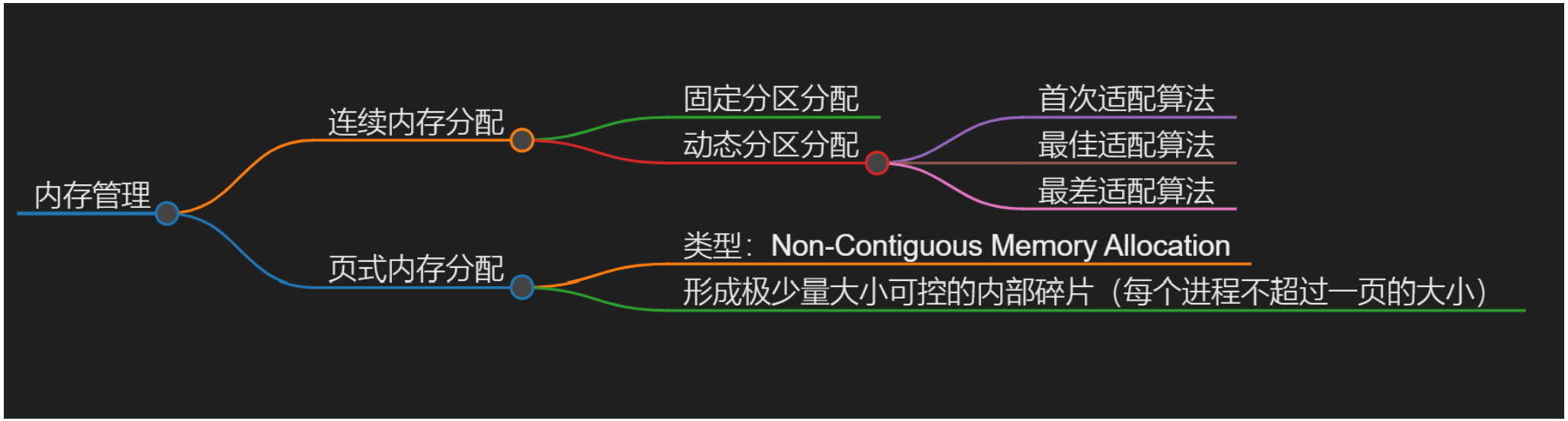
操作系统

L14 主存管理（下）

胡燕
大连理工大学 软件学院

0-概念回顾

内存分配


内存管理
连续内存分配
固定分区分配
动态分区分配
首次适配算法
最佳适配算法
最差适配算法
页式内存分配
类型：Non-Contiguous Memory Allocation
形成极少量大小可控的内部碎片（每个进程不超过一页的大小）

填空题 5分

设置

若系统采取可变分区分配算法进行内存分配，且当前空闲分区链表长为n。当前一个进程退出，动态分区分配算法会把其所用的内存区域回收。如果该进程地址区域两侧都是空闲区，请问在完成回收后，空闲分区链表的长度会更新为 （ [填空1] ）。

作答

填空题 5分

设置

在一分页系统中，逻辑地址的长度为16位,页面大小设定为4096。现有一逻辑地址2F6AH，且逻辑页0、1、2分别被映射到5、10、11。请问该逻辑地址对应的物理地址是（ [填空1] ）。

作答

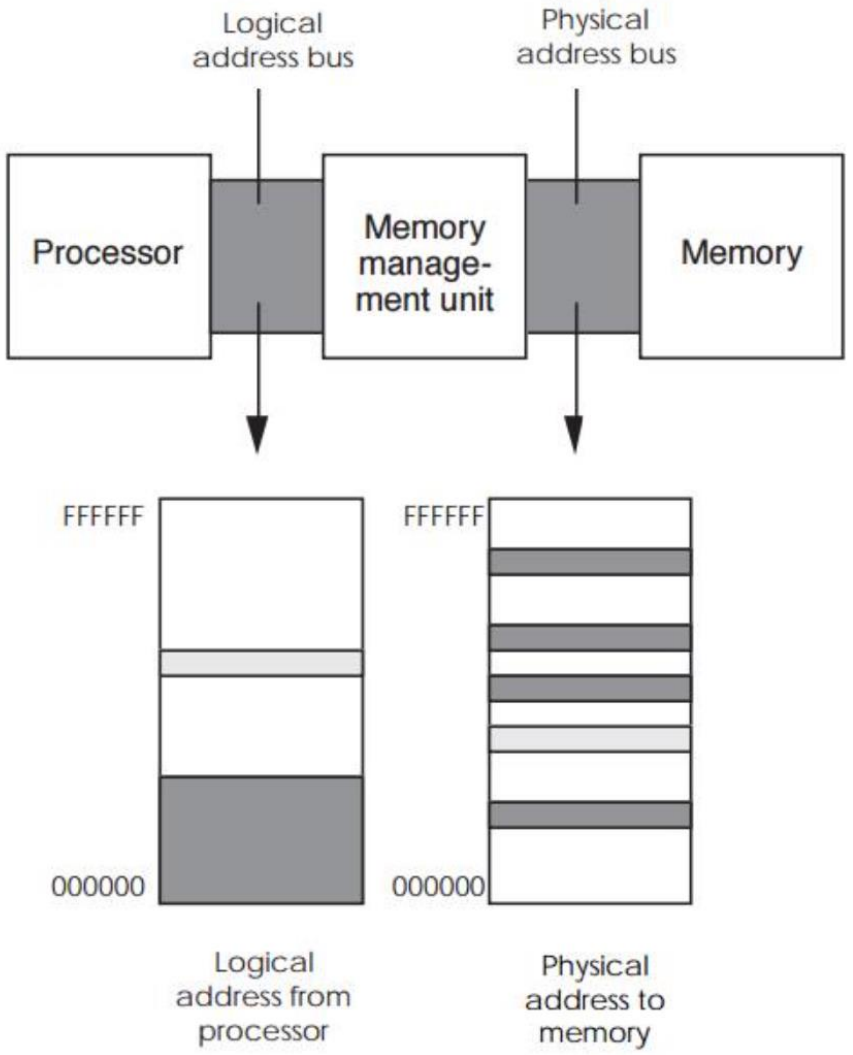
分页硬件

Paging Hardware

01

1-分页的硬件支持

分页硬件基本结构



https://www.brainkart.com/article/Memory-management-address-translation_7703/

为什么需要分页硬件

做人要“务虚”，做事要“务实”

1、做人务虚，谦虚低调 **2、做事务实，脚踏实地**

为了有务虚的底气，得有硬实力支撑

Linux为每个进程营造的逻辑地址空间大小：

32位处理器：4GB

64位处理器：256TB (用了48位)

实际的物理内存大小远小于此

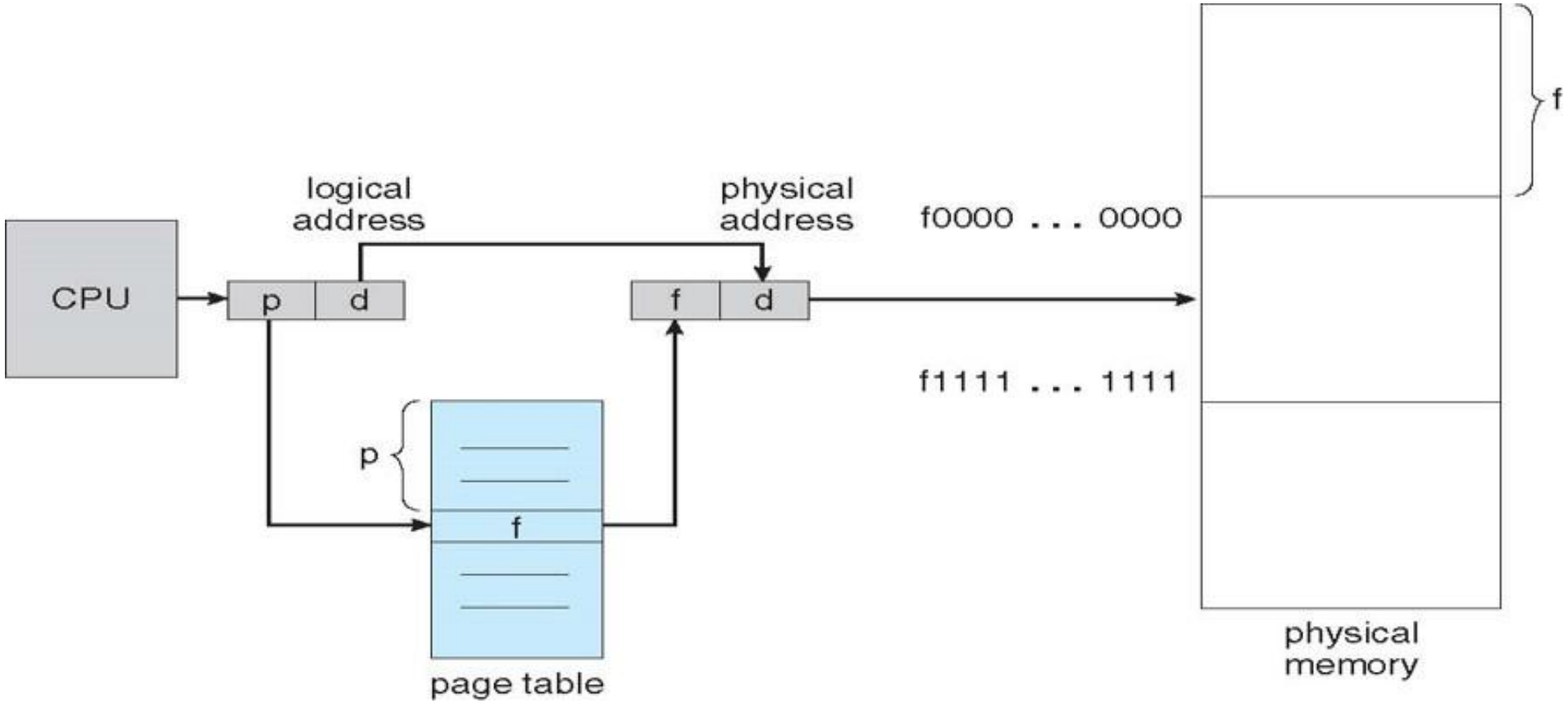
为了使编程人员有足够的虚存空间，可以在编程时不受物理内存空间的局限，需要内存管理硬件支持

↓

内存管理硬件（Memory Management Unit, MMU）

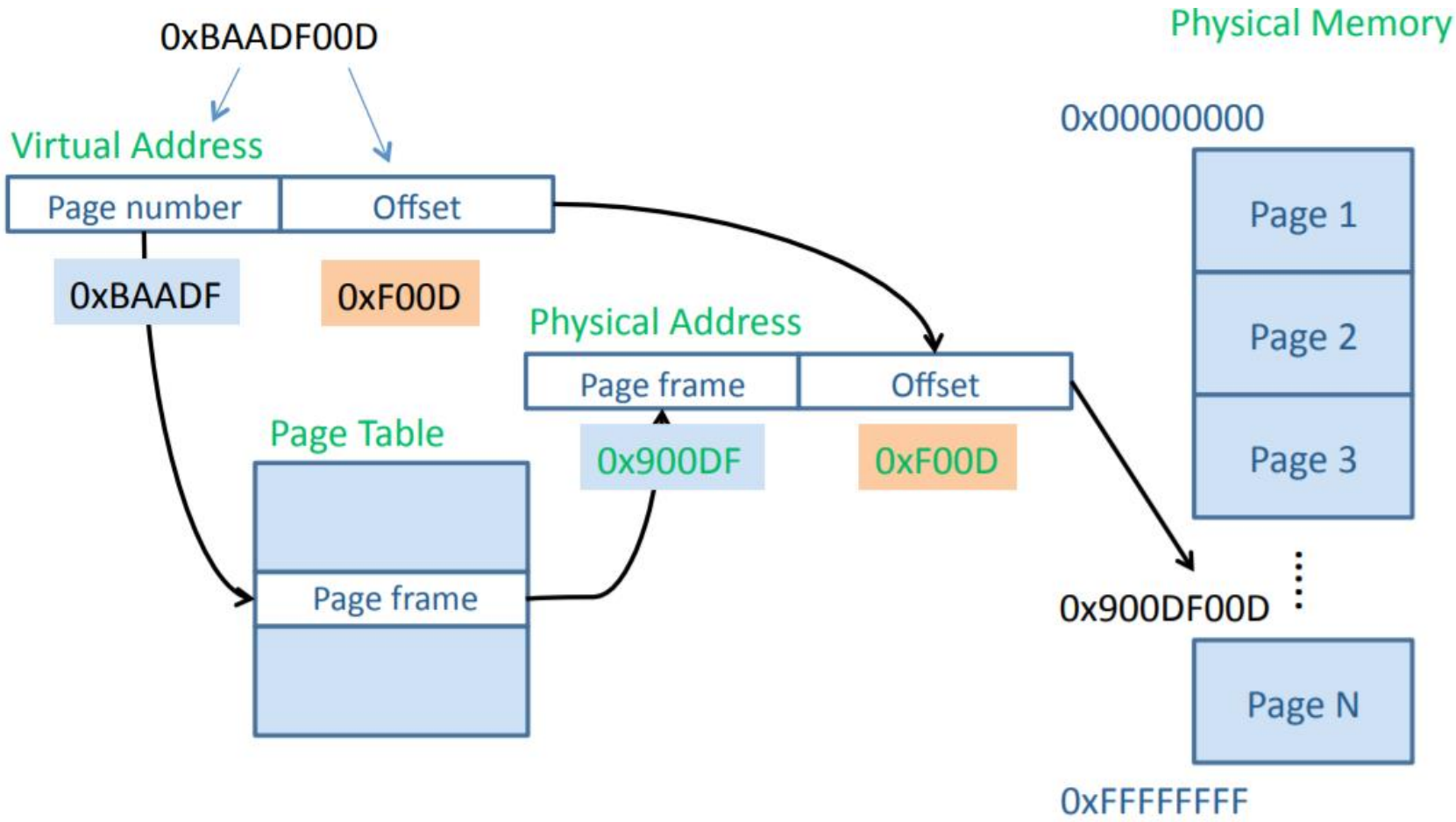
分页硬件基本结构

分页硬件的逻辑示意图



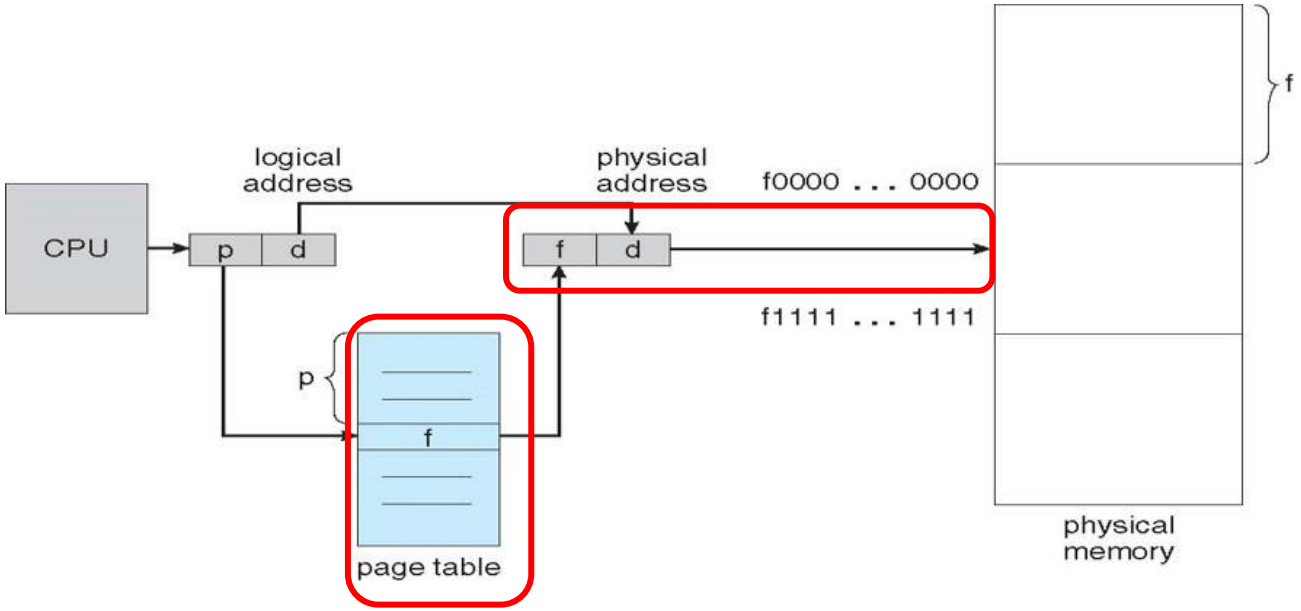
分页硬件基本结构

分页硬件的地址翻译示意：



分页硬件基本结构

分页机制引入的访存开销



页表：存储于主存 → 基于逻辑页号p查表，涉及1次内存访问

得到物理页号f后，将f与页内偏移组合得到物理地址，进行的第2次内存访问

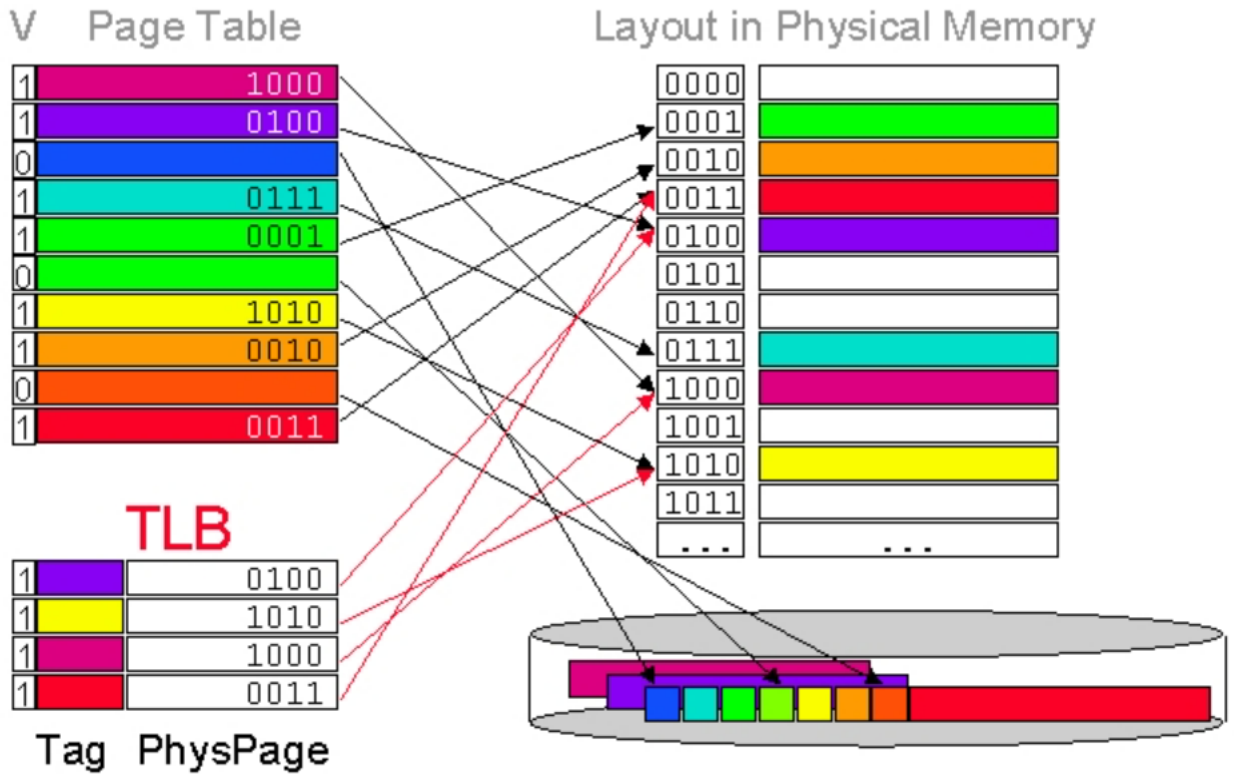
原本1次物理内存访问，因引入分页机制，需要2次内存访问才能完成：效率减半！

TLB：降低分页管理开销的硬件缓存方案

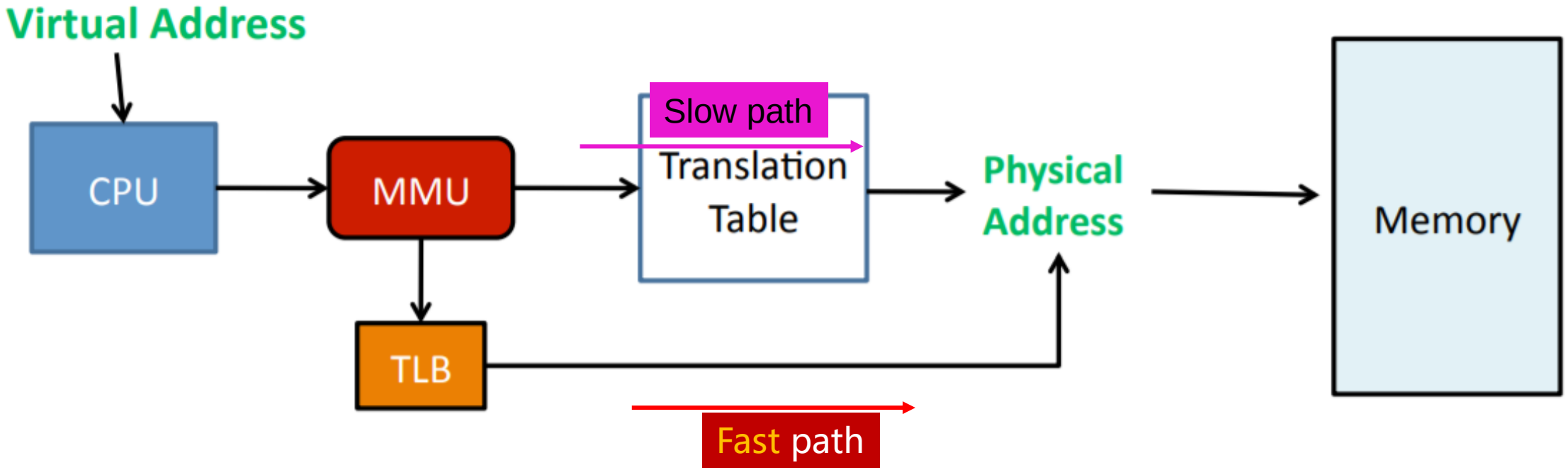
为何引入TLB：引入页表，多了一次内存访问（访存开销翻倍）

解决思路：缓存

A translation lookaside buffer (TLB) is a type of memory cache that stores recent translations of virtual memory to physical addresses to enable faster retrieval. This high-speed cache is set up to keep track of recently used page table entries (PTEs). Also known as an address-translation cache, a TLB is a part of the processor's memory management unit (MMU).

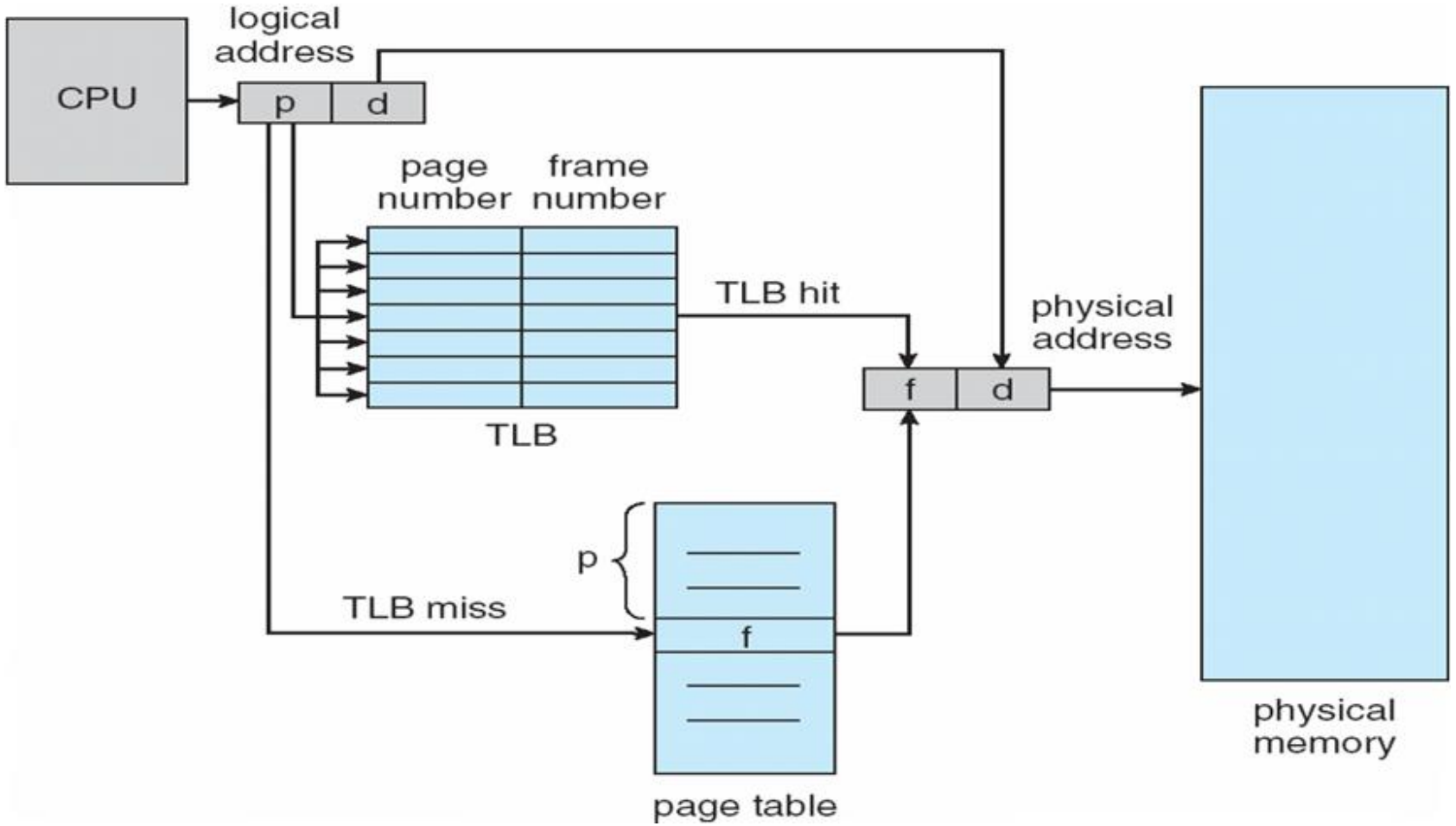


TLB：降低分页管理开销的硬件缓存方案



目标：多走fast path,尽量少走或不走slow path

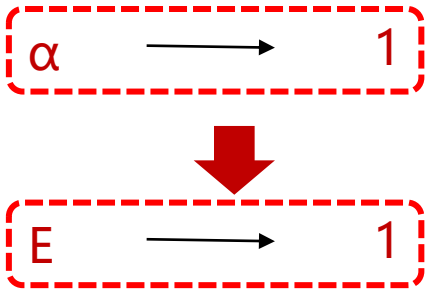
增加了TLB支持的分页硬件示意图



增加了TLB支持的分页硬件：访存效率分析

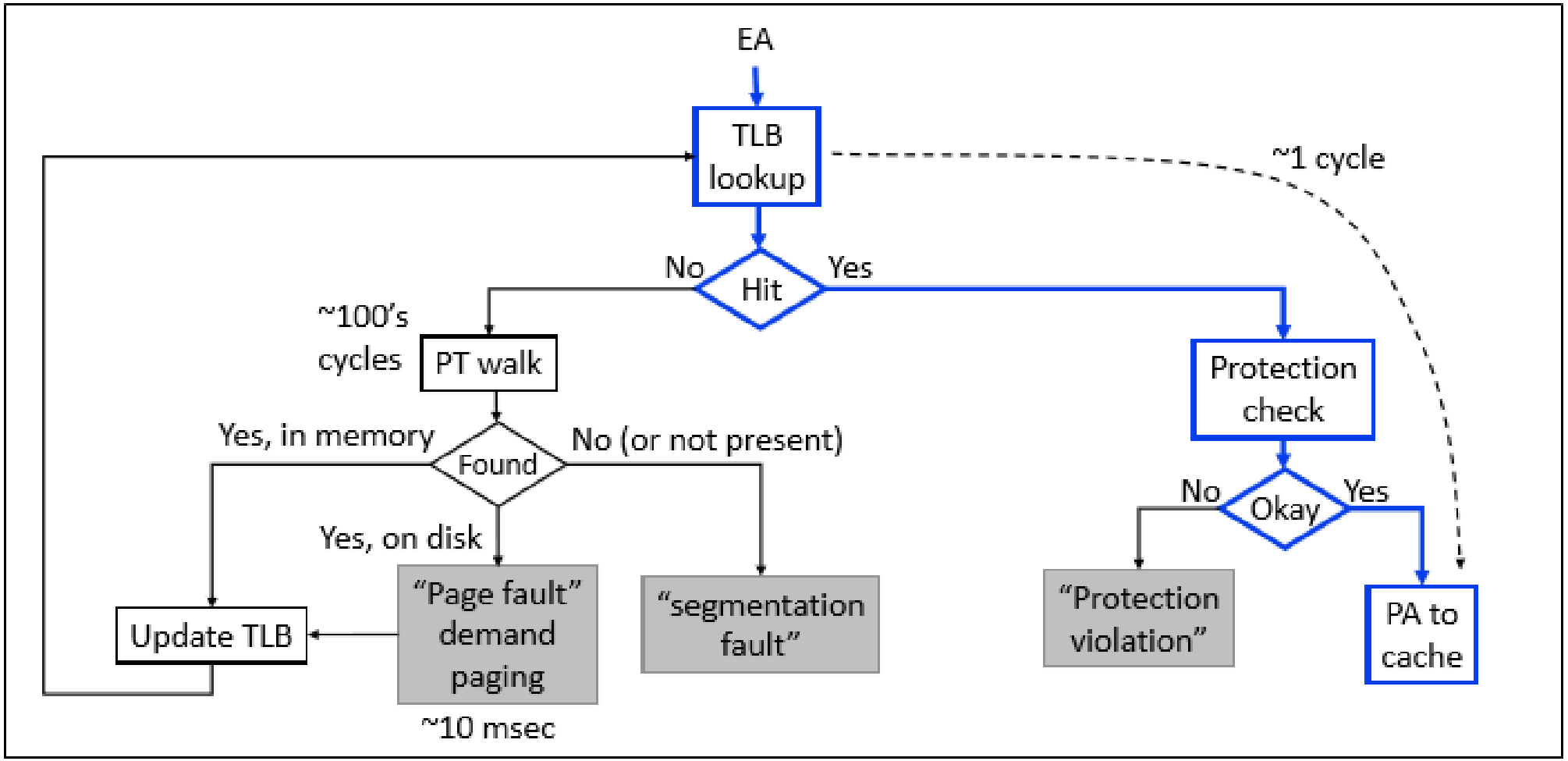
假设单次内存访问的时间为1
TLB的访问时间为ε（这是个远小于1的值）
TLB的页表项访问命中率= α

内存有效访问时间 E
= (1 + ε) α + (2 + ε)(1 – α)
= 2 + ε – α

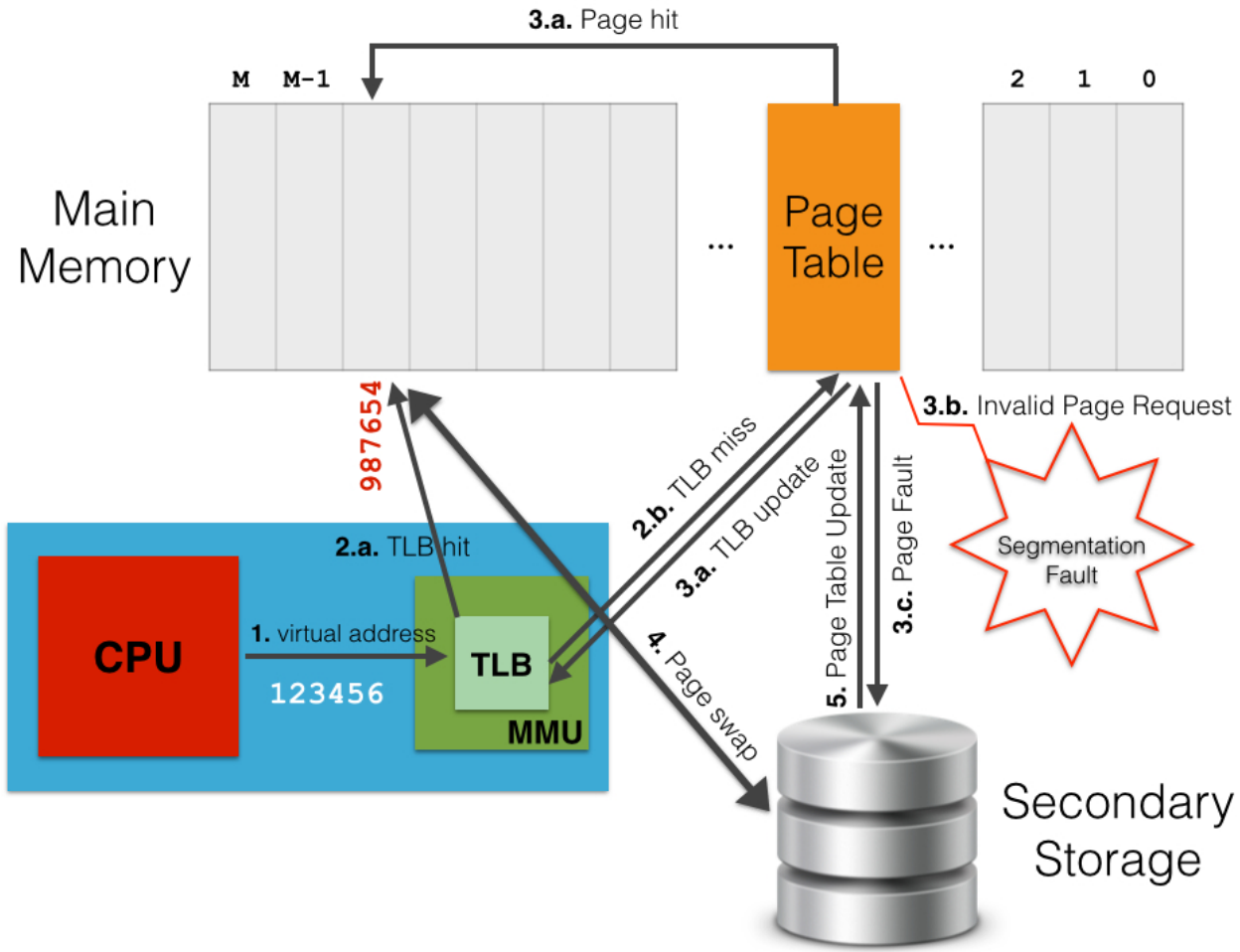


TLB

增加了TLB支持的分页硬件：一次分页地址的完整寻址流程



增加了TLB支持的分页硬件：一次分页地址的完整寻址流程



增加了TLB支持的分页硬件：TLB内容样例

Valid	Virtual page	Modified	Protection	Page frame
1	140	1	RW	31
1	20	0	R X	38
1	130	1	RW	29
1	129	1	RW	62
1	19	0	R X	50
1	21	0	R X	45
1	860	1	RW	14
1	861	1	RW	75

页表结构

Page Table Structure

02

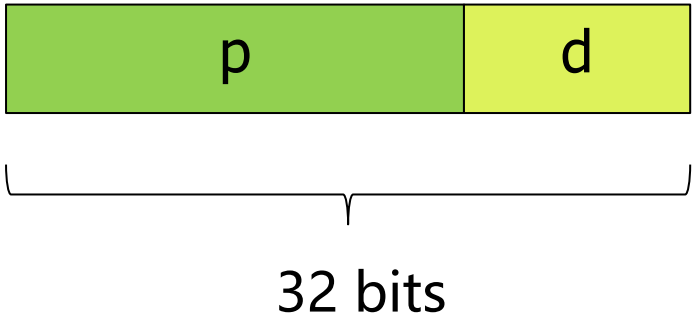
单级页表的不足

页表开销：32位，单级页表

CPU的地址位：32位

按字节编址，32位的逻辑地址空间=2^32字节=4GB

假设页的大小设为4KB，d=?



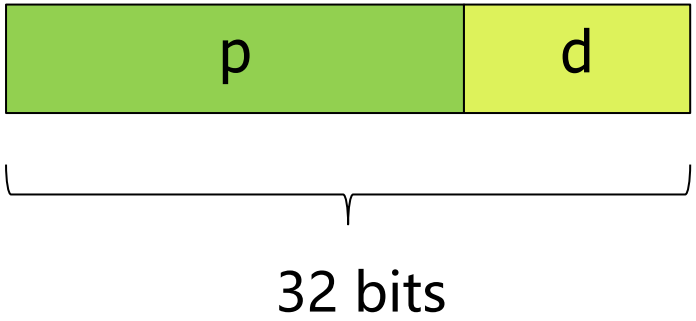
单级页表的不足

页表开销：32位，单级页表

CPU的地址位：32位

按字节编址，32位的逻辑地址空间=2^32字节=4GB

假设页的大小设为4KB，d= 12



单级页表的不足

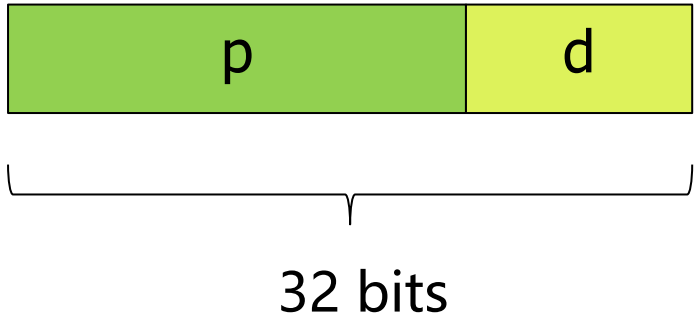
页表开销：32位，单级页表

CPU的地址位：32位

按字节编址，32位的逻辑地址空间=2^32字节=4GB

假设页的大小设为4KB，d= 12

逻辑页号位数p = ?



单级页表的不足

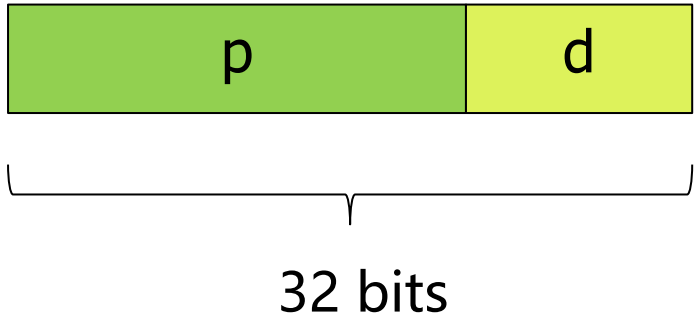
页表开销：32位，单级页表

CPU的地址位：32位

按字节编址，32位的逻辑地址空间=2^32字节=4GB

假设页的大小设为4KB，d= 12

逻辑页号位数p = 20



单级页表的不足

页表开销：32位，单级页表

CPU的地址位：32位

按字节编址，32位的逻辑地址空间=2^32字节=4GB

假设页的大小设为4KB，d= 12

逻辑页号位数p = 20

页表项数= 2^20　　每个页表项占据字节数= 4

页表需占用的空间大小= ？

单级页表的不足

页表开销：32位，单级页表

CPU的地址位：32位

按字节编址，32位的逻辑地址空间=2^32字节=4GB

假设页的大小设为4KB，d= 12

逻辑页号位数p = 20

页表项数= 2^20　　每个页表项占据字节数= 4

页表需占用的空间大小= 4MB

单级页表的不足

页表开销：64位，单级页表

CPU的地址位：64位

按字节编址，64位的逻辑地址空间=2^64字节=16PB

假设页的大小设为4KB，d= 12

逻辑页号位数p = 52

页表项数= 2^52　　每个页表项占据字节数= 8

页表需占用的空间大小= ？

单级页表的不足

页表开销：64位，单级页表

CPU的地址位：64位

按字节编址，64位的逻辑地址空间=2^64字节=16PB

假设页的大小设为4KB，d= 12

逻辑页号位数p = 52

页表项数= 2^52　　每个页表项占据字节数= 8

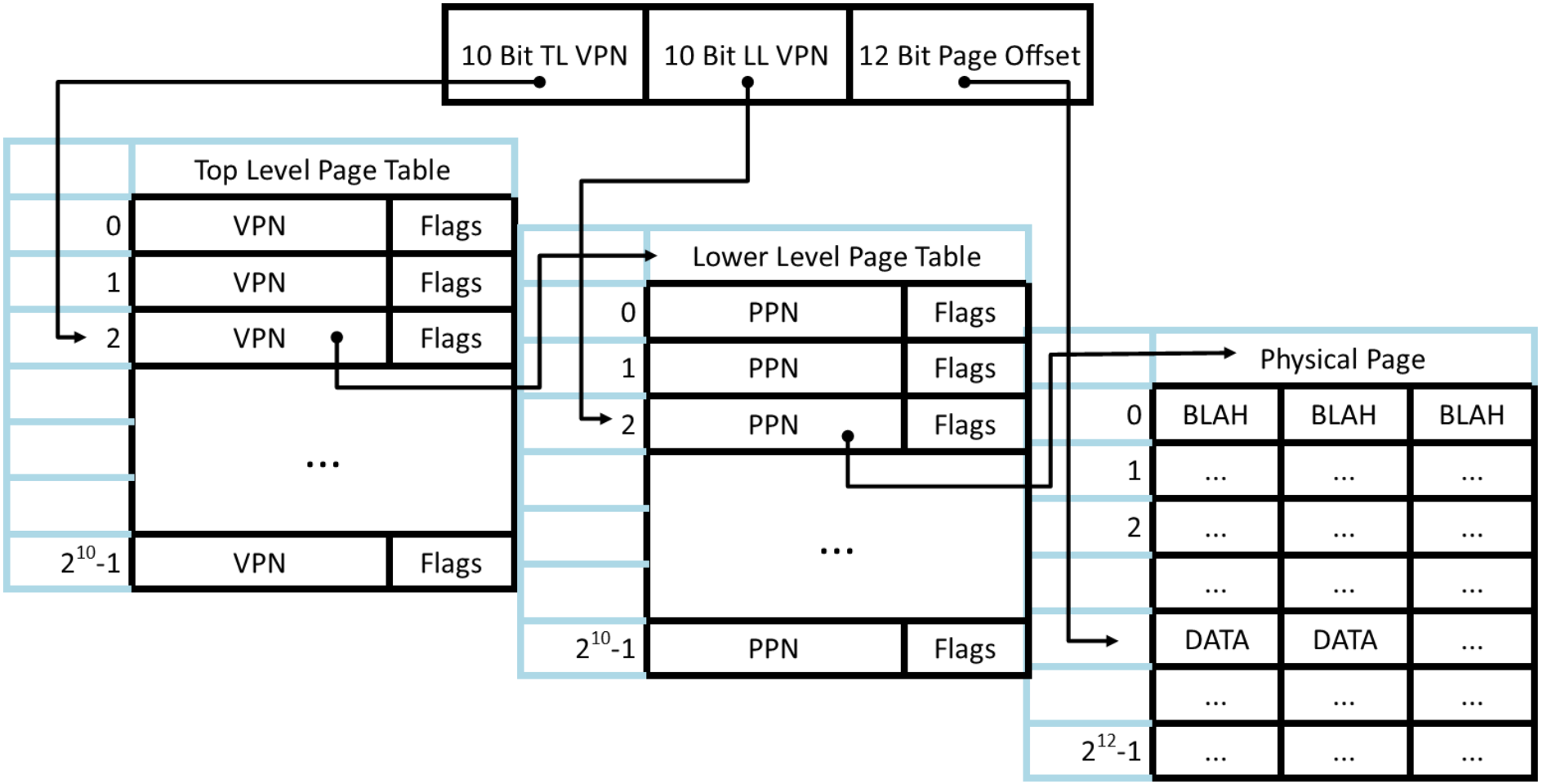
页表需占用的空间大小= 32TB

不足：页表占据空间过大

多级页表

处理大地址空间的页表方案1：多级页表

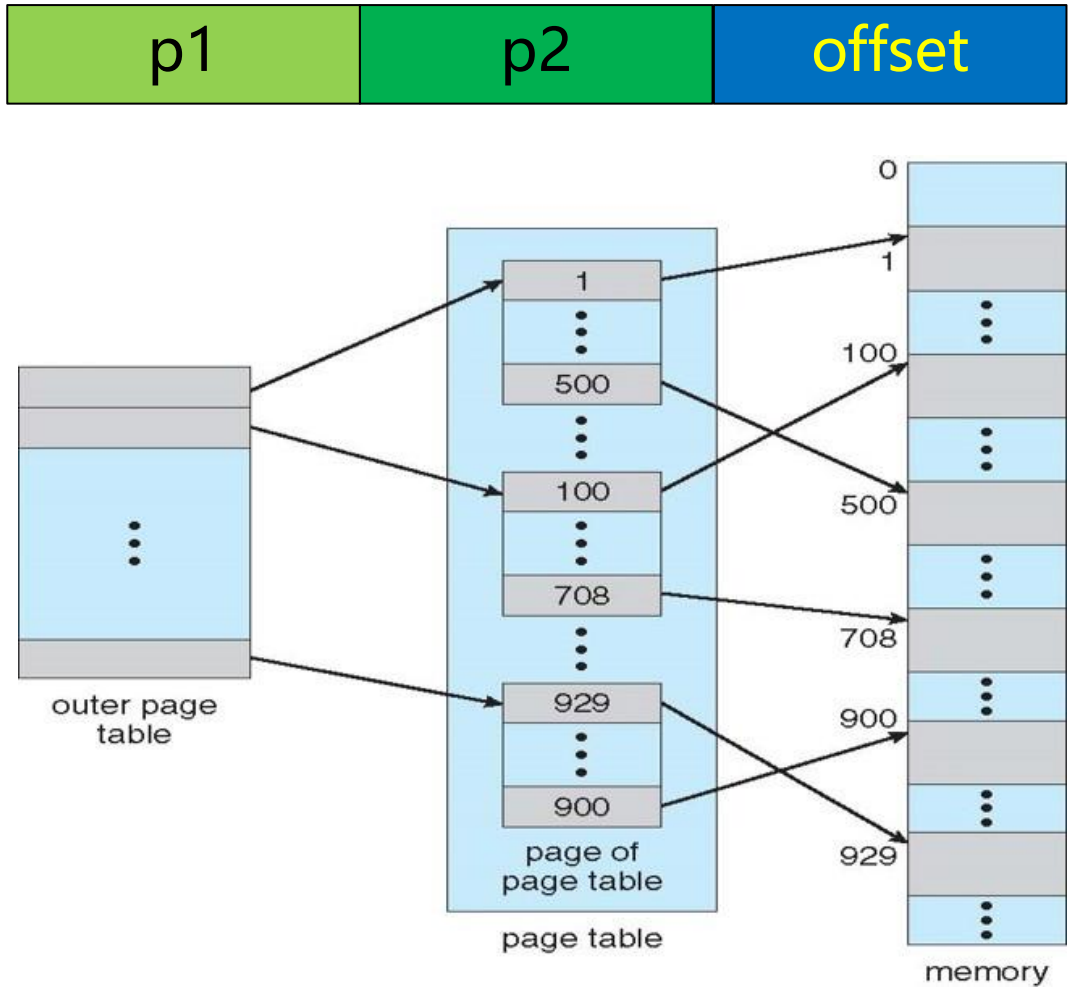
两级页表示意图



多级页表

处理大地址空间的页表方案1：多级页表

两级页表示意图



处理大地址空间的页表方案1：多级页表

练习：假设CPU支持38位的逻辑地址和32位的物理地址。如果使用二级页表，页面大小为16KB，每个页表项占据4个字节。则应为虚拟地址中的第一级和第二级页表分别分配多少位?

处理大地址空间的页表方案1：多级页表

练习：假设CPU支持38位的逻辑地址和32位的物理地址。如果使用二级页表，页面大小为16KB，每个页表项占据4个字节。则应为虚拟地址中的第一级和第二级页表分别分配多少位?

解答：

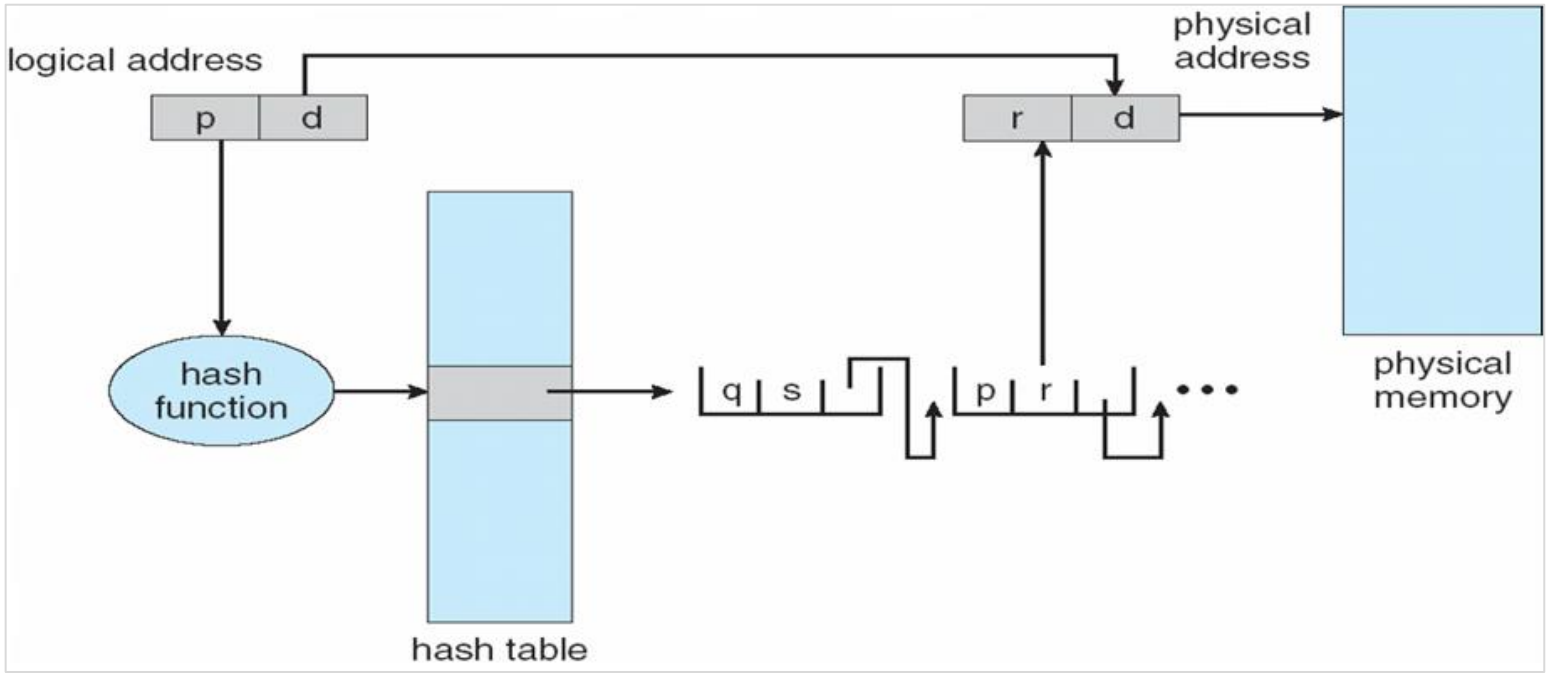
页大小=16KB → 页偏移的位数d=14

第二级页表分配位数p2=log(16KB/4B) =12

第三级页表分配位数p1=log(16KB/4B) =12

哈希页表

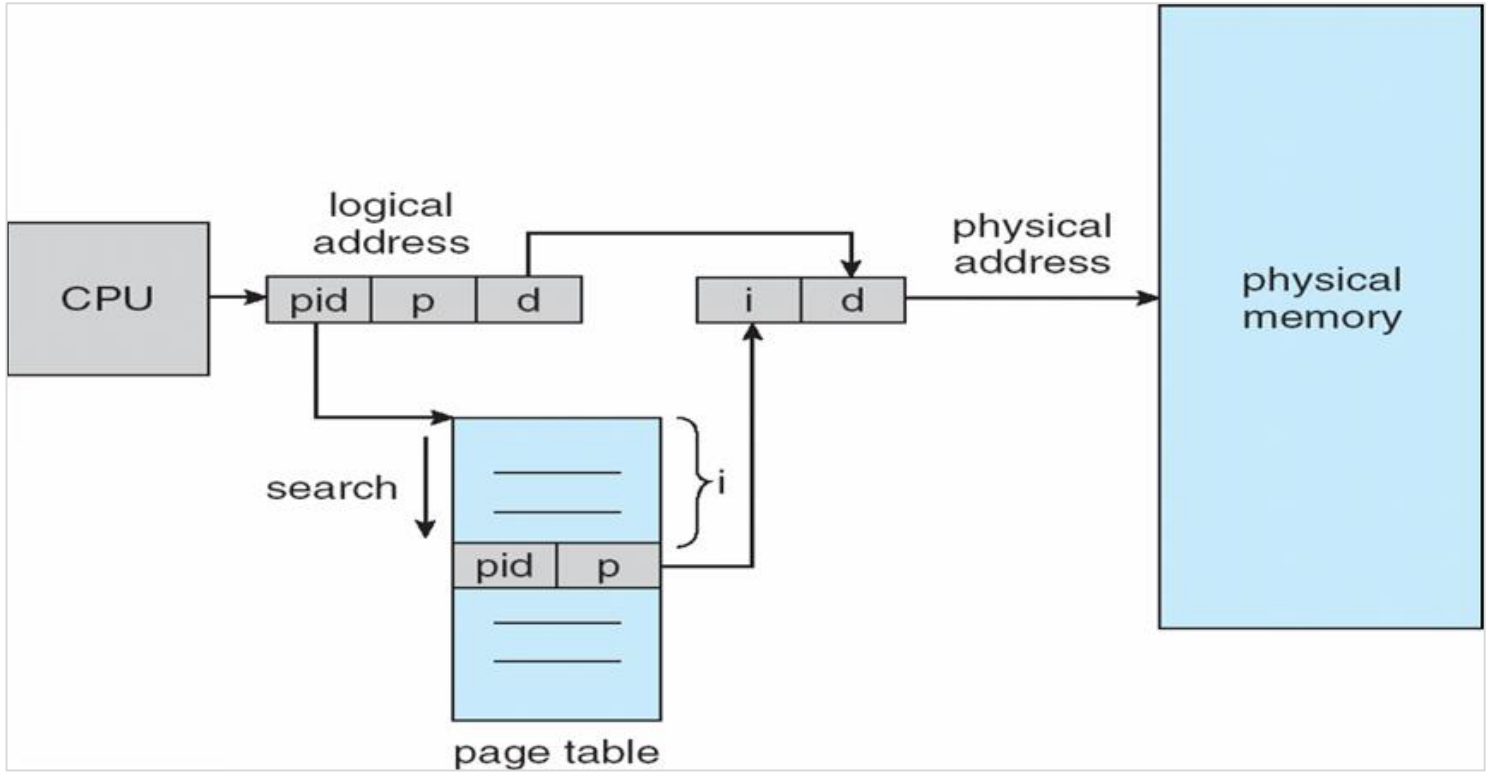
处理大地址空间的页表方案2：哈希页表



- 虚拟页号经过哈希函数转换后，指向其中某个页表项
- 哈希函数值相同的虚拟页号，指向同一个页表项，它们在该页表项下组成一个链表
- 地址翻译时，由虚拟页号哈希后锁定对应链表，搜索与虚拟页号的匹配项
- 如果找到匹配项，则找到虚拟页号对应的物理页帧

反置页表

处理大地址空间的页表方案3：反置页表



用物理页号（页框号）检索反置页表

目标是(pid, p)，
如果在反置页表中索引为i的表项的内容是(pid,p)，则对ID为pid的进程的逻辑页号p对应的物理页号是i

利用哈希表，使得查页表操作能一次命中，或者耗费较少的查找次数

反置页表优点：
在逻辑地址空间远大于物理空间的前提下，页表所占空间相对较小

反置页表劣势：
页地址翻译过程中，页表项检索过程耗时很长，导致地址翻译效率低

问题

哈希页表是否适合处理大地址空间?

问题

哈希页表是否适合处理大地址空间?

解答:

当一个任务占用大的虚拟地址空间的一小部分时，哈希页表非常适合。

哈希页表的缺点是在同样的哈希页表上映射多个页面而引起的冲突。

如果多个页表项映射在同个入口处，则可能导致其相应的哈希入口负担过重。

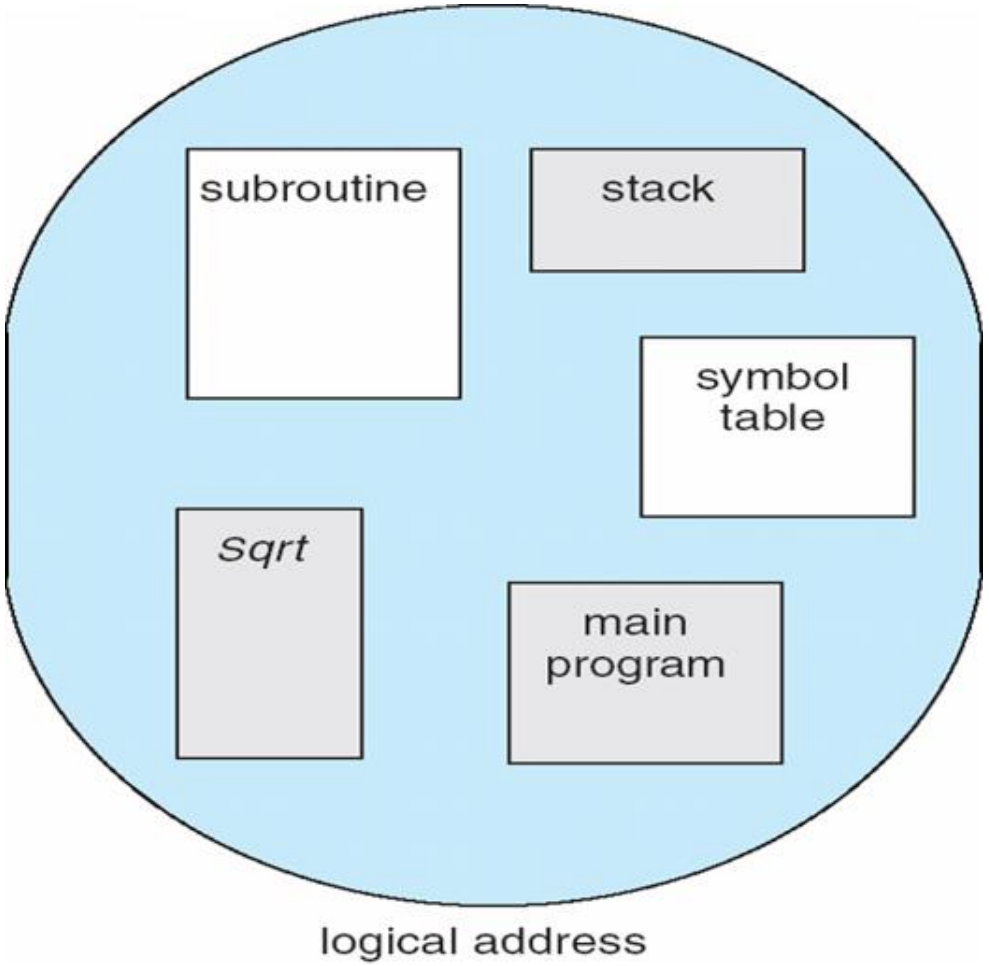
分段机制

Segmentation Mechanism

03

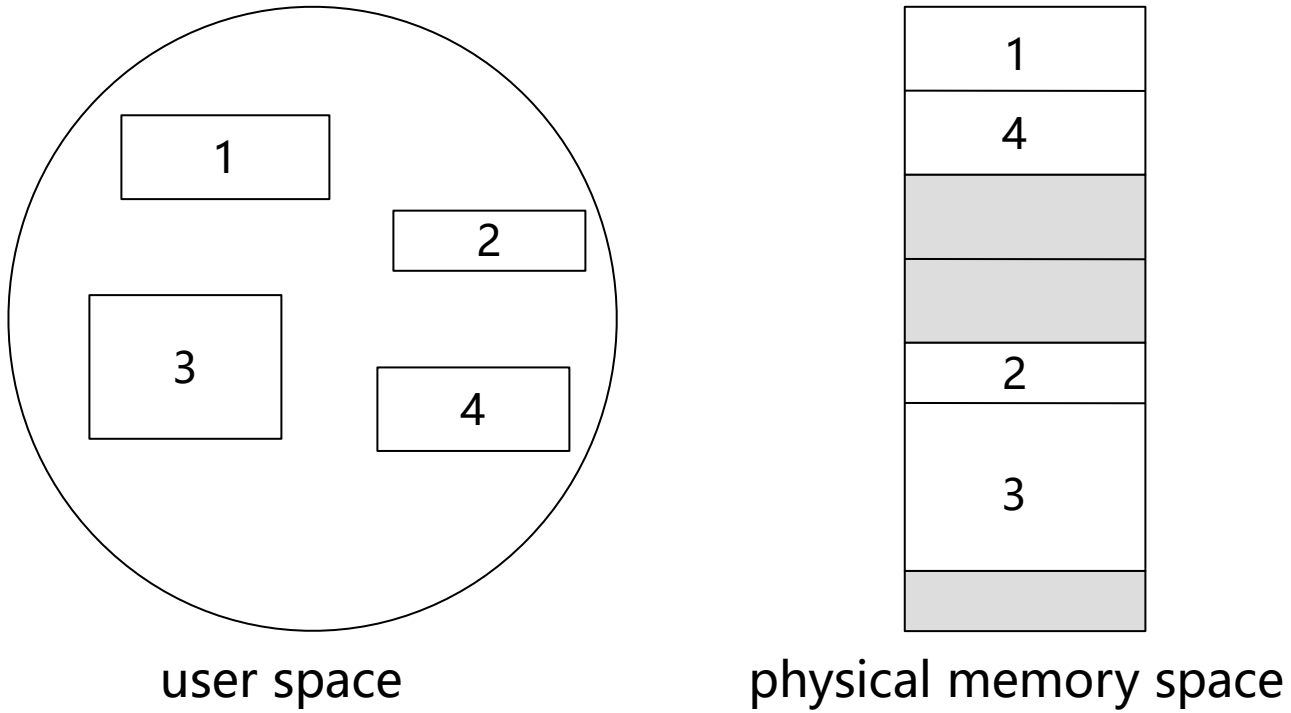
分段概念

进程地址空间分块示意图



分段概念

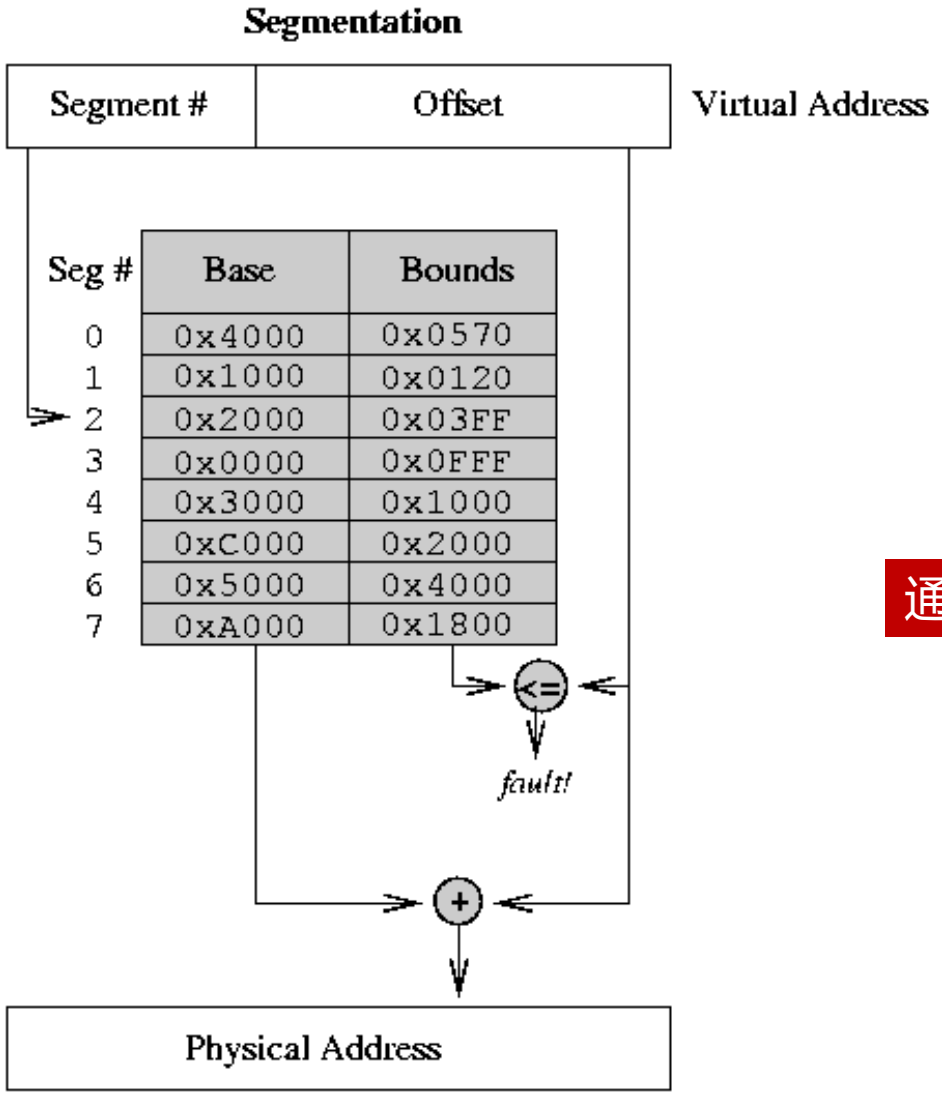
程序逻辑空间分段



user space　　physical memory space

分段概念

分段结构



分段机制下的逻辑地址（段地址）

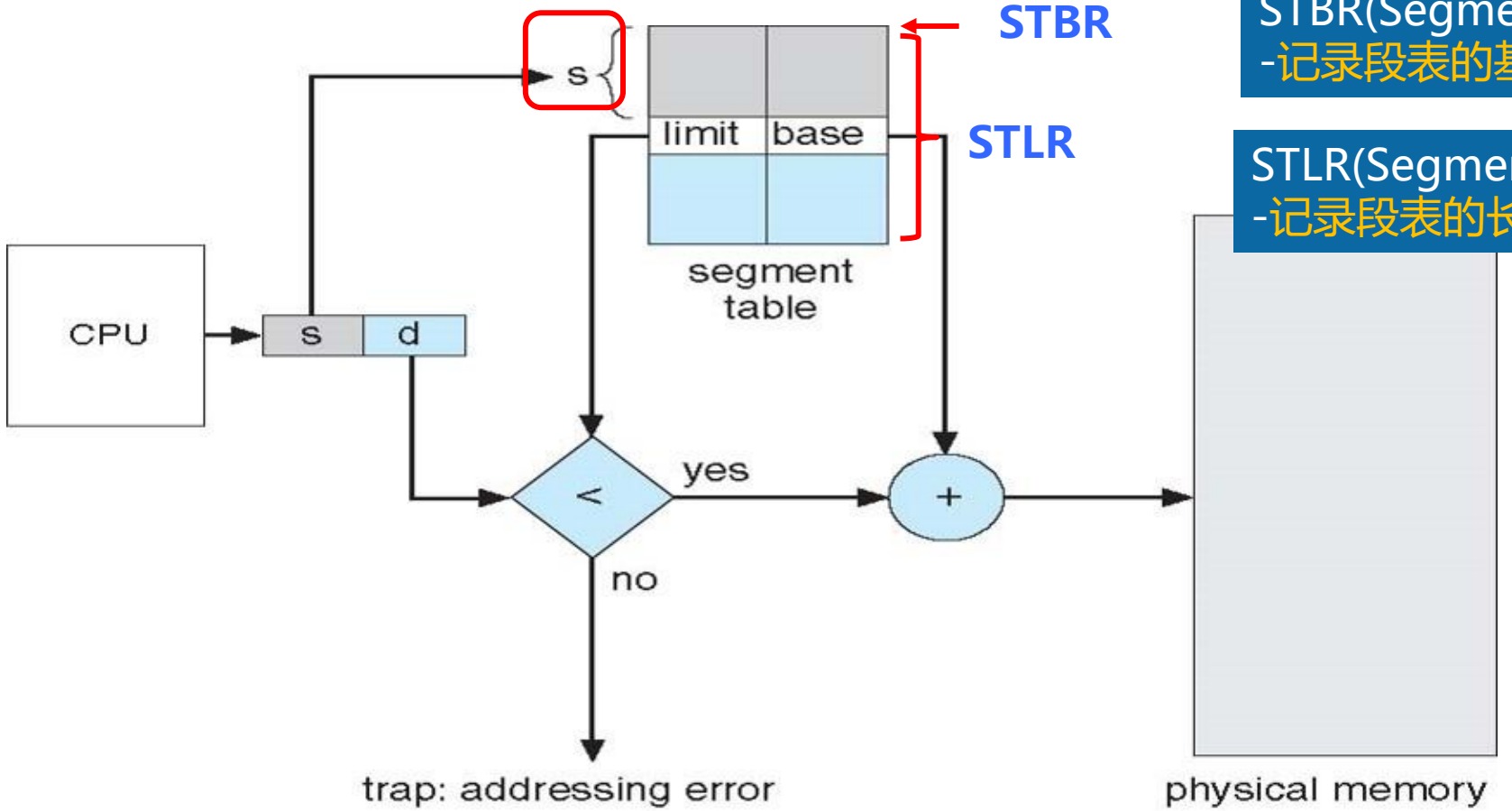
段号（Segment #） + 段内偏移（offset）

通常存储在段寄存器内

占据一个CPU字
（例如，32位CPU下，
偏移量用32位地址表示）

分段概念

分段结构：硬件逻辑示意图



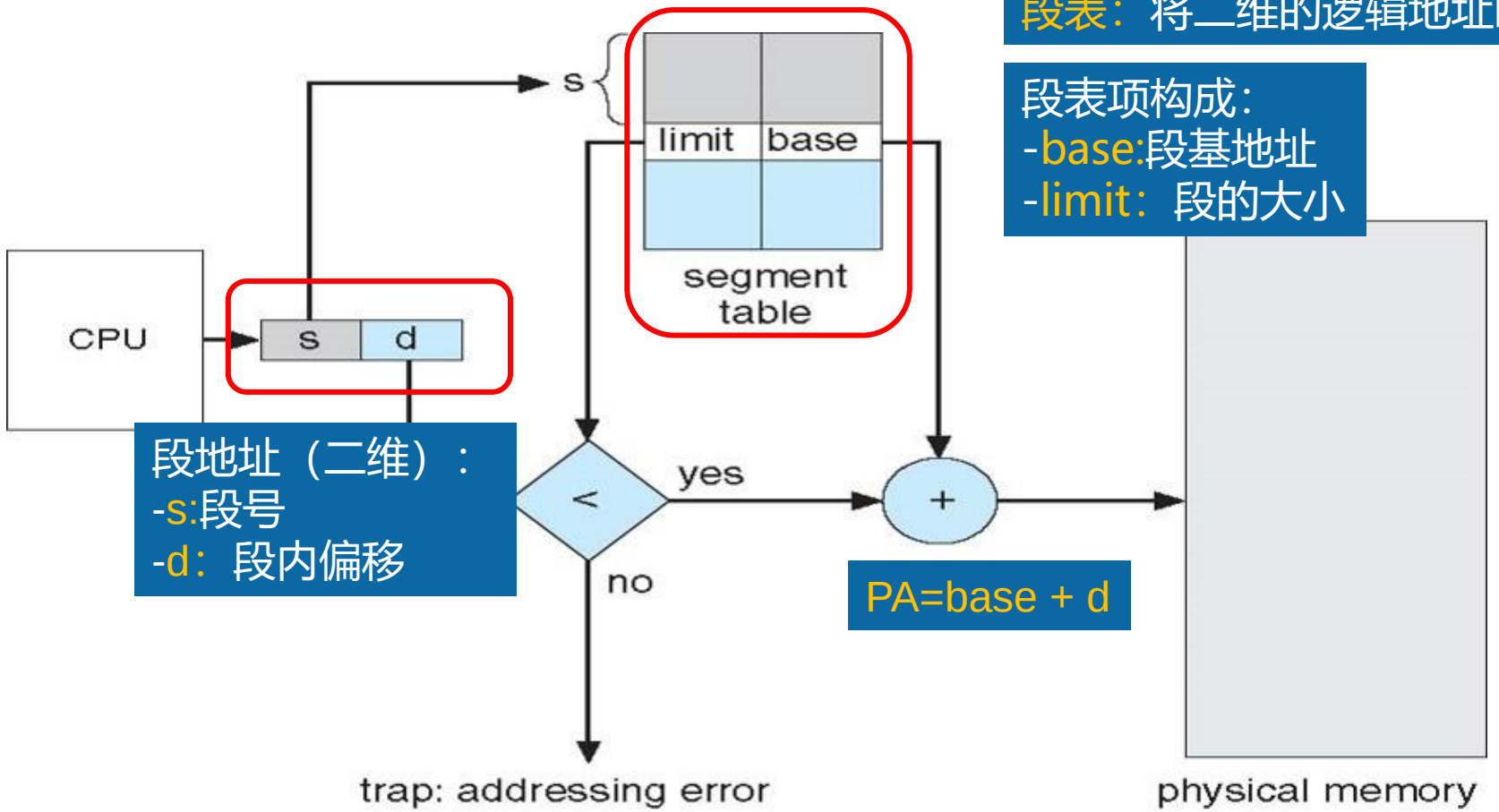
STBR(Segmentation Table Base Register)
-记录段表的基地址

STLR(Segmentation Table Length Register)
-记录段表的长度

访存要求： s < STLR

分段概念

分段结构：硬件逻辑示意图

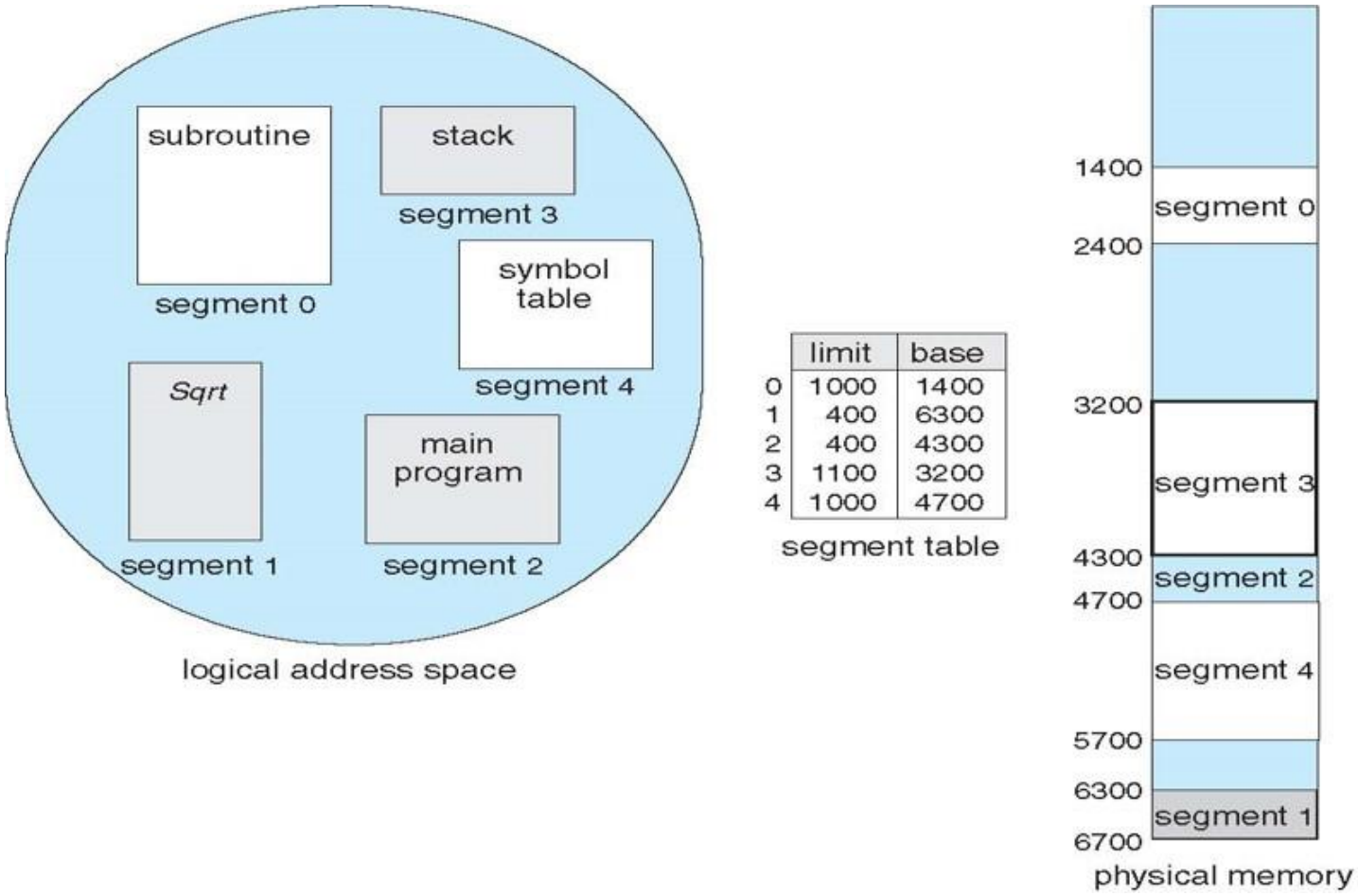


段表：将二维的逻辑地址映射到物理地址

段表项构成：
-base:段基地址
-limit：段的大小

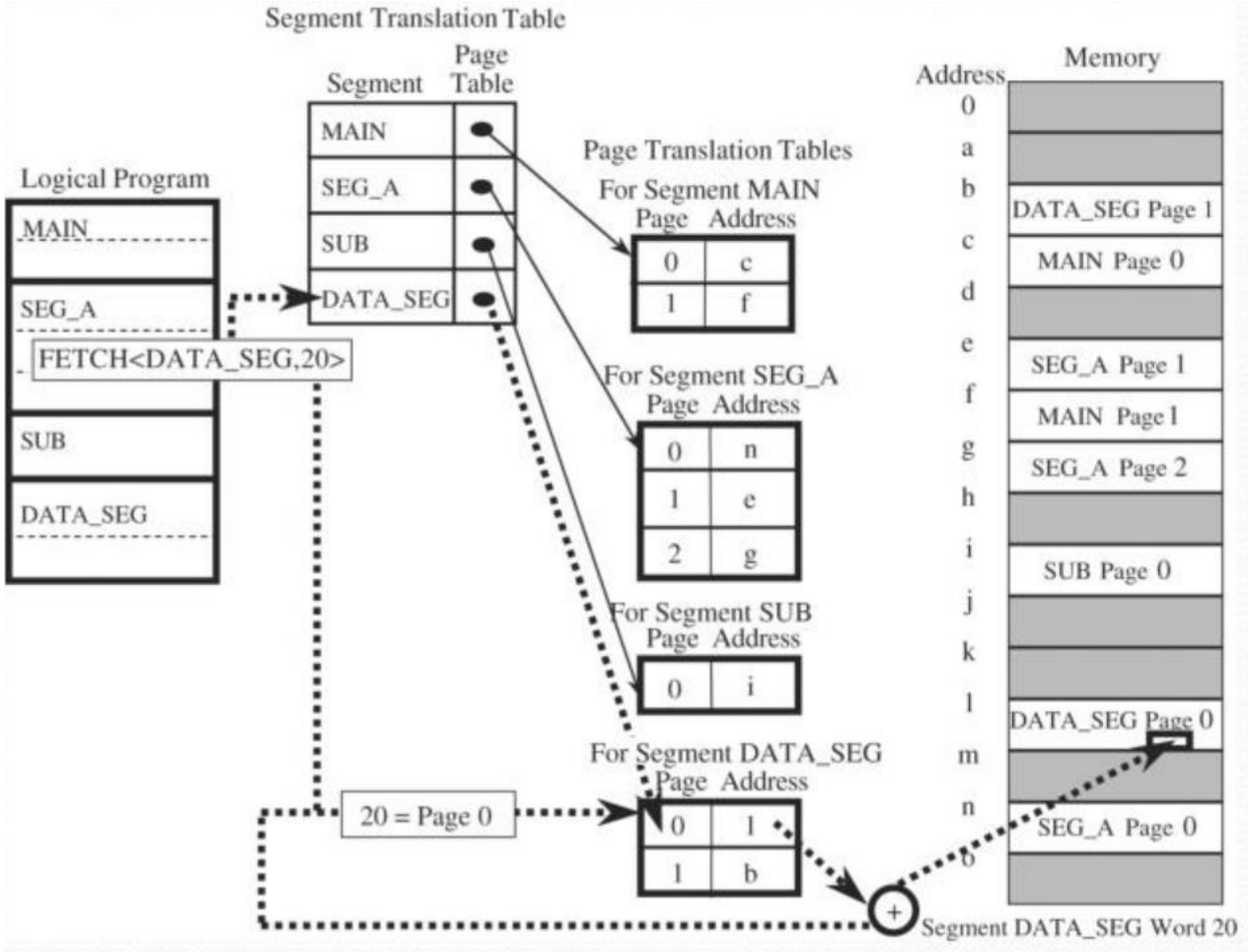
分段概念

分段结构：段式管理和地址转换



段页式内存管理

段页式管理：段式 + 页式



段内分页

以段为单位，整体保护

段页式内存管理

现状：无论Linux还是Windows，在现代CPU上基本都弱化对于段机制的使用

采用扁平式段布局（段寄存CS、DS等都设为0）：每个段都涵盖整个进程逻辑地址空间

现代操作系统中还保留段的概念，纯粹是因为CPU为了兼容而保留了16位系统中遗留下的这一机制

操作系统中采用扁平式的分段管理，也就为了糊弄CPU

intel 处理器从4位机到最新的64位，历时发展数十年，从8086开始，Intel CPU正式进入x86时代，而段寄存器也是这个时候诞生。
8086处理器位数变成16位，但是地址总线又变成了20根。为了能够访问到整个地址空间，在CPU里添加了4个段寄存器，分别为CS（代码段寄存器）DS（数据段寄存器） SS（堆栈段寄存器）ES(扩展段寄存器)。所以段寄存器就是为了解决CPU位数和地址总线不同的问题而诞生的。

Summary

小结：

- 分页的硬件支持
- 页表结构
- 分段机制

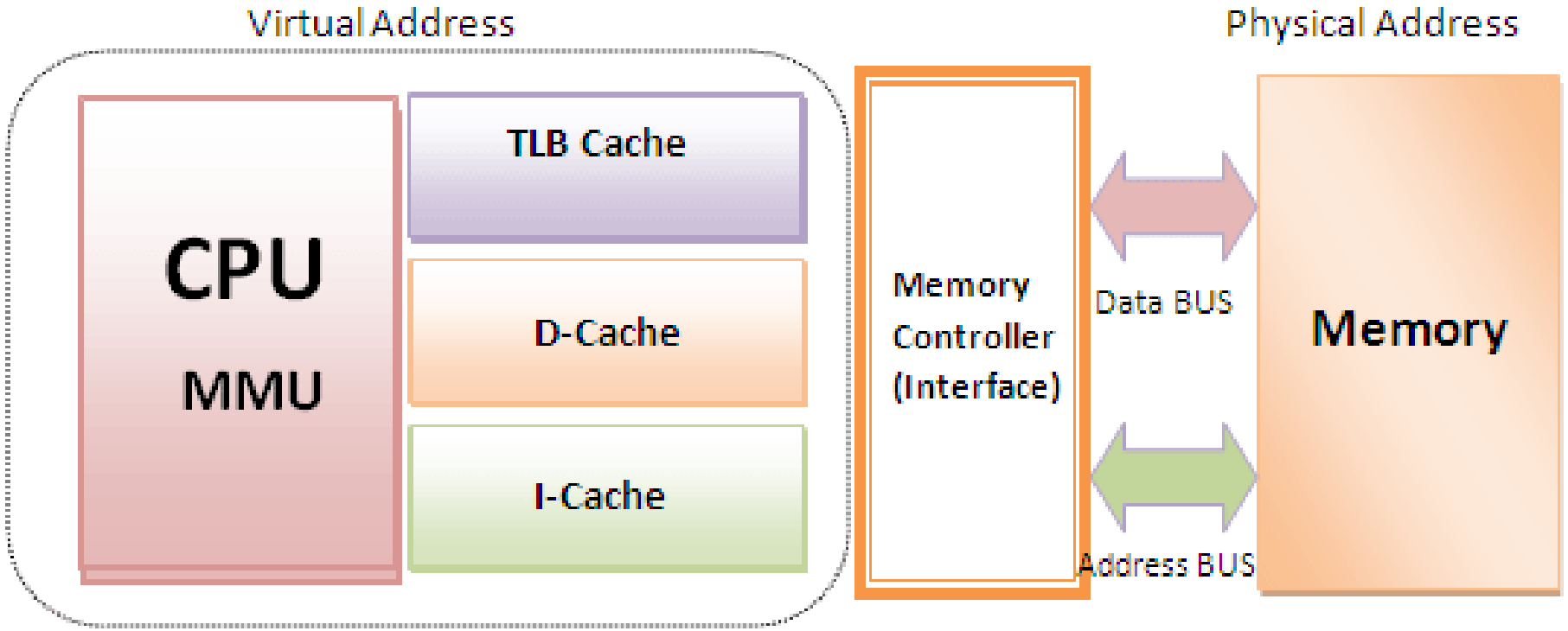
练习

1.在一个分段存储管理系统中，其段表如下表所示：

Seg #	Base	Limit
0	210	500
1	2350	20
2	100	90
3	1350	590
4	1938	95

试求（0，430），（2，500），（4，112）对应的物理地址是什么。

分页硬件基本结构


Virtual Address
Physical Address
CPU
MMU
TLB Cache
D-Cache
I-Cache
Memory Controller (Interface)
Data BUS
Address BUS
Memory

分页硬件基本结构

分页硬件示例：AArch64 MMU (ARM)

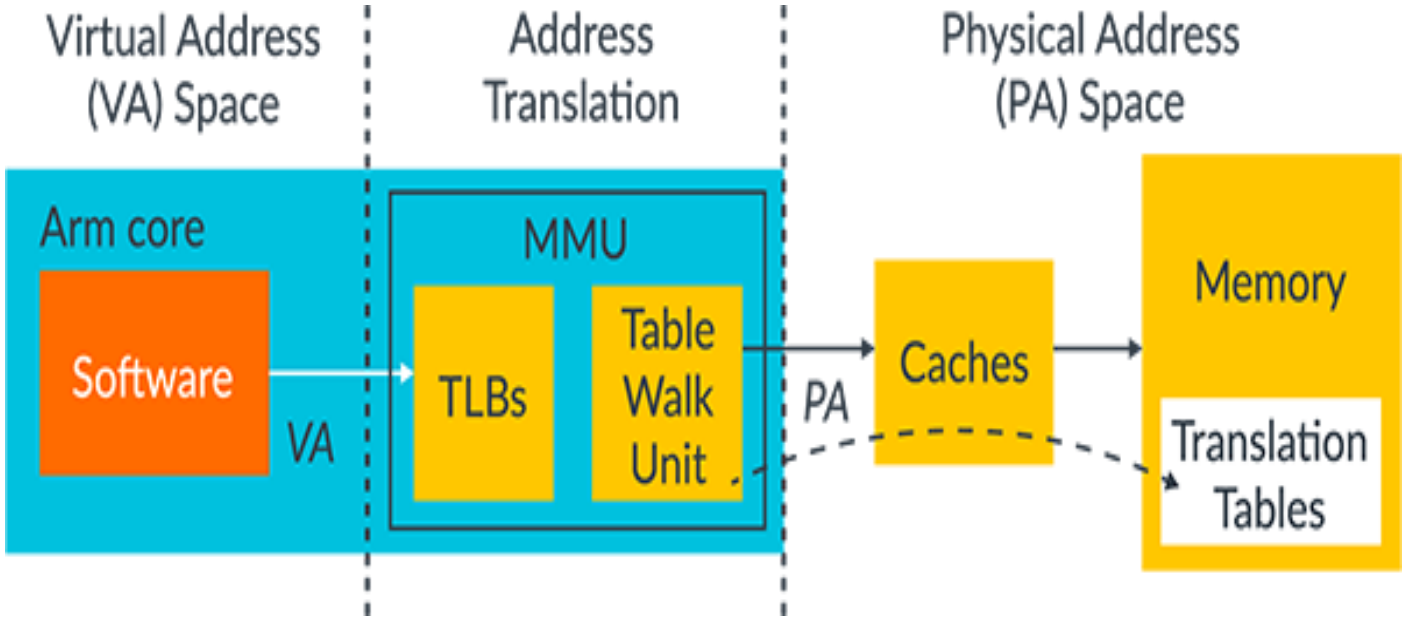
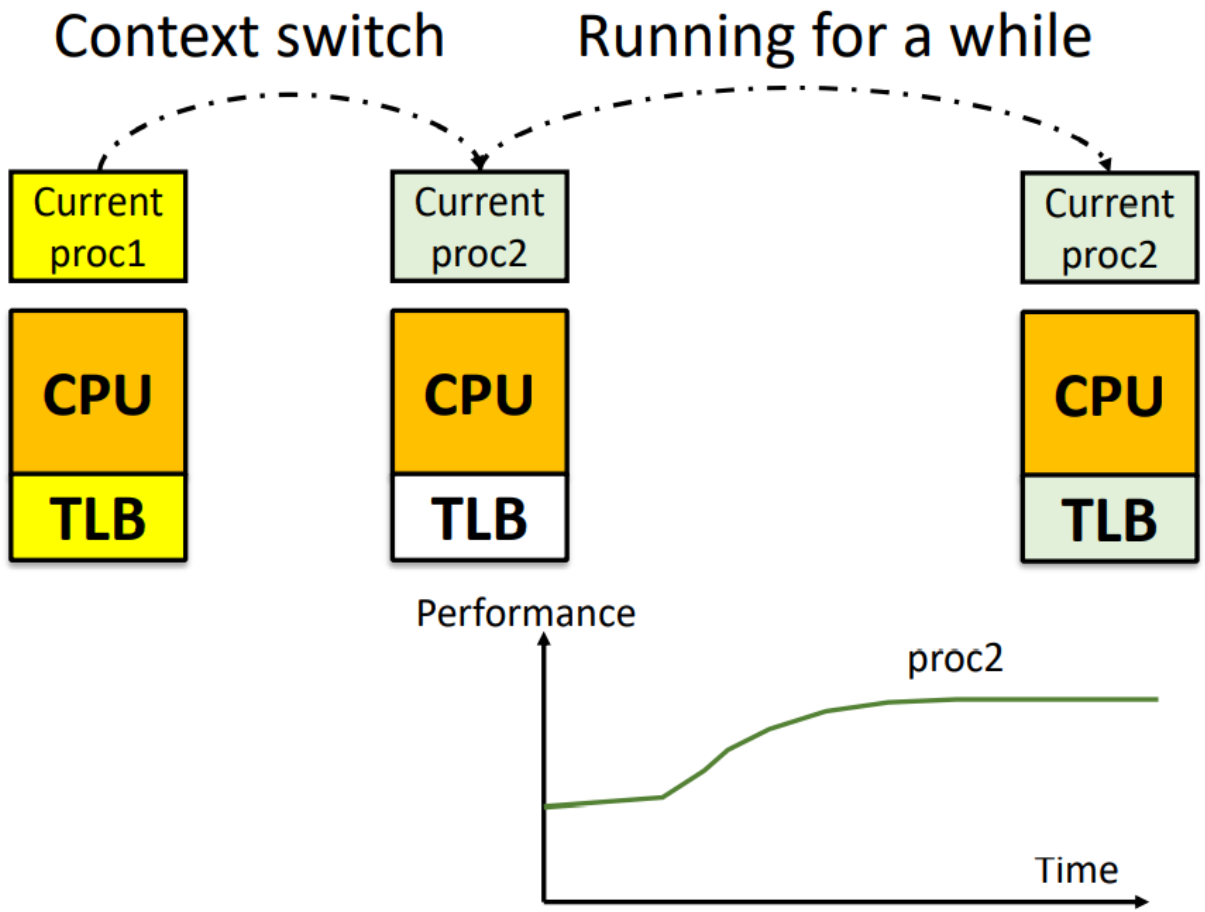


Table Walk Unit : 负责读取内存中的地址转换表

TLB: 负责为MMU缓存少量地址转换表项

TLB对性能影响示意：Process Cold Start



进程P1刚切换到P2，
由于TLB flush操作，
进程P2的性能较慢，后慢慢攀升
（命中率α 逐渐提升）

谢 谢!
Thank you!