**Sistemas Operacionais**

****

|  |  |  |  |
| --- | --- | --- | --- |
| **Digitado por:** | **Professor:** | **Versão** | **Ano** |
| Ricardo Kim | Nicolas | 1.0 | 2017 |

**ÍNDICE**

[1. Ementa, Apresentação e Observações. 4](#_Toc497919092)

[2. Visão Geral de Sistemas Operacionais (SO) 5](#_Toc497919093)

[2.1. S.O. entendido como Máquina Simplificada 5](#_Toc497919094)

[2.2. S.O. entendido como gerente de recursos 5](#_Toc497919095)

[2.3. Breve histórico 5](#_Toc497919096)

[3. Sistemas Programaveis 6](#_Toc497919097)

[3.1. Sistema Mono Programável 6](#_Toc497919098)

[3.2. Sistema Multi Programável 7](#_Toc497919099)

[3.2.1. Sistemas de tempo Compartilhado 7](#_Toc497919100)

[3.2.2. Sistemas de Tempo Real 8](#_Toc497919101)

[3.3. Sistemas com Múltiplos Processadores. 9](#_Toc497919102)

[3.3.1. Sistemas fortemente acoplados. 10](#_Toc497919103)

[3.3.2. Sistemas fracamente acoplados. 11](#_Toc497919104)

[3.3.3. Vantagens 11](#_Toc497919105)

[3.3.4. Desvantagens. 11](#_Toc497919106)

[4. Processamento paralelo 12](#_Toc497919107)

[4.1. Algumas limitações 12](#_Toc497919108)

[4.2. Utilização dos sistemas fracamente acoplados. 12](#_Toc497919109)

[4.2.1. Sistema Multi Programáveis: 12](#_Toc497919110)

[4.3. Técnica DMA (Direct Memory Acess) 14](#_Toc497919111)

[4.4. Interrupção e Exceção 15](#_Toc497919112)

[4.4.1. Técnica de Buffering 16](#_Toc497919113)

[4.4.2. Técnica de Spooling 17](#_Toc497919114)

[4.4.3. Retratância 17](#_Toc497919115)

[5. Núcleo do sistema operacional 18](#_Toc497919116)

[5.1. Modo de acesso 19](#_Toc497919117)

[5.2. Instruções privilegiadas 20](#_Toc497919118)

[5.3. POSIX (portable operate system for unix) 22](#_Toc497919119)

[6. Processos 23](#_Toc497919120)

[6.1. Estrutura do Processo 24](#_Toc497919121)

[6.1.1. Contexto de Hardware 24](#_Toc497919122)

[6.1.2. Contexto de Software 24](#_Toc497919123)

[6.1.3. Espaço de endereçamento 25](#_Toc497919124)

[7. Estados dos Processos 26](#_Toc497919125)

[7.1. Diagrama de Estados dos Processos 26](#_Toc497919126)

[7.2. Processos do Tipo 26](#_Toc497919127)

[7.2.1. CPU - Bound 26](#_Toc497919128)

[7.2.2. I/O - Bound 27](#_Toc497919129)

[7.2.3. Foreground 27](#_Toc497919130)

[7.2.4. Background 27](#_Toc497919131)

[8. Processos e Threads 27](#_Toc497919132)

[8.1. Programa, Memória 28](#_Toc497919133)

[8.2. Threads versus Processos 29](#_Toc497919134)

[8.2.1. Exemplo novo Processo 29](#_Toc497919135)

[8.2.2. Exemplo novo Threads 30](#_Toc497919136)

[8.3. Problemas de concorrência 33](#_Toc497919137)

[9. Sincronização e comunicação entre processos 34](#_Toc497919138)

[9.1. Problemas do compartilharmento de recursos 35](#_Toc497919139)

[9.2. Exclusão Mútua em Sistema de Memória Compartilhada 36](#_Toc497919140)

[10. Gerenciamento do processador 38](#_Toc497919141)

[10.1. Política do escalonamento 38](#_Toc497919142)

[10.2. Critérios do escalonamento 38](#_Toc497919143)

# Ementa, Apresentação e Observações.

* Bibliografia: Arquitetura de Sistemas Operacionais ed4, Francis B Machado e Talli.
  + Visão Geral de SOs
  + Concorrência
  + Processos
  + Comunicação entre processos
  + Gerência de memória
* Prova Intermediária 1.
  + Capítulo 1

# Visão Geral de Sistemas Operacionais (SO)

Um SO pode ser entendido como: **Máquina Simplificada** ou **Gerente de Recursos**

* Programa: é destinado ao usuário final (ex: Planilha, Jogos, etc)
* Utilitário: é complemento do SO (ex: DOS: coype, type; Linux: cp, cat)

## S.O. entendido como Máquina Simplificada

O SO permite ao usuário utilizar o sistema computacional de forma a abstrair (“esconder”) a complexidade desse sistema computacional

ex: ao se gravar (salvar) um arquivo o usuário apenas “clica” no ícone para salvar

Por trás dessa ação existem diversas operações tais como:

1. Transformar os Bytes da memória RAM para a controladora do disco.
2. A controladora deve mover o braço de leitura/gravação sobre o disco.
3. Deve-se aguardar a trilha e o setor passar sobre a leitura de gravação e ao gravar.
4. Ao gravar deve-se magnetizar a alimalha de ferro contida no disco. (que é um Plastico).
5. Sendo que o bit 1 é um pólo magnético e o 0 (zero) é de polaridade inversa.

## S.O. entendido como gerente de recursos

Os sistemas computacionais apresentam diversos recursos.

* Processador: Gerente do processador.
* Memória: Gerente de Memória
* Arquivos: Gerente de Arquivos.
* Dispositivos: E/S (entrada/saída), I/O (input/output), Gerente de dispositivos
* Redes: Gerente de Rede.

O SO deve controlar diversos recursos, Portanto, podemos compará-lo ao Maestro de Orquestra.

## Breve histórico

|  |  |
| --- | --- |
| Windows:   1. NS-DOS (CP/M). 2. 3.X (DOS). 3. 95. 4. Me & NT. 5. 2K. | Linux   1. Unix 2. Linux (GNU). 3. Distribuições: 4. Debian & Red Hat. 5. Debian: Ubuntu, Mint. 6. Red Hat: Fedora. |

# Sistemas Programaveis



## Sistema Mono Programável

Esse sistema permite apenas um processo (programa em execução) possa a vir a ser executado.

Esse tipo de sistema foi um dos primeiros a surgirem e apresentava diversas propriedades, porém destacaremos algumas de suas limitações.

* **Limitações quanto à memória principal**
  + A figura abaixo representa esse tipo de sistema.
  + 
  + Desperdicio de memória, isto é a área de memória livre mostrada na figura não podia ser executadas, isso era uma característica indesejavel.
  + Aplicações com tamanho superior de memória destinada a aplicação não podiam ser executadas, isso porque não podiam ficar residentes na memória.
* **Ociosidade do processador**
  + A figura a seguir representar o processador em execução
  + 
  + Podemos notar que enquanto ocorre operação no disco (I/O), a CPU fica ociosa, isso é limitação do sistema.

## Sistema Multi Programável

Esse tipo de sistema permita que dois ou mais processos venham a ser executados de forma concorrente.

Eles apresentam a característica de possuírem a interrupção de dispositivos (I/O)

As limitações referente a memória mostrada no sistemas mono programáveis foram superadas, além disso houve uma diminuição da ociosidade da CPU (para positivo),

Podemos verificar o comportamento de processador analisando a figura seguinte.



Da figura, podemos notar que a CPU passa a executar o processo P2, enquanto o processo P1 faz uso.

Existem diversos tipos de sistemas multi programáveis, porém iremos destacar apenas 2 subtipos.

### Sistemas de tempo Compartilhado

* + Esse tipo de sistema além de apresentar interrupção de dispositivo de I/O (Input/Output), ele também representa interrupção de relógio (clock) Isso introduziu o conceito conhecido como fatia de tempo. Isto é o processador realiza um escalonamento em função do tempo.
  + Para melhor compreender o conceito de fatia de tempo veremos a figura a seguir.
  + 
  + A figura anterior mostra que a CPU escalona (reveza) entre os processos P1 e P2, isso se deve a fatia-de-tempo, fornecida pelo S.O.
  + O usuário tem uma impressão de que os processos P1 e P2 serão executados de forma simultânea (pseudo-paralelismo)
  + 

### Sistemas de Tempo Real

Esse tipo de sistema geralmente é usado para monitorar ambientes que envolvem riscos (Ex: Sistema de controle aéreo, Sistema de controle de caldeiras industriais, Sistema de ferrovias).

**Algumas Características**

Basicamente elas não usam o processo de escalonamento por fatia-de-tempo, a invéz disso, adota-se o critério de prioridade de processo.

Isto é um processo de menor prioridade é interrompido por um processo de maior prioridade.

Esse tipo de sistema apresenta controle rigoroso quanto aos limites de tempo necessário a alocação de recursos. (Não pode haver demanda).

Em geral, esse tipo de sistema permanece permanente monitorando ambientes que existem riscos.

## Sistemas com Múltiplos Processadores.

A arquitetura com múltiplos processadores já a muito tempo tem sido considerada no seu limiar. Isso porque a alta freqüência dos processadores tem gerado calor devido ao efeito Jaule e (elétrons chocam gerando calor).

Os fios delgados dos chips podem apresentar até mesmo efeitos colaterais como os de ordem quântica.

Muito já se fez para aumentar o desempenho dos processadores.

Ex: aumento de freqüência, colocar mais registradores, pipeline.

Porém o gargalo de Von Newmann tem sido um obstáculo para aumentar o desempenho computacional.

O barramento retorna um fator da queda de desempenho



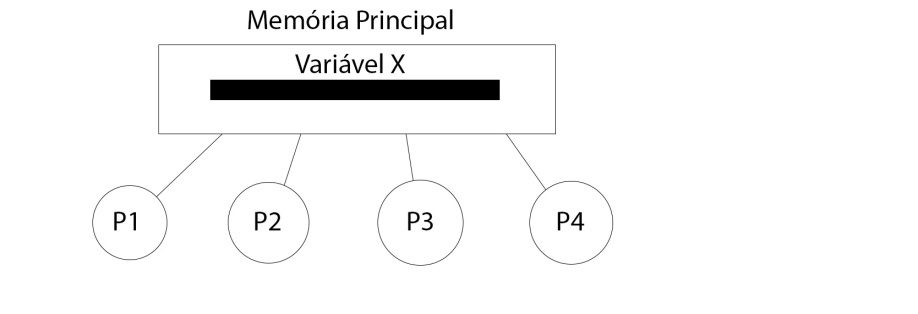
Uma das alternativas para aumentar o desempenho é adotar o processamento paralelo.

Fatores que favorecem processamento paralelo

* Aumento de velocidade
* Lidar com volumes de dados

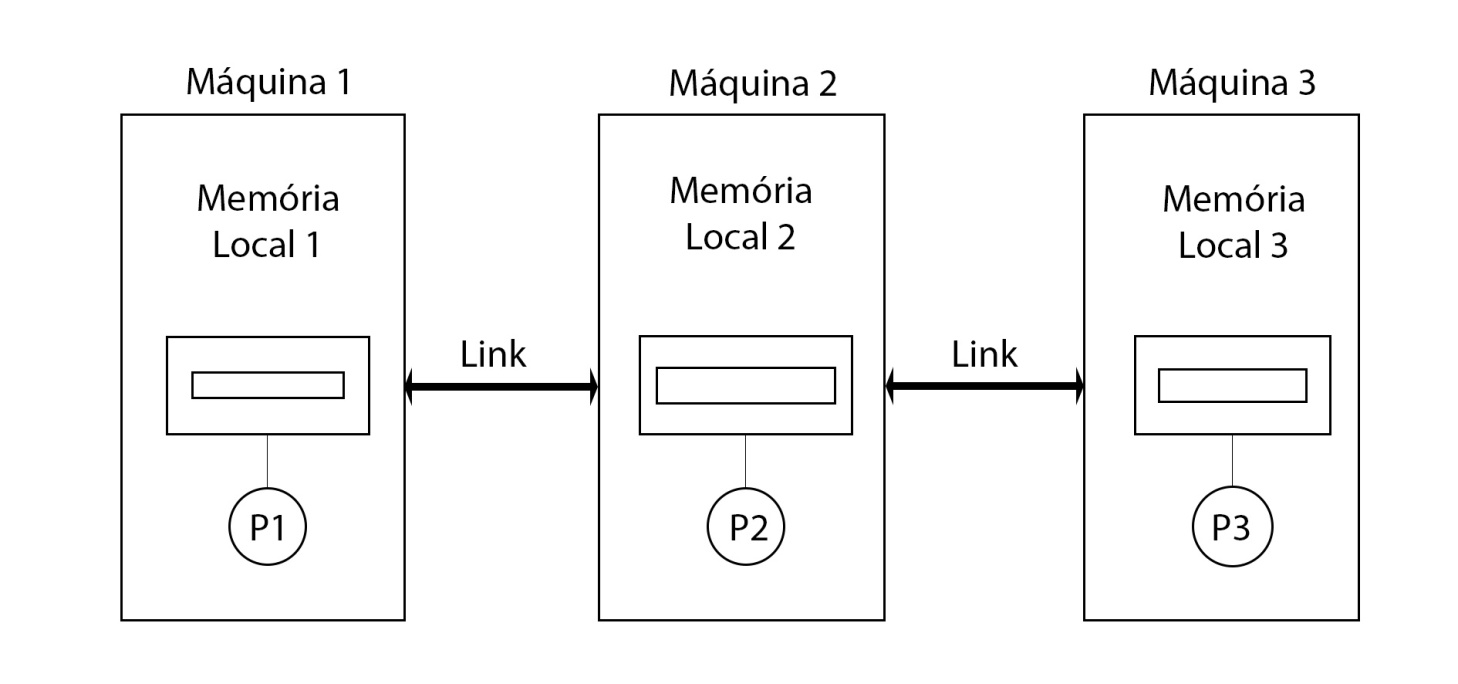
Existem diversas maneiras de se classificar os sistemas de múltiplos processadores, apresentamos a clássica classificação a seguir:

### Sistemas fortemente acoplados.

* + Esse tipo de sistema também é conhecido como sistema de memória compartilhada (Sharing Memory System)
  + 
  + A figura mostra 4 processadores compartilhando o mesmo espaço de endereçamento de memória.
  + Algumas propriedades
    - Apresenta certa velocidade para transmissão de dados via barramento (CPU-Memória).
    - Existe certa facilidade quanto a comunicação e compartilhamento de dados por threads ou processos distintos. (favorecendo processamento paralelo e cooperativo).
  + Algumas dificuldades
    - Devido a memória ser compartilhada pode haver inconsistências devido ao mau uso de variáveis comuns, em áreas de memória em geral
  + Algumas outras alternativas
    - Interface Gráfica
      * GP – General Purpose
      * GPU – Graphic Processor Unit

### Sistemas fracamente acoplados.

Esse tipo de sistema também é conhecido como sistema da memória distribuída (Distribution Memory System).

A forma a seguir representa esse tipo de sistema.

A figura anterior representa três máquinas independentes investigações por link (Rede) de comunicação.

Note que cada máquina apresenta o seu próprio espaço de endereçamento de memória.

Esse tipo de sistema permite utilizar máquinas com baixo desempenho, porém vista de forma agrupada permitem que se enxergue um único sistema computacional com grande desempenho.

### Vantagens

* Alta escalabilidade isto é, pode-se agregar mais equipamento ao sistema sem haver grande esforço.
* Obs: Os sistemas fortemente acoplados nem sempre apresentam escalabilidade

### Desvantagens.

* Possibilidade de distribuir o software por diversos máquianas (locais ou mesmo distantes. Escalabilidade Geográfica) Isso facilita o processamento coorporativo.

# Processamento paralelo

Há uma possibilidade de se realizar processamento paralelo com diversas máquinas independentes. Esse tipo de paralelismo é outra alternativa em relação Sistemas Fortemente Acoplados.

## Algumas limitações

Como a figura anterior mostra, devido a memória ser distribuída o processador P2 não pode fazer acesso a variável X da memória local 1, para isso, há a necessidade de se transmitir o dado via o link da máquina 1 para a máquina 2.

Isso é um fator de baixo desempenho porque a transmissão via rede é mais demorada do que um barramento CPU-Memória (Sistemas Fortemente acoplados). Outro problema é que o dado copiado para a memória local 2 e isso gera replicação de dados podendo ocorrer inconsistências de dados.

Existem certas dificuldades no desenvolvimento de algoritmos distribuídas.

## Utilização dos sistemas fracamente acoplados.

* Redes
* Clusters
* Cerid
* Cloud Computer
* IOT
* BigData
* Smart Cities

### Sistema Multi Programáveis:

Concorrentes

Multi usuário

Permitiram Multi programáveis 🡪 Interrupção dispositivo.

#### Sistemas antigos sem interrupção de dispositivos E/S

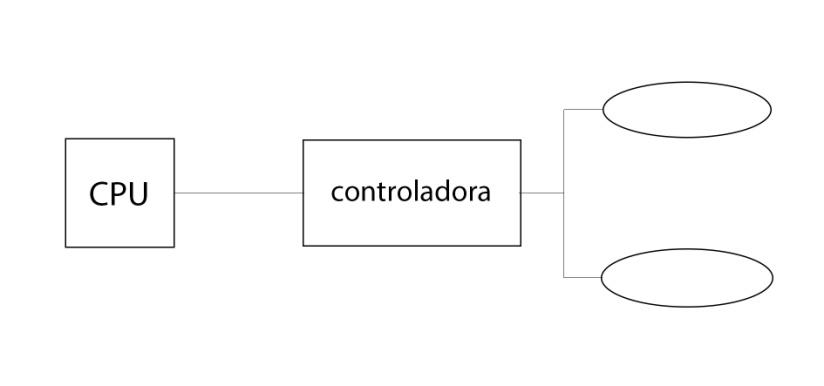
A CPU necessitava apresentar instruções relativas aos dispositivos de E/S cujas instruções dependiam dos distintos tipos de periféricos.

Havia grande dependência entre a CPU e o dispositivo para discos CPU necessitava sem saber volume, cilindro.

Além disso, a CPU ficava em um monitoramento constante em caso de uso do periférico, e ela ficava em loop para monitorar por exemplo, se a leitura do disco encerrou.

Isso fazia com que a CPU ficasse ocupada no loop sem poder executar o outro processo Espera Ocupada (Busy Wait) .

Com o objetivo de aprimorar o sistema, foi feito:

* Uma controladora (interface) de periférico (controladora de disco).
* O conceito de interrupção
* Interface controladora de disco
* 

A CPU fica menos dependente do periférico. Isso gera comando de E/S padronizado e é a CPU não necessita saber do periférico.

#### Introdução do conceito de interrupção

A CPU não monitora o dispositivo de E/S. Com isso a CPU fica livre para executar outro processo enquanto há leitura do disco para o processo anterior. Quando a leitura encerrar a controladora envia sinal eletrônico (interrupção) para CPU com objetivo de avisar que a operação de leitura encerrou com isso a CPU pode retomar o processamento.

## Técnica DMA (Direct Memory Acess)

Como vimos com a evolulção da controladora de dispositivos I/O, os sistemas computacionais tornaram-se mais eficientes.

[Controladora] 🡪 [Controladora] 🡪 [Periféricos]

Seria interessante notar que uma melhoria foi introduzida nas controladoras E/S (I/O).

Essa melhoria refere-se à capacidade da controladora do dispositivo E/S (I/O) poder acessar a Memória Principal para realizar a transferencia de dados entre os dispositivos de E/S.

Para melhor esclarecer isso, vamos verfificar como o sistema funciona, quando ocorre a execução dessa operação de instrução de I/O.

Exemplo:

Suponha que um programa possua uma instrução de leitura de um disco (READ).

A CPU irá deixar a controladora de dispositivo de I/O executar a operação de leitura do disco.

Nesse caso,a controladora irá transferir os dados do disco e grava-los numa área da Memória Principal e ao término da leitura a controladora envia um sinal eletrônico (interrupção) a CPU.

Esse sinal informa a CPU que a leitura ocorreu.

Esse mecânismo em que a controladora transfere dados entre a Memória Principal e o dispositivo de I/O é conhecido como técnica de DMA (Direct Memory Access).

É interessante destacar que as antigas controladoras de dispositivos de I/O **não** tinham a capacidade de lidar com a **Memória Principal** nesse caso a CPU era responsável por transferir os dados da memória e isso dispendia precioso tempo da CPU.

## Interrupção e Exceção

A interrupção/exceção permitiu a implementação de sistemas multi-programáveis. Os dispositivos passam a ser utilizados de forma concorrente entre os processos.

Os autores do livro texto diferenciam as Interrupções das exceções, porém destaca se existem outros autores e fabricantes que fazem distinção entre Interrupção e Exceção.

Por simplificada de, seguiremos a definição do nosso livro texto.

* Interrupção: é um evento assíncrono, isto é, esse evento pode ocorrer de maneira aleatória (não está associado ao tempo). Esse evento é externo as instruções do programa em execução.
  + Exemplo: Ocorre quando um dispositivo de I/O envia sinal informando que a operação de I/O encerrou.
  + Obs.:

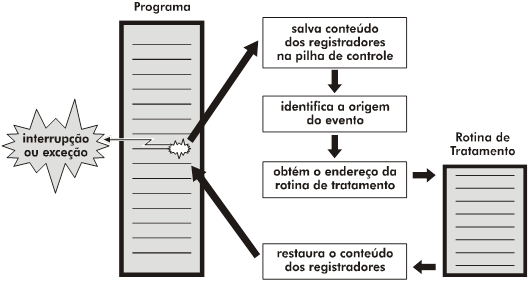
Note que nem sempre é possível prever quando isso ocorre.

* Exceção: é um evento síncrono, que resulta da execução de uma instrução de um programa.
  + Exemplo: uma exceção é quando existe uma divisão por zero. Note que sempre que o programa passar por essa operação haverá a ocorrência da exceção (EVENTO PREVISÍVEL).
  + Obs.:

Ao final da execução de cada instrução do programa, a Unidade de controle verifica se ocorreu uma Interrupção/Exceção.

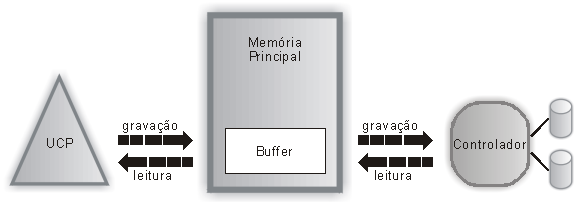
Caso ocorreu a interrupção/Exceção ocorre um desvio do fluxo de execução e algumas ações são executadas conforme a figura a seguir.

Mecanismo de interrupção/exceção



|  |  |
| --- | --- |
| Via Hardware | 1. Um sinal de interrupção é gerado o para o processador 2. Após o término de execução de instrução concorrente o processador identifica o pedido de interrupção 3. O conteúdo do registrador PC (IP) e de status (PSW) é salvo 4. O processador identifica qual a rotina de tratamento será executada e carrega o registrador PC (IP) com o endereço inicial dessa rotina |
| Via Software | 1. A rotina de tratamento salva os demais registradores na pulha de controle do programa 2. A rotina de tratamento é executada 3. Após o término da execução de rotina de tratamento os registradores são restaurados ao processador e retorna a execução do programa interrompido |

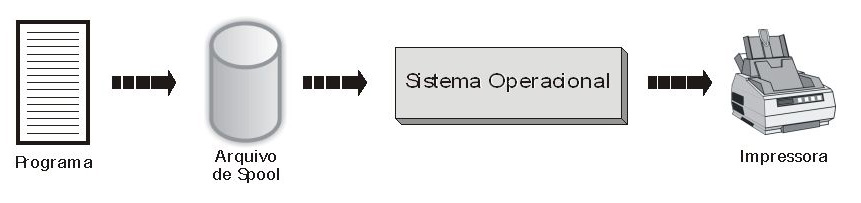
### Técnica de Buffering



Consiste em utilizar uma área na memória principal para transferência de dados entre dispositivos de E/S e a MP

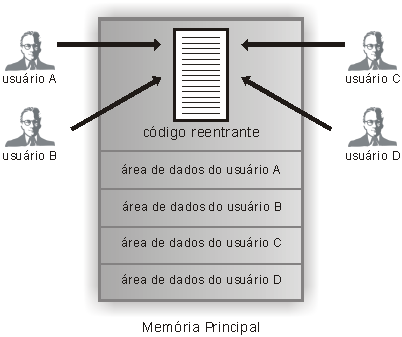
* Numa operação de leitura de dispositivo:
  + Dado é transmitido para buffer sem onerar o processador, dispositivo fica liberado após conclusão
* Numa operação de gravação para o dispositivo:
  + Dado é escrito rapidamente no buffer, liberando o processador para outras atividades enquanto este é gravado no dispositivo.
* Unidade de transferência é o registro, cujo tamanho depende do tipo de dispositivo
  + Vários registros podem ser armazenados num buffer
  + Buffer conterá dados lidos mas ainda não processados, ou dados processados mas ainda não gravados.
* Buffering minimiza problemas decorrentes da disparidade de velocidades, compatibilizando tempos de acesso de maneira que processador e dispositivo de E/S não fiquem ociosos

### Técnica de Spooling



* Simultaneous Peripheral Operation ON-line, instroduzido no fim dos anos de 1950
* Técnica usa uma área do disco (arquivo) para buffering de impressão
* Tudo que é sumetido para impressão vai para o arquivo de spool, liberando o programa de origem
* SO é quem gerencia os Jobs de impressão

### Retratância



* Proteção do sistema
  + SO deve garantir a confiabilidade e a integridade dos programas e dados dos usuários, além do próprio SO
    - Cada programa possui uma área reservada para código e outra para dados que devem ser preservadas durante sua execução
    - Idem para o SO em si
    - Arquivos também devem ter sua integridade garantida quando acessados por múltiplos programas e/ou múltiplos usuários
    - Recursos não devem ser monopolizados

# Núcleo do sistema operacional

Quando falamos a respeito de SOs, na realizadade, estamos nos refereindo ao seu núcleo. Isso uer dier que a interface, os editores, os compiladores e utilitários que vêm na instalação do SO, não fazem parte do núcleo. Esses elementos complementam o SO, mas não serão considerados neste curso como o SO própriamente dito.

O sistema operacional, é um conjunto de rotinas, que ficam permanentemente acionadas ou podem ser ativadas por eventos.

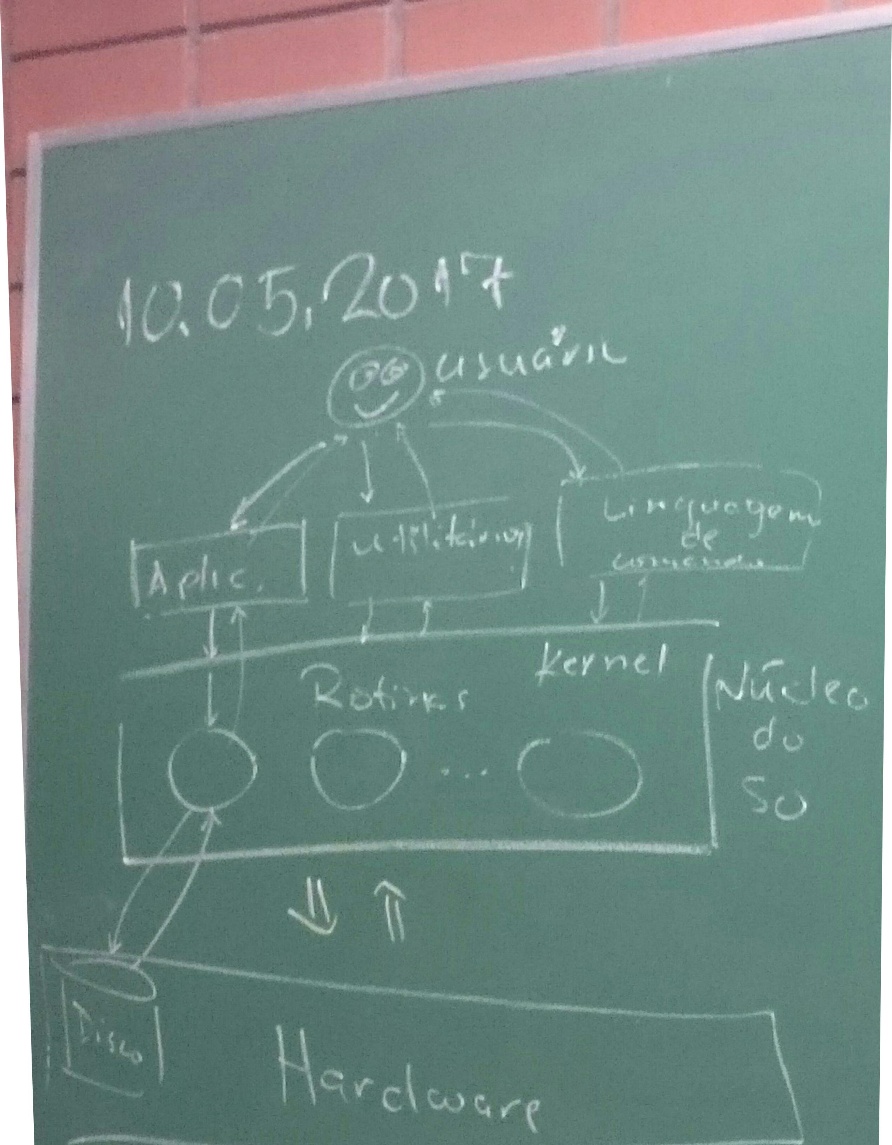
O núcleo do SO (Kernel), pe constituído de um conjunto de rotinas executadas de forma concorrente. Em geral reagem a eventos, e podem ser criadas, ou eliminadas de acordo com os eventos.

Essas rotinas nem sempre são executadas como as aplicações tradicionais, onde essas aplicações têm começo meio e fim. Portanto as rotinas nem sempre apresentam o mesmo padrão de comportamento das aplicações tradicionais.

Funções principais do núcleo do sistema operacional:

* Gereciamento de Interrupções
* Gerenciamento de processos (programas em execução)
  + Criar
  + Remover
  + Mudar de estado
  + Comunicação e sincronização de processos
  + Escalonamento de processos
* Gerência de memória
  + Alocar memória
  + Liberar memória
  + Paginação de memória
* Gerenciamento de dispositivos
  + Montar dispositívo
  + Escrever no dispositívo
  + Ler do dispositivo
  + Desmontar o dispositivo
* Gerencia de arquivos
  + Ler
  + Escrever
  + Gravar
  + Abrir
  + Fechar
  + Remover
  + Atributos
* Gerenciar Rede
* Contabilização do sistema operacional
* Auditoria

## Modo de acesso

****

O modo de acesso está relacionado ao processador (CPU) e serve como uma forma de proteger o SO

De forma simplificada, os processadores podem apresentar os seguintes modos de acesso:

* **Modo núcleo (Kernel)**
  + Quando processador encontra-se nesse modo, ele libera o uso de todas as suas instruções (instruções privilegiadas e também instruções não privilegiadas.
* **Modo usuário (user mode)**
  + Nesse modo o processador “bloqueia” as instruções privilegiadas e apenas libera as instruções não privilegiadas.
  + Obs: Através de bits contidos no registrador de status, pode -se saber qual é o modo de acesso do processador.

## Instruções privilegiadas

São as instruções que podem desestabilizar o SO.

Ex:

* Instruções para:
  + Criar processo e gerenciar processos
  + Gerenciar a memória
  + Gerenciar arquivos
  + Gerenciar dispositivos.

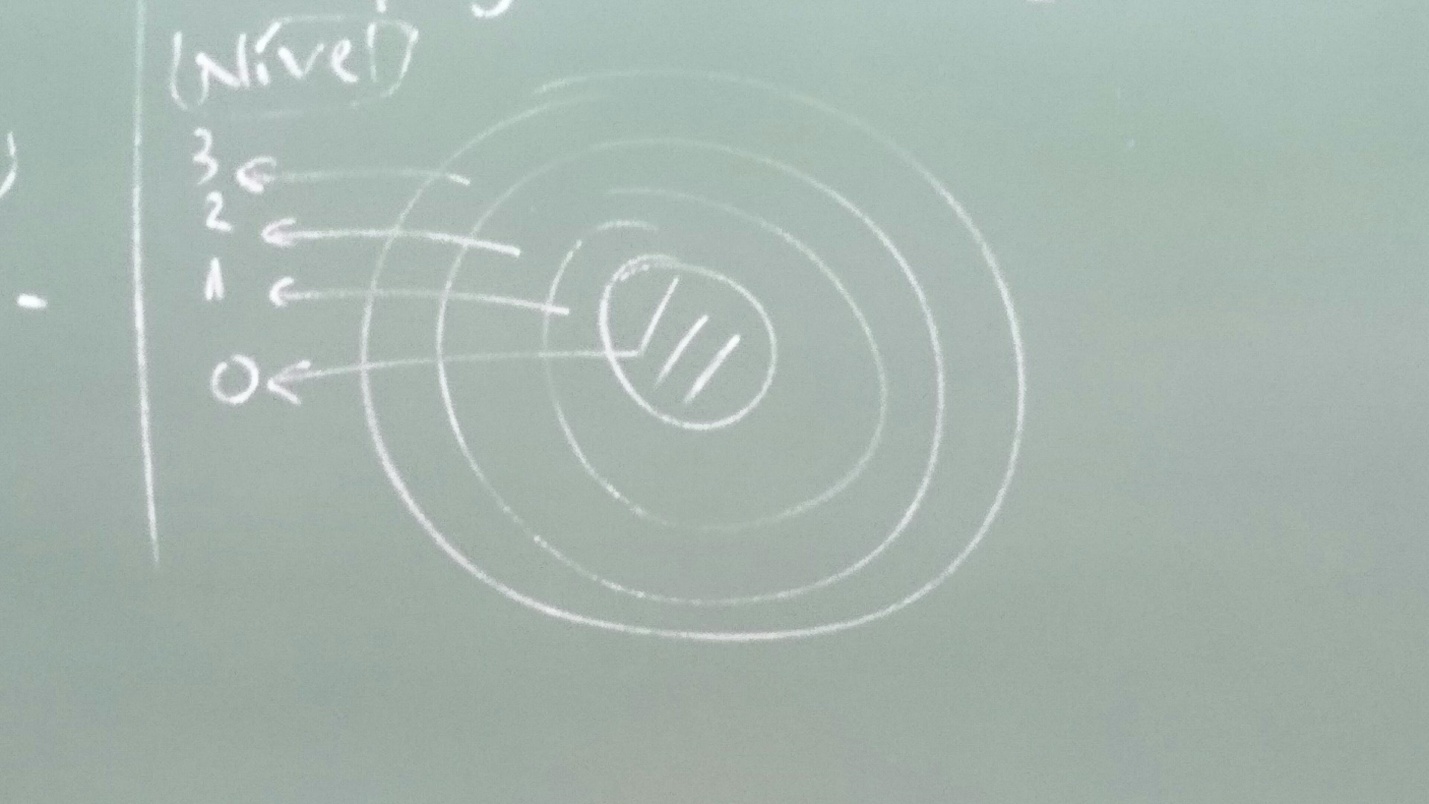
**Instruções não privilegiadas**

São instruções (de CPU) que não desestabilizam o SO.

Ex: Instruções lógicas aritméticas

Como vimos na aula anterior o modo de acesso refere-se a maneira como a CPU disponibiliza suas instruções

Os modos de acesso do processador podem ser representados segundo a figura a seguir:



Nesse modelo, notamos que o nível 0 é aquela que a CPU libera todas as suas instruções.

O nível 3 é onde existe o maior grau de bloqueios das instruções, sendo que os níveis de 1 e 2 são estados intermediários em termos de bloqueio das instruções do processador.

Por simplicidade, consideraremos dois modos de acesso referentes as instruções do processador:

* **Modo Kernel (modo núcleo)**
  + Condição em que o processador libera o uso de todas as instruções (libera as instruções privilegiadas e não privilegiadas
* **Modo usuário**
  + Neste modo a CPU bloqueia as instruções privilegiadas.
  + PSW (registrador de status) apresentam um conjunto de bits que indicam o modo de acesso da CPU

**Chamadas de sistema**

Chamada de sistema é a maneira como a APLICAÇÃO solicita serviço às rotinas do NÚCLEO da SO.

Cada tipo de SO apresenta uma denominação específica às chamadas de sistema.

Por exemplo:

* Unix/Linux: system call
* Windows: API (aplication Program Interface).

Alguns exemplos de chamadas de sistema:

Exemplo de chamada de Windows:

**GetSystemTime**: obtem data/hora

**System.TimeToDateTime**: converte a data hora para um determinado formato

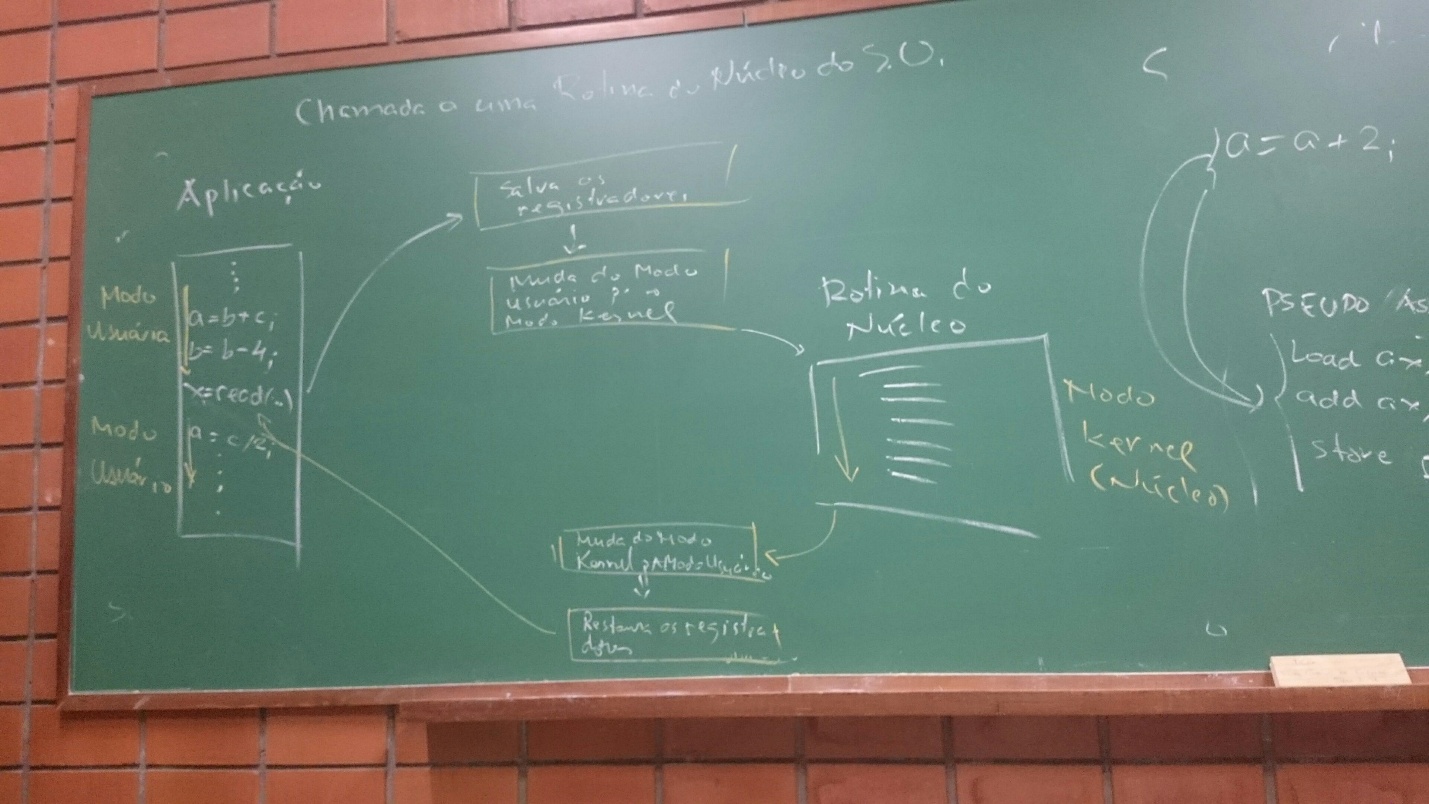
Exemplo de chamada de Unix/Linux

|  |  |
| --- | --- |
| Processo | fork (cria uma cópia do processo) |
| Memória | malloc/free |
| Arquivo | open,close,read,write |
| Dispositivos | mount,umount |

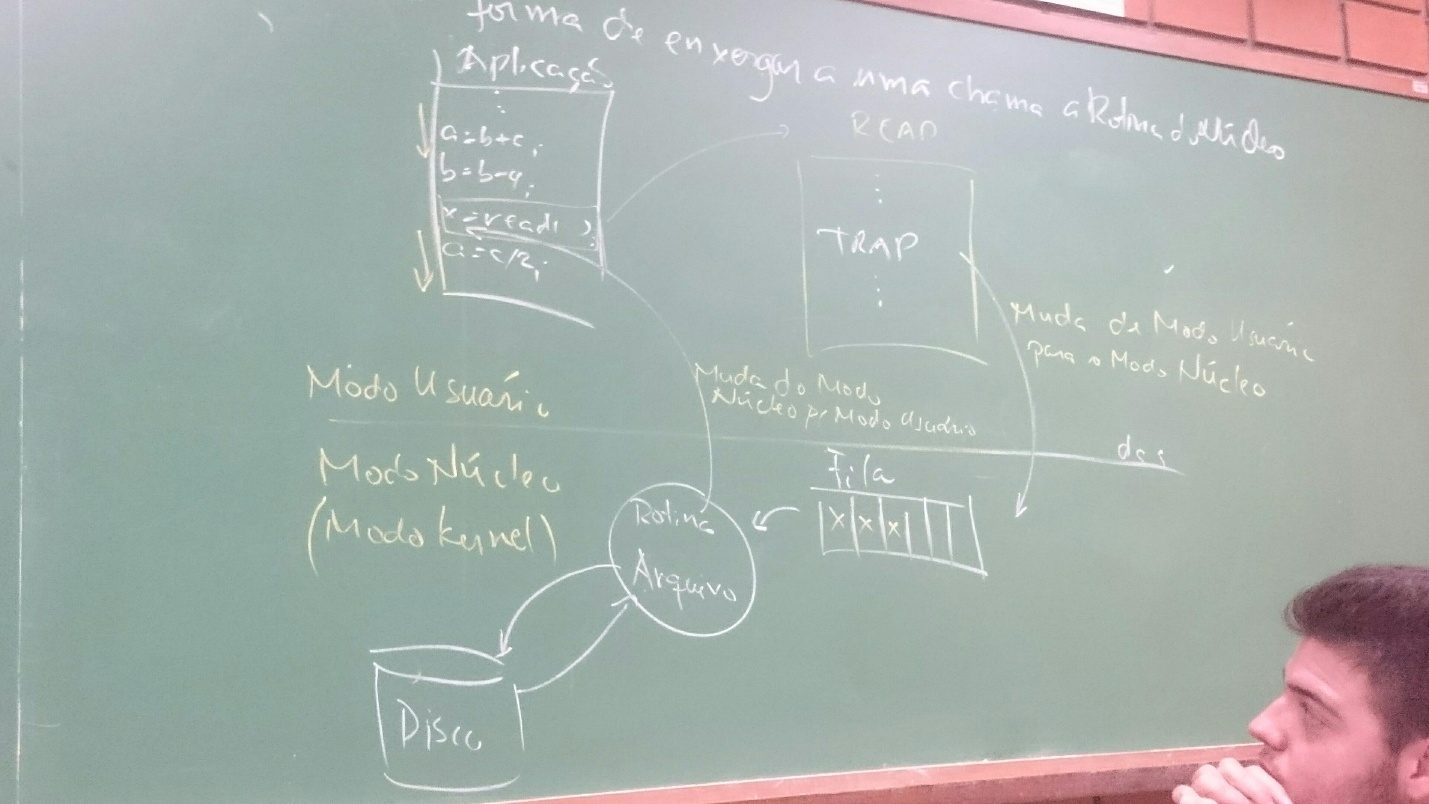
Podemos notar que as chamadas de sistema lidam com:

|  |  |
| --- | --- |
| Processo/threads | cria, elimina, pausa, mudar de prioridade |
| Memória | alocação/desalocar, escrever, ler |
| Arquivos | abrir, fechar, escrever, ler, permissões |
| Dispositivo | alocar/desalocar, escrever, ler |

**Chamada a uma rotina do núcleo do sistema operacional**



**Outra forma de enxergar uma chamada a rotina do núcleo**



## POSIX (portable operate system for unix)

Devido as diferentes distribuições de S.O. Unix, começou a ocorrer diferentes implementações que dependemdo tipo de distribuição UNIX.

Para minimizar a questão de PORTABILIDADE de código (programa) foi desenvolvida uma biblioteca cujo o objetivo era padronizar as interfaces das chamadas de sistemas entre as diferentes distribuições UNIX.

Essa ideia foi ampliada para permitir a PORTABILIDADE de código para outros S.O (Windows)

**Linguagem de comandos**

* Unix/Linux
  + ShellScript
    - Csh
    - Sh
    - Bash
    - Ksh
* Windows
  + Bat

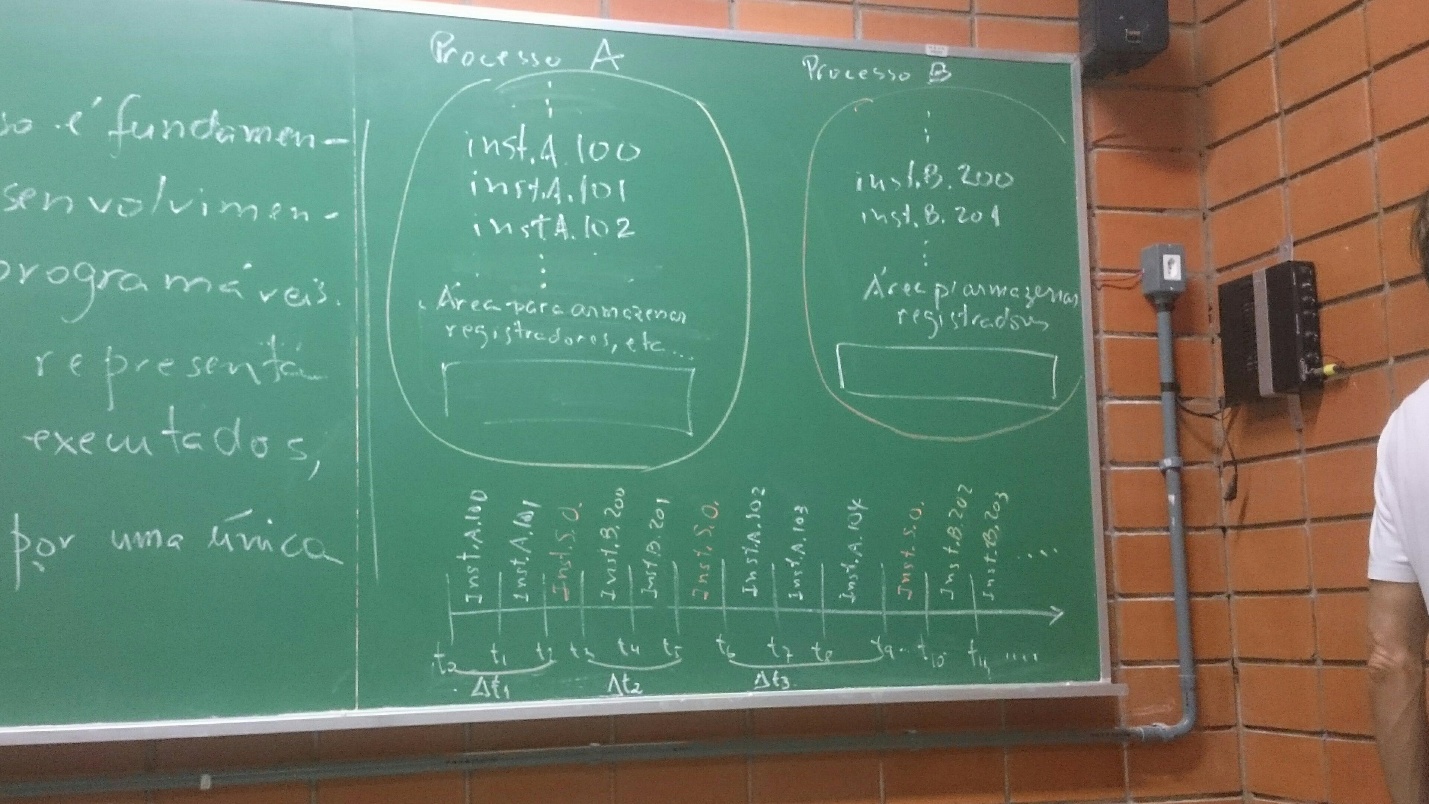
(Interface gráfica)

# Processos

De maneira simplificada, um processo é um programa em execução, porém este capítulo irá explorar algumas características referentes por processos.

O conceito de processo é fundamental para permitir o desenvolvimento de sistemas multi-programáveis.

A figura a seguir representa dois processos sendo executados de forma concorrente, por uma única CPU



A figura anterior mostra os processos de B sendo escalonados pelo SO.

Note que é a fatia-de-tempo destinada a execução das instruções (instrução A.100, instrução A.101) referentes ao Processo A, a seguir, vem à execução da instrução do SO (instrução SO) que realizada a **mudança** **de contexto**.

A instrução (ou instruções do SOP) realiza, entre outras coisas, o armazenamento dos registradores da CPU na área (de memória) do processo A.

Em seguida, essas mesmas instruções do SO obtêm da área do Processo B a cópia dos registradores que lá se encontram.

Feito isso, as instruções do Processo B (instrução B.200, instrução B.201) são executadas no instante de tempo (fatia-de-tempo/quantum), logo que acaba a fatia-de-tempo do Processo B, o SO intervém e grava dados do Processador A, dessa maneira, realiza o ESCALONAMENTO dos processos.

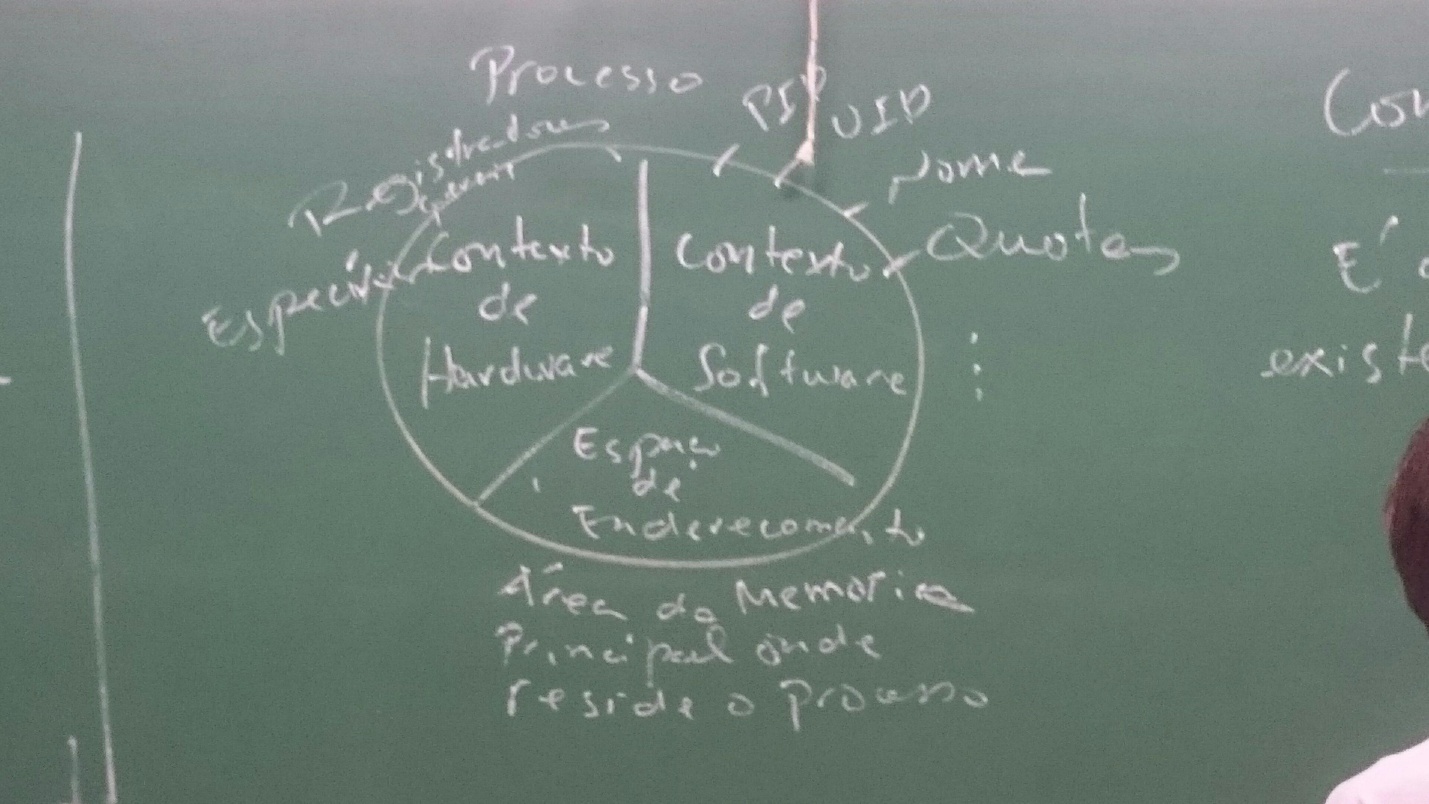
**Obs**: A CPU não “compreende” o conceito de processos. Ela somente executa a instrução apontada pelo registrador (IP/PC) o SO é responsável pela “visão” do conceito de processo.

I/O Bound = recruta a CPU em excesso.

## Estrutura do Processo

Podemos considerar que um processo apresenta:

* Contexto de Hardware
* Contexto de Software
* Espaço de Endereçamento



### Contexto de Hardware

É a área do processo onde existem as cópias dos registradores da CPU.

O contexto de Hardware é importante para ser possível realizar entre outras coisas, a Mudança de contexto que é relacionado ao escalonamento do processo.

**Mudança de contexto:**

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

### Contexto de Software

É a parte do processo que apresenta:

* **PID**: Process Identification, que é o número do processo
* **UID**: User Identification, que é o número do usuário que se loga e execura o processo
* Nome do processo
* Cotas: memória, nº processos filho, nº arquivos abertos, disco, etc
* Permissões

### Espaço de endereçamento

E a área de memória onde o processo fica residente (instruções, estruturas, e dados)

# Estados dos Processos

Em SO moderno nos muitos processos concorrerem pelos recursos computacionais nem todos processos podem fazer uso do processador e por essa razão ficam alocados em filas aguardando o momento de fazer uso desse precioso componente que é o processador.

Para poder melhor compreender o conjunto de processos que alocam dinamicamente os recursos computacionais, iremos analisar o diagrama de estados apresentado a seguir.

## Diagrama de Estados dos Processos



## Processos do Tipo

### CPU - Bound

Esse tipo de processo passa maior parte do tempo fazendo uso da CPU, então em geral são processos que fazem muitos cálculos.



Passa a maior parte do tempo no estado Pronto ou Execução

### I/O - Bound

Esse tipo de processo passa a maior parte do tempo fazendo operação de I/O



### Foreground

Tem canal de comunica diratamente ligados as disponibilidade I/O interativo(teclado)

### Background

O processo não tem comunicação direta com canais de recursos I/O interativo (não tem com tecladas)

# Processos e Threads

Os primeiros SOs, apenas apresentavam o conceito de processo. (Instanciar uma estrutura de dados PCB)

|  |
| --- |
| Ponteiros |
| Nome |
| Estado |
| Registradores |
| Arquivos |

Um problema verificado com o conceito de processo é que muitas vezes havia instanciação de 1 programa idêntico que gerava dois processos

## Programa, Memória



A figura anterior mostra que o processo 1 e o processo 2 apresentaram as mesmas instruções Esses processos escalonam cada um deles apresentam uma única linha de execução (thread) com sua pilha.

Esse modelo, chamada de ambiente monothread, apresenta redundância de instruções, ocupava mais recursos (memória) gerava tempo relativamente elevado na criação, na remoção, na troca de informações etc.

Por essa razão, os pesquisadores passaram a propor novas alternativas e assim surgiu o modelo Multithreads mostrado na figura abaixo:



No modelo Multithread existem certas vantagens quando comparado ao ambiente Monothread.

→ existe uma melhor utilização dos recursos (menor desperdício de memória)

→ o segundo thread em diante leva menos tempo para ser criado eliminado.

→ a comunicação entre threads é mais rápida (compartilham área de memória, compartilham variáveis, etc.

Porém devemos ter o cuidado quanto ao compartilhamento de um mesmo recurso por diferentes threads, exemplo:



A Threads1 e Threads2 ao compartilhar com uma mesma variável global, podem gerar resultados inconsistentes (race conditions).

## Threads versus Processos

Como nós vimos ao ser criado um novo processo, instancia-se uma estrutura (PCB. Isso significa que há um Contexto de Hardware, Contexto de Software e Espaço de Endereçamento.

Para Unix-Like, um processo pode ser representado de forma simplificada conforme a figura abaixo:

|  |  |
| --- | --- |
| Pilha | Variáveis locais |
| Heap | Região utilizada para alocar/desalocar estrutura dinâmica (malloc, new) |
| Dados | Alocação de variáveis globais (atributos) |
| Texto | Instruções |

A seguir um exemplo de programa escrito em código C que gera um novo processo.

### Exemplo novo Processo

#include<stdio.h>

#include <unistd.h>

int valor **=** 0**;**

void func1**(){**

int i**=** 0**;**

**for** **(**i **=** 0**;** i**<=**3 **;** i**++)**

printf**(**"\nValor= %d\n"**,++**valor**);**

**}**

int main**(){**

int pid **=** 0**;**

printf**(**"\nInicio \n"**);**

pid **=** fork**();**

**if(**pid**!=** 0**){**

printf**(**"\nprocesso pai \n"**);**

func1**();**

**}else{**

printf**(**"\nprocesso filho\n"**);**

func1**();**

**}**

printf**(**"\n fim \n"**);**

**return** 0**;**

**}**

O programa acima pode girar a seguinte saída.

1. Início
2. Processo Filho
3. Valor = 1
4. Processo Pai
5. Valor = 1
6. Valor = 2
7. Valor = 2
8. Valor = 3
9. Valor = 3
10. Fim
11. Fim

Note que a chama de sistema pid=fork();

Novo processo onde para o processo pai, for != 0 e para o filho pid = 0

Houve a criação estruturas independentes, cada qual com uma variável global valor.

A seguir, um exemplo com uso de threads

### Exemplo novo Threads

#include <stdio.h>

#include <pthread.h>

int valor **=** 0**;**

void func1**(){**

int i **=** 0**;**

**for** **(**i **=** 1 **;** i**<=**3 **;** i**++)**

printf**(**"\nValor = %d"**,++**valor**);**

**}**

int main**(){**

int pid**=**0**;**

pthread\_t t1**,**t2**;**

printf**(**"\ninicio\n"**);**

pthread\_create**(&**t1**,NULL,** func1**,NULL);**

pthread\_create**(&**t2**,NULL,** func1**,NULL);**

func1**();**

printf**(**"\nfim\n"**);**

pthread\_join**(**t1**,NULL);**

pthread\_join**(**t2**,NULL);**

**return** 0**;**

**}**

(parametro de compilação) gcc prog1.c -lpthread

Uma possível saída do programa anterior seria a seguinte

1. Inicio
2. Valor = 1
3. Valor = 2
4. Valor = 2
5. Valor = 3
6. Valor = 4
7. Valor = 5
8. Fim

(detalhes em na linha 3 e 4 da saída)

Do programa anterior, podemos representa-lo conforme a figura abaixo.



Observe que não houve a criação de dois processos. Os threads t1, e t2 concorrem pela mesmo variável global valor.

Note que a mesma variável foi incrementada por ambas as threads

→ importante: porque pode ter ocorrido nas duas saídas?

valor = 2

valor = 2

Pode ter ocorrido que um thread (1) incrementou valor e antes de imprimir na tela escalonou o segundo thread (2) incrementa o valor e imprime na tela 2. em seguida a thread 1 imprime o valor 2 novamente.

Esse tipo de problema ocorreu porque um mesmo recurso que é a variável valor foi utilizado de maneira concorrente por dois threads esse tipo de situação é conhecido como condição de corrida (race condition).

Concorrência e mecanismos de comunicação e sincronização

Os sistemas multiprogramaveis apresentam concorrência entre processos e threads

Nesses ambientes, a CPU escalona os processos e os threads e, além disso, variáveis memórias, arquivos e demais recursos podem ser acessados de maneira concorrente.

Já vimos, em aulas anteriores, exemplo de programas que geram concorrência de processos e de threads.

Em ambientes com mais que um processador, podemos ter programas paralelos.

O paralelismo pode ser uma solução muito interessante para aumentar o desempenho computacional.

A seguir vermos um programa que realiza a soma de dois vetores em um terceiro utilizando uma solução paralela.

#include <stdio.h>

#include <pthread.h>

#define tam 100000

int vetor1**[**tam**],** vetor2**[**tam**],** vetor3**[**tam**];**

void preencher**(){**

int i **=** 0**;**

**for** **(**i **=** 0**;** i **<** tam **;** i**++){**

vetor1**[**i**]** **=** i**;**

vetor2**[**i**]** **=** **(** i **+** 10 **);**

**}**

**}**

void **\***soma1**(){**

int i **=** 0**;**

**for** **(**i **=** 0**;** i **<** **(**tam**/**2**);** i**++){**

vetor3**[**i**]** **=** vetor1**[**i**]** **+** vetor2**[**i**];**

**}**

**}**

void **\***soma2**(){**

int i **=** 0**;**

**for** **(**i **=** **(**tam**/**2**)** **;** i **<** tam **;** i**++){**

vetor3**[**i**]** **=** vetor1**[**i**]** **+** vetor2**[**i**];**

**}**

**}**

int main **(** void **){**

pthread\_t t1**,** t2**;**

preencher**();**

pthread\_create**(&**t1**,NULL,**soma1**,NULL);**

pthread\_create**(&**t2**,NULL,**soma2**,NULL);**

pthread\_join**(**t1**,NULL);**

pthread\_join**(**t2**,NULL);**

int i**=** 0**;**

**for** **(**i **=** 0 **;** i**<** tam **;** i**++){**

printf**(**"%d\n"**,** vetor3**[**i**]);**

**}**

**return** 0 **;**

**}**

O programa anterior realiza a soma vetorial de dois vetores em um terceiro. Para obter vantagem em termos de desempenho, foram criados dois threads, soma1, e soma2. Sendo que soma1 realiza a soma da primeira metade dos vetores, e soma2 realiza a soma da segunda metade.

Os threads soma1 e soma2 poderiam ser alcançados em núcleos diferentes e realizar processamento paralelo. Dessa maneira teríamos um melhor desempenho computacional. (Aumento da Speedup).

## Problemas de concorrência

Processamento concorrente pode economizar tempo e recursos computacionais, porém devemos estar atentos à possíveis problemas que ocorrem em situações que um recurso está sendo atualizado por vários processos ou threads concorrentes.

O programa a seguir apresneta um problema em decorrência do mal uso de threads concorrentes.

#include <stdio.h>

#include <pthread.h>

#include <unistd.h>

int saldo **=** 1000**;**

void **\***soma**(){**

int i**;**

**for** **(** i **=** 0 **;** i **<=** 10 **;** i**++)**

saldo **=** saldo **+**1**;**

**}**

void **\***subtrai**(){**

int i**;**

**for** **(** i **=** 0 **;** i **<=** 10 **;** i**++)**

saldo **=** saldo **-**1**;**

**}**

int main **(** void **){**

pthread\_t t1**,** t2**;**

printf**(**"\nsaldo inicial = %d\n"**,**saldo**);**

pthread\_create**(&**t1**,NULL,**soma**,NULL);**

pthread\_create**(&**t2**,NULL,**subtrai**,NULL);**

//pthread\_join(t1,NULL);

//pthread\_join(t2,NULL);

printf **(**"\nSaldo Final = %d\n"**,**saldo**);**

**return** 0 **;**

**}**

Se considerarmos uma situação em que as operações fossem uma situação em que as operações fossem sequenciais, para o programa anterior poderíamos ter as operações abaixo:

Saldo inicial = 1000

1010

1000

Saldo final = 1000

Numa solução com threads concorrentes, poderíamos ter uma outra saída

Saldo inicial = 1000

1003

998

999

Saldo final = 999

# Sincronização e comunicação entre processos

Como vimos em sistemas multiprogramações, os processos/threads são executados de maneira concorrente.

Por essa razão há a necessidade que venham a comunicar-se entre si.

Os mecanismos de comunicação e sincronização são de grande importância para sistemas concorrentes.

Apenas para ilustrar vamos supor que a expressão abaixo venha a ser executada:

Se considerarmos um processamento sequencial, poderíamos representar assim:

Caso a fizéssemos a mesma operação de maneira paralelamente teriamos.

4t = soma

Podemos notar que, neste exemplo didático o tempo sequencial foi de 6t e o paralelo foi de 4t considerando

Temos:

Isso significa que o processamento paralelo é 1.5 vezes mais rápido do que o sequencial.

Nem sempre é possível paralelizar um programa.

Um problema relacionado a paralelização é quando há uma dependência de dados:

Ex:

Note que existe uma dependência envolvendo a variável “S”

O paralelismo pode ser representado de diversas maneiras abaixo, vemos um exemplo:

Vamos ver um exemplo paralelo em uma representação em (Pseudo) pascal program express.

**var** x**,**t1**,**t2**,**t3**,**real;

**begin**

parbegin

t1**:=**sqrt**(**300**);**

t2**:=**30.2**\***500**.**2**;**

t3**:=**302**/**7;

parend

x**:=**t1**+**t2**+**t3**;**

writeln**(**‘x**=**’ **,** x**);**

**end**

No programa antes, o bloco entre parbegin/parend é concorrente.

Isso significa que não é possível saber qual a ordem de execução entre as expressões

## Problemas do compartilharmento de recursos

exemplo p.104

**PROGRAM** Conta\_Corrente**;**

READ**(**arq\_Contas**,** Reg\_Cliente**);** //le arquivo

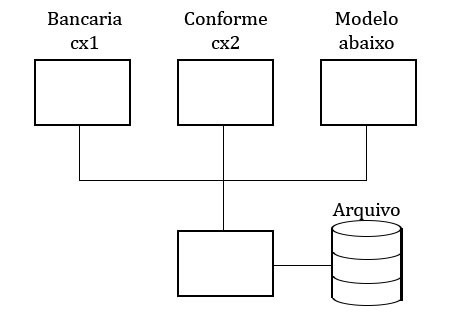
READLN**(**Valor**);**// le valor do teclado

Reg**.**Cliente**.**Saldo **:=** Reg\_Cliente**.**Saldo e Valor**;**

WRITE **(**arq\_Contas**,** Reg\_Cliente**);**

**END.**

Supor que os programa anterior, venha a ser executado em uma rede de uma agência bancária, conforme o modelo abaixo:



Carta de tempo

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| caixa | instrucao | saldo arquivo | valor | saldo |
|  |  | 1000 | \* | - |
| 1 | read | 1000 | 100 | 1000 |
| 1 | readln | 1000 | 100 | 1000 |
| 1 | := | 1000 | 100 | 1100 |
| 2 | read | 1000 | \* | 1000 |
| 2 | readln | 1000 | -300 | 700 |
| 1 | write | 110 |  | 1100 |
| 2 | write | 700 (erro) | -300 | 700 |

## Exclusão Mútua em Sistema de Memória Compartilhada

Para esse tipo de sistema é muito comum em ligação de semáforos e monitores.

Semáforo Binário

É constituído de uma variável especial. Supondo ser esse variável ser ‘s’.

Além disso, possui duas funções que lidam com essa variável.

Down(s)

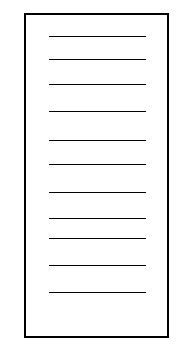
Up(s)

A variável ‘s’ pode apresentar:

S = 0: Não existe recurso(Impede thread/processo de acessar a seção crítica).

S = 1: Recurso Livre (Libera Thread/Processo para acessar a seção crítica).

A figura abaixo mostra a seção crítica com as funções down(s) e up(s).



Lógica envolvida com as funções do semáforo

down(s)

se S = 1 então

S = S - 1

senão

coloca o processo an fila de espera do semáfoto

fim se

fim

up(s)

se existe processo na fila de espera do semáforo então

escolhe um processo para acessar a seção crítica

senão

s = s + 1

fim se

fim

Observação: Down e Up são funções atômicas (invisíveis). Isso significa que não existe escalonamento no Interior dessas funções.

Além dos semáforos binários, podemos encontrar os semáforos contadores, cujo o valor é um inteiro positivo maior ou igual a zero.

Os semáforos contadores podem ser usados para controlar mais do que um recurso compartilhado 3° 1, 2, 3...

Se S = 0 🡪 não existe recurso livre

# Gerenciamento do processador

## Política do escalonamento

Na fila de processos no estado pronto (ready) quem escolhe o processo a ser executado é a rotina do scheduler.

Dispatcher faz a ‘troca do processo’ após o processo deixar a CPU.

A política de escalonamento está relacionada na escolha do processo que fará uso da CPU (Fila de pronto).

## Critérios do escalonamento

Uso da CPU: aumentar o tempo de ocupação da CPU.

* Through put: n° de processos executados por unidade de tempo.
* Tempo do CPU: é o tempo que o processo usa da CPU.
* Tempo de Espera: Tempo total do processo na fila de pronto.
* Tempo Turn Around: Tempo que o processo leva na sua criação até o fim do processamento. (soma do tempo de espera (fila de pronto), soma tempo de CPU soma na fila de espera)
* Tempo de resposta: tempo entre a requisição de um processo até o instante de resposta (última teoria digitada até a exibição de resposta na tela).

Muitas vezes desejam-se situações conflitantes:

Escalonamento não preemptivo.

1. FIFO scheduling
2. Shortest\_job\_First (SJF)

Escalonamento preemptivo.

1. Escalonamento Circular (Round Robin Scheduling)
   1. É um FIFO com “time slice” fatia de tempo
2. Escalonamento por prioridade
   1. Não usa fatia de tempo
   2. Existe valor (prioridade) associado ao “job” (processo). De tempos em tempos o SO verifica se existem alguns processos de maior prioridade, se sim, preempção.

