TESTE DE PERFORMANCE 7

Engenharia da Computação
Projeto em Arquitetura de Computadores,
Sistemas Operacionais e Redes
Kaio Henrique Silva da Cunha
Prof.: Alcione Dolavale

Fortaleza, CE
15/11/2021

Teste de Performance 3

A aplicação começa por importar os módulos necessários para capturar informações da arquitetura do computador e para exibir interfaces gráficas.

```
# TP3

import pygame, psutil, cpuinfo, platform
from pygame.locals import *
```

Também são declaradas as funções para formatar as informações da máquina, exibir informações da CPU, utilização dos núcleos, do disco, e da memória, respectivamente. É importante observar que a captura da frequência(freq) não funcionou como esperado. Por algum motivo, na minha máquina virtual(Linux) ele estava retornando ‘None’. Acabei decidindo por enviar com uma solução temporária que evita um erro em tempo de execução.

```
def mostra_titulo():
    tela.blit(s1, (0,s1_posicao))
    texto_titulo = "Informaçoes do Computador e Monitoramento"
    font = pygame.font.Font(None, 30)
    text = font.render(texto_titulo, 1, orangesidecar)
    s1.blit(text, (100, 14))

def uso_memoria():
    mem = psutil.virtual_memory()

    larg = largura_tela - 2*20
    pygame.draw.rect(s2,cinzaclaro, (20, 40, larg, 40))
    tela.blit(s2, (0,s2_posicao))

    larg = larg*mem.percent/100
    pygame.draw.rect(s2, darkseagreen, (20, 40, larg, 40))
    tela.blit(s2, (0,s2_posicao))

    total = round(mem.total/(1024*1024*1024),2)
    texto_memoria = "Uso de Memória (Total: " + str(total) + "GB):"
    text = font.render(texto_memoria, 1, branco)
    s2.blit(text, (20, 10))
```

```
def uso_disco():
    disco = psutil.disk_usage('.')

    larg = largura_tela - 2*20
    pygame.draw.rect(s3, cinzaclaro, (20, 40, larg, 40))
    tela.blit(s3, (0, s3_posicao))

    larg = larg*disco.percent/100
    pygame.draw.rect(s3, verde, (20, 40, larg, 40))
    tela.blit(s3, (0, s3_posicao))

    total = round(disco.total/(1024*1024*1024), 2)
    texto_barra = "Uso de Disco: (Total: " + str(total) + "GB):"
    text = font.render(texto_barra, 1, branco)
    s3.blit(text, (20, 10))

def mostra_info_cpu():
    mostra_texto(s4, "Nome Processador:", "brand", 10)
    mostra_texto(s4, "Arquitetura:", "arch", 30)
    mostra_texto(s4, "Palavra (bits):", "bits", 50)
    mostra_texto(s4, "Frequência (MHz):", "freq", 70)
    mostra_texto(s4, "Núcleos (físicos):", "nucleos", 90)
    tela.blit(s4, (0, s4_posicao))
```

```
def mostra_texto(surface, nome, chave, pos_y):
    text = font.render(nome, True, branco)
    surface.blit(text, (20, pos_y))
    if chave == "freq":
        freq = psutil.cpu_freq()
        if freq:
            resultado = freq.current
        else:
            resultado = "unknown"
            #resultado = str(round(psutil.cpu_freq().current, 2))
    elif chave == "nucleos":
            resultado = str(psutil.cpu_count())
            resultado = resultado + " (" + str(psutil.cpu_count(logical=False)) + ")"
    else:
            resultado = str(info_cpu[chave])
    text = font.render(resultado, True, branco)
    surface.blit(text, (190, pos_y))
```

```
def mostra_uso_cpu(surface, l_cpu_percent):
    num_cpu = len(l_cpu_percent)
    x = y = 10
    espacos = 10
    alt = surface.get_height() - 2*y -120
    larg = (surface.get_width() - 2*y - (num_cpu+1)*espacos)/num_cpu
    bordas_laterais = x + espacos
    for i in l_cpu_percent:
            pygame.draw.rect(surface, olive, (bordas_laterais, 120, larg, alt))
            pygame.draw.rect(surface, cinzaclaro, (bordas_laterais, 120, larg, (1-i/100)*alt))
            bordas_laterais = bordas_laterais + larg + espacos
```

```
def plataforma():
    plat_processor = platform.processor()
    plat_node = platform.node()
    plat_platform = platform.platform()
    plat_system = platform.system()

    tela.blit(s5, (0, s5_posicao))

    texto_processor = '{:30}'.format("Família do processador:") + str(plat_processor)
    text = font.render(texto_processor, True, branco)
    s5.blit(text, (20, 8))

    texto_node = '{:30}'.format("Nome do computador:") + str(plat_node)
    text = font.render(texto_node, True, branco)
    s5.blit(text, (20, 28))

    texto_platform = '{:30}'.format("Plataforma:") + str(plat_platform)
    text = font.render(texto_platform, True, branco)
    s5.blit(text, (20, 48))

    texto_system = '{:30}'.format("Sistema Operacional:") + str(plat_system)
    text = font.render(texto_system, True, branco)
    s5.blit(text, (20, 68))
```

```
def ip():
    dic_interfaces = psutil.net_if_addrs()
    ip_maquina = dic_interfaces['ens33'][0].address

    tela.blit(s6, (0, s6_posicao))

    texto_barra1 = '{:30}'.format("IP da Máquina:") + str(ip_maquina)
    text = font.render(texto_barra1, 1, branco)
    s6.blit(text, (20, 15))
```

Logo após os imports, são declaradas as variáveis que serão utilizados para a aplicação como um todo, como largura e altura da tela, surfaces, cores, relógio e contador. Ou seja, todo o material usado para fazer o layout da aplicação. Realizei algumas mudanças no layout para que as informações ficassem melhor organizadas. Em vez de calcular a posição das surfaces dos núcleos “manualmente”, como fiz na versão anterior, calculei o espaço das 6 surfaces baseadas umas nas outras.

```
# recursos para layout

orangesidecar = (242, 218, 189)
preto = (0, 0, 0)
branco = (255, 255, 255)
olive = (82, 142, 66)
verde = (83, 189, 78)
darkseagreen = (149, 209, 137)
cinzaclaro = (192,192,192)
cinzaescuro = (35,35,35)

largura_tela = 600
altura_tela = 700
tela = pygame.display.set_mode((largura_tela, altura_tela))
pygame.display.set_caption("Teste de Performance 3")
pygame.display.init()
```

```
s1_altura = int(0.06 * altura_tela)
s2_altura = int(0.12 * altura_tela)
s3_altura = int(0.12 * altura_tela)
s4_altura = int(0.52 * altura_tela)
s5_altura = int(0.12 * altura_tela)
s6_altura = int(0.06 * altura_tela)

s1 = pygame.surface.Surface((largura_tela, s1_altura))
s1.fill(cinzaescuro)
s2 = pygame.surface.Surface((largura_tela, s2_altura))
s2.fill(cinzaescuro)
s3 = pygame.surface.Surface((largura_tela, s3_altura))
s3.fill(cinzaescuro)
s4 = pygame.surface.Surface((largura_tela, s4_altura))
s4.fill(cinzaescuro)
s5 = pygame.surface.Surface((largura_tela, s5_altura))
s5.fill(cinzaescuro)
s6 = pygame.surface.Surface((largura_tela, s6_altura))
s6.fill(cinzaescuro)
```

```
s1_posicao = 0
s2_posicao = s1_altura
s3_posicao = s1_altura + s2_altura
s4_posicao = s1_altura + s2_altura + s3_altura
s5_posicao = s1_altura + s2_altura + s3_altura + s4_altura
s6_posicao = s1_altura + s2_altura + s3_altura + s4_altura + s5_altura
```

No trecho abaixo inicia-se um loop onde eventos são captados para terminar a aplicação ou não, dependendo de uma variável booleana chamada “terminou”. É também onde as funções são chamadas e o relógio é utilizado para atualizar a tela e exibir o uso dos recursos variando.

```
# preparacao e chamada da aplicacao

info_cpu = cpuinfo.get_cpu_info()

pygame.font.init()
font = pygame.font.Font(None, 23)

clock = pygame.time.Clock()

cont = 60
```

```
terminou = False
while not terminou:

    for event in pygame.event.get():
            if event.type == pygame.QUIT:
                    terminou = True

    if cont == 60:
            mostra_titulo()
            uso_memoria()
            uso_disco()
            mostra_info_cpu()
            mostra_uso_cpu(s4, psutil.cpu_percent(interval=2, percpu=True))
            plataforma()
            ip()
            cont = 0

    pygame.display.update()

    clock.tick(60)
    cont = cont + 1

pygame.display.quit()
```

1. **Descreva de maneira teórica (pode utilizar exemplos) as diferenças de arquitetura de CPU.**

   Os processadores são um mundo à parte dentro do universo da computação. Eles podem ser single-core ou multi-core. Essa característica indica a quantidade de núcleos de processamento, que pode ir de 1 até 32. Quanto maior a quantidade, maior a capacidade de realizar tarefas ao mesmo tempo. Eles podem ser também de 32 ou 64

bits. Isso diz respeito à capacidade de processamento. Os de 64 aproveitam capacidades maiores de memória RAM. As duas empresas mais conhecidas na fabricação de processadores são Intel e AMD. Foi a Intel, inclusive, que criou o recurso de hyperthreading, que simula, em cada núcleo, 2 cores diferentes.

2. **Descreva de maneira teórica o que é a palavra do processador.**

   Uma palavra é basicamente uma sequência de bits que são processados em conjunto.

3. **Descreva de maneira teórica a diferença entre os núcleos físicos e lógicos. E em que influencia no processamento a utilização de núcleos lógicos.**

   É a diferença entre um núcleo físico, real, e um núcleo virtual, emulado pelo próprio processador para auxiliar os físicos. Como dito anteriormente, estão presentes na linha Core da Intel. Ex: i3, i5 e i7.

Teste de Performance 4

Ao código anterior adicionei duas novas funções, uma que mostra informações dos arquivos no diretório, como tamanho, data de modificação, data de criação e nome; outra, que exibe informações de um processo iniciado pela própria função. A seguir, imagens das funções e de seus respectivos outputs.

```python
# TP4

import os,time,psutil,subprocess

# Mostra informaçoes de um diretorio
def mostra_diretorio():
    lista = os.listdir()
    dic = {}

    for i in lista:
        if os.path.isfile(i):
            dic[i] = []
            dic[i].append(os.stat(i).st_size)
            dic[i].append(os.stat(i).st_atime)
            dic[i].append(os.stat(i).st_mtime)

    titulo = '{:11}'.format("Tamanho")
    titulo += '{:27}'.format("Data de Modificaçao")
    titulo += '{:27}'.format("Data de Criaçao")
    titulo += "Nome"
    print(titulo)

    for i in dic:
        kb = dic[i][0]/1000
        tamanho = '{:10}'.format(str('{:.2f}'.format(kb))+" KB")
    print(tamanho, time.ctime(dic[i][2]), time.ctime(dic[i][1]), " ", i)
```

```python
# Mostra informaçoes de um processo
def info_processos():
    # calculadora(linux)
    pid = subprocess.Popen("bc").pid
    p = psutil.Process(pid)
    print("\n")
    print("Nome:", p.name())
    print("Executável:", p.exe())
    print("Tempo de criação:", time.ctime(p.create_time()))
    print("Tempo de usuário:", p.cpu_times().user, "s")
    print("Tempo dte sistema:", p.cpu_times().system, "s")
    print("Percentual de uso de CPU:", p.cpu_percent(interval=1.0), "%")
    perc_mem = '{:.2f}'.format(p.memory_percent())
    print("Percentual de uso de memória:", perc_mem, "%")
    mem = '{:.2f}'.format(p.memory_info().rss/1024/1024)
    print("Uso de memória:", mem, "MB")
    print("Número de threads:", p.num_threads())

mostra_diretorio()
info_processos()
```

```
Tamanho    Data de Modificaçao        Data de Criaçao            Nome
6.18 KB    Sun Oct 24 21:44:02 2021 Sun Oct 24 21:44:02 2021   tests.py


Nome: bc
Executável: /usr/bin/bc
Tempo de criação: Sun Oct 24 21:51:02 2021
Tempo de usuário: 0.0 s
Tempo dte sistema: 0.0 s
Percentual de uso de CPU: 0.0 %
Percentual de uso de memória: 0.00 %
Uso de memória: 0.00 MB
Número de threads: 1
```

Teste de Performance 5

Para esse teste de performance, além de alguns ajustes nas funções print(), removi as chamadas das funções criadas no TP4 e passei a chamá-las através de uma função do módulo *sched.* A classe do scheduler define uma interface genérica para agendar eventos e definir suas prioridades. Esse módulo trabalha em conjunto com o módulo *time*. No código, utilizo time para definir o tempo no ínicio da função, e depois o fim. A primeira função é chamada com um delay de 2 segundos, e a segunda com 3. Ambas possuem a mesma prioridade. O output da função scheduler ficou assim:

```
Size       Modification Date          Creation Date              Name
4.02 KB    Wed Nov  3 21:04:06 2021 Wed Nov  3 21:04:09 2021   test2.py


Name: bc
Executable:
Creation Time: Mon Nov  1 20:10:16 2021
User Time: 0.0 s
System Time: 0.0 s
CPU Usage: 0.0 %
Memory Usage(%): 0.00 %
Memory Usage(MB): 0.00 MB
Threads: 1


---------- SCHEDULING TIME DIFFERENCE ----------

Beginning of Time Count:  Wed Nov  3 21:50:35 2021

Duration of lista_arquivos():  6.01 seconds. Clock Time:  6.49

Duration of mostra_processos():  6.01 seconds. Clock Time:  6.49

End of Time Count:  Wed Nov  3 21:50:41 2021
```

```python
def scheduler():
    scheduler = sched.scheduler(time.time, time.sleep)
    inicio = time.time()
    inicio_form = time.ctime()

    # enter: delay, priority, function name, arguments tuple
    scheduler.enter(2, 1, mostra_diretorio, (lista_arq,))
    clock_lista = time.process_time()
    scheduler.run()

    scheduler.enter(3, 1, info_processos, (pid,))
    clock_mostra = time.process_time()
    scheduler.run()

    tempo_fim = time.time()
    duracao_lista_arquivos = tempo_fim - inicio
    tempo_fim = time.time()
    duracao_mostra_processos = tempo_fim - inicio

    print("---------- SCHEDULING TIME DIFFERENCE ----------\n")
    print("Beginning of Time Count: ", inicio_form, "\n")
    print("Duration of lista_arquivos(): ", '{:.2f}'.format(duracao_lista_arquivos), "seconds. Clock Time: ", '{:.2f}'.format(clock_lista),"\n")
    print("Duration of mostra_processos(): ", '{:.2f}'.format(duracao_mostra_processos), "seconds. Clock Time: ", '{:.2f}'.format(clock_mostra),"\n")
    print("End of Time Count: ", time.ctime(), "\n")

scheduler()
```

Teste de Performance 6

Para o teste de performance 6 precisei importar os seguintes módulos:

```
# TP6

import psutil
import os
import ipaddress
```

A primeira função criada retorna um dicionário com as interfaces de rede(NIC - Network Interface Card). Para isso, foi preciso chamar a função net_if_address() do módulo psutil.

```
def list_nic():
    for interface in psutil.net_if_addrs():
        print(interface)
```

Em seguida, uma função que lista algumas informações da interface de rede passada como input, a opção escolhida foi a rede “ens33”.

```
def list_network_info(interface):
    # Lista algumas informações de rede passada como input
    print("IPv4: ", psutil.net_if_addrs()[interface][0][1])
    print("IPv6: ", psutil.net_if_addrs()[interface][1][1])
    print("MAC address: ", psutil.net_if_addrs()[interface][0][2])
```

```
elif choice == 2:
    # interface = input("Digite o nome da interface de rede: ")
    # list_network_info(interface)
    print("\n")
    list_network_info("ens33")
```

A função list_port_nmap(ip) recebe um endereço IP como parâmetro e imprime na tela todas as portas abertas. Para isso, utiliza o nmap.

```
def list_port_nmap(ip):
    # Lista as portas abertas usando nmap
    # no IP fornecido como input todas as portas estão fechadas
    print(os.system(f"nmap -p- {ip}"))
```

```
elif choice == 3:
    # ip = input("Digite o IP: ")
    # list_port_nmap(ip)
    print("\n")
    list_port_nmap("172.16.68.2")
```

Por fim, as funções search_subnet(), que imprime informações da subnet, e search_devices(subnet), que busca por hosts na rede através do nmap.

```
def search_subnet(ip):
    # ipadress é um modulo utilizado para inspecionar e manipular endereços IP
    # a funçao ip_interface() retorna um objeto que será armazenado na variável 'net'
    net = ipaddress.ip_interface(ip)
    print("Address:", net.ip)
    print("Mask:", net.netmask)
    print("Cidr:", str(net.network).split('/')[1])
    print("Network:", str(net.network).split('/')[0])
    print("Broadcast:", net.network.broadcast_address)

def search_devices(subnet):
    # o script abaixo retorna uma lista com os hosts que estão na subnet fornecida como input
    print(os.system(f"nmap -sP  {subnet}"))
```

```
elif choice == 4:
    # ip = input("Digite o IP: ")
    # search_subnet(ip)
    print("\n")
    search_subnet("172.16.68.2")
elif choice == 5:
    # subnet = input("Digite o subnet: ")
    # search_devices(subnet)
    print("\n")
    search_devices("192.168.1.0/24")
elif choice == 6:
    cond = False
else:
    print("Opção inválida")
```

O resultado de cada função, respectivamente, foi:

```
1 - Listar interfaces de rede
2 - Listar informações de rede
3 - Listar portas abertas
4 - Buscar subnet
5 - Listar hosts na rede
6 - Sair
```

```
 lo
 ens33
 docker0
```

```
IPv4:  172.16.68.2
IPv6:  fd15:4ba5:5a2b:1008:4606:2a84:345d:927c
MAC address:  255.255.255.0
```

```
show more (open the raw output data in a text editor) ...
Host is up (0.0021s latency).
All 65535 scanned ports on ubuntu (172.16.68.2) are closed

Nmap done: 1 IP address (1 host up) scanned in 11.87 seconds
0
```

```
Address: 172.16.68.2
Mask: 255.255.255.255
Cidr: 32
Network: 172.16.68.2
Broadcast: 172.16.68.2
```

```
Host is up (0.048s latency).
Nmap scan report for 192.168.1.5
Host is up (0.060s latency).
Nmap done: 256 IP addresses (4 hosts up) scanned in 7.22 seconds
0
```

É necessário pontuar que o nmap foi utilizado no lugar do ping. Normalmente, a varredura com ping é feita primeiro e, em seguida, os hosts são encontrados para serem rastreados em busca de portas abertas.

Para os próximos TPs, planejo integrar todos os trabalhos numa única interface.

Teste de Performance 7

Para o teste de performance 7 adicionei duas funções. A primeira recebe o nome de uma interface de rede como parâmetro e imprime quatro informações: IPv4, IPv6, MAC Address, Bytes recebidos e Bytes enviados. Para isso, ela utiliza o módulo psutil e sua função net_if_addrs(), que através de um dicionário onde informamos a chave 'interface' e a posição das informações, devolve as 4 informações que desejamos. A chamada da função ocorre dentro do loop da aplicação, e por conveniência, preferi deixar estático, buscando informações da interface 'ens33'.

```python
def byte_consumption(interface):
    # A interface de rede é passada como parametro
    # O script retorna o consumo de bytes da interface de rede
    # O consumo de bytes é calculado a partir da soma dos bytes de entrada e saída
    # Tudo isso é feito através das funçoes psutil.net_io_counters() e psutil.net_io_counters(pernic=interface)
    print("IPv4: ", psutil.net_if_addrs()[interface][0][1])
    print("IPv6: ", psutil.net_if_addrs()[interface][1][1])
    print("MAC: ", psutil.net_if_addrs()[interface][0][2])
    print("Bytes recebidos: ", psutil.net_io_counters(pernic=True)[interface].bytes_recv)
    print("Bytes enviados: ", psutil.net_io_counters(pernic=True)[interface].bytes_sent)
```

```python
elif choice == 6:
    # interface = input("Digite o nome da interface de rede: ")
    # byte_consumption(interface)
    print("\n")
    byte_consumption("ens33")
```

A segunda função imprime informações do consumo de um processo específico. Ao receber o PID de um processo, a função utiliza a função Process(pid) e memory_info() para buscar informações de consumo de CPU e memória relativas ao processo em questão. Tudo isso é

feito envolto em um try-catch, que evita uma interrupção repentina do programa caso o usuário forneça um PID inválido. Também por questão de conveniência, decidi manter o PID fixo na chamada da função.

```python
def process_consumption(pid):
    # O script retorna o consumo de bytes do processo passado como input
    try:
        # Lista info de memória do processo e converte para bytes(rss), que em seguida
        # divide por 1024 * 1024 para obter o consumo em MB
        print("Memória: ", round((psutil.Process(pid).memory_info().rss))/(1024*1024), "MB")
        print("CPU: ", psutil.Process(pid).cpu_percent(), "%")
        print(psutil.Process(pid).cpu_times())
    except:
        print("Processo não encontrado")
        pass
```

```python
elif choice == 7:
    # pid = input("Digite o PID: ")
    # get_process_consumption(pid)
    print("\n")
    process_consumption(4896)
```

O resultado de ambas as funções é, respectivamente:

```
IPv4:  172.16.68.2
IPv6:  fd15:4ba5:5a2b:1008:3af:766d:9be4:7bf8
MAC:  255.255.255.0
Bytes recebidos:  19104052
Bytes enviados:  3114721
```

```
Memória:  56.0859375 MB
CPU:  0.0 %
pcputimes(user=3.26, system=1.1, children_user=0.0, children_system=0.0)
```

O IPv6 é o padrão de endereço de Protocolo da Internet (IP) de próxima geração destinado a complementar e, eventualmente, substituir o IPv4, o protocolo que muitos serviços da Internet ainda usam hoje. Cada computador, telefone celular, componente de automação residencial, sensor IoT, e qualquer outro dispositivo conectado à Internet precisa de um endereço IP numérico para se comunicar entre outros dispositivos. O esquema de endereço

IP original, chamado IPv4, está ficando sem endereços devido ao seu uso generalizado devido à proliferação de tantos dispositivos conectados.

Cada interface de rede tem um endereço de hardware conhecido como MAC. Onde os endereços IP estão associados ao TCP/IP(software de rede), os endereços MAC são vinculados ao hardware dos adaptadores de rede. Por esse motivo, o endereço MAC às vezes é chamado de endereço de hardware de rede ou endereço físico. Aqui está um exemplo de um endereço MAC para uma NIC Ethernet: 00: 0a: 95: 9d: 68: 16.

Bytes recebidos: número de bytes que a interface especificada recebeu. Bytes enviados: número de bytes que a interface especificada enviou. Com essas informações, e estabelecendo um limite de tempo, é possível chegar na velocidade de banda para download e upload.

Referências

Material da disciplina no Moodle e gravações das aulas. Acesso em nov. 2021.