

Universidade Federal do Piauí - UFPI

Campus Senador Helvídio Nunes Barros - CSHNB

Sistemas de Informação - Sistemas Distribuídos - prof. Rayner Gomes

Aluna: Vandirleya Barbosa da Costa

Tema: Processos

3º Lista de Exercícios de Fixação - 3º Semana

Atenção: Para todas as perguntas, cite o *site*, ou o material utilizado como fonte da pesquisa. Como uma atividade de formação intelectual, não copie e cole, o intuito não é saber sua capacidade de apertar CTRL-C e CTRL-V, mas sua capacidade de resumir, explicar e transmitir um novo conteúdo.

1. Em relação ao capítulo de processos explique os conceitos:

a. atomicidade

Refere-se à propriedade de uma operação ser indivisível, ou seja, ela ocorre completamente ou não ocorre. Em sistemas distribuídos, a atomicidade é importante para garantir que operações sejam realizadas de forma consistente, mesmo em caso de falhas.

b. exclusão mútua

É o princípio pelo qual apenas um processo pode acessar uma seção crítica de código de cada vez. A exclusão mútua é essencial para evitar condições de corrida e inconsistências de dados em sistemas com múltiplos processos.

c. semáforos

São variáveis utilizadas para controlar o acesso a recursos compartilhados em sistemas de processamento paralelo ou distribuído. Os semáforos permitem a sincronização entre processos, podendo ser binários (valores 0 ou 1) ou contadores, que permitem múltiplos acessos simultâneos.

d. região crítica

É uma parte do código onde um recurso compartilhado é acessado. O conceito de região crítica está relacionado com a exclusão mútua, pois apenas um processo pode estar em uma região crítica de cada vez para evitar inconsistências.

2. Cite um exemplo de uma computação paralela e uma concorrente.

- *Computação Paralela:* Calcular a média de um conjunto de números dividindo-o em várias partes e processando essas partes simultaneamente em múltiplos núcleos

de um processador.

- **Computação Concorrente:** Processar múltiplas requisições de um servidor web simultaneamente, onde cada requisição pode ser tratada de forma intercalada por meio de threads.

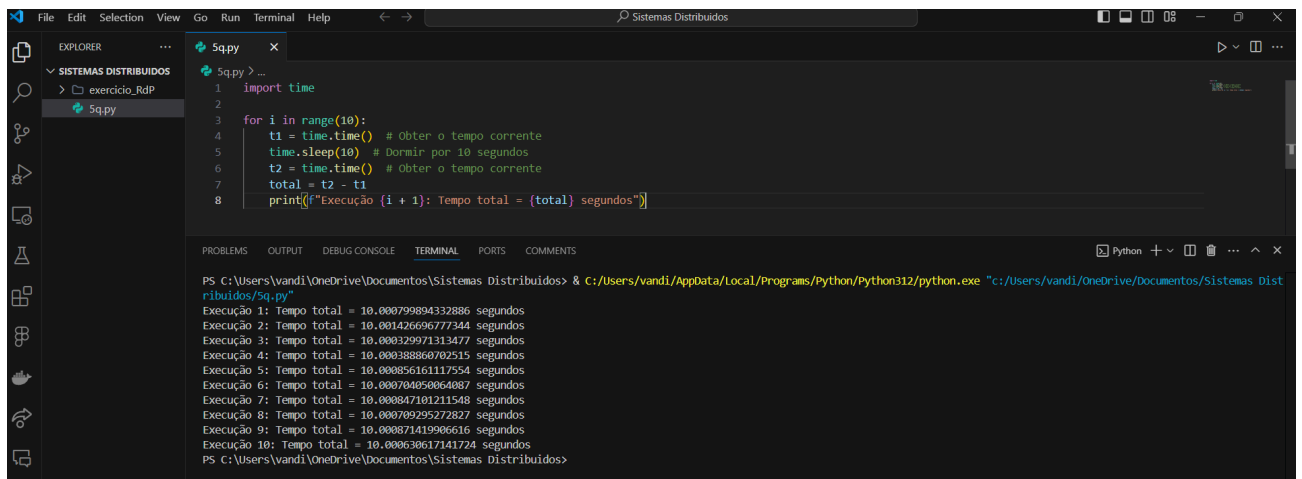
3. O que caracteriza um processo distribuído?

Um processo distribuído é caracterizado por ser executado em um sistema distribuído, onde múltiplos processos podem rodar em diferentes nós de uma rede, comunicando-se por meio de mensagens. Esses processos não compartilham memória e precisam lidar com a comunicação e sincronização entre si para alcançar objetivos comuns, como a execução de uma tarefa distribuída ou a manutenção de uma base de dados replicada.

4. Baseado na solução de Peterson, explique o porquê da solução funcionar.

A solução de Peterson funciona garantindo que dois processos não entrem simultaneamente na sua região crítica. Ela usa duas variáveis: **flag** e **turn**. A variável **flag[i]** indica se o processo **i** quer entrar na região crítica, e a variável **turn** indica qual processo deve ter prioridade. O funcionamento é baseado em um protocolo de espera ocupada, onde cada processo sinaliza sua intenção de entrar e cede a vez ao outro se necessário. Assim, a exclusão mútua é garantida, pois apenas um processo por vez poderá acessar a região crítica.

5. Faça um programa em C/Python/Java, seu programa deve fazer os seguintes passos: (i) obter o tempo corrente (tempo t1); (ii) dormir por 10 segundos; (iii) obter o tempo corrente (tempo t2). Execute o programa 10 vezes, para cada execução registre o tempo total (total = t2 - t1). Na média o valor total é maior ou menor que 10s? Se há alguma diferença, qual a explicação?



The image shows a VS Code editor window with a file named 'Sq.py' open. The script is a Python program that imports the 'time' module and runs a loop 10 times. In each iteration, it records the current time, sleeps for 10 seconds, records the time again, and prints the total time taken for that iteration. The terminal output shows the results of these 10 iterations, with each total time being slightly more than 10 seconds.

```
1 import time
2
3 for i in range(10):
4     t1 = time.time() # Obter o tempo corrente
5     time.sleep(10) # Dormir por 10 segundos
6     t2 = time.time() # Obter o tempo corrente
7     total = t2 - t1
8     print(f"Execução (i + 1): Tempo total = {total} segundos")
```

Terminal Output:

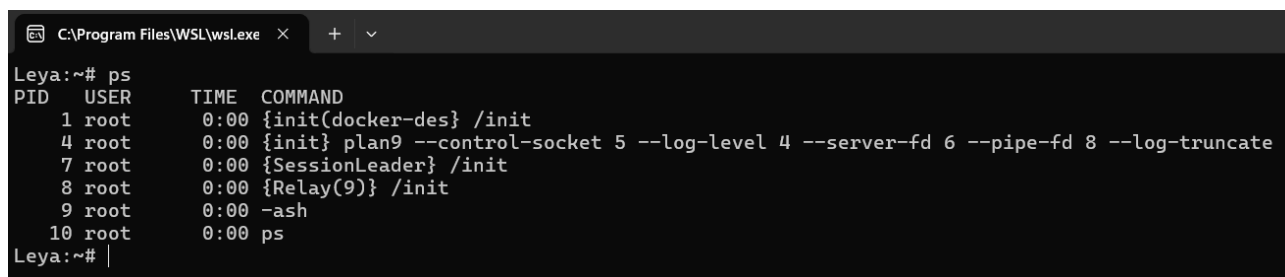
```
PS C:\Users\vandi\OneDrive\Documentos\Sistemas Distribuidos> & C:/Users/vandi/AppData/Local/Programs/Python/Python312/python.exe "c:/Users/vandi/OneDrive/Documentos/Sistemas Distribuidos/Sq.py"
Execução 1: Tempo total = 10.000799894332896 segundos
Execução 2: Tempo total = 10.001426696777344 segundos
Execução 3: Tempo total = 10.000329971313477 segundos
Execução 4: Tempo total = 10.000388860702515 segundos
Execução 5: Tempo total = 10.000856161117554 segundos
Execução 6: Tempo total = 10.000784050064887 segundos
Execução 7: Tempo total = 10.000847101211548 segundos
Execução 8: Tempo total = 10.000709295272827 segundos
Execução 9: Tempo total = 10.000871419906616 segundos
Execução 10: Tempo total = 10.000630617141724 segundos
PS C:\Users\vandi\OneDrive\Documentos\Sistemas Distribuidos>
```

Na maioria dos testes, o valor total tende a ser ligeiramente maior que 10 segundos. Isso ocorre devido a atrasos no agendamento dos processos pelo sistema operacional e na precisão do temporizador. Além disso, fatores como a latência da chamada `time.sleep()` podem causar pequenas variações. A diferença geralmente está na ordem de milissegundos.

6. Alguns comandos no linux nos ajudam a visualizar e obter informações dos processos, pesquise, explique e demonstre a função dos comandos:

a. ps

O comando `ps` é usado para listar os processos em execução no sistema. Ele fornece informações como o identificador do processo (PID), o nome do processo, o usuário que o está executando, e o uso de recursos. Com opções adicionais, como `ps aux` ou `ps -ef`, você pode obter uma lista completa dos processos em execução com detalhes sobre o uso de CPU e memória.



The image shows a terminal window with the command 'ps' executed. The output is a table with columns for PID, USER, TIME, and COMMAND. The processes listed are /init, plan9, SessionLeader, Relay(9), -ash, and ps.

```
C:\Program Files\WSL\wsl.exe x + v
Leya:~# ps
PID  USER    TIME  COMMAND
1    root     0:00  {init(docker-des} /init
4    root     0:00  {init} plan9 --control-socket 5 --log-level 4 --server-fd 6 --pipe-fd 8 --log-truncate
7    root     0:00  {SessionLeader} /init
8    root     0:00  {Relay(9)} /init
9    root     0:00  -ash
10   root     0:00  ps
Leya:~#
```

b. top

O comando `top` exibe uma lista dinâmica dos processos em execução no sistema, atualizada em tempo real. Ele mostra o uso de CPU, memória, tempo de execução e outras métricas de cada processo. O comando é útil para monitoramento de recursos e para identificar processos que consomem muitos recursos.

```
C:\Program Files\WSL\wsl.exe x + v
Mem: 489940K used, 3288612K free, 2224K shrd, 3252K buff, 130320K cached
CPU:  0% usr  0% sys  0% nic 99% idle  0% io  0% irq  0% irq
Load average: 0.04 0.01 0.00 1/195 12
PID  PPID  USER  STAT  VSZ  %VSZ  CPU  %CPU  COMMAND
  8    7  root   S    2300  0%   11  0%  {Relay(9)} /init
  7    1  root   S    2284  0%   11  0%  {SessionLeader} /init
  4    1  root   S    2280  0%    6  0%  {init} plan9 --control-socket 5 --log-level 4 --server-fd 6 --pipe-fd 8 --
  1    0  root   S    2280  0%    9  0%  {init(docker-des)} /init
  9    8  root   S    1696  0%    2  0%  -ash
 12   12  root   R    1624  0%    4  0%  top
```

c. kill

O comando **kill** é usado para enviar um sinal a um processo específico. O sinal mais comum é o **SIGTERM** (sinal 15), que solicita ao processo que encerre graciosamente, ou o **SIGKILL** (sinal 9), que força a finalização imediata do processo. Para usá-lo, é necessário fornecer o PID do processo.

```
C:\Program Files\WSL\wsl.exe x + v
Leya:~# ps
PID  USER  TIME  COMMAND
  1  root   0:00  {init(docker-des)} /init
  4  root   0:00  {init} plan9 --control-socket 5 --log-level 4 --server-fd 6 --pipe-fd 8 --log-truncate
 13  root   0:00  {SessionLeader} /init
 14  root   0:00  {Relay(15)} /init
 15  root   0:00  -ash
 20  root   0:00  ps
Leya:~# kill
ash: you need to specify whom to kill
Leya:~# kill 15
Leya:~# ps
PID  USER  TIME  COMMAND
  1  root   0:00  {init(docker-des)} /init
  4  root   0:00  {init} plan9 --control-socket 5 --log-level 4 --server-fd 6 --pipe-fd 8 --log-truncate
 13  root   0:00  {SessionLeader} /init
 14  root   0:00  {Relay(15)} /init
 15  root   0:00  -ash
 21  root   0:00  ps
Leya:~# |
```

d. killall

O comando **killall** permite enviar sinais para todos os processos com um determinado nome. Por exemplo, se houver vários processos chamados "firefox" em execução, você pode encerrar todos eles de uma só vez com **killall**.

```
C:\Program Files\WSL\wsl.exe x + v
Leya:~# killall
killall: you need to specify whom to kill
Leya:~# ps
PID  USER  TIME  COMMAND
  1  root   0:00  {init(docker-des)} /init
  4  root   0:00  {init} plan9 --control-socket 5 --log-level 4 --server-fd 6 --pipe-fd 8 --log-truncate
 13  root   0:00  {SessionLeader} /init
 14  root   0:00  {Relay(15)} /init
 15  root   0:00  -ash
 24  root   0:00  ps
Leya:~#
Leya:~# killall ash
Leya:~# ps
PID  USER  TIME  COMMAND
  1  root   0:00  {init(docker-des)} /init
  4  root   0:00  {init} plan9 --control-socket 5 --log-level 4 --server-fd 6 --pipe-fd 8 --log-truncate
 13  root   0:00  {SessionLeader} /init
 14  root   0:00  {Relay(15)} /init
 15  root   0:00  -ash
 26  root   0:00  ps
Leya:~# |
```

e. nice

O comando **nice** é usado para definir a prioridade de execução de um processo. Ele permite que você inicie um novo processo com um nível de prioridade especificado. Quanto maior o valor do **nice**, menor a prioridade do processo.

A screenshot of a Windows terminal window. The title bar shows 'C:\Program Files\WSL\wsl.exe' and standard window controls. The terminal content shows a user prompt 'Leya:~#' followed by the command 'nice', which returns the value '0'. The prompt then changes to 'Leya:~# |' with a cursor.

```
C:\Program Files\WSL\wsl.exe × + v
Leya:~# nice
0
Leya:~# |
```

Bom trabalho!