processo Neto))

Nesse exemplo, o pai cria um filho que, por sua vez, cria um neto. Cada processo exibe uma mensagem indicando sua posição na hierarquia.

```python
import os
import sys

pid = os.fork()
if pid == 0:
    pid2 = os.fork()
    if pid2 == 0:
        print("Neto: executando.")
        sys.exit(0)
    else:
        os.wait()
        print("Filho: finalizando após o neto.")
        sys.exit(0)
else:
    os.wait()
    print("Pai: todos os processos filhos finalizaram.")
```

O que acontece:

- O processo pai cria um filho; esse filho, por sua vez, cria um neto.
- O neto imprime sua mensagem e termina.
- O filho espera o término do neto antes de imprimir sua mensagem e encerrar.
- O pai, por fim, espera o filho terminar para imprimir sua mensagem.

3. Comunicação Básica com Pipe

Em Python, o comando `os.pipe` nos cria dois descritores, sendo eles:

- **r**: o descritor de leitura (read), de onde os dados são lidos;
- **w**: o descritor de escrita (write), onde os dados são enviados.

Além disso, utilizamos outros métodos das bibliotecas do Python, sendo elas:

- **os.close(descritor)**: fecha um dos lados do pipe para evitar vazamentos de recursos e indicar que aquele lado não será utilizado;
- **os.write(descritor, dados)**: escreve dados (em formato binário) no lado de escrita do pipe;
- **os.read(descritor, tamanho)**: lê até o número especificado de bytes do lado de leitura do pipe;
- **decode()**: os pipes enviam bytes e não strings, portanto, convertemos bytes lidos para uma string, utilizando a codificação padrão (geralmente UTF-8).

Este exemplo mostra como um processo pai envia uma mensagem ao filho utilizando um pipe criado com os.pipe.

```
import os

r, w = os.pipe()
pid = os.fork()
if pid == 0:
   os.close(w)
   data = os.read(r, 100)
   print("Filho recebeu:", data.decode())
   os.close(r)
else:
   os.close(r)
   os.write(w, b"Mensagem do pai para o filho")
   os.close(w)
```

O que acontece:

- **os.pipe()**: Cria um pipe, retornando uma tupla com dois descritores de arquivo:
  - O primeiro elemento (r) é o descritor para leitura.
  - O segundo elemento (w) é o descritor para escrita.
- **os.fork()**: Cria um novo processo. O processo filho retorna 0 e o pai recebe o PID do filho.
- **os.close(descritor):** Fecha o descritor especificado. No exemplo:
  - No processo filho, fecha-se o lado de escrita (w) pois ele apenas precisa ler.
  - No processo pai, fecha-se o lado de leitura (r) pois ele apenas precisa escrever.
- **os.write(descritor, dados)**: Escreve dados binários no descritor de escrita. No exemplo, o pai envia uma mensagem codificada em bytes.
- **os.read(descritor, tamanho):** Lê até o número especificado de bytes a partir do descritor de leitura. Aqui, lê até 100 bytes.
- **data.decode():** Converte os bytes lidos para uma string, utilizando a codificação padrão (normalmente UTF-8), para que possam ser exibidos de forma legível.

4. Pipes Nomeados

Experimente o código abaixo abrindo em dois terminais, no primeiro passe o parâmetro A e no segundo o B.

```
python3 codigo.py A
python3 codigo.py B
```

```
import os
import sys
import threading
```

```
fifo_A_to_B = "fifo_A_to_B"
fifo_B_to_A = "fifo_B_to_A"

role = sys.argv[1] if len(sys.argv) > 1 else None
if role not in ["A", "B"]:
    print("Usage: python chat.py [A|B]")
    sys.exit(1)

if not os.path.exists(fifo_A_to_B):
    os.mkfifo(fifo_A_to_B)
if not os.path.exists(fifo_B_to_A):
    os.mkfifo(fifo_B_to_A)

if role == "A":
    input_fifo = fifo_B_to_A
    output_fifo = fifo_A_to_B
else:
    input_fifo = fifo_A_to_B
    output_fifo = fifo_B_to_A

def read_messages():
    with open(input_fifo, "r") as fifo_in:
        while True:
            line = fifo_in.readline()
            if not line:
                continue
            if line.strip() == "exit":
                print("\nPeer has exited. Exiting chat.")
                os._exit(0)
            print("\nPeer:", line.strip())

reader_thread = threading.Thread(target=read_messages, daemon=True)
reader_thread.start()

with open(output_fifo, "w") as fifo_out:
    while True:
        message = input("You: ")
        fifo_out.write(message + "\n")
        fifo_out.flush()
        if message.strip() == "exit":
            break
```

O código implementa um chat entre dois processos diferentes usando pipes nomeados (FIFOs), que são arquivos especiais no sistema de arquivos usados para comunicação entre

processos independentes. Isso contrasta com os pipes anônimos, que funcionam apenas entre processos com relação direta, como pai e filho criados via fork(). Neste caso, cada processo é iniciado manualmente em um terminal diferente, passando o argumento A ou B para definir seu papel na conversa.

Dois arquivos FIFO são utilizados: um para enviar mensagens de A para B e outro de B para A. O programa cria esses arquivos se eles ainda não existirem e define, com base no papel do processo, qual será o canal de entrada e qual será o de saída. A leitura das mensagens recebidas é feita em uma thread separada, o que permite que o processo continue aceitando entradas do usuário enquanto escuta mensagens do outro processo. A escrita ocorre linha por linha, e cada mensagem é enviada imediatamente com flush() para não ficar presa no buffer. O chat continua até que um dos dois envie a palavra exit, momento em que ambos os processos encerram a execução.

Ao final da conversa, os arquivos FIFO permanecem no sistema de arquivos e devem ser removidos com rm. Esse exemplo ilustra bem como dois processos independentes podem se comunicar de forma simples, utilizando apenas recursos básicos do sistema operacional, como arquivos e threads. Ele reforça conceitos importantes como comunicação entre processos, concorrência e uso prático de FIFOs no ambiente Unix.

Exercícios

Após os exemplos anteriores, seguem faça os dois exercícios a seguira para fixar o conceito de criação de processos utilizando apenas o método fork.

Exercício 1: Impressão Alternada de Mensagens

Objetivo: Criar um programa onde o processo pai e o processo filho se alternam para imprimir mensagens em sequência.

Requisitos:

- O pai e o filho devem imprimir, alternadamente, mensagens para indicar a sua vez (por exemplo, "Pai fala" e "Filho fala") em 5 iterações.
- Utilize os.fork() para criar o filho e os.wait() no pai para sincronizar as execuções.
- Certifique-se de que as mensagens não se sobreponham, controlando a ordem de execução.

Exercício 2: Contagem Paralela

Objetivo: Desenvolver um programa que utilize o fork para criar dois processos, onde:

- O processo pai conta de 1 a 5, com uma pausa de 1 segundo entre cada número.

- O processo filho conta de 6 a 10, também com uma pausa de 1 segundo.

Requisitos:

- Use os.fork() para criar o processo filho.
- Utilize time.sleep(1) para criar a pausa entre as iterações.
- Garanta que as contagens ocorram de forma independente e que o pai aguarde o término do filho antes de finalizar.