Sistemas Operacionais

Aula 01 - Conceitos Básicos

Sistema computacional — o que é

Um **sistema computacional** é a combinação de hardware (CPU, memória, dispositivos de E/S, barramentos) e software (sistema operacional, bibliotecas, aplicações) que permite executar programas e oferecer serviços ao usuário. Hardware fornece recursos; software gerencia e usa esses recursos.

Importância do Sistema Operacional (SO)

O SO é a camada que **gerencia recursos** e **fornece abstrações** (processos, arquivos, dispositivos, rede). Sem o SO, programar e coordenar hardware fica muito mais difícil. O SO:

- isola processos (segurança/estabilidade);
- gerencia CPU, memória e I/O (eficiência);
- fornece interface (APIs, sistema de arquivos, shell/GUI).

Sistema sem SO vs com SO

- **Sem SO (bare-metal)**: programas rodam diretamente no hardware. Uso em sistemas embarcados ou firmware. Vantagens: mais controle, menor overhead. Desvantagens: sem multitarefa, sem proteção, difícil reutilização.
- **Com SO**: múltiplos processos, proteção de memória, drivers, abstrações, multitarefa, segurança. Ideal para sistemas gerais (desktop, servidores).

Máquina multinível (modelo em camadas)

Conceito: computador visto como várias camadas/níveis de abstração (hardware → microarquitetura → ISA → SO → runtime/VM → linguagem de alto nível → aplicação). Cada camada esconde detalhes da anterior e oferece serviços para a próxima. Ex.: código Java roda sobre JVM que usa SO que usa hardware.

Definição de SO (resumida)

Programa que atua como **gerente de recursos** e **intermediário** entre hardware e aplicações, oferecendo:

- abstrações (processos, threads, arquivos, sockets),
- mecanismos (escalonamento, alocação de memória, sincronização),
- serviços (I/O, segurança, comunicação).

Vantagens do SO

- Multiprogramação e multitarefa.
- Isolamento e proteção entre programas.
- Gerenciamento de recursos (memória, CPU, disco).
- Interface uniforme para dispositivos (drivers).
- Segurança, autenticação e logging.

Principais funções do SO

- **Gerência de processos:** criação, escalonamento, troca de contexto, sincronização.
- **Gerência de memória:** alocação, paginação, proteção, swap.
- **Sistema de arquivos:** organização, persistência e acesso a dados.
- **Gerência de dispositivos:** drivers e controladoras.
- **Serviços de rede:** pilhas de rede, sockets.
- **Segurança e controle de acesso.**
- **Interfaces usuário/programador:** shell, APIs, syscalls.

Interação com o SO

- Aplicações usam **chamadas de sistema (system calls)** para pedir serviços (I/O, criar processos, alocar memória).
- Há separação **user-space** / **kernel-space** para segurança; dispositivos e recursos sensíveis acessados via syscalls ou drivers.
- Bibliotecas (libc, frameworks) encapsulam syscalls.

Processamento (CPU, interrupções, escalonamento)

- CPU executa instruções; múltiplos processos/threads são multiplexados via **escalonador (scheduler)**.
- **Interrupções** e exceções permitem responder a eventos (timer, I/O).
- **Troca de contexto:** salvar/restaurar registradores e PC ao mudar de processo.
- Cores múltiplos → paralelismo real; escalonador decide onde rodar cada thread.

Memória e influência do cache em C e Java

Cache (L1/L2/L3) é memória rápida entre CPU e RAM; latência e localidade importam muito.

Efeitos gerais:

- **Localidade de referência** (temporal/espacial) melhora hits no cache e performance.
- **Miss de cache** causa grande penalidade (espera de memória).
  Especificamente em **C**:
- Dados contíguos (arrays, estruturas compactas) -> melhor localidade.
- Evitar ponteiros dispersos e encadeamentos longos (pointer chasing).
- Alinhamento e padding controlados pelo programador.
- Otimizações de loop (blocking, loop interchange) para cache-friendly code.

Especificamente em **Java**:

- Objetos no heap são acessados por referências; **array de primitivos** tem excelente localidade.
- **Array de objetos** pode ter má localidade porque cada objeto pode estar espalhado.
- **Garbage Collector** pode mover objetos (melhora localidade quando faz compaction).

ROM e CMOS

- **ROM (Read-Only Memory):** memória não volátil onde firmware (ex.: BIOS) pode residir; geralmente não se altera em operação normal.
- **CMOS:** tradicionalmente refere-se a memória alimentada por bateria que guarda configurações do sistema e relógio (RTC) — no passado baseado em tecnologia CMOS. Hoje as configurações do firmware podem estar em NVRAM/flash, mas o termo CMOS ainda é usado para o setup do BIOS/UEFI.

Boot-Up (sequência)

1. **Power-on** → energia aplicada.
2. **POST (Power-On Self Test):** testes básicos de hardware.
3. Firmware (BIOS/UEFI) inicializa controladoras e configurações.
4. **Bootloader** (p.ex. MBR/UEFI boot manager) carrega o kernel do SO da mídia (disco, SSD).
5. Kernel inicializa subsistemas, drivers, monta root filesystem.
6. Serviços e processos de usuário são iniciados (init/systemd).
   Resultado: sistema operacional ativo e pronto.

Boot-up BIOS (legacy) vs UEFI (moderno)

- **BIOS (legacy):** firmware armazenado em ROM, carrega MBR, tem interface limitada.
- **UEFI:** substitui BIOS, suporta partições GPT, drivers no firmware, boot seguro, interface mais rica e programas no ambiente firmware.

Dispositivo de E/S (I/O device)

- Dispositivo que envia/recebe dados (disco, teclado, rede, GPU, impressora).
- Tipos: **block devices** (discos) e **character devices** (teclado, terminal).
- Comunicação por **portas I/O**, DMA (transferência direta memória ↔ dispositivo) ou acesso mapeado em memória.

Controladoras (controllers)

- **O que são:** hardware que controla um tipo de dispositivo (ex.: controlador SATA para discos, controlador USB para portas USB).
- **Função:** implementam o protocolo físico/ligação com o dispositivo e expõem registros e buffers que o sistema usa para comandar o hardware.
- **Como são comandadas:** o SO (via driver) escreve **comandos especiais** nos registradores da controladora; a controladora converte esses comandos em **sinais elétricos** para acionar o dispositivo.

Interface: Controladora ↔ Driver

- A controladora fornece uma **interface hardware** (registros, portas I/O, linhas de status).
- O **driver** implementa a **camada software** que traduz chamadas do SO em leituras/escritas nesses registradores e trata as interrupções geradas pela controladora.
- Resumindo: *controladora = hardware; driver = software que a controla*.

Driver (definição e responsabilidades)

- **O que é:** software que permite ao SO usar um dispositivo físico específico.
- **Responsabilidades principais:**
  - Inicializar o dispositivo e a controladora.
  - Enviar comandos e configurar registradores.
  - Tratar interrupções do dispositivo.

Drivers diferentes para controladoras e SOs

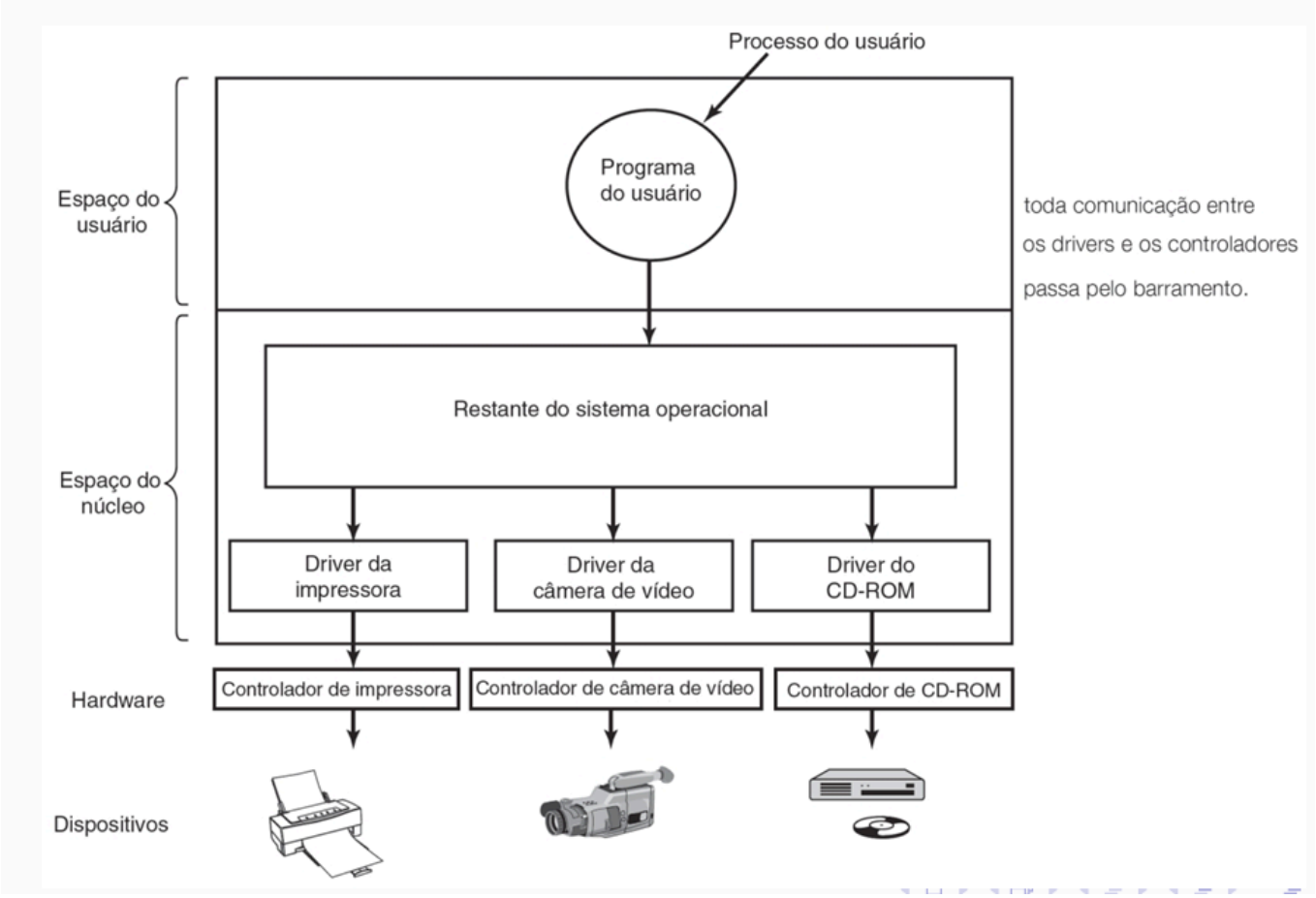
- Cada controladora e cada SO têm interfaces diferentes → **drivers específicos** são necessários.

- Um mesmo dispositivo pode precisar de drivers diferentes em Windows e Linux.

Carregamento dinâmico de drivers

- **Depende do SO e do dispositivo** — muitos sistemas suportam carregar drivers em tempo de execução (sem reiniciar).

- Windows: Adiciona-se uma entrada a um arquivo do sistema informando que ele precisa do driver e então reiniciar o sistema. No momento da inicialização, o sistema busca os drivers de que precisa e os carrega
- Linux: Carrega-se um módulo do kernel, em tempo de execução, contendo o driver



Aula 02 - Tipos e Estruturas de um S.O.

SO monotarefa vs multitarefa

Monotarefa

- **O que é:** executa um único processo/programma por vez na CPU.
- **Funcionamento:** o programa ocupa totalmente a máquina até terminar ou ceder controle (cooperativo).
- **Vantagens:** simplicidade, menor overhead, fácil determinismo.
- **Desvantagens:** pouca interatividade, baixa utilização de I/O e CPU quando há esperas.
  Multitarefa
- **O que é:** vários processos/threads coexistem; CPU é compartilhada entre eles.
- **Como funciona (visão geral):**
  - **Preemptiva:** o SO interrompe processos (timer interrupt) para trocar; melhora responsividade.
  - **Cooperativa:** processo cede voluntariamente o CPU; simples mas arriscado (processo travado trava tudo).
  - **Multiprocessamento:** com vários núcleos, há execução real paralela.
- **Mecanismos importantes:**
  - **Escalonador (scheduler):** decide qual processo/thread roda; políticas: round-robin, prioridade, SJF, O(1)/com múltiplos níveis, real-time (EDF/RMS).
  - **Troca de contexto (context switch):** salvar/restaurar registradores, PC, SP, tabela de páginas possivelmente — tem custo.
  - **Sincronização/IPC:** mutexes, semáforos, filas, pipes, sockets para coordenação entre tarefas.
  - **Threads:** threads leves dentro de um processo compartilham memória; facilitam concorrência.
- **Vantagens:** melhor utilização de CPU/I/O, interatividade, execução concorrente de serviços.
- **Desvantagens:** complexidade, overhead de troca de contexto, problemas de concorrência (race conditions, deadlocks), gerenciamento de memória mais complexo.

Multiprogramação e pseudoparalelismo

- **Multiprogramação:** manter vários programas carregados em memória para que o CPU tenha sempre algo para executar quando um programa esperar por I/O — objetivo: aumentar utilização da CPU.
- **Pseudoparalelismo:** numa CPU única, a ilusão de execução simultânea é produzida por rápidas trocas de contexto entre processos (time-slicing). Em múltiplos núcleos, paralelismo real substitui pseudoparalelismo.

Processos CPU-bound vs I/O-bound

- **CPU-bound:** gastam a maior parte do tempo executando instruções; precisam de muito tempo de CPU (ex.: compressão, cálculos numéricos).
- **I/O-bound:** passam maior parte do tempo esperando I/O (ex.: leitura de disco, rede); precisam de breves fatias de CPU.
- **Consequência para escalonamento:** escalonadores eficientes tentam intercalar I/O-bound (curtas rajadas) com CPU-bound para maximizar throughput e responsividade — ex.: round-robin favorece I/O-bound para latência menor.

Tipos de processamento: batch vs interativo (lote e não-batch)

- **Batch (processamento em lote):**
  - Jobs acumulados e executados sem interação humana.
  - Vantagem: alto throughput, agendamento otimizado.
  - Uso: mainframes, processamento noturno, grandes análises.
- **Interativo:**
  - Sistema responde a comandos do usuário em tempo real (desktop, shell).
  - Exige baixa latência e resposta rápida.
- **Spooling:** técnica de enfileirar saída de jobs (print spooler, mail spool) — permite desacoplar produção de dados e consumo (ex.: imprimir enquanto processo continua).

Tipos de SO por uso

- **Interativo:** foco em resposta ao usuário (desktop, mobile).
- **Servidor:** otimizado para throughput, concorrência de conexões.
- **Tempo real (RTOS):**
  - **Hard real-time (crítico):** deadlines rígidos — falha ao perder deadline é inaceitável (controle de freios, aviônica).
  - **Soft real-time (não crítico):** perdas de deadline degradam desempenho mas não causam catástrofe (streaming de vídeo).
- **Embutido (embedded):** SO leve ou nenhum SO, recursos restritos (ex.: microcontroladores).

Modos de acesso ao hardware: usuário vs kernel

- **Modo usuário (user mode):** privilégios restritos; aplicações não podem executar instruções sensíveis nem acessar hardware diretamente.
- **Modo kernel (kernel mode / supervisor):** privilégios totais; o SO executa operações críticas e manipula hardware.
- **Troca entre modos:** feita por system calls, traps, interrupções; garante proteção — aplicações pedem serviços e o kernel executa em modo privilegiado.

Organização interna do SO — principais estruturas

Estrutura monolítica

- **Princípio:** kernel único que contém subsistemas (gerência de processos, memória, sistema de arquivos, drivers) rodando no mesmo espaço de endereço.
- **Vantagens:** desempenho (chamadas internas são simples), fácil comunicação interna.
- **Desvantagens:** grande base de código, difícil manutenção, bugs em drivers/parte do kernel podem derrubar todo o sistema.

Sistemas em camadas (layered)

- **Princípio:** kernel dividido em camadas, cada camada oferece serviços à camada superior e usa a inferior. Ex.: hardware → gerenciamento de memória → processos → aplicações.
- **Vantagens:** modularidade, facilita design/verificação (cada camada isolada).
- **Desvantagens:** overhead de chamadas entre camadas, nem sempre natural dividir em camadas puras.

Microkernel

- **Princípio:** núcleo mínimo no kernel (escalonamento, IPC, gerência básica de memória). Serviços como drivers, sistema de arquivos e servidores de rede rodam em espaço de usuário como processos separados.
- **Vantagens:** maior modularidade, segurança e robustez (falha de um driver user-space não derruba kernel), facilidade de portabilidade.
- **Desvantagens:** overhead de IPC e contexto entre servidores e kernel (podendo afetar desempenho), design mais complexo de comunicação.
- **Exemplos:** MINIX, QNX, microkernel-based designs; macOS/XNU híbrido tem microkernel elements.

Máquina virtual (virtual machine / hypervisor)

- **Princípio:** camada que permite executar múltiplos sistemas operacionais convidados sobre o mesmo hardware, cada um isolado.
- **Tipos:**
  - **Type 1 (bare-metal) hypervisor:** roda diretamente sobre hardware (ex.: Xen, VMware ESXi).
  - **Type 2:** roda sobre um SO hospedeiro (ex.: VirtualBox).
- **Vantagens:** isolamento forte, consolidação, snapshots, portabilidade de ambientes.

- **Desvantagens:** overhead (virtualização de I/O/CPU), complexidade, necessidade de gerenciamento de recursos; sem paravirtualização pode haver perda de desempenho.
- **Observação:** com hardware de virtualização (VT-x, AMD-V) overhead é muito reduzido.

Modelo cliente-servidor (no contexto de SO e serviços)

- **Princípio:** funcionalidades oferecidas por servidores (no mesmo host ou remotos); clientes solicitam serviços via IPC, sockets ou RPC.
- **Vantagens:** modularidade, escalabilidade, possibilidade de distribuir serviços, fácil isolamento (servidores podem rodar em processos separados).
- **Desvantagens:** overhead de comunicação, complexidade de sincronização/recuperação de falhas, latência adicional se for remoto.

Comparações rápidas (prós/cons resumidos)

- **Monolítico**
    - Alto desempenho interno
    - − Pior isolamento; manutenção difícil
- **Microkernel**
    - Robustez, segurança, modularidade
    - − IPC mais caro; possível queda de desempenho
- **Camadas**
    - Arquitetura clara, fácil prova de corretude
    - − Possível overhead e camadas artificiais
- **Máquina virtual**
    - Isolamento, consolidação, flexibilidade
    - − Overhead e complexidade de gestão
- **Cliente-servidor**
    - Distribuição e reutilização de serviços
    - − Comunicação e latência; tolerância a falhas exigente

Aula 03 - Chamadas ao Sistema e Interrupções

Vimos que apenas as aplicações do usuário roda em modo usuário, apenas o SO roda em modo kernel (pelo menos assim deveria ser).

Syscall(chamada ao sistema)

- **Definição**: Mecanismo que permite **aplicações em modo usuário** solicitarem **serviços privilegiados** do sistema operacional (modo kernel).
- **Exemplos Comuns**:
  - Acesso a arquivos (`read` , `write`)
  - Gerenciamento de processos (`fork` , `exec`)
  - Comunicação entre processos (pipes, sockets).

2. Funcionamento

1. **Transição de Modo**:
   - Aplicação executa uma instrução especial (ex: `TRAP` ou `SYSCALL`).
   - **CPU alterna do modo usuário para modo kernel** (aumentando privilégios).
2. **Processamento pelo SO**:
   - Kernel identifica a chamada (via número ou tabela de syscalls).
   - Executa a operação privilegiada (ex: ler disco, alocar memória).
3. **Retorno**:
   - Resultado é repassado à aplicação.
   - CPU volta ao modo usuário.

Variação entre SOs

- **Números/APIs**: Diferem entre Linux, Windows, macOS.
  - Ex: Linux usa `sys_read` ; Windows usa `ReadFile` .
- **Conceitos**: Semelhantes (modo usuário/kernel, tabela de syscalls).

As chamadas ao sistema são implementadas através de instruções especiais do processador conhecidas como TRAP (ou SYSCALL/SYSENTER em arquiteturas mais modernas). Estas instruções possuem características únicas que as tornam fundamentais para a segurança e funcionamento dos sistemas operacionais:

1. Natureza da Instrução TRAP

- É uma *interrupção programada* gerada por software (diferente de interrupções de hardware)
- Atua como um "porteiro eletrônico" que controla o acesso ao modo kernel
- Cada arquitetura possui sua própria implementação:
  - x86: INT 0x80 (antigo), SYSENTER (moderno)
  - ARM: SVC (Supervisor Call)
  - RISC-V: ECALL

2. Fluxo de Execução

1. **Preparação**:
   - A aplicação carrega parâmetros em registradores específicos
   - O número da syscall é armazenado (ex: EAX em x86)
2. **Execução da TRAP**:
   - O processador:
     a) Alterna para modo kernel (bit de privilégio muda)
     b) Salva o PC (Program Counter) atual
     c) Redireciona execução para o vetor de interrupções
3. **Processamento no Kernel**:
   - O SO consulta uma tabela de syscalls (ID → função)
   - Valida parâmetros e permissões
   - Executa a operação privilegiada
4. **Retorno** (IRET/SYSEXIT):
   - Restaura o contexto do usuário
   - Retoma execução na instrução pós-TRAP
   3. Casos Especiais

- **Tratamento e Exceções**: Algumas TRAPs sinalizam erros como:
  - Divisão por zero (exceção aritmética)
  - Page fault (acesso a memória inválida)
  - Overflow
- **Syscalls Bloqueantes**: Operações como read() podem:
  - Suspender o processo até que dados estejam disponíveis
  - Permitir que outro processo use a CPU durante a espera
  *_Considerações de Segurança_*
- O kernel **nunca** confia nos parâmetros do usuário:
  - Verifica todos os ponteiros (copiando dados para kernelspace)
  - Valida permissões (ex: acesso a arquivos)
- O espaço de endereçamento é totalmente separado:
  - Código do usuário não pode acessar memória do kernel
  - Registradores sensíveis são protegidos
  `read()` é uma **chamada de sistema fundamental** usada para ler dados de uma fonte (arquivo, dispositivo, socket, etc.) e armazená-los em um buffer na memória. Seus parâmetros são:
- `fd` (file descriptor): Identificador do recurso (ex: 0 = teclado, 1 = tela, 3+ = arquivos)
- `buffer`: Endereço de memória onde os dados serão salvos
- `nbytes`: Número máximo de bytes a serem lidos

Passo a Passo da Execução

Fase 1: Preparação no Modo Usuário

1. **Empilhamento de Parâmetros (Passos 1-3)**
   - O programa empilha os argumentos na ordem inversa (convenção *cdecl*):
2. **Chamada à Biblioteca (Passo 4)**
   - Invoca a função `read()` da biblioteca padrão (ex: `libc` em Linux)

Fase 2: Transição para o Kernel

3. **Registro da Syscall (Passo 5)**
   - A biblioteca armazena o **número da syscall** em um registrador específico:
     - Ex: Linux x86 → `eax = 3` (código de `read()`)
4. **Instrução TRAP (Passo 6)**
   - Executa `int 0x80` (x86) ou `syscall` (x86_64)
   - **Efeitos imediatos**:
     - CPU alterna para **modo kernel** (bit de privilégio elevado)
     - Salta para o **vetor de interrupções** do SO

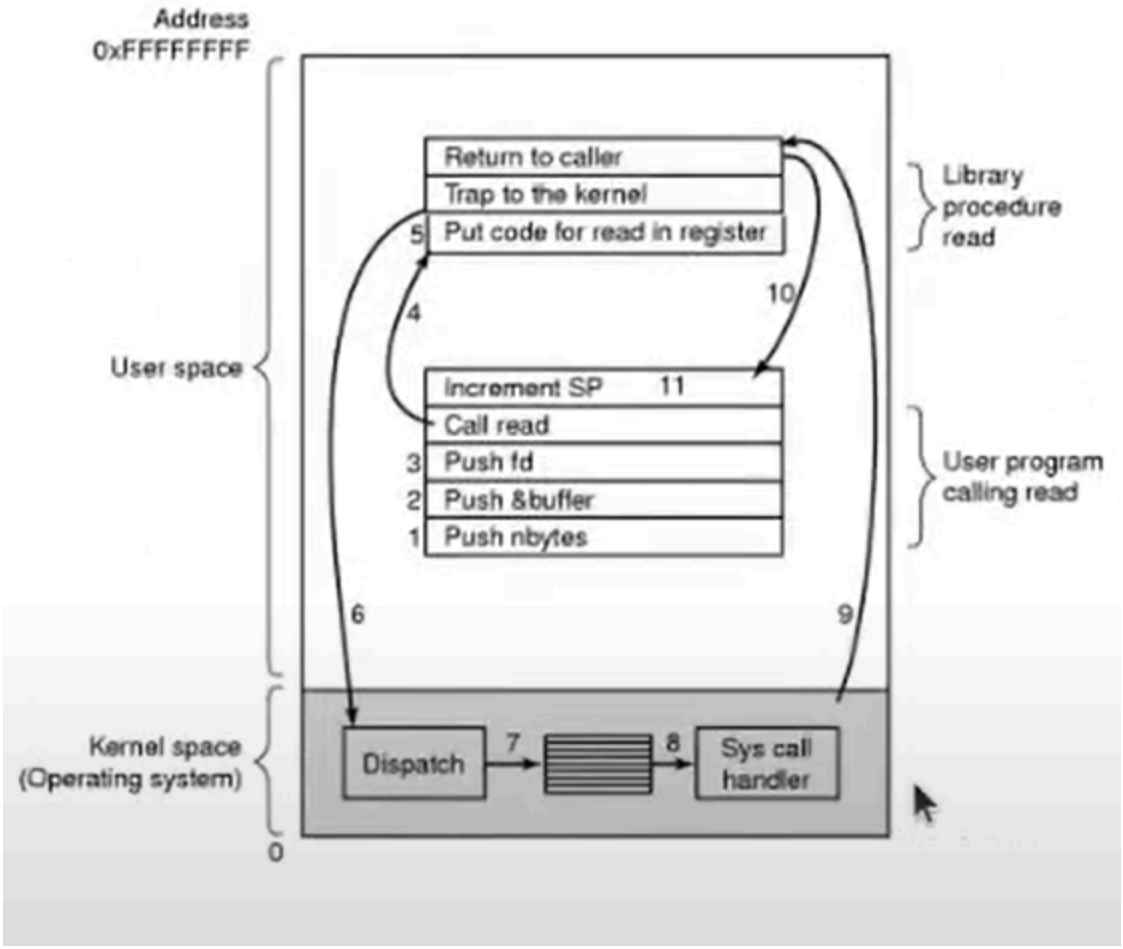
Fase 3: Processamento no Kernel

5. **Despacho (Passo 7)**
   - O kernel consulta a **tabela de syscalls** usando o número em `eax`
   - Localiza o endereço da rotina `sys_read()`
6. **Validação e Execução (Passo 8)**
   - Verifica:
     - Se `fd` é válido
     - Se `buffer` está em memória acessível
     - Permissões do processo
   - **Operações comuns**:
     - Se dados estão em cache → copia para `buffer`
     - Se não → bloqueia processo até dados estarem disponíveis

Fase 4: Retorno ao Usuário

7. **Restauração (Passo 9)**
   - Kernel coloca em `eax`:
     - Número de bytes lidos (sucesso)
     - Código de erro (falha, ex: `-EBADF`)
   - Executa `iret` / `sysret` para voltar ao modo usuário

8. **Retorno da Biblioteca (Passo 10)**
    - A função `read()` da biblioteca:
        - Interpreta o resultado em `eax`
        - Define variável global `errno` se houve erro
9. **Limpeza da Pilha (Passo 11)** : Ajuste do stack pointer (SP):



- write() e exit() através da interrupção 0x80
  - Instrução em assembly – trap – no Linux para x86

```
section    .data                    ;declaração da seção
    msg    db    "Ola, mundo!",0xa  ;declara localização (declare byte): nosso string
    len    equ   $ - msg            ;define constante (equ): tamanho do nosso string
section    .text                    ;declaração de seção
    global _start                   ;ponto de entrada para o linker (ld)

_start:              ;diz ao linker o ponto de entrada
    ;escreve o string na stdout
    mov    edx,len    ;terceiro argumento: tamanho da mensagem
    mov    ecx,msg    ;segundo argumento: ponteiro para a mensagem a ser escrita
    mov    ebx,1      ;primeiro argumento: descritor de arquivo (stdout)
    mov    eax,4      ;número da chamada ao sistema (sys_write)
    int    0x80       ;chama o kernel

    ;e sai
    mov    ebx,0    ;primeiro argumento da chamada ao sistema: código de saída
    mov    eax,1    ;número da chamada ao sistema (sys_exit)
    int    0x80     ;chama o kernel
```

Esquecer de chamar o *exit()* pode ocasionar em um dos dois cenários:

- Uma instrução ilegal é rodada, ai o programa é forçasadamente parado.
- Uma instrução legal porém indesejável é rodada, ai o comportamento será imprevisível. Por conta dessa situação nada agradável, o compilador automaticamente no final do código chama exit(), caso o programa já não tenha chamado.

O que são Wrappers?

Wrappers (ou "invólucros") são **funções de biblioteca** que encapsulam chamadas ao sistema, fornecendo uma interface mais amigável e portável para os programadores. Eles abstraem a complexidade de fazer chamadas diretas ao kernel, padronizando o acesso a recursos do sistema operacional.

Funções Principais dos Wrappers

1. **Tradução de Parâmetros**
   - Convertem estruturas de dados de alto nível (ex.: `FILE*` em C) para formatos que o kernel entende (ex.: *file descriptors*).
   - Exemplo:
     // Wrapper fopen() em C
     FILE *f = fopen("arquivo.txt", "r"); // Usa open() internamente
2. **Padronização entre Sistemas**

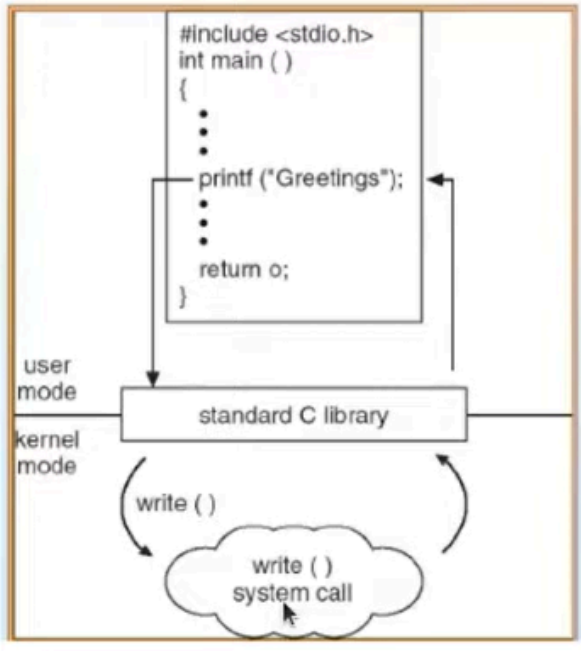
- Em Windows, APIs como `ReadFile()` são wrappers que se comunicam com o kernel NT (via `NtReadFile`).
- Em Linux, a `glibc` fornece wrappers como `read()` para a syscall `sys_read`.

3. **Tratamento de Erros**
   - Convertem códigos de erro do kernel (ex.: `-ENOENT`) em valores padronizados (ex.: `errno` em C).
4. **Segurança Adicional**
   - Validam parâmetros antes de passar para o modo kernel.
   - Exemplo: Verificam se ponteiros de buffer são válidos

Vantagens dos Wrappers

1. **Portabilidade**
   - Código funciona em diferentes SOs sem modificação.
   - Ex: `fopen()` funciona igual em Linux e Windows.
2. **Segurança** : Previnem uso incorreto de syscalls (ex.: passar ponteiros inválidos).
3. **Produtividade**
   - Funções mais simples que syscalls diretas.
   - Ex: `printf()` vs `write()`.

- Programa em C que invoca a função de biblioteca printf(), que por sua vez chama o system call write()
  - write() e exit() através da instrução *int 0x80*



Interrupções

Interrupções são **sinais que pausam a execução corrente** da CPU para lidar com eventos prioritários. Podem ser:

- **Software**: Geradas pelo programa (ex.: chamadas ao sistema via `TRAP`).
- **Hardware**: Geradas por dispositivos externos (ex.: teclado, relógio, disco).

2. Como o Escalonador Interrompe Processos?

- **Problema**: O escalonador não está sempre em execução.
- **Solução**:
  - **Interrupções de clock**: Disparam periodicamente (ex.: a cada 10ms).
  - **Rotina de Tratamento**: O SO registra um tratador que:
    1. Salva o estado do processo atual (registradores, PC).
    2. Invoca o escalonador para decidir o próximo processo.

3. Tratamento de Interrupções

Passos do Hardware/SO:

1. **Detecção**: CPU identifica o sinal de interrupção.
2. **Mudança de Contexto**:
   - Salva registradores na pilha do kernel.
   - Alterna para modo kernel.
3. **Despacho**:
   - Consulta o **Vetor de Interrupções** (tabela de endereços).
   - Salta para a **Rotina de Serviço** correspondente.
4. **Retorno**: Restaura o contexto e volta à execução normal.
   Exemplo de Vetor (x86):

Endereço	Rotina Associada
0x00	Divisão por zero
0x20	Interrupção de timer
0x80	Chamadas ao sistema

1. Arranjo de Interrupções (Vetor de Interrupções)

O **Arranjo de Interrupções** é uma estrutura crítica do sistema operacional, localizada em uma região específica da memória (geralmente no início, como `0x0000` ou `0x8000`). Ele mapeia **cada tipo de interrupção** para a **rotina de tratamento** correspondente.

Características Principais:

- **Organização**:
  - Cada entrada no arranjo contém um **endereço de memória** (ponteiro) para uma rotina de serviço.
  - Exemplo em x86 (IDT - Interrupt Descriptor Table):
  - Entrada 0x00: Divisão por zero → handler_divide_error
    Entrada 0x20: Timer → handler_timer
    Entrada 0x80: Syscall → handler_syscall
- **Associação a Dispositivos**:
  - **Cada classe de dispositivo** (teclado, disco, clock) tem uma entrada dedicada.
  - Exemplo:
    - Disco rígido → Interrupção 0x13 (`handler_disk`)
    - Teclado → Interrupção 0x21 (`handler_keyboard`)
- **Integração com o BCP**:
  - Quando uma interrupção ocorre, o **Bloco de Controle de Processo (BCP)** salva o estado do processo atual (registradores, PC) antes de tratar a interrupção.
  - Após o tratamento, o SO usa o BCP para restaurar o contexto ou escalonar outro processo.

Interrupções x Traps

Ambas interrompem a execução normal, mas têm origens e propósitos distintos:

Característica	Interrupções (Hardware)	Traps (Software)
Origem	Dispositivos externos (teclado, disco, timer).	Geradas pela CPU (instruções do processo).
Causa	Sinal elétrico (ex.: timer, E/S concluída).	Chamada ao SO (`TRAP`) ou erro (divisão por zero).
Sincronismo	Assíncrona (pode ocorrer a qualquer momento).	Síncrona (ocorre durante a execução do processo).
Relacionamento	Pode não ter relação com o processo atual.	Sempre associada ao processo corrente.
Exemplos	Timer, teclado, disco.	`read()` , `fork()` , acesso inválido à memória.

Tratamento de Interrupções: Onde e Como Salvar o Contexto?

Quando uma interrupção ocorre, o hardware e o sistema operacional precisam **salvar o estado atual** da execução para que possam retomar o processo posteriormente sem corromper seus dados. Esse é um dos desafios mais críticos no design de sistemas operacionais.

1. O Que Precisa Ser Salvo?

O hardware salva **automaticamente** pelo menos:

- **Contador de Programa (PC)**: Para saber onde retomar a execução.
- **Registradores de Status**: Flags da CPU (ex.: zero, carry).
  Em casos mais complexos (troca de processo), o **SO salva no BCP**:
- Todos os registradores gerais (EAX, EBX, etc.)
- Ponteiro de pilha (ESP)
- Estado da FPU (se usado)

2. Locais Possíveis para Salvar o Contexto

Opção 1: Registradores Internos da CPU

- **Vantagem**: Acesso ultrarrápido.
- **Problema**:
  - Espaço extremamente limitado.
  - Bloqueia a CPU até o tratamento terminar (não pode confirmar a interrupção ao dispositivo).
  - Exemplo: Se uma interrupção de disco ocorrer durante outra, os dados podem ser sobrescritos.

Opção 2: Pilha do Processo Usuário

- **Vantagem**: Natural para chamadas de função.
- **Problemas**:
  - **Ponteiro inválido**: Se a pilha estiver em uma página não mapeada (page fault durante a interrupção → crash).
  - **Conflito com escalonamento**: Outro processo pode ser ativado antes do fim do tratamento.

Opção 3: Pilha do Kernel

- **Vantagem**:
  - Sem riscos de page fault (memória sempre acessível).

    - Isolada do espaço do usuário.
- **Problema**:
    - **Overhead**: Troca de modo usuário→kernel exige atualização de MMU/TLB (lento).
    - **Arquitetura-dependente**: Em MIPS, usa-se um endereço fixo (ex.: `0x80000000`).

Opção 4: Área Especial de Memória (Ex: MIPS)

- Algumas arquiteturas dedicam um registrador para armazenar um **endereço fixo** onde o contexto é salvo.
    - Exemplo: MIPS usa `k0` e `k1` (registradores reservados ao kernel).

Ciclo Básico de Interrupção

Durante **cada ciclo de instrução**, a CPU verifica a existência de interrupções pendentes. O fluxo é:

1. **Verificação**
    - Ao final da execução de uma instrução, a CPU checa a **linha de interrupção** (sinal elétrico).
    - Se não houver interrupção: busca a próxima instrução.
2. **Tratamento** (se houver interrupção)z
    - **Suspensão**: Pausa o programa atual.
    - **Salvamento**: Armazena no mínimo:
        - Contador de Programa (PC) → Para retomar depois.
        - Registradores de status (flags).
    - **Redirecionamento**: PC aponta para a **ISR** (Interrupt Service Routine).
    - **Execução**: Roda a rotina de tratamento.
    - **Retorno**: Restaura o contexto e continua o programa interrompido.

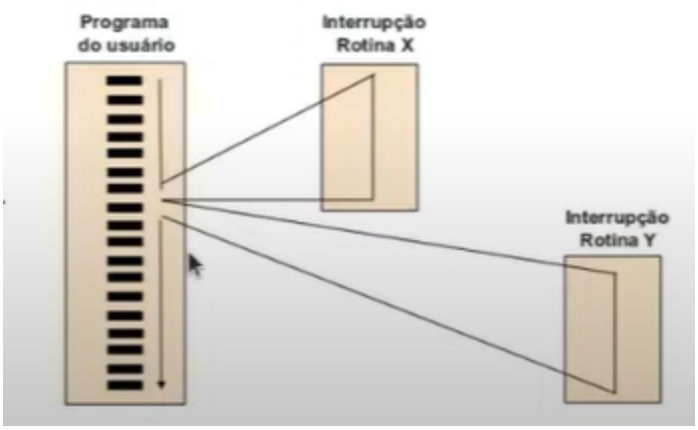
Modelos para Múltiplas Interrupções

Quando várias interrupções ocorrem simultaneamente, o SO usa estratégias diferentes:

Modelo Sequencial (Simples)

- **Funcionamento**:
    - Desabilita **todas** as interrupções durante o tratamento.
    - Só verifica novas interrupções após terminar a atual.
- **Vantagem**: Simplicidade (sem risco de conflito).

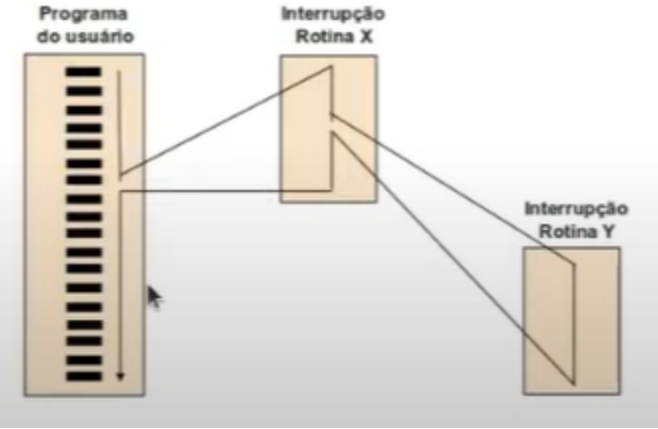
- **Desvantagem**:
  - **Perda de dados**: Se um dispositivo rápido (ex.: rede) enviar múltiplas interrupções, algumas podem ser ignoradas.
  - **Latência**: Dispositivos ficam esperando.



Modelo em Cascata (Prioridades)

- **Funcionamento**:
  - Cada interrupção tem uma **prioridade fixa** (ex.: rede > disco > impressora).
  - Interrupções de **alta prioridade** podem interromper rotinas de baixa prioridade.
- **Vantagens**:
  - **Responsividade**: Dispositivos críticos são atendidos primeiro.
  - **Eficiência**: Evita perda de dados em dispositivos rápidos.
- **Desvantagem**:
  - Complexidade (gerenciamento de pilhas de contexto aninhadas).
- **Exemplo de Prioridades**:

Prioridade	Dispositivo
Alta	Rede
Média	Disco
Baixa	Impressora



Aula 04 - Processos e Escalonamento

Processo é a coisa central no SO. Ele tem o contexto dos programas, mantendo o registro de tudo que tá acontecendo na máquina.

- **Processo**: É a unidade central de execução em um sistema operacional, representando um programa em execução. Inclui:
  - Código do programa
  - Dados de entrada/saída
  - Estado atual (executando, bloqueado, pronto)
  - Contexto de execução
- **Diferença entre Processo e Programa**:
  - Programa: Algoritmo codificado, visão estática da tarefa
  - Processo: Instância única em execução, visão dinâmica pelo SO

Tipos de Processos

Processos em Primeiro Plano (Foreground)

- Interagem diretamente com o usuário
- Exemplos:
  - Leitura de arquivos
  - Iniciação de programas via interface
- Características:
  - Entrada/saída pelo terminal
  - Requer atenção do usuário

Processos em Segundo Plano (Background/Daemons)

- Operam independentemente da interação do usuário

- Exemplos:
  - Serviços de email
  - Serviços de impressão
- Características:
  - Entrada/saída por arquivos
  - Executam funções específicas do sistema
- Um mesmo programa pode ter múltiplas instâncias como processos distintos

Componentes Fundamentais de um Processo

1. **Programa**:
   - Contém as instruções que serão executadas pelo processo
2. **Espaço de Endereçamento**:
   - Área de memória exclusiva do processo
   - Intervalo de endereços que o processo pode acessar (de 0 até um máximo específico)
   - Protegido contra acesso por outros processos
3. **Contextos**:
   - **Hardware**:
     - Registradores do processador (PC, ponteiro de pilha, registradores gerais)
     - Estado atual da execução
   - **Software**:
     - Variáveis do programa
     - Lista de arquivos abertos
     - Alarmes pendentes
     - Relacionamentos com outros processos

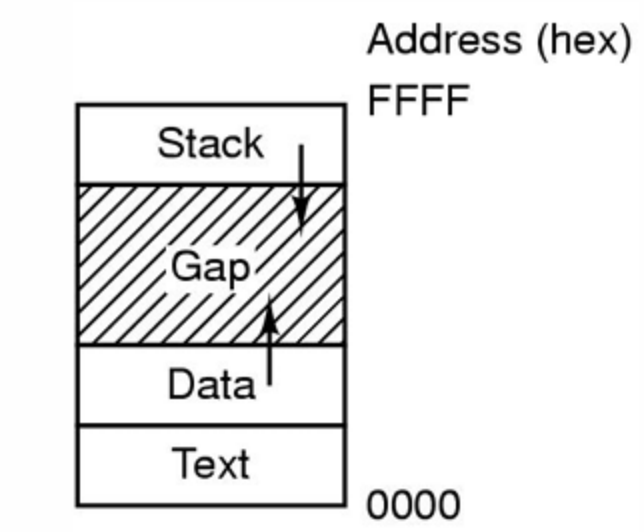
Estrutura do Espaço de Endereçamento

O espaço de memória de um processo é organizado em três segmentos principais:

1. **Segmento de Texto**:
   - Contém o código executável do programa
   - Parte somente-leitura (não modificável durante execução)
2. **Segmento de Dados**:
   - Armazena as variáveis do programa
   - Área de memória modificável
3. **Pilha de Execução**:
   - Gerencia o fluxo de execução
   - Armazena:

- Chamadas de procedimentos/funções
- Parâmetros de funções
- Variáveis locais
- Endereços de retorno

Gap: Permite que a pilha se expanda dinamicamente.



Pilha de execução (Stack)

- Contém um *stack frame* para cada rotina chamada que ainda não terminou.
- Cada *stack frame* guarda:
  - **Variáveis locais** da rotina.
  - **Endereço de retorno** (para saber onde continuar a execução após a função terminar).
  - **Parâmetros** passados para a rotina.
- **Exemplo**: Se a função `DrawSquare` chama `DrawLine`, a pilha terá:
  1. Parâmetros e variáveis locais de `DrawSquare`.
  2. Endereço de retorno para `DrawSquare`.
  3. Parâmetros e variáveis locais de `DrawLine`.
  4. Endereço de retorno para `DrawLine`.

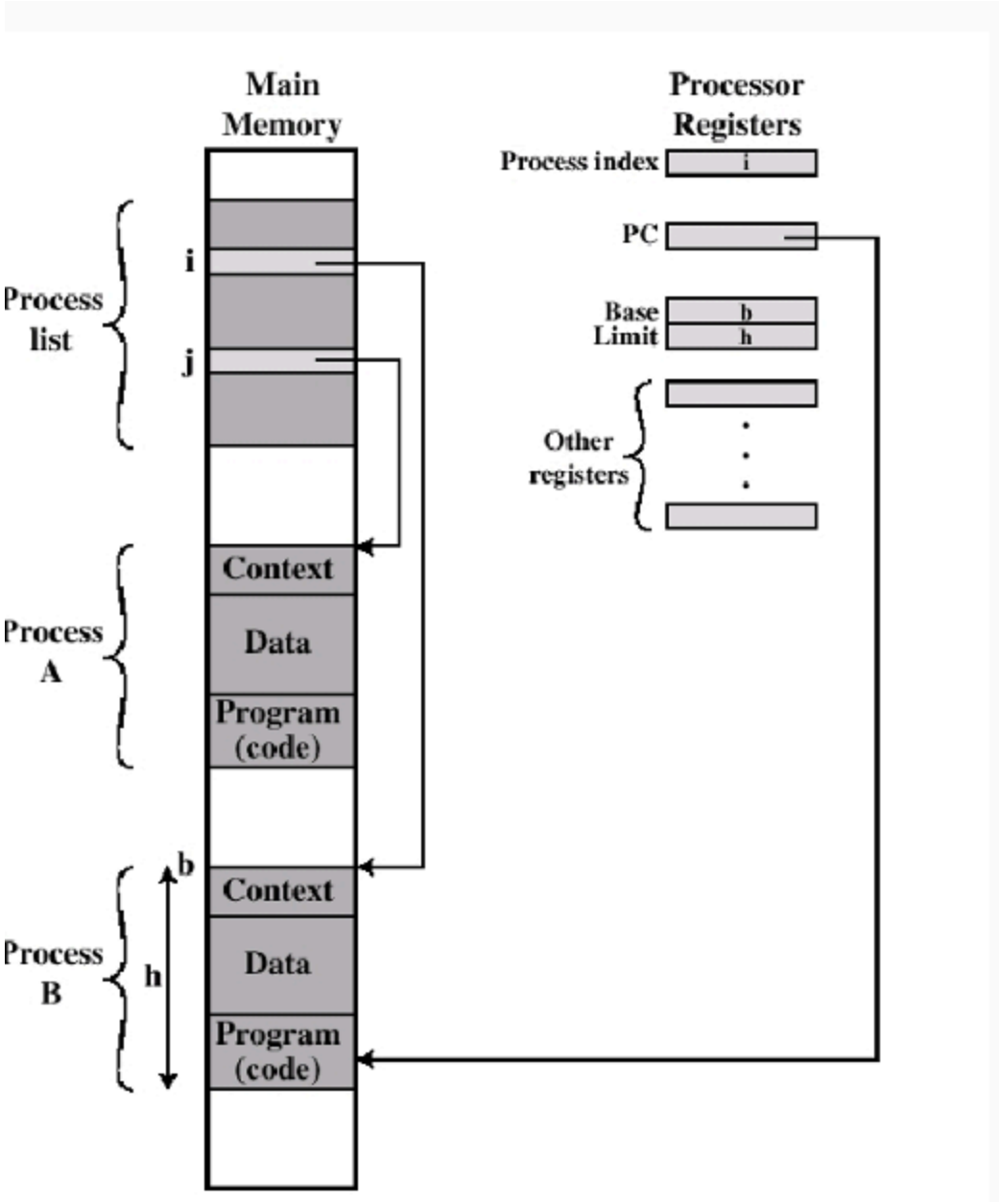
Sub-rotinas e a pilha

- Ao **chamar** (`Call`) uma sub-rotina:
  - O endereço da próxima instrução é salvo na pilha.
  - O fluxo de execução salta para o código da sub-rotina.
- Ao **retornar** (`Return`):
  - O endereço salvo é retirado (*desempilhado*).
  - O fluxo volta exatamente para onde parou.

- Chamadas podem ser **aninhadas**: sub-rotinas chamam outras sub-rotinas, empilhando sucessivos endereços de retorno.

Contexto de Processo

- O **contexto** é o conjunto de informações que descrevem o estado atual de um processo enquanto ele está rodando.
- Inclui:
  - Registradores de uso geral.
  - **PC** (Program Counter) → aponta para a próxima instrução.
  - **SP** (Stack Pointer) e outros registradores.
- Quando um processo é executado, seu **PC** é carregado nos registradores do processador.
- Na dia ilustração, o processo rodando é aquele cujo **PC** está atualmente nos registradores da CPU. (pelo que eu entendi)



Criação de Processos

- Processos podem ser criados em vários momentos:
  1. Inicialização do sistema.
  2. Chamada de sistema feita por outro processo.
  3. Solicitação do usuário.
  4. Inicialização em *batch* (processamento em lotes).
- Cada sistema operacional tem formas diferentes de criar processos:
  - **UNIX (`fork`)**:
    - Cria um clone exato do processo pai (mesma memória, descritores, conexões).
    - Depois o filho pode usar `execve` para carregar um novo programa na memória.
    - O clone pode manipular recursos do pai antes de executar outro código.

- **Windows (`CreateProcess`)**:
  - Cria diretamente um processo filho já carregando o novo programa.

Funcionamento do `fork`

- `fork()` é chamado pelo pai e retorna **duas vezes**:
  - Uma no pai (retornando o PID (número maluco???, não é zero, por isso sabemos que é o pai) do filho).
  - Uma no filho (retornando 0).
- Após o fork, pai e filho possuem cópias independentes das variáveis e recursos, mas inicialmente idênticas.
- Isso permite execução paralela ou coordenação entre eles.
  Compartilham de arquivos abertos durante esse processo, se trocamos o stdin e stdout podemos enganar o filho e o pai.

Hierarquia de Processos

- **UNIX**:
  - Existe hierarquia: pai cria filho, que pode criar netos.
  - Pode ser visualizada com `pstree` no Linux.
- **Windows**:
  - Não há hierarquia real.
  - Pai recebe um identificador (*handle*) para o filho.
  - O pai pode repassar esse identificador para outros processos, “transferindo” o controle.

Troca de contexto

- Quando a CPU alterna entre processos:
  - Salva o contexto do processo atual na **memória** (lista de processos).
  - Carrega o contexto do próximo processo nos **registradores**.
- Isso garante que cada processo retome exatamente do ponto onde foi interrompido.

Hierarquia e Gerenciamento de Processos em Unix/Linux

Hierarquia de Processos

1. **Estrutura em Árvore**:
   - Todos os processos descendem do processo `init` (PID 1)
   - `init` cria processos filhos para gerenciar terminais e serviços do sistema

2. **Grupos de Processos**:
    - Processos relacionados formam grupos (processo pai + descendentes)
    - Sinais do teclado são enviados para todo o grupo associado à janela atual
3. **Relação Pai-Filho**:
    - Processo pai conhece seus filhos (mantém seus PIDs)
    - Processo filho não conhece o pai após criação
    - Comunicação deve ser explicitamente estabelecida pelo pai
    - Pai não pode transferir a "paternidade" do processo para outro

Término de Processos

1. Término Voluntário

- **Normal**:
    - Processo conclui sua execução
    - Chamadas de sistema: `exit()` (Unix), `ExitProcess()` (Windows)
- **Por Erro**:
    - Processo detecta condição inviável (ex: arquivo não encontrado)
    - Encerra com código de erro (ex: compilador com arquivo inexistente)

2. Término Involuntário

- **Erro Fatal**:
    - Causado por bugs no programa
    - Exemplos: divisão por zero, acesso a memória inválida
    - SO interrompe via sinal (ex: SIGSEGV para segmentation fault)
- **Por Outro Processo**:
    - Chamadas: `kill()` (Unix), `TerminateProcess()` (Windows)
    - Processo solicitante precisa ter permissões adequadas
    - Usado para:
        - Gerenciamento de sistema
        - Interrupção de processos congelados
        - Controle de serviços

Fluxo Completo do Ciclo de Vida

1. `init` cria processos base
2. Processos criam filhos (fork + exec)
3. Execução normal ou com erros

4. Término voluntário ou involuntário
5. Pai recebe status do filho (via wait/waitpid)

Estados Básicos de um Processo

1. **Executando (Running)**:
   - Processo está ativamente usando a CPU
   - Em sistemas com múltiplos núcleos, vários processos podem estar neste estado simultaneamente
2. **Pronto (Ready)**:
   - Processo está em memória e apto a executar
   - Aguardando sua vez na escalonamento da CPU
   - Possui todos os recursos necessários, exceto o processador
3. **Bloqueado (Blocked/Waiting)**:
   - Processo não pode continuar até que um evento externo ocorra
   - Exemplos: espera por E/S, liberação de recurso, sinal

O Estado Zumbi

- Processo que terminou sua execução mas ainda mantém uma entrada na tabela de processos
- Ocorre quando:
  - Processo filho termina antes do pai
  - Pai não chamou `wait()` ou `waitpid()` para coletar seu status de saída
  - O SO mantém informações mínimas (PID, status de saída) até o pai as consultar

Problemas Causados:

1. **Esgotamento de PIDs**:
   - Tabela de processos tem tamanho fixo
   - Muitos zumbis podem impedir criação de novos processos
2. **Consumo de Recursos**:
   - Cada zumbi ocupa uma entrada valiosa na tabela
   - Embora não use CPU/memória, limita capacidade do sistema

Soluções e Boas Práticas:

1. **Chamada wait()**:
   - Pai deve sempre chamar wait() para liberar recursos do filho
   - Implementar handlers para SIGCHLD (sinal de término de filho)

2. **Adoção por init**:
    - Se o pai termina sem wait(), init (PID 1) adota o zumbi
    - init periodicamente chama wait() para limpeza automática

Tabela de Processos

- **Função**: Estrutura central que gerencia todos os processos no sistema
- **Organização**:
    - Arranjo ou lista encadeada de ponteiros para Blocos de Controle de Processo (BCP)
    - Cada entrada contém:
        - PID (Process ID) - número único que identifica o processo
        - Ponteiro para o BCP correspondente
        - Estado do processo (executando, pronto, bloqueado, zumbi)

Bloco de Controle de Processo (BCP)

É a estrutura de dados fundamental que o sistema operacional utiliza para gerenciar cada processo individualmente. Ele funciona como um "dossiê completo" do processo.
Além do PID, ele também usará o PPID (Parent Process ID), que nada mais é que o identificador do processo pai.

- **Conteúdo**:
    - Contexto de hardware (registradores, PC, SP)
    - Contexto de software (arquivos abertos, variáveis de ambiente)
    - Informações de gerenciamento (prioridade, tempo de CPU)
    - Ponteiros para espaço de endereçamento
- **Exclusões**:
    - Não armazena o conteúdo real do espaço de memória do processo
    - Mantém apenas referências às estruturas de memória

Implementação de Processos

Process management	Memory management	File management
Registers	Pointer to text segment	Root directory
Program counter	Pointer to data segment	Working directory
Program status word	Pointer to stack segment	File descriptors
Stack pointer		User ID
Process state		Group ID
Priority		
Scheduling parameters		
Process ID		
Parent process		
Process group		
Signals		
Time when process started		
CPU time used		
Children's CPU time		
Time of next alarm		

Algumas informações do BCP

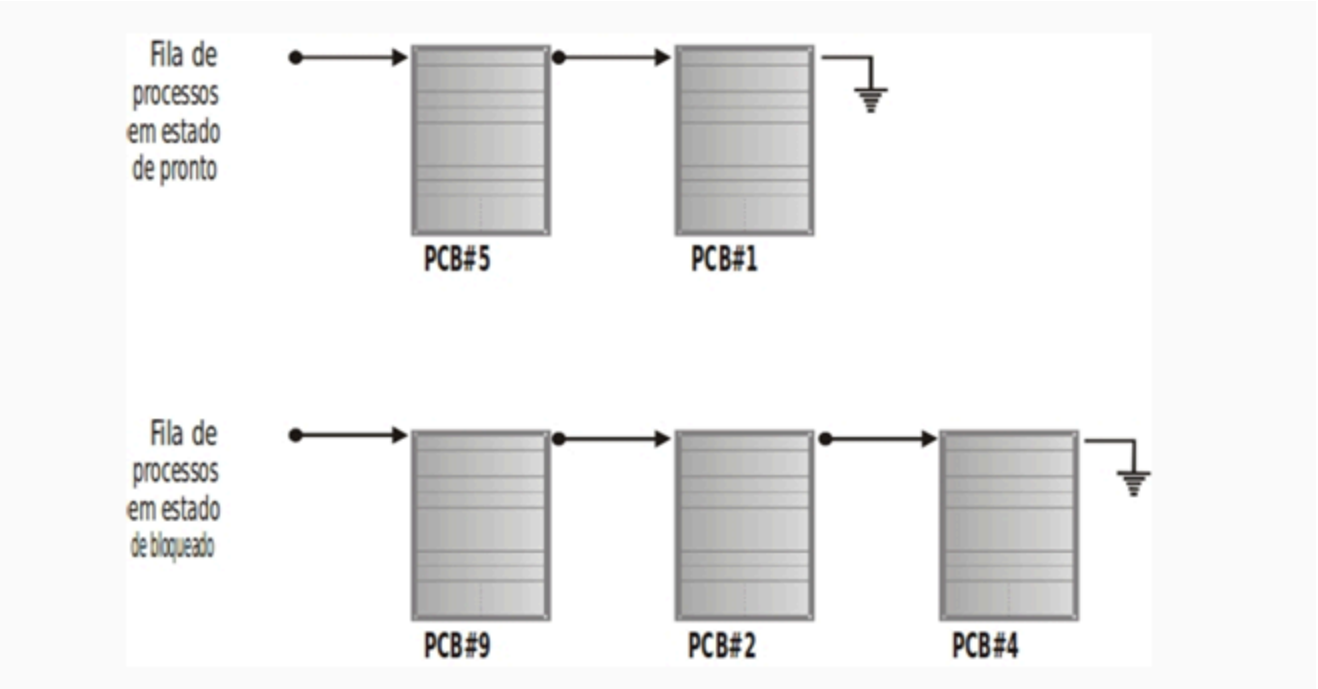
O SO segue o seguinte processo para criar novos processos:

1. **Atribuição de PID**: Identificador único para cada processo
2. **Alocação na Tabela de Processos**: Nova entrada é criada
3. **Alocação de Memória**: Espaço para código, dados e pilha
4. **Inicialização do BCP (Bloco de Controle de Processo)**:
   - Armazena contexto do processo
   - Mantém informações de registradores, estado, etc.
5. **Inserção na Fila Adequada**: Pronto ou bloqueado, conforme a situação inicial

Estados de Processos e Filas

Filas de Gerenciamento:

- **Fila de Prontos**:
  - Processos que estão aptos a executar
  - Exemplo: PCB#5, PCB#1 (esperando vez na CPU)
- **Fila de Bloqueados**:
  - Processos aguardando eventos externos
  - Exemplo: PCB#9, PCB#2, PCB#4 (esperando E/S)



Transições de Estado:

1. **Execução → Bloqueado**:
   - Quando o processo requer E/S ou outro recurso não disponível
   - Ocorre via chamadas de sistema (block/pause) ou automaticamente
2. **Bloqueado → Pronto**:
   - Quando o recurso esperado se torna disponível
   - O SO move o processo para a fila de prontos
3. **Execução → Pronto**:
   - Durante trocas de contexto pelo escalonador
   - Processo é interrompido para dar vez a outro

O Papel do Escalonador

Funções Principais:

1. **Seleção de Processos**:
   - Decide qual processo da fila de prontos ganhará a CPU
   - Utiliza diferentes algoritmos (FIFO, Round Robin, Prioridades, etc.)
2. **Gerenciamento de Transições**:
   - Coordena as mudanças entre estados
   - Trabalha em conjunto com o dispatcher para trocas de contexto
3. **Balanceamento de Carga**:
   - Garante uso justo e eficiente da CPU

- Prevenção de starvation (processos que nunca executam)

Integração com Hardware:

- **Interrupções**: Mecanismo para recuperar o controle da CPU
- **Dispatcher**:
  - Componente de baixo nível que efetiva a troca
  - Salva/restaura contextos durante mudanças