Implementação de Tarefas em Kernel Completo

Fundamentos dos Sistemas de Tempo Real

Rômulo Silva de Oliveira

Edição do Autor, 2018

www.romulosilvadeoliveira.eng.br/livrotemporeal

Outubro/2018

Introdução

- Cada vez mais aplicações complexas são incluídas em equipamentos e veículos
- Aplicações que incluem um grande número de tarefas de tempo real

- Muitas aplicações de tempo real, mesmo em equipamentos, precisam
  - Interface gráfica de usuário
  - Sistema de arquivos
  - Pilhas de protocolos de comunicação
  - Controle de acesso
  - Serviços providos por um sistema operacional

- Aplicações deste tipo requerem um kernel completo como sistema operacional multitarefa
  - Exemplo clássico neste caso é o Linux

O Sistema Operacional Tradicional 1/8

- O sistema operacional atua como um mediador entre as tarefas da aplicação e os recursos do sistema
  - Especialmente os periféricos
- Torna o uso do computador mais conveniente e mais eficiente

- O sistema operacional torna mais simples o trabalho ao esconder as complexidades do hardware
  - Tanto de usuários finais como dos programadores das aplicações

- Ele cria abstrações mais convenientes do que os recursos do hardware
  - Arquivos e diretórios são abstrações criadas a partir de espaço em disco
  - Processos e threads são criados a partir da divisão do tempo do processador
  - Portas de comunicação são criadas a partir do controlador ethernet
  - Espaços de endereçamento são criados a partir dos circuitos de memória
  - Etc

O Sistema Operacional Tradicional 2/8

- O desenvolvimento é facilitado
- O acesso aos periféricos não é mais feito acessando diretamente os registradores dos controladores de periféricos
  - Mas solicitando este acesso ao kernel
  - Que se encarrega dos detalhes operacionais do periférico em questão
- O código da aplicação pode solicitar ao kernel o envio de um pacote de dados pela porta serial
  - Sem precisar detalhar a série de comandos necessários
- Os comandos detalhados são enviados pelo kernel
  - Módulo do kernel de device-driver da porta serial

O Sistema Operacional Tradicional 3/8

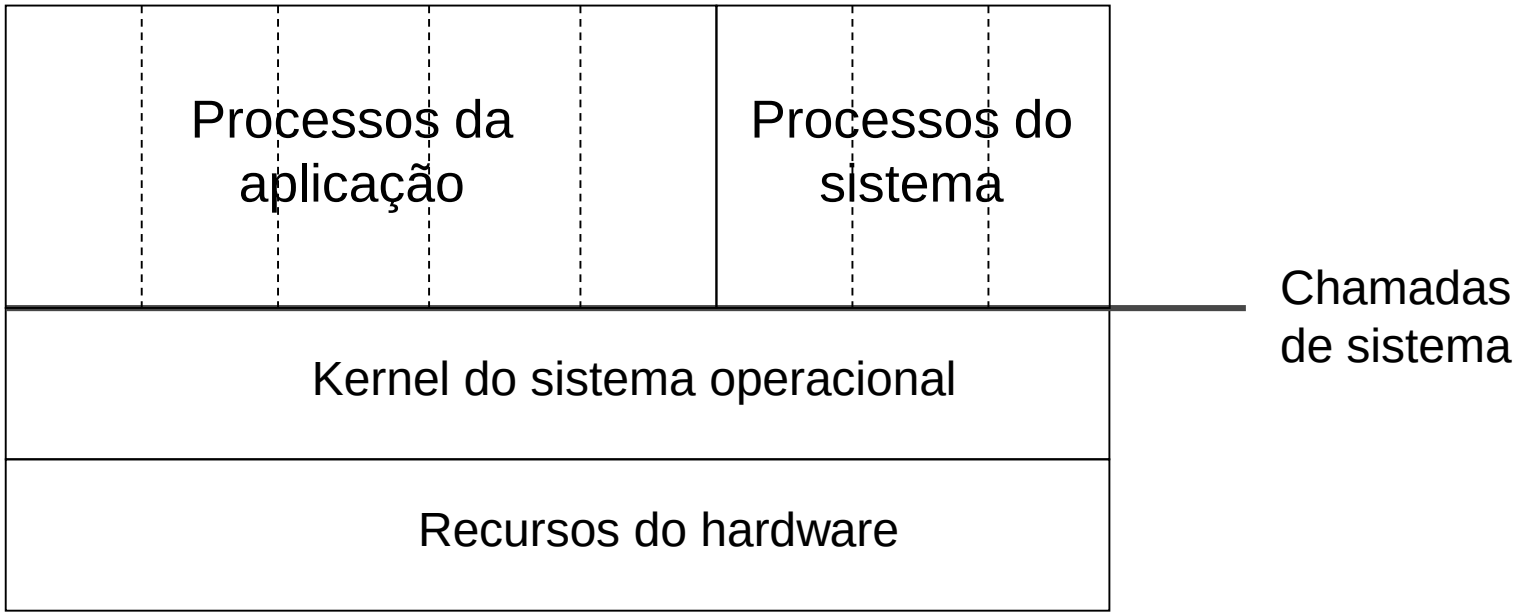
- O sistema operacional fornece uma gama de serviços
- Pode ser mais ou menos extensa, dependendo do seu tamanho
- Serviços típicos:
  - Execução de programas
  - Gerência do processador
  - Gerência de memória sofisticada
  - Sistemas de arquivos
  - Gerência de periféricos
  - Proteção entre processos
  - Pilhas de protocolo de comunicação
  - Controle de acesso
  - Contabilizações

O Sistema Operacional Tradicional 4/8

- O componente mais importante de um sistema operacional é o seu **kernel**
  - Núcleo ou miolo em português, mas o termo kernel está estabelecido
- Trata-se da camada de software que controla diretamente os recursos do hardware
- Usualmente os sistemas de arquivos, pilhas de protocolos de comunicação e device-drivers são implementados dentro do kernel
- Acima do kernel executa o código da aplicação
  - Também podem ser executadas atividades do sistema
  - Servidores de impressão ou interface com usuário humano
  - São chamados de programas de sistema (system programs)
- Quando o kernel oferece apenas serviços básicos de gerência do processador ele é chamado de microkernel

O Sistema Operacional Tradicional 5/8

- A **multiprogramação (*multiprogramming*)** é usada para realizar a execução aparentemente simultânea de vários programas em um único processador
  - A partir da divisão do tempo do processador e de outros recursos



O Sistema Operacional Tradicional 6/8

- O emprego de multiprogramação torna o uso do computador muito mais eficiente

- Considere que o programa em execução precise ler informações de um arquivo as quais estão em um certo setor do disco magnético
  - O tempo aproximado para ler um setor do disco fica em torno de 10ms

- Suponha que neste computador a frequência do clock seja 2GHz
  - Neste caso o período do clock é de 0,5ns
  - Se uma instrução de máquina demora 4 clocks em média para executar, teremos 2ns

- Durante um acesso ao disco é possível executar 5 milhões de instruções de máquina
- Graças à multiprogramação, enquanto um programa espera pelo acesso ao disco, outro programa utiliza o processador

O Sistema Operacional Tradicional 7/8

- Podemos entender um **processo (*process*)** como uma abstração criada pelo kernel
- A qual possui uma série de atributos
  - Espaço de endereçamento, descritores de arquivos abertos, permissões de acesso, contabilizações, etc
- Todo processo também possui pelo menos um fluxo de execução
  - Este fluxo de execução é chamado de thread (thread of execution)
  - Composta basicamente pelo conteúdo dos registradores do processador
- Nos sistemas operacionais mais antigos, processos possuíam apenas uma thread
- Sistemas operacionais modernos: um processo pode possuir várias threads
- Na verdade threads existem dentro dos processos, usando a memória, os arquivos e demais recursos do seu processo
  - Todas as threads de um processo compartilham os recursos deste processo

O Sistema Operacional Tradicional 8/8

- Existem bibliotecas que executam fora do kernel e são capazes de implementar o conceito de
  thread a nível de usuário (user-level thread)
  - Implementam threads no programa de aplicação sem o conhecimento do kernel

- Para o kernel trata-se de um processo normal com apenas uma thread
  - Mas os recursos são na verdade divididos entre threads
  - Que somente existem a nível de código de usuário

- Neste livro iremos considerar apenas as chamadas
  threads a nível de kernel (kernel-level threads)
  - Threads implementadas e reconhecidas pelo kernel do sistema operacional

Terminologia: Processos, Threads e Tarefas 1/3

- **Processo** é uma abstração criada pelo kernel a qual possui um conjunto de atributos
  - Espaço de endereçamento, descritores de arquivos abertos, direitos de acesso, contabilizações, etc
- Todo processo possui obrigatoriamente pelo menos uma thread
  - Podendo possuir mais de uma thread caso o kernel permita
- Um pequeno microkernel tipicamente implementa apenas threads, isto é, todos os fluxos de execução da aplicação são threads que compartilham todos os recursos do sistema
  - Não existe sentido em pensar em processos
- Um kernel antigo não implementa threads explicitamente, apenas processos
  - Cada processo contem um único fluxo de execução (uma única thread implícita)
- Um kernel moderno implementa processos os quais podem incluir uma ou mais threads
  - Dentro do kernel também existem threads que executam exclusivamente código do kernel

Terminologia: Processos, Threads e Tarefas 2/3

- Na literatura em geral existe uma certa confusão sobre como são empregados os termos processos, threads e tarefas (tasks)

- O termo **processo** será usado para denotar uma abstração criada pelo kernel a partir da gerência dos recursos do sistema

- O termo **thread** representa um fluxo de execução, caracterizado principalmente pelo conteúdo dos registradores a cada momento
  - Todo processo tem pelo menos uma thread

- Uma **tarefa** corresponde à execução de um trecho de código que deve respeitar certos requisitos temporais
  - Como é implementada depende do sistema operacional em questão

Terminologia: Processos, Threads e Tarefas 3/3

- Na aplicação de tempo real uma tarefa pode aparecer como um conjunto de funções, uma única função, ou mesmo uma sequência de linhas de código
  - A definição do que é uma tarefa é feita no escopo da aplicação

- Por exemplo, ler o sensor de temperatura e colocar este valor na tela pode ser uma tarefa da aplicação

- Receber uma mensagem pela rede de comunicação e fechar a válvula de combustível pode ser outra tarefa

- Muitas vezes a tarefa da aplicação de tempo real é implementada como um processo ou thread no âmbito do sistema operacional
- Por vezes uma tarefa da aplicação de tempo real é implementada diretamente como um tratador de interrupção

Chamadas de Sistema 1/3

- Processos solicitam serviços ao kernel através de **chamadas de sistema (*system calls*)**
- Chamadas de sistema são implementadas como interrupções de software cujo tratador faz parte do kernel
- Um processo desejando ler um arquivo deve colocar em um registrador específico o código numérico da operação e gerar a interrupção de software que aciona o kernel
- Programas escritos em linguagens de alto nível usam as bibliotecas da linguagem de programação
  - As funções da biblioteca que fazem as chamadas de sistema
- Caso um processo possua 5 threads
  - Possível que cada uma delas faça uma chamada de sistema
  - Este processo teria então 5 chamadas de sistema sendo atendidas simultaneamente

Chamadas de Sistema 2/3

- Idealmente, o kernel nunca deveria executar
  - O computador foi comprado para executar código de aplicação
- As vezes um processo da aplicação requer um serviço do kernel
  - Faz uma chamada de sistema
  - O kernel entra em execução, atendendo esta chamada de sistema
- Se a chamada de sistema pode ser atendida imediatamente
  - O processo da aplicação retoma sua execução
- Se não é possível (uma leitura de teclado, a espera por um pacote da rede ethernet, ou mesmo uma chamada do tipo “sleep”)
  - O kernel comanda o controlador de periférico apropriado
  - O processo da aplicação precisará aguardar
- Quando a thread de um processo não pode continuar a executar, pois está esperando por algum evento, o kernel coloca outra thread para executar
  - Do mesmo processo ou de outro processo

Chamadas de Sistema 3/3

- Quando no futuro o evento esperado pela chamada de sistema acontecer
  - O controlador do periférico em questão gera uma interrupção e aciona o kernel
  - O kernel volta a executar
  - Faz o tratamento dos dados que for necessário
  - A thread que fez a chamada de sistema pode continuar sua execução

- Esta forma básica de kernel comporta-se como um sistema reativo (reactive system)

- Todo kernel é construído com este comportamento reativo em mente

- Entretanto, kernels de grande porte como Linux ou Windows precisam de mecanismos mais sofisticados
  - Para lidar com o paralelismo e a complexidade dos muitos serviços oferecidos

Estados da Thread de um Processo 1/5

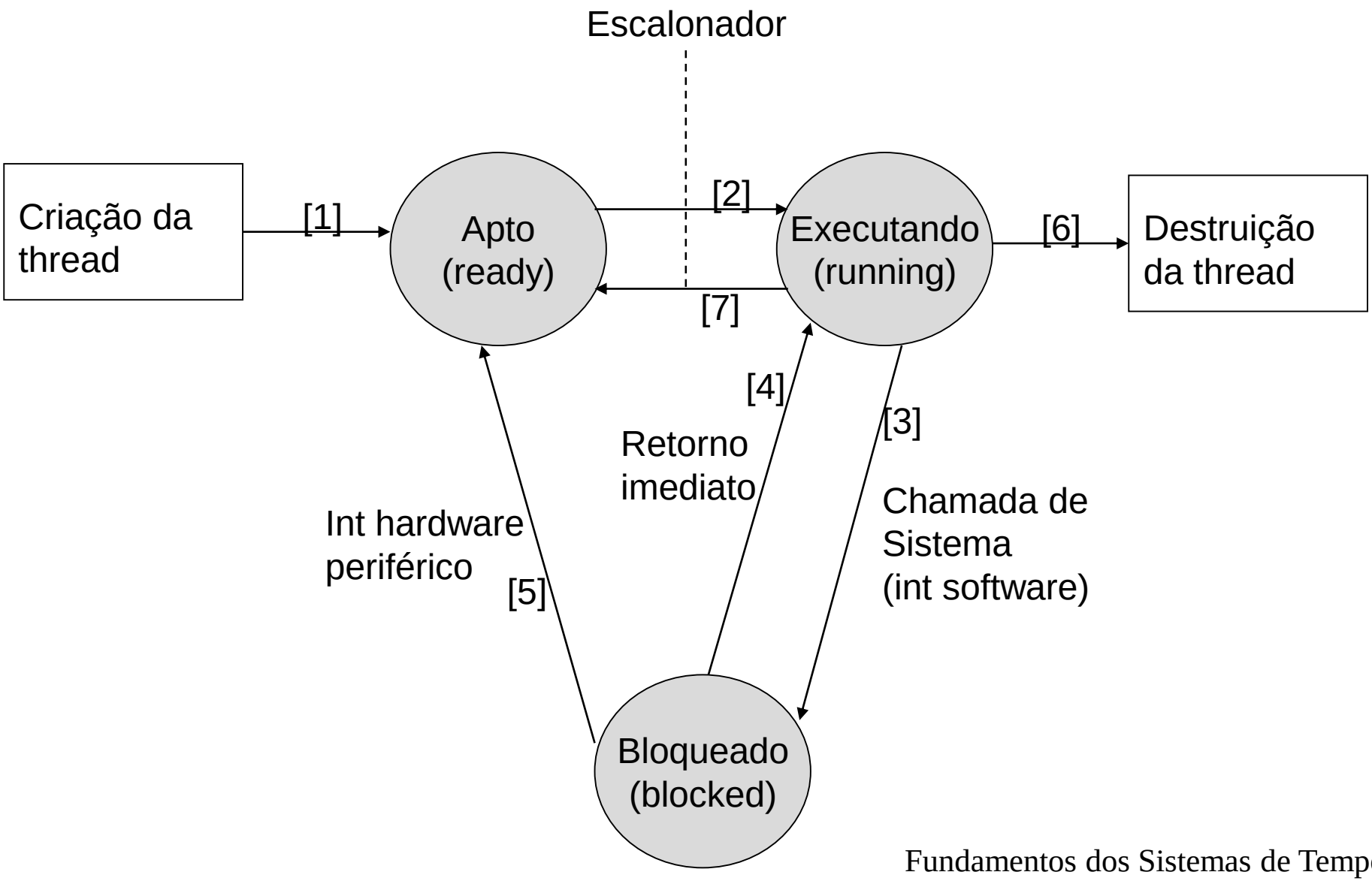
- Tanto processos como threads podem ser criados e destruídos dinamicamente
  - Através de chamadas de sistema realizadas por processos e threads já existentes

- As threads dos processos passam por diversos estados ao longo de sua existência, da mesma forma que as threads em um microkernel

- Os estados apresentados referem-se ao fluxo de execução (control flow)
  - Característica da thread em si, e não do processo como um todo

- Podemos falar em estados das threads em vez de estados dos processos
- Se um processo tem 3 threads
  - Perfeitamente possível que cada thread esteja em um estado diferente

Estados da Thread de um Processo 2/5



Estados da Thread de um Processo 3/5

- Na multiprogramação diversos processos compartilham o computador
- É desejável que o sistema operacional seja capaz de evitar que um processo com problemas comprometa todo o sistema

- Mesmo quando o computador está embutido em um sistema maior (embedded system), como um equipamento industrial
  - A detecção de problemas no software pode permitir um desligamento controlado do equipamento, evitando prejuízos maiores

- O kernel do sistema operacional implementa a proteção entre processos através de recursos específicos da arquitetura do processador (hardware)

Estados da Thread de um Processo 4/5

- A forma usual é definir dois modos de operação para o processador: **modo usuário (*user mode*)** e **modo supervisor (*supervisor mode*)**

- O kernel executa em modo supervisor
  - Não existem restrições e qualquer instrução pode ser executada

- Processos da aplicação executam em modo usuário
  - Algumas instruções não podem ser executadas

- Essas instruções são chamadas de instruções privilegiadas
- Caso uma thread da aplicação tente executar uma instrução privilegiada em modo usuário:
  - O hardware automaticamente gera uma interrupção de proteção
  - Chamada de exceção (exception)

- O kernel será acionado e poderá, por exemplo, abortar esta thread
  - Ou abortar o seu processo, ou tomar outra ação apropriada para a violação detectada

Estados da Thread de um Processo 5/5

- Nem todos os processadores incluem o hardware necessário para a implementação da proteção entre processos

- Para que a proteção entre processos seja completa também é necessário proteger a memória
- Caso uma thread de um processo tente acessar uma memória que não foi alocada para o seu processo, novamente uma interrupção de proteção será gerada
  - Kernel é acionado
  - As medidas necessárias poderão ser tomadas

Chaveamento de Contexto 1/4

- A operação de suspender a thread em execução e colocar uma outra thread (talvez de outro processo) para executar é chamada de **troca de contexto** (***context switch***)
- Na multiprogramação isto acontece o tempo todo
- Quando a thread de um processo tem sua execução suspensa é necessário salvar o seu contexto de execução.
- Para que uma thread de qualquer processo possa passar a executar, ocupando o processador, é necessário recarregar o seu contexto
- A troca de contexto é sempre uma operação delicada
- O conteúdo de todos os registradores precisa ser salvo
- O código do kernel precisa usar registradores para realizar a operação

Chaveamento de Contexto 2/4

- Kernel mantém uma estrutura de dados chamada de **Bloco Descritor de Processo** (PCB – Process Control Block)
  - Informações relativas ao contexto de execução de cada processo
- Também contem informações sobre
  - Segmentos de memória ou endereço de tabelas de páginas
  - Número identificador do processo (PID - Process Identification)
  - Prioridade do processo
  - Usuário associado com este processo
  - Indicação dos arquivos abertos pelo processo
  - Contabilizações de uso de recursos
  - Direitos especiais de acesso aos recursos do sistema
  - Quaisquer outros dados que o kernel possa precisar sobre cada processo

Chaveamento de Contexto 3/4

- No caso de processos com múltiplas threads, torna-se necessário o emprego de um
  Bloco Descritor de Thread (TCB - Thread Control Block)
- As informações específicas de cada thread são mantidas no seu respectivo TCB
  - Basicamente conteúdos dos registradores quando a thread não estiver executando, além de um apontador para o PCB de seu processo
- Como cada thread faz parte de um processo, todas as demais informações são obtidas no PCB do seu processo
- PCB não inclui campos para salvar o conteúdo dos registradores
  - Inclui uma série de apontadores para os TCBs de suas threads
- Como a entidade a ser escalonada no uso do processador é a thread e não o processo, a fila de aptos passa a ser formada por TCBs

Chaveamento de Contexto 4/4

- Threads podem ser usadas tanto em código da aplicação no espaço do usuário
- Como também para organizar o código do próprio kernel
- Threads internas do kernel não fazem parte de um processo
  - Não estão associadas com nenhum PCB
  - São tratadas de forma diferenciada
- Em geral podem acessar toda a memória do computador
  - Possuem amplos direitos de acesso
  - São threads que sempre executam código do kernel

Gerência de Memória 1/5

- Em um kernel de sistema operacional que implementa proteção entre processos, para que a proteção entre eles seja completa também é necessário proteger a memória

- É necessário um componente específico de hardware junto ao processador para verificar a validade de cada acesso

- Dependendo da complexidade e dos serviços providos por tal componente, diferentes gerências de memória são empregadas

- Partições variáveis usando uma Unidade de Proteção de Memória
- Paginação usando uma Unidade de Gerência de Memória

Gerência de Memória 2/5

- É útil fazer distinção entre memória física e memória lógica
- A **memória física (*physical memory*)** é aquela usualmente chamada de RAM ou memória principal (main memory)
  - Implementada pelos circuitos integrados (CIs) de memória
- Acessada através de endereços físicos (physical addresses) de memória
  - Indicam qual posição nos CIs de memória será lida ou escrita a cada momento

- O **espaço de endereçamento físico (*physical address space*)** é composto pelos endereços de memória física válidos no computador em questão
  - Para os quais existem posições associadas nos CIs de memória

Gerência de Memória 3/5

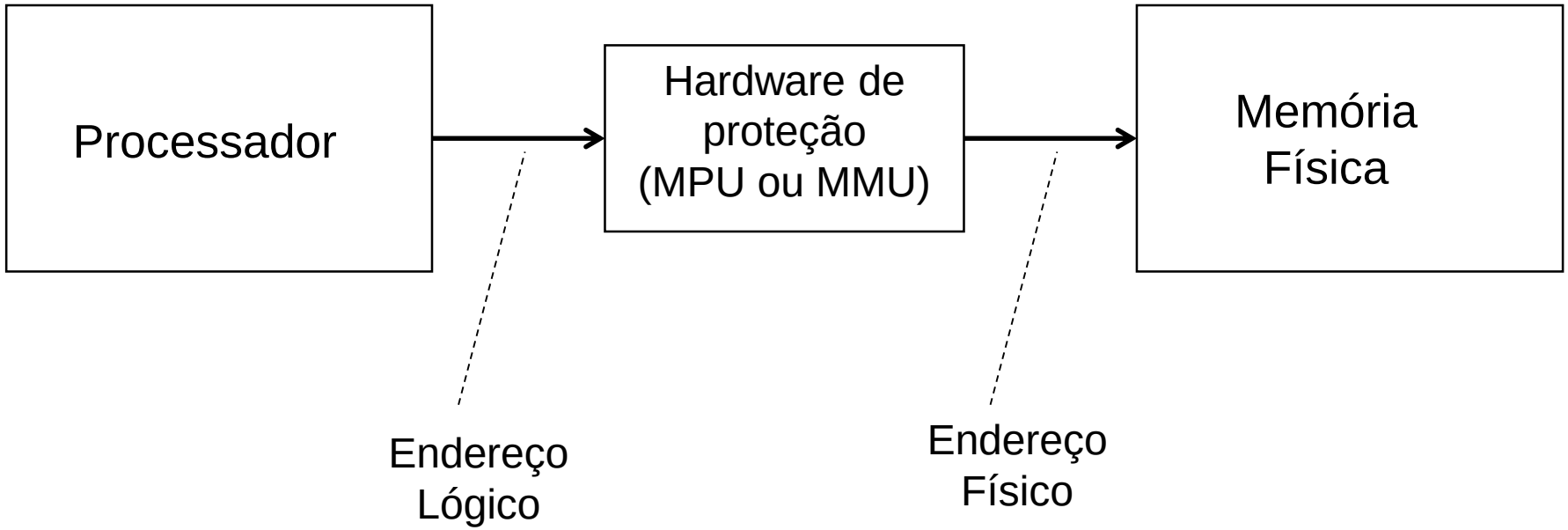
- A **memória lógica** (***logical memory***) de um processo é aquela acessada pelas threads do processo em questão
- O computador possui apenas uma memória física
  porém cada processo possui a sua memória lógica
- Os endereços manipulados pelos programas são portanto
  endereços lógicos (***logical addresses***)
  - Fazem referência à memória lógica do processo em questão
- Todas as threads de um processo compartilham sua memória lógica
- O **espaço de endereçamento lógico** (***logical address space***) é o conjunto de endereços lógicos a disposição do processo
  - Permitem o acesso a sua memória lógica

Gerência de Memória 4/5

- O objetivo da **proteção de memória** é exatamente impedir que um processo acesse a memória lógica de outro processo
- Para isto é utilizado um componente específico de hardware que fica entre o processador e os CIs de memória
- Tipicamente o processador e o componente de proteção ficam no mesmo CI, e a memória física em CIs separados

- A complexidade do hardware de proteção de memória varia muito
  - Pode não existir em um microcontrolador simples
  - Prover suporte para vários tipos de gerência de memória sofisticada em processadores mais avançados

Gerência de Memória 5/5



Partições Variáveis 1/8

- Uma forma muito comum de gerência de memória em pequenos computadores é através de **partições variáveis** (*variable partitioning*)
- A memória física é dividida em **partições** (*memory partitions*)
  - Tamanhos são definidos conforme as necessidades dos processos
- Tipicamente a memória lógica de um processo corresponde a uma área contígua de memória
  - Associada com uma partição específica da memória física
- Em alguns sistemas um processo pode ter sua memória lógica dividida em várias algumas áreas contíguas
  - Número pequeno de partições
  - Por exemplo: código, pilha e variáveis globais

Partições Variáveis 2/8

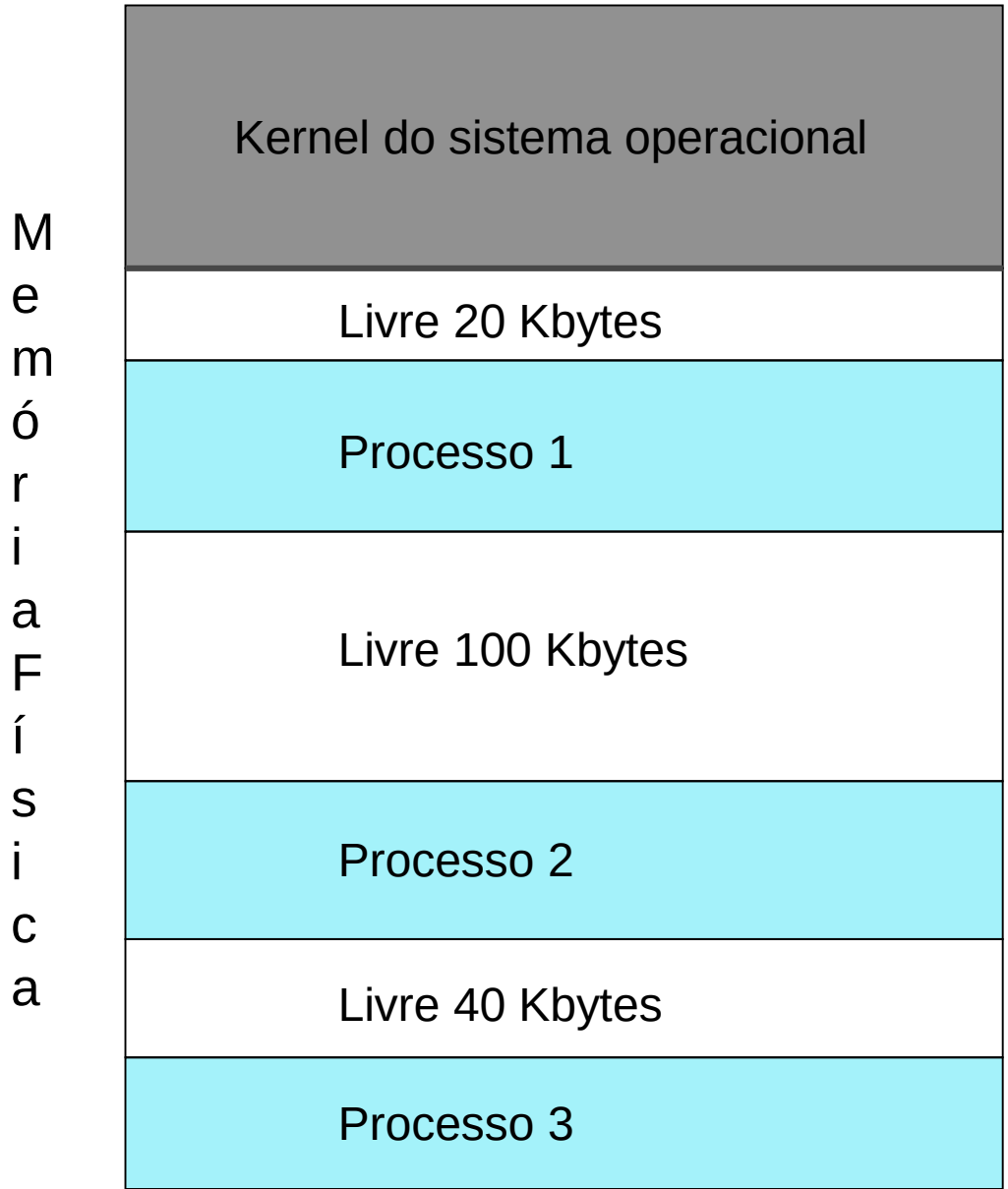
- O tamanho de cada partição é ajustado para a necessidade de cada processo
- Processo é criado, e necessita de uma memória lógica de um certo tamanho
  - A lista é percorrida
  - Uma região de memória física livre com tamanho suficiente é escolhida
- Se região de memória física livre é maior que a memória lógica do processo
  - Apenas o necessário será alocado
- Por exemplo,
  - Processo a ser criado tem uma memória lógica de 100 Kbytes
  - A região de memória física livre tem 150 Kbytes
  - Restará ainda uma região de memória livre com 50 Kbytes

Partições Variáveis 3/8

- Existem diferentes algoritmos para selecionar a área livre a ser usada
- No **Best-Fit** é escolhida a menor região livre disponível com tamanho suficiente
  - Tenta preservar as regiões livres grandes para os processos maiores
- No **Worst-Fit** é escolhida a maior região livre existente, desde que grande o suficiente
  - Busca gerar a maior sobra possível
  - Provavelmente mais útil do que uma sobra pequena
- No **First-Fit** percorre a lista de regiões de memória livre e utiliza a primeira que tiver tamanho suficiente
  - Implementação mais simples
- Variação popular do First-Fit é o **Circular-Fit**
  - First-Fit inicia a busca sempre do início da memória
  - Circular-Fit inicia a busca do ponto onde parou a última pesquisa, tentando distribuir as alocações pela memória
- Não existe dominância de um dos algoritmos sobre os demais
  - Na prática, os algoritmos First-Fit e Circular-Fit são mais usados

Partições Variáveis 4/8

- Quando processo termina, a memória física usada por ele é liberada
- Caso esta região de memória física liberada seja adjacente a uma outra região livre
  - As duas são concatenadas



Partições Variáveis 5/8

- Proteção no caso de partições variáveis pode ser obtida através de um componente de hardware chamado de **Unidade de Proteção de Memória (MPU – *Memory Protection Unit*)**
- Na MPU não existe tradução de endereços (address translation)
  - Os valores dos endereços lógicos são iguais aos valores dos endereços físicos
- O que a MPU faz é verificar se os endereços lógicos gerados por um processo são válidos
  - Estão dentro do seu espaço de endereçamento
  - A MPU dispõe de registradores de limite (fence registers)

- No momento da carga do contexto de execução de um processo
  - Kernel carrega nos registradores da MPU
  - Endereços que representam o limite inferior e o limite superior de cada região de memória alocada
- Durante a execução do processo, a cada acesso à memória,
  - Hardware da MPU compara o endereço gerado com os limites da memória do processo
- Se o endereço for válido, a memória é acessada normalmente
- Caso o endereço esteja fora do espaço lógico do processo
  - Uma exceção de memória (memory exception) é sinalizada
  - Kernel volta a executar, e poderá abortar o processo

Partições Variáveis 6/8

- Gerência de memória precisa ser analisada com respeito às limitações
  - Resultam em subutilização da memória
- **Fragmentação interna (*internal fragmentation*)**
  - Quando mais memória do que é necessário é alocada para o processo

- Memória pode ser gerenciada em pedaços chamados parágrafos, ao invés de bytes
- Neste caso a menor região de memória física livre tem o tamanho de um parágrafo

Partições Variáveis 7/8

- Quando a memória livre total é maior do que o solicitado pelo processo
- Mas o pedido não pode ser atendido em função de como memória é gerenciada
- Ocorre **fragmentação externa (*external fragmentation*)**

- Por exemplo, memória livre total de 160 Kbytes
  - Duas regiões de 80Kbytes cada uma
  - Processo necessita uma região de memória de 110 Kbytes
  - O mesmo não poderá ser atendido
  - Temos fragmentação externa

- Dinâmica intensa de alocações e liberações de regiões de memória com tamanhos diferentes:
  fragmentação externa torna-se um grande problema

Partições Variáveis 8/8

- No caso de **sistemas embutidos ou embarcados (*embedded systems*)**
- Fragmentação externa pode não existir
- Neste tipo de sistema o computador encontra-se embutido dentro de algum equipamento ou máquina maior
- O software é responsável por controlar o equipamento maior
  - Processos são todos criados na inicialização do equipamento
  - Somente são destruídos no desligamento do equipamento
  - Como não existe destruição de processos, não existe liberação de memória
    - Não existe a formação de regiões de memória livre separadas
  - “malloc()” e “free()” usam a memória do próprio processo

Paginação 1/16

- Com partições variáveis a fragmentação externa pode tornar-se um grande problema
- Caso uma região de memória alocada precise ser aumentada
  - Somente será possível se depois dela na memória física vier uma região de memória livre
- A origem do problema está na necessidade de cada região ocupar uma área contígua de memória
- Se pudéssemos quebrar o processo em pequenos pedaços
  - Espalhar os pedaços pela memória física
  - E ainda assim o processo funcionar como se ocupasse uma área contígua
  - Então não existiria mais fragmentação externa
- A paginação faz exatamente isto

Paginação 2/16

- Na **paginação (*paging*)**
  a memória lógica de cada processo é dividida em
  páginas lógicas (*logical pages*) de igual tamanho
- O tamanho da página varia de sistemas para sistema
  - É sempre uma potência de 2
  - Tipicamente cada página ocupa entre 4 Kbytes e 64 Kbytes
- Esta divisão é feita pelo kernel
  - Completamente transparente para o programador e o compilador
- A memória física também é dividida em
  páginas físicas (*physical pages* ou *frames*)
- As páginas físicas possuem o mesmo tamanho que as páginas lógicas

Paginação 3/16

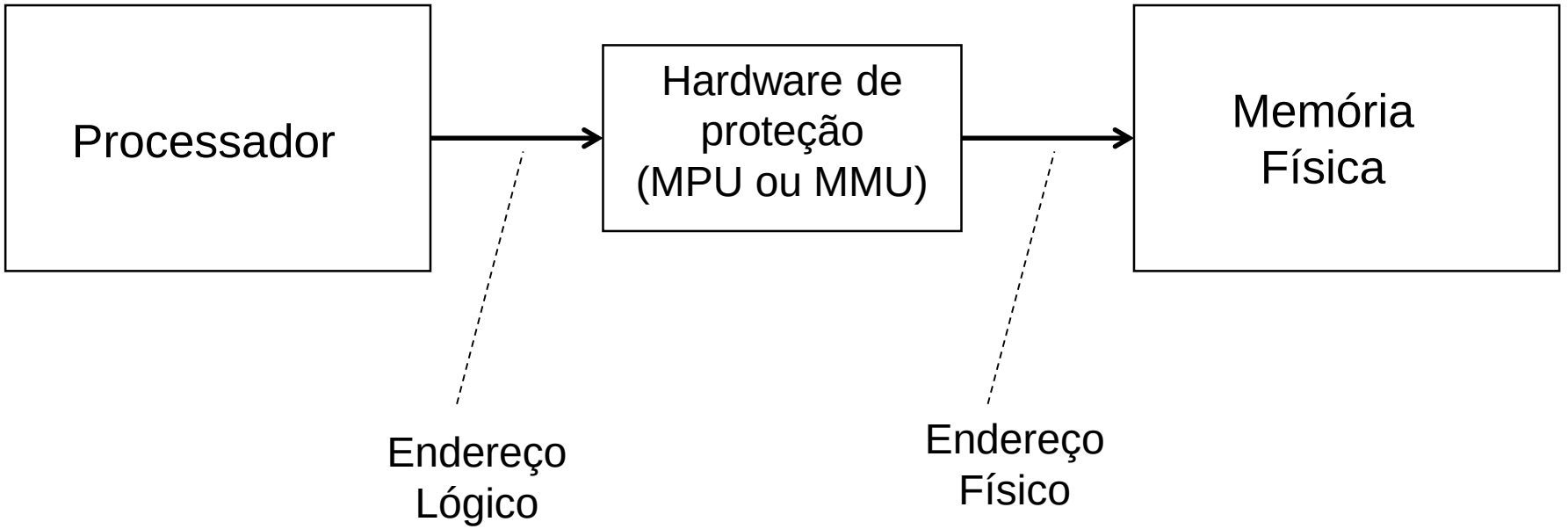
- No momento de alocar a memória lógica do processo
  - Cada página lógica é posicionada individualmente
- Qualquer página lógica pode ser colocada em qualquer página física
- A memória lógica do processo ficará espalhada pela memória física
- Mas não existe mais fragmentação externa

- Se existirem 10 páginas físicas livres espalhadas pela memória
  - Um processo com tamanho de 10 páginas lógicas poderá ser carregado
- Se um processo precisa aumentar de tamanho
  - Basta alocar para ele uma página física livre qualquer da memória
- As limitações das partições variáveis desapareceram

Paginação 4/16

- Porém as instruções do programa executado pelo processo imaginam que o mesmo ocupa uma área contígua de memória
- As instruções de máquina de uma função são supostas contíguas na memória
  - Mas se elas ocuparem duas ou mais páginas lógicas
  - Estarão na verdade em várias páginas físicas espalhadas pela memória
- A implementação da paginação requer um hardware chamado de **Unidade de Gerência de Memória** (**MMU – *Memory Management Unit***)
  - Fica entre o processador e a memória física
  - É responsável pela **tradução de endereços** (***address translation***)
  - São utilizadas tabelas de páginas

Paginação 5/16



Paginação 6/16

- A **tabela de páginas** (***page table***) indica onde na memória física foi colocada cada página lógica
- Tipicamente existe uma tabela de páginas separada para cada processo
- Se um processo possui 30 páginas lógicas
  - Sua tabela de páginas precisa ter 30 entradas
  - Cada uma indicando onde a respectiva página lógica está na memória física
- Funcionamento da tabela de páginas será ilustrado com páginas de apenas 4 bytes
- Memória física de 28 bytes, divididos em 7 páginas físicas de 4 bytes
- Cada **endereço físico** (***physical address***) é composto por 5 bits, de 00000 até 11011
- Cada endereço físico é dividido em duas partes:
  - Número da página física e deslocamento dentro da página
- O **deslocamento** dentro da página (***offset***) indica a posição do byte na página
- Página tem 4 bytes, existem os deslocamentos 00, 01, 10 e 11
- **Número da página física** (***physical page number***) é único para cada página
  - Vai de 000 até 110

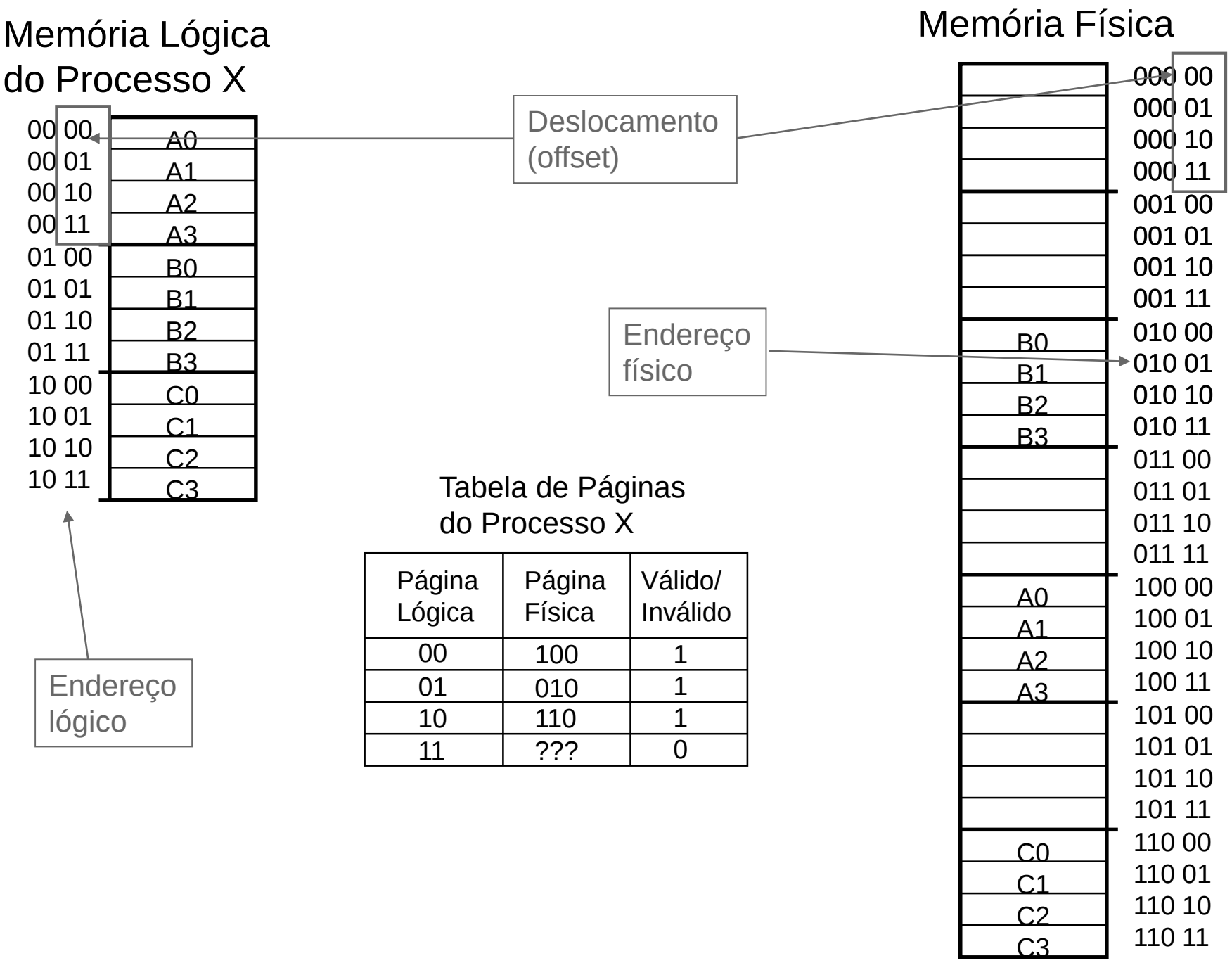


Tabela de Páginas
do Processo X

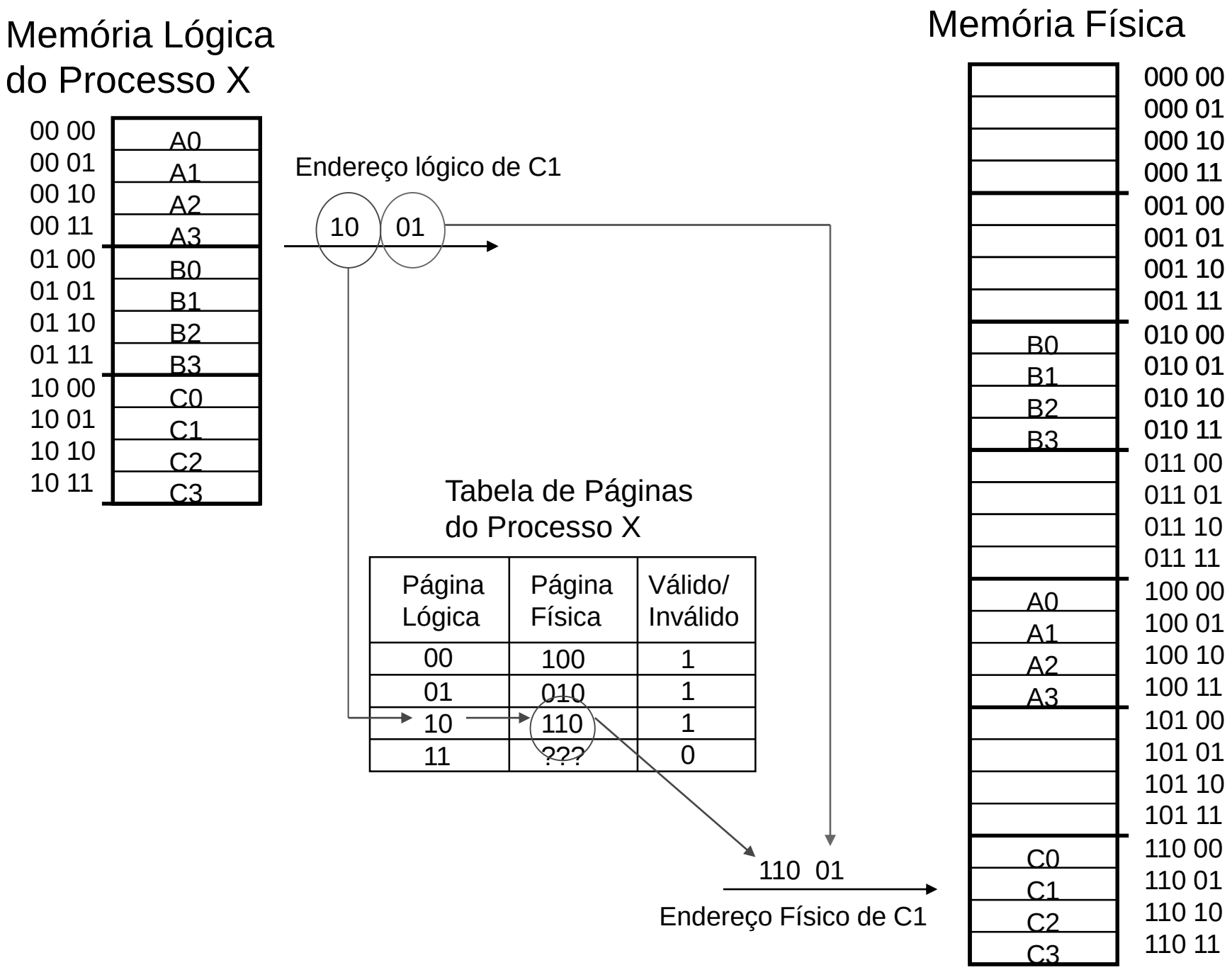
Página Lógica	Página Física	Válido/ Inválido
00	100	1
01	010	1
10	110	1
11	???	0

Paginação 8/16

- Memória lógica do processo X é composta por 12 bytes
  - Divididos em 3 páginas lógicas de 4 bytes
- Cada endereço lógico de 4 bits é dividido em
  - número da página lógica (logical page number)
  - deslocamento dentro da página (offset)
- No momento de alocar memória física
  - Cada página lógica é colocada em uma página física diferente
  - Seu número anotado na tabela de páginas do processo
  - Foram usadas as páginas físicas 100, 010 e 110 para as páginas lógicas 00, 01 e 10
- Tabela de páginas também possui um bit de válido/inválido para cada página lógica
  - Como a página lógica 11 não existe para o processo X, esta entrada é marcada inválida
- O i-ésimo byte em uma página lógica também será o i-ésimo byte na página física
- B3 tem deslocamento 11 na página lógica e deslocamento 11 na página física
  - O tamanho da página precisa ser potência de dois

Paginação 9/16

- Durante a execução, o processo gera endereços lógicos
- Endereços lógicos são traduzidos para endereços físicos pela MMU
  - No momento do acesso à memória física
- Suponha que o processo deseja acessar o byte C1 cujo endereço lógico é 1001
- MMU separa o endereço lógico em
  - Número da página lógica (10)
  - Deslocamento (01)
- Como o deslocamento do byte dentro da página lógica é igual ao seu deslocamento dentro da página física, este valor é simplesmente copiado
- Já o número da página lógica é usado para indexar a tabela de páginas
- Caso a entrada em questão seja inválida
  - MMU gera uma exceção de acesso à memória
- Se a entrada é válida, o número da página física em questão (110) é obtido da tabela
  - Colocado com o deslocamento no endereço físico


Memória Lógica
do Processo X
00 00
00 01
00 10
00 11
01 00
01 10
01 11
10 00
10 01
10 10
10 11
A0
A1
A2
A3
B0
B1
B2
B3
C0
C1
C2
C3
Endereço lógico de C1
10
01
Tabela de Páginas
do Processo X
Página Lógica
Página Física
Válido/ Inválido
00
100
1
01
010
1
10
110
1
11
???
0
110 01
Endereço Físico de C1
Memória Física
000 00
000 01
000 10
000 11
001 00
001 01
001 10
001 11
010 00
010 01
010 10
010 11
011 00
011 01
011 10
011 11
100 00
100 01
100 10
100 11
101 00
101 01
101 10
101 11
110 00
110 01
110 10
110 11
B0
B1
B2
B3
A0
A1
A2
A3
C0
C1
C2
C3

Paginação 11/16

- A tradução de endereço lógico em endereço físico ocorre a cada acesso à memória
  - Pode ser necessária várias vezes na execução de uma única instrução de máquina
- A MMU mantém dentro dela uma memória pequena, porém rápida, onde ficam armazenadas as entradas da tabela de páginas mais recentemente utilizadas
- Essa memória interna à MMU é chamada de ***Translation Lookaside Buffer*** (**TLB**)
- Ela não é grande o suficiente para conter toda a tabela de páginas
  - Fica completa na memória principal
- Quando a entrada da tabela de páginas necessária para a tradução de endereço lógico em endereço físico não está na TLB
  - A MMU acessa a tabela de páginas completa na memória e copia para a TLB a entrada
  - Em seguida faz a tradução e o acesso à memória física solicitado pelo processo
  - Neste caso o tempo de acesso à memória na prática dobra
- Esta penalização é tolerada por que a taxa típica de acertos na TLB é alta
  - Taxas de acerto acima de 90% são comuns

Paginação 12/16

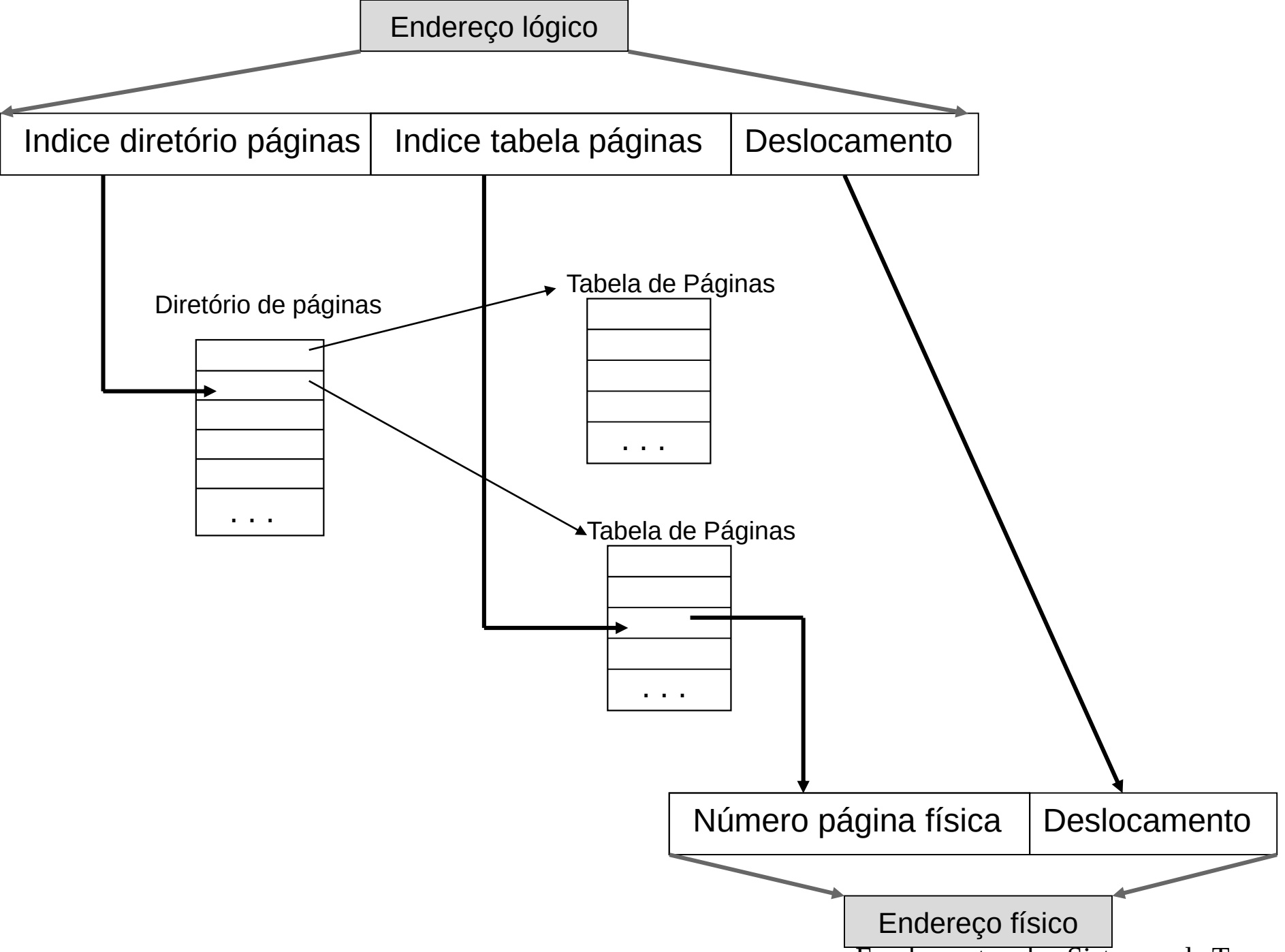
- Cada processo possui a sua própria tabela de páginas
- Faz parte do chaveamento de contexto informar qual tabela de páginas a MMU deve usar
- A gestão da memória em termos de partes ocupadas e partes livres é feita em termos de páginas
- Se o processo precisa de 87 Kbytes e o sistema trabalha com páginas de 2 Kbytes
  - Ele receberá 44 páginas, ou seja, 88 Kbytes
- Na paginação temos uma pequena fragmentação interna
- O tamanho da página é definido pelo hardware da MMU
  - O desenvolvedor do kernel pode optar entre alguns valores possíveis suportados

Paginação 13/16

- Toda a gestão da memória é feita em termos de páginas
- Desejado que a tabela de páginas ocupe exatamente uma página
- Suponha que em certo sistema as páginas são de 4 Kbytes
  - E cada entrada da tabela ocupe 4 bytes
- Então a tabela de páginas teria sempre 1K entradas
  - Suportando uma memória lógica de até 1K páginas lógicas
- Se o processo não precise de tantas páginas lógicas
  - Basta marcar as entradas não usadas como inválidas
- Porém, caso o processo precise de 2K páginas lógicas
  - Precisaria de uma tabela de páginas com 2K entradas
  - O que ocuparia 2 páginas da memória para ser armazenado
  - Não é conveniente ter que procurar na memória física duas páginas livres adjacentes para armazenar tal tabela

Paginação 14/16

- Solução típica é empregar tabelas de **páginas hierárquicas (*multilevel page tables*)**
- No caso de uma tabela de páginas em dois níveis
  - O endereço lógico é dividido em número da página lógica e deslocamento
  - Porém agora o número da página lógica é dividido em duas partes
  - A primeira parte indexa uma tabela de páginas de primeiro nível
    - Diretório de páginas (page directory)
  - A MMU descobre qual tabela de páginas de segundo nível a ser usada
  - A segunda parte indexa a tabela de páginas de segundo nível
    - Descobre onde está a página física a ser acessada
- Entradas não usadas pelo processo tanto no diretório de páginas como nas tabelas de páginas são marcadas como inválidas


Endereço lógico
Indice diretório páginas
Indice tabela páginas
Deslocamento
Diretório de páginas
Tabela de Páginas
. . .
Tabela de Páginas
. . .
. . .
Número página física
Deslocamento
Endereço físico

Paginação 16/16

- Tamanhos reais de memória lógica, memória física, tabelas de páginas, etc, variam de processador para processador
- Exemplo: processador Intel 80386
- Endereços lógicos (chamados endereços lineares) possuem 32 bits
  - Permitindo um espaço de endereçamento lógico de até 4 Gbytes
- Páginas são de 4 Kbytes
  - Deslocamento é composto por 12 bits
- Sobram 20 bits para o número da página lógica
- Uma tabela de páginas hierárquica com dois níveis é empregada
- Dos 20 bits disponíveis
  - Primeiros 10 bits são usados para indexar diretório de páginas com 1K entradas
  - É identificada qual tabela de páginas deve ser usada
  - Ela é indexada com os 10 bits restantes para determinar a página física

Outros Aspectos da Paginação

- A memória lógica dos processos pode ser dividida
- Por exemplo:
  - Código do programa, código da biblioteca da linguagem, pilha onde ficam parâmetros de funções e variáveis locais, variáveis globais inicializadas e memória alocada dinamicamente
- Em algumas arquiteturas, cada divisão tem endereçamento próprio
- Este esquema é chamado de **segmentação** (***segmentation***)
- Endereço lógico é composto por
  - Um número de segmento
  - Um deslocamento dentro do segmento
- Uma **tabela de segmentos** (***segment table***) informa onde o segmento foi colocado na memória física
- Segmentos possuem tamanhos diversos
  - Gera fragmentação externa
  - Eliminada com a paginação de cada segmento
- Segmentação permite o compartilhamento de código entre processos

Memória Virtual com Paginação por Demanda 1/4

- Em geral imagina-se que um programa precise estar completo na memória do computador para executar
- Isto não é verdade
- Considere um grande aplicativo como o editor de texto Word da Microsoft
  - Centenas de funcionalidades
  - Cada funcionalidade está associada com código executável que a implementa
  - Durante uma sessão de trabalho com o Word, caso o usuário não ative uma determinada opção do menu, aquele código que a implementa jamais será executado
  - Não existe razão para colocá-lo na memória do computador
- O código e as variáveis que implementam estas funcionalidades não precisam ficar na memória do computador o tempo todo
  - O processo que executa o programa precisa achar que eles estão lá

Memória Virtual com Paginação por Demanda 2/4

- A técnica de gerência de memória que permite executar programas que estão apenas parcialmente na memória física é chamada de **memória virtual (*virtual memory*)**
- Implementação típica emprega
  paginação por demanda (***demand paging***)
- Memória virtual requer a existência de um ou mais dispositivos de **armazenamento secundário (*secondary storage*)**
  - Como uma unidade de disco rígido (hard disk drive)
- No armazenamento secundário os programas são mantidos completos
- Apenas a parte necessária deles é trazida para a memória principal

Memória Virtual com Paginação por Demanda 3/4

- Na paginação por demanda apenas as páginas lógicas efetivamente acessadas pelo processo são alocadas na memória física
- Se uma página lógica jamais for acessada pelo processo, ela nunca ocupará espaço na memória física
- Páginas lógicas que fazem parte do espaço de endereçamento do processo mas ainda não foram carregadas para a memória física são marcadas como inválidas na tabela de página
- Quando for acessada, ocorrerá uma interrupção de proteção
  - Neste caso uma **falta de página (*page fault*)**
- A página demandada é carregada para a memória física
  - A tabela de páginas é atualizada
  - O processo pode seguir sua execução

Memória Virtual com Paginação por Demanda 4/4

- **Tempo efetivo de acesso à memória** (effective memory access time)
  - Média ponderada dos tempos de acesso
    quando ocorre e quando não ocorre uma falta de página
- Pode ser muito mais lento que apenas paginação pura
- Sistemas operacionais de tempo real evitam empregar memória virtual
- O impacto deste mecanismo na variância dos tempos de execução das tarefas é muito grande
- Tarefas com requisitos temporais rigorosos em geral desligam o mecanismo
- Ou simplesmente configurar o kernel para que memória virtual seja totalmente desligada

Resumo

- Introdução
- O Sistema Operacional Tradicional
- Terminologia: Processos, Threads e Tarefas
- Chamadas de Sistema
- Estados da Thread de um Processo
- Chaveamento de Contexto
- Gerência de Memória
- Partições Variáveis
- Paginação
- Outros Aspectos da Paginação
- Memória Virtual com Paginação por Demanda