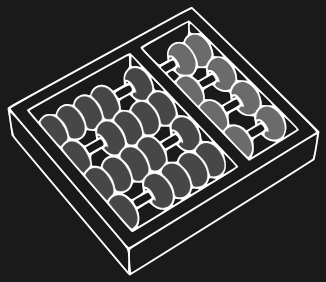


UNICAMP

Kernel

Prof. Carlos A. Astudillo

castudillo@ic.unicamp.br

Sistemas Operacionais (MC504A)

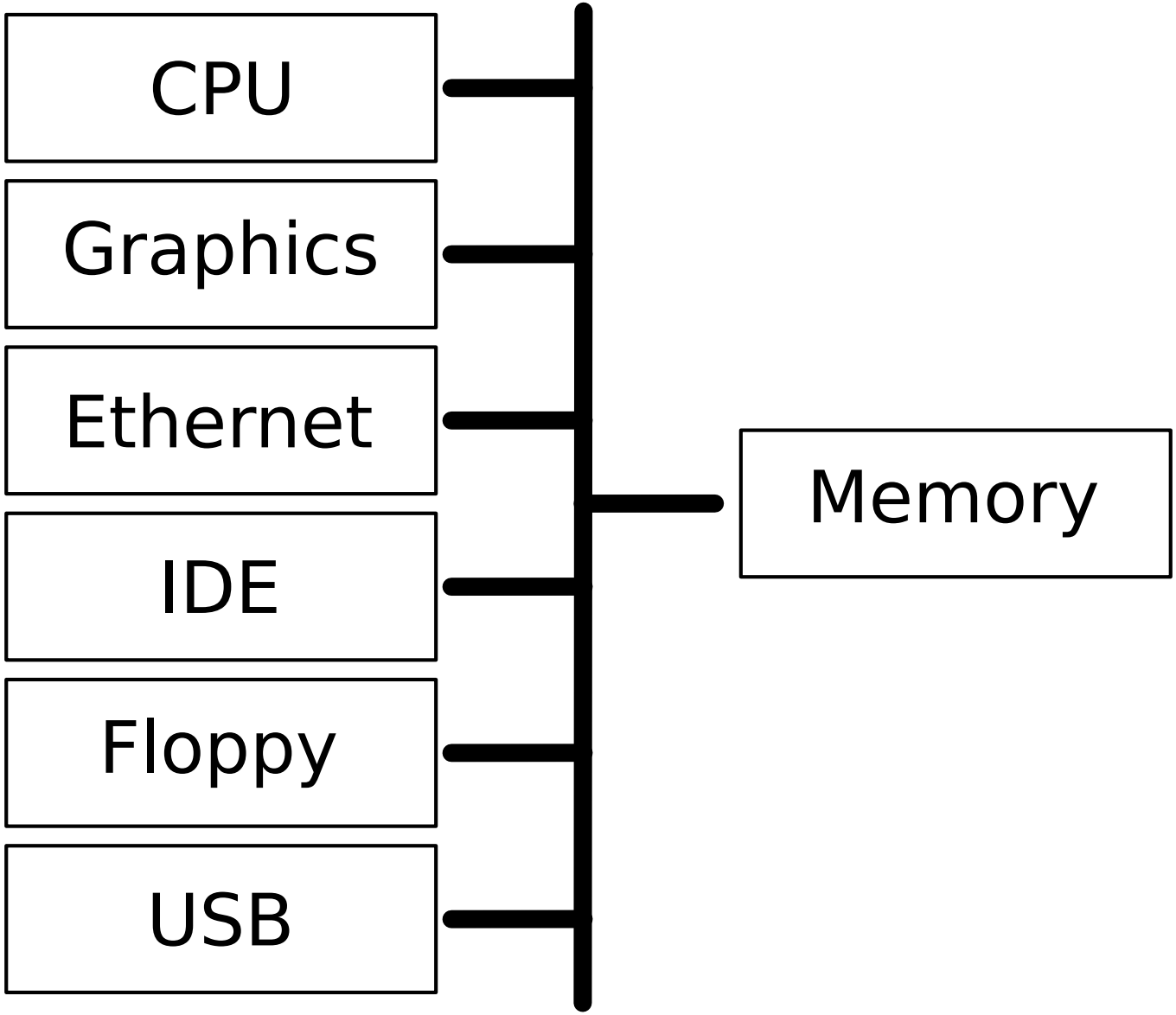
2º Semestre 2023

Adaptado de Dave Eckhard

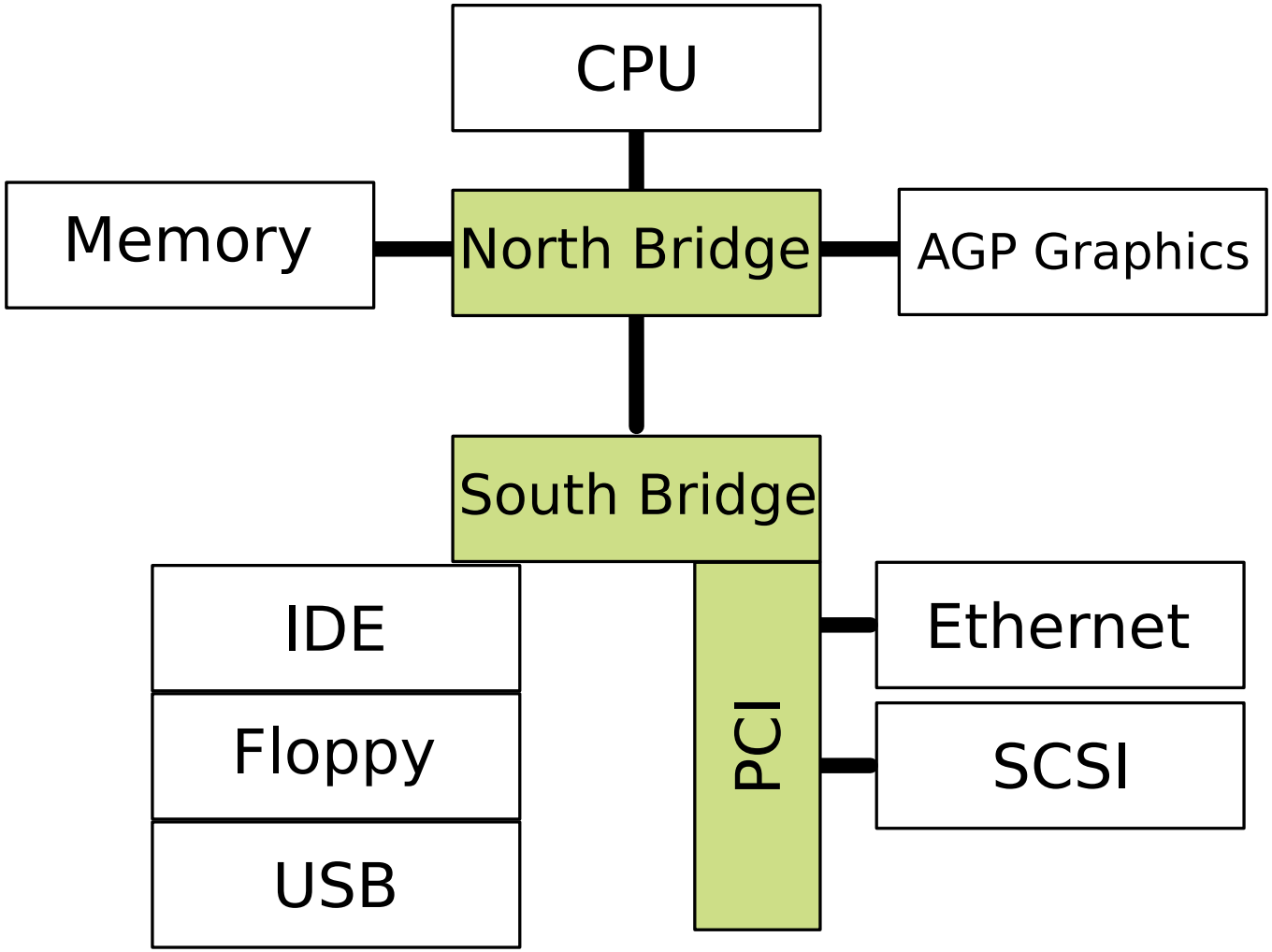
Objetivos da aula

- Hardware
- Estado da CPU
- Chamadas do sistema
- Troca de contexto
- Interrupções
- Condições de corrida
- Emascarado de interrupções

Dentro da computadora: histórico

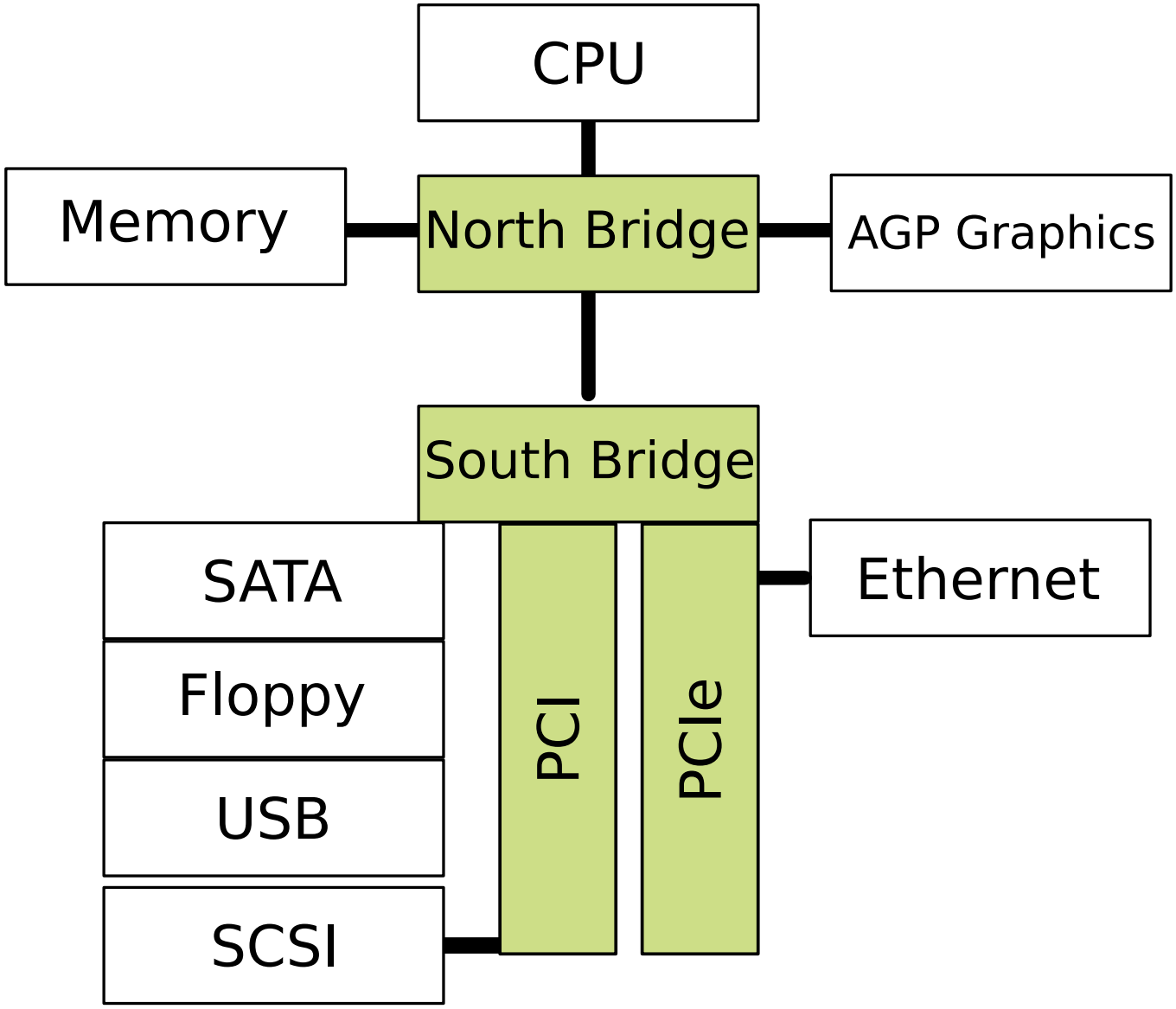


Dentro da computadora: 1997–2004



PCI: Peripheral Component Interconnect

Dentro da computadora: 2004–



PCIe: PCI Express

CPU

- Registradores de usuário (IA32)
  - Propósito geral: `%eax`, `%ebx`, `%ecx`, `%edx`
  - Stack pointer: `%esp`
  - Frame (base) pointer: `%ebp`
  - Registradores misteriosos de strings: `%esi`,`%edi`

CPU

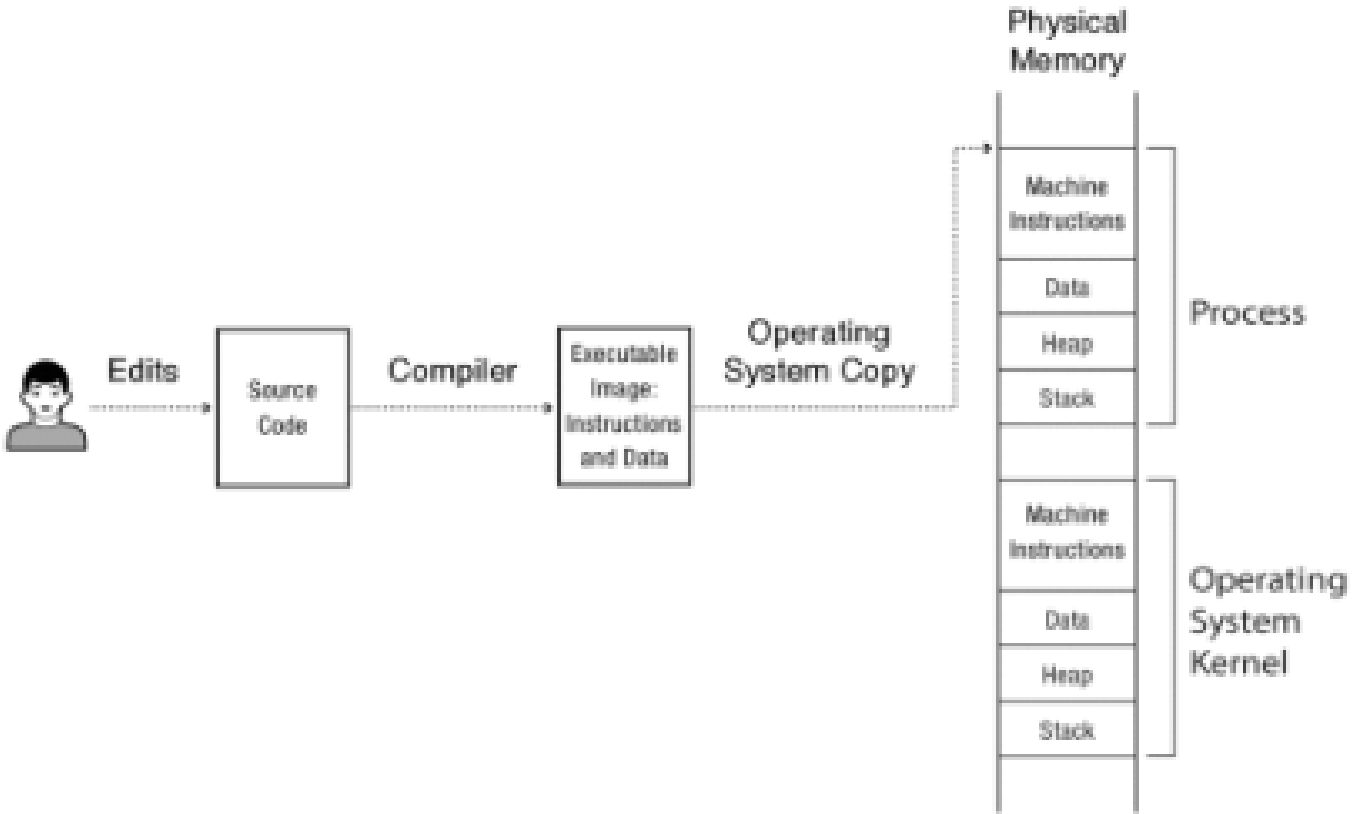
- Registradores que não são do usuário: do estado do processador
  - Modo atual: usuário ou kernel
  - Interrupções: on/off
  - Memoria virtual: on/off
  - Modelo de memória
    - pequeno, médio, grande, etc.

CPU

- Registradores de números de ponto flutuante
  - Parte lógica dos *registradores de usuário*
  - As vezes, são registradores especiais
    - Algumas máquinas não têm operações de ponto flutuante
    - Alguns processos não usam ponto flutuante

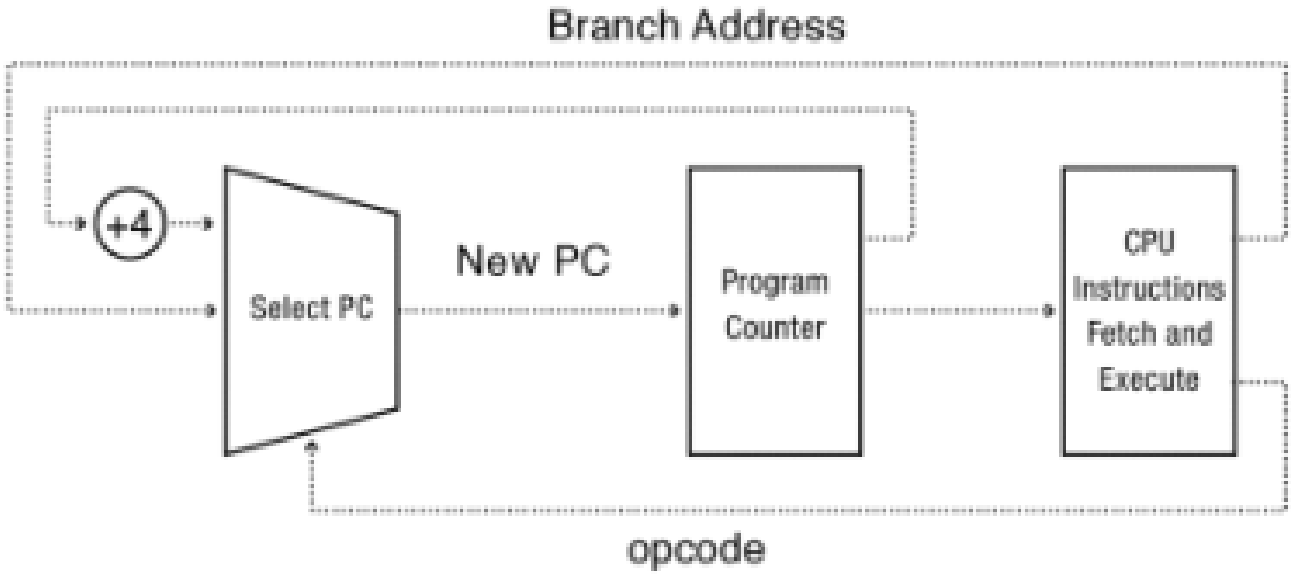
Funcionamento em Modo Dual

Relembrando



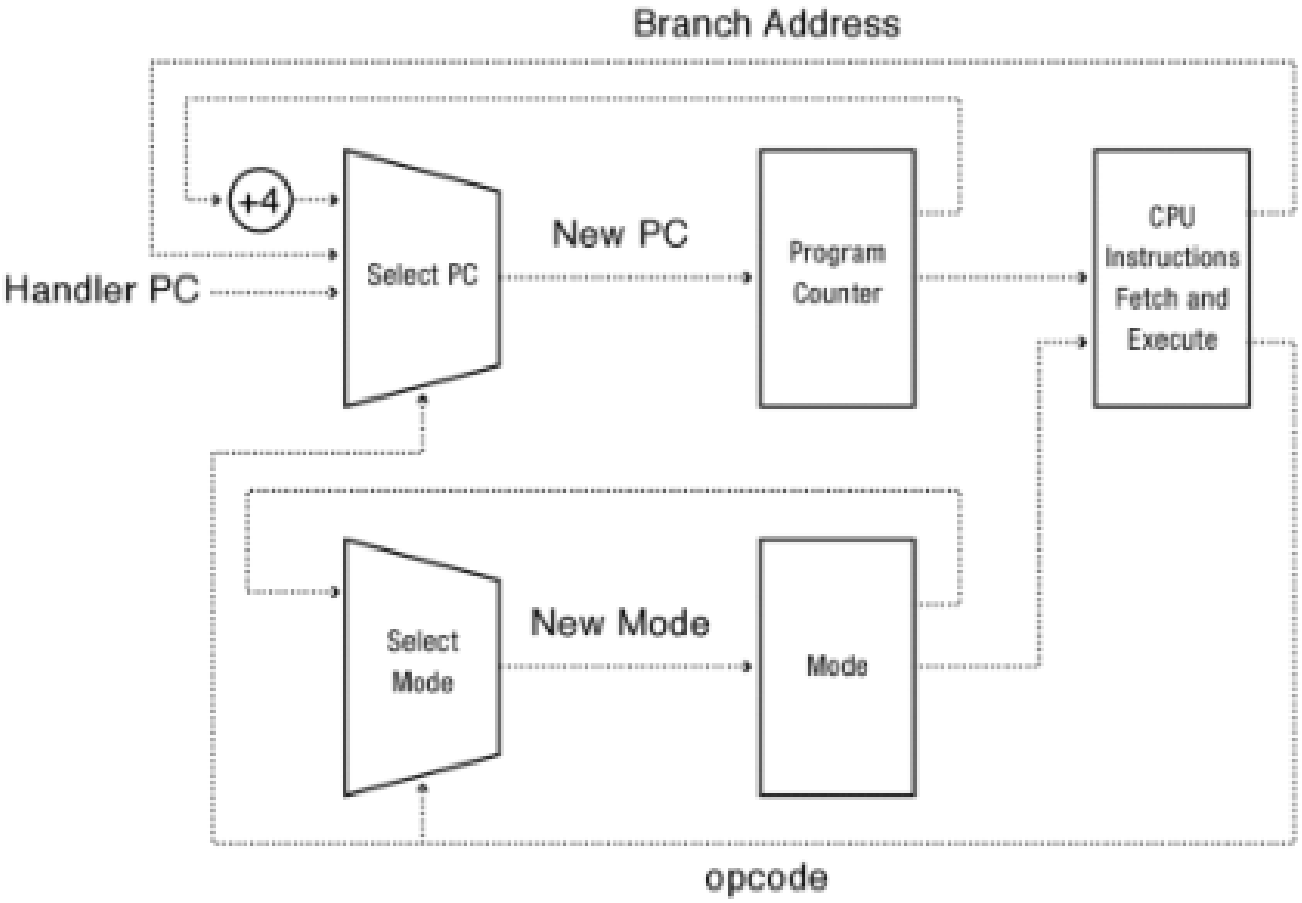
Operação em Modo Dual

Funcionamento Básico de uma CPU



Operação em Modo Dual

Modo Usuário e Modo Kernel



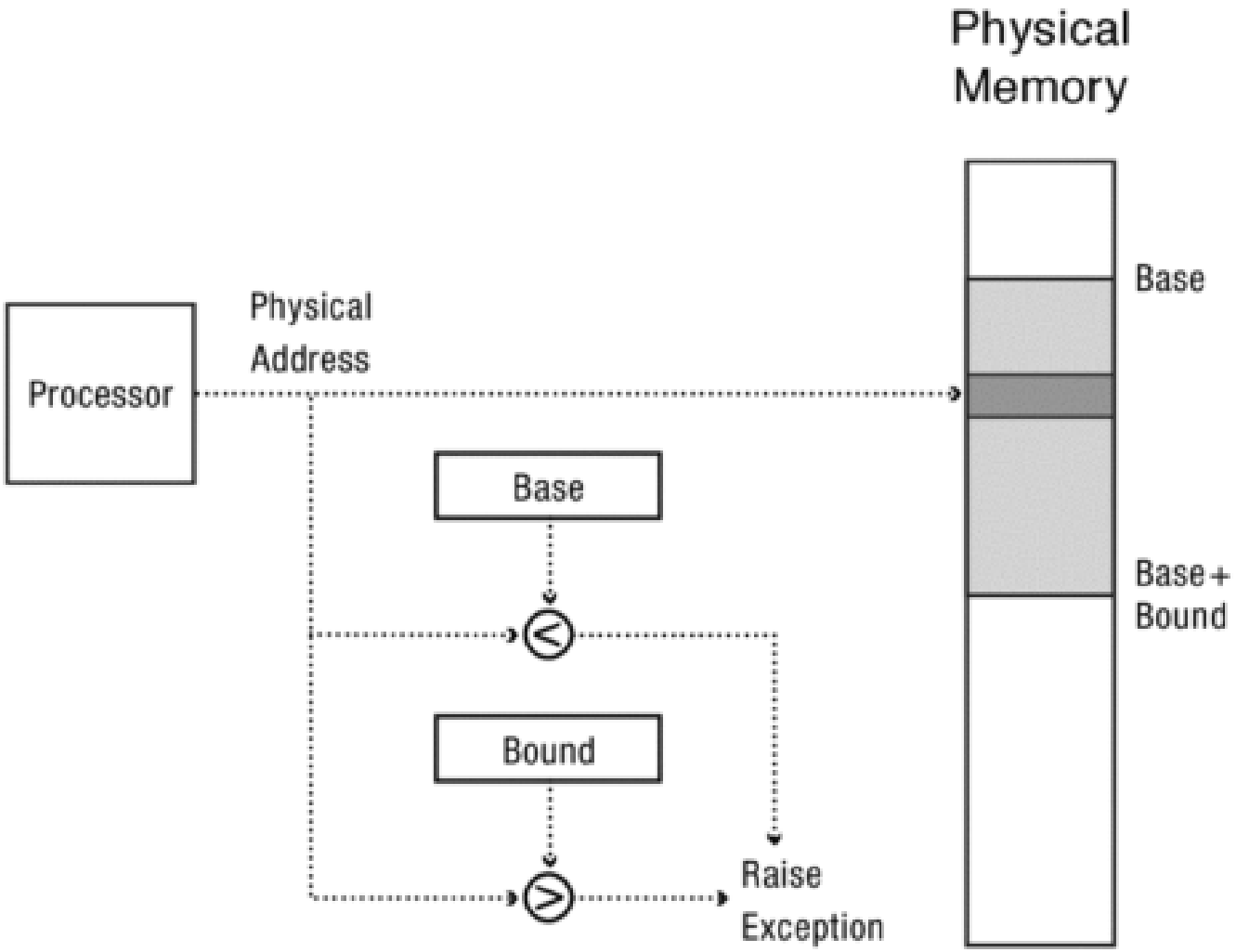
Operação em Modo Dual

O Hardware deve suportar:

- Instruções Privilegiadas: Instruções apenas disponíveis em modo kernel. O SO precisa modificar o nível de privilégio, ajustar o acesso a memória e habilitar e desabilitar interrupcões.
- Proteção à Memória: registradores base and bound para limitar memória acessada por um processo
- Interrupções de Temporizadores: O kernel deve ser capaz de transferir o controle de volta para o kernel de tempos em tempos. Hardware timer!

Operação em Modo Dual

Proteção à Memória



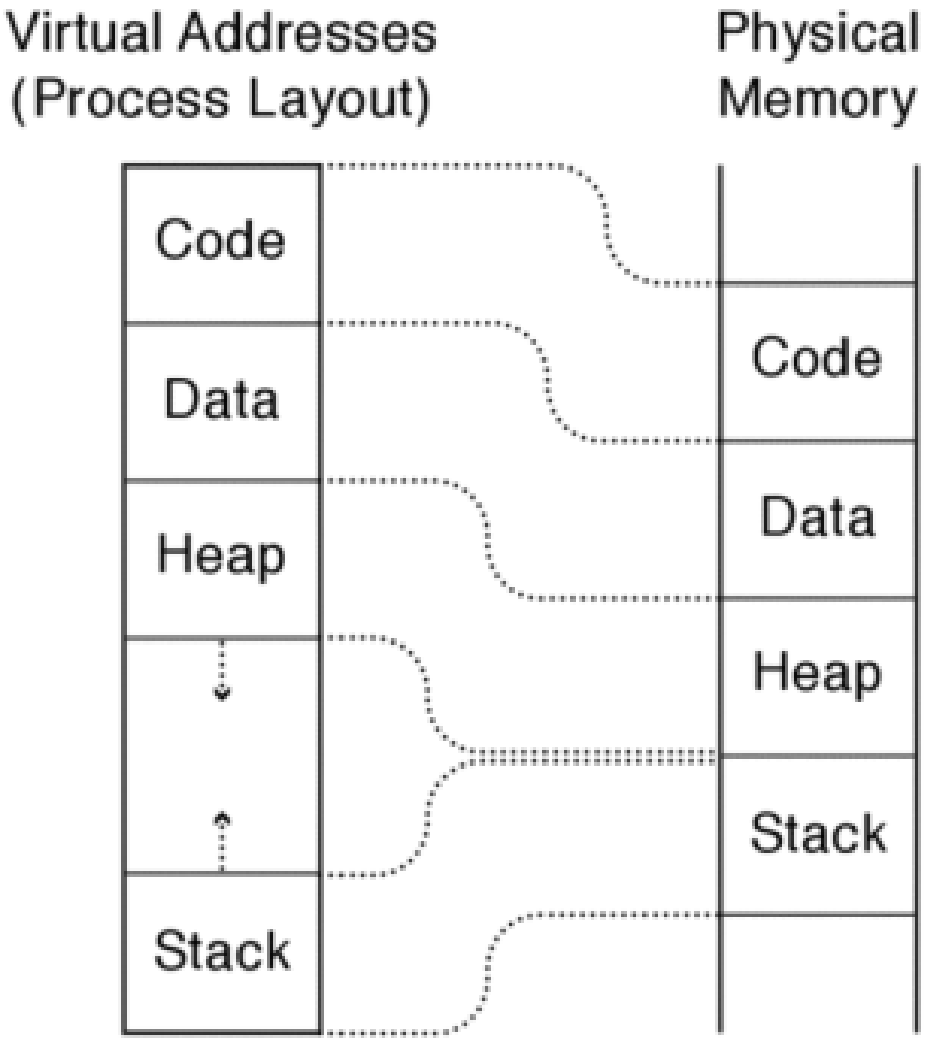
Operação em Modo Dual

Problema com endereços físicos

- O Stack e Heap
- Compartilhamento de Memória
- Endereços de mémoria física
- Fragmentação de Memória

Operação em Modo Dual

Endereços Virtúais



Tipos de transfência de modo

Usuário para kernel

Motivos para o kernel tomar o controle de um processo de usuário:

- Interrupção: temporizadores, I/O, interprocessadores, *polling*
- Exceção de processador: evento HW causado pelo programa de usuário. Ex., divisão por zero, out of range, etc.
- Chamadas de sistema: procedimento fornecido pelo kernel que pode ser chamado do nível de usuário. **instrução trap or syscall**

Tipos de transfência de modo

Kernel para usuário

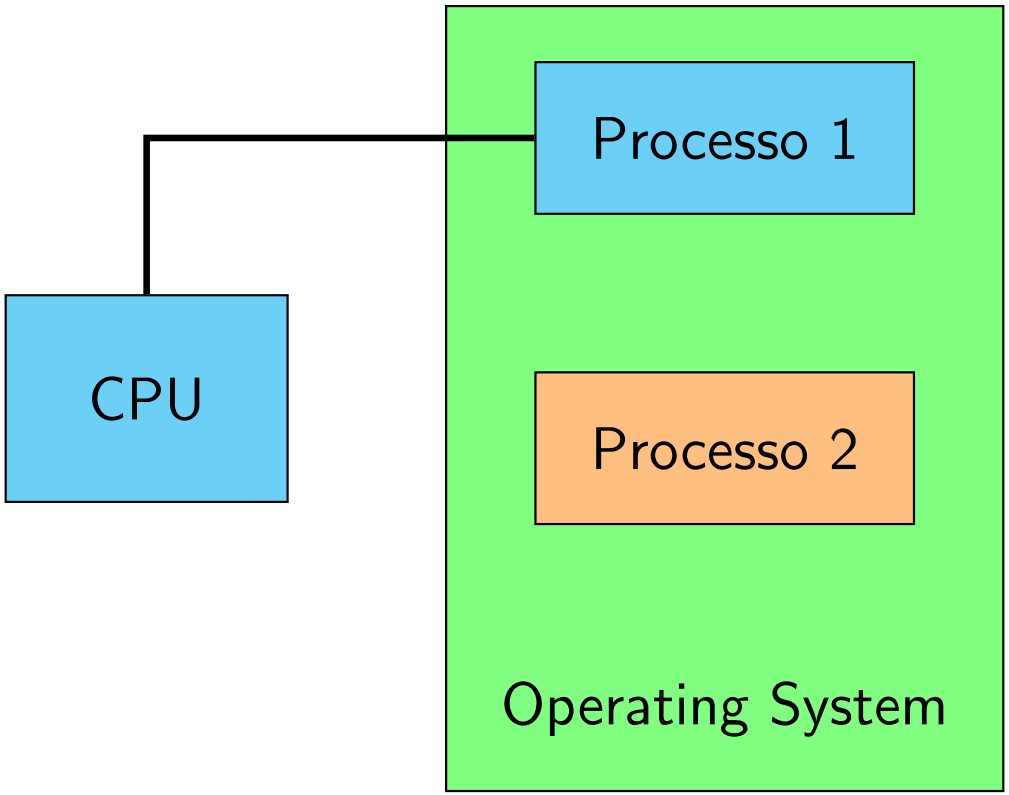
Motivos para o kernel tomar o controle de um processo de usuário:

- Novos processo
- Depois de uma interrupção, exceção, ou chamada de sistema
- Troca para um outro processo
- Upcall de nível de usuário (notificação asincrona de eventos)

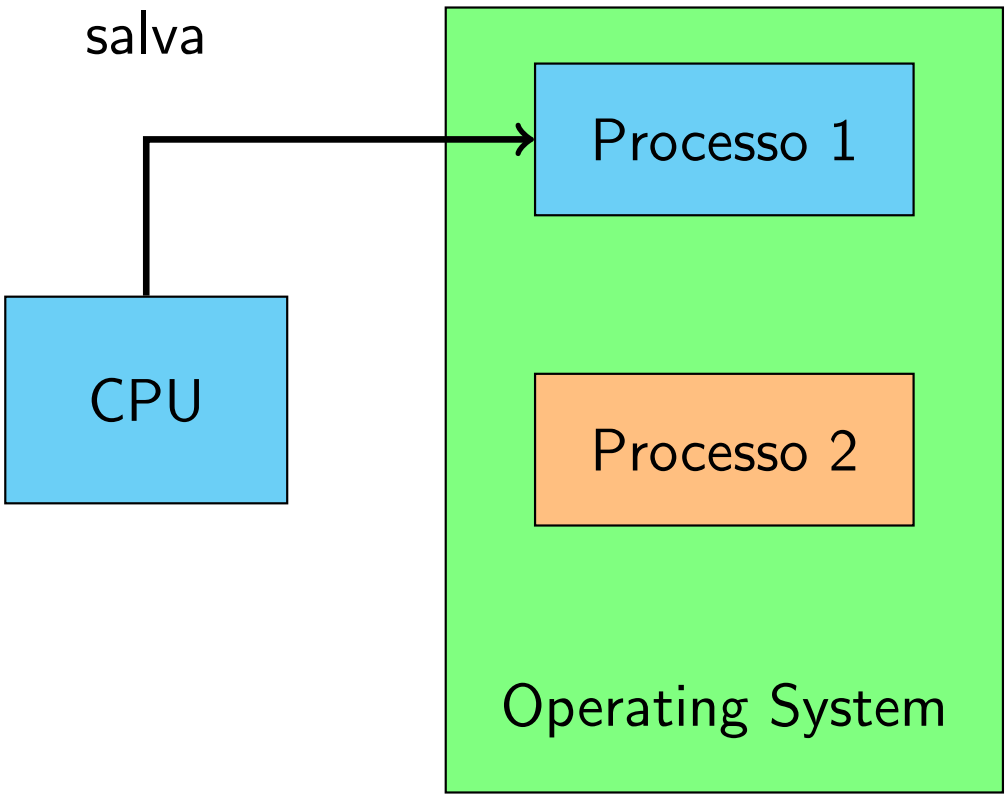
getpid()

- O processo do usuário está se executando
  - Processo do usuário executa `getpid()` (da biblioteca)
  - A biblioteca executa `TRAP $314159`
    - Nos Intel, `TRAP` é chamado `INT`, mas não é uma interrupção
- O mundo muda
  - Alguns registradores se baixam a memória em alguma parte
  - Alguns registradores se carregam na memória de alguma parte
- O processador há entrado ao modo de kernel ()

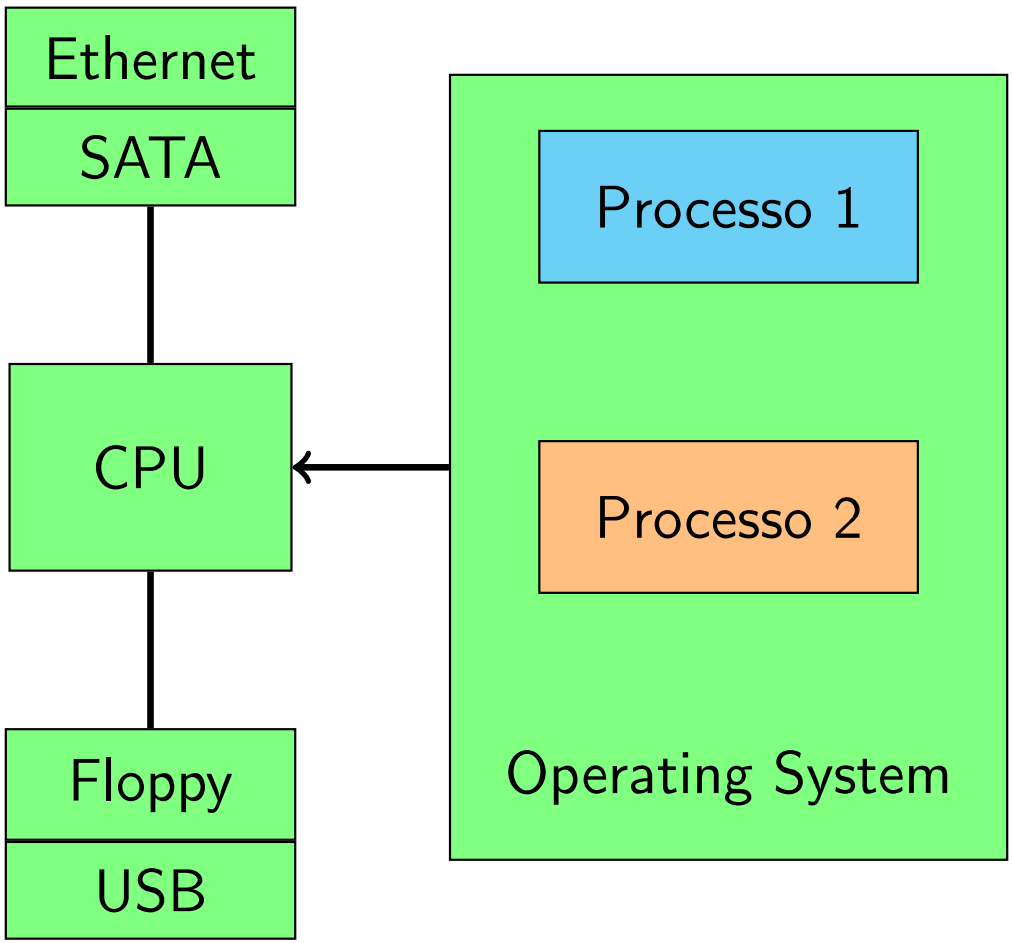
Modo de usuário



Entrando em modo de kernel



Entrando em modo kernel



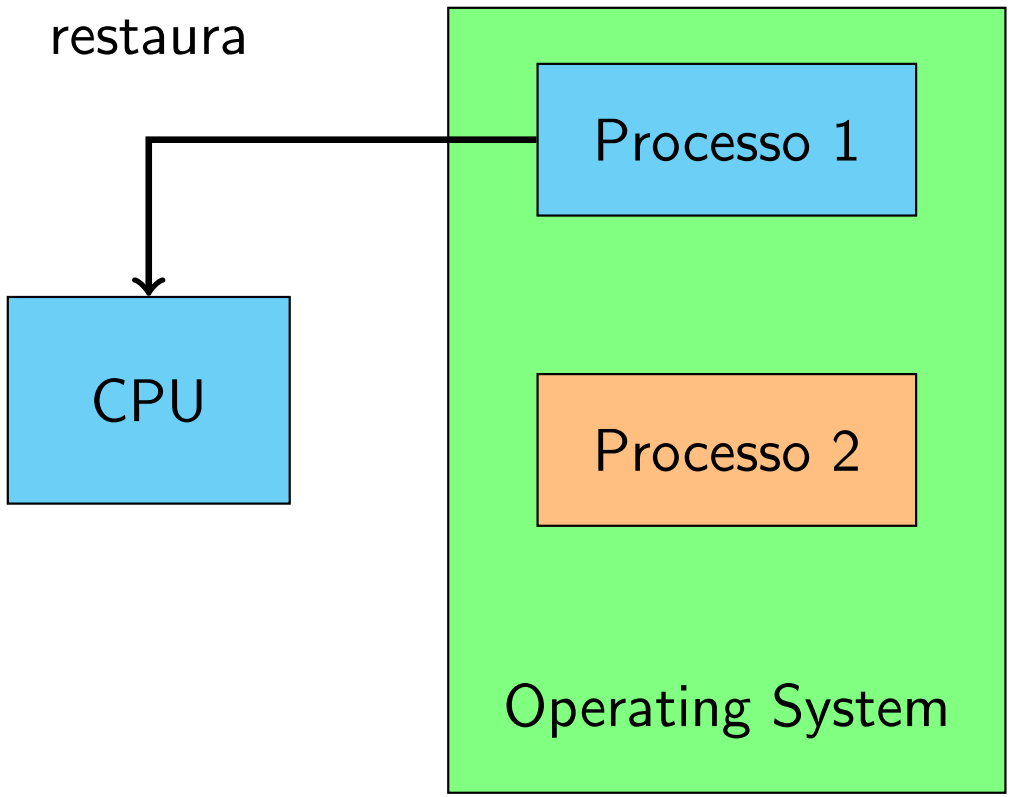
Ambiente de execução do kernel

- As linguagens de tempo de execução diferem
  - ML: pode não ter stack (só heap)
  - C: baseado num stack
- O processador é mais ou menos agnóstico
  - Alguns assumem ou requerem um stack
- O administrador de *trap* constrói o ambiente de execução do kernel
  - Dependendo do processador
    - Muda ao stack correto
    - Salva os registradores
    - Liga a memória virtual
    - Limpa os caches

A história de `getpid()`

- O processo se executa em modo de kernel
- Retorna da interrupção
  - O estado do processador é restaurado a modo de usuário
- O processo de usuário continua executando-se
  - A biblioteca retorna `%eax` como o valor de `getpid()`

Volta a modo de usuário



getpid()

- O que é a chamada do sistema `getpid()`?
  - Uma função de C que podemos chamar para obter o identificador do processo
  - Uma instrução simples que modifica `%eax`
  - Código privilegiado que pode acessar ao estado interno do sistema operacional
- Perguntas?

read()

- O processo de usuário está executando-se
  - `count = read(7, buf, sizeof(buf));`
- O processo de usuário para-se (pausa)
- O sistema operacional solicita uma leitura de disco
- O tempo passa
- O sistema operacional copia dados para o buffer do usuário
- O processo do usuário volta a executar-se

`read()`, mas com detalhes

P1: `read()`

- Trap para o modo de kernel

Kernel: disse-lhe ao disco *lê o setor 2781828*

Kernel: muda a executar P2

- Volta ao modo de usuário—mas executa P2, não P1
- P1 está bloqueado numa *chamada ao sistema*
  - O `%eip` de P1 esta em alguma parte do kernel
- Marcado como '*não pode executar mais instruções*'

P2: calcula 1/3 da mineração de um bitcoin

`read()`, mas com detalhes

Disco: termine de ler!

- Envia uma sinal de *pedido de interrupção*
- CPU para a execução de P2
- Interrompe para modo de kernel
- Executa o código do *gerenciador de interrupções de disco*

Kernel: muda a execução a P1

- Volta da interrupção—a P1, P2 continua parado
- P2 pode executar instruções, mas não faz
  - P2 não esta executando-se
  - Mas não esta bloqueado
  - Seu estado é *executável*, a diferença do P1 antes do disco terminar

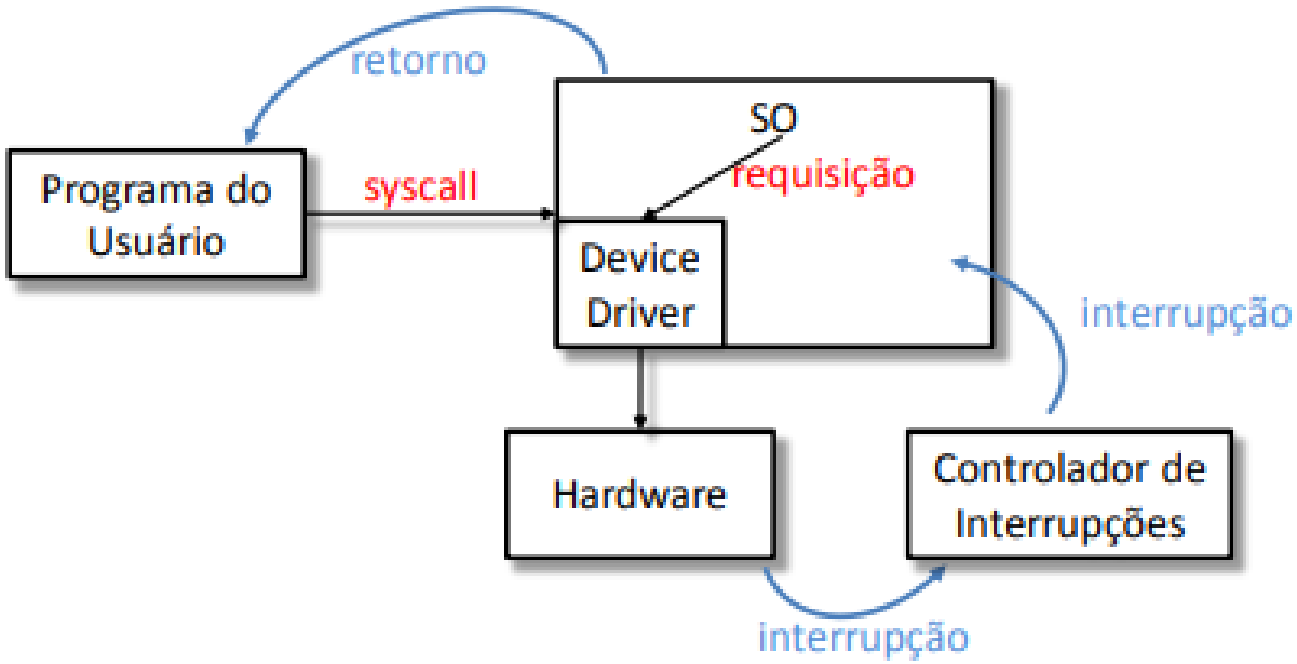
read()

- Qual é a diferença entre `read` e `getpid`?
- Por que?

? Dúvidas?

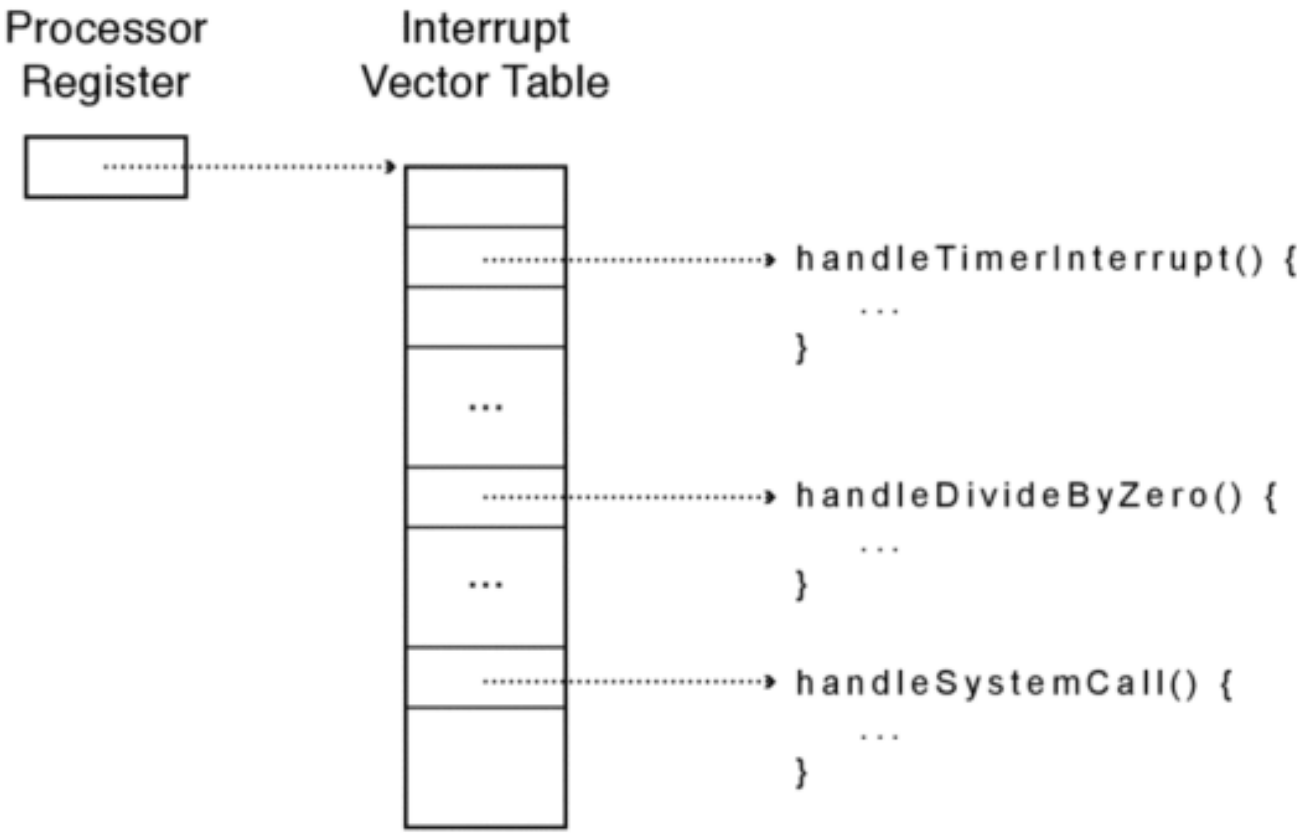
Interrupções

Procedimento geral interrupção



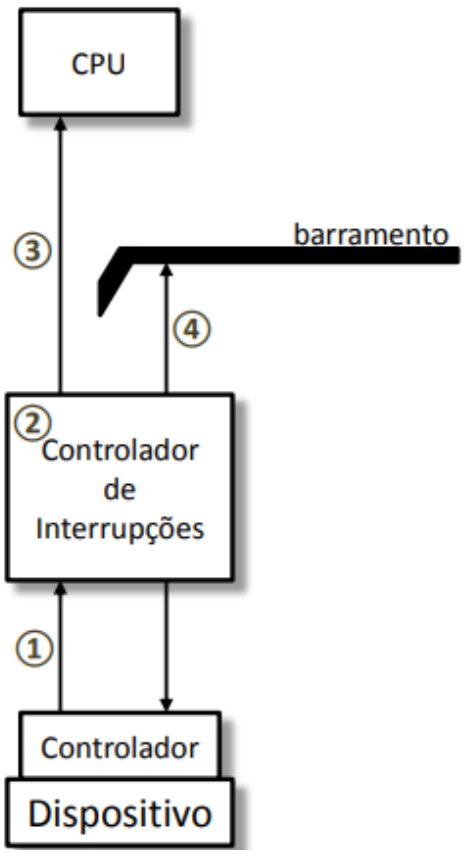
Interrupções

Tabela de vetores de interrupções



Interrupções

Interrupção HW



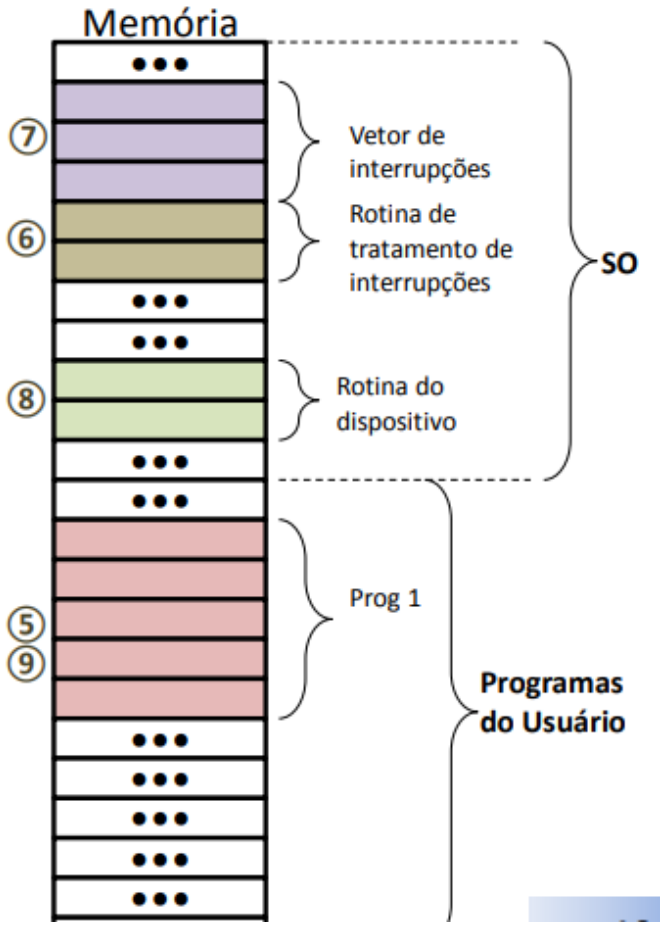


Tabela de vetores de interrupções

- Como sabe a CPU como gerenciar cada interrupção?
  - Interrupção de disco ⇒ invocar ao driver do disco
  - Interrupção do mouse ⇒ invocar ao driver do mouse
- Precisa conhecer
  - Onde armazenar os registradores
    - Frequentemente, propriedade do processo atual, não da interrupção
  - Carregar valores novos na CPU
    - Chave: novo contador de programa, novo registrador de estado
    - Estes definem o novo ambiente de execução

Envio de interrupções

- Lookup table
  - O controlador de interrupções diz: esta é a fonte da interrupção #3
  - CPU vá trazer a entrada #3 da tabela
    - A tabela baseada em ponteiros é construída quando o SO é inicializado
    - O tamanho da tabela define-se por HW
- Armazenar o estado do processador
- Modificar o estado do CPU de acordo à entrada da tabela
- Iniciar a execução do gerenciador da interrupção

Retorno da interrupção

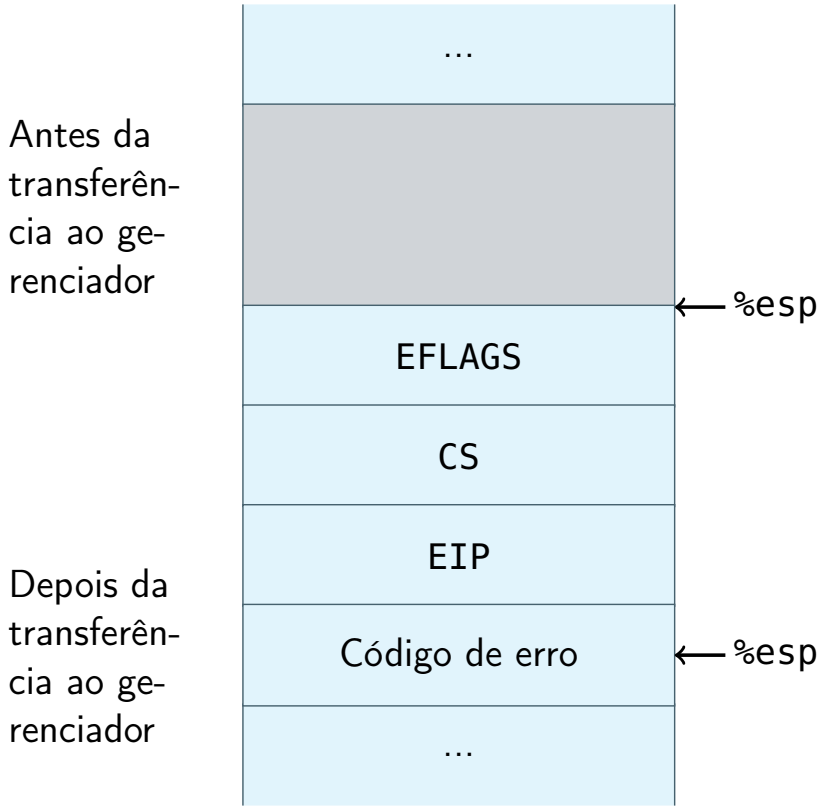
- Operação de retorno de uma interrupção
  - Carregar o estado do processador aos registradores
  - Restaurar o contador do programa reativa o código antigo
  - As instruções de hardware restauram alguma parte do estado
  - O kernel deve restaurar o resto

x86/IA32

- A CPU salva o estado do processador
  - Armazenado no stack do kernel
- A CPU modifica o estado de acordo à tabela de entradas
  - Carrega informação privilegiada, o contador do programa
- Inicia o gerenciador da interrupção
  - Usa o stack do kernel para suas operações
- Termina o gerenciador da interrupção
  - Vazia o stack a seu estado original
  - Invoca o retorno de interrupção (`iret`)
    - Os registradores se carregam do stack do kernel
    - O modo pode mudar de kernel a usuário
    - Pode que o código fique em modo de kernel

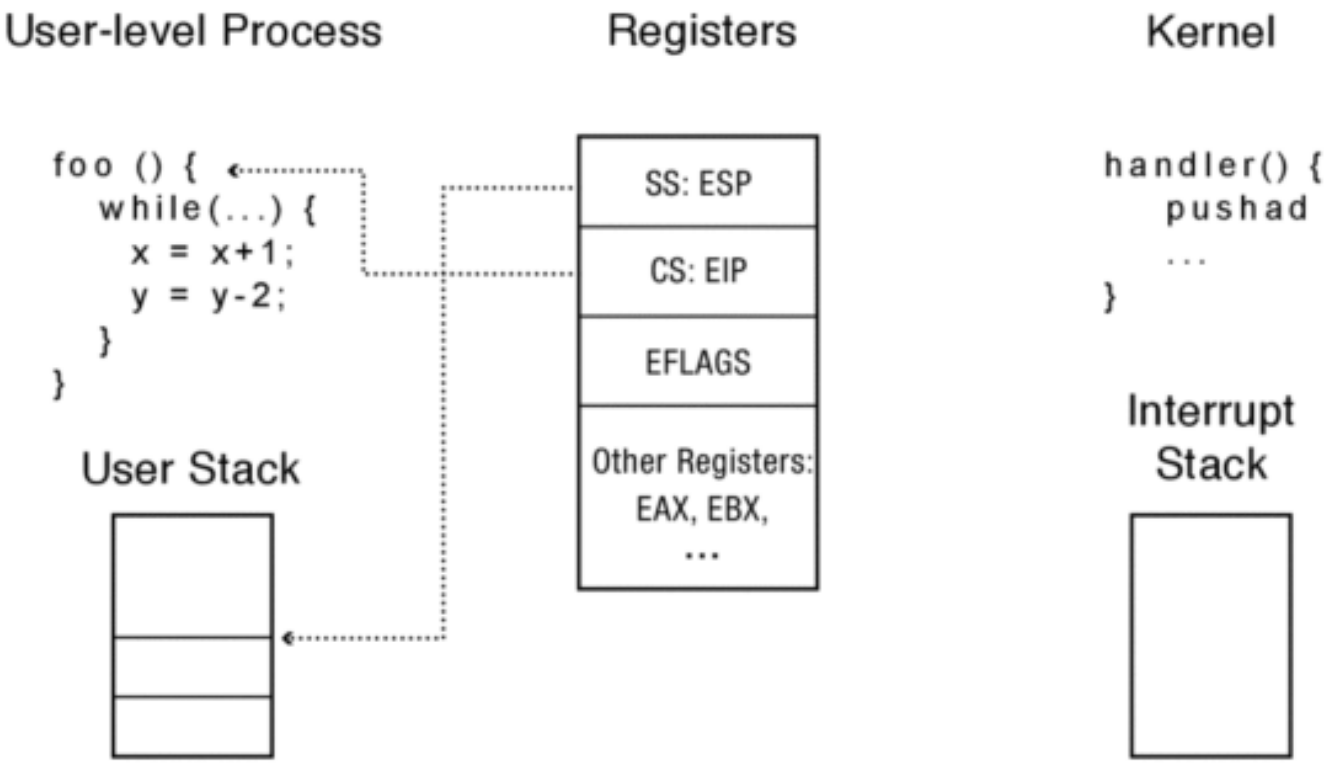
IA32 modo de tarefa única

- Hardware inserta os registradores no stack atual (não muda o stack)
  - `EFLAGS` (estado do processador)
  - `CS`/`EIP` (endereço de retorno)
  - Código de erro (algumas interrupções/erros, não todas: consultar arquitetura)
  - `iret` restaura o estado de `EIP`, `CS`, `EFLAGS`



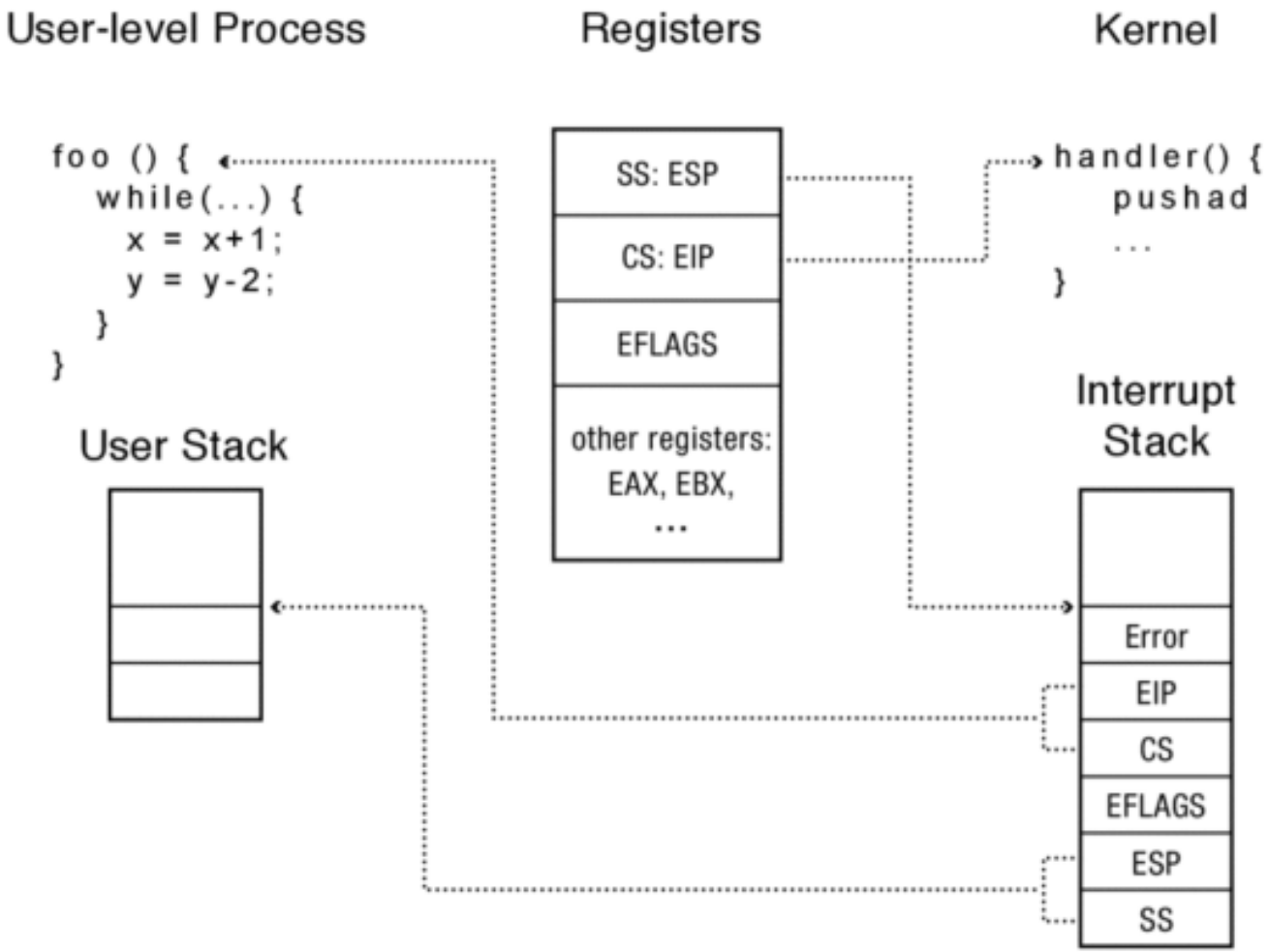
x86/IA32

Estado do sistema **antes** de um gerenciador de interrupção ser chamado



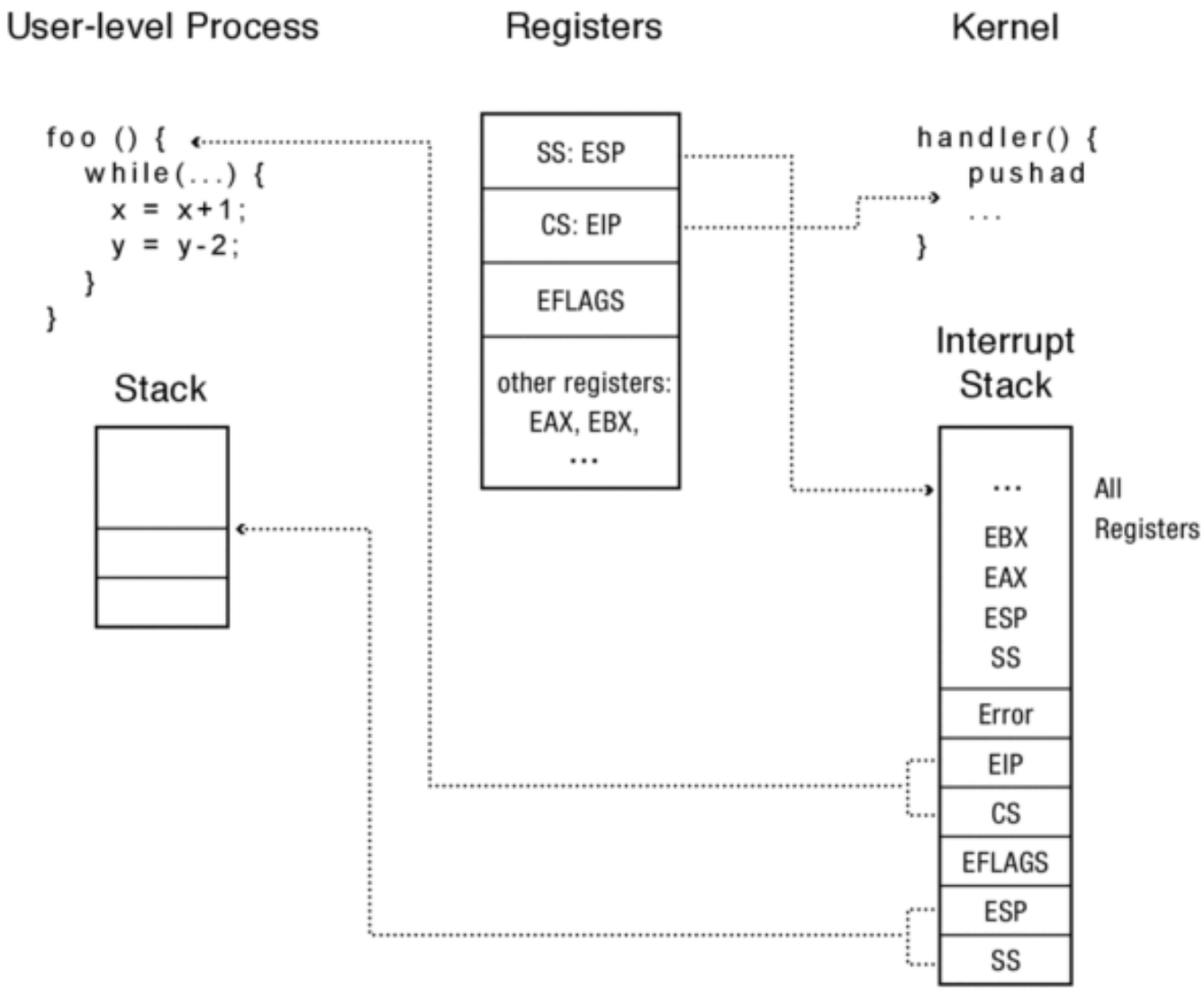
x86/IA32

Estado do sistema **depois** depois do HW pular para o gerenciador de interrupção



x86/IA32

Estado do sistema **depois** do gerenciador de interrupção ter começado execução



Condições de corrida

- Duas atividades concorrentes
  - Programa de computador, disco
- Várias sequencias de execução produzem varias respostas
  - O disco interrompe antes ou depois da chamada?
- A sequencia de execução não está controlada
  - Qualquer resultado é possível aleatoriamente
- O sistema produz respostas aleatórias
  - Uma resposta ou outra ganha a corrida

Condições de corrida

Driver do disco

- Uma parte quer lançar requisições de I/O ao disco
  - Se o disco esta desocupado, envia o pedido
  - Se o disco esta ocupado, enfileira o pedido para depois
- A ação do gerenciador da interrupção depende do estado da fila
  - Trabalho na fila $\Rightarrow$ enviar o seguinte pedido ao disco
  - Fila vazia $\Rightarrow$ deixa o disco ocioso
- Vários possíveis ordenes de execução
  - A interrupção de disco antes ou depois da prova "*disco esta desocupado*"
- O sistema produz respostas aleatórias
  - Trabalho na fila, então transmitamos ao seguinte pedido (bem)
  - Trabalho na fila, deixemos descansar o disco (*say what!?*)

Condições de corrida

Esqueleto do driver

```
dev_start (request) {
  if (device_idle) {
    device_idle = 0;
    send_device (request);
  } else {
    enqueue(request);
  }
}

dev_intr () {
  // finish up previous request code
  if (new_request = head()) {
    send_device(new_request);
  } else {
    device_idle = 1;
  }
}
```

Caso Bom

Processo de usuário	Gerenciador da interrupção
`if (device_idle)`	
não, então...	
`enqueue(request)`	
	Interrupção
	fazemos trabalho
	`new = 0x80102044;`
	`send_device(new);`
	Retornamos da interrupção

Caso ruim

Processo de usuário	Gerenciador da interrupção
`if (device_idle)`	
não, então...	
	Interrupção
	fazemos trabalho
	`new = 0;`
	`device_idle=1;`
	Retornamos da interrupção
`enqueue(request)`	

O que falhou?

- Executa o algoritmo
  - Examina o estado
  - Faz uma ação (segundo o estado)
- O gerenciador da interrupção executa seu algoritmo
  - Examina o estado
  - Realiza uma ação (segundo o estado)
- Vários possíveis estados de término
  - Dependem de quando se executa o código do gerenciador da interrupção
- O sistema produz varias saídas "aleatórias"
  - Estudem a condição (e evitem este problema em seus projetos)

O que podemos fazer?

- Duas soluções
  - Temporariamente suspender/emascarar/diferir as interrupções dos dispositivos enquanto se verifica e enfileira
  - Utilizar uma estrutura de dados que seja livre de bloqueio
- Considerar
  - Evitar bloquear todas as interrupções
  - Evitar bloquear por muito tempo

Temporizador (timer)

- Comportamento simples
  - Conta algo: ciclos de CPU, ciclos de bus, microssegundos
  - Quando chega ao limite, emite uma interrupção
  - Reestabelece o contador ao valor inicial
    - O faz a nível de HW, no fundo, não precisa esperar ao SW para reestabelecer-se
- Resumo
  - Não há pedidos, não há resultados
  - Um fluxo constante de eventos distribuídos equitativamente no tempo

Por que precisamos um timer?

- Por que parar uma execução perfeita?
- Evitamos que os aplicativos acaparem o processador `while(1) continue;`
- Mantemos a hora do dia
  - Calendário garantido por bateria conta só os segundos (não muito corretamente)
- Interrupção de duplo propósito
  - Mantém o tempo: `++ticks_since_boot;`
  - Evita monopolização do CPU: força a troca de processos

Pontos chave

- Uma abstração do hardware (detalhes em Arquitetura de Computadores)
- Modos de execução: kernel e de usuário
  - Memoria virtual
  - Código privilegiado
- Exemplos de chamadas ao sistema
- Interrupções
  - Armadilhas (*trap*): síncronas
    - Excepções: erros em código, programas malignos, ou benignos (debugger)
    - Chamadas ao sistema
  - Interrupções: assíncronas
    - Interrupções por tempo
    - Dispositivos
- Condições de corrida
- Emascarado de las interrupções