# Aula 03 – Chamadas ao Sistema e Processos

Norton Trevisan Roman
Clodoaldo Aparecido de Moraes Lima

25 de agosto de 2014

# Chamadas ao Sistema

- Modos de Acesso ao Hardware:
  - Modo usuário;
  - Modo kernel ou Supervisor ou Núcleo;
  - São determinados por um conjunto de bits localizados no registrador de status do processador: PSW (program status word);
  - Por meio desse registrador, o hardware verifica se a instrução pode ou não ser executada pela aplicação;
  - Protege o próprio kernel do Sistema Operacional na RAM contra acessos indevidos;

# Chamadas ao Sistema

- Modo usuário:
  - Aplicações não têm acesso direto aos recursos da máquina, ou seja, ao hardware;
  - Quando o processador trabalha no modo usuário, a aplicação só pode executar instruções sem privilégios – um subconjunto das instruções da máquina
  - Por que?
    - Para garantir a segurança e a integridade do sistema;
    - Se o código no modo Usuário tenta fazer algo que não devia, ocorre uma interrupção
    - Em vez do sistema todo cair, somente a aplicação problemática cairá

# Chamadas ao Sistema

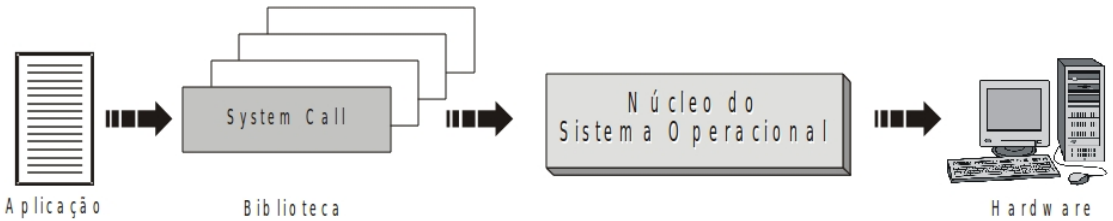
- Modo Kernel:
  - Aplicações têm acesso completo aos recursos da máquina, ou seja, ao hardware;
  - Executa operações com privilégios;
  - Quando o processador trabalha no modo kernel, a aplicação tem acesso ao conjunto total de instruções;
  - Apenas o SO tem acesso às instruções privilegiadas;
  - Problemas ocorridos no modo Kernel podem travar a máquina toda.

# Chamadas ao Sistema

- Se uma aplicação precisa realizar alguma instrução privilegiada, ela realiza uma chamada de sistema, que altera do modo usuário para o modo kernel;
  - Ex: Ler um arquivo
- Chamadas de sistemas são a porta de entrada para o modo Kernel;
  - São a interface entre os programas do usuário no modo usuário e o Sistema Operacional no modo kernel;
  - As chamadas diferem de SO para SO. No entanto, os conceitos relacionados às chamadas são similares independentemente do SO;

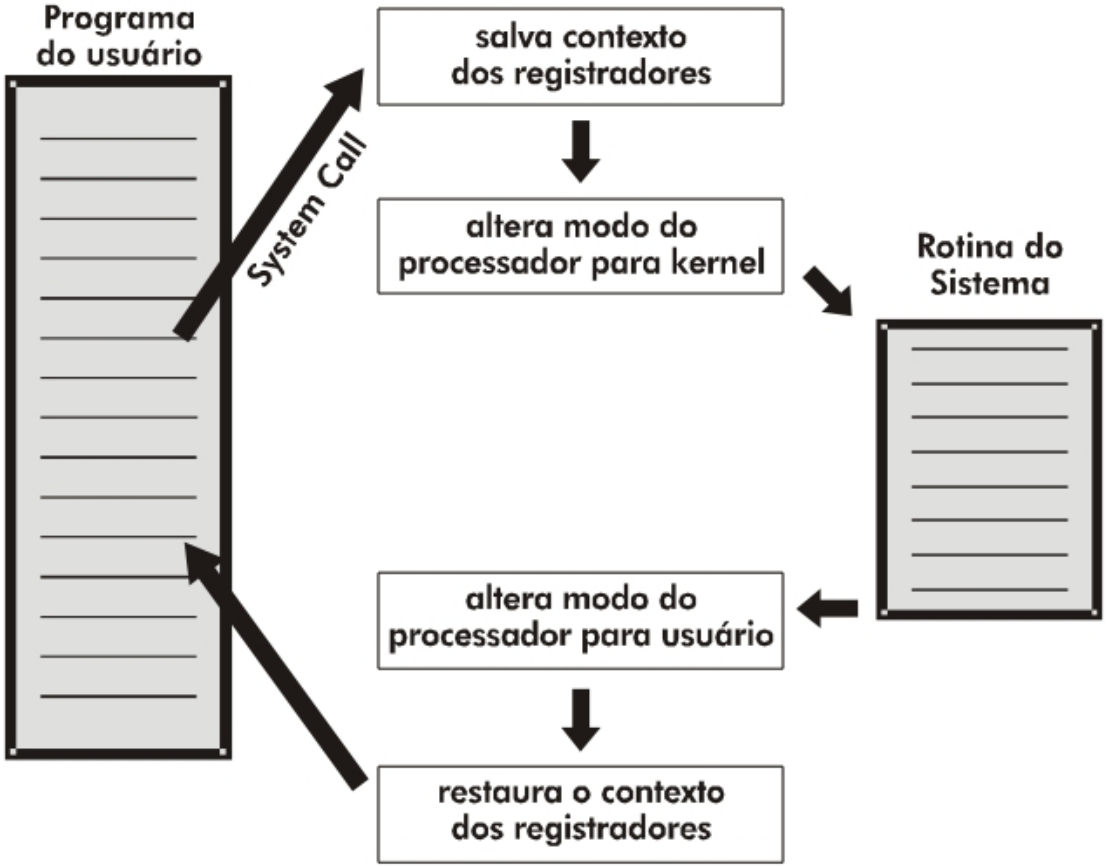
# Chamadas ao Sistema

- São feitas por meio de instruções TRAP
  - Instrução de chamada ao sistema
    - Instrução que permite o acesso ao modo kernel;
    - Normalmente usada para E/S
  - Transferem o controle para o SO → Interrupção de software ou Exceção
    - Podem sinalizar exceções (Divisão por zero, Acesso inválido de memória), Overflows etc
  - Após terminadas, o SO transfere o controle para a instrução seguinte à chamada



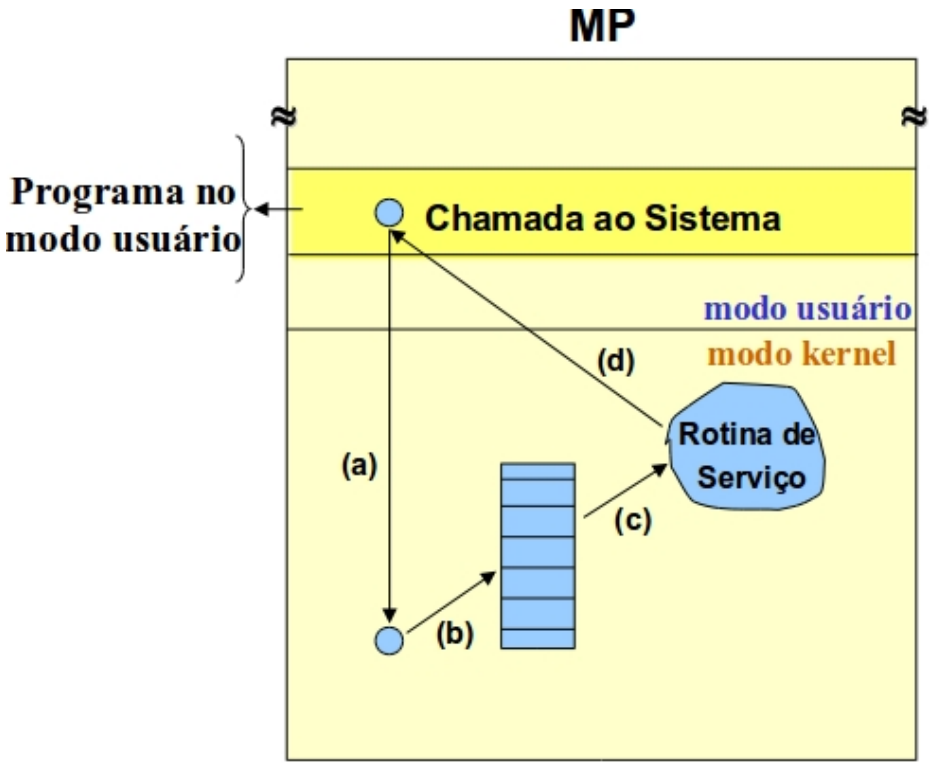
# Chamadas ao Sistema

- Chamada a uma rotina do sistema: TRAP



# Chamadas ao Sistema

- (a) Aplicativo faz chamada ao sistema (TRAP)
- (b) Através de uma tabela, o SO determina o endereço da rotina de serviço
- (c) Rotina de Serviço é acionada
- (d) Serviço solicitado é executado e o controle retorna ao programa aplicativo

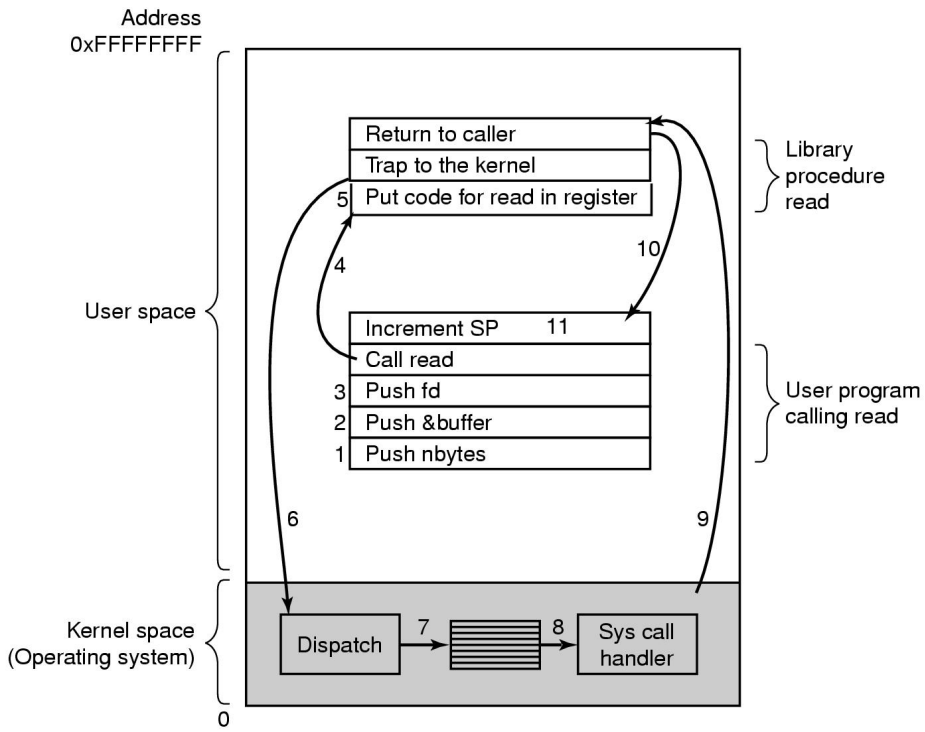


# Chamadas ao Sistema

- A rotina de tratamento salva somente os registradores que irá modificar
  - Antes de usar algum registrador, ela guarda seu valor anterior
  - Ao final, devolve esse valor
  - Com isso, ganha tempo, sem precisar transferir todos à memória
- Onde salva?
  - Depende do hardware
  - Ex: MIPS $\rightarrow$ em uma região da memória, cujo endereço é armazenado em um registrador específico

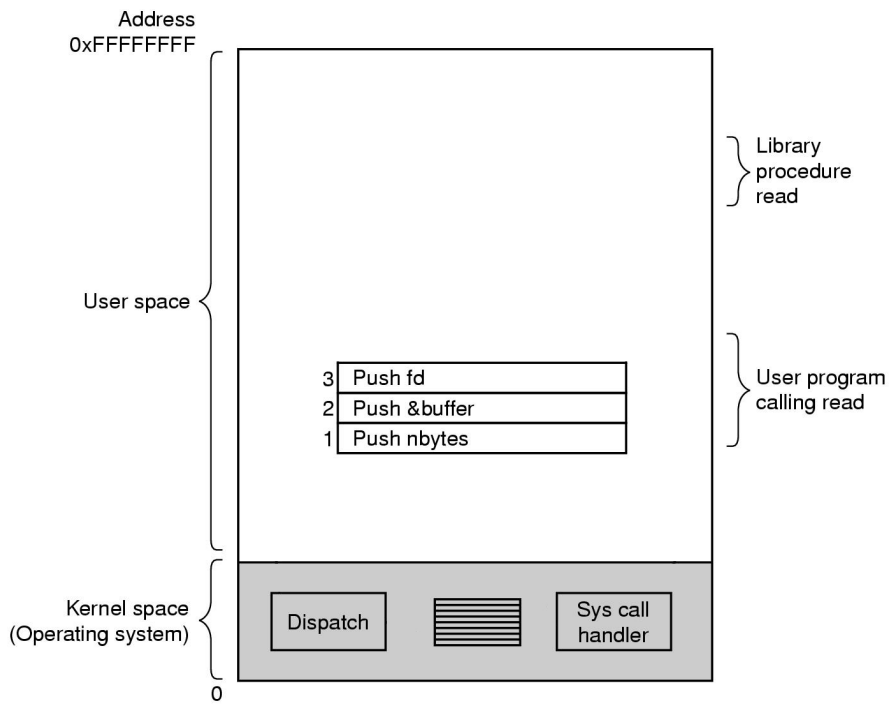
# Chamadas ao Sistema

- read(fd,buffer,nbytes)
  - fd: especificador do arquivo
  - buffer: endereço na memória do primeiro byte da área onde deve ser armazenado o conteúdo lido
  - nbytes: número de bytes a serem lidos



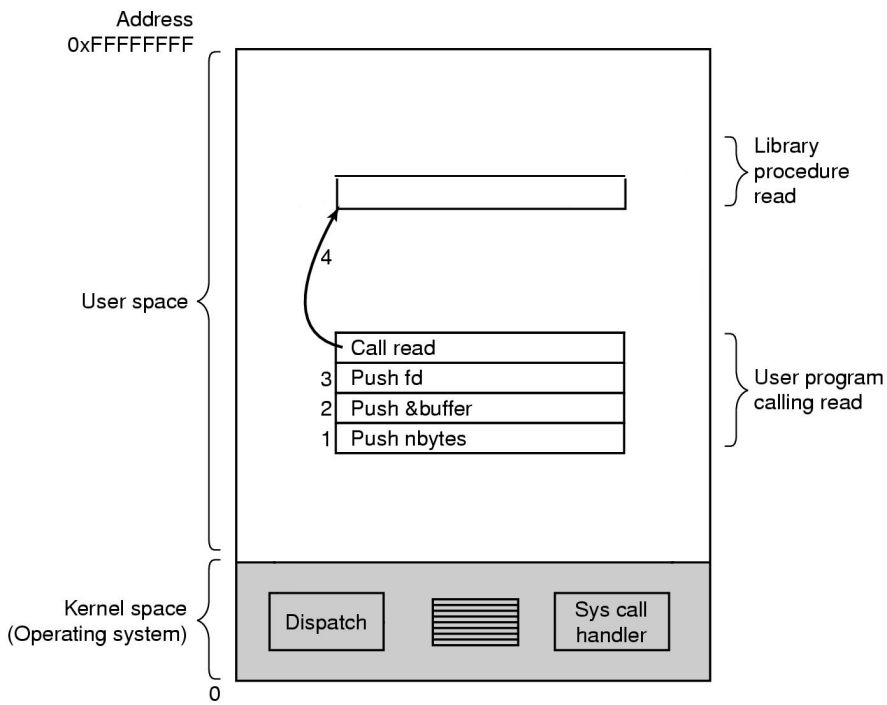
# Chamadas ao Sistema

- read(fd,buffer,nbytes)
  - Chamando read...
    - O programa armazena os parâmetros na pilha de execução (1-3)



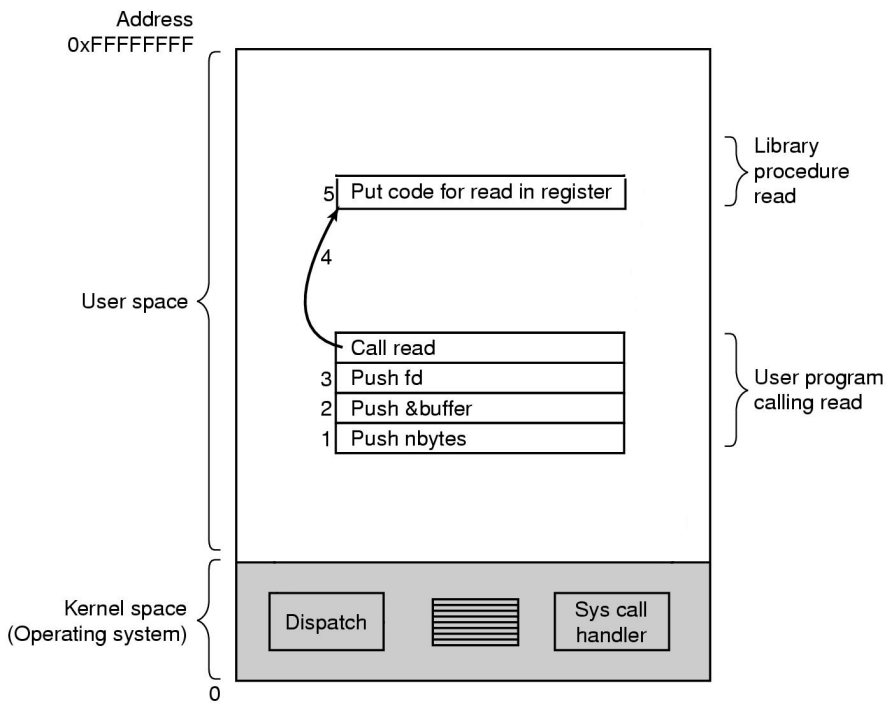
# Chamadas ao Sistema

- read(fd,buffer,nbytes)
  - Chamando read...
    - O programa armazena os parâmetros na pilha de execução (1-3)
    - Chama o procedimento read da biblioteca (4)



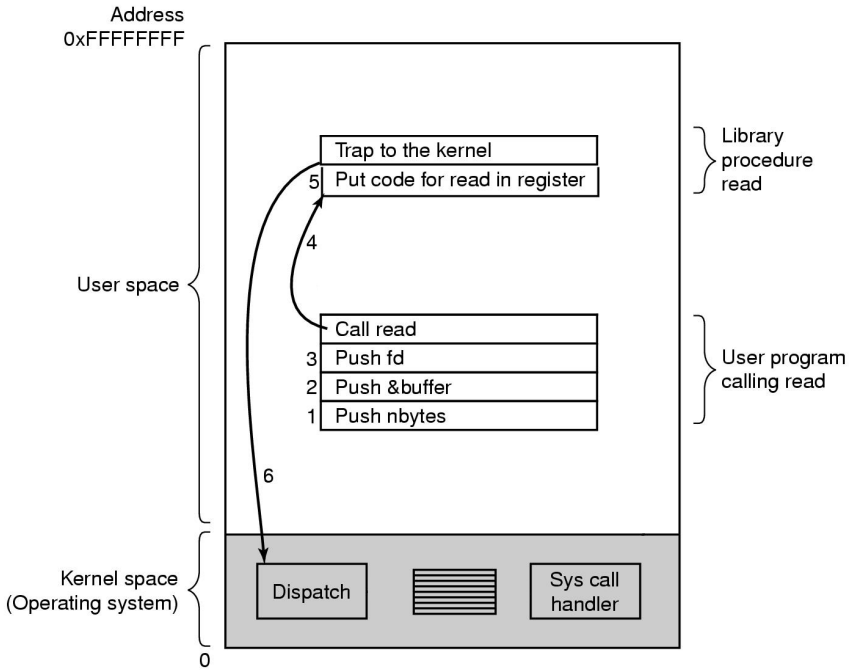
# Chamadas ao Sistema

- read(fd,buffer,nbytes)
  - Chamando read...
    - O programa armazena os parâmetros na pilha de execução (1-3)
    - Chama o procedimento read da biblioteca (4)
  - Executando read...
    - Read coloca o número da chamada ao SO em um registrador específico (5)



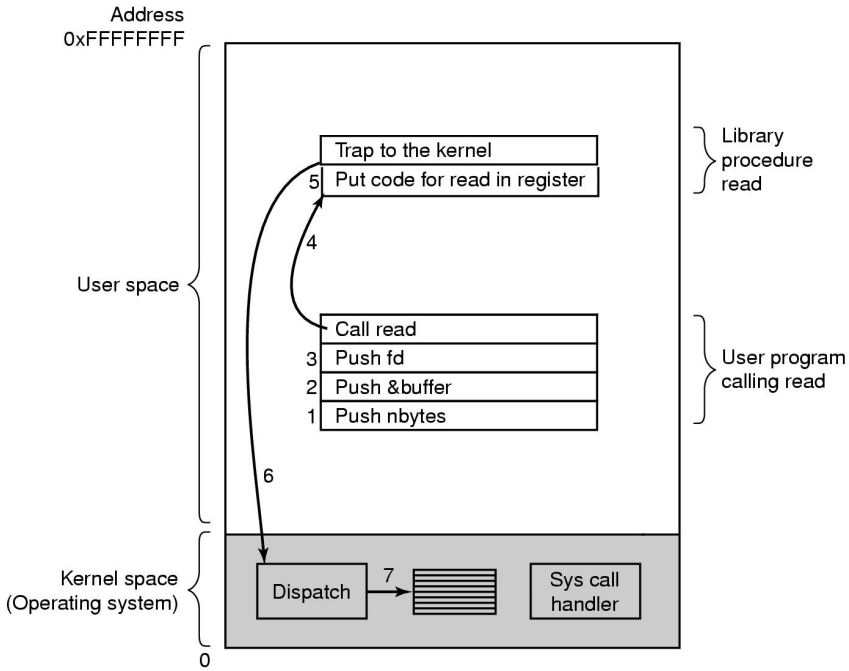
# Chamadas ao Sistema

- read(fd,buffer,nbytes)
  - Executando read...
    - Read executa uma instrução TRAP (passa do modo Usuário para Kernel e executa um determinado endereço no kernel $\rightarrow$ o S.O. tem o controle) (6)



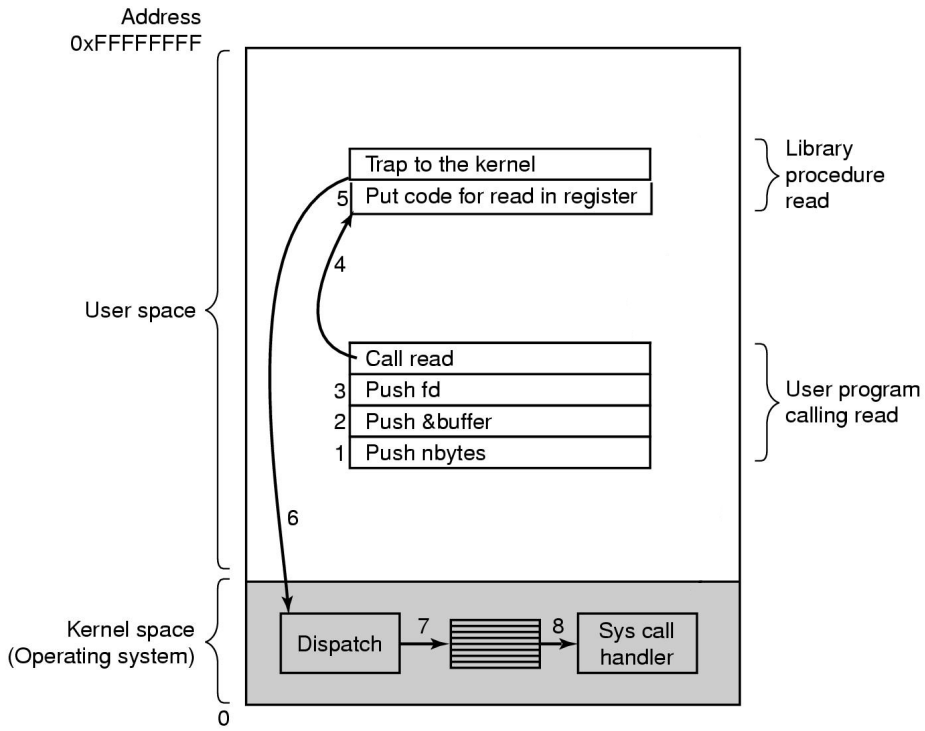
# Chamadas ao Sistema

- read(fd,buffer,nbytes)
  - Executando read...
    - Read executa uma instrução TRAP (passa do modo Usuário para Kernel e executa um determinado endereço no kernel $\rightarrow$ o S.O. tem o controle) (6)
    - O kernel busca os parâmetros, verificando o número da chamada ao SO e chamando o procedimento para seu tratamento (7)
    - Faz isso indexando uma tabela que contém na linha k um ponteiro para a rotina que executa a chamada de sistema k



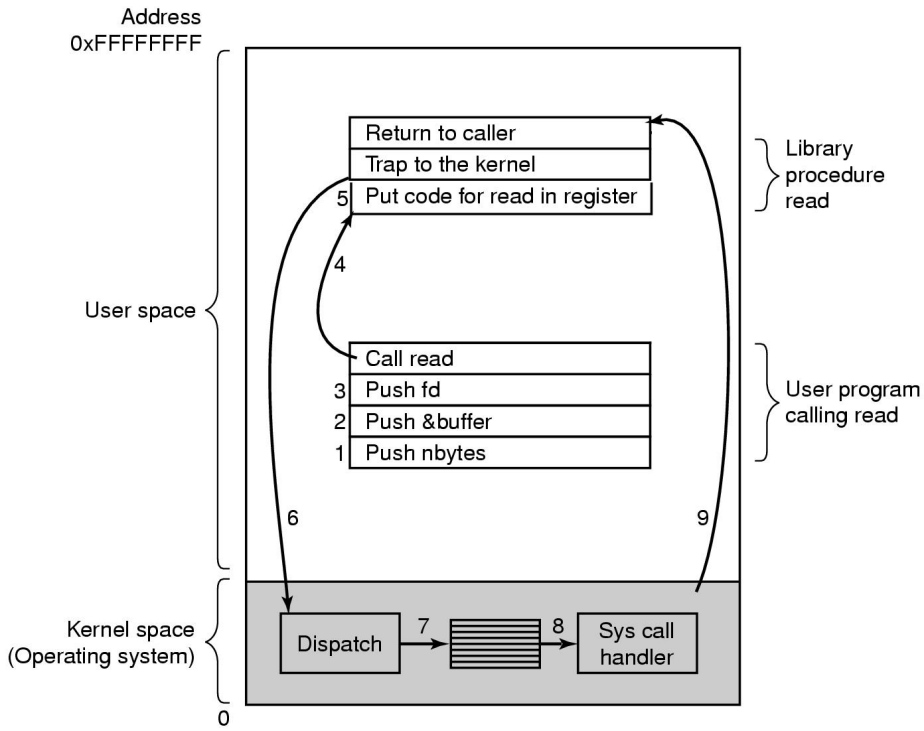
# Chamadas ao Sistema

- read(fd,buffer,nbytes)
  - Executando read...
    - Esse procedimento de tratamento da chamada é executado (8)



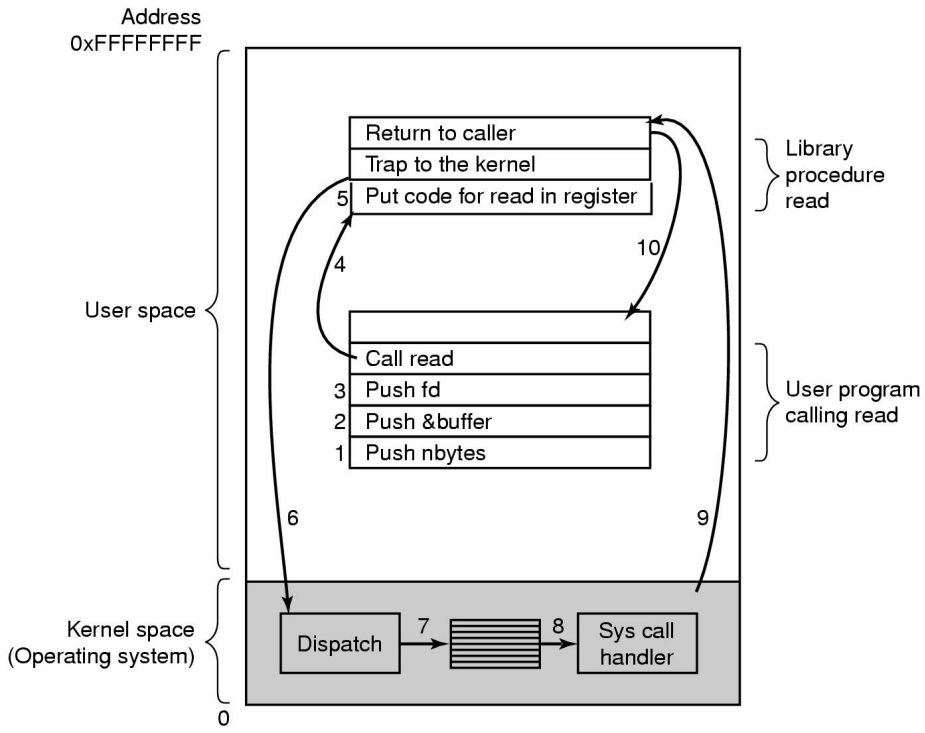
# Chamadas ao Sistema

- read(fd,buffer,nbytes)
  - Executando read...
    - Esse procedimento de tratamento da chamada é executado (8)
    - Terminado o procedimento, o controle pode retornar à instrução seguinte à TRAP (9)



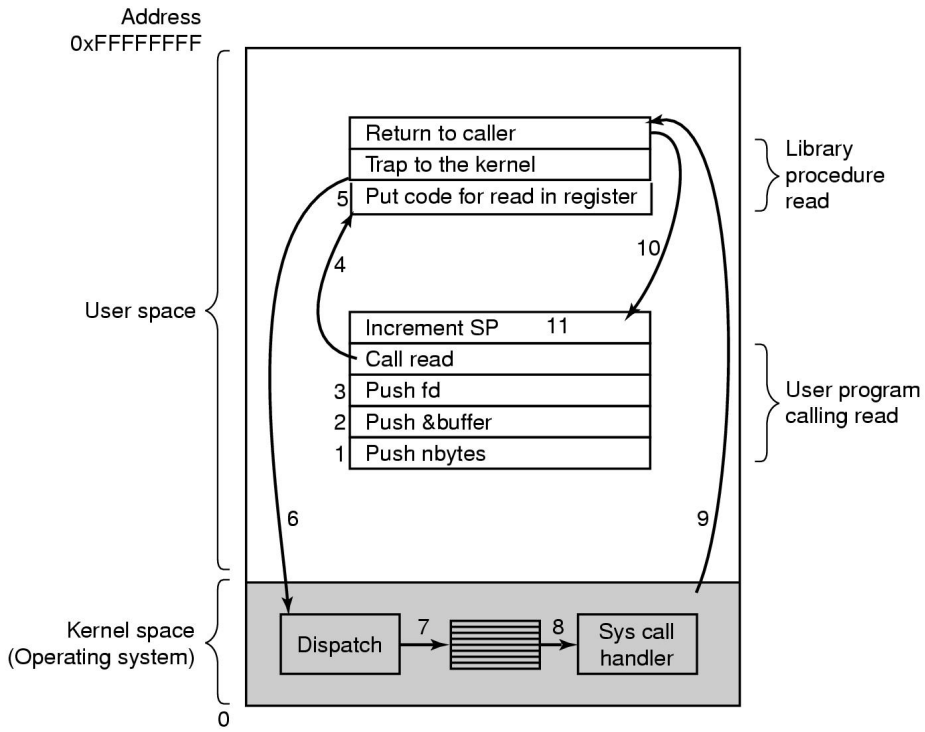
# Chamadas ao Sistema

- read(fd,buffer,nbytes)
  - Executando read...
    - Esse procedimento de tratamento da chamada é executado (8)
    - Terminado o procedimento, o controle pode retornar à instrução seguinte à TRAP (9)
    - Read então retorna ao programa do usuário (10)



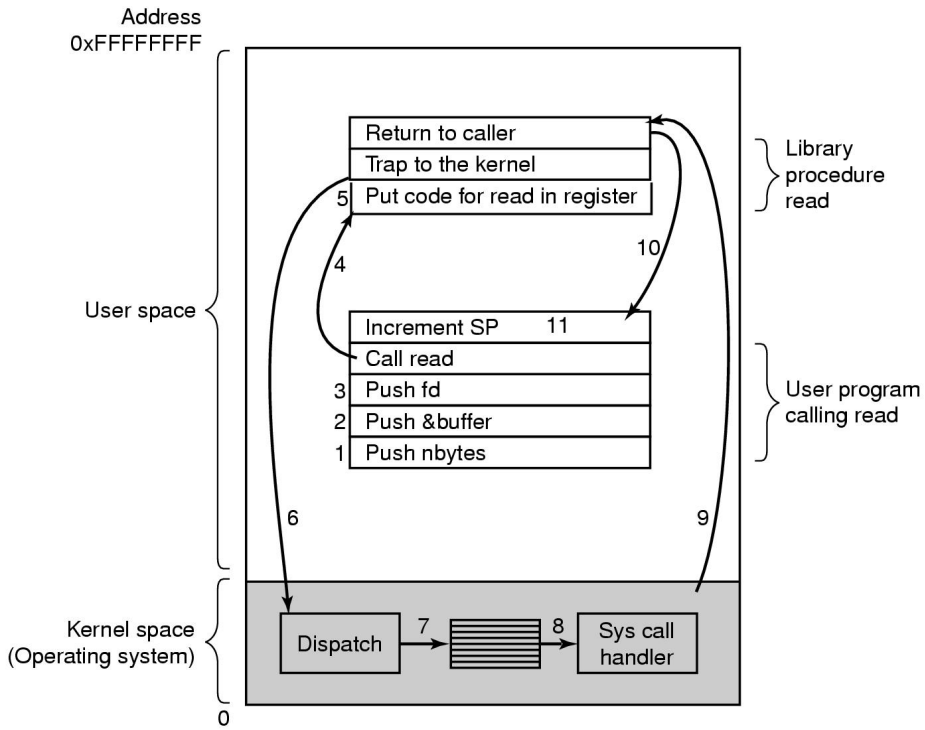
# Chamadas ao Sistema

- read(fd,buffer,nbytes)
  - O programa limpa a pilha após a chamada do procedimento (11)
    - Incrementa o ponteiro da pilha o suficiente para remover os parâmetros da chamada a read (lembre que a pilha está de cabeça para baixo)



# Chamadas ao Sistema

- read(fd,buffer,nbytes)
  - Voltemos ao passo 9
    - Em vez de retornar, a chamada pode bloquear quem a chamou
    - Ex: esperando do teclado
    - O S.O. nesse caso verifica se algum outro processo pode ser executado




- Quando a entrada estiver disponível, os passos 9 a 11 desse processo podem ser executados

# Chamadas ao Sistema – Invocação direta

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

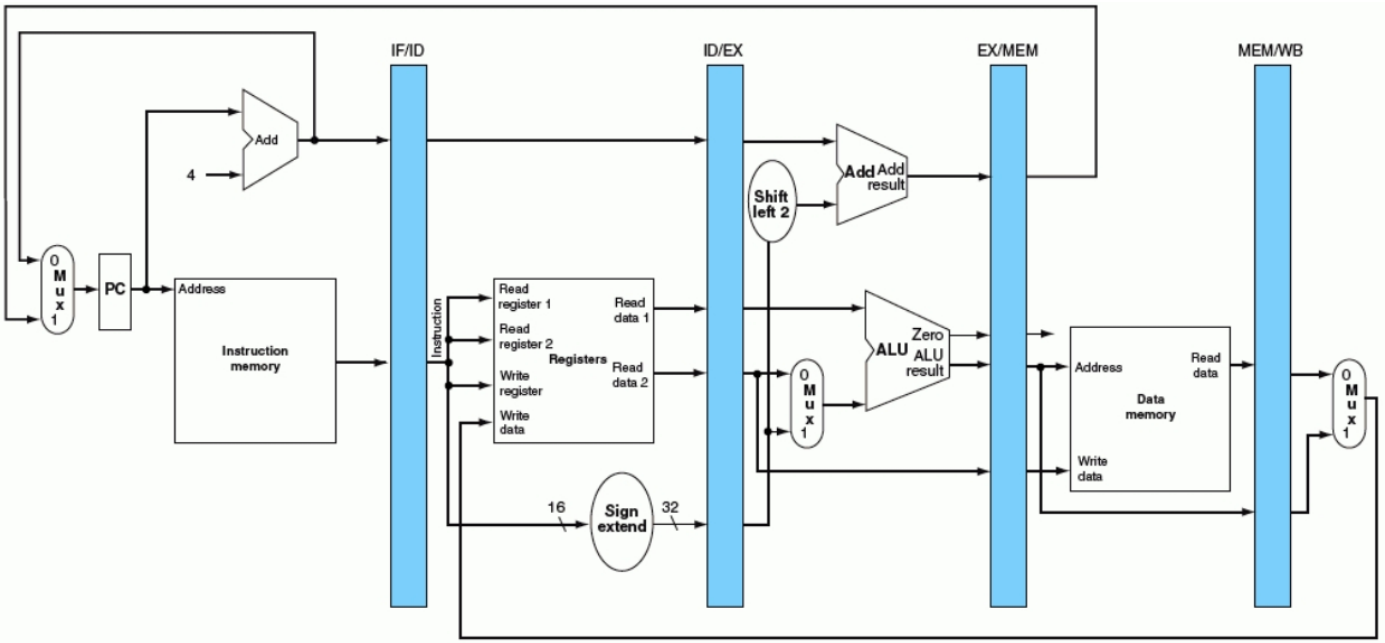
```
section    .data                        ;declaração da seção
    msg    db    "Ola, mundo!",0xa     ;nosso string
    len    equ   $ - msg               ;tamanho do nosso string
section    .text ;declaração de seção
    global _start    ;ponto de entrada para o linker (ld)

_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
```

# Invocação direta – Pé no Hardware

- O que acontece que esquecemos de chamar exit() e o programa ainda detém tempo de CPU?

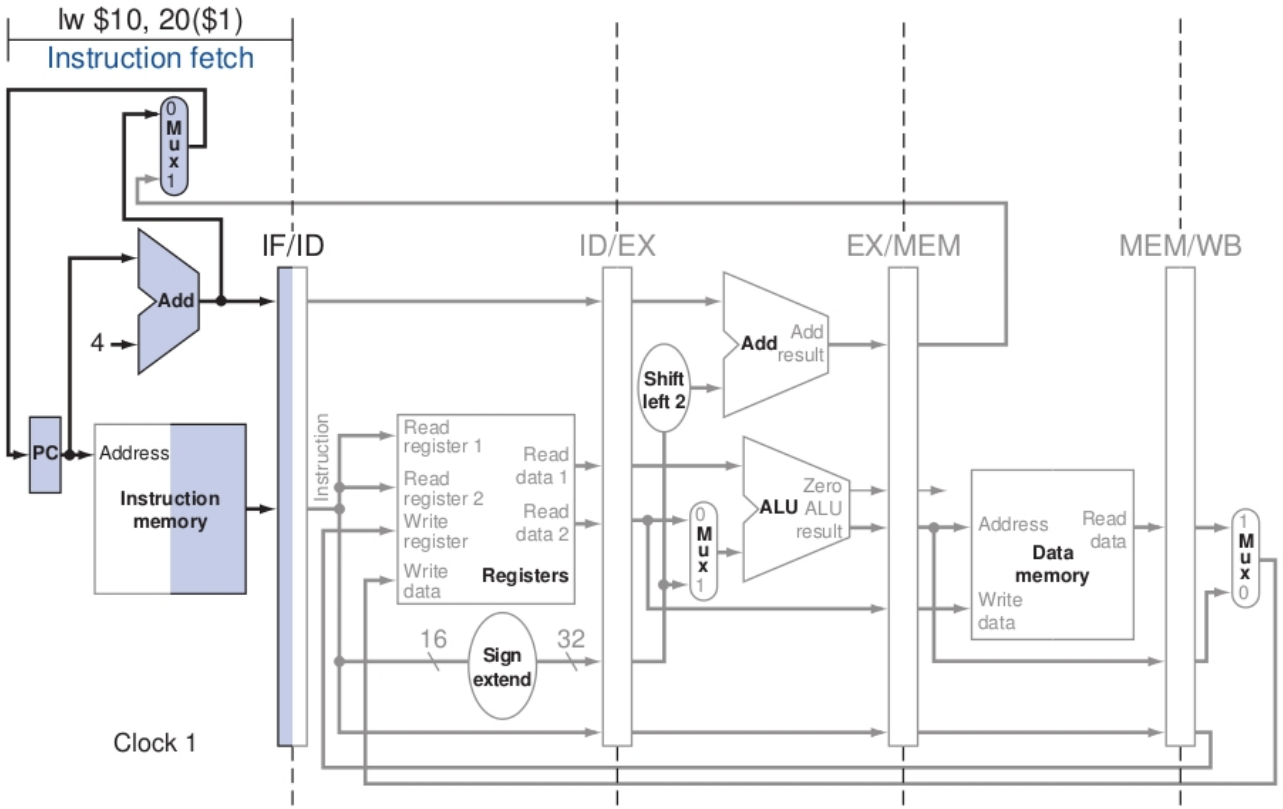
```
;e sai
mov     ebx,0     ;primeiro argumento da chamada ao sistema: código de saída
mov     eax,1     ;número da chamada ao sistema (sys_exit)
int     0x80      ;chama o kernel
```

- Olhemos a pipeline (BEM simplificada):



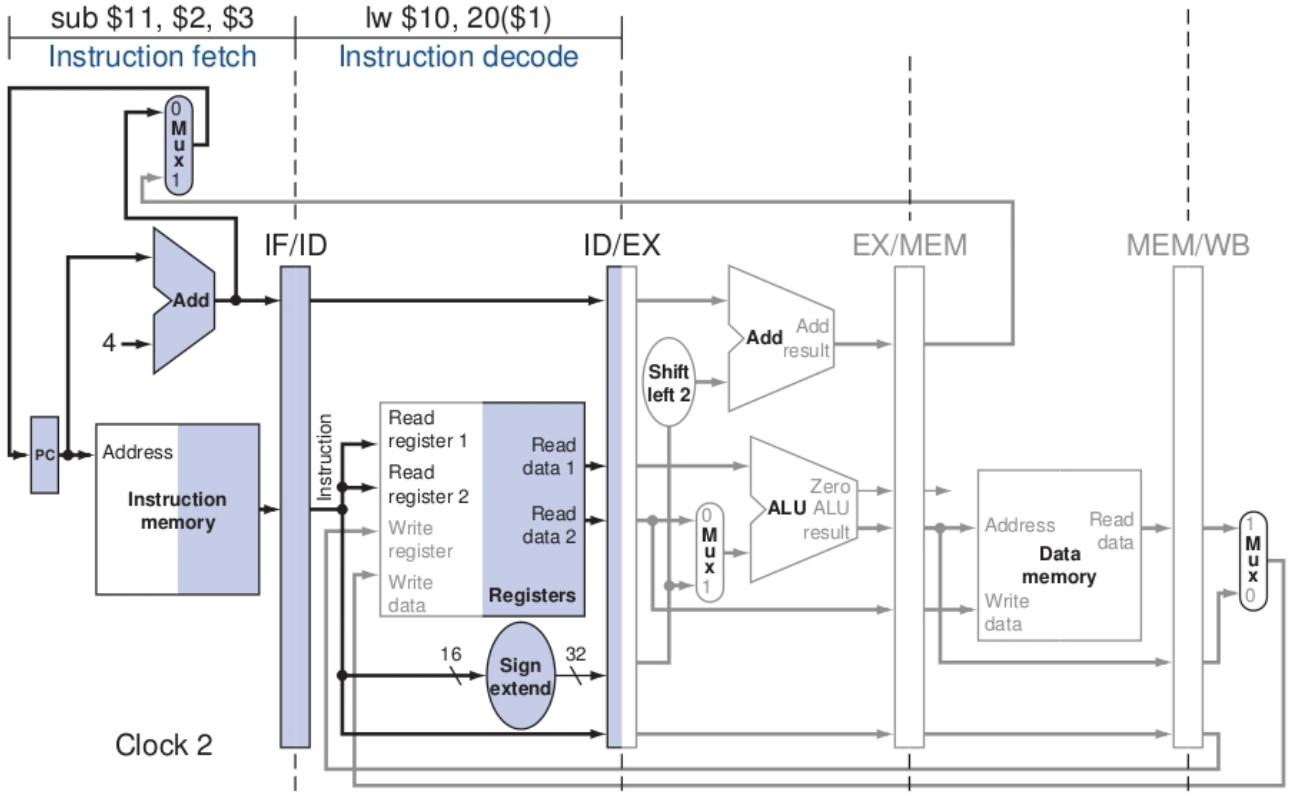
# Invocação direta – Pé no Hardware

- Primeiro ciclo de clock
  - A instrução é buscada na memória
  - 4 (bytes $\rightarrow$ 32b) são somados ao PC e armazenados nele.



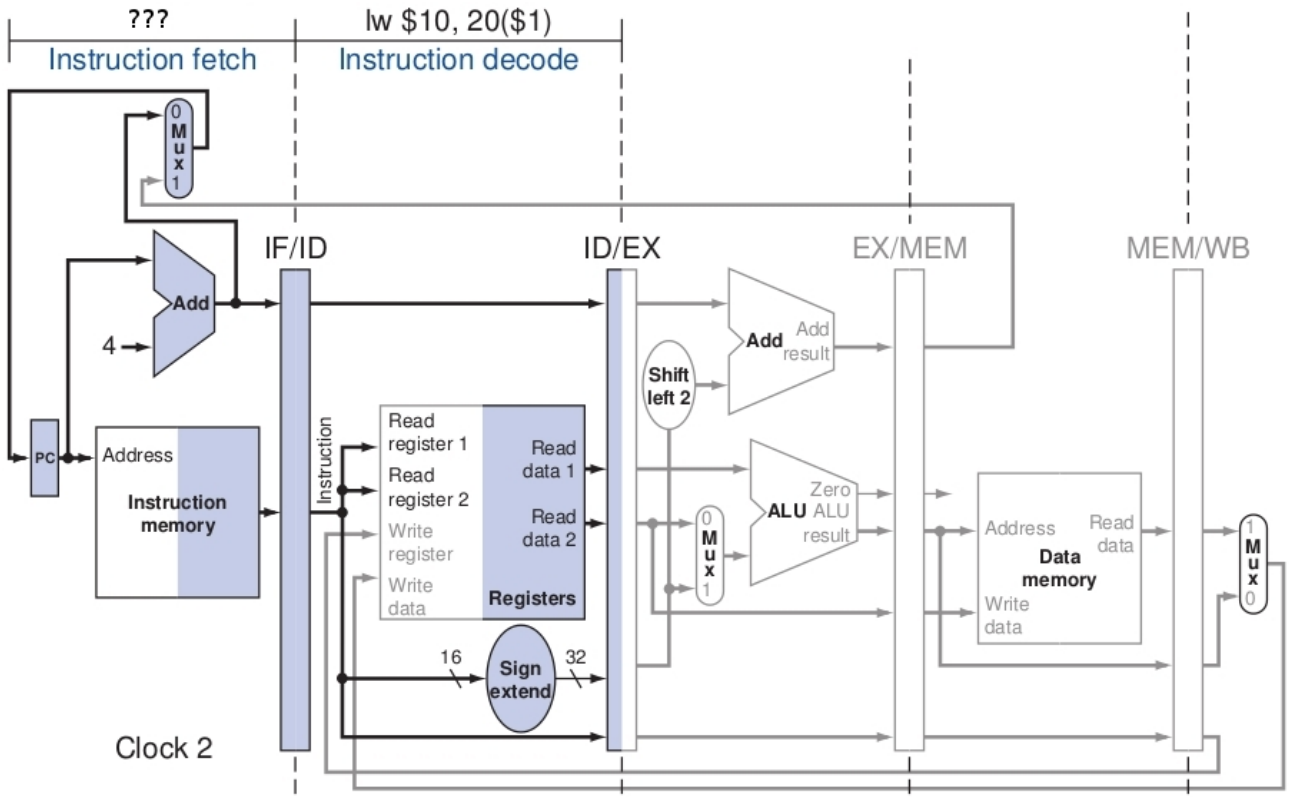
# Invocação direta – Pé no Hardware

- Segundo ciclo de clock
  - Nova instrução é buscada na memória (enquanto a outra está no estágio 2)
  - 4 (bytes $\rightarrow$ 32b) são somados ao PC e armazenados nele.



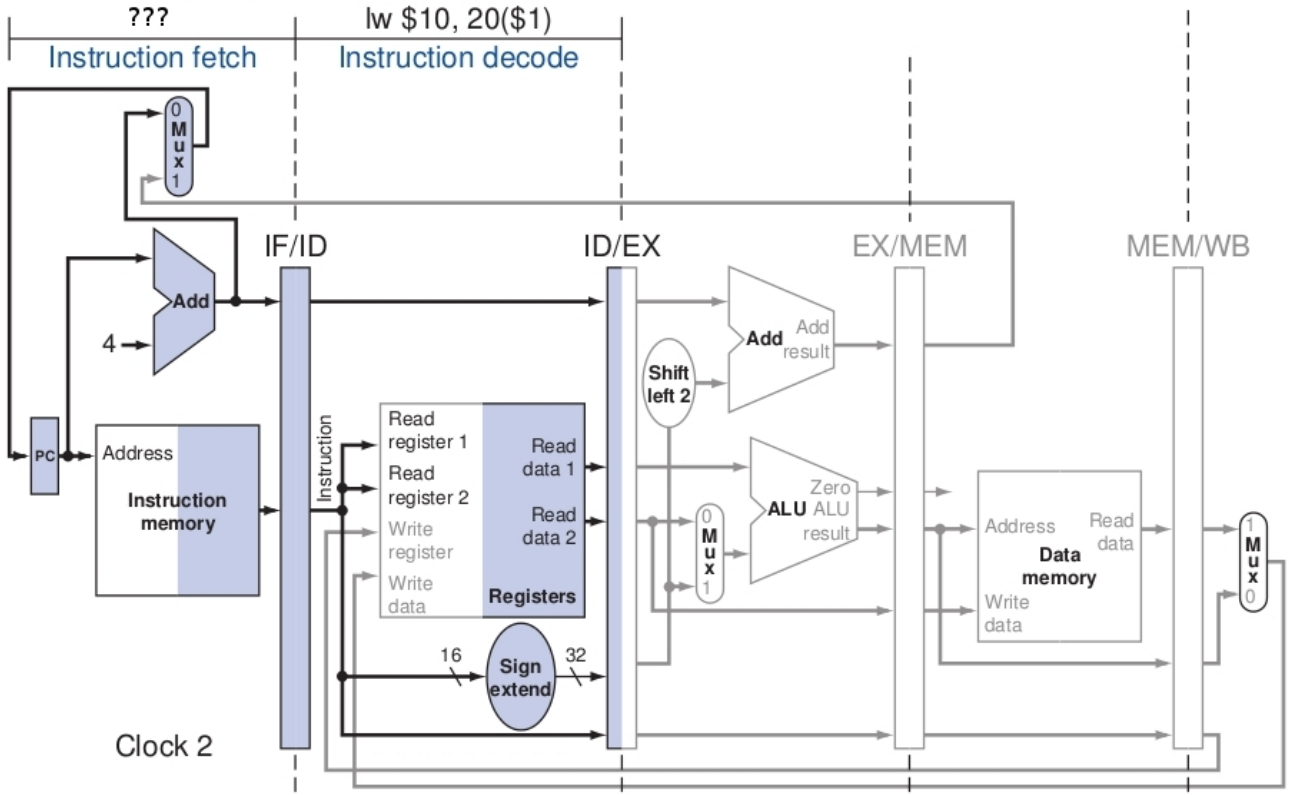
# Invocação direta – Pé no Hardware

- Segundo ciclo de clock
  - Mas se o programa havia acabado, que instrução é essa?
    - O que há na memória naquela posição



# Invocação direta – Pé no Hardware

- Segundo ciclo de clock
  - A instrução pode ser algo que não deveríamos fazer (pouco provável)
  - Ou instrução desconhecida → instrução ilegal



# Invocação direta – exit()

- Esquecer de chamá-lo pode fazer com que:
  - Uma instrução ilegal seja rodada $\rightarrow$ o programa é forçosamente parado
  - Uma instrução legal, porém indesejada, seja rodada $\rightarrow$ comportamento imprevisível
- Por conta disso compiladores sempre incluem um exit() ao final do código compilado, se o programador não o fizer explicitamente

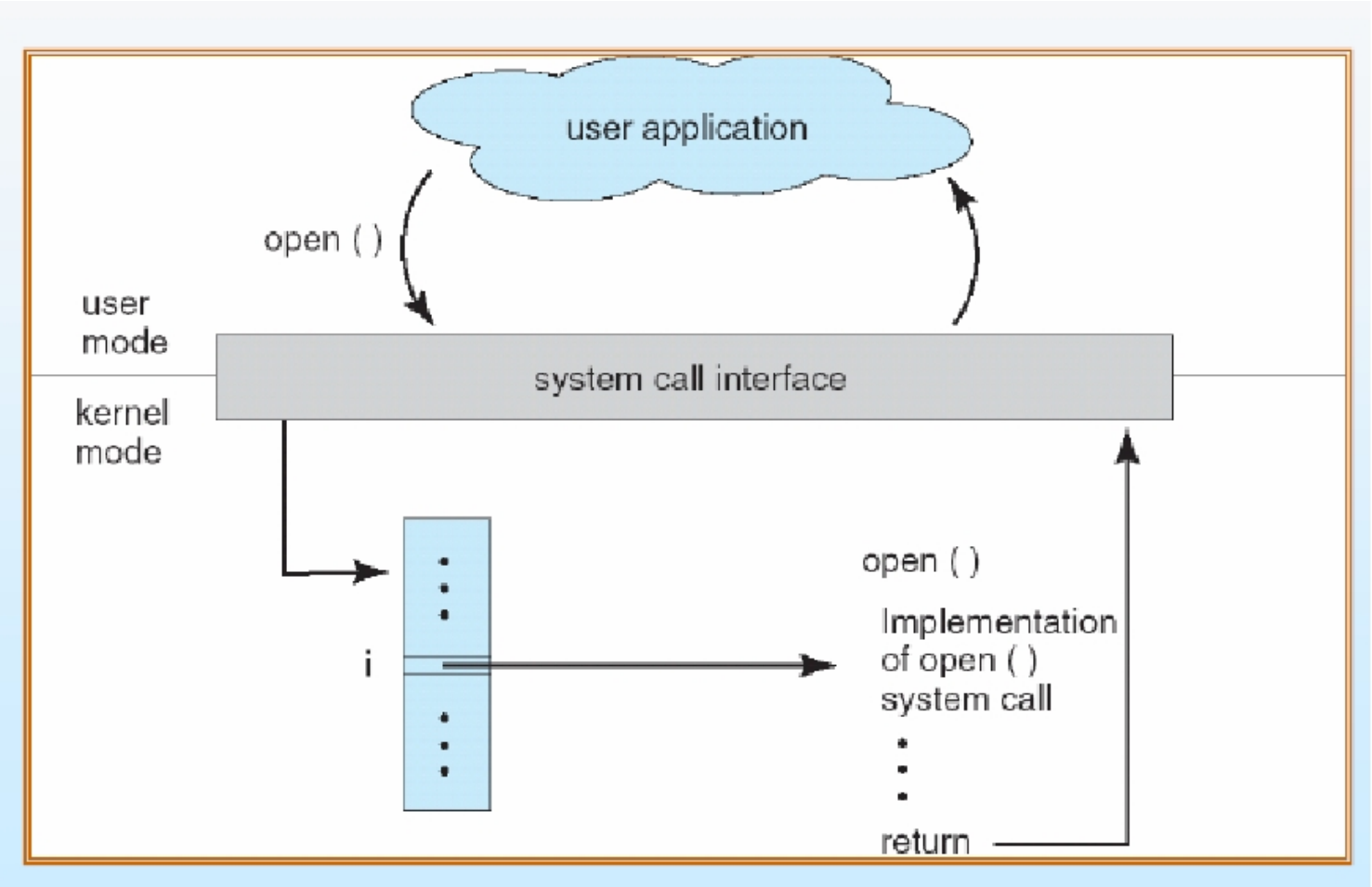
# Chamadas ao Sistema

- Outros exemplos:
  - Chamadas para gerenciamento de processos:
    - Fork (CreateProcess – WIN32) – cria um processo;
  - Chamadas para gerenciamento de diretórios:
    - Mount – monta um diretório;
  - Chamadas para gerenciamento de arquivos:
    - Close (CloseHandle – WIN32) – fechar um arquivo;
  - Outros tipos de chamadas:
    - Chmod: modifica permissões de arquivos, diretórios, aplicativos etc;

# Chamadas ao Sistema

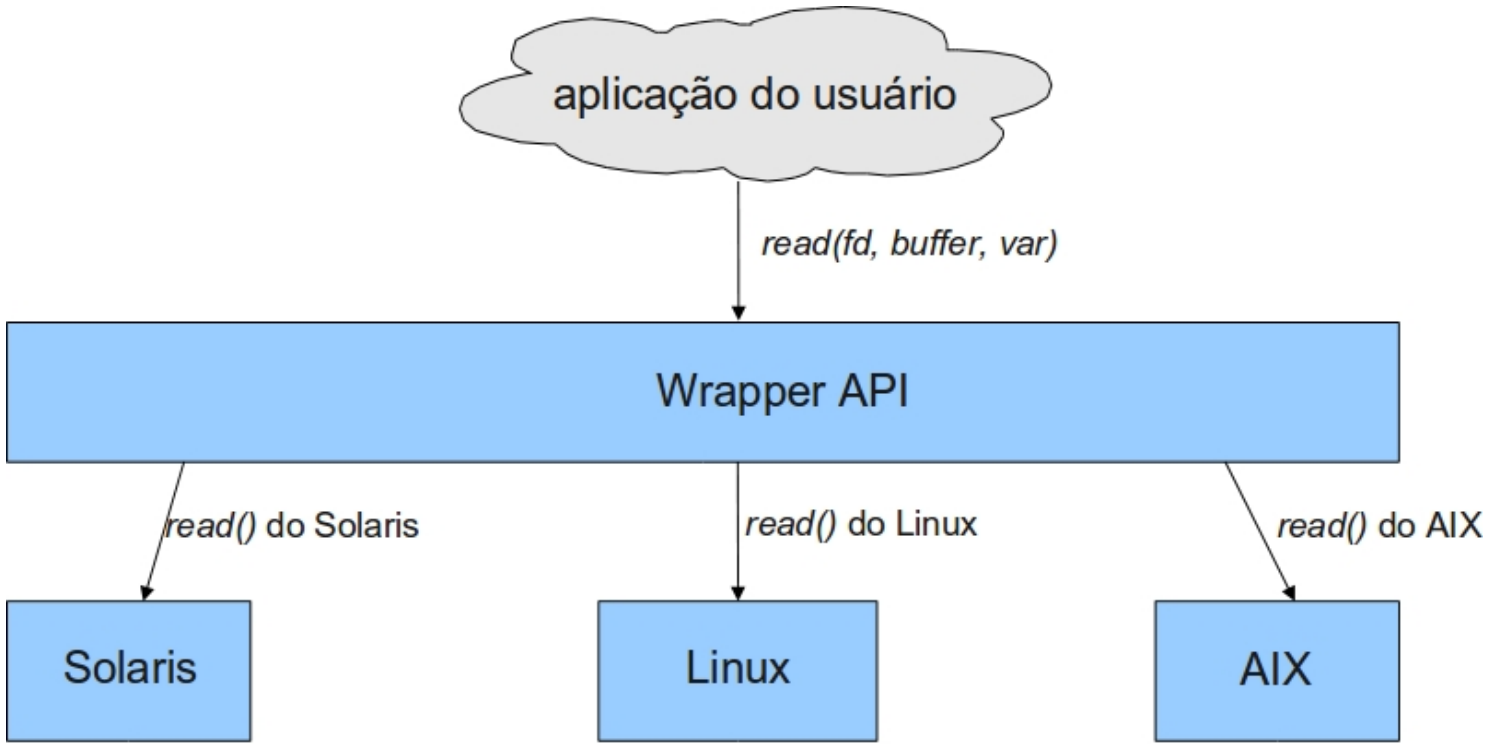
- Interface das Chamadas ao Sistema (Wrappers)
  - Chamadas a bibliotecas
    - Em Windows, desacopladas das chamadas reais ao sistema
  - Interface de programação fornecida pelo SO
  - Geralmente escrita em linguagem de alto nível (C, C++ ou Java)
  - Normalmente as aplicações utilizam uma Application Program Interface (API)
    - Interface que encapsula o acesso direto às chamadas ao sistema

# Chamadas ao Sistema



# Chamadas ao Sistema

- Portabilidade usando Wrappers

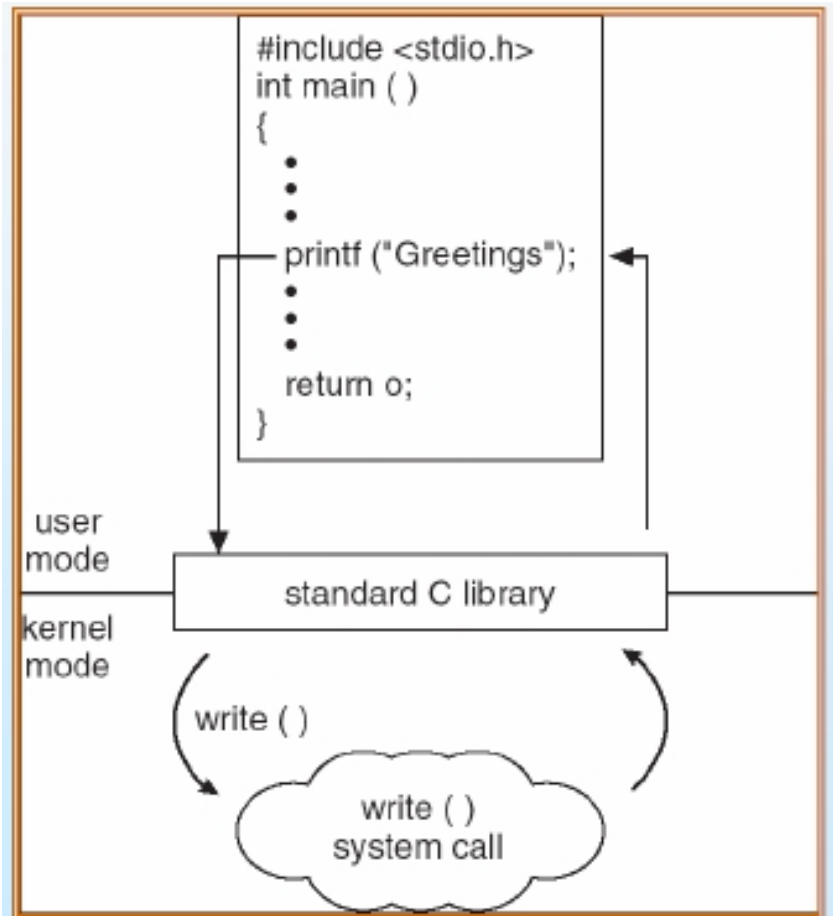


# Chamadas ao Sistema

- Interface das Chamadas de Sistema (Wrappers)
  - Mais utilizadas:
    - Win32 API para Windows
    - POSIX API para praticamente todas as versões de UNIX
    - Java API para a Java Virtual Machine (JVM).
  - Motivos para utilizar APIs em vez das chamadas ao sistema diretamente
    - Portabilidade – independência da plataforma
    - Esconder complexidade inerente às chamadas ao sistema
    - Acréscimo de funcionalidades que otimizam o desempenho

# Chamadas ao Sistema

- 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*



# Processos

- Conceito central do SO;
  - Um processo é caracterizado por um programa em execução
  - Armazena todas as informações necessárias para executar um programa
- Mas existe uma diferença sutil entre processo e programa:
  - Um processo pode ser composto por vários programas, dados de entrada, dados de saída e um estado (executando, bloqueado, pronto)
  - Contém o código do programa e sua atividade atual

# Processos × Programas

- Programa:
  - Um programa pode ter várias instâncias em execução (em diferentes processo)
  - Algoritmo codificado
  - Forma como o programador vê a tarefa a ser executada
- Processo:
  - Um processo é único
  - Código acompanhado de dados e momentos de execução (contexto)
  - Forma pela qual o SO vê um programa e possibilita sua execução

# Processos

- Em primeiro plano:
  - Interagem com o usuário
    - Leitura de um arquivo;
    - Iniciar um programa (linha de comando ou um duplo clique no mouse);

# Processos

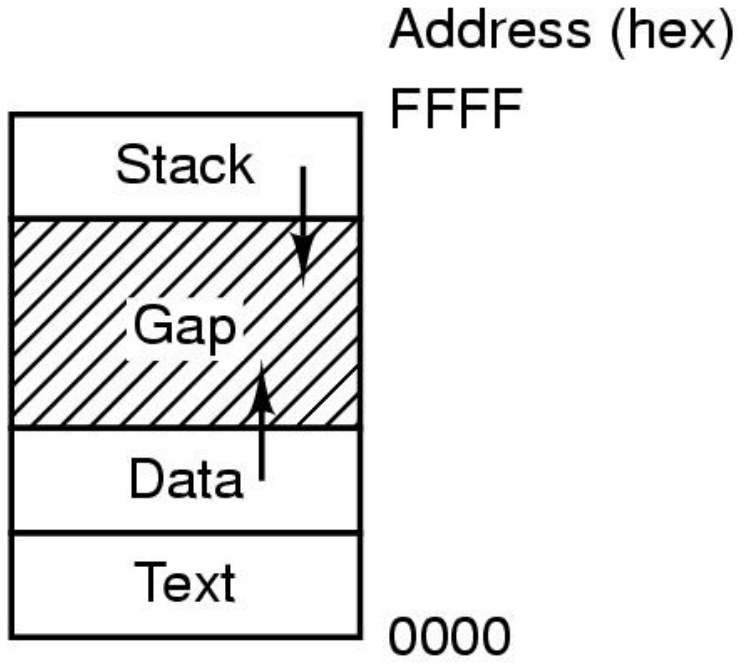
- Em segundo plano:
  - Processos com funções específicas que independem de usuários – daemons:
    - Recepção e envio de emails;
    - Serviços de Impressão;

# Processos

- Cada processo possui:
  - Programa (instruções que serão executadas);
  - Um espaço de endereçamento:
    - Lista de posições de memória, variando de 0 a um máximo, que o processo pode ler e escrever
  - Contextos de hardware: informações de registradores;
    - PC, Ponteiro da pilha, registradores de uso geral etc
  - Contextos de software: atributos;
    - Variáveis, lista de arquivos abertos, alarmes pendentes, processos relacionados etc

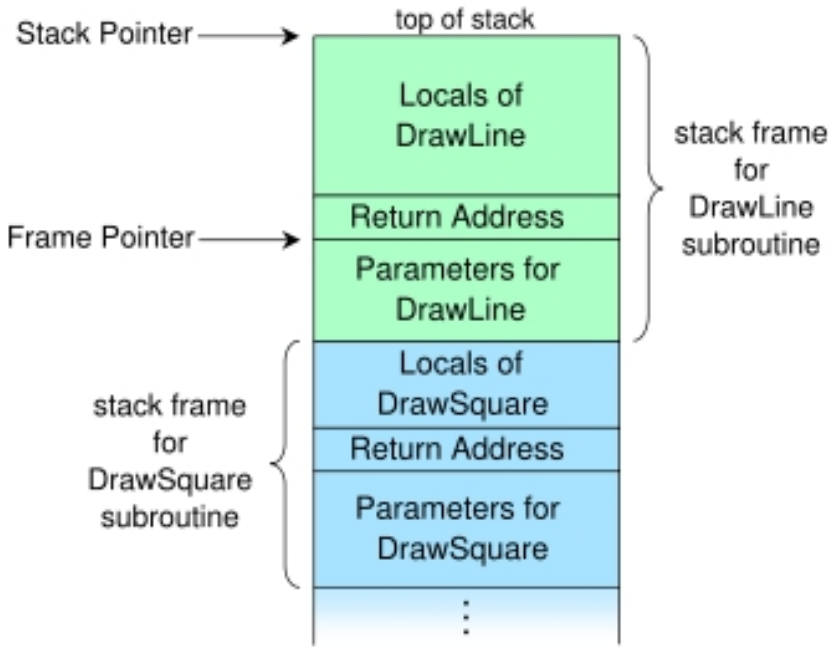
# Processos – Espaço de endereçamento

- Basicamente, possui três segmentos:
  - Texto:
    - código executável do(s) programa(s);
  - Dados:
    - as variáveis;
  - Pilha de Execução:
    - Controla a execução do processo
    - Empilhando chamadas a procedimentos, seus parâmetros e variáveis locais etc



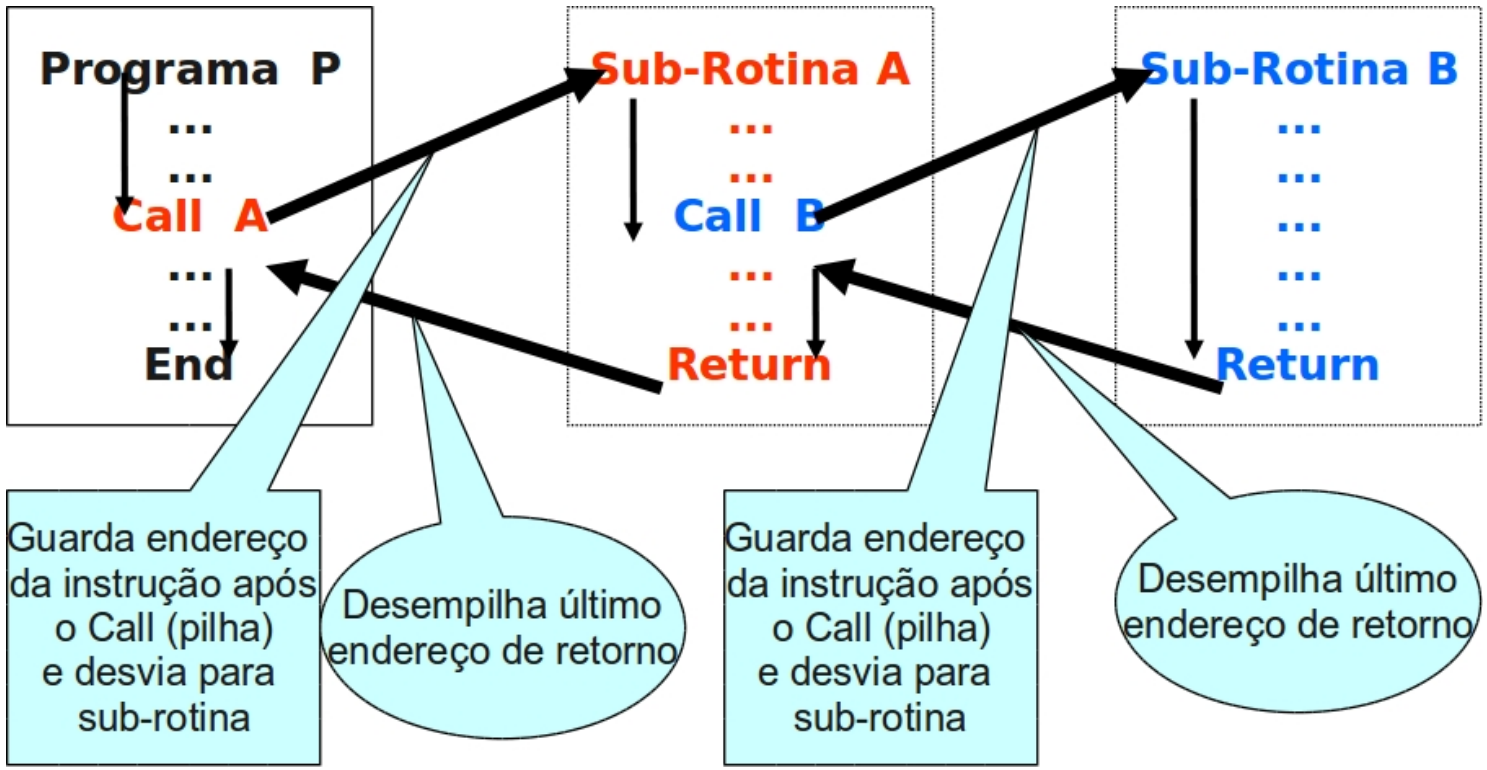
# Processos – Espaço de endereçamento

- Pilha de execução:
  - Contém uma estrutura para cada rotina chamada que ainda não retornou
  - Possui:
    - Variáveis locais da rotina
    - Endereço de retorno a quem chamou
    - Parâmetros



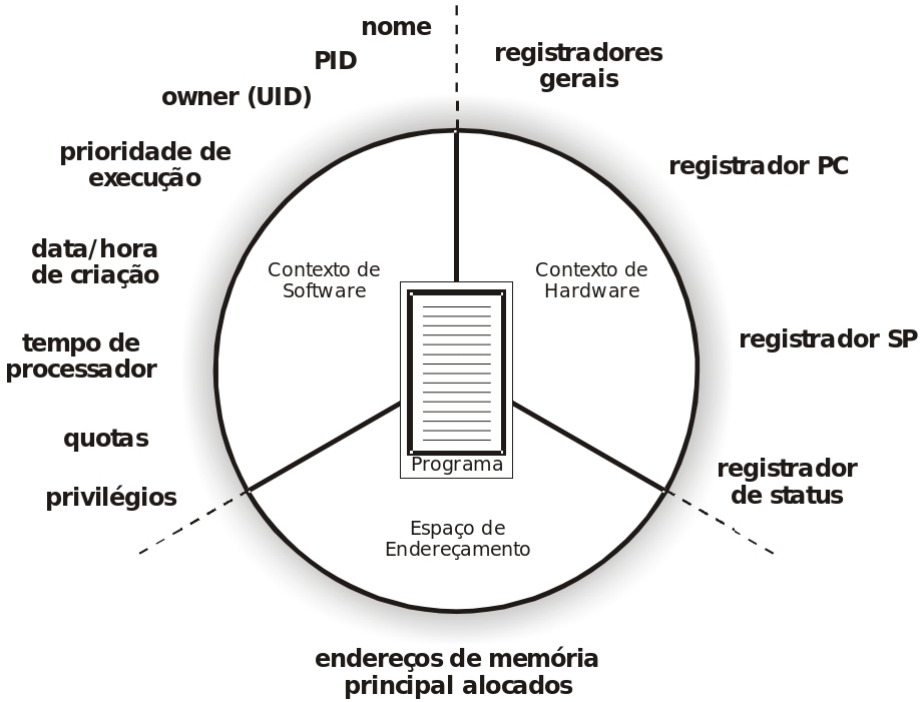
# Subrotinas e a Pilha de Execução

- Funcionamento de Sub-rotinas



# Processos – Contexto

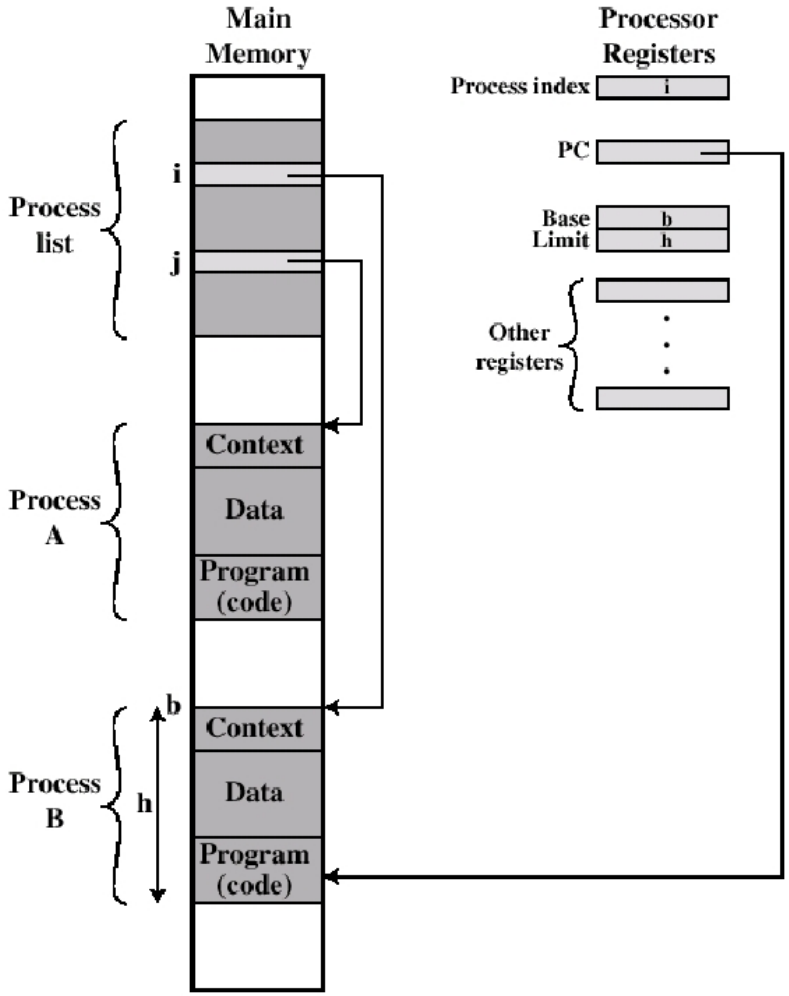
- Contém a informação de que o SO precisa para, após suspender um processo, trazê-lo de volta à execução
  - Ex: Ponteiros de arquivos abertos: posição do próximo byte a ser lido em cada arquivo




- É o estado de uma tarefa em um determinado instante

# Processos – Contextos

- Dentre outras coisas, corresponde ao conteúdo dos registradores usados por um determinado processo, enquanto roda
  - Quando um processo é executado, seu contador de programa é carregado no PC
  - Que programa está rodando ao lado?



# Referências Adicionais


- Patterson, D.A.; Hennessy, J.L: Computer Organization and Design: The Hardware Software interface. 3 ed. Elsevier:Nova Iorque, 2005.
- http://www.codinghorror.com/blog/2008/01/understanding-user-and-kernel-mode.html