Gerenciamento de Processos

Conceitos e Comandos

1. Introdução ao Sistema Operacional

- Quando o assunto é sistema operacional, logo pensamos em Windows, Linux, Android etc. Esses são apenas alguns dos principais exemplos, mas o que de fato é um sistema operacional (S.O.) Podemos dizer que basicamente um S.O. possui as seguintes funções:

1. Apresentar ao usuário uma forma de “conversar” com o hardware sem precisar utilizar linguagem de máquina;

2. Gerenciar um sistema complexo: processadores, memórias, discos, dispositivos de entrada e saída (E/S), arquivos etc.

3. Permitir aos programas o armazenamento e a obtenção de informações;

4. Controlar o fluxo de dados entre os componentes do computador;

5. Responder a erros e a pedidos do usuário;

6. Impor o escalonamento entre programas que solicitam recursos (memória, disco, entre outros);

1. Introdução ao Sistema Operacional

- Como vimos, o S.O faz o “meio de campo” entre o hardware e os programas do usuário. Sempre que for necessário acessar um hardware (disco por exemplo), é realizada no SO uma chamada de sistema (system call), que abstrai os detalhes de baixo nível.

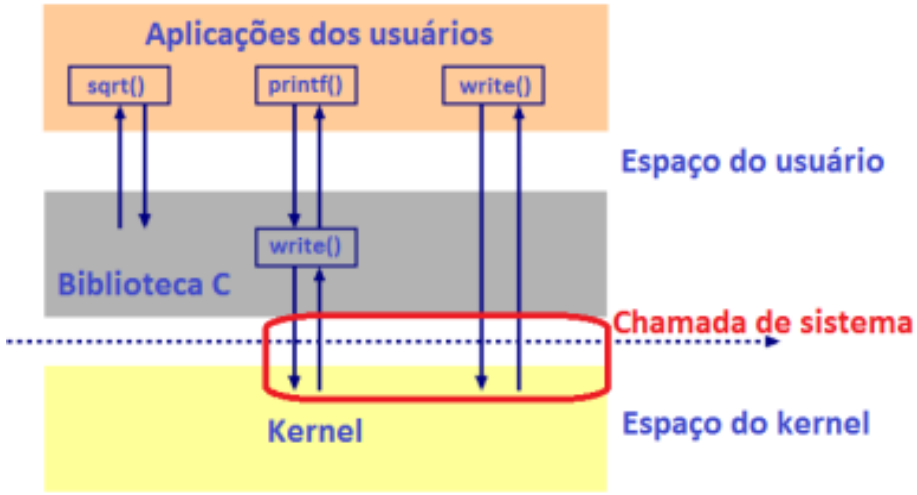
- O kernel é o núcleo do sistema operacional, que possui controle total do sistema. O kernel é um dos primeiros programas a ser carregado durante a inicialização.

- Quando necessário, uma system call pode solicitar que o SO entre no **modo kernel** (pode executar instruções privilegiadas, como gravação no disco) ou no **modo usuário** (apenas instruções não privilegiadas).

1. Introdução ao Sistema Operacional

- Como exemplo, vamos pensar em um programador que precisa criar um programa que calcule uma raiz quadrada e grave o resultado no disco.

- O comando SQRT não precisa realizar uma system call, pois recebe um valor e retorna sua raiz quadrada. Mas para gravar este dado (write) é necessário acessar o disco. Para isso, o S.O deve entrar em modo kernel.



1. Introdução ao Sistema Operacional

- Para gerenciar um SO podemos utilizar sua interface gráfica (botões, menu, janelas, etc.). Entretanto, nem todos os sistemas possuem interface gráfica.

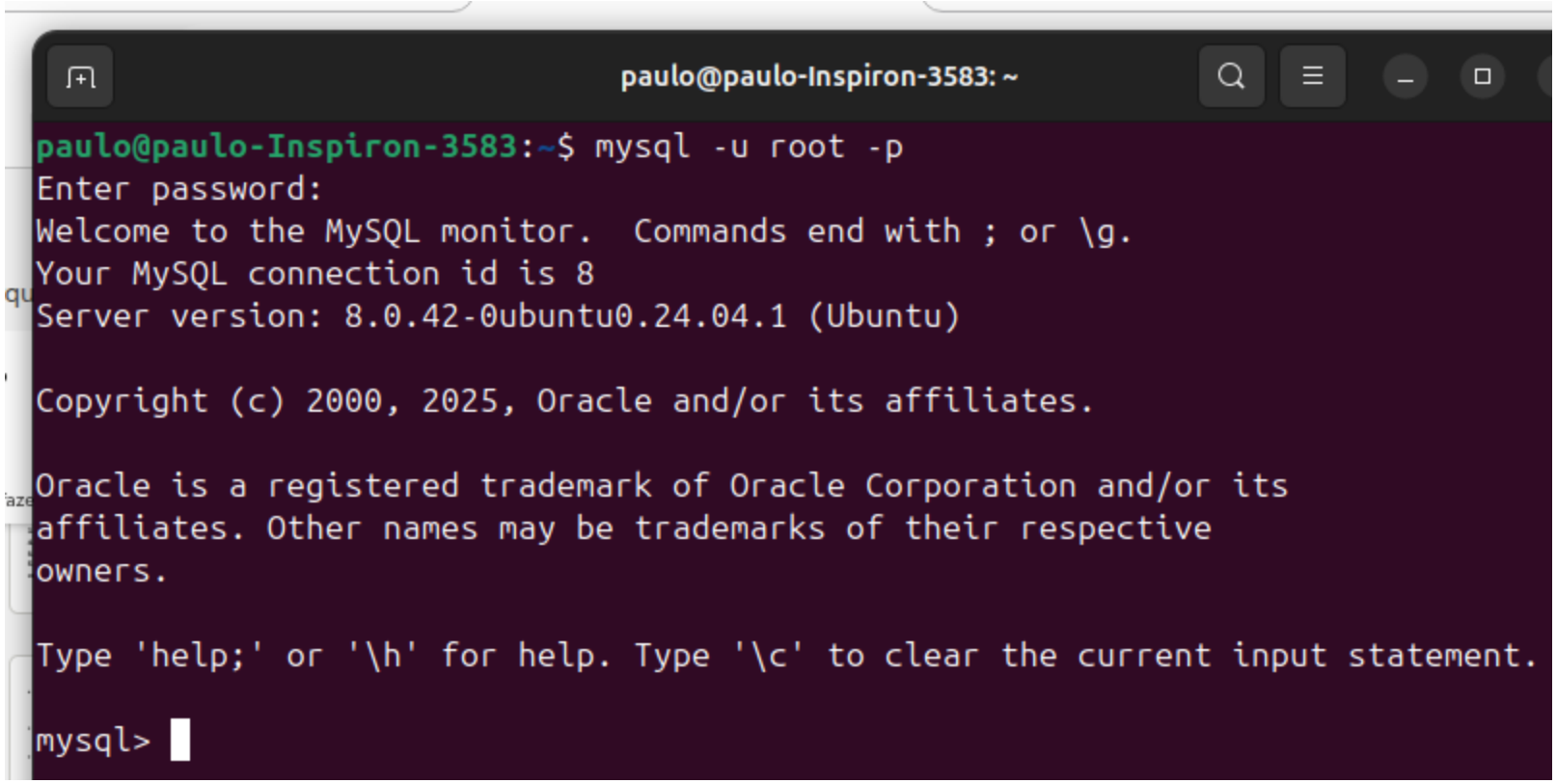
- Alguns sistemas operacionais possuem como única forma de gerenciamento um terminal de comandos, onde o usuário pode entrar com códigos específicos e gerir o SO.

- Um das vantagens de não ter interface é a economia de recursos computacionais. Entretanto, usuários menos experientes podem ter dificuldades.

- O Linux possui um terminal para scripts, conhecido como **Shell Script**. No windows temos o **Power Shell**.

1. Introdução ao Sistema Operacional

- A imagem abaixo mostra como é um terminal Linux:



2. Tipos de Gerenciamentos Realizados pelo SO

- No geral, existem quatro tipos de gerenciamento realizados por um S.O:

1. **Gerenciamento de processos**: Processo é a unidade básica de trabalho do SO. Ele é responsável por criar, excluir, ordenar e gerenciar a comunicação dos processos.

2. **Gerenciamento de memória**: Controla que setores da memória estão sendo usados e por quais processos. Além disso, é responsável pela alocação e liberação dinâmica do espaço.

3. **Gerenciamento de dispositivos de E/S:** Gerenciamento dos sinais que informam as ações que o usuário espera que o dispositivo realize, o tratamento das interrupções e erros gerados.

4. **Gerenciamento de armazenamento:** Fornecimento do sistema de arquivos (NTFS, EXT4) para definir a forma como os dados serão armazenados. Gerenciamento do espaço em dispositivos de armazenamento de dados (HD, SSD, pen drive, entre outros).

3. Tipos de Kernel

- Em relação à arquitetura do kernel, o sistema operacional pode ser classificado como:

1. **Monolítico**: Os processos são executados no espaço de núcleo, com acesso completo ao hardware. Como todos os módulos são executados em um mesmo espaço de endereçamento, se houver ocorrência de erro em um desses espaços, todo o sistema pode ser afetado.

2. **Microkernel** (micronúcleo): É um núcleo de tamanho reduzido e, por esse motivo, ele executa o mínimo de processos possível no espaço do Kernel. Alguns desses processos são executados no espaço do usuário. Com um micronúcleo, se ocorrer um erro, basta reiniciar o serviço que apresentou o problema.

3. **Híbrido**: funciona como um meio-termo, se comparado a sistemas monolíticos e de micronúcleos. O híbrido combina a estabilidade e a segurança do microkernel com o desempenho do monolítico.

4. Monotarefa x Multitarefa

- Um **processo** é uma instância de um programa em execução. Por exemplo, um navegador Web pode estar aberto em cinco janelas diferentes, cada uma com um site. Cada janela é um processo diferente, pois é uma instância diferente do mesmo programa.

- Um sistema **multitarefa** possui a capacidade de executar vários processos simultaneamente. O sistema operacional divide o tempo do processador (CPU) entre os processos para fornecer a ilusão de execução simultânea.

- Lembrando que essa impressão de execução simultânea só ocorre se o sistema for **preemptivo** (capacidade de interromper a execução de uma tarefa para que outra tarefa use o CPU.

- O **escalonamento** (definição da ordem ou prioridade) ocorre pelo uso de algum algoritmo, sendo que cada processador possui o seu.

4. Monotarefa x Multitarefa

- Os sistemas **monotarefa** não permitem a execução de mais de um processo ao mesmo tempo, portanto não é necessário compartilhar o uso do processador. São menos complexos.

- Os processos podem ser executados em **primeiro plano (foreground)** ou em **segundo plano (background)**. Um processo em primeiro plano é aquele que é visível e pode interagir com o usuário (ex.: navegador Web).

- Um processo em segundo plano é aquele que está em execução, mas não é visível ou diretamente acessível ao usuário. Ele continua a funcionar, mas suas operações podem ocorrer sem interação direta do usuário (ex. Serviço de banco de dados).

4. Monotarefa x Multitarefa

- Um outro tipo de processo é conhecido como **daemon** ("serviço"). Este é um tipo especial de processo executado em segundo plano, geralmente sem interação direta com o usuário.

- O termo daemon tem origem do sistema operacional Unix e representa processos que são iniciados durante o boot do sistema e continuam a ser executados enquanto o sistema está ativo.

- Alguns exemplos são o daemon do sistema de impressão (cupsd) e o daemon de agendamento de tarefas (cron).

5. Conceitos Complementares de SO

- **Journaling**: técnica usada em alguns sistemas de arquivos de sistemas operacionais modernos para melhorar a integridade e recuperação de dados em caso de falhas. A ideia é manter um “jornal" (ou log) que registra as operações que serão realizadas antes de serem efetivamente aplicadas no sistema de arquivos. Isso ajuda a garantir a consistência dos dados em caso de falhas (falta de energia, panes do sistema etc.).

- **Spool de impressão**: refere-se a um sistema que permite que vários trabalhos de impressão sejam enviados para uma fila e processados em ordem. A ideia é melhorar a eficiência do processo de impressão, armazenado temporariamente os arquivos em um local (“spool”). Enquanto um trabalho estiver no spool, ele pode ser cancelado, pausado e até mesmo priorizado.

- Além disso, o SO oferecem uma série de outros serviços, como: manutenção da data/hora, lista dos usuários, serviços de acessibilidade, sistema de segurança, controle de acesso e permissões, firewall, etc.

6. Gerenciamento de Processos

- Um SO permite que diversas atividades sejam realizadas ao mesmo tempo, mesmo que a máquina possua apenas um processador.

- Isso é possível devido ao pseudoparalelismo que existe com o escalonamento de uso do processador, sendo que cada processo recebe uma fatia de tempo, de acordo com alguma política ou algoritmo.

- Lembrando que processos são uma instância de um programa em execução. Eles são executados por um tempo no processador (time slice).

- Após o fim de sua time slice, é feita a chamada **troca de contexto**, onde o processo passa o uso do processador para que outro processo possa ser executado.

- Após algum tempo, o processo ganha mais uma time slice para continuar sua execução. Isso chama-se **multitarefa** ou **multiprogramação**.

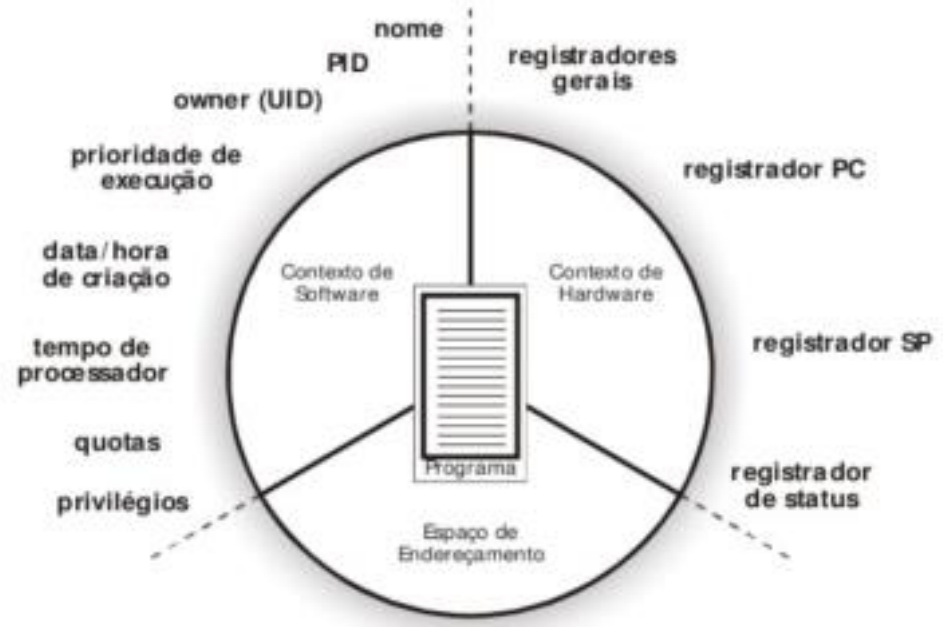
7. Criação e Estrutura de Processos

- Um processo é criado sempre que abrimos um programa (Word por exemplo). Dentro" de um processo, além do código do programa em si, podemos encontrar os seguintes componentes:

1. **Contexto de software**: informações como nome do processo, identificador (PID), proprietário (owner - UID), prioridade de execução, entre outros;

2. **Contexto de hardware**: valores de registradores;

3. **Espaço de endereçamento**: espaço reservado para os dados do processo (ex.: texto editado através do Word).



7. Criação e Estrutura de Processos

- No Kernel do SO, existe uma estrutura que serve para armazenar as informações necessárias para tratar um determinado processo. Esta estrutura é conhecida como **Bloco de Controle do Processo** (BCP, ou PCB - Proccess Control Block).

- O PCB possui informações críticas do processo (identificador, registradores, prioridade, etc.) e deve ficar armazenado em uma área da memória protegida de usuários.

- A **Tabela de Processo** é uma estrutura de dados responsável por habilitar o sistema operacional a localizar e acessar rapidamente o bloco de controle de processo (PCB) de um processo.

7. Criação e Estrutura de Processos

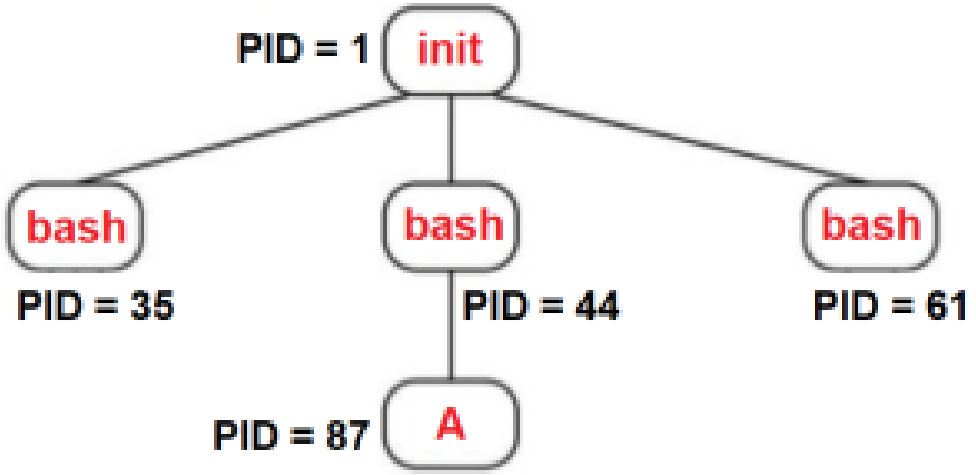
- Em determinado momento ocorre o término do processo, normalmente devido a alguma das seguintes situações:

1. Término normal (voluntário);
2. Término por erro (voluntário), ex.: divisão por zero;
3. Erro fatal (involuntário), ex.: programa recebe como parâmetro o nome de um arquivo que não existe;
4. Eliminado por outro processo (involuntário), ex.: comando kill (Linux).

8. Hierarquia de Processos

- Em relação à hierarquia, cada processo possui apenas 1 pai (processo que o gerou) e 0 ou mais filhos. Por exemplo, no Linux o processo “init” (PID = 1) é o primeiro a ser executado (após o boot). A função dele é controlar todos os outros processos que são executados no computador.

- Digamos que a partir do "init" sejam abertos 3 shells (bash) e a partir de um deles seja executado um programa “A”. A figura abaixo representa esta hierarquia:



9. Estado de um Processo

- Um processo pode estar em três estados distintos:

1. **Executando**: realmente utilizando o processador (em execução);

2. **Pronto**: temporariamente parado, mas pronto (aguardando em “uma fila”) para utilizar o processador;

3. **Bloqueado** (ou em espera): incapaz de executar até que algum evento ocorra (ex.: terminar de receber os dados de um arquivo do HD).

- As quatro trocas possíveis entre os estados são apresentadas na figura abaixo:

9. Estado de um Processo

- Dependendo do comportamento e do tipo do processo, ele ficará mais tempo bloqueado, executando ou pronto.

- Processos que passam a maior parte computando são conhecidos como **CPU-bound**. Eles tendem a ir poucas vezes para o estado bloqueado, pois utilizam muito o processador e quando a fatia de tempo termina vão para o estado “pronto”. Ex: software de cálculos matemáticos

- Já os processos que esperam muito por E/S, conhecidos como **I/O-bound**, tendem a ficarem bloqueados seguidamente, indo depois para o estado “pronto” e somente depois poderem utilizar a CPU.

10. Threads

- Um **processo** é um programa em execução, com seus próprios recursos (como memória, arquivos abertos, etc). Uma **thread** é a **menor unidade de execução** dentro de um processo.

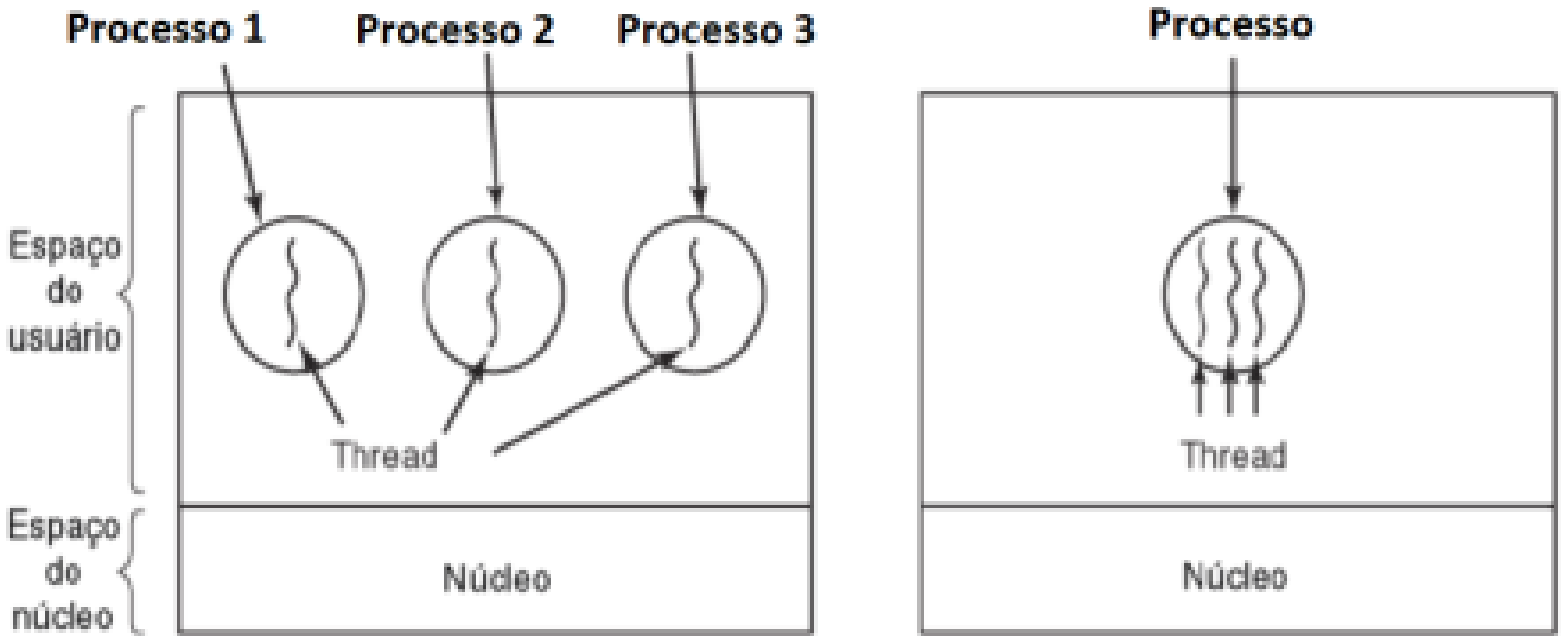
- Uma **thread** é como um “fio” dentro desse processo que executa instruções. Um processo pode ter **uma ou várias threads**. Quando tem mais de uma, dizemos que ele é **multithread**.

- As threads compartilham o mesmo espaço de memória do processo, mas elas podem ser executadas de forma paralela, agilizando a execução do processo como um todo.

- Imagine um navegador, ele pode ter uma thread para carregar a página, outra pode reproduzir um vídeo, outra pode lidar com a entrada do teclado. Tudo isso **dentro do mesmo processo** do navegador.

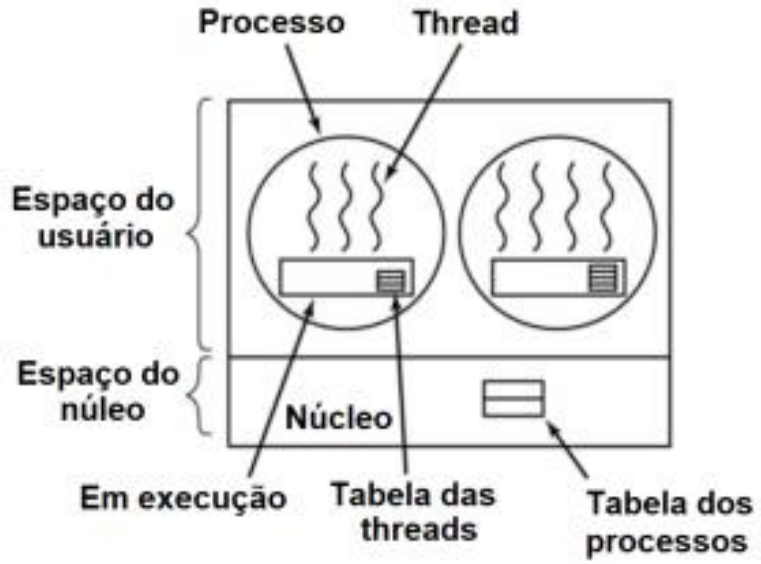
10. Threads

- Como é possível ver na imagem, podemos criar vários processos para executar determinada tarefa (no geral mais custoso) ou então ter apenas um processo com várias threads:



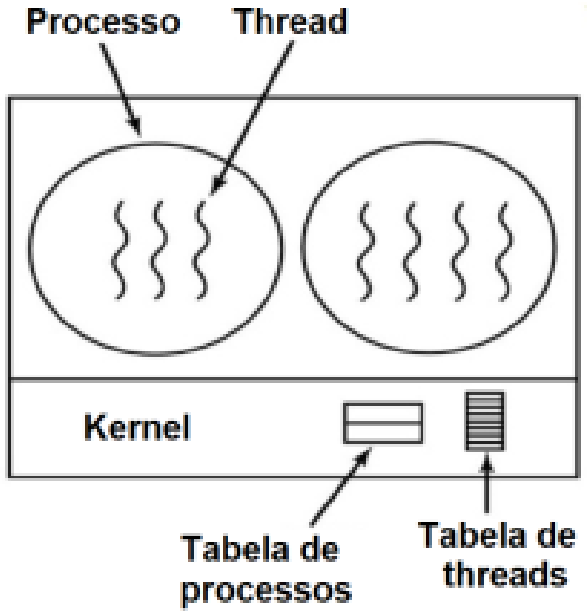
10. Threads

- Geralmente as threads são divididas em duas categorias: thread ao nível do **usuário** e thread ao nível do **kernel** (núcleo).
- Quando falamos em thread ao nível de usuário significa que o controle e escalonamento das threads fica a cargo do espaço do usuário. Na figura podemos ver que a tabelas dos processos fica no kernel, mas as tabelas das threads de cada processo ficam nos próprios processos:



10. Threads

- Já as threads ao nível do kernel queremos dizer que o controle das threads fica a cargo do kernel, ou seja, tanto a tabela de processos quanto a tabela de threads ficam no kernel:



11. Comunicação entre Processos

- Os processos frequentemente necessitam se comunicar. Um exemplo simples de entender é quando um usuário utiliza uma sequência de comandos no terminal do Linux utilizando o pipe separando cada um deles.

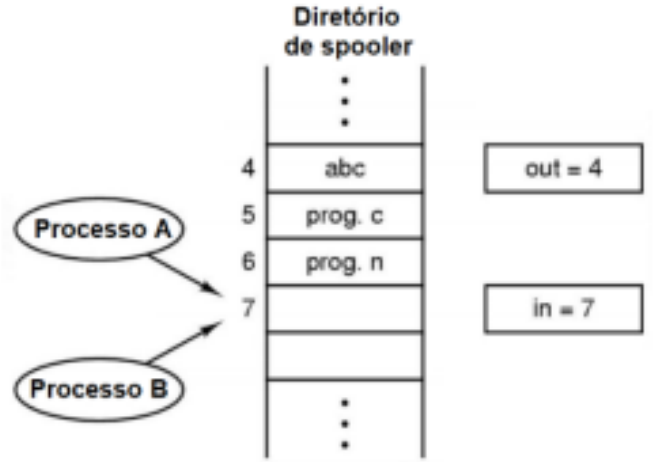
- Com isso a saída de um comando é a entrada do seguinte. Ex: $ ls | grep x | sort -r | tee saida.txt

- No exemplo acima, o resultado do “ls” é a entrada para o comando “grep x”. O resultado é a entrada para “sort -r” e o resultado deste último é a entrada para “tee arquivo_saida”.

- Em resumo, o comando lista arquivos no diretório atual (ls), filtra apenas os que contêm a letra "x" no nome (grep), ordena esses nomes em ordem decrescente (sort), exibe o resultado no terminal e também salva no arquivo `saida.txt`.

11. Comunicação entre Processos

- **Condições de corrida**: Alguns processos que trabalham juntos podem compartilhar espaço de armazenamento (HD ou RAM), ou seja, cada processo pode ler e escrever nesse local.
- Como exemplo podemos citar o diretório de spooler, onde os arquivos são colocados para que o daemon de impressora verifique de tempos em tempos o que deve ser impresso.
- Imagine a situação em que dois processos (A e B) quase que simultaneamente decidem enviar para o diretório de spooler o arquivo a ser impresso:



11. Comunicação entre Processos

- **Seções críticas**: Para evitar as condições de corrida ou outras situações que envolvem memória compartilhada, deve-se encontrar uma maneira de proibir que mais de um processo leia e modifique dados compartilhados ao mesmo tempo.

- Para isso é necessária uma **exclusão mútua**. O problema mostrado na figura anterior ocorreu porque o processo B começou a utilizar uma das variáveis compartilhadas antes que o processo A tivesse terminado de trabalhar com ela.

- Nem sempre o sistema acessa alguma região compartilhada, mas quando isso ocorre ele está acessando uma **região crítica** ou seção crítica.

11. Comunicação entre Processos

- Para evitar problemas nas regiões críticas, são necessárias quatro condições:

1. Dois processos não podem estar ao mesmo tempo dentro de uma seção crítica;

2. Nenhuma suposição pode ser realizada sobre as velocidades ou sobre o número de processadores;

3. Nenhum processo que está sendo executado fora de uma seção crítica pode bloquear outros processos;

4. Nenhum processo deve ter que esperar eternamente para entrar em uma seção crítica.

11. Comunicação entre Processos

- **Semáforo**: é um tipo de variável inteira que controla quantos processos podem entrar na região critica. As operação **down** decrementa o semáforo, e a operação **up** incrementa. Isso evita as condições de corrida.

- Como exemplo: há um semáforo que está com o valor 0 (já tem processo(s) ocupando a região crítica). Outros 3 chegam a essa região e “tentam entrar”, ficando bloqueados.

- Um processo sai da região crítica, é aplicada a operação up (valor do semáforo passa para 1) e um dos três processos pode entrar (de acordo com a fila).

- Quando tal processo entrar é aplicada a operação down (semáforo passa a ser 0).

- **Resumindo**: o valor do semáforo diz quantos processos podem entrar.

11. Comunicação entre Processos

- **Mutex**: É uma técnica simplificada do semáforo, que utiliza apenas os estados “livre” ou “ocupado”. Com isso, é possível representá-lo com apenas 1 bit.

- Quando um processo precisa entrar na região crítica, ele chama **mutex_lock**, que será bem-sucedida e bloqueará a região se estiver livre.

- Caso contrário, o processo ficará bloqueado até que o processo que estiver na região crítica saia e faça o **mutex_unlock** (desbloquear).

12. Escalonamento de Processos

- **Escalonador** (scheduler): algoritmo responsável por determinar a ordem de execução dos processos. Ele gerencia a alocação de recursos da CPU entre os vários processos concorrentes.

- **Throughput** ("Taxa de Transferência"): critério de escalonamento que representa o número de processos executados em um determinado intervalo de tempo. Quanto maior o número de processos que podem ser concluídos em um intervalo de tempo, maior é o throughput.

- **Preempção**: quando um processo é retirado do CPU para dar a vez a outro processo. Nenhum processo pode monopolizar o processador, senão gera starvation.

- **Starvation** (inanição, ou privação): Ocorre quando um processo por algum motivo (geralmente baixa prioridade) não consegue ser executado, pois sempre existe outro processo com mais prioridade.

12. Escalonamento de Processos

- **Sistema multitarefa preemptivo:** um sistema que possibilita a execução de mais de um processo ao mesmo tempo, fazendo a preempção quando um processo ultrapassa sua time slice, e alocando o CPU para outro processo.

- **Tempo de turnaround**: tempo de existência de um processo, ou seja, o tempo desde a sua criação até seu término. As políticas de escalonamento buscam minimizar o tempo de turnaround.

12. Escalonamento de Processos

- As categorias de algoritmos de escalonamento são:

1. **Lote**: geralmente utilizado em computadores de grande porte, sem usuários esperando uma resposta rápida. São aceitáveis algoritmos não-preemptivos ou algoritmos preemptivos com longos períodos de tempo para cada processo. Isso reduz as trocas de processo, o que melhora o desempenho;

2. **Interativo**: em um ambiente em que os usuários interagem, a preempção é fundamental para que não ocorre uma monopolização da CPU por um processo

3. **Tempo real**: Neste tipo de aplicação os processos sabem que não podem ser executados por longos períodos de tempo. Normalmente os processos realizam suas atividades e são rapidamente bloqueados. Ex: radar que registra a velocidade do veículo e fotografa se ultrapassar um determinado limite.

12. Escalonamento de Processos

Escalonamento de sistemas de Lote:

- Os algoritmos mais utilizados para sistemas de lote são:

1. First-Come First-Served (FCFS): O primeiro processo a chegar é o primeiro a ser executado. Basicamente os processos prontos são executados na ordem da fila e quando um processo é bloqueado (aguardando uma E/S), ele retorna para o fim da fila quando estiver pronto novamente.

Problema: Se um processo começar a ser executado, nunca for bloqueado e tiver uma estimativa de execução de 4h, ele monopolizará o processador (algoritmo é não-preemptivo, uma característica de sistemas de lote.

2. Shortest-Job First (SJF): Os processos mais curtos (preciso de menos tempo de CPU) são colocados na frente de processos maiores na fila do escalonamento para execução.

Problema: Starvation (um processo grande nunca é executado porque sempre tem um processo menor)

12. Escalonamento de Processos

Escalonamento de sistemas Interativo:

- Os algoritmos mais utilizados para sistemas interativos são:

1. **Round-robin**: É o algoritmo mais conhecido, ele realiza um rodízio entre os processos, sendo que a cada processo é atribuído um intervalo de tempo (quantum), durante o qual ele pode ser executado. Se ao final do quantum o processo ainda estiver em execução é realizada a preempção da CPU para que ela possa executar outro processo.

Observação: A duração do quantum é importante, pois trocar de um processo para outro exige uma quantidade de tempo para salvar e carregar registradores e mapas de memória, atualizar tabelas e listas, etc. Vamos supor que esse chaveamento de contexto demore 1ms e que o quantum seja de 9ms. Nesse caso, 10% da CPU seriam desperdiçados em sobrecarga administrativa.

12. Escalonamento de Processos

2. Escalonamento por prioridade: Cada processo recebe uma prioridade e o processo na "fila de pronto" com a maior prioridade tem a permissão para executar. Deve-se cuidar com o surgimento de starvation.

3. Escalonamento por múltiplas filas: Os processos são agrupados em classes e cada classe possui uma prioridade: 1 quantum, 2 quanta (o plural de quantum), 4, 8, e assim por diante. Na medida em que um processo utiliza todos os quanta destinados a ele, ele é movido para a classe seguinte (a que recebe mais quanta). Por exemplo, se um processo precisa de 50 quanta, primeiro ele recebe 1, depois 2, 4, 8, 16, 32.

4. Escalonamento garantido: Se houver N usuários trabalhando em uma CPU, cada um recebe cerca de 1/N do seu poder de processamento. O sistema monitora quanto da CPU cada processo de cada usuário utilizou e oferece mais poder de CPU para quem teve menos.

13. Deadlock

- Um **deadlock** (impasse) ocorre quando um determinado processo A está esperando por um evento que o processo B deve gerar, enquanto o mesmo processo B aguarda um evento que somente o processo A pode prover.

- Observe que os dois processos nunca serão finalizados, pois um ficará aguardando o outro por um período indeterminado.

- Por isso os sistemas operacionais possuem a capacidade de garantir (por algum tempo) que um processo tenha o acesso exclusivo a determinados recursos, sejam de hardware ou de software.

13. Deadlock

- Para que um deadlock ocorra, precisa existir as chamadas quatro condições (de Coffman):

1. **Condição de exclusão mútua**: cada recurso ou está correntemente atribuído a exatamente um processo ou está disponível.

2. **Condição de posse e espera**: os processos que possuem recursos garantidos anteriormente podem solicitar novos recursos (um acumulador de recursos!).

3. **Ausência de preempção**: os recursos garantidos não podem ser retirados à força de um processo.

4. **Condição de espera circular**: um encadeamento circular de dois ou mais processos, cada um esperando por um recurso mantido pelo próximo do encadeamento.

14. Gerenciamento de Processos no Windows 11

- O Windows usa um escalonador chamado: **Preemptivo baseado em prioridades + round-robin.** Ele trabalha com **32 níveis de prioridade**: **0–15 → processos normais (usuário) e 16–31 → processos do sistema (tempo real)**

- Por exemplo, um navegador recebe prioridade média, já um driver de hardware recebe prioridade alta.

- O sistema sempre escolhe o processo com **maior prioridade,** mas se houver vários com a mesma prioridade, utiliza o **round-robin** (cada um ganha um “tempo de CPU”).

- O Windows **altera a prioridade automaticamente,** onde programas interativos ganham prioridade maior (para responder rápido) e processos em segundo plano perdem prioridade. Isso melhora a **experiência do usuário**

15. Gerenciamento de Processos no Ubuntu

- O Ubuntu usa o escalonador do kernel Linux chamado: **Completely Fair Scheduler (CFS). Est técnica** tenta ser **justo (fair),** de forma que cada processo deve receber uma “parte igual” da CPU ao longo do tempo.

- O ubuntu usa uma estrutura chamada: **árvore vermelha-negra (red-black tree),** onde os processos são ordenados pelo tempo que já usaram CPU . Quem usou **menos CPU** ganha prioridade.

- Cada processo tem um contador de uso do CPU chamado vruntime. Se rodou muito tem vruntime alto, do contrário vruntime baixo. O sistema sempre escolhe o processo com **menor vruntime.**

- No Linux, é possível definir prioridade com o comando `nice` (−20 a +19). Mas diferentemente do Windows, não é uma prioridade “absoluta”, é apenas um **peso na distribuição da CPU.**

15. Gerenciamento de Processos no Ubuntu

- A tabela abaixo faz um comparativa entre os dois sistemas:

Característica	Windows 11	Ubuntu 24 (Linux)
Tipo	Prioridade + round-robin	Justiça (CFS)
Foco	Responsividade	Equidade
Prioridade	32 níveis fixos	Peso (nice)
Ajuste automático	Sim (dinâmico)	Menos agressivo
Estrutura interna	Filas por prioridade	Árvore rubro-negra
Escolha do processo	Maior prioridade	Menor vruntime

16. Comparação no Gerenciamento de Processos

- Ambos os sistemas evitam problemas com deadlock por meio de técnicas de timeout.

- Já o **starvation**, o Windows trata isso muito bem com o uso da técnicas Boost dinâmico de prioridade**.** Se um processo fica esperando muito, o sistema **aumenta sua prioridade temporariamente.**

- Com relação ao Ubuntu, o **Completely Fair Scheduler** já foi criado para evitar starvation. No Linux, a prioridade (nice) não bloqueia execução, apenas altera **quanto tempo recebe.** Isso é crucial para evitar starvation.

- É importante reforçar que o Linux não dá exatamente o mesmo tempo de CPU para todos. Ele dá tempo **proporcional ao peso (prioridade).** Assim, todos executam, mas não igualmente.

16. Comparação no Gerenciamento de Processos

- Essa técnica do Linux deixa o gerenciamento de processos mais estável, pois se um processo crítico entra em loop infinito, no Windows (prioridade alta) pode travar tudo, no Linux ele não monopoliza totalmente.

- Para servidores, o Linux possibilita que serviços diferentes coexistam melhor, nenhum serviço derruba os outros. Starvation também é praticamente impossível de ocorrer.

- Em contrapartida, se o servidor tem um processo **extremamente prioritário** (controle de robô, sensor) o CFS pode não ser eficiente.

17. Comandos Linux - ps

- O comando "**ps**" no Linux (Process Status) é usado para listar os processos em execução no sistema.

- A forma mais comum de usar o ps é: "os aux", onde a mostra processos de **todos os usuários,** u mostra em formato detalhado (inclui usuário, CPU, memória...) e x inclui processos **sem terminal (daemons/serviços).**

```
#forma mais comum de usar
ps aux
```

```
root@notebook1:~# ps ax
    PID TTY      STAT   TIME COMMAND
      1 ?        Ss     0:01 /sbin/init
      2 ?        S      0:00 [kthreadd]
      3 ?        S      0:00 [pool_workqueue_release]
      4 ?        I<     0:00 [kworker/R-kvfree_rcu_reclaim]
      5 ?        I<     0:00 [kworker/R-rcu_gp]
```

17. Comandos Linux - ps

- Também é possível variar o comando mostrando apenas as 10 primeras linhas (head) ou ordenar por algum parâmetro, como mostra o quadro:

```
#mpstra apenas os 10 primeiros processos
ps aux | head

#ordena por uso do cpu
ps aux --sort=-%cpu | head
```

18. Comandos Linux - kill

- Apesar do nome, o `kill` **não serve só para "matar" processos,** ele envia **sinais (signals)** para um processo. Para usar o kill, basta indicar o código do sinal e o PID do processo.

```
#pede "educadamente" para o processo 2021 ser encerrado (salva dados)
#é a opção padrão
kill -15 2021 ou kill 2021

#Encerra o processo imediatamente, não pode ser ignorado
kill -9 2021

#-19 pausa o processo e –18 continua o processo
kill -19 2021

#reinicia o processo
kill -1 2021
```

19. Comandos Linux – Outros Comandos

- O comando "**jobs**" mostra os processos executados em segundo plano.

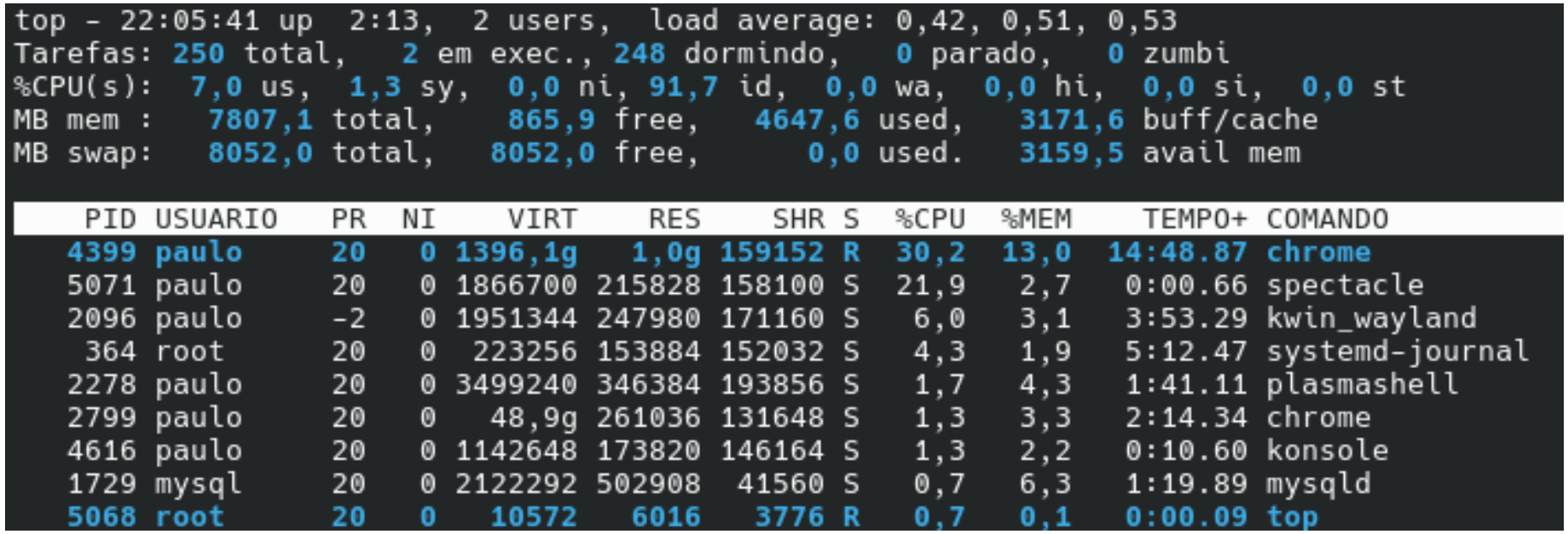
- O comando "time" colocando antes de qualquer comando, mede o tempo que levou a execução. Por exemplo: time updatedb

- O comando "nohup" permite que o comando continue a ser executado, mesmo que o usuário faça logout. Exemplo: nohup updatedb & -executa o comando em segundo plano mesmo com logouto do usuário.

20. Comandos Linux - top

- O comando "top" é o mais usado para verificar os processos. É uma interface que mostra os processos e o uso de hardware em tempo real.

#ordena pelo uso do cpu. Se for "m" ordena pela memória
top p



21. Comandos Linux - renice

- O comando "**renice**" muda a prioridade do processo. Pode variar de –20 (alta) até 19 que é baixa. Usuário que não é root, só pode colocar de 0 até 19.

```
#coloca prioridade mais alta para o processo 2021
renice –20 2021
```

22. Comandos Linux - nice

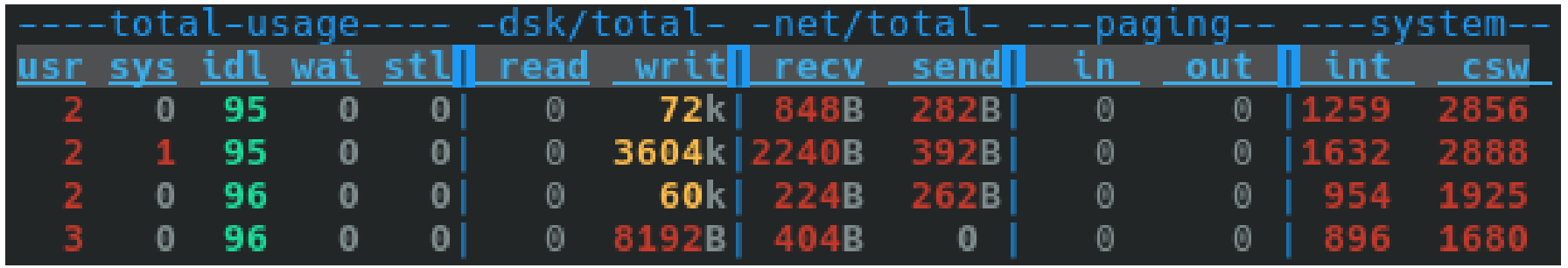
- O comando "**nice**" é parecido com "renice". Mas ele cria o processo já com a prioridade, enquanto o renice muda a prioridade de um processo em execução.

```
#faz um updatedp com prioridade 10
nice 10 updatedb
```

23. Comandos Linux - dstat

- O comando "**dstat**" mostra o throughput geral (Monitora CPU, memória, uso de disco (E/S) e rede). Ele faz o monitoramento de recursos do sistema em tempo real, utilizada para coletar e exibir estatísticas de desempenho.

```
sudo apt install dstat
dstat
```



23. Comandos Linux - dstat

- O resultado do "**dstat**" mostra alguns parâmetros:

🔹 1. CPU → `usr sys idl wai stl`

Esses valores são **percentuais (%) de uso da CPU**:

- **usr (user)**
  → Tempo executando programas do usuário
  Ex: seu código Python, navegador
- **sys (system)**
  → Tempo gasto pelo kernel (sistema operacional)
- **idl (idle)**
  → Tempo ocioso (CPU parada)
- **wai (iowait)**
  → Tempo esperando operações de I/O (disco, rede)
  ⚠️ Se estiver alto → gargalo de disco ou rede
- **stl (steal)**
  → Tempo "roubado" pela virtualização (quando você está em VM)

🔹 2. Disco → `read writ`

- **read**
  → Dados lidos do disco (por segundo)
- **writ (write)**
  → Dados escritos no disco (por segundo)

👉 Exemplo:

```
read 1M  writ 500k
```

→ 1 MB/s leitura, 500 KB/s escrita

🔹 3. Rede → `recv send`

- **recv (receive)**
  → Dados recebidos pela rede
- **send**
  → Dados enviados

👉 Exemplo:

```
recv 2M  send 300k
```

→ Download de 2 MB/s, upload de 300 KB/s

🔹 4. Sistema → `in out`

- **in (interrupts)**
  → Interrupções por segundo (hardware chamando CPU)
- **out (context switches)**
  → Trocas de contexto entre processos

👉 Isso mostra "atividade interna" do sistema

🔹 5. Kernel → `int csw`

- **int (interrupts)**
  → Número de interrupções por segundo
- **csw (context switches)**
  → Quantas vezes a CPU trocou de processo

👉 Valores altos podem indicar:

- muitos processos concorrendo
- sobrecarga no sistema