操作系统原理

第九章：内存管理

洪明坚

重庆大学软件学院

February 19, 2019

目录

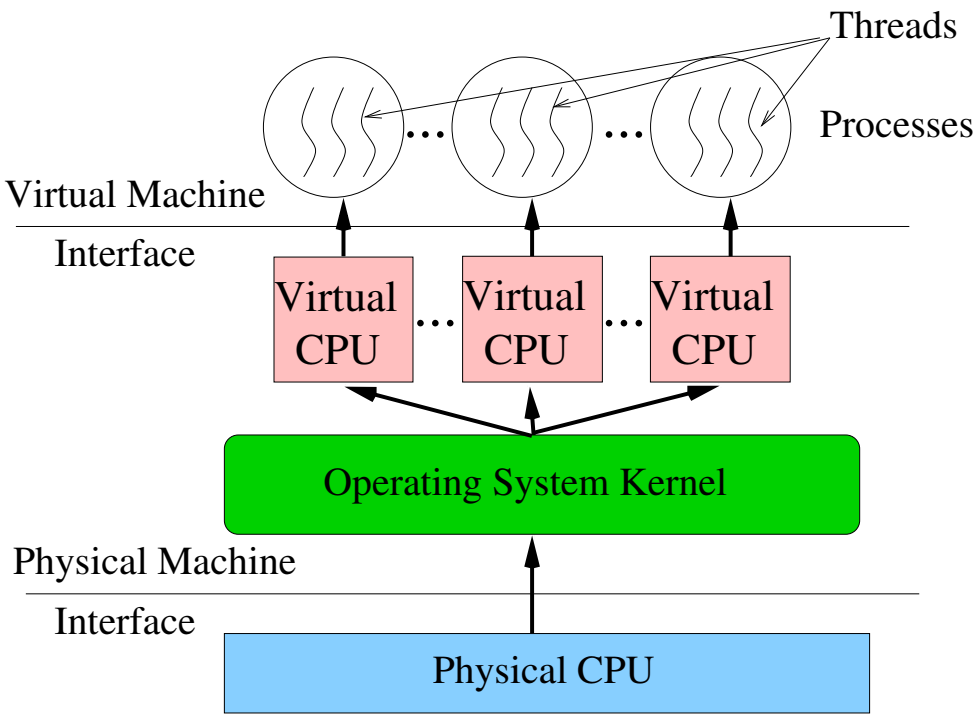
1. 内存管理介绍
2. 基本方法
3. 分页内存管理
4. 分段内存管理
5. 段页式内存管理

Outline

1. 内存管理介绍
2. 基本方法
3. 分页内存管理
4. 分段内存管理
5. 段页式内存管理

内存管理介绍

- 进程管理回顾
  - 进程管理提供了一个虚拟的机器接口，让每一个进程都以为是自己在独占CPU，如下图所示：



- 内存管理的任务
  - 提供一个虚拟的机器接口，让每一个进程都以为是自己在独占RAM。

Questions

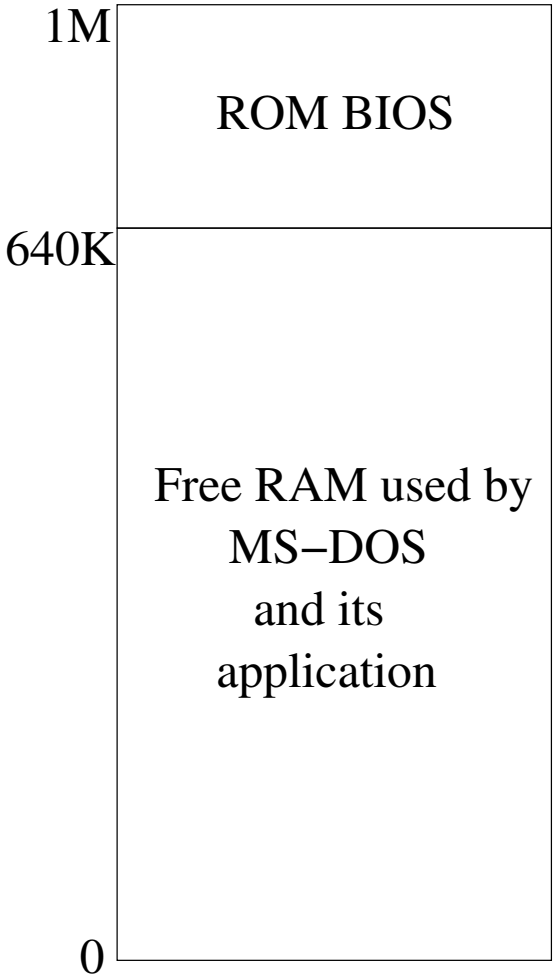
- Any questions?

Outline

1. 内存管理介绍
2. 基本方法
3. 分页内存管理
4. 分段内存管理
5. 段页式内存管理

基本方法- 以MS-DOS为例

- MS-DOS (Microsoft Disk Operating System)
  - 单用户、单任务
  - 只能访问1MB内存
    - INTEL 8086/80286只有20根地址线
  - 没有任何保护机制
    - INTEL 8086/80286没有提供硬件保护机制支持



640K should be enough for anybody – Bill Gates

问题（一）

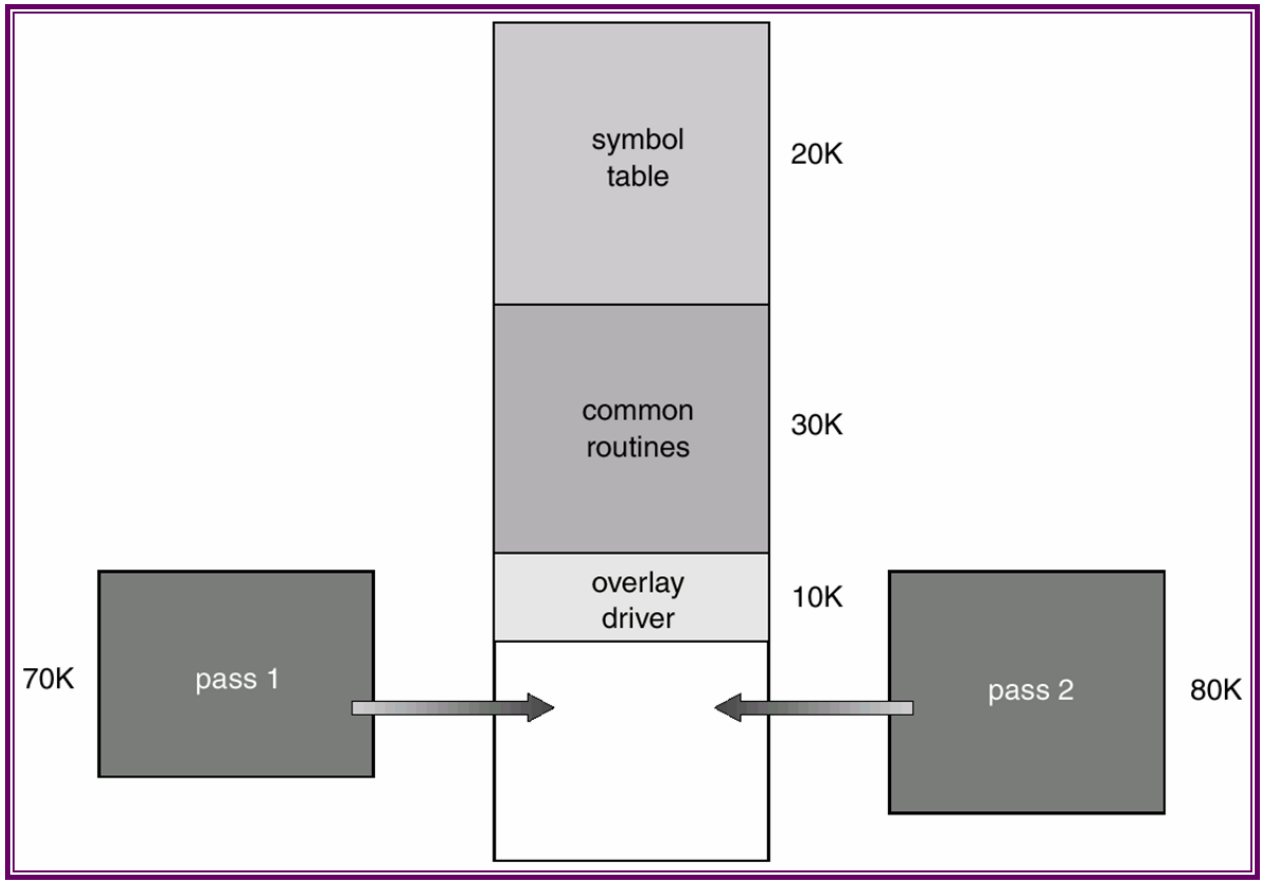
- 在MS-DOS中，
  - MS-DOS自己要占用1/3左右；
  - 剩余部分留给系统唯一的进程使用。
- 问题
  - 如果某个MS-DOS下的应用程序需要超过640K的内存才能运行，怎么办？
- Solution
  - **Overlay**: A technique to enable a process to be larger than the amount of memory allocated to it.
  - The idea of overlay is keeping in memory only those instructions and data that are needed at any given time.

Overlay: An example

- A two-pass assembler
  - Pass 1: 扫描源代码，找到所有的符号并构造符号表（symbol table）；
  - Pass 2: 根据前面的符号表产生机器代码。
- 为一致起见，假设
  - 系统目前只有150K内存可以使用，而该assembler需要200K的内存才能运行；
  - 该assembler在运行时主要包括如下模块，
    - Pass 1，需要70K内存；
    - Pass 2，需要80K内存；
    - 符号表，需要20K内存；
    - Pass 1和2共用代码（common routines），需要30K内存。

Overlay: How to do?

- 程序员根据程序的特点，将一些独立的模块根据需要调入内存，覆盖已经使用过的模块，如下图所示：
- 除了程序本身外，程序员还要提供一个所谓的“overlay driver”来负责模块的调入调出。



Overlay: 结论

- 前面的例子是最简单情况下的overlay。一般情况下，对复杂的系统进行overlay要求开发人员对系统进行精心的设计和实现，需要很高的技巧；
- 因此，目前这种技术已经很少被使用，除了在一些内存资源相当紧张的嵌入式系统中；
- 但是，Overlay的思想非常重要：
  - 无论进程在运行时占有多大的内存，在某一段时间内，它只会访问其中的一部分。

Questions

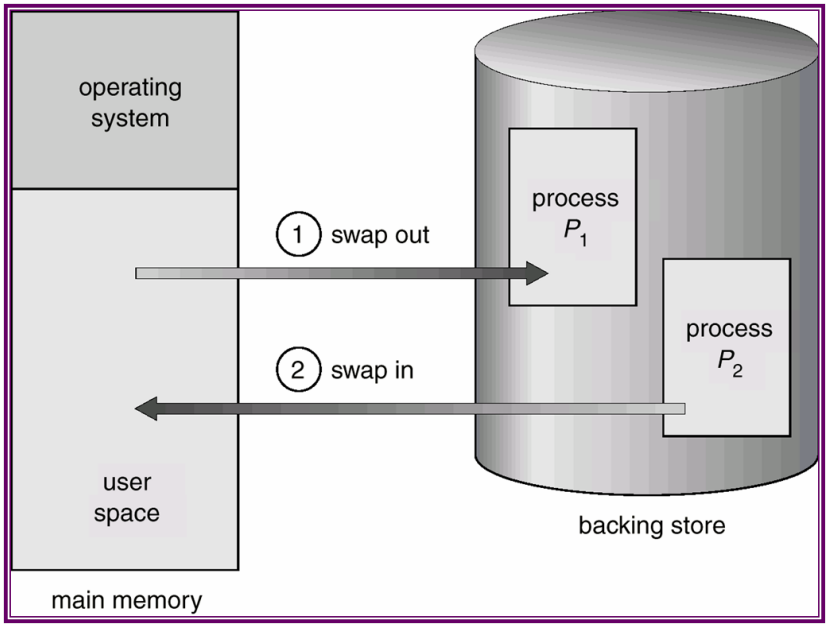
- Any questions?

问题（二）

- 假设MS-DOS支持多任务，即系统中有多个进程。当然，
  - 进程必须在内存中才能运行；
  - 运行中的进程可能会申请额外的内存。
- 问题
  - 假设系统目前有两个进程：P_1和P_2，而且系统已经没有内存可以使用。此时，正在运行的P_1又要申请更多的内存才能继续运行，怎么办？
- Solution
  - **Swap**: A process can be swapped temporarily out of memory to a backing-store, and then brought into memory for continued execution.

Swap: How to do?

- 此时，OS可以将（不在运行的）P_2交换到backing-store中；同时释放P_2所占用的内存，并分配给P_1让其继续运行。
- 当调度器重新调度P_2运行时，OS从backing-store中加载P_2到内存继续运行。
  - 此时可能要把P_1交换到backing-store中以释放足够的内存空间供P_2运行。



Swap: 结论

- Swap方法存在的限制：
  - 要求计算机必须配备足够大的backing-store；
    - backing-store一般是快速、大容量的硬盘。
  - Context switch要花费大量的时间：
    - 主要用于磁盘数据传输。因此，
    - 调度算法的设计尤其重要，应尽量减少context switch。
  - 被swap-out的进程必须被重新swap-in到相同的内存地址才能继续运行。
- 因此，这种原始的swap-in/out已经很少被使用。
- 但是，swap的思想非常重要：
  - 当系统内存不足时，可以向backing-store“借”一部分来使用。

Questions

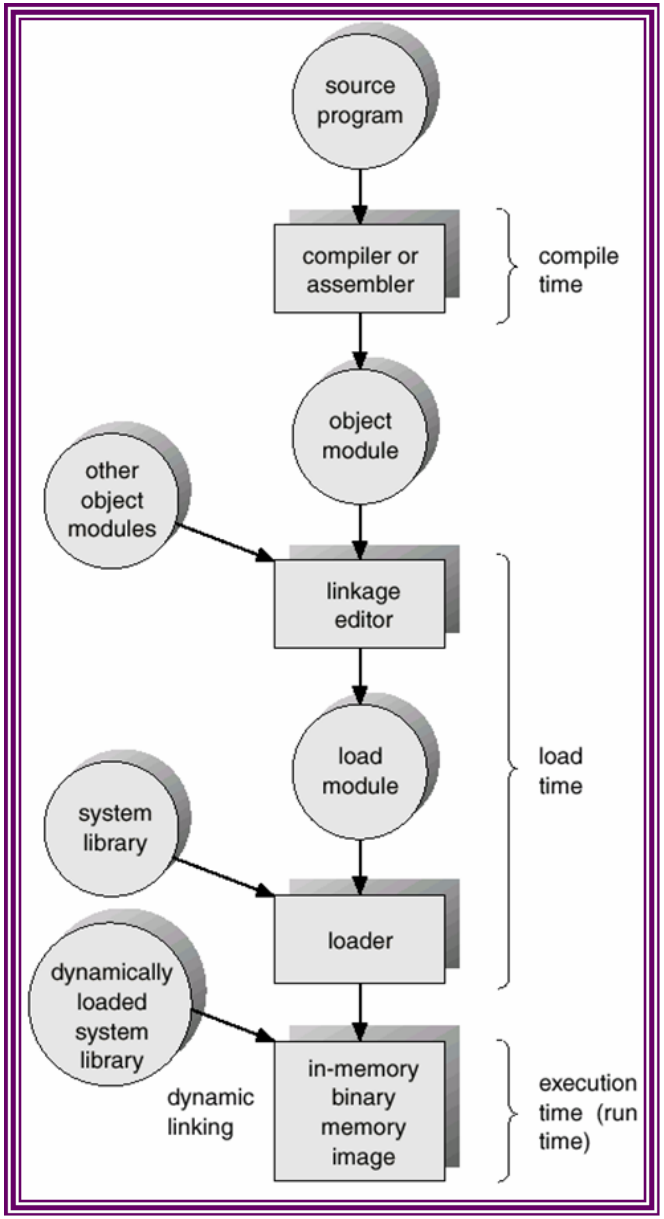
- Any questions?

多任务系统的内存管理

- 多任务环境下会带来许多内存管理问题：
  - 重定位（relocation）问题；
  - 内存保护（protection）问题；
  - 内存分配（allocation）问题。
- 下面将逐一展开讲解。

Relocation: 背景

- 源程序变成进程的过程：
  - 程序员编写的代码（source code）必须先被编译成目标文件（object file）；
  - 然后通过链接器（linkage editor）链接成可执行文件（executable）；
  - 最后由操作系统加载可执行文件到内存从而形成进程，如图所示。



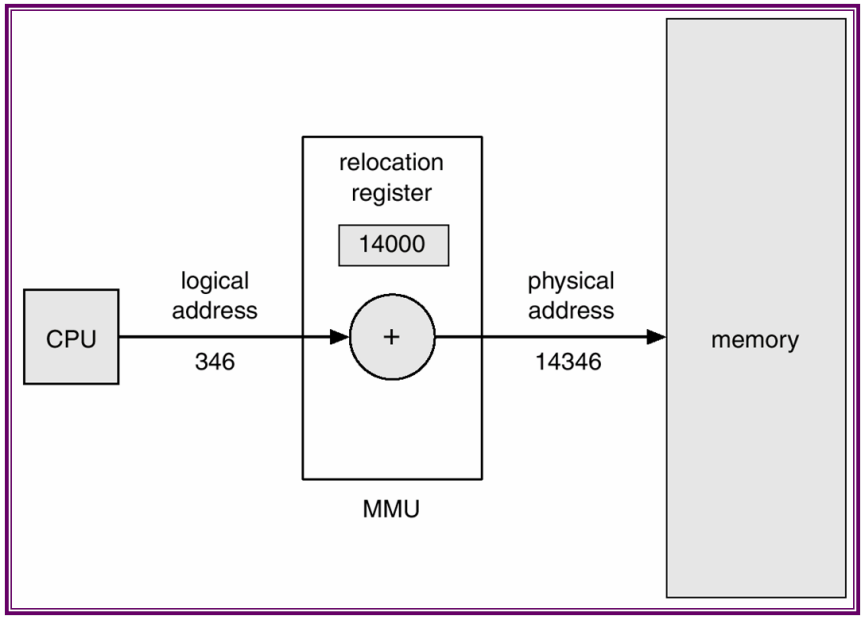
Relocation: 概念

- 在多任务的环境中，可执行文件可能会被加载到内存中的任何位置运行。
- 链接器在生成可执行文件时必须确定程序中各个符号（如函数、全局变量）的地址。
  - 把程序中的符号映射为地址的过程叫做地址绑定（address binding）。
- 事实上，由于链接器无法预知程序将被加载到哪个内存位置，因此无法完成**绝对的**地址绑定。
  - 因此，链接器只能假定程序中第一条指令的地址是0，从而用相对于它的偏移量来进行**相对的**地址绑定；
  - 这样的程序只能被加载到0地址的内存运行；如果该程序被加载到其他非0的地址，必须对程序中所引用的地址进行修改才能运行，这个修改过程就称为重定位。

Relocation: How to do?

- 几个概念：
  - 逻辑地址（logical address）指程序中引用的地址，亦即CPU产生的地址；
  - 物理地址（physical address）指系统中内存单元所看到的地址；
  - 内存管理单元（Memory Management Unit, MMU）指专门完成逻辑地址到物理地址转换的硬件单元，一般是CPU的一部分。

- 假设某个程序被加载到地址14000运行，正在访问逻辑地址为346的内存，如图所示：



Relocation: 结论

- 前面介绍的是最简单的重定位技术，只是冰山的一角；
- 我们今天使用的操作系统（如Windows，Linux）所采用的重定位技术相当复杂，涉及到编译器、链接器、可执行文件、函数库、操作系统内核和硬件等多个组件之间的协作才能完成；
- 在讲完虚拟内存技术后，我们将继续深入讲解这部分内容。

Questions

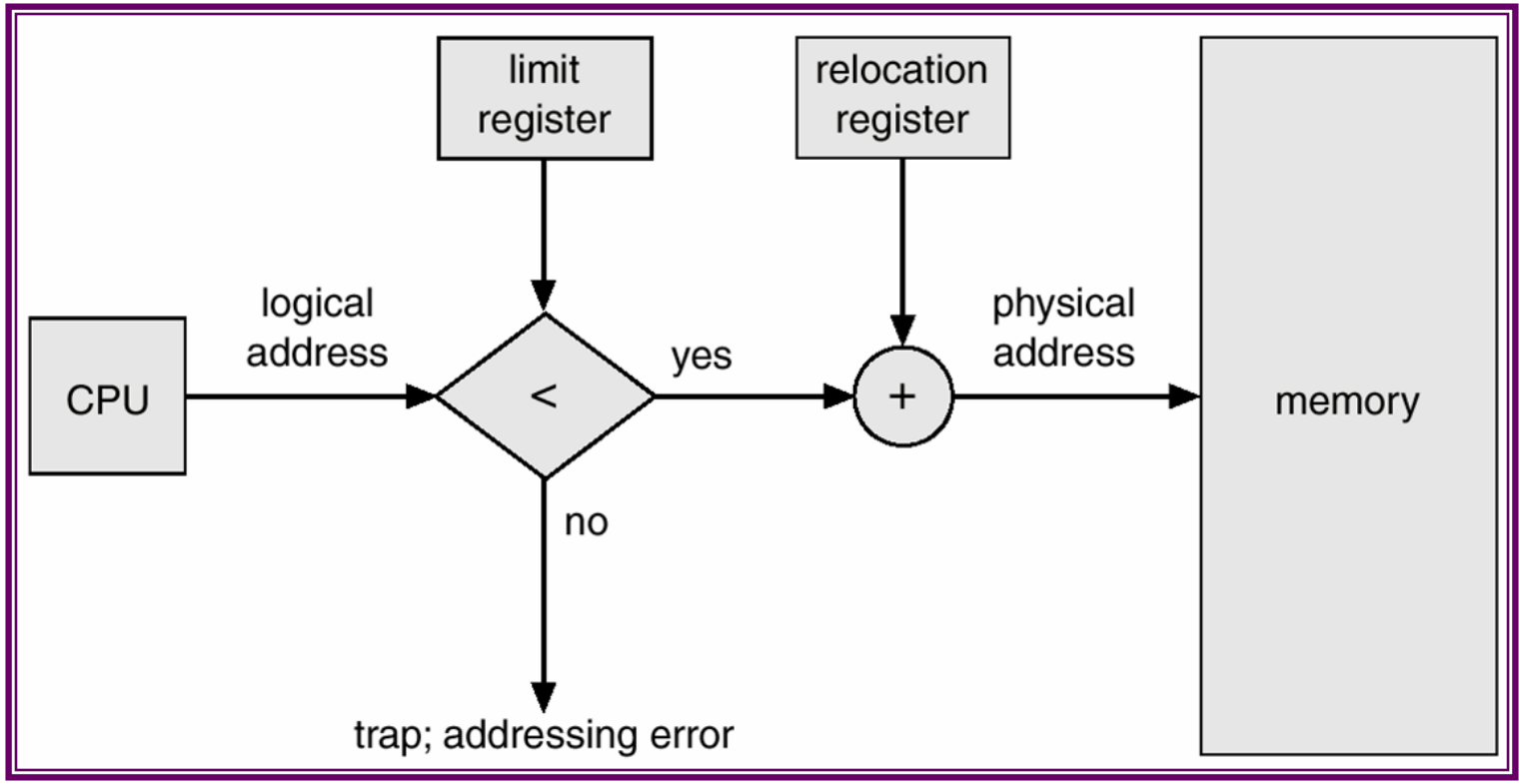
- Any questions?

Protection: 问题

- 在多任务的环境中，如何保护各个进程的内存不被其他进程访问或破坏是必须解决的问题。
  - 其实，即使在单任务的环境中也存在着操作系统如何保护自己不被应用程序访问或破坏的问题。
  - 在MS-DOS中，应用程序可以合法地把MS-DOS的内核所占的内存全部破坏。
- 问题：操作系统如何保护自己和应用程序的内存？

Protection: How to do?

- 对应用程序访问的每一个内存地址进行检查，看是否超出了内存范围。
  - 为了获得最好的性能，一般用MMU通过硬件来实现这种检查的功能。



Protection: 总结

- 内存保护是现代操作系统必备的功能之一，
  - 它需要硬件的支持才能实施保护。
- 后面讲解的各种内存管理方法都包含有内存保护的方法。

Questions

- Any questions?

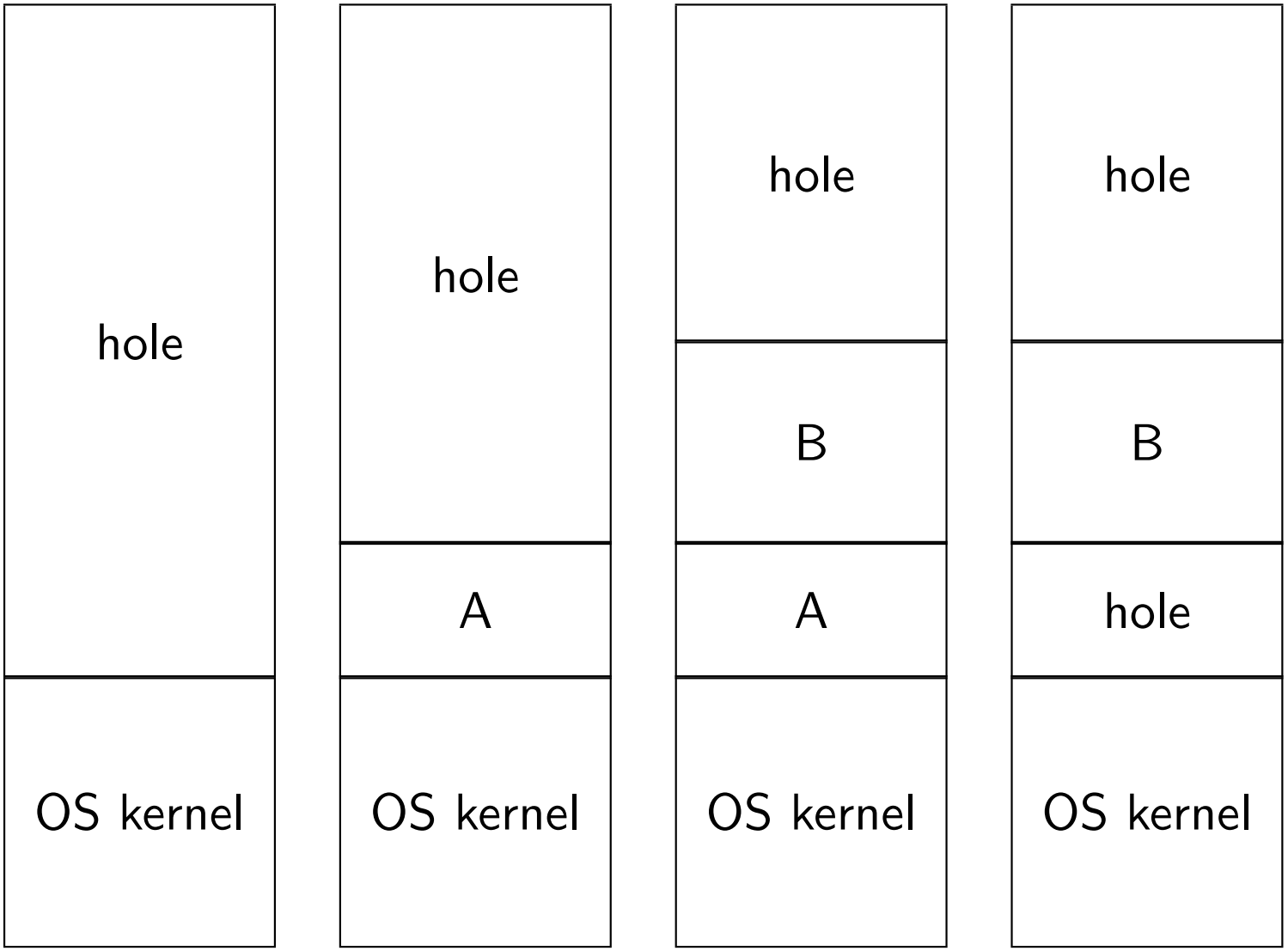
Allocation: 问题

- 在多任务的环境中，
  - 操作系统需要为每个新创建的进程分配一定数量的内存才能运行；
  - 当进程退出时，操作系统要回收它所占用的内存。
- 问题：
  - 操作系统如何有效地管理内存的分配和回收，以尽量满足进程的需求？
  - 这个问题也称为动态内存分配问题（Dynamic memory allocation problem）。

Allocation: How to do?

- The OS keeps a table indicating which parts of memory are available and which are occupied.
  - Available memory blocks are called **holes**.
  - Initially, all memory is available for user processes.
  - When a process arrives and needs memory, OS search for a hole large enough for it.
  - When a process terminates, it returns the memory to OS as a hole, and
  - If the new hole is adjacent（相邻）to other holes, these adjacent holes are merged（合并）to form one larger hole.

Allocation: Example



Initially | A enters | B enters | A leaves | Going on

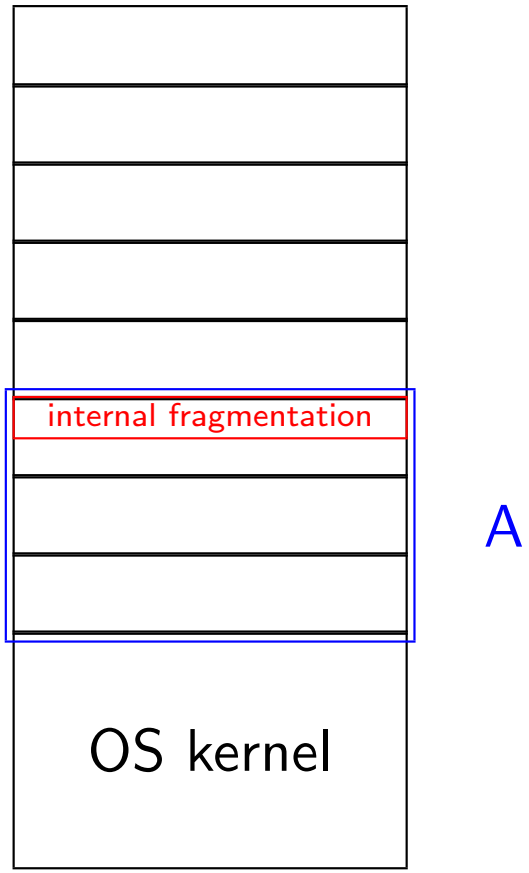
.....

Allocation: 算法

- 随着系统中进程的创建和退出，系统中的内存可能会形成很多小的hole，
  - 这些hole既不足以满足任何的进程需求，也不能被合并以形成大的hole。这些hole被称为外部碎片（external fragmentation）。
- 因此，必须设计出专门算法来进行内存的分配和释放以减少外部碎片，提高内存使用率。简单、常用的有：
  - 首次适应（First fit）：Allocate the first hole that is big enough.
  - 最佳适应（Best fit）：Allocate the smallest hole that is big enough.
  - 最坏适应（Worst fit）：Allocate the largest hole.

Allocation: 另一种方法

- 我们把系统内存分成固定大小的内存块，
  - 操作系统以块为单位进行内存的分配和释放。
  - 最终分配给进程的内存可能会比所请求要多一点，如右图所示。
  - 我们称这多出来的部分为内部碎片（Internal Fragmentation）。

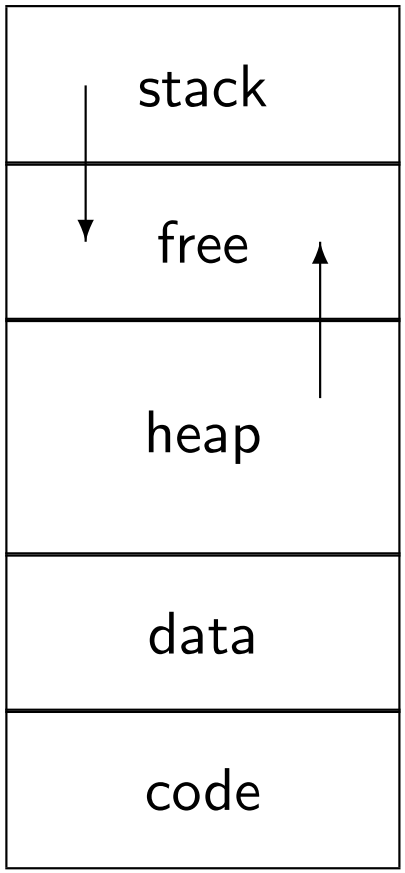


Allocation: 总结（1/2）

- Allocation主要解决在多任务的环境中，操作系统采用什么算法进行内存的分配和回收，减少外部碎片或内部碎片，提高内存利用率。
  - 相对于外部碎片，内部碎片的情况不是很严重;
  - 后面我们将介绍两种消除外部碎片的方法。

Allocation: 总结（2/2）

- 事实上，内存的分配和回收问题不仅出现在OS中，应用程序面临着同样的问题。
  - 当进程被创建时，操作系统会采用某种算法分配一块足够大的内存给进程，由进程自己管理（其中的一部分），如图所示：
- 在该图中，
  - **heap**由进程自己管理。库函数new/delete（或malloc/free）就是操纵heap中的内存。它们所采用的分配与回收算法非常通用，相应的空间和时间利用率也很一般。在一个大型的项目中，已有的库函数可能不会满足应用的要求，需要自己开发更好的来替换它们以负责heap中内存的分配与回收。



Questions

- Any questions?

Outline

1. 内存管理介绍
2. 基本方法
3. 分页内存管理
4. 分段内存管理
5. 段页式内存管理

介绍（1/2）

- 基本内存管理方法存在着很大的限制，
  - 每个进程所占用的物理内存必须是连续的，
  - 系统可能会产生大量的外部碎片，最后不可避免地要进行compaction，
  - 对整个进程进行swap-in/swap-out非常地耗时。
- 分页很好地解决了上面的问题，
  - 进程所占用的物理内存不必连续，
  - 没有外部碎片，但是会有一定量的内部碎片，
  - 对进程所占用的部分内存进行swap-in/swapout。
- 在早期，分页系统主要由硬件来实现；如今，分页由硬件和操作系统共同完成。

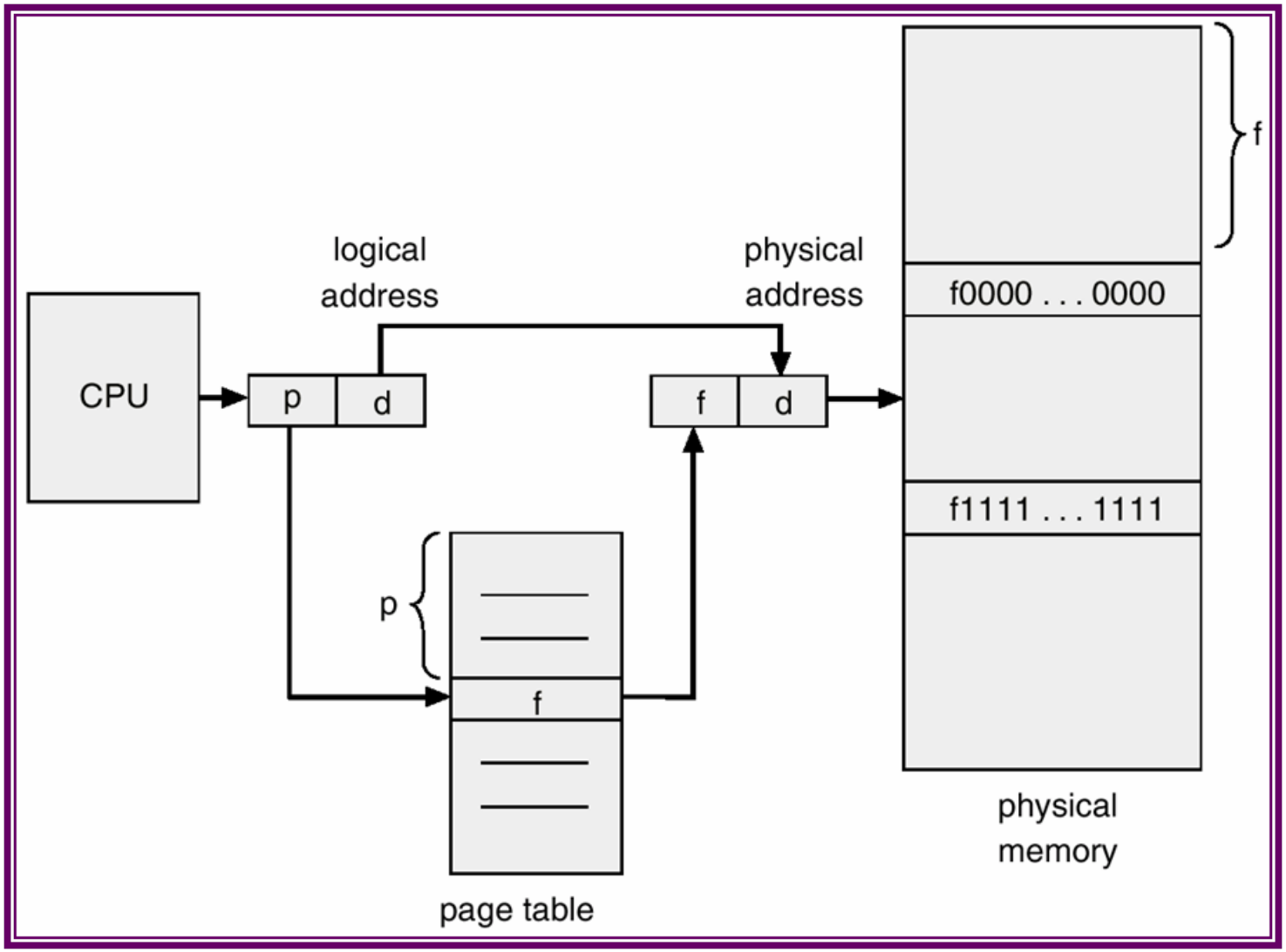
介绍（2/2）

- 基本概念
  - Physical memory is broken into fixed-sized blocks called **frames**.
  - Logical memory is also broken into blocks of the same size called **pages**.
  - **Page tables** are used to map pages to frames.
    - An entry of the page table is called **Page Table Entry (PTE).**

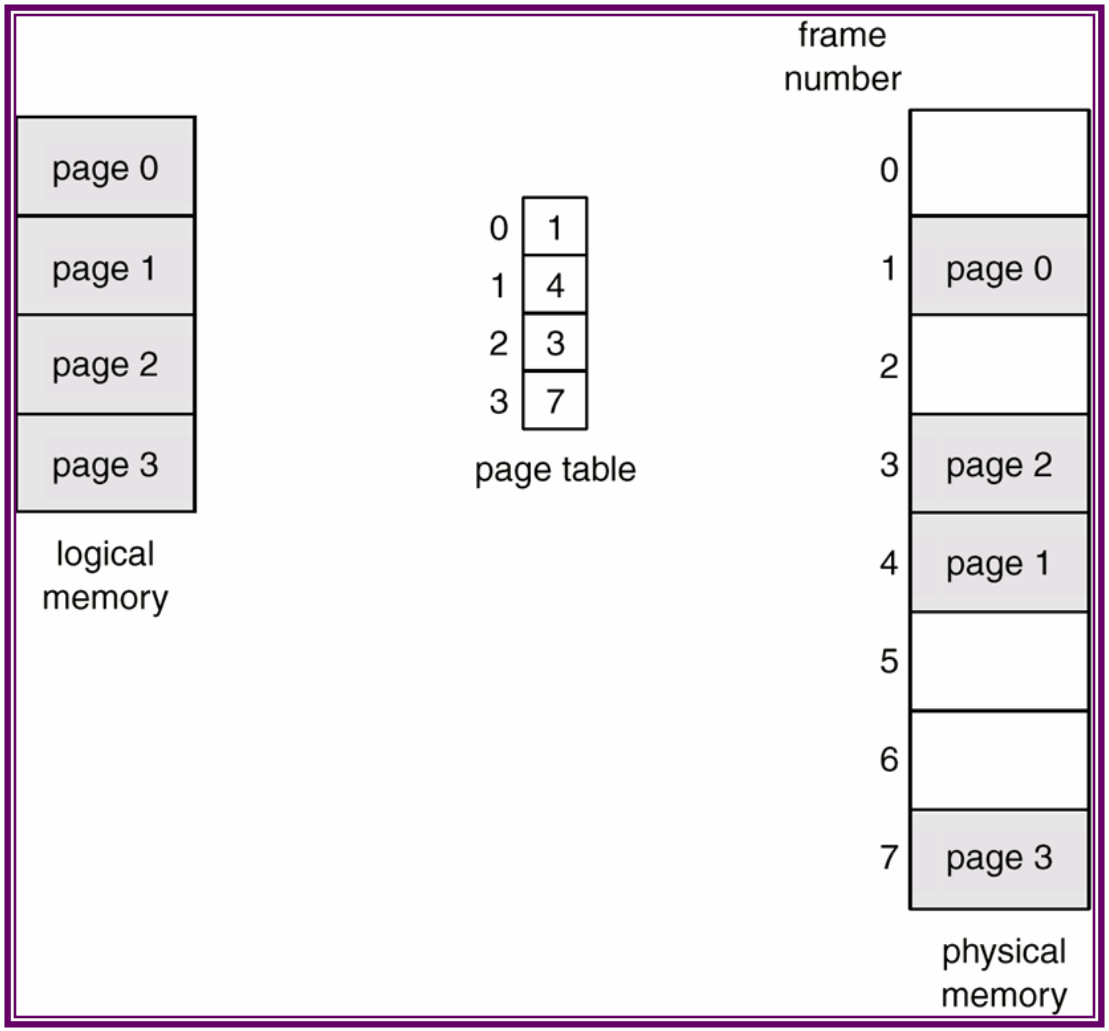
方法

- 把逻辑地址分成两部分：
  - 第一部分称为**page number**；
  - 第二部分称为**page offset**。
- 地址转换
  - 在page table的帮助下，MMU把CPU产生的逻辑地址转换成物理地址。

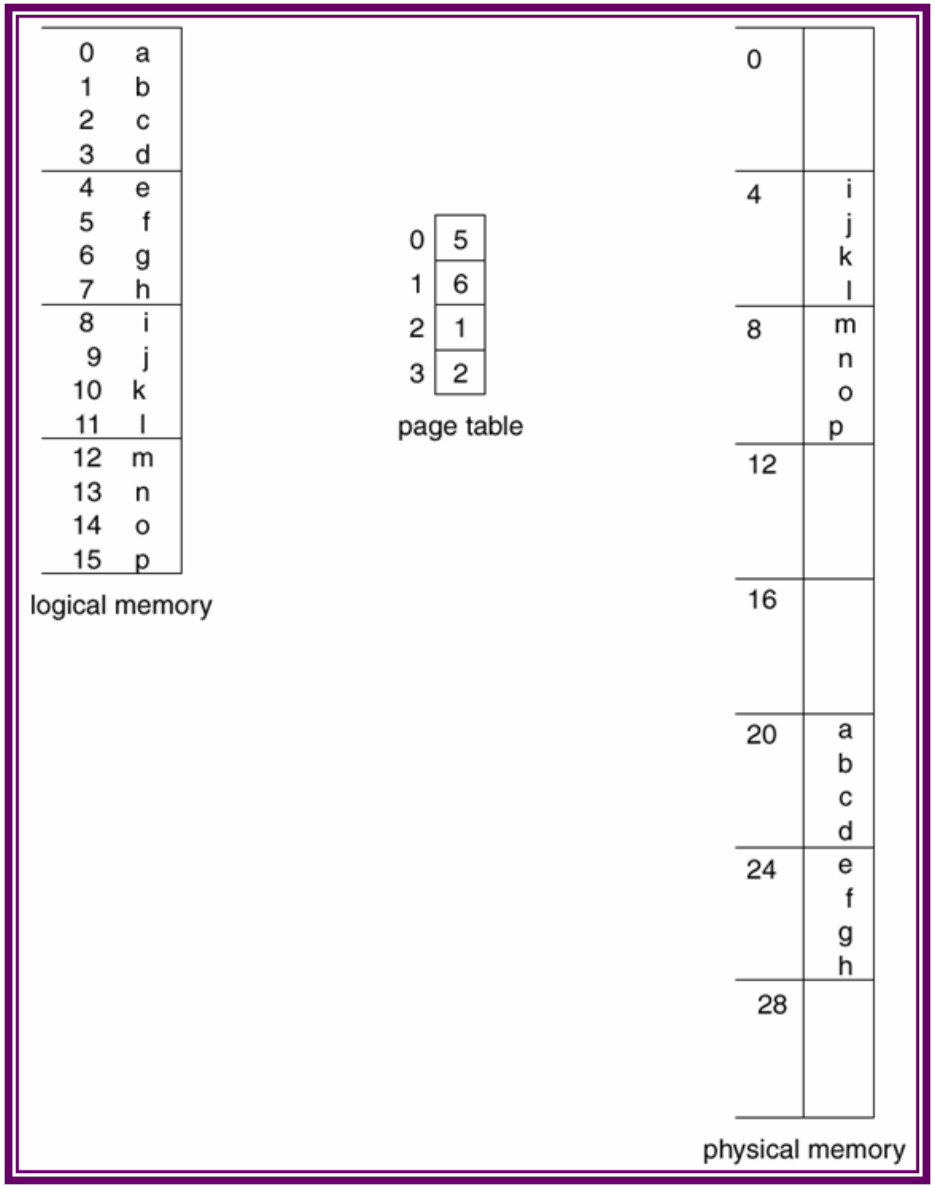
过程


CPU
logical address
p
d
physical address
f
d
p
f
page table
f
f0000 . . . 0000
f1111 . . . 1111
physical memory

例子（1/2）

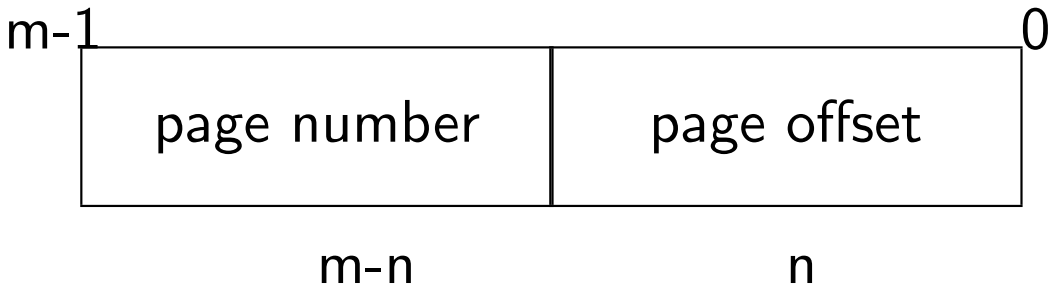

frame
number
page 0
page 1
page 2
page 3
logical
memory
0 1
1 4
2 3
3 7
page table
0
1 page 0
2
3 page 2
4 page 1
5
6
7 page 3
physical
memory

例子（2/2）


0 a
1 b
2 c
3 d
4 e
5 f
6 g
7 h
8 i
9 j
10 k
11 l
12 m
13 n
14 o
15 p
logical memory
0 5
1 6
2 1
3 2
page table
0
4 i j k l
8 m n o p
12
16
20 a b c d
24 e f g h
28
physical memory

地址划分

- page和frame：
  - 它们的大小必须相同，都是2的幂次方；
  - 具体由CPU体系结构决定，常见的有1K、4K、8K、4M等。
- 地址划分
  - 如果逻辑地址空间的大小为2^m，而page或frame的大小为2^n（$n < m$），那么逻辑地址的划分如图所示：



结论

- An important aspect of paging is the clear separation between user's view of memory and the actual physical memory,
  - The difference them is reconciled by address translation hardware;
  - This mapping is hidden from the user and is controlled by the OS.
- 由于是操作系统在控制地址映射，它必须
  - 记录系统物理内存的使用状况。一般使用一个称为frame table的数据结构来保存系统中每一个frame的状态；
  - 为每一个进程保存一个page table。

Questions

- Any questions?

实现

- Page table必须被保存在内存中。CPU中的两个寄存器记录了它的信息：
  - **Page-table base register (PTBR)** 保存了page table的地址；
  - **Page-table length register (PTLR)** 保存了page table的大小。
- 因此在分页中，每一个内存访问都需要两次内存操作，
  - 一次访问page table，一次访问内存数据。
  - 考虑到CPU访问内存的频率，这种地址转换成了系统性能的颈瓶。
- 为了提高地址转换效率，MMU中包含了一个高速缓存称为**translation look-aside buffers(TLBs)** 。

TLB

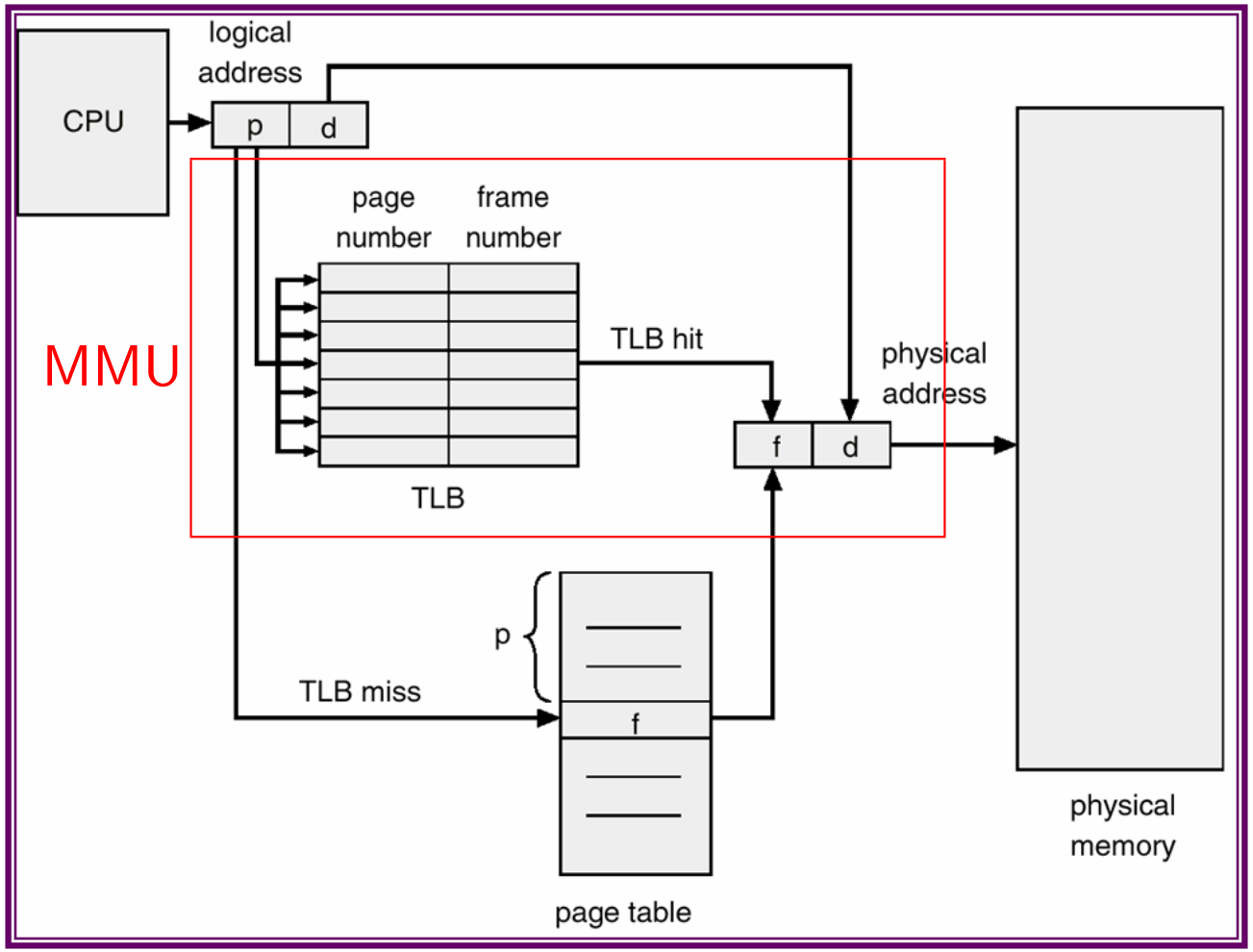
- TLB结构如图所示：

Page #	Frame #

TLB

- Address translation of A,
  - If A is in TLB (TLB hit), get frame # out;
  - Otherwise, get frame # from the page table in the memory.

Address translation with TLB



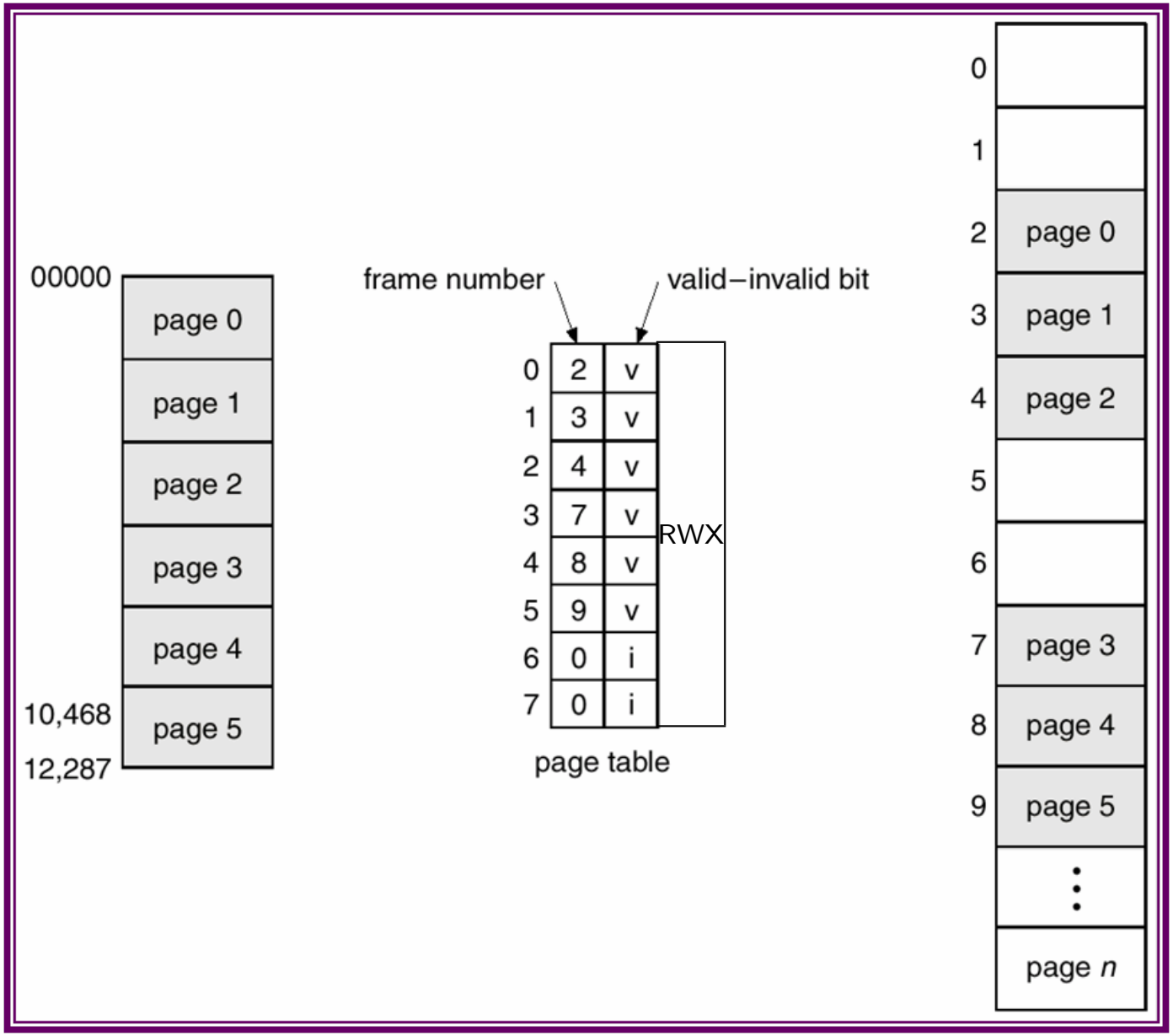
性能

- 假设
  - TLB lookup time $= \epsilon$,
  - Memory cycle time is 1 microsecond (10^{-6} second),
  - TLB hit ratio $= \alpha$, percentage of times that a page number is found in the TLB.
- Effective Access Time (EAT)
  - $EAT = (1+\epsilon)\alpha + (2+\epsilon)(1-\alpha) = 2+\epsilon-\alpha$

内存保护

- 在分页系统中，内存保护是以页为单位。
  - 保护信息通常都保存在PTE中，
  - 可以提供只读、读写和执行（RWX - Read, Write, eXecute）保护。
- 此外，不是所有的PTE都可以使用。因此，PTE中的一位表示该PTE是否可以使用（valid/invalid），
  - 仅当该位有效时，MMU才能用它进行地址转换，
  - 否则，MMU将通过异常向OS报告错误。

例子


00000
page 0
page 1
page 2
page 3
page 4
10,468
page 5
12,287
frame number
valid–invalid bit
0 2 v
1 3 v
2 4 v
3 7 v
4 8 v
5 9 v
6 0 i
7 0 i
RWX
page table
0
1
2 page 0
3 page 1
4 page 2
5
6
7 page 3
8 page 4
9 page 5
page n

Questions

- Any questions?

页表问题

- 假设
  - 逻辑地址空间大小：2^{32}，即4GB；
  - page和frame的大小：2^{12}，即4KB；
  - sizeof(PTE) = 4B。
- 一个页表要消耗多少内存？
  - $\frac{2^{32}}{2^{12}} \times 4 = 4MB$.
- 记住：每一个进程都需要一个页表，
  - 一个只有256MB内存的系统如何能同时运行几十个进程呢？

页表结构

- 事实上，很少的进程会使用整个地址空间（如32位机器上的4GB和64位机器上的$2^{64}B$）；
  - 也就是说，页表中大部分的PTE都是处于invalid状态。但是，
  - OS无法预测一个进程的页表大小，因此只能根据进程使用内存的情况动态地伸缩。
  - 显然线性的页表结构太简单，不能满足这个要求，我们需要新的页表结构。
- 现有的页表结构：
  - Hierarchical Page Tables
  - Hashed Page Tables
  - Inverted Page Tables

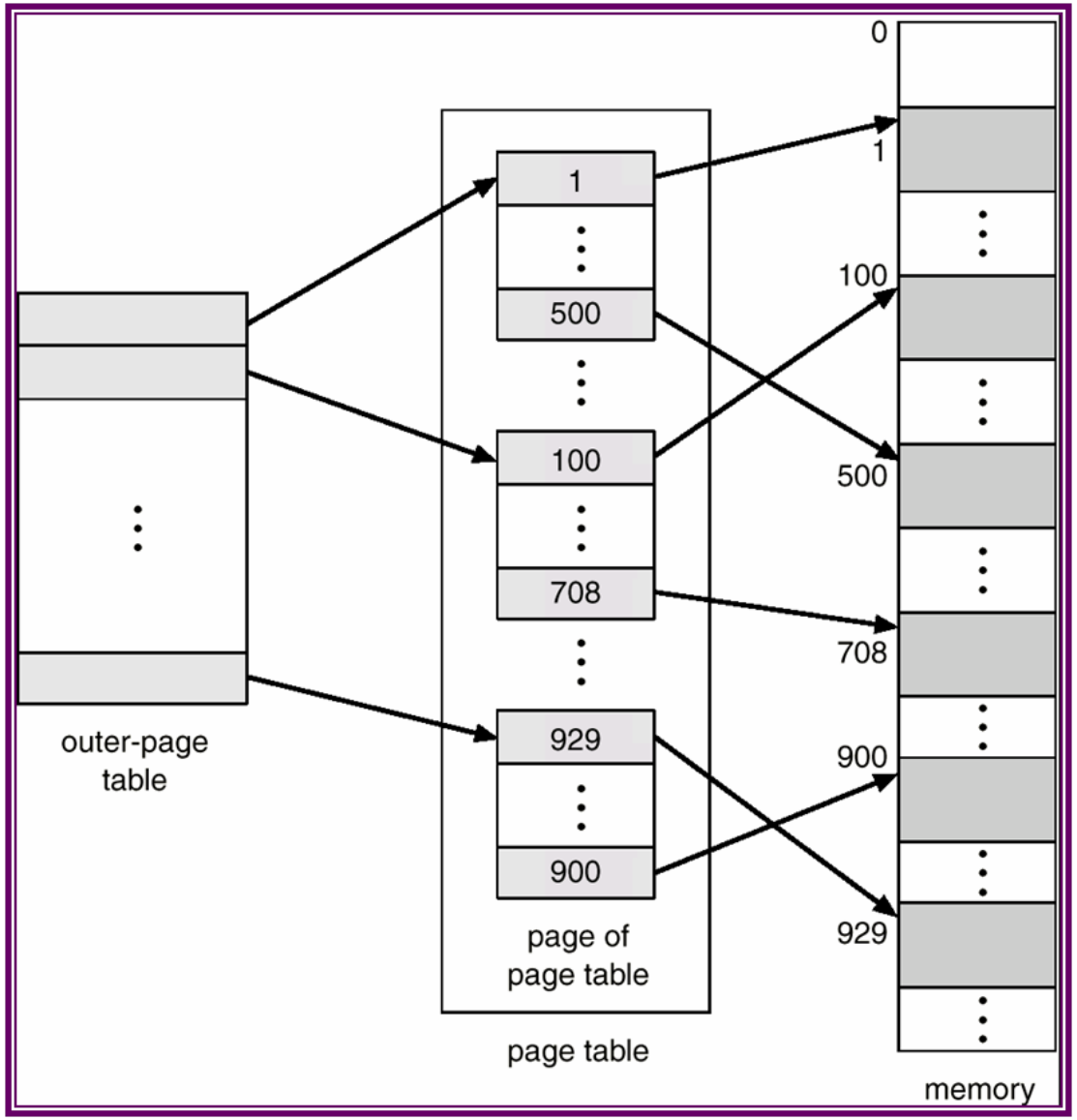
Hierarchical Page Tables（层次型页表）

- 这种方法采用“树”结构来组织页表，形成一个层次结构的页表；
  - 根据这颗树的深度可以分为：一、二、三级页表等；其中，一级页表（也称为单级页表）就退化成了线性页表。
  - 这里以32位逻辑地址、页面大小为4KB为例来讲解二级页表。

二级页表设计

- Paging the page table
  - 把一个巨大的线性页表分割成很多小的页表：
    - 每一个小页表包含1024项PTE，因此需要1024个小页表才能覆盖4GB的地址空间；
    - 又sizeof(PTE)=4B，所以一个小页表恰好占4KB，即一个页面。
  - 然后通过一个称为outer page table的表把这些小页表组织起来。
    - 显然，outer page table中包含1024项指向小页表的指针。
    - 通常称outer page table为**page directory**，其中的每一项称为**Page Directory Entry（PDE）**。
    - 由于sizeof(**PDE**)=4B，所以page directory也占4KB，即一个页面。

二级页表结构


0
1
100
500
708
900
929
1
500
100
708
929
900
outer-page table
page of page table
page table
memory

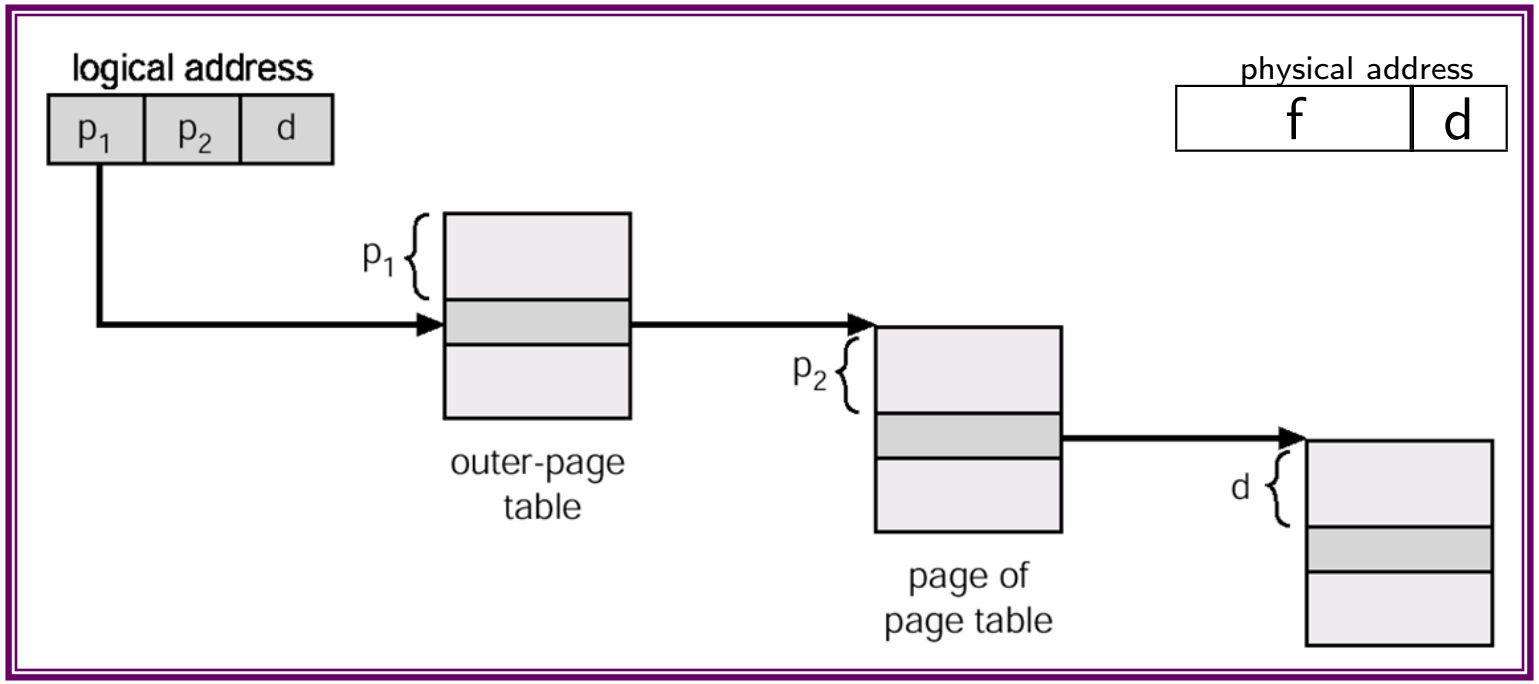
地址划分

- Remember that in single level page tables, a logical address is divided into:
  - a page number consisting of 20 bits,
  - a page offset consisting of 12 bits.
- Now since the page table is paged, the page number is further divided into:
  - a 10-bit page number,
  - a 10-bit page offset.

	page number	page offset
采用一级页表的逻辑地址：	p	d
	20	12

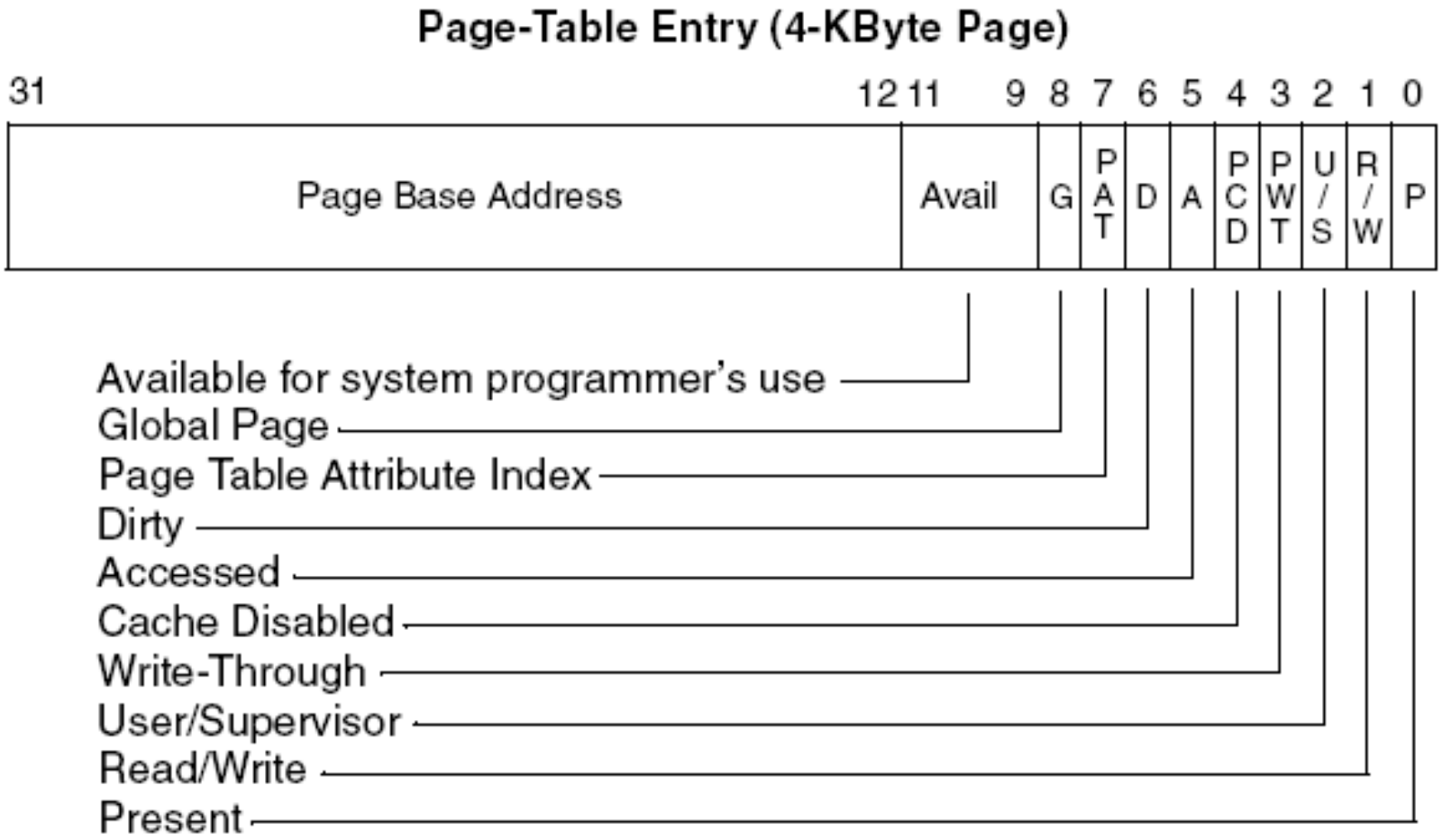
采用二级页表的逻辑地址：	p_1	p_2	d
	10	10	12

地址转换


logical address
p1
p2
d
physical address
f
d
p1
outer-page table
p2
page of page table
d

例子（1/3）

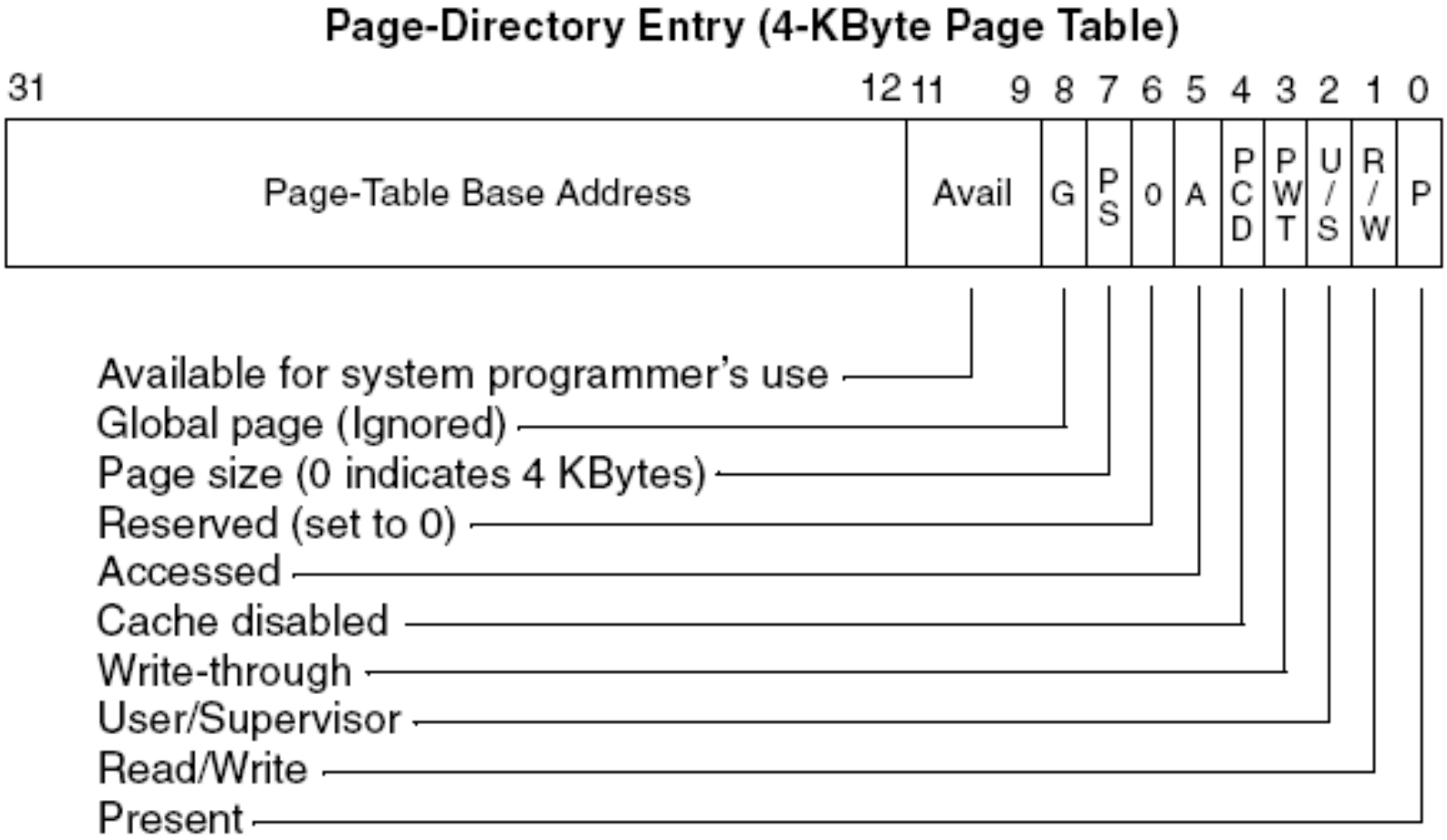
- Intel IA-32(x86，i386)中的PTE：



- 其中的“Present”为就是前面讲到的valid/invalid位。

例子（2/3）

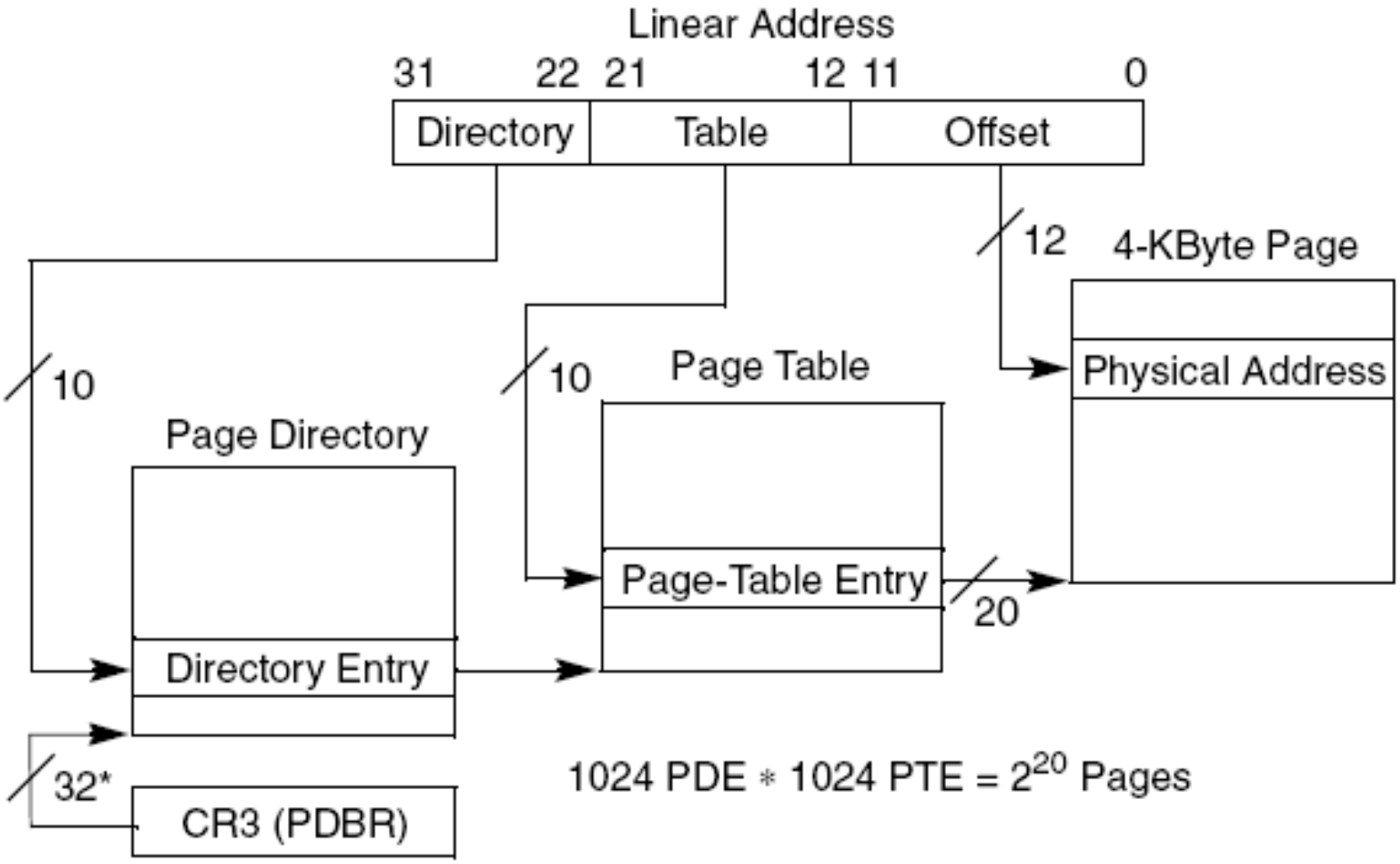
- Intel IA-32(x86，i386)中的PDE：



- 其中的“Present”为就是前面讲到的valid/invalid位。

例子（3/3）

- Intel IA-32(x86，i386)中的地址转换：



- 其中的“PDBR”全称为Page Directory Base Register。

Questions

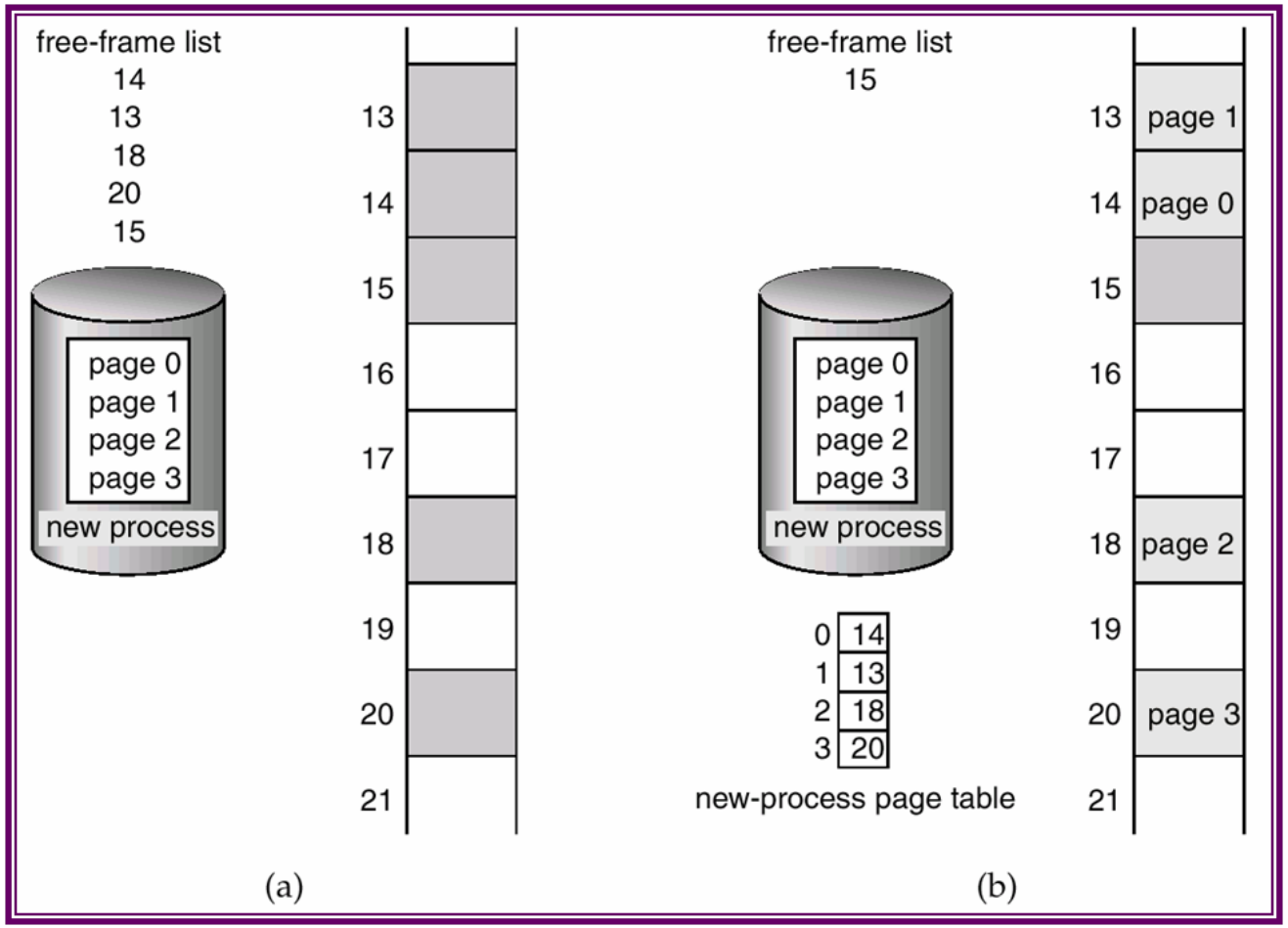
- Any questions?

Frame management(1/2)

- 前面介绍了页面的管理，以及页面到frame的映射。
  - 但是，最终保存数据的地方是frame，而不是页面。
- 操作系统需要管理系统中所有frame的分配和回收，
  - 当进程被创建时，操作系统要分配足够多的frame给它，并在该进程的页表中做相应的映射；
  - 当进程退出时，操作系统回收它所占用的frame。

Frame management(2/2)

- 最简单的方法是维护一个空闲的frame链表（free-frame list），如图所示：

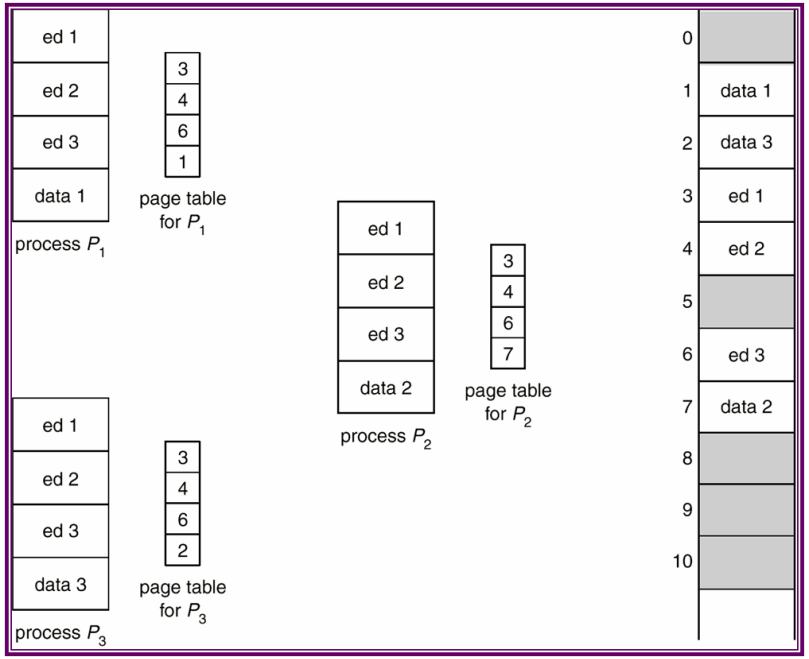


Questions

- Any questions?

页面共享

- 在分页系统中，显然应该以页面为单位进行内存共享。
- 例子
  - 假设系统中有三个用户同时运行某个编辑器在编辑各自的文件，显然编辑器的代码可以被共享，而各自的文件数据则是私有的。



Questions

- Any questions?

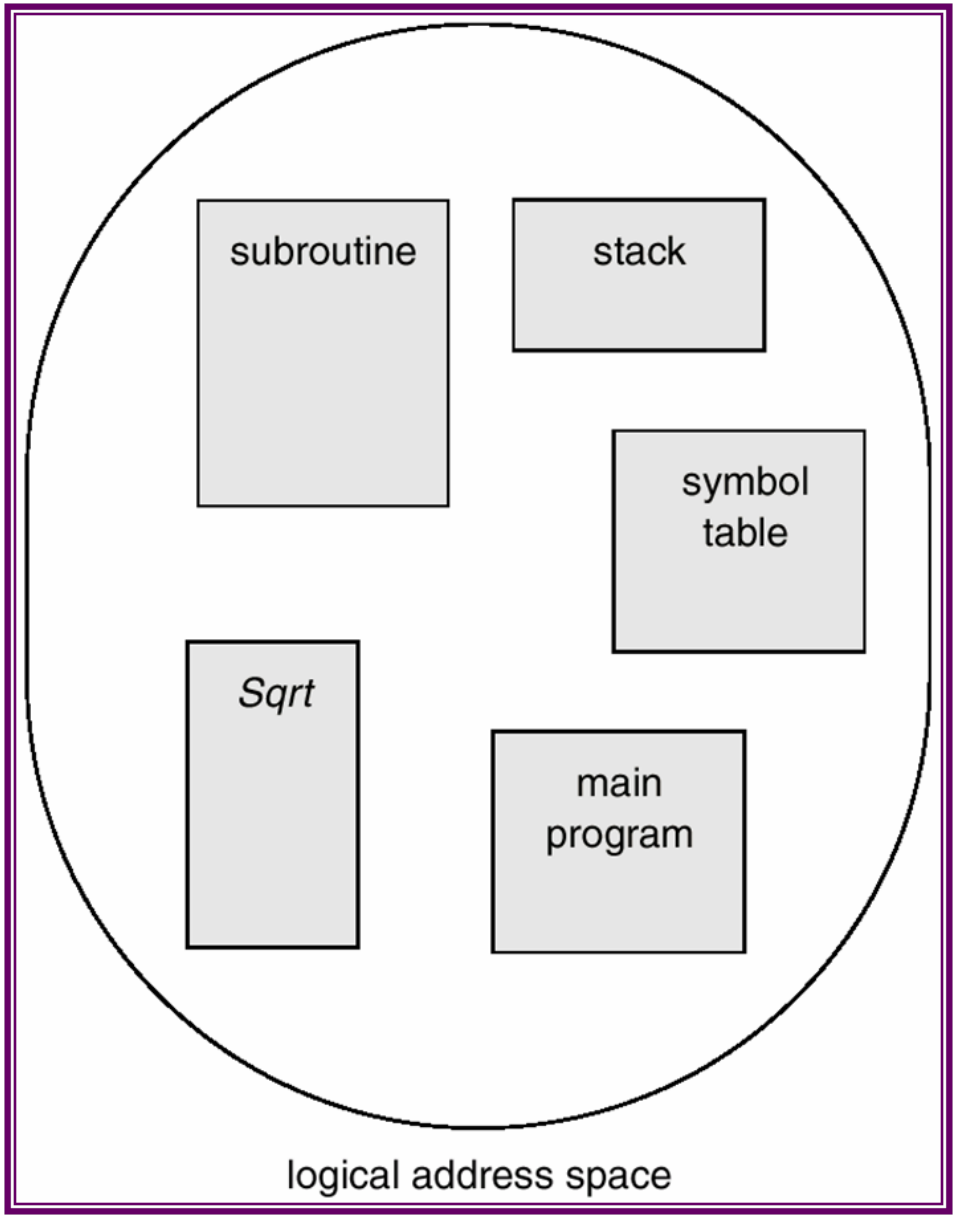
Outline

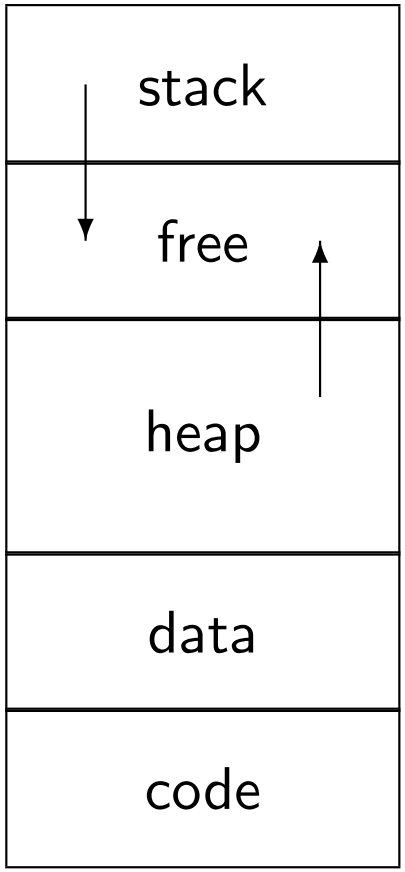
1. 内存管理介绍
2. 基本方法
3. 分页内存管理
4. 分段内存管理
5. 段页式内存管理

介绍

- Why segmentation?
  - Usually, users do not think of memory as a linear array of bytes or pages, some containing instructions and others contains data.
  - Rather, users prefer to view memory as a collection of variable-sized segments.
- 什么是分段？
  - 分段把进程的逻辑地址空间分成一个个大小不等的段，每一段集中了一种类型的数据，如代码、数据、栈等等。

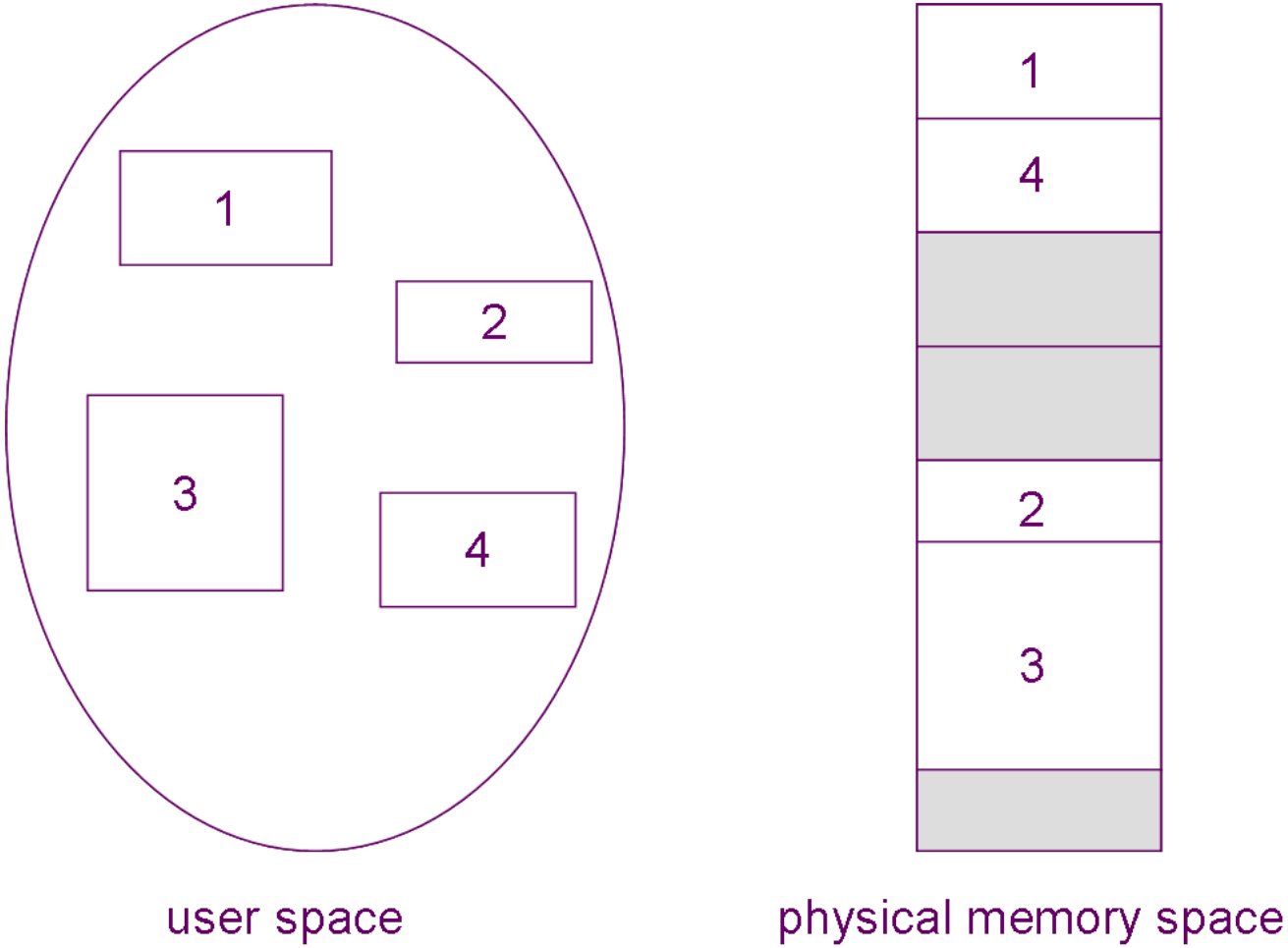
User's view of memory(1/2)





User's view of memory(2/2)

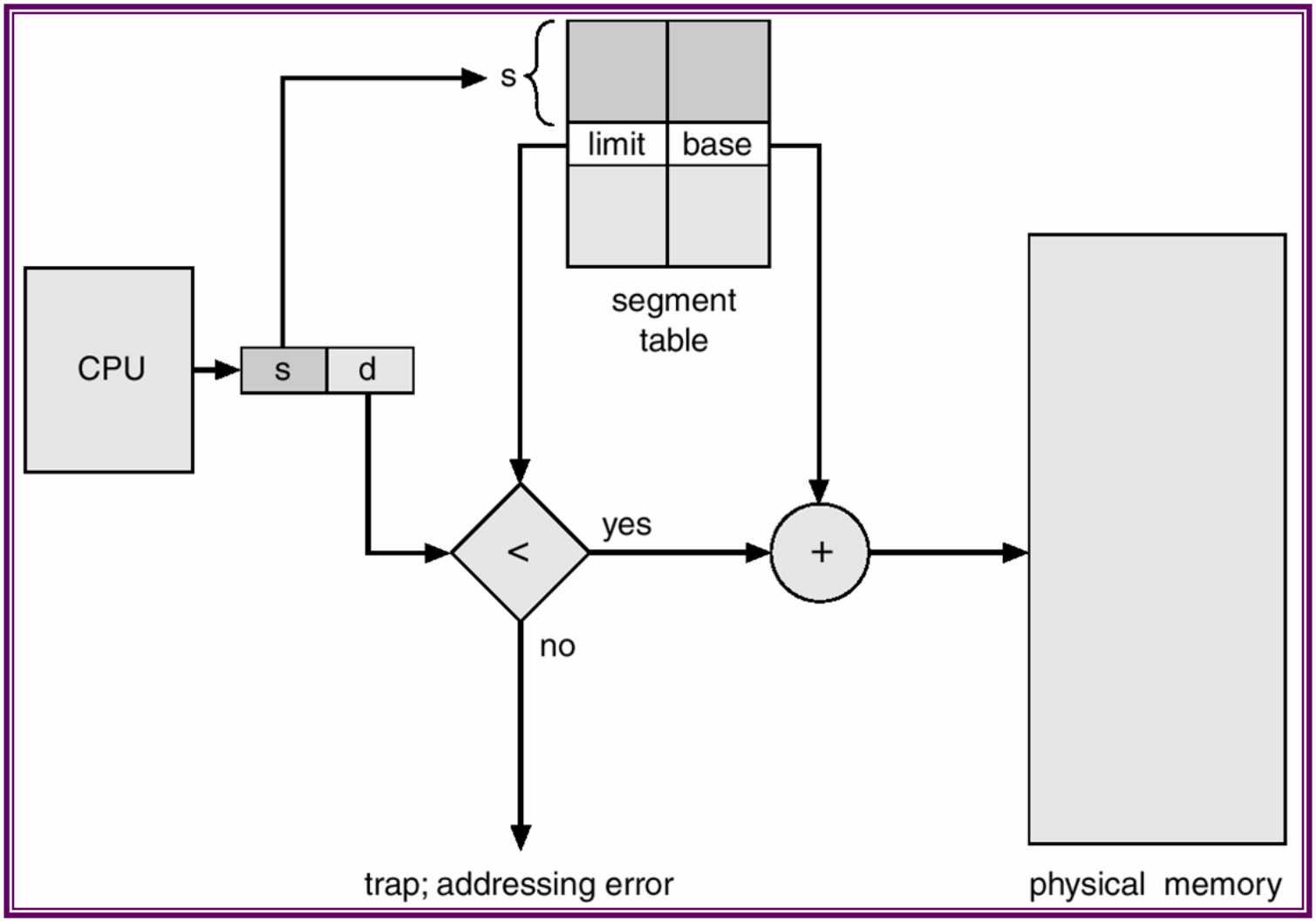
- 与分页系统一样，这些段在物理内存中也不一定是连续的，如图所示：



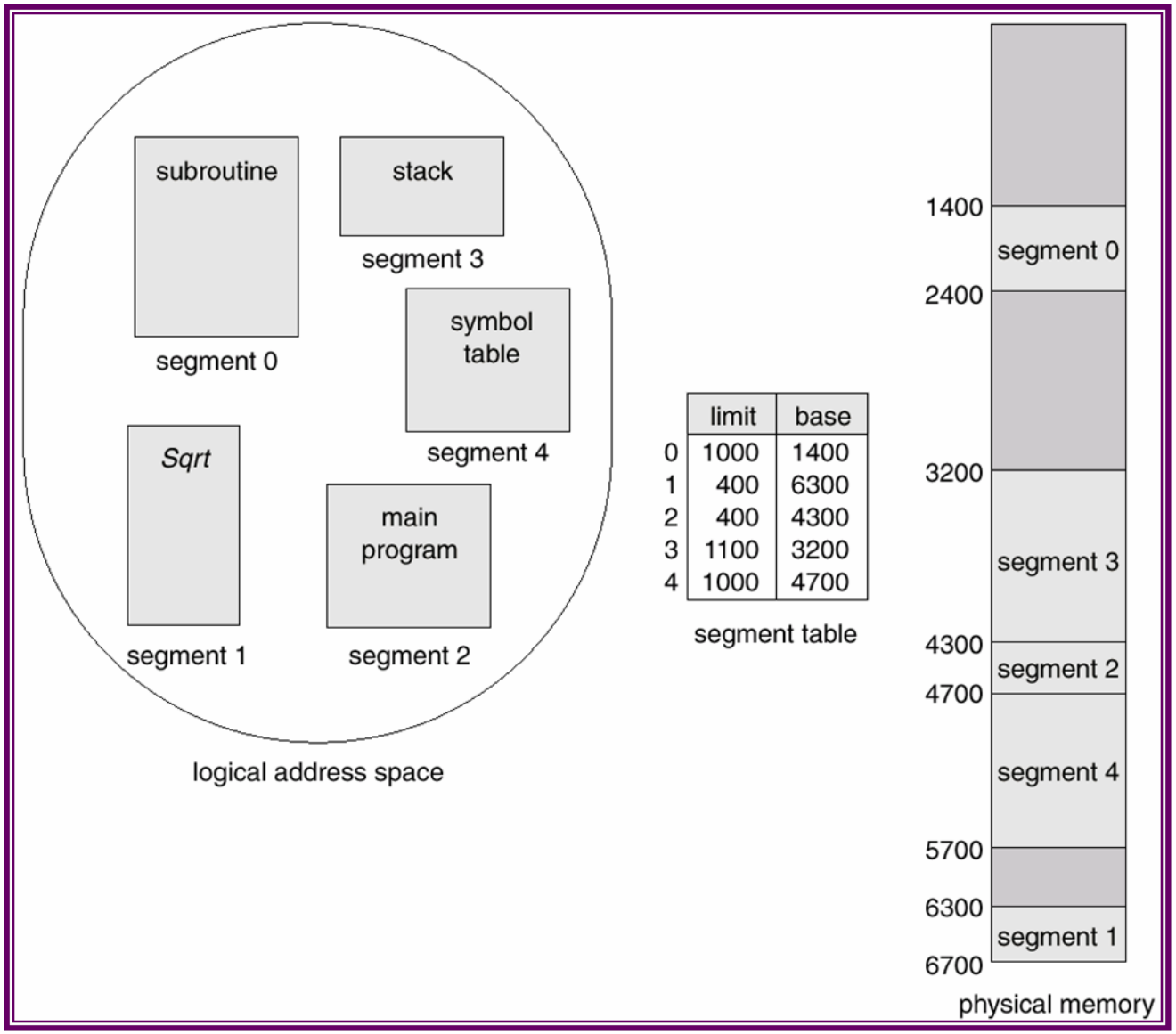
实现

- Logical address consists of two parts:
  <segment-number, offset>
- **Segment table** is used to map logical address to physical address. Each table entry has
  - **base** - contains the starting physical address where the segments reside in memory;
  - **limit** - specifies the length of the segment;
  - **protection bits (RWX)** of the segment.
- **Segment-table base register (STBR)** points to the segment table's location in the memory.
- **Segment-table length register (STLR)** indicates number of segments used by a process.

地址转换

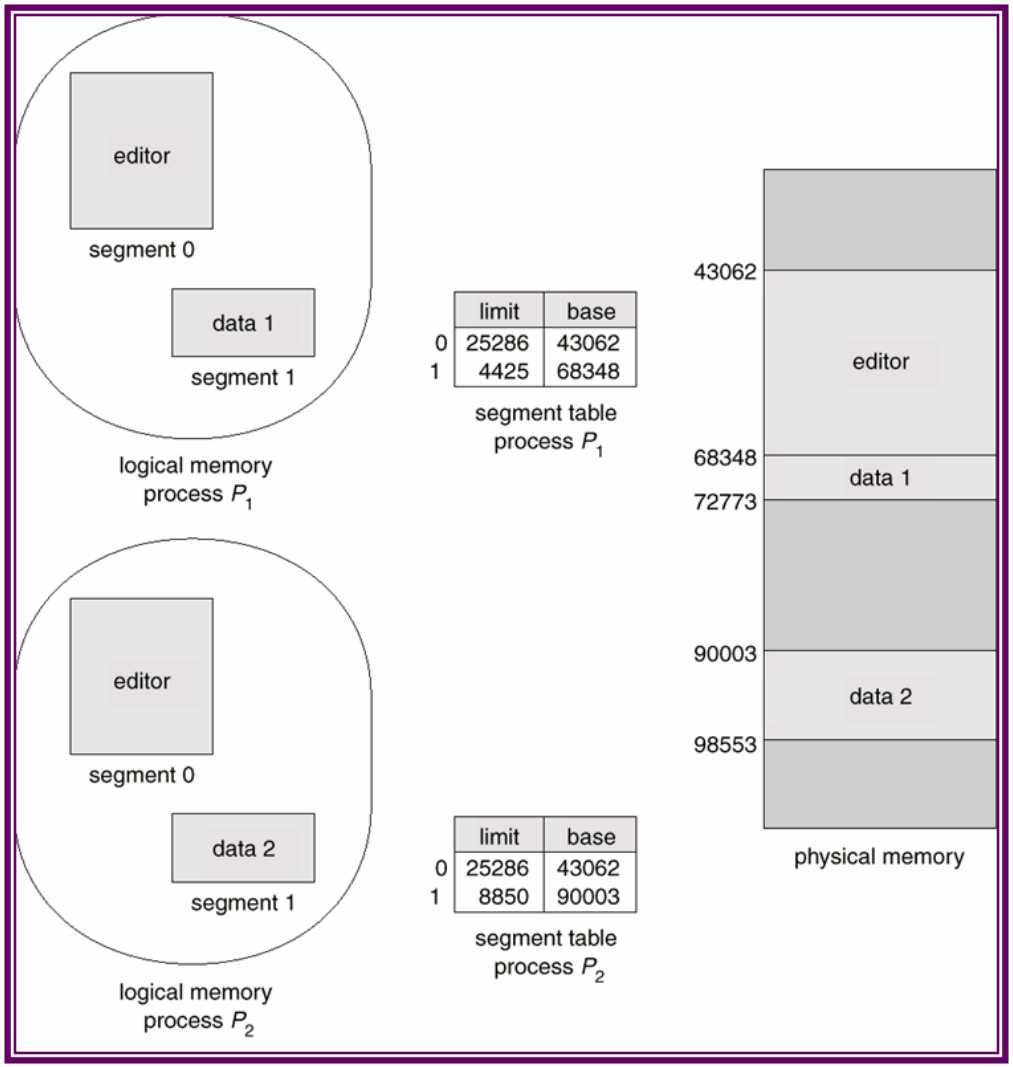

s
limit
base
segment
table
CPU
s
d
<
yes
+
no
trap; addressing error
physical memory

例子


subroutine
stack
segment 3
symbol table
segment 0
segment 4
Sqrt
main program
segment 1
segment 2
logical address space
limit base
0 1000 1400
1 400 6300
2 400 4300
3 1100 3200
4 1000 4700
segment table
1400
segment 0
2400
3200
segment 3
4300
segment 2
4700
segment 4
5700
6300
segment 1
6700
physical memory

内存保护与共享

- 当然，在分段系统中，内存的保护与共享以段为单位。如图所示：



结论

- 由于段的长度是根据需要变化的，因此分段会产生外部碎片。
  - 这个问题的严重程度与操作系统的内存分配算法和进程各个段的平均大小有关。
- 由于系统无法预测各个进程使用内存的状况，外部碎片很难控制。因此，单纯的分段系统目前很少使用。
- 各个硬件产商也不再支持单纯的分段系统。
  - INTEL 80386以前的CPU只支持分段，而80386以后（包括80386）的CPU则增加了分页支持。

Questions

- Any questions?

Outline

1. 内存管理介绍
2. 基本方法
3. 分页内存管理
4. 分段内存管理
5. 段页式内存管理

介绍

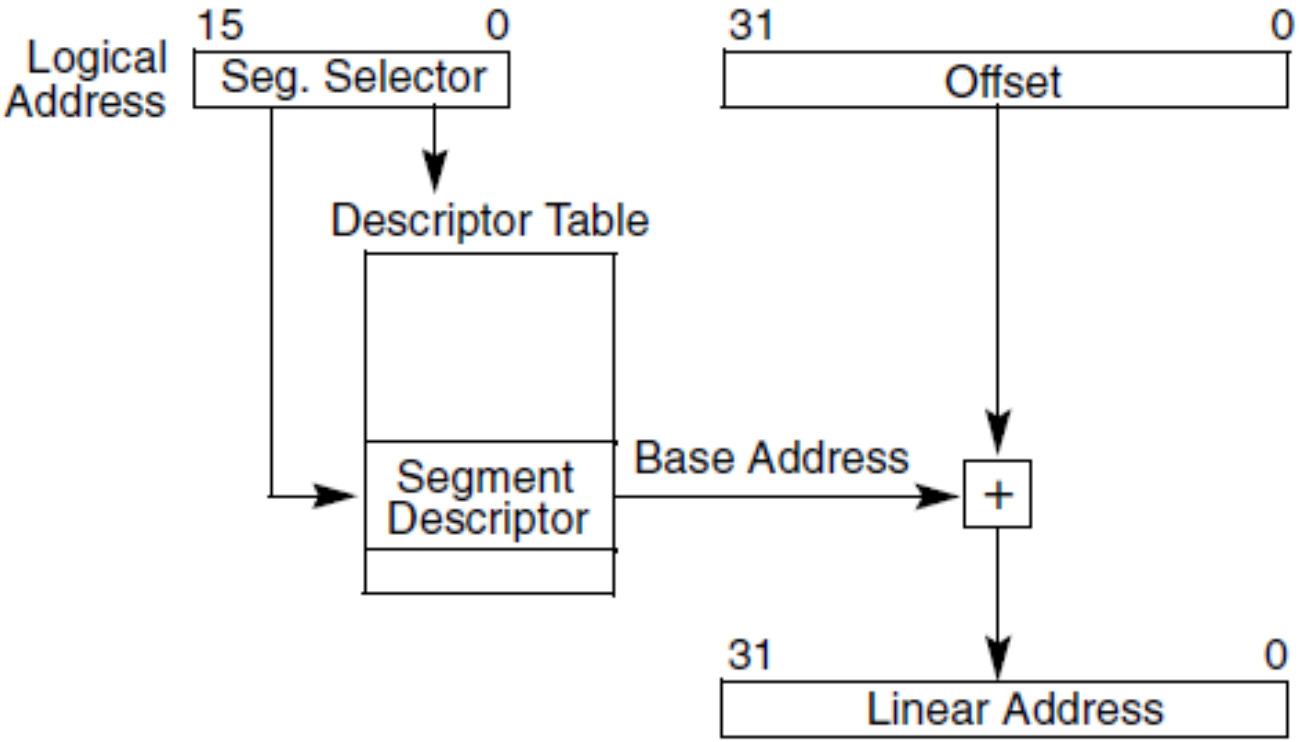
- 分页和分段系统有各自的优缺点，因此人们就考虑结合分页和分段，这就形成了段页式内存管理。
  - INTEL公司的IA-32（80386以后的CPU的统称，包含80386）是段页式内存管理的典范。
  - 因此，下面我们以IA－32为例来讲解段页式内存管理。

IA－32内存管理

- 在IA－32中，
  - 分段是必须的
    - 每个段最大可以为4GB。
    - 段表项被称为描述符（descriptor），相应地段表被称为描述符表（descriptor table）；
    - 逻辑地址<segment-number, offset>中的“segment-number”被称为段选择子（segment selector, 简称selector）。
- 而分页是可选的
  - 页面和frame大小为4KB，采用二级页表。

地址转换（1 / 3）

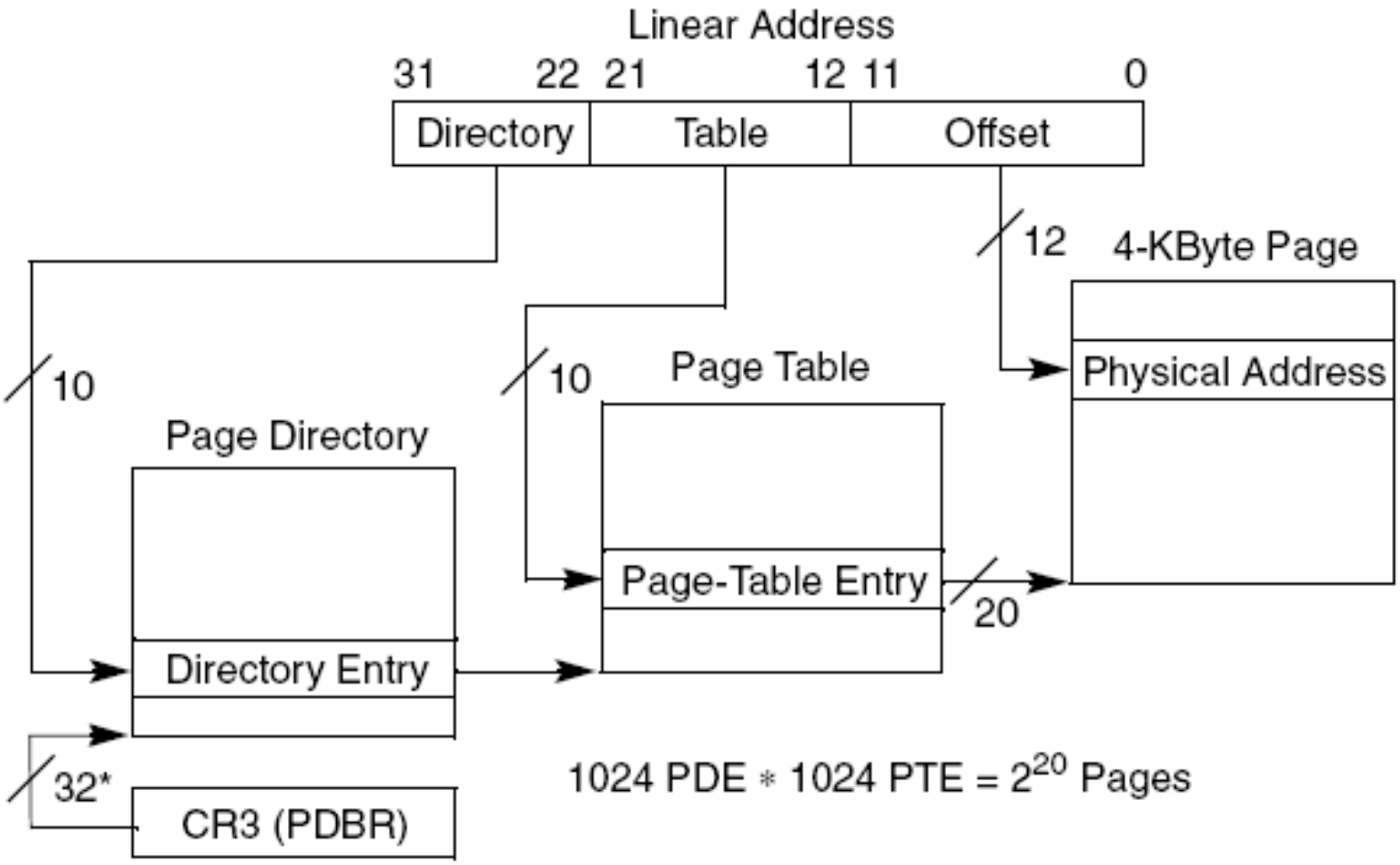
- 段页式内存管理的地址转换包括两个步骤：先分段，再分页。
  - 下图给出了分段部分的地址转换。



- 图中的“Linear Address”是逻辑地址经分段系统转换后的地址。
  - 如果CPU关闭了分页功能，它就直接作为物理地址发到地址总线；
  - 否则，它将继续被分页系统转换成物理地址。

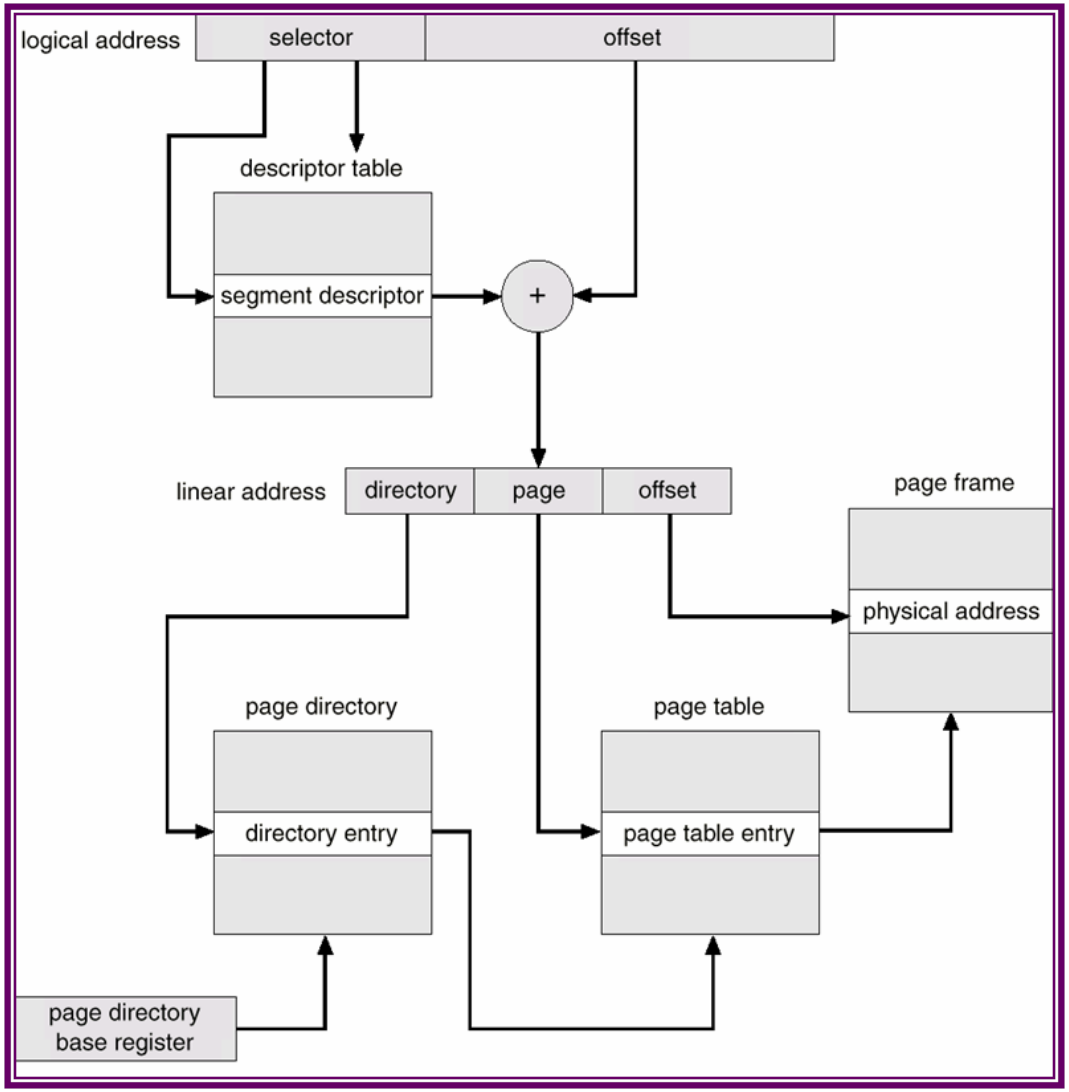
地址转换（2 / 3）

- 下图是前面见过的分页地址转换：

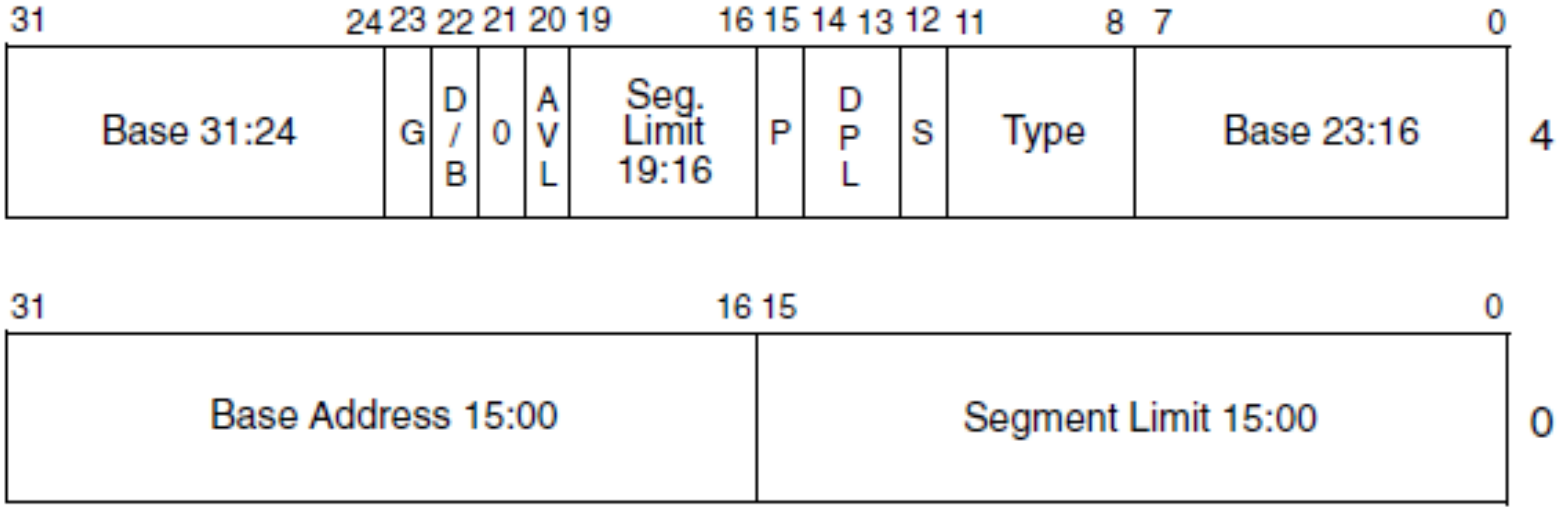


地址转换（3 / 3）

- Put them all together



段描述符（descriptor）



AVL — Available for use by system software
BASE — Segment base address
D/B — Default operation size (0 = 16-bit segment; 1 = 32-bit segment)
DPL — Descriptor privilege level
G — Granularity
LIMIT — Segment Limit
P — Segment present
S — Descriptor type (0 = system; 1 = code or data)
TYPE — Segment type

Questions

- Any questions?