SISTEMAS OPERACIONAIS

ADS

SANDRO ROBERTO ARMELIN

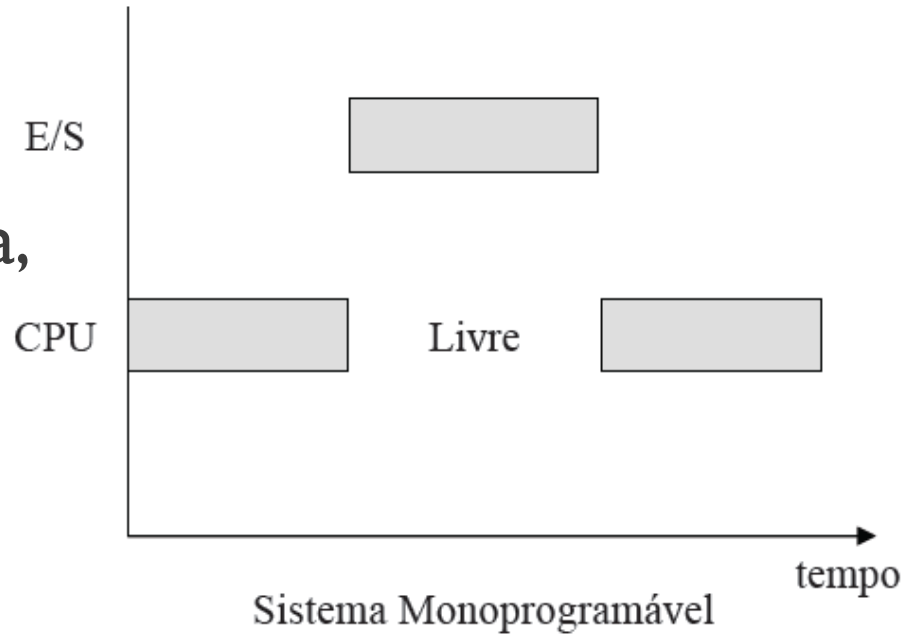
PROCESSOS

CONCORRÊNCIA

- Sistema Operacional – Visto como um conjunto de rotinas que executam concorrentemente de forma ordenada.
- Possibilidade de o processador executar instruções em paralelo com as operações e E/S permite que diversas tarefas sejam executadas concorrentemente.
- Sistemas Multiprogramáveis – Surgiram a partir da limitação dos sistemas monoprogramáveis (Processador dedicado).

TIPOS DE SISTEMAS OPERACIONAIS

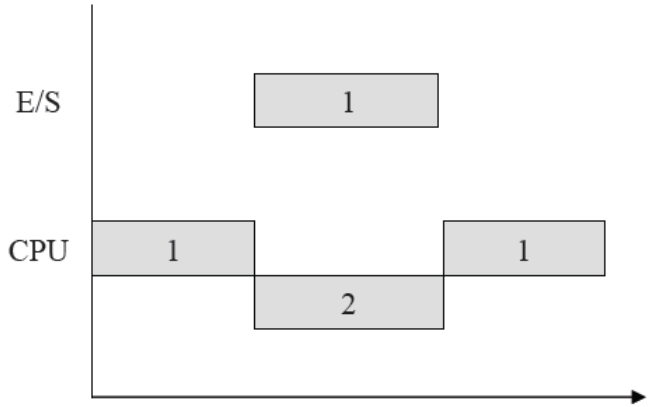
- Sistemas Monoprogramáveis/Monotarefa
  - Voltados tipicamente para a execução de um único programa.
  - Qualquer outra aplicação, para ser executada, deveria aguardar o término do programa corrente.
  - Processador, a memória e os periféricos permanecem exclusivamente dedicados à execução de um único programa.



Sistema Monoprogramável

TIPO DE SISTEMAS OPERACIONAIS

- Sistemas Multiprogramáveis/Multitarefa
  - Recursos compartilhados entre os diversas aplicações: enquanto um programa espera por um evento, outros programas podem estar processando neste mesmo intervalo de tempo.
  - O sistema operacional se incumbe de gerenciar o acesso concorrente aos seus diversos recursos.



Sistema Multiprogramável

PROCESSOS.

- Conceito Fundamental e mais importante para os Sistemas Operacionais.
- **Uma abstração de um programa em execução.**
- Permite a capacidade de operações "pseudo"concorrentes, mesmo quando há apenas uma CPU.
- Base para implementação de sistemas multiprogramaveis.
- A gerencia é função do SO, que para executar os programas é associado um programa a um processo.
- **Ao executar um programa o usuário tem impressão de possuir todo processador para seu uso. Não é verdade, visto que os recursos estão sendo compartilhados e a unidade de processamento também.**

PROCESSOS.

- Considere um servidor WEB – Solicitações de páginas Web chegam de toda parte.
- Quando uma solicitação chega, o servidor verifica se a pagina necessária esta em cache. Se estiver, é enviada de volta; Se não,
- Uma solicitação de acesso ao disco é iniciada para buscá-la. Entretanto, do ponto de vista da CPU, as solicitações de acesso ao disco duram uma eternidade. Enquanto espera que a solicitação ao disco seja concluída, muitas outras solicitações podem chegar.
- Se houver múltiplos discos, algumas delas podem ser enviadas rapidamente para outro discos antes de a primeira solicitação ser atendida.
- EVIDENTE QUE É NECESSÁRIO UMA FORMA DE CONTROLAR E MODELAR ESSA SIMULTANEIDADE. AÍ QUE ENTRA OS PROCESSOS E AS THREADS.

PROCESSOS.

- Impressão ao usuário de PARALELISMO.
- PSEUDOPARALELISMO para diferenciar do verdadeiro paralelismo de hardware nos sistema multiprocessadores.
- Chaveamento entre os processos em execução com taxa não uniforme.
- Única CPU? E processadores multinúcleo? Fato que um processador executa um processo de cada vez, com mais núcleos, **cada um** deles pode executar apenas **um processo por vez**.

PROCESSOS – MESMO PROGRAMA SENDO EXECUTADO 2X

- Mesma aplicação sendo executada **2 vezes**: **2 processos**.
- Exemplo: Abrir Processador de texto duas vezes ou imprimir dois arquivos ao mesmo tempo.
- S.O. PODE COMPARTILHAR O MESMO CÓDIGO ENTRE OS PROCESSOS, ESTANDO **APENAS UMA COPIA NA MEMÓRIA**. MAS EXISTEM 2 PROCESSOS EM EXECUÇÃO.

CRIAÇÃO DOS PROCESSOS:

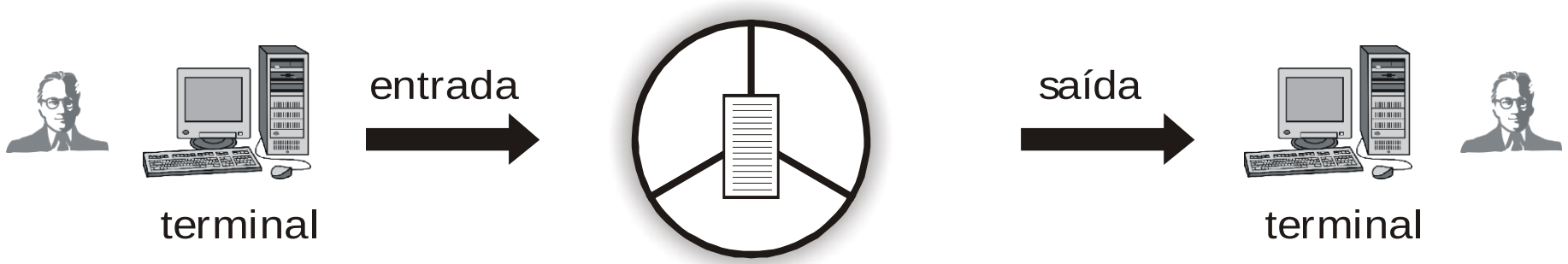
- **Durante inicialização do S.O.** – vários processos são criados, alguns **background** e outros **foreground**.
- **Por um outro processo em execução** – Um processo em execução faz chamadas de sistema (System calls) para criar novos processos.
- **Requisição do usuário** – Abrindo um aplicação.
- **Tarefa em lote** – Batch job.

PROCESSOS FOREGROUND E BACKGROUND

- **Foreground (Primeiro plano)** – Interagem com usuários. .
- **Background (Segundo plano)** – Não estão associados a usuários em particular.
  - Processo designado a aceitar mensagens eletrônicas sendo recebidas. Fica a maior parte do tempo inativo mas surgindo quando mensagem chega.

PROCESSOS FOREGROUND E BACKGROUND

(a) Processo Foreground

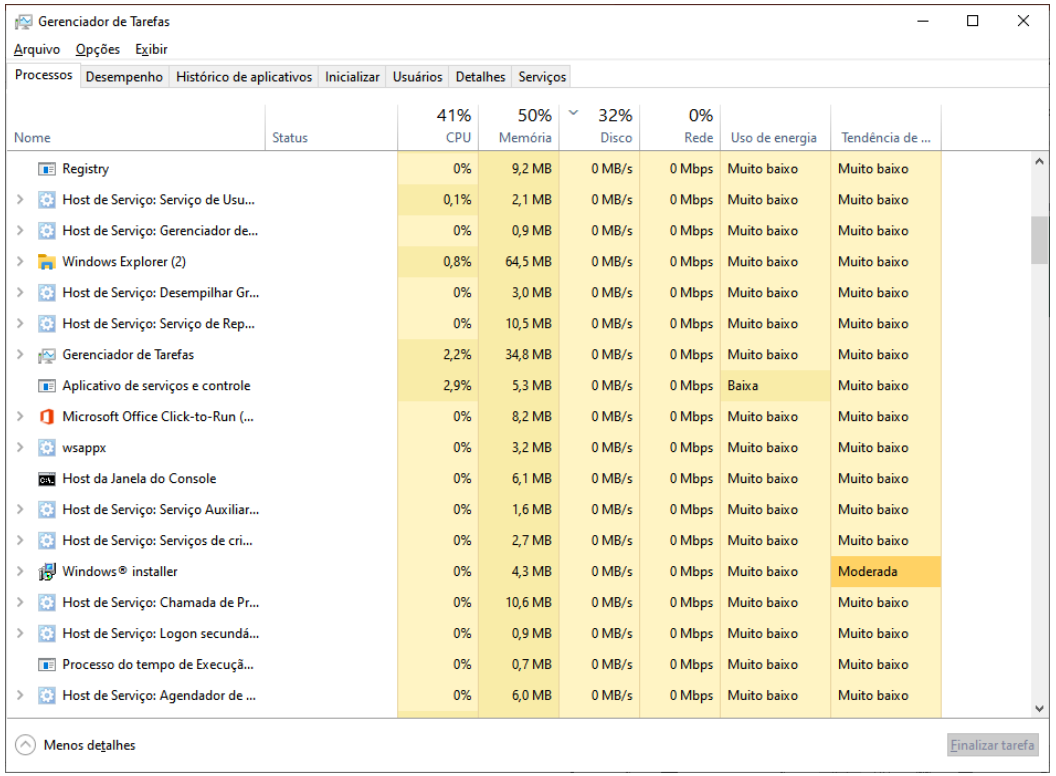


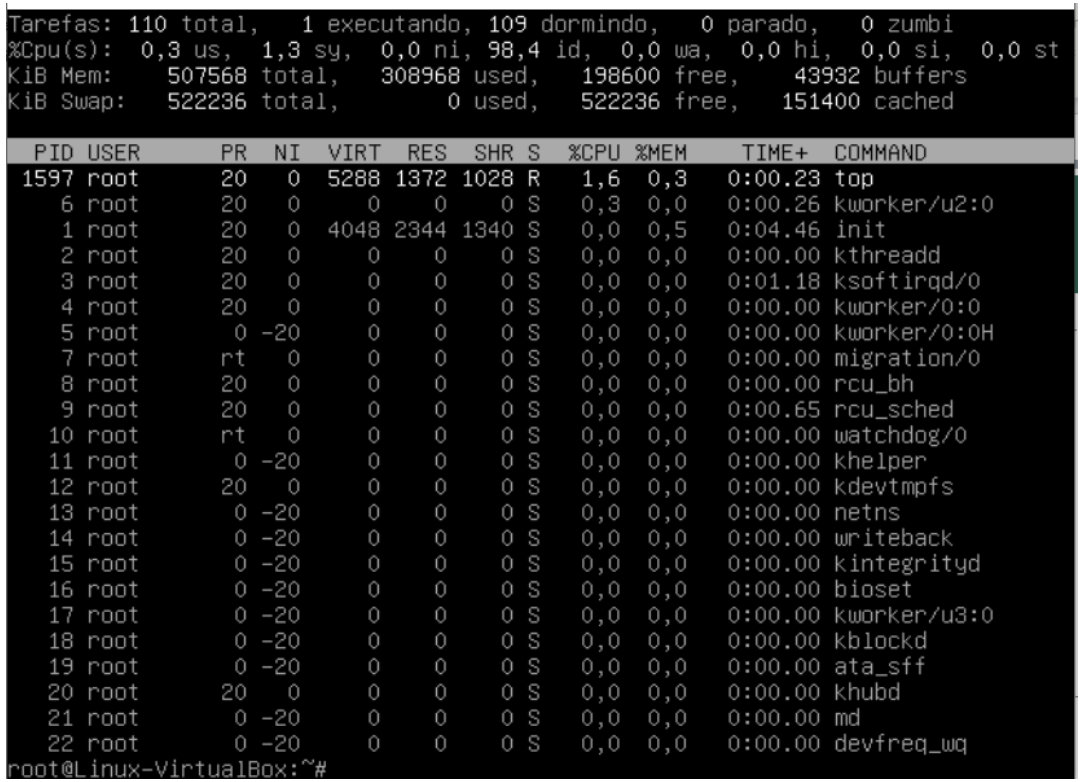
(b) Processo Background

TERMINO DE PROCESSOS, APÓS EXECUÇÃO DO SEU TRABALHO:

- **Saída normal** – Voluntária – Terminam o seu trabalho
- **Saída por erro** – Voluntária – Processo descobre um erro. Ex: digitando um comando inexistente por exemplo.
- **Erro fatal** – Erro no programa. Ex: referencia à memória inexistente ou divisão por zero.
- **Cancelamento por outro processo** – Cancelamento do processo. Kill no Linux.

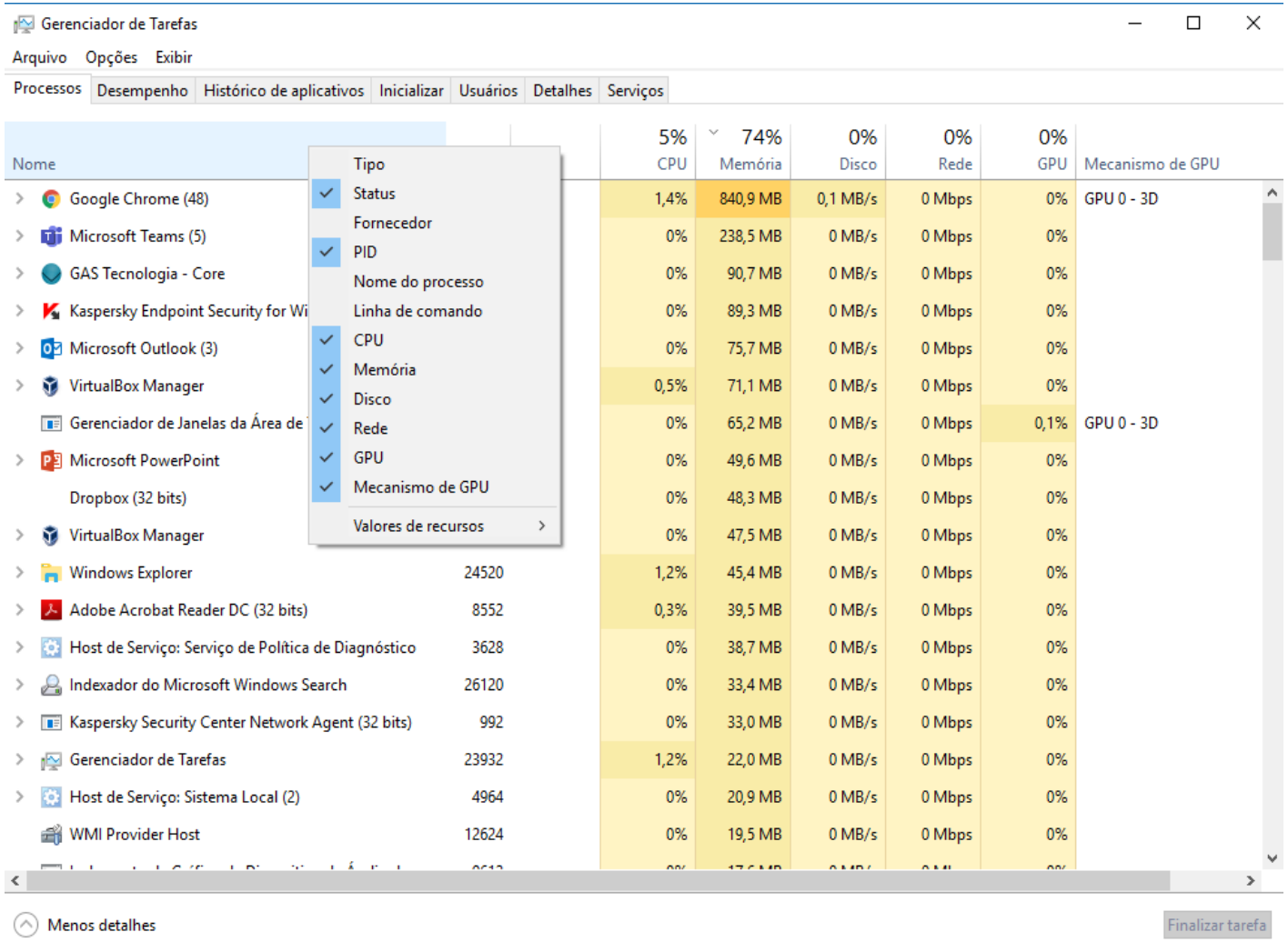
GERENCIAMENTO DE PROCESSOS WINDOWS X LINUX



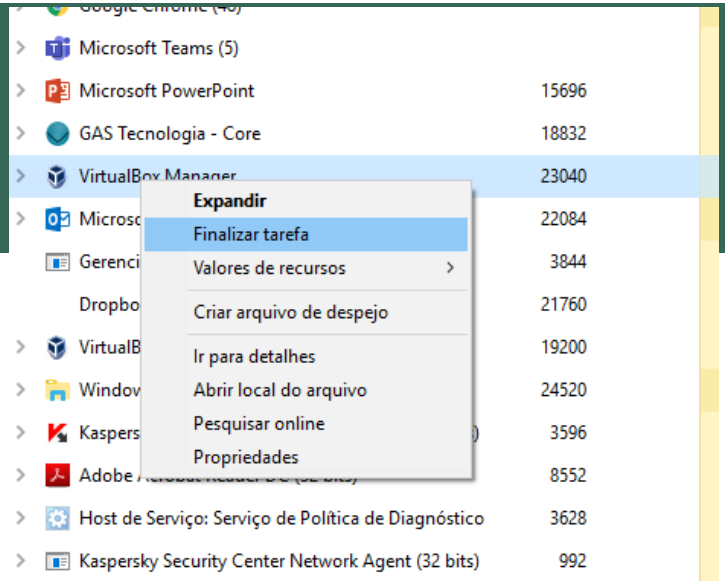


WINDOWS – GERENCIADOR DE TAREFAS.

- Processos
- Desempenho
- Histórico aplicativos
- Inicializar
- Usuários
- Detalhes
- Serviços
- Gpu – Processamento gráfico
  - Junho 2017.

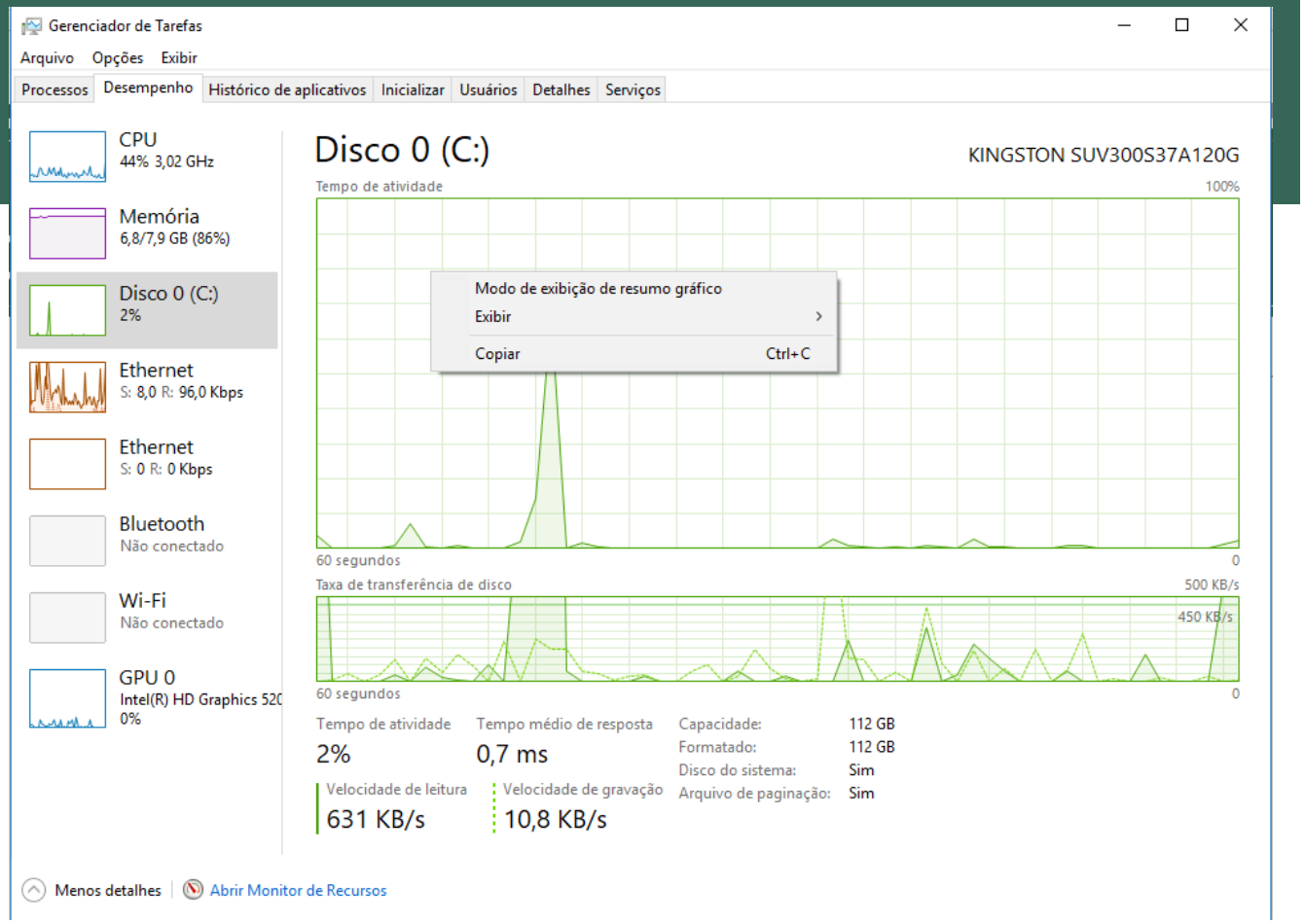


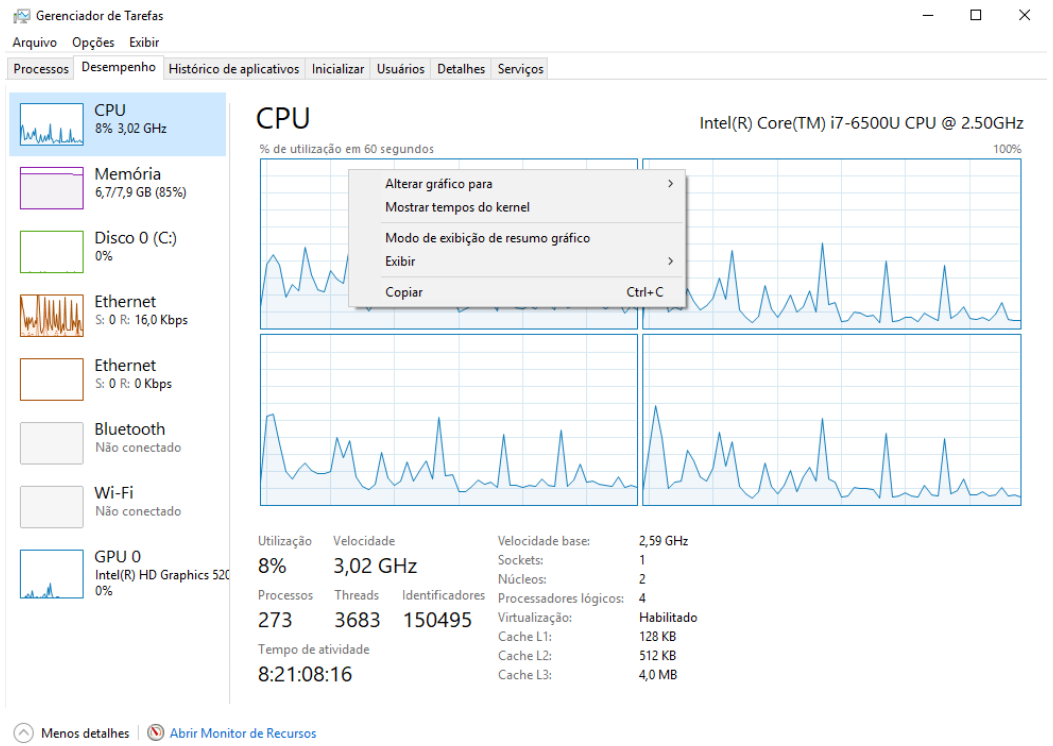
PROCESSOS



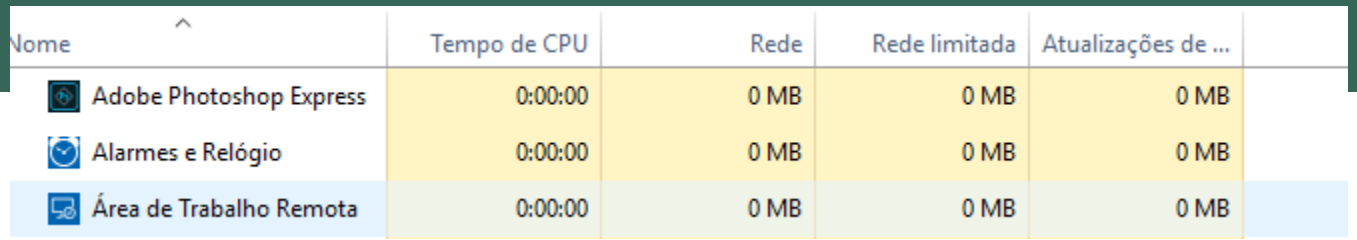
- Gerenciar processos
- Expandir processos vinculados
- Arquivo de despejo – cria arquivo dmp que pode ser analisado pelo o **Microsoft Debugging Tools for Windows** (ferramenta de depuração da Microsoft).
- Ir para detalhes.
- Exibir local do arquivo.

DESEMPENHO


Gerenciador de Tarefas
Arquivo Opções Exibir
Processos Desempenho Histórico de aplicativos Inicializar Usuários Detalhes Serviços
CPU
44% 3,02 GHz
Memória
6,8/7,9 GB (86%)
Disco 0 (C:)
2%
Ethernet
S: 8,0 R: 96,0 Kbps
Ethernet
S: 0 R: 0 Kbps
Bluetooth
Não conectado
Wi-Fi
Não conectado
GPU 0
Intel(R) HD Graphics 520
0%
Disco 0 (C:)
KINGSTON SUV300S37A120G
Tempo de atividade
100%
Modo de exibição de resumo gráfico
Exibir
Copiar Ctrl+C
60 segundos
0
Taxa de transferência de disco
500 KB/s
450 KB/s
60 segundos
0
Tempo de atividade
2%
Tempo médio de resposta
0,7 ms
Capacidade: 112 GB
Formatado: 112 GB
Disco do sistema: Sim
Arquivo de paginação: Sim
Velocidade de leitura
631 KB/s
Velocidade de gravação
10,8 KB/s
Menos detalhes
Abrir Monitor de Recursos

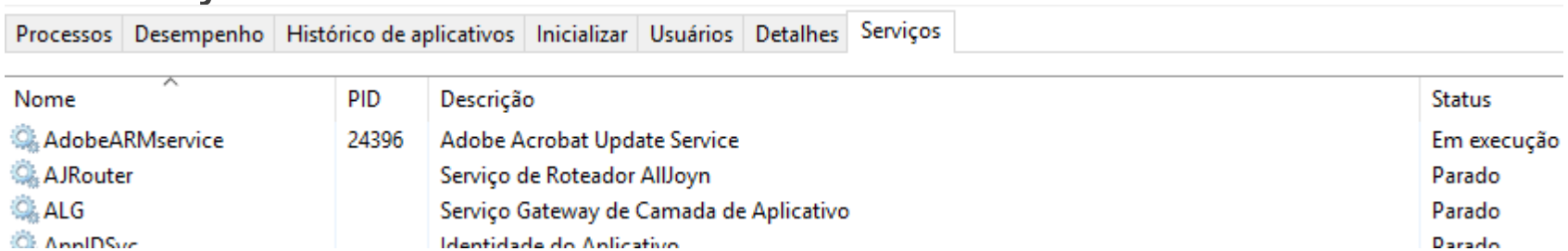

Gerenciador de Tarefas
Arquivo Opções Exibir
Processos Desempenho Histórico de aplicativos Inicializar Usuários Detalhes Serviços
CPU
8% 3,02 GHz
Memória
6,7/7,9 GB (85%)
Disco 0 (C:)
0%
Ethernet
S: 0 R: 16,0 Kbps
Ethernet
S: 0 R: 0 Kbps
Bluetooth
Não conectado
Wi-Fi
Não conectado
GPU 0
Intel(R) HD Graphics 520
0%
CPU
Intel(R) Core(TM) i7-6500U CPU @ 2.50GHz
% de utilização em 60 segundos
100%
Alterar gráfico para
Mostrar tempos do kernel
Modo de exibição de resumo gráfico
Exibir
Copiar Ctrl+C
Utilização
8%
Velocidade
3,02 GHz
Processos
273
Threads
3683
Identificadores
150495
Tempo de atividade
8:21:08:16
Velocidade base: 2,59 GHz
Sockets: 1
Núcleos: 2
Processadores lógicos: 4
Virtualização: Habilitado
Cache L1: 128 KB
Cache L2: 512 KB
Cache L3: 4,0 MB
Menos detalhes
Abrir Monitor de Recursos

OUTROS SERVIÇOS...



Nome	Tempo de CPU	Rede	Rede limitada	Atualizações de ...
Adobe Photoshop Express	0:00:00	0 MB	0 MB	0 MB
Alarmes e Relógio	0:00:00	0 MB	0 MB	0 MB
Área de Trabalho Remota	0:00:00	0 MB	0 MB	0 MB

- Histórico de aplicativos por usuário.
- Inicializar – Aplicativos programados para inicializar com o S.O.
- Usuários conectados.
- Detalhes do processos / threads.
- Serviços.



Processos | Desempenho | Histórico de aplicativos | Inicializar | Usuários | Detalhes | Serviços

Nome	PID	Descrição	Status
AdobeARMservice	24396	Adobe Acrobat Update Service	Em execução
AJRouter		Serviço de Roteador AllJoyn	Parado
ALG		Serviço Gateway de Camada de Aplicativo	Parado
AppIDSvc		Identidade do Aplicativo	Parado

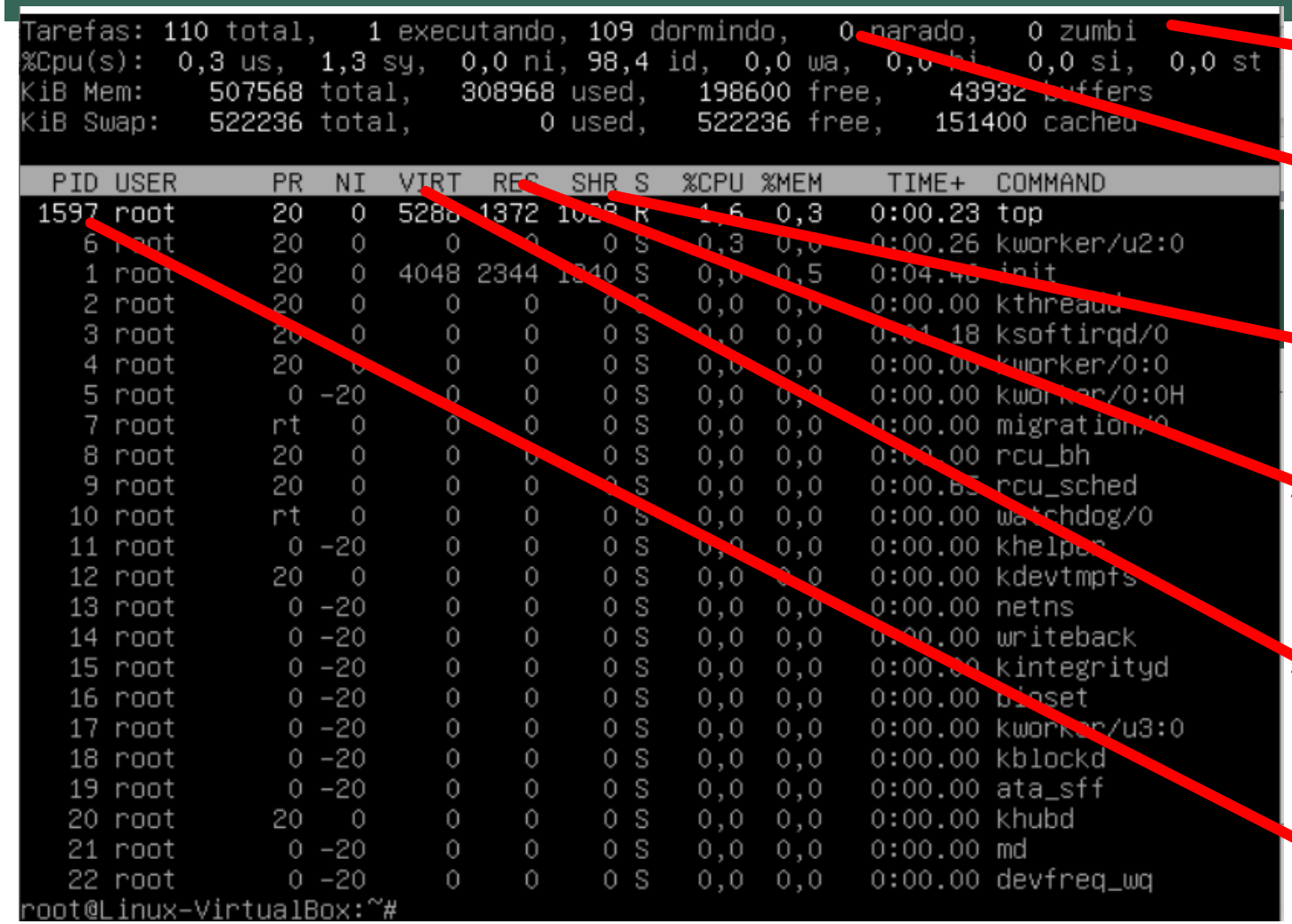
PROCESSOS X SERVIÇOS.

- Aplicativo – Interação do usuários na área de trabalho.
- Processo – Instancia de um determinado executável. Uma aplicação = vários processos.
- Serviço – Processos que são executados em segundo plano (Background) sem interação com a área de trabalho.

LINUX.

- Comandos:
- Top
- PS – informações reduzidas dos processos daquela interface tty.
- PS aux | more | PS aux | grep –i vim
- Kill ID do processo
- Renice prioridade –p ID
  - Prioridade variando de 19 (menos significativa) a -20 (mais significativa).
  - Por padrão todos processos usuários ganham prioridade 0
  - Ex. renice -10 –p 1516

TOP



Processo "morto" porem continua existindo.

Processo em Stop.

Quanto do tamanho VIRT é realmente compartilhável (memória ou bibliotecas).

Representa o tamanho residente. Quantidade de memória física real consumida de um processo, equivalente a %MEM

Tamanho virtual de um processo, que é a soma de memória que está utilizando.

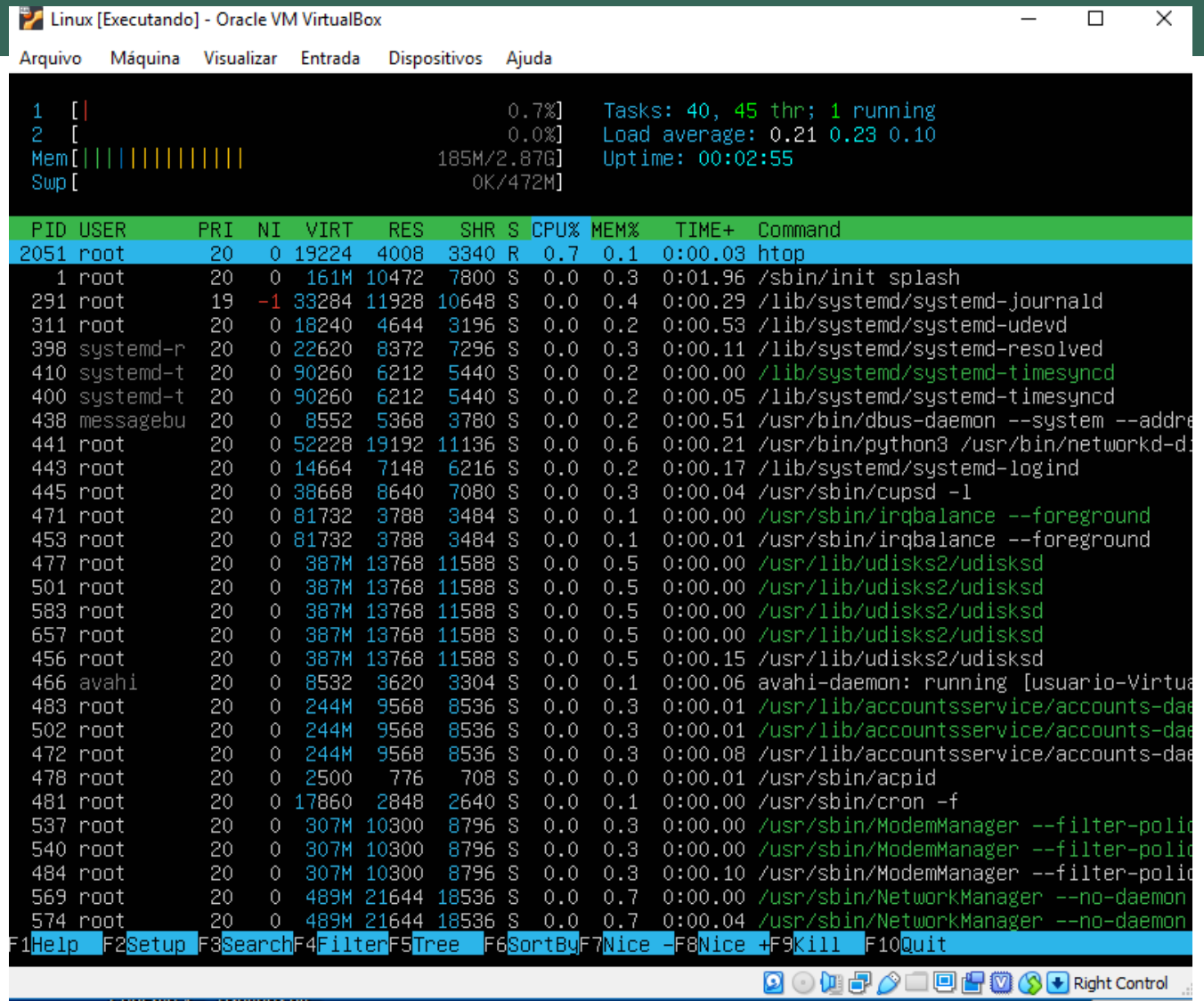
ID do processo.

LINUX, OS PROCESSOS POSSUEM UM DOS SEGUINTES STATUS:

- Running: Processo ativo, executando.
- Waiting: "Esperando" por algum evento do sistema. I/o em disco por exemplo.
  - Em situação de falha, pode ser que um processo em estado de waiting nunca saia, configurando deadlock.
- Suspended: "Suspenso", parado pelo usuário.
- Zombie: Finalizou a execução mas ainda esta na tabela de processos porque algum processo "pai" ainda não sabe que ele terminou. Processo normal é para ser comunicado.

ALGUMAS OPÇÕES COMANDO TOP.

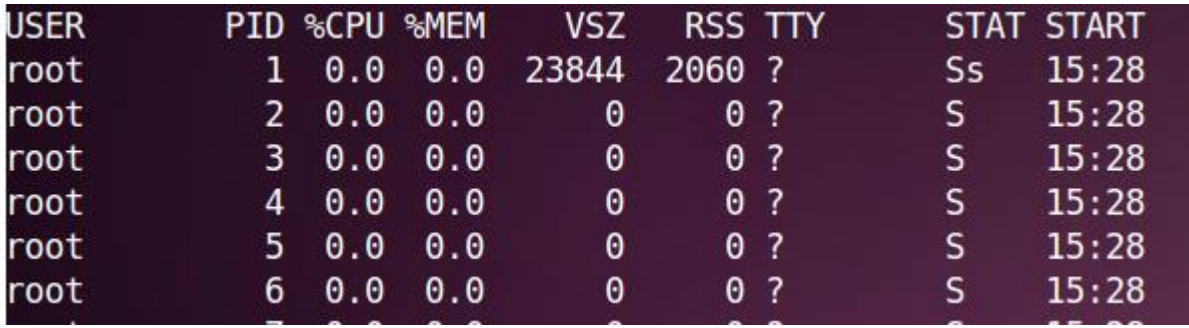
- Top –u root
- Outra opção htop
  - sudo apt-get install htop



PS -AUX

- **Propriedade do comando OS**
- **A:lista todos processos.**
- **U: exibe nome usuário owner**
- **X:processos sem relação com terminal.**

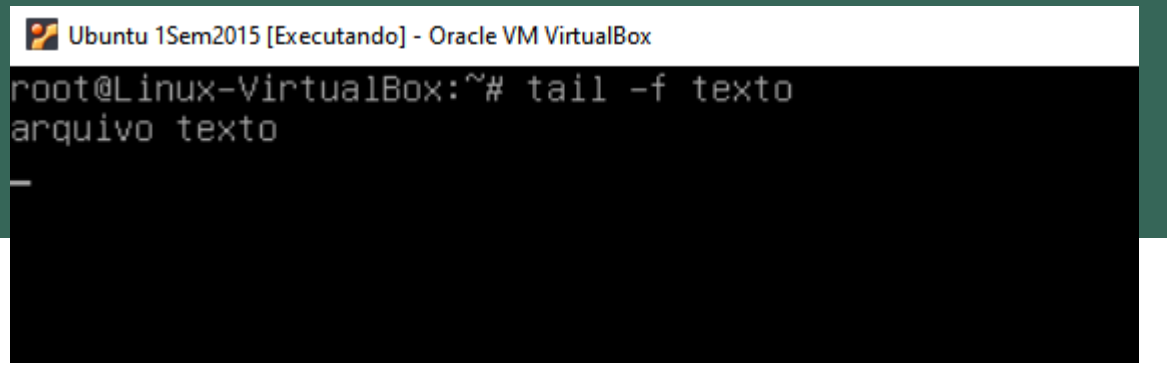
- **USER** Usuário que iniciou o processo (dono).
  PID Número único do processo.
  %CPU Utilização da CPU em porcentagem.
  START A hora em que o processo foi iniciado. Caso a hora seja do dia anterior, é representado pelo dia e mês.
  COMMAND O comando executado e todos seus argumentos.

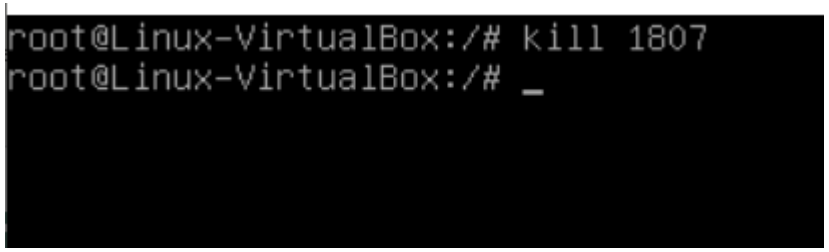


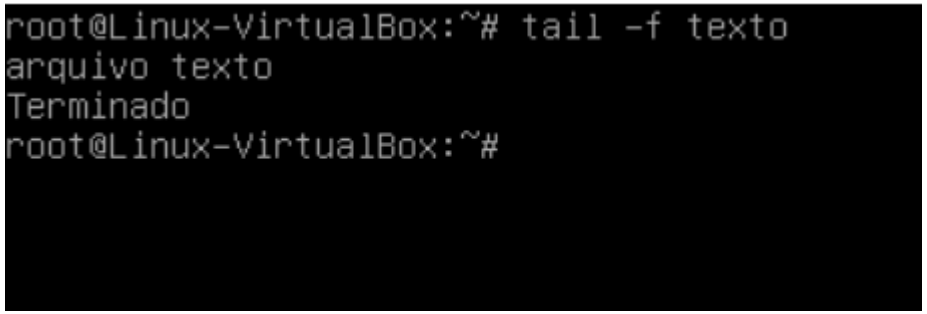
USER	PID	%CPU	%MEM	VSZ	RSS	TTY	STAT	START
root	1	0.0	0.0	23844	2060	?	Ss	15:28
root	2	0.0	0.0	0	0	?	S	15:28
root	3	0.0	0.0	0	0	?	S	15:28
root	4	0.0	0.0	0	0	?	S	15:28
root	5	0.0	0.0	0	0	?	S	15:28
root	6	0.0	0.0	0	0	?	S	15:28

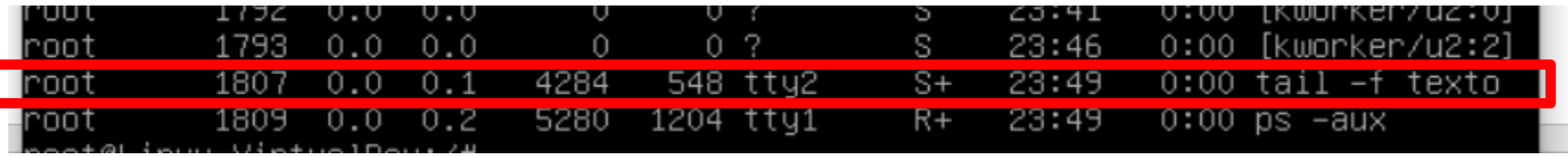
“MATANDO” UM PROCESSO.

- Criar em tty1 um arquivo de texto.
- Abrir interface tty2 e executar comando tail –f arquivotexto.
- Voltar para interface 1 e utilizar o comando para descobrir qual o ID do processo tail e utilizar o comando Kill.



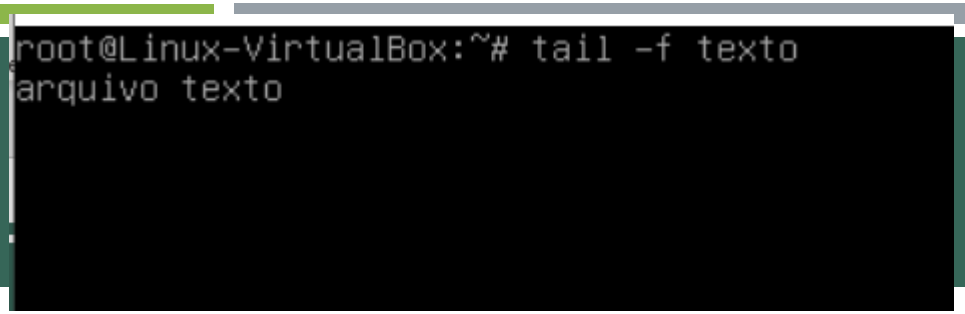


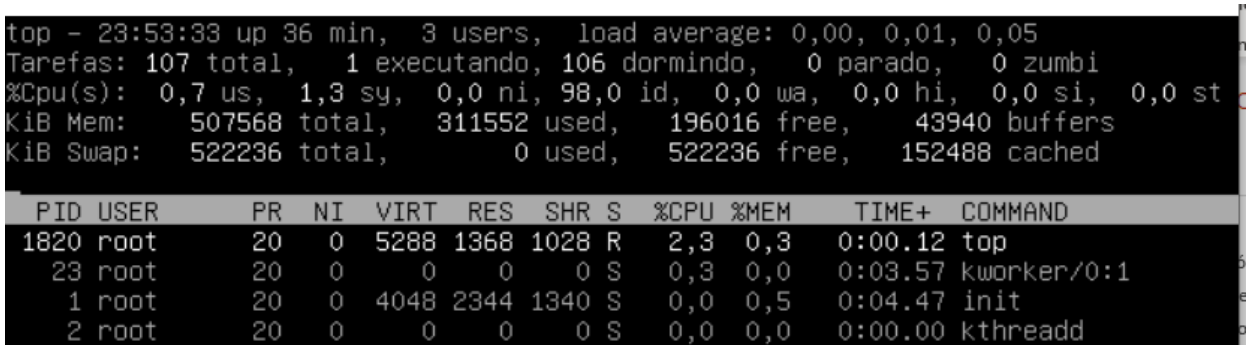


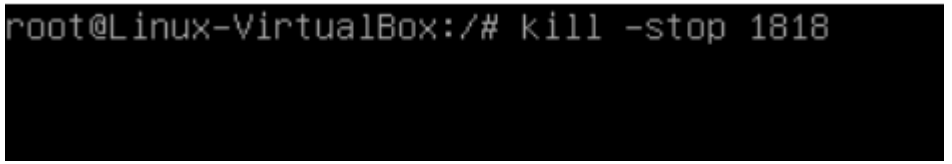


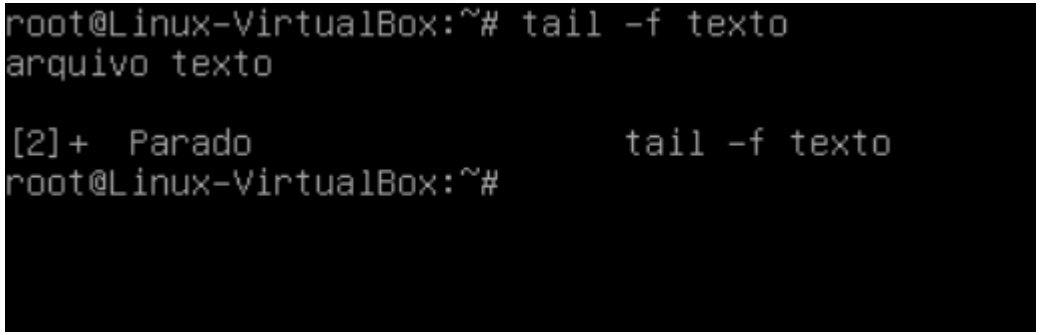
“PARANDO” UM PROCESSO

- Coloca novamente o comando tail para rodar...
- Utiliza comando ps –aux para conferir o ID e o top para conferir os processos com status parado.
- Utiliza o comando Kill –STOP ID
- Como é um comando shell ele aborda a execução.
- Kill –cont id para reativar o processo.



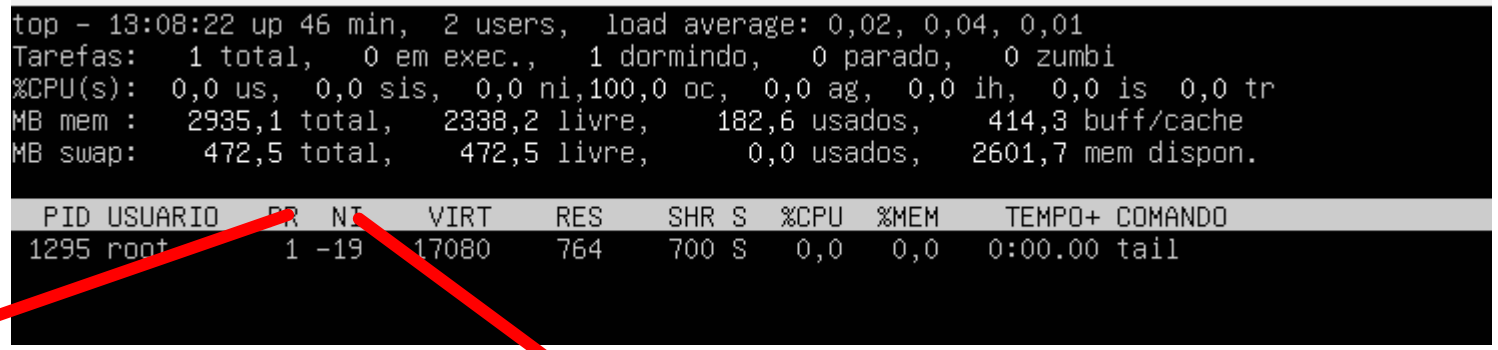






PRIORIDADE – COMANDO RENICE

- Prioridade que vão de -20 a +20.
- Menor o número, maior sua prioridade.
  - -20 = prioridade mais alta possível
  - 19 = prioridade mais baixa possível
  - 0 = neutro
- Renice -18 idprocess



Priority Nível de prioridade definida pelo Kernel.

Valor definido pelo comando RENICE.

RENICE - PRIORIDADE

- Aumentar prioridade – fazer com que SO execute + vezes o processo.
- PS –a na outra interface (tty) para visualizar o id do processo.
- Ou, ps –a | grep tail
- No top.
- Renice -19 1295
- Top –p idprocess

```
  34 root       0 -20       0       0       0 I   0,0   0,0   0:00.00 tpm_dev_wq
1380 root      20   0   20532    3744    3156 R   0,0   0,1   0:00.20 top
1295 root      20   0   16692     764     700 S   0,0   0,0   0:00.00 tail
 312 root      20   0   18240    4568    3140 S   0,0   0,2   0:00.55 systemd-udevd
 394 systemd+  20   0   90260    6120    5340 S   0,0   0,2   0:00.29 systemd-timesyn
```

```
top - 13:08:22 up 46 min,  2 users,  load average: 0,02, 0,04, 0,01
Tarefas:   1 total,   0 em exec.,   1 dormindo,   0 parado,   0 zumbi
%CPU(s):  0,0 us,  0,0 sis,  0,0 ni,100,0 oc,  0,0 ag,  0,0 ih,  0,0 is  0,0 tr
MB mem :   2935,1 total,   2338,2 livre,    182,6 usados,    414,3 buff/cache
MB swap:    472,5 total,    472,5 livre,      0,0 usados,   2601,7 mem dispon.

  PID USUARIO    PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TEMPO+ COMANDO
 1295 root       1 -19   17080    764    700 S   0,0   0,0   0:00.00 tail
```

```
root@usuario-VirtualBox:~# renice -19 1295
1295 (process ID) com prioridade antiga 0, prioridade nova -19
```

ESTADOS DE PROCESSOS.

ESTADOS DE UM PROCESSO.

- **EM EXECUÇÃO**: Utilizando a CPU naquele momento.
- **PRONTO**: Parado para dar lugar a outro processo. Aguardando o escalonamento.
- **BLOQUEADO**: Não pode executar pois depende de um evento externo. *Mesmo com CPU disponível não pode executar.

TRANSIÇÕES ENTRE OS ESTADOS.

- **T1** EXECUÇÃO PARA BLOQUEADO: Processo bloqueia aguardando uma entrada. Evento dependente de Entrada e saída.
- **T2** EXECUÇÃO PARA PRONTO: Escalonador seleciona outro processo. Processo já teve seu tempo necessário de CPU, dando oportunidade para outro.
- **T3** PRONTO PARA EXECUÇÃO: **Escalonador de processos** seleciona um dos processos da fila. Algoritmos para decidir quando e por quanto tempo os processos deve executar. **Competição** / **Equilíbrio**.
- **T4** BLOQUEADO PARA PRONTO: A entrada se torna disponível processo volta para a fila de pronto.

DE FORMA GERAL, O QUE OS PROCESSO FAZEM:

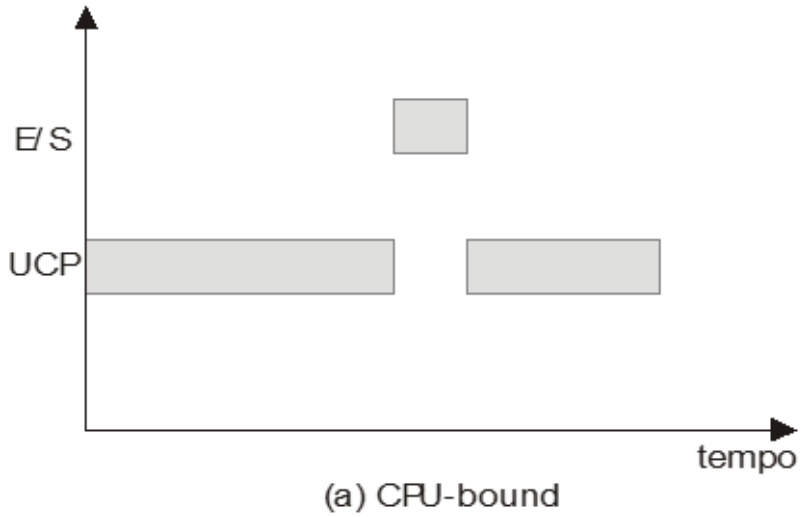
- Alguns são responsáveis por executar comandos dos usuários.
- Outros são parte do S.O. - responsabilidade de fazer requisição por serviço de arquivos ou gerenciar funcionamento de um acionador de disco.
- Em geral: PROCESSOS DE USUÁRIOS, DE DISCO, DE TERMINAIS, ETC.
- **Eles Bloqueiam quando estão a espera de que algo aconteça**.

TABELA DE PROCESSOS.

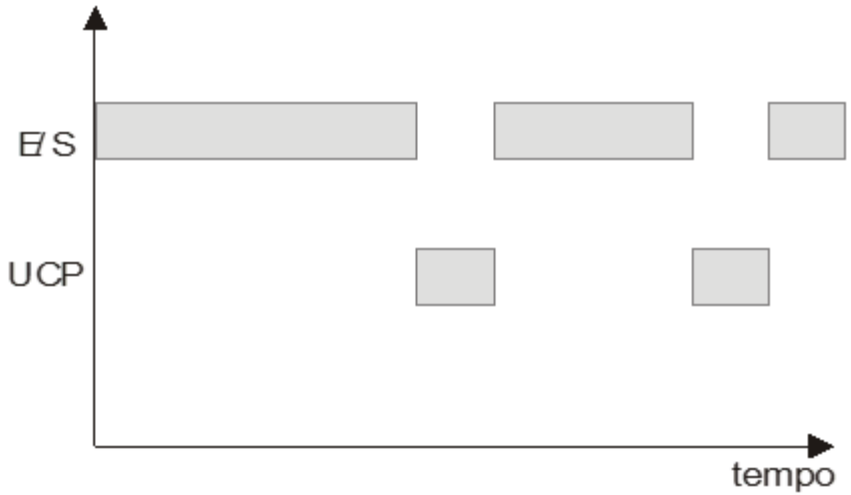
- Principal responsável pela possibilidade de implementação de processos.
- Também chamada de Process Control Blocks.
- Informações de cada processo:
  - Estado
  - ID
  - Contador de programa – Qtd de vezes que foi executado pelo processador.
  - Ponteiro da pilha
  - Alocação de memória,
  - Estado dos arquivos abertos.

PROCESSOS CPU-BOUND E I/O BOUND.

- Processo **CPU-BOUND** quando o processo passa maior parte do tempo no estado de execução, ou seja, utilizando mais o processador, realizando assim, **poucas** operações de leitura e gravação.
- Processo **I/O-BOUND** processo passa maior parte do tempo no estado de espera, pois realiza mais operações de E/S.



(a) CPU-bound



(b) I/O-bound

FUNÇÃO GETPID() E GETPPID()

- Função do padrão POSIX (Portable Operation System Intarface) – Familia de normas definidas pelo IEE para compatibilidade entre sistemas.
- **POSIX X LINUX** – Unix serviu de base para o padrão.
- getpid (): retorna o ID do processo de chamada.
- getppid (): retorna o ID do processo do pai do processo de chamada. Se o processo de chamada foi criado pela função fork () e o processo pai ainda existir no momento da chamada. Caso contrário, essa função retornará um valor de 1, que é o ID do processo para o processo init.
- Getpgrp(): retorna o ID do grupo do processo caso tenha filhos (threads)

LINGUAGEM C – AMBIENTE LINUX UBUNTU.

- gcc –v
- Compilar gcc codigo.c -o codigo
- Executar: ./código

- **Instalando o gcc...**
- rm /var/lib/dpgk/lock
- apt-get update ou apt update
- apt-get install gcc ou apt install gcc

GNU nano 3.2 id.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main()
{
 system("echo 'teste \033[01;32m mensagem\033[01;37m.'");
 printf ("Eu sou processo %d. Meu pai é o processo %d. Id do grupo é o processo %d,",
                                                 getpid(), getppid(), getpgrp());
 printf("\n");
 system("ps -a | grep tty3");
 int id;
 scanf("%d",&id);
 exit(0);
}
```

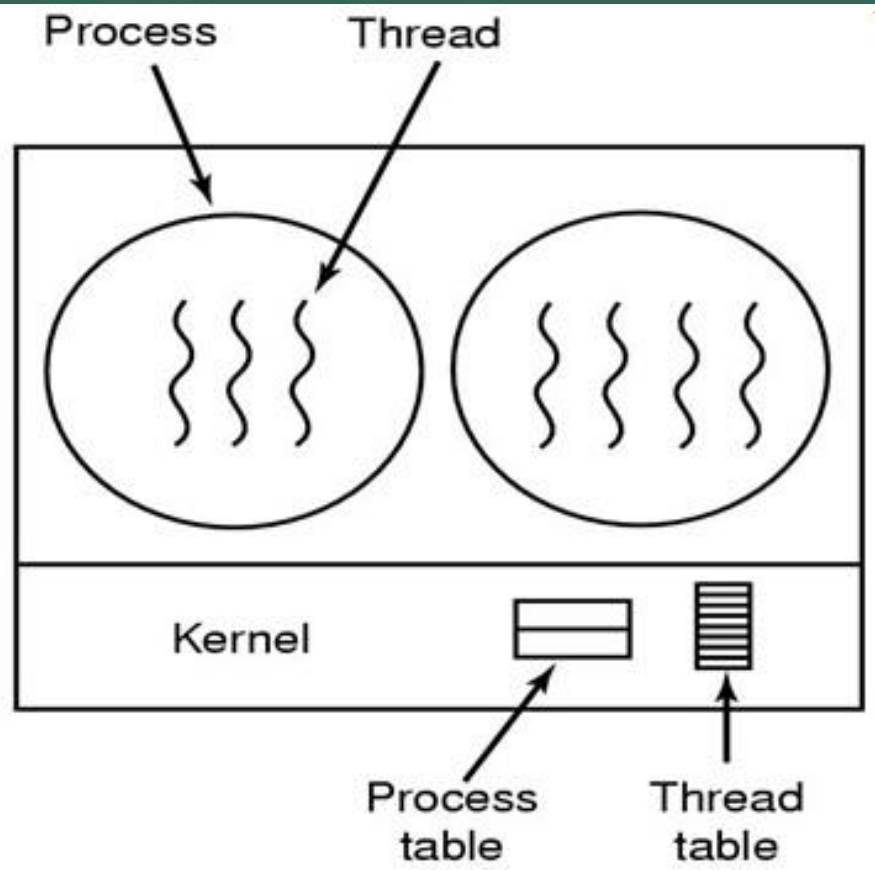
```
root@usuario-VirtualBox:/home# cat id.c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main()
{
 system("echo 'teste \033[01;32m mensagem\033[01;37m.'");
 printf ("Eu sou processo %d. Meu pai é o processo %d. Id do grupo é o processo %d,",getpid(), getpp
id(), getpgrp());
 printf("\n");
 system("ps -a | grep tty3");
 exit(0);
}

root@usuario-VirtualBox:/home#
```

AVALIANDO PROCESSOS...

```
root@usuario-VirtualBox:/home#
root@usuario-VirtualBox:/home#
root@usuario-VirtualBox:/home#
root@usuario-VirtualBox:/home# ./id
teste  mensagem.
teste  mensagem.
Eu sou processo 3468. Meu pai é o processo 1881. Id do grupo é o processo 3468,
 1881 tty3     00:00:00 bash
 3468 tty3     00:00:00 id
 3471 tty3     00:00:00 sh
 3472 tty3     00:00:00 ps
 3473 tty3     00:00:00 grep
root@usuario-VirtualBox:/home#
```

THREADS

- ***Mini processos*** – Processos dentro de um processo.
- **Inicialmente CADA PROCESSO** tem um espaço de endereçamento e uma única thread.
- Processos paralelos, sub-processos, processos filho.



THREADS

- **Porque threads?**
- Há situações desejável múltiplas threads de controle para **mesmo espaço endereçamento**.
- **Processos paralelos que são executados de forma independente.**

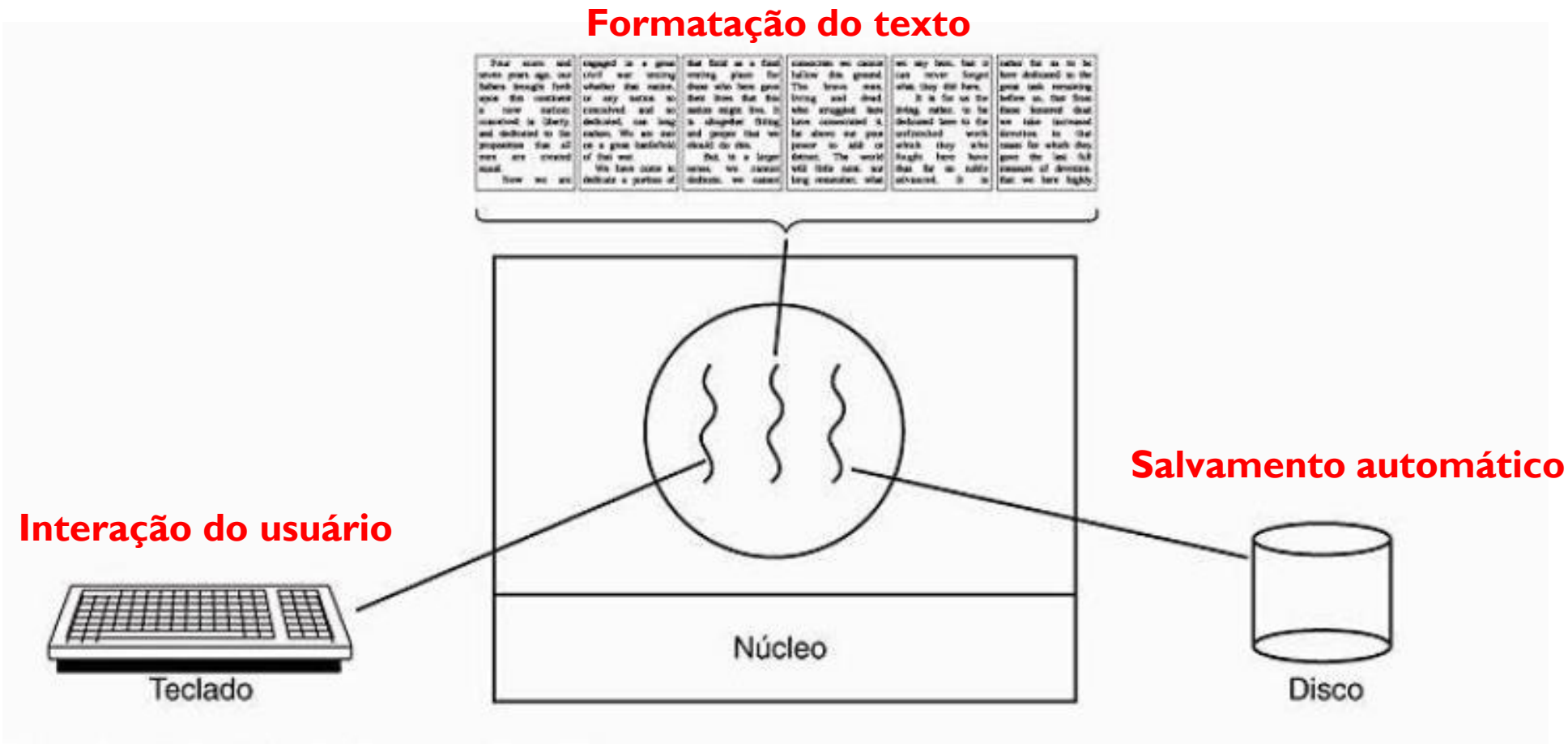
ARGUMENTOS PARA AS THREADS

- MESMO ARGUMENTO DA EXISTENCIA DOS PROCESSOS, Mas a divisão de tarefas para os próprios processos em mesmo endereçamento.
- Criação de Threads são mais fáceis, pois não tem recursos associados a elas. 100x mais rápido do que criar um processo.

THREADS – UM EXEMPLO.

- Considere um processador de texto com documento com muitas páginas.
- Alteração em uma página e na sequencia outra alteração em outra página requer uma reorganização do editor de texto.
- Thread, enquanto uma thread interage com usuário (teclado e mouse) a outra faz reformatação em segundo plano.
- Outra thread para salvamento automático do editor. Não sendo necessário bloquear o teclado e mouse.
- 3 Processos não funcionaria pois todos estariam trabalhando sobre o documento. 3 Threads que compartilham a memória comum.

PROCESSADOR DE TEXTO COM TRÊS THREADS.

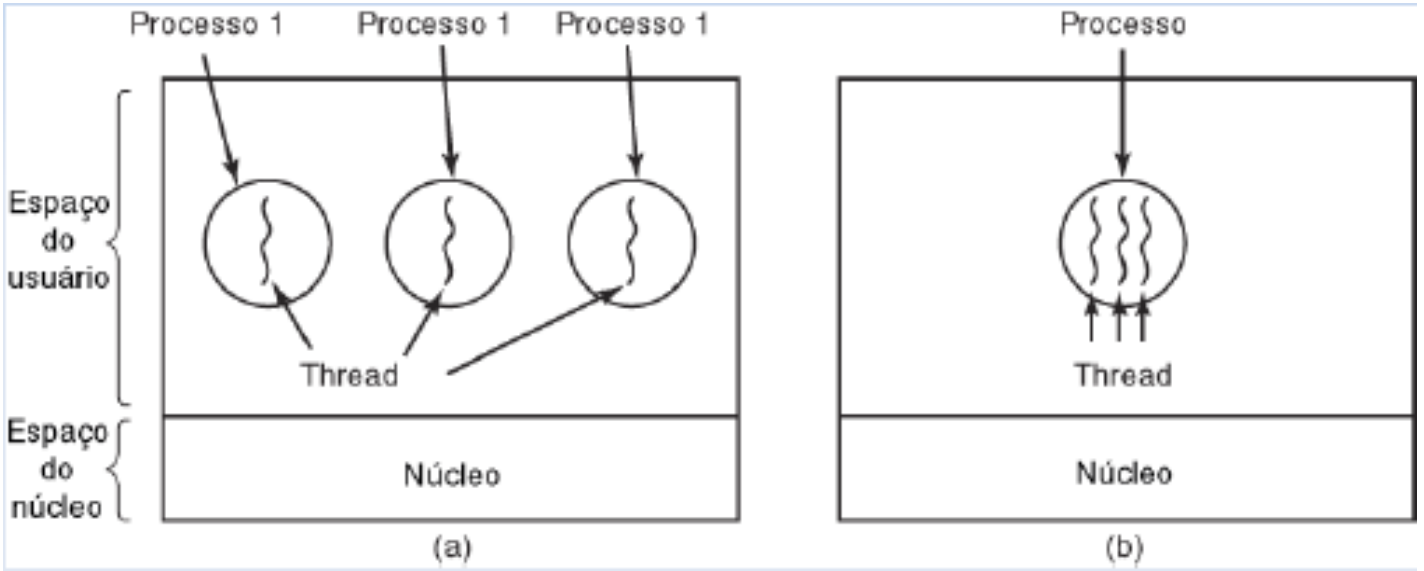


OUTRO EXEMPLO...

- Matriz de uma planilha eletrônica.
- Interação do usuário com dados de entrada em células interferindo no resultado de outras células devido a utilização de formulas.
- Uma thread em segundo plano realiza o recalculo das células.

THREADS.

- Figura (a) três processos tradicionais, cada um com sua thread de controle.
  - Cada um deles opera em espaço de endereçamento diferentes.
- Figura (b) único processo com 3 threads.
  - Todos os threads compartilham o mesmo espaço de endereçamento.



IMPORTANTE SOBRE OS DOIS TIPOS DE THREADS.

- Todos os threads tendo o mesmo espaço de endereçamento significa que compartilham variáveis globais.
- Cada thread pode ter acesso a qualquer endereço de memoria dentro do espaço de endereçamento do processo.
- Não há proteção entre threads.
- Não tem necessidade pois são processos gerenciados por processo pai.
- Já em processos independentes, podem ser usuários diferentes.

ESTADOS DE UMA THREAD.

- **Mesmo conceito de processos.**
- Estados:
  - **EM EXECUÇÃO**: Utilizando a CPU naquele momento.
  - **PRONTO**: Parado para dar lugar a outro processo. Aguardando o escalonamento.
  - **BLOQUEADO**: Não pode executar pois depende de um evento externo

CRIANDO UM PROCESSO FILHO – FORK.

- **FORK** – Função de chamada de sistema padrão POSIX.
- Cria um processo filho. O processo pai é aquele que usou o fork.
- Por exemplo, suponha que você programou um software em C, e nele usou a chamada **fork()**. Esse programa em C, executando, é o processo pai.
  Fork cria o processo filho idêntico ao pai, inclusive tendo as mesmas variáveis e registros.
- É uma copia exatamente igual ao pai.
- Apesar de processo filho, vai ser executado independente. O que acontece em um processo não ocorre nos outros, São processos distintos.

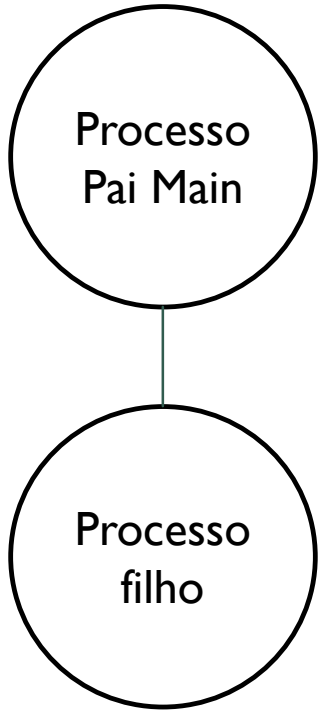
FUNÇÃO WAIT.

- Permite que o processo pai espere o término de um processo filho.
- A função devolve o status de retorno de qualquer processo filho que termine.

FORK 1

```
root@usuario-VirtualBox:/home# gcc fork.c -o fork
root@usuario-VirtualBox:/home# ./fork
eu sou o pai e espero pelo filho 1453
sou filho 1453 e espero 10 segundo
ja esperou e vou embora
meu filho acabou de terminar vou terminar tambem
root@usuario-VirtualBox:/home#
```

```
 GNU nano 3.2

#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <wait.h>
int main()
{
 int id;
 id = fork();
 if (id != 0)
 {
   printf ("eu sou o pai e espero pelo filho %d\n",id);
   wait(0);
   printf ("meu filho acabou de terminar vou terminar tambem\n");

 }else{
  printf ("sou filho %d e espero 10 segundo\n",getpid());
 sleep(10);
  printf ("ja esperou e vou embora\n");

 }
exit(0);
}
```



A partir do fork o bloco de código é compartilhado para os 2 processos, desta forma o if é utilizado para que cada processo execute somente sua parte.
O Wait aguarda o processo dependente terminar.

OBSERVAÇÕES:

- **Processos antes da execução do programa fork**
- **Processos durante a execução dos 10 segundos do processo filho.**
- **Processos após termino do fork.**

tty4

```
root@usuario-VirtualBox:~# ps -a
  PID TTY          TIME CMD
 1629 tty3     00:00:00 bash
 1735 tty4     00:00:00 bash
 1745 tty4     00:00:00 ps
root@usuario-VirtualBox:~# ps -a
  PID TTY          TIME CMD
 1629 tty3     00:00:00 bash
 1735 tty4     00:00:00 bash
 1758 tty3     00:00:00 fork
 1759 tty3     00:00:00 fork
 1760 tty4     00:00:00 ps
root@usuario-VirtualBox:~# ps -a
  PID TTY          TIME CMD
 1629 tty3     00:00:00 bash
 1735 tty4     00:00:00 bash
 1774 tty4     00:00:00 ps
root@usuario-VirtualBox:~# _
```

tty3

```
root@usuario-VirtualBox:/home# ./fork
eu sou o pai e espero pelo filho 1668
sou filho 1668 e espero 10 segundo
ja esperou e vou embora
meu filho acabou de terminar vou terminar tambem
root@usuario-VirtualBox:/home# ps -a
  PID TTY          TIME CMD
 1629 tty3     00:00:00 bash
 1669 tty3     00:00:00 ps
root@usuario-VirtualBox:/home# ./fork
eu sou o pai e espero pelo filho 1759
sou filho 1759 e espero 10 segundo
ja esperou e vou embora
meu filho acabou de terminar vou terminar tambem
root@usuario-VirtualBox:/home# _
```

Observando os processos antes de executar.
Durante os 10 segundos que o processo filho aguarda abrir outra interface (tty4) e aplicar o comando ps.

FORK 2

- Processo filho conta ate 5
- E depois volta o controle para o pai.
- Após fork, os dois processos estão rodando independentemente.
- Função wait (processoesperado, status, parametrozero para aguardar)

```
 GNU nano 2.2.6                Arquivo: fork2.c

#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
int main()
{
 pid_t filho;
 int i, status;
 filho = fork();
 if (filho == 0) //codigo do filho
 {
   printf("Sou processo filho.\n");
   for (i=0; i<5; i++)
    {
      printf("%d\n",i);
      sleep(2);
    }
    exit(0);
 }else{ //codigo do pai
     wait(filho,&status,0);
      printf("Sou pai. Agora posso executar o meu codigo");
 }
 return(0);
}
```

REFERÊNCIA BIBLIOGRÁFICA.

- TANEMBAUM, Andrew S. Sistemas Operacionais Modernos. 2° Ed. Pearson, 2005.
- MACHADO, Francis Berenger; MAIA, Luiz Paulo. Arquitetura de Sistemas Operacionais. 4° Edição, LTC, 1996.