地址转换

邰颖
南京大学
智能科学与技术学院

内存虚拟化的效率、控制与灵活性

- 类似CPU虚拟化，内存虚拟化也采用类似有限直接执行（LDE，Limited Direct Execution）的策略，以实现效率（efficiency）和控制（control）。

- 在内存虚拟化中，效率和控制依赖于硬件支持（hardware support），例如：
  - 寄存器（registers）
  - MMU（内存管理单元，Memory Management Unit）
  - TLB（转换后备缓冲，Translation Lookaside Buffer）
  - 页表（page table）

- 同时提供灵活性（flexibility），允许进程自由使用其地址空间
  - *地址转换（address translation）*

地址转换（Address Translation）

- **通过地址转换，硬件**将**虚拟地址**转换为**物理地址**
  - 实际存储的数据位于**物理地址**中

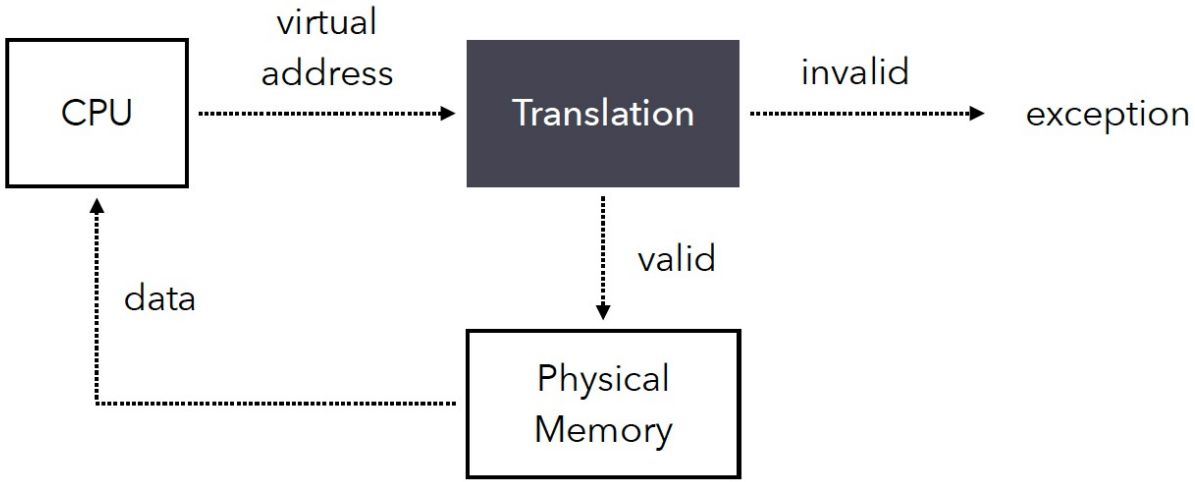
- **操作系统（OS）**需要在关键时刻**干预**以配置硬件：
  - 操作系统必须**管理内存**，以确保合理分配和使用。

- 一些假设（不一定现实）：
  - 假设1：地址空间在**物理内存中是连续存放的（contiguously）**。
  - 假设2：地址空间大小**不会太大（not too big）**，即小于物理内存的大小。
  - 假设3：**所有进程的地址空间大小相同（the same）**。

地址转换 (Address Translation)

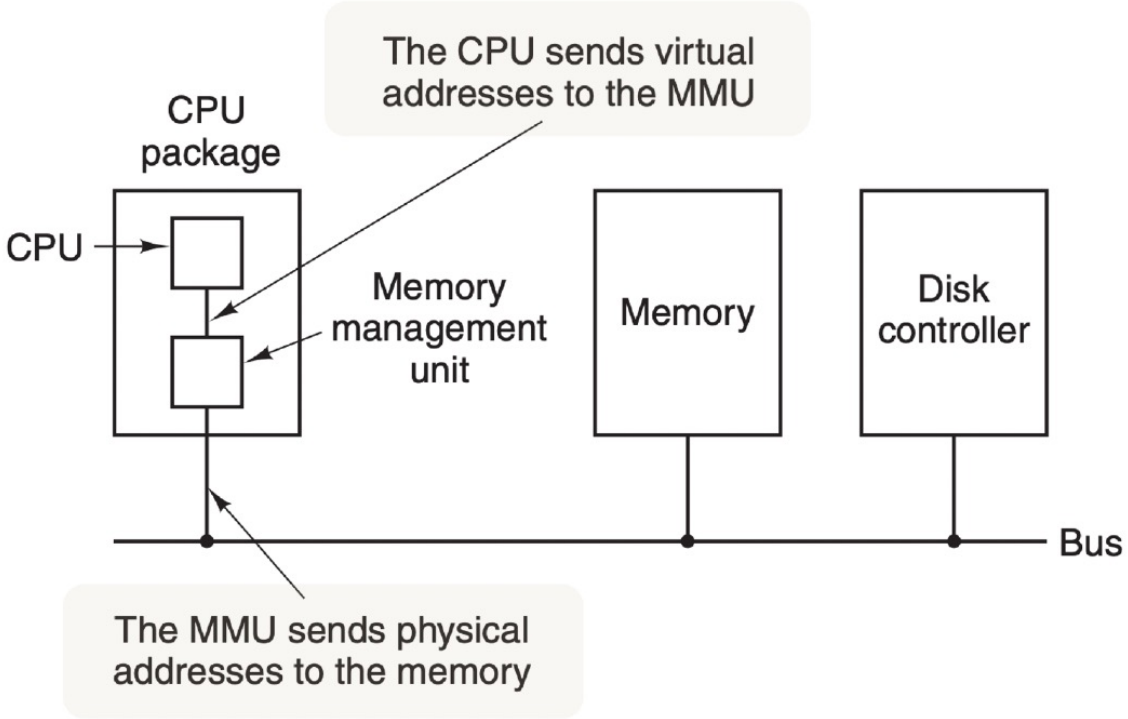
地址转换 (Address Translation): 将 ⟨**pid, virtual address**⟩ 映射到其相应 **physical address** 的一个函数

- 进程只感知虚拟地址，无法直接访问物理地址

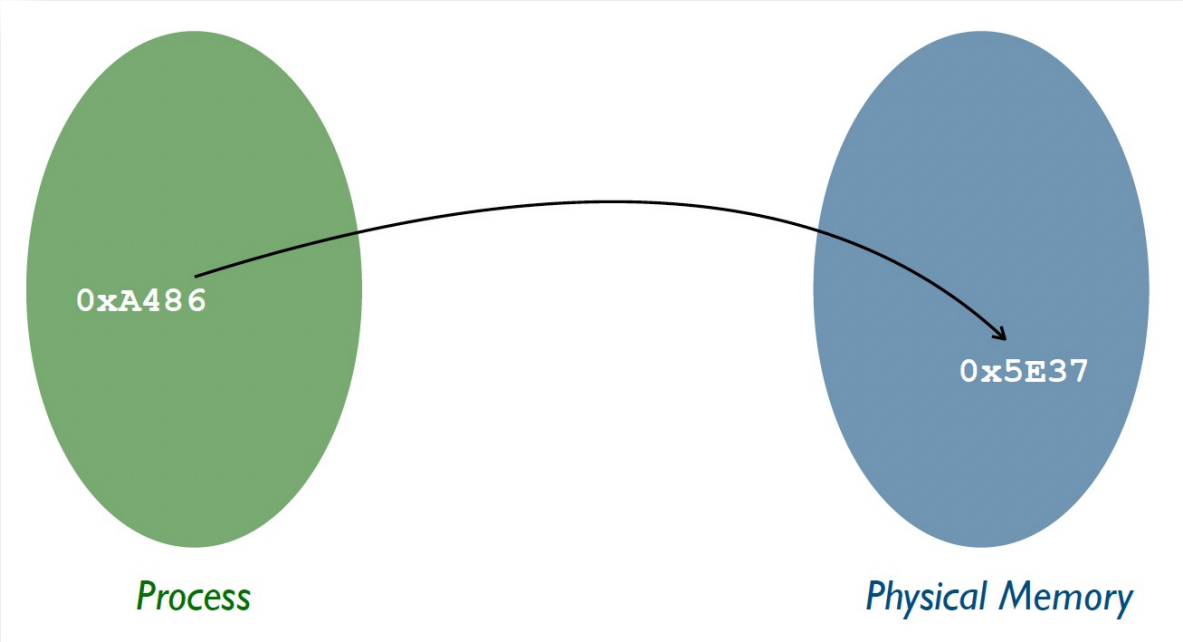


地址转换 (Address Translation)

为了提高地址转换效率，通常需要硬件 (Memory Management Unit, MMU) 提供支持



地址转换（Address Translation）

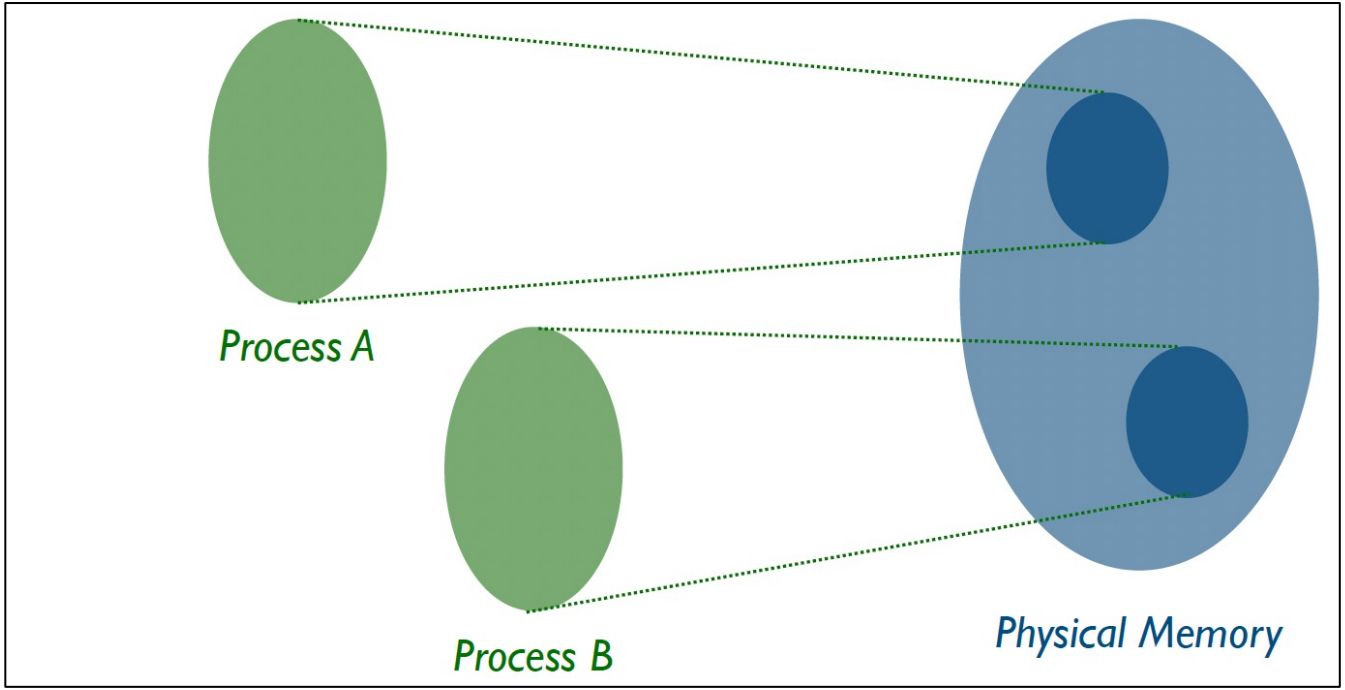


优点，可以很自然地提供：

- 保护
- 重定位
- 数据共享
- 多路复用

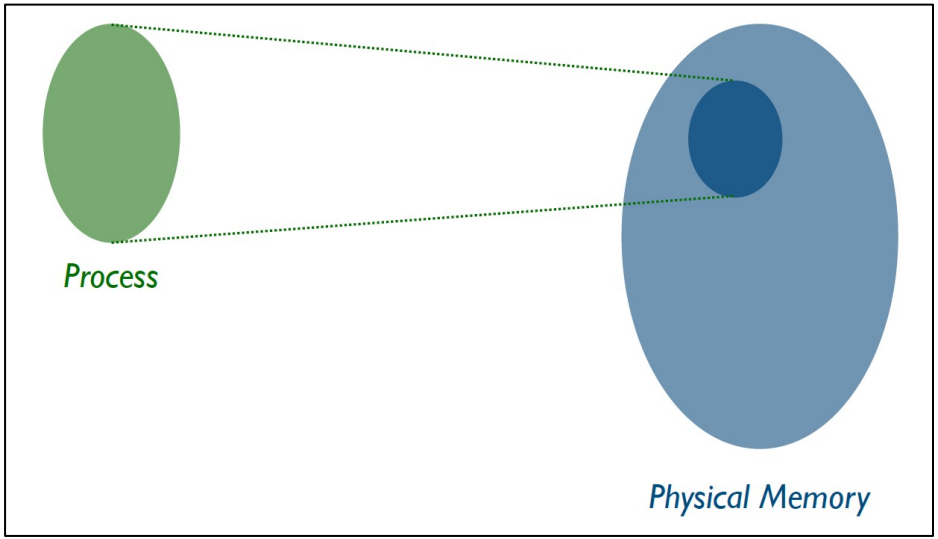
地址转换（Address Translation）：保护

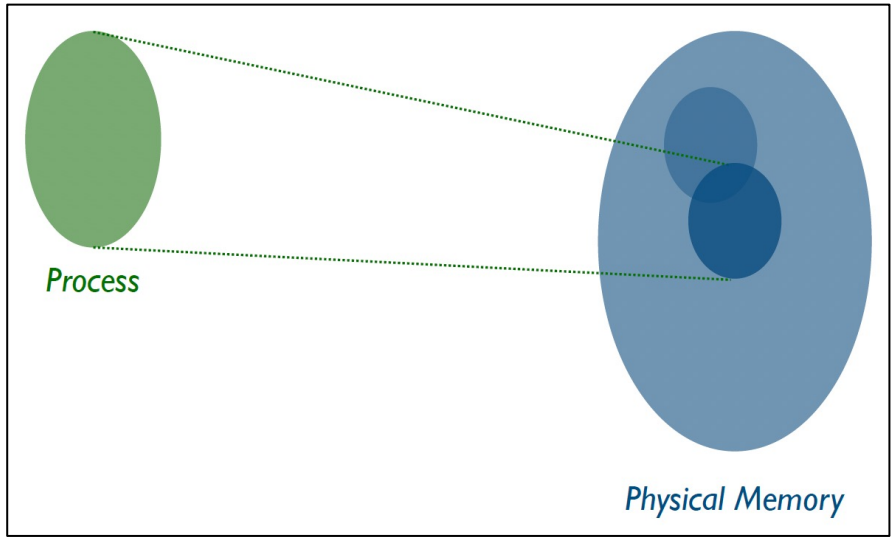
- 保护：不同进程使用不同的地址转换函数 (值域不相交)：将不同进程的虚拟地址空间映射到互不重叠的物理内存区域 (进程间/进程和 OS 间的保护)



地址转换（Address Translation）：重定位

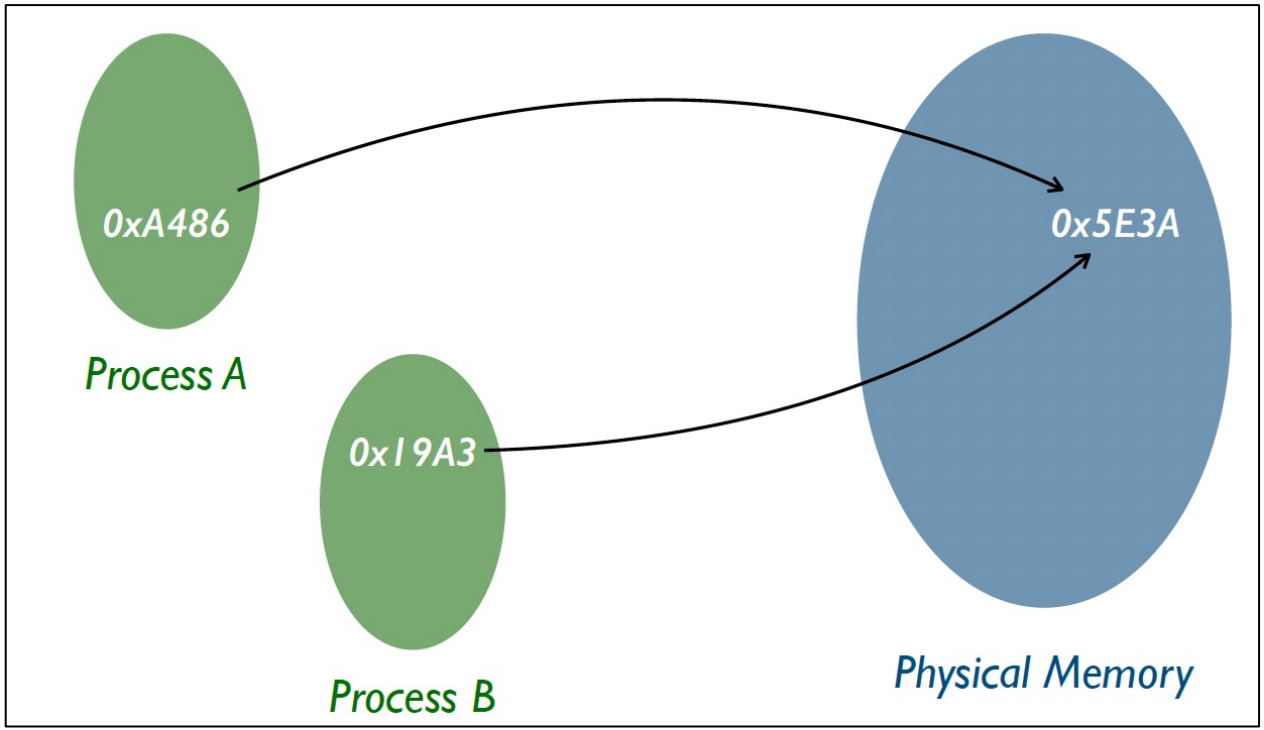
- 重定位：进程使用的地址转换函数可在运行时随时间而变化：允许将进程加载到物理内存的任意地址 (动态重定位)





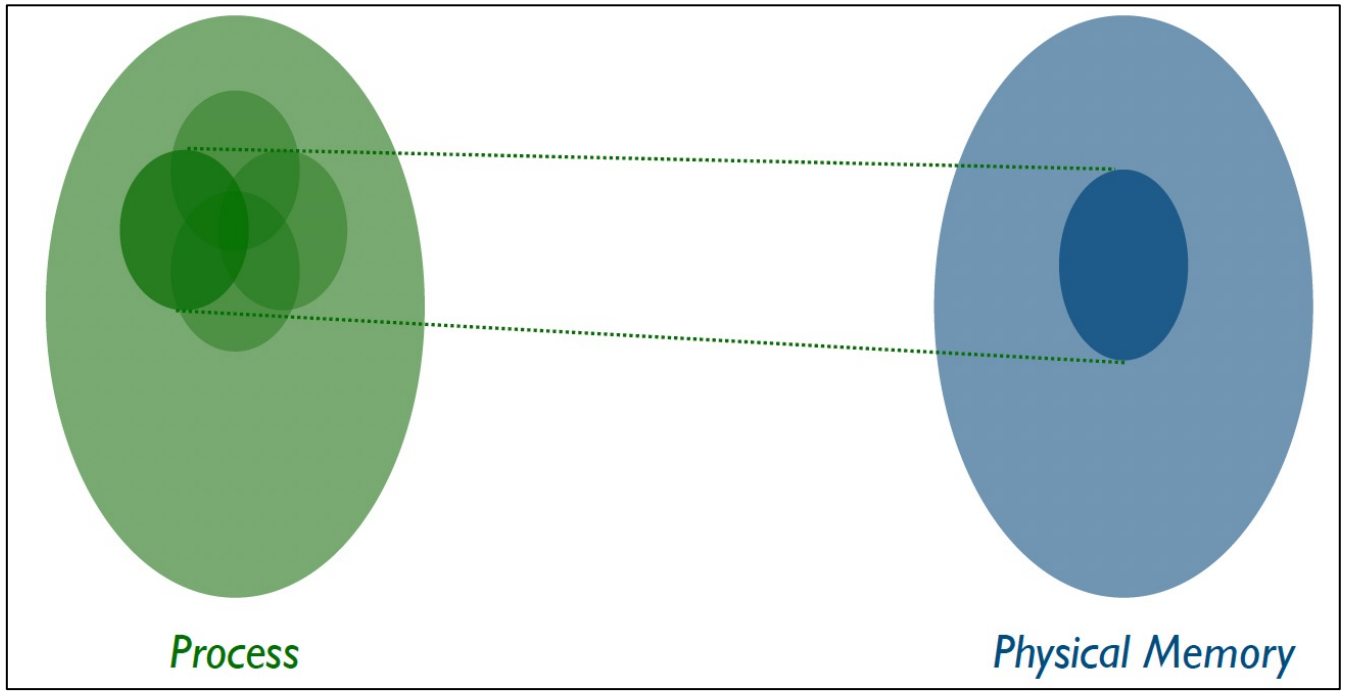
地址转换（Address Translation）：数据共享

- 数据共享：将不同进程的不同虚拟地址映射到同一个物理地址



地址转换（Address Translation）：多路复用

- 多路复用：进程的虚拟地址空间的不同部分可以在不同时间映射到相同的物理内存区域，从而实现对有限物理内存的高效利用



示例——地址转换（Address Translation）

- C 语言代码

```c
void func() {
    int x = 3000;
    x = x + 3;  // 这是我们关注的代码行
    ...
}
```

•操作步骤

- 加载（Load）：从内存中读取一个值
- 增加（Increment）：将值加 3
- 存储（Store）：将更新后的值存回内存

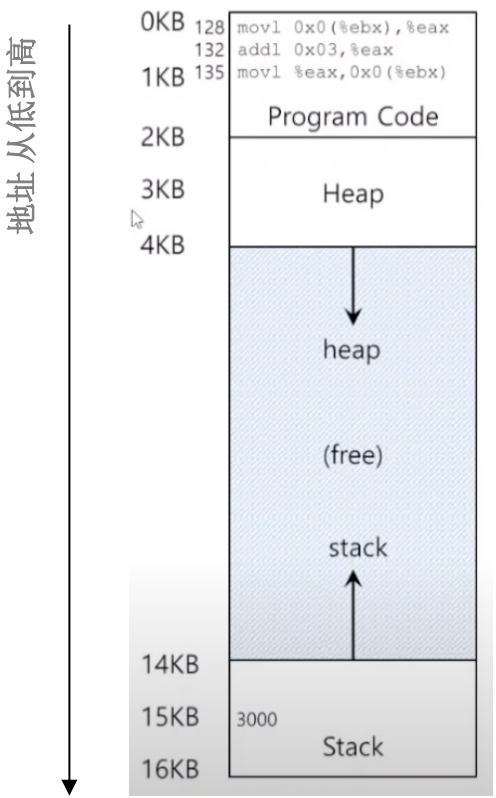
示例——地址转换（续）

- 汇编代码

```
128 : movl 0x0(%ebx), %eax   ; 将内存地址 0+ebx 处的值加载到 eax
132 : addl $0x03, %eax       ; eax 寄存器中的值加 3
135 : movl %eax, 0x0(%ebx)   ; 将 eax 中的值存回内存地址 0+ebx
```

- **假设 x 的地址存放在 ebx 寄存器中。**
- **加载（Load）：** 从 ebx 指向的内存地址读取值，并存入 eax。
- **加法（Add）：** 将 eax 中的值加 3。
- **存储（Store）：** 将 eax 中的值写回 ebx 指向的内存地址。

示例：地址转换（续）



执行过程：

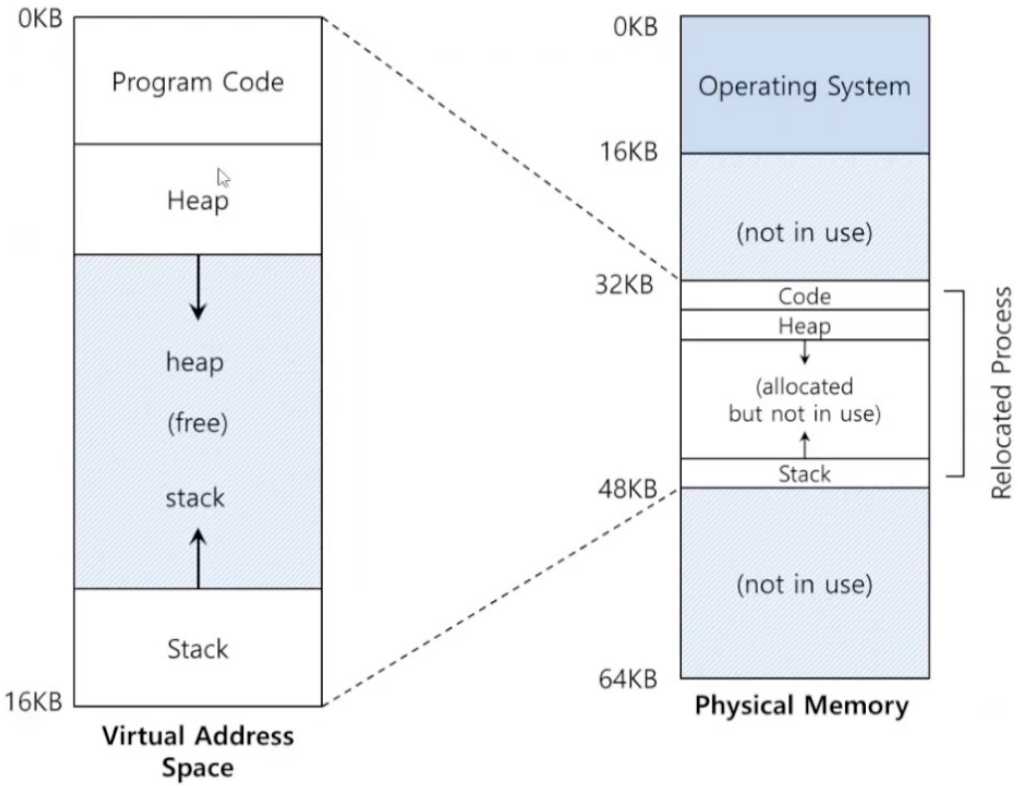
- 从地址 **128** 取指令
- 执行该指令（从地址 15KB 处加载数据3000）
- 从地址 **132** 取指令
- 执行该指令（无内存访问）
- 从地址 **135** 取指令
- 执行该指令（将数据存储到 15KB 处）

地址空间重定位（Address Space Relocation）

- 进程对其**虚拟地址空间**（**virtual address space**）的视图是从 0 到 16KB。
  - 这可能**精确映射**到物理内存。

- 假设操作系统希望将进程放置到**物理内存的其他位置**，而不是地址 0 处？
  - **是否可以在不改变进程虚拟地址空间的情况下重定位地址空间？**

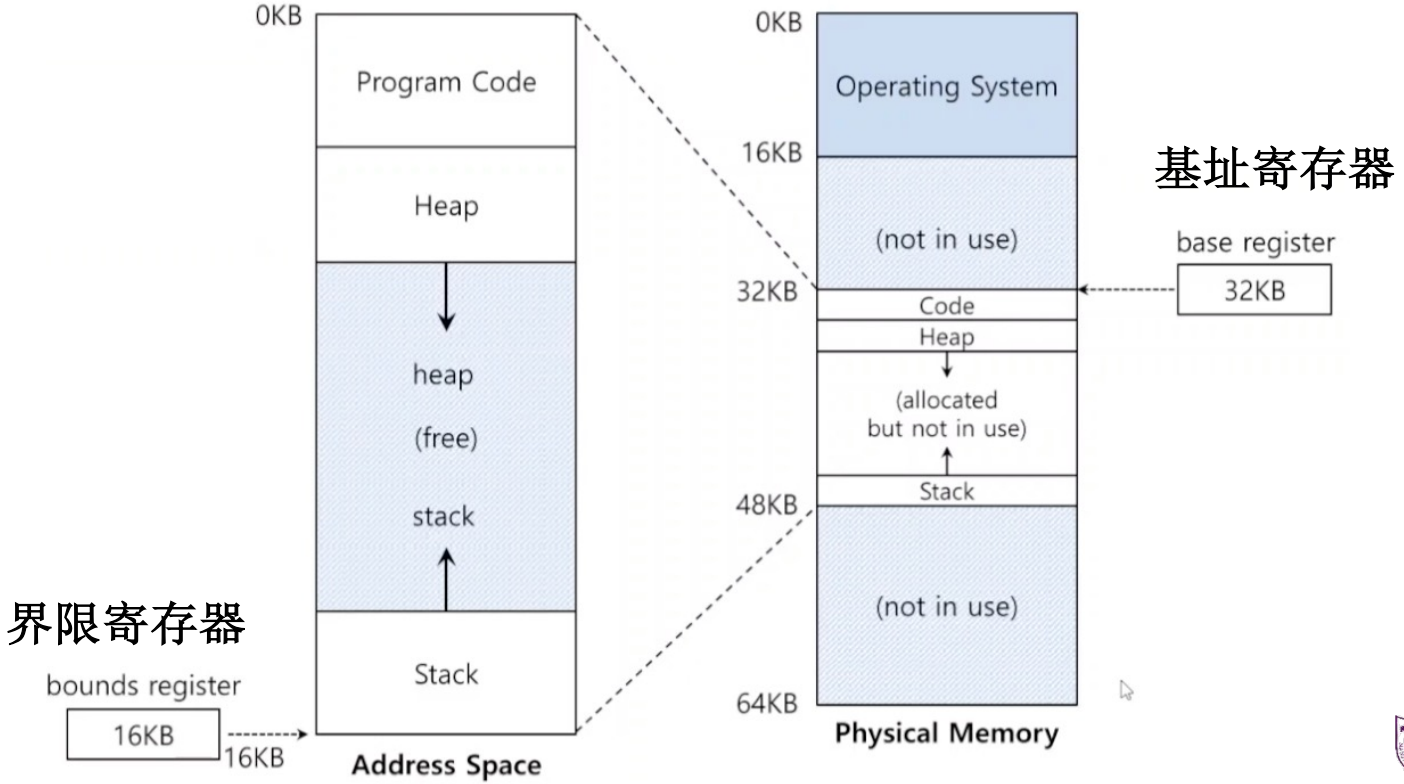
单个重定位进程（A Single Relocated Process）



重定位进程

动态重定位（基于硬件）：使用基址寄存器和界限寄存器

- 基址加界限机制（base and bound），有时又称为动态重定位（dynamic relocation）



动态重定位（基于硬件）：使用基址寄存器和界限寄存器

- 程序在编写和编译时，假设它被加载到地址 **0**（零）。
- 当程序开始运行时，操作系统决定进程应加载到物理内存的哪个位置：
  - 设置基址寄存器（base register）的值。

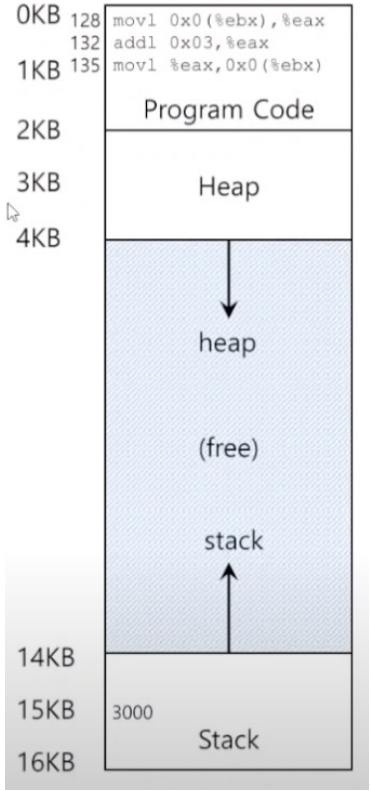
$$phycal\ address = virtual\ address + base$$

  - 每个虚拟地址必须不大于界限（**bound**）且不能为负：

$$0 \leq\ virtual\ addressvirtual\ address < bounds$$

- 需要连续分配物理内存，容易导致外部碎片问题

动态重定位（基于硬件）：使用基址寄存器和界限寄存器




```
128 : movl 0x0(%ebx), %eax
```

取指令（Fetch）： 从地址 128 取指令

- **物理地址计算：** 32896 = 128 + 32KB（基址寄存器）

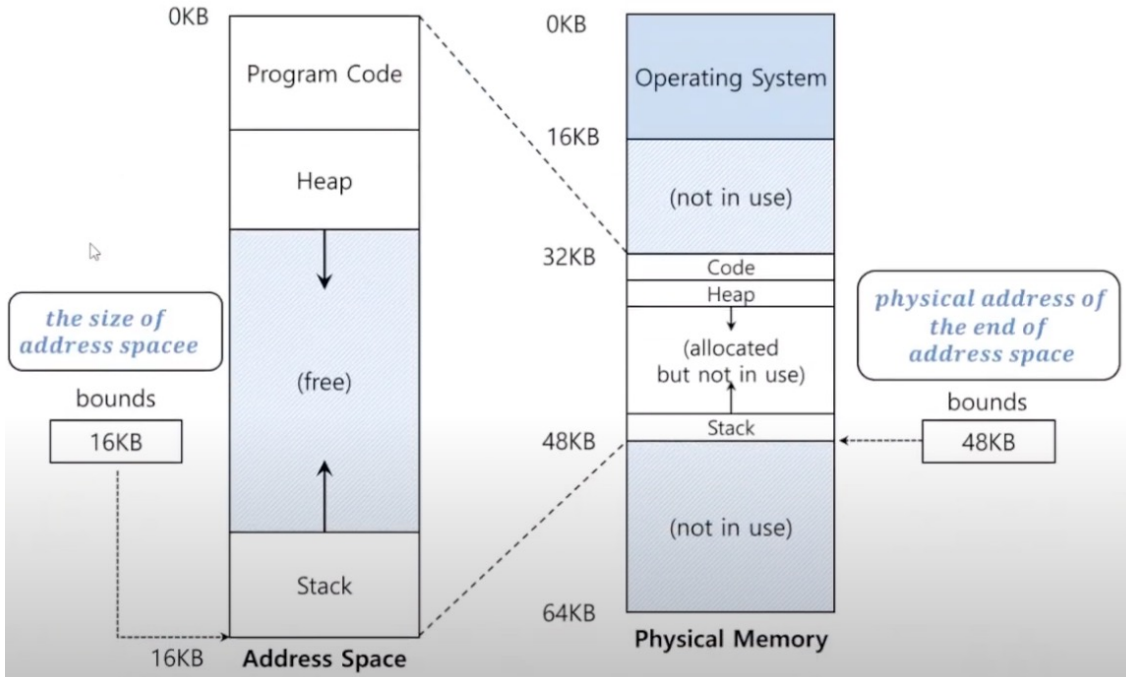
执行指令（Execute）： 从 15KB 地址加载数据

- **物理地址计算：** 47KB = 15KB + 32KB（基址寄存器）

界限寄存器的两种定义方式

•地址空间视角：
- 界限寄存器（**Bounds Register**）定义为地址空间的大小（例如16KB）。

•物理内存视角：
- 界限寄存器定义为地址空间结束的物理地址（例如 48KB）。



地址转换示例（Example Translations）

- 已知进程信息：
  - 地址空间大小为 4KB（bound register）
  - 加载到物理地址 16KB（base register）

- 示例地址转换：

虚拟地址（Virtual Address）	物理地址（Physical Address）
0	16KB
1KB	17KB
3000	19384
4400	错误（超出范围）

动态重定位所需的硬件支持总结

- **处理器模式**：通过**处理器状态字（Processor Status Word）**确定**内核模式（Kernel Mode）**和**用户模式（User Mode）**。
- **内存管理单元（MMU）** 需要**基址寄存器（Base Register）** 和 **界限寄存器（Bounds Register）** 进行地址转换和访问控制。
- **基址寄存器和界限寄存器的修改** 仅允许在**内核模式**下进行：例如，在**调度器切换进程**时，操作系统需要更新这些寄存器。
- **处理器应能够在非法内存访问时生成异常（Exceptions）**。

动态重定位所需的硬件支持总结（续）

- 硬件需求总结

表 15.2　　动态重定位：硬件要求

硬件要求	解释
特权模式	需要，以防用户模式的进程执行特权操作
基址/界限寄存器	每个 CPU 需要一对寄存器来支持地址转换和界限检查
能够转换虚拟地址并检查它是否越界	电路来完成转换和检查界限，在这种情况下，非常简单
修改基址/界限寄存器的特权指令	在让用户程序运行之前，操作系统必须能够设置这些值
注册异常处理程序的特权指令	操作系统必须能告诉硬件，如果异常发生，那么执行哪些代码
能够触发异常	如果进程试图使用特权指令或越界的内存

操作系统在动态重定位中必须解决的问题

操作系统必须**采取措施（take action）**来实现**基址-界限（base-and-bounds）**方法。

四个关键时刻操作系统需要介入：

- **当进程开始运行（starts running）**
  - 在物理内存中找到合适的地址空间，并维护**空闲列表（free list）**。
- **当进程终止（terminated）**
  - 释放该进程的内存，使其可供其他进程使用。
- **当发生上下文切换（context switch）**
  - 保存和恢复**基址-界限寄存器对（base-and-bounds register pair）**
  - 当进程被阻塞时，可以轻松地将其移动到不同的内存位置。
- **系统启动时（boot time）**，操作系统必须使用**特权指令（privileged instructions）**设置异常处理程序（exception handlers）。

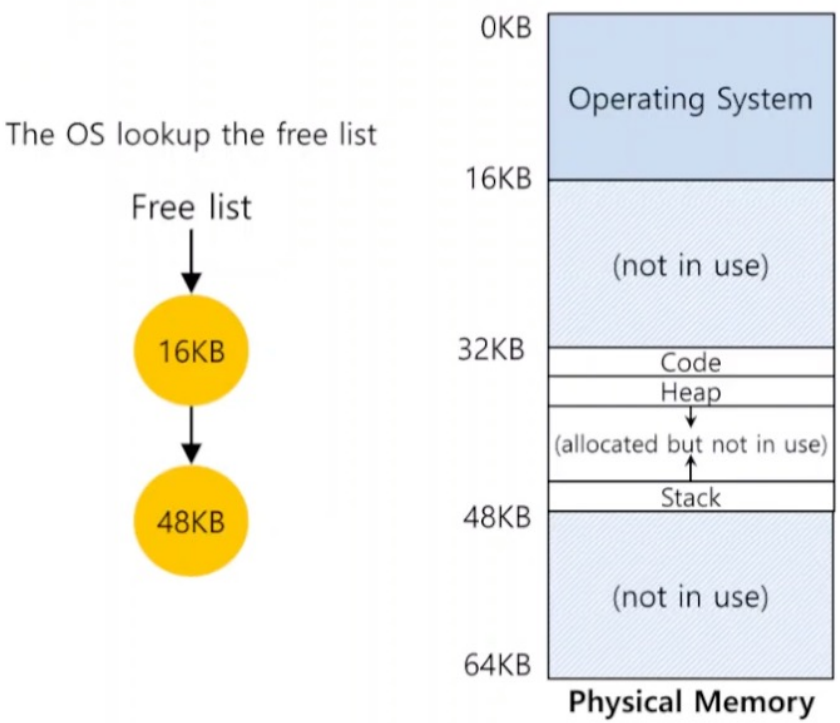
操作系统在动态重定位中必须解决的问题（续）

表 15.3　　动态重定位：操作系统的职责

操作系统的要求	解释
内存管理	需要为新进程分配内存 从终止的进程回收内存 一般通过空闲列表（free list）来管理内存
基址/界限管理	必须在上下文切换时正确设置基址/界限寄存器
异常处理	当异常发生时执行的代码，可能的动作是终止犯错的进程

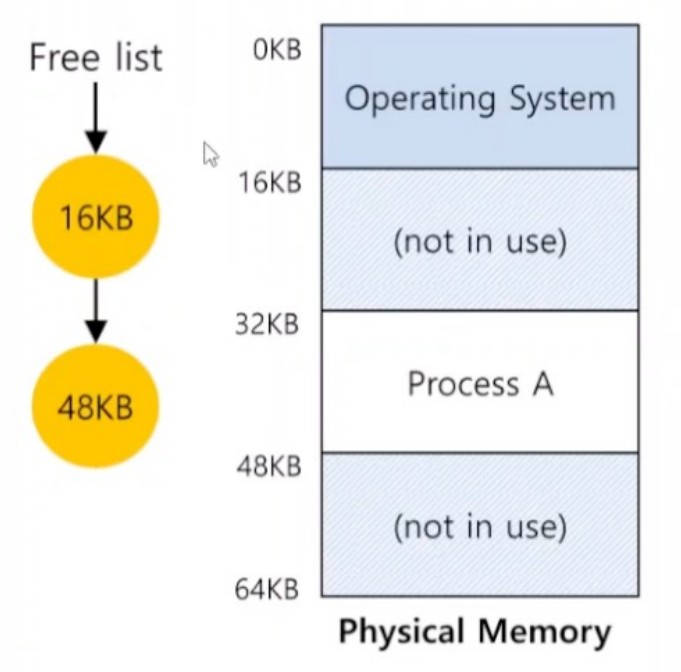
操作系统问题：当进程开始运行时

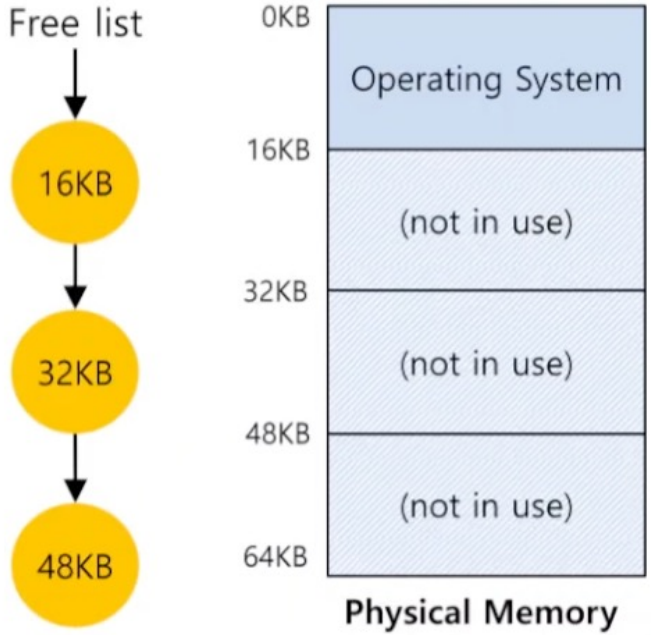
- 操作系统必须**找到一个空闲的地址空间**，以便为新进程分配内存。
  - **空闲列表（Free List）**：物理内存中**未被占用的地址范围**的列表。



操作系统问题：当进程终止时

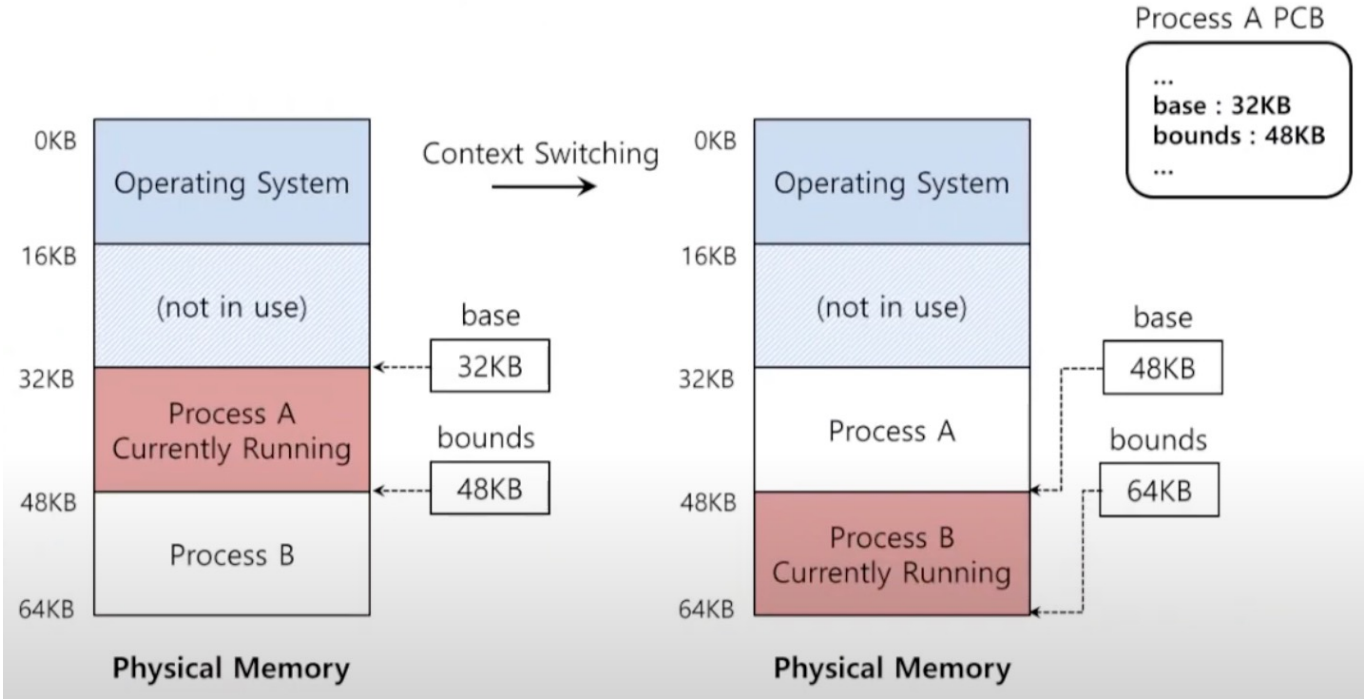
- 操作系统必须将内存归还到空闲列表（free list）





操作系统问题：当上下文切换发生时

- 操作系统必须**保存和恢复**基址寄存器和界限寄存器的值。
  - 存储在进程结构（Process Structure）**或**进程控制块（PCB, Process Control Block）中



操作系统问题：在启动时使用动态重定位的受限直接执行

表 15.4　　受限直接执行协议（动态重定位）

操作系统@启动（内核模式）	硬件	
初始化陷阱表		
	记住以下地址： 系统调用处理程序 时钟处理程序 非法内存处理程序 非常指令处理程序	
开始中断时钟		
	开始时钟，在 x ms 后中断	
初始化进程表 初始化空闲列表		

操作系统问题：运行时的有限直接执行与动态重定位

操作系统@运行（核心模式）	硬件	程序（用户模式）
为了启动进程 A： 在进程表中分配条目 为进程分配内存 设置基址/界限寄存器 从陷阱返回（进入 A）		
	恢复 A 的寄存器 转向用户模式 跳到 A（最初）的程序计数器	
		进程 A 运行 获取指令
	转换虚拟地址并执行获取	
		执行指令
	如果显式加载/保存 确保地址不越界 转换虚拟地址并执行 加载/保存	
		……
	时钟中断 转向内核模式 跳到中断处理程序	

操作系统问题：运行时的有限直接执行与动态重定位

操作系统@启动（内核模式）	硬件	
处理陷阱 调用 switch()例程 　将寄存器（A）保存到进程结构（A） 　（包括基址/界限） 　从进程结构（B）恢复寄存器（B） 　（包括基址/界限） 从陷阱返回（进入 B）		
	恢复 B 的寄存器 转向用户模式 跳到 B 的程序计数器	
		进程 B 运行 　执行错误的加载
	加载越界 转向内核模式 跳到陷阱处理程序	
处理本期报告 　决定终止进程 B 　回收 B 的内存 　移除 B 在进程表中的条目		

动态重定位的其他问题：内部碎片

- **内部碎片**：已经分配的内存单元内部有未使用的空间（即碎片），造成了浪费
  - 重定位的进程使用了32KB～48KB的物理内存（见P15内容）
  - 由于该进程的栈区和堆区并不很大，导致这块内存区域中大量的空间被浪费
  - 需要更复杂的机制，以便更好地利用物理内存，避免内部碎片
    - 分页（paging）/分段（segmentation）

小结

介绍了虚拟内存使用的一种特殊机制，即地址转换（address translation），

介绍了地址转换的4个优点：保护、重定位、数据共享、多路复用

介绍了地址空间重定位的方法：基址加界限寄存器的动态重定位