

进程虚地址空间管理 和延迟的内存分配技术

关键数据结构 1

page结构和mem_map数组

- 每个主存页框有一个Page结构
登记这个主存页框的使用情况
- mem_map是Page结构数组
系统用主存页框号，查数组得Page结构

Type	Name	Description
unsigned long	flags	Array of flags (see Table 8-2). Also encodes the zone number to which the page frame belongs.
atomic_t	_count	Page frame's reference counter.
atomic_t	_mapcount	Number of Page Table entries that refer to the page frame (-1 if none).
unsigned long	private	Available to the kernel component that is using the page (for instance, it is a buffer head pointer in case of buffer page; see " Block Buffers and Buffer Heads " in Chapter 15). If the page is free, this

可以使用/proc文件系统和pmap (1)工具查看给定进程的内存空间和其中所含的内存区域。我们来看一个非常简单的用户空间程序的例子，它其实什么也不做，仅仅是为了做说明用：

```
int main(int argc, char *argv[])
{
    return 0;
}
```

下面列出该进程地址空间中包含的内存区域。其中有代码段、数据段和bss段等。假设该进程与C库动态连接，那么地址空间中还将分别包含libc.so和ld.so对应的上述三种内存区域。此外，地址空间中还要包含进程栈对应的内存区域。

/proc/<pid>/map的输出显示了该进程地址空间中的全部内存区域：

```
rml@phantasy:~$ cat /proc/1426/maps
```

```
00e80000 - 00faf000 r-xp 00000000 03:01 208530 /lib/tls/libc-2.3.2.so
00faf000 - 00fb2000 rw-p 0012f000 03:01 208530 /lib/tls/libc-2.3.2.so
00fb2000 - 00fb4000 rw-p 00000000 00:00 0
08048000 - 08049000 r-xp 00000000 03:03 439029 /home/rml/src/example
08049000 - 0804a000 rw-p 00000000 03:03 439029 /home/rml/src/example
40000000 - 40015000 r-xp 00000000 03:01 80276 /lib/ld-2.3.2.so
40015000 - 40016000 rw-p 00015000 03:01 80276 /lib/ld-2.3.2.so
4001e000 - 4001f000 rw-p 00000000 00:00 0
bffffe00 - c0000000 rwxp fffff000 00:00 0
```

libc.so的代码
数据和bss.
example程序的代
码和数据
ld.so的代码、数据和
bss.

bss和
数据
段

堆栈

偏移 主设备号 次设备号 文件

每行数据格式如下：

开始-结束 访问权限 偏移 主设备号 次设备号 i节点 文件

(1) 工具将上述信息以更方便阅读的形式输出：

进程的地址空间！

内存区域 memory region

- 一个内存区域就是一个逻辑段。在进程的虚地址空间中，每个逻辑段占连续空间，尺寸是整数个逻辑页面。所有逻辑页面的访问权限相同，系统处理这些逻辑页面的方法也相同。
- 内存区域的属性
 - 有文件的段。缺页时，会读文件填充主存页框
 - 代码段 只读，可执行，有文件
缺页时，系统会读一个磁盘文件填充分配给代码段的主存页框
 - 数据段 可读，可写，不可执行，有文件
第一次缺页时，系统会读一个磁盘文件填充分配给数据段的主存页框
 - mmap的段 可读，可写，不可执行，有文件
第一次缺页时，系统会读一个磁盘文件填充分配给代码段的主存页框。mmap的段关闭时，主存页框中的内容刷回磁盘

虚空间的起始地址 **虚空间的结束地址** **文件** **文件中的偏移量**

内存区域 memory region

- 内存区域的属性 续 匿名段，没有文件。
缺页时，用0填充物理页框
 - BSS段
 - heap
 - 共享内存的段
 - Stack VM_GROWSDOWN

虚空间的起始地址 虚空间的结束地址 ~~文件——文件中的偏移量~~

关键数据结构 2

vm_area_struct是内存区域描述符， 每个内存区域一个

Type	Field	Description
<code>struct mm_struct *</code>	<code>vm_mm</code>	Pointer to the memory descriptor that owns the region.
<code>unsigned long</code>	<code>vm_start</code>	First linear address inside the region.
<code>unsigned long</code>	<code>vm_end</code>	First linear address after the region.
<code>struct</code>	<code>vm_next</code>	Next region in the process list.
<code>vm_area_struct *</code>		
<code>pgprot_t</code>	<code>vm_page_prot</code>	Access permissions for the page frames of the region. PTE中的访问控制比特 prot
<code>unsigned long</code>	<code>vm_flags</code>	Flags of the region.
<code>struct rb_node</code>	<code>vm_rb</code>	Data for the red-black tree (see later in this chapter).
<code>union</code>	<code>shared</code>	Links to the data structures used for reverse mapping (see the section " Reverse Mapping for Mapped Pages " in Chapter 17).

关键数据结构 2

vm_area_struct 和 内存区域

<code>struct list_head</code>	<code>anon_vma_node</code>	Pointers for the list of anonymous memory regions (see the section " Reverse Mapping for Anonymous Pages " in Chapter 17).
<code>struct anon_vma *</code>	<code>anon_vma</code>	Pointer to the <code>anon_vma</code> data structure (see the section " Reverse Mapping for Anonymous Pages " in Chapter 17).
<code>struct</code>	<code>vm_ops</code>	操作这个内存区域的函数 the memory region.
<code>vm_operations_struct *</code>		
<code>unsigned long</code>	<code>vm_pgoff</code>	内存区域在磁盘文件中的起始地址（偏移量，以字节计） anonymous pages, it is either zero or equal to <code>vm_start/PAGE_SIZE</code> (see Chapter 17).
<code>struct file *</code>	<code>vm_file</code>	指向负责填充这个内存区域的磁盘文件，可执行文件/函数库
<code>void *</code>	<code>vm_private_data</code>	Pointer to private data of the memory region.

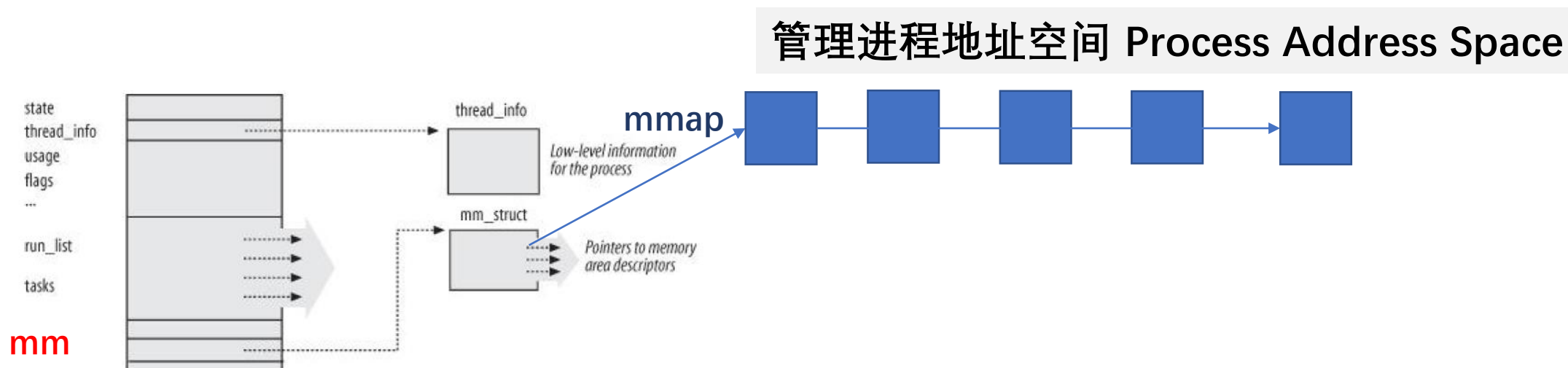
关键数据结构 2

vm_area_struct中的vm_ops

Method	Description
--------	-------------

<code>open</code>	Invoked when the memory region is added to the set of regions owned by a process.
<code>close</code>	Invoked when the memory region is removed from the set of regions owned by a process.
<code>nopage</code>	Invoked by the Page Fault exception handler when a process tries to access a page not present in RAM whose linear address belongs to the memory region (see the later section " Page Fault Exception Handler ").
<code>populate</code>	Invoked to set the page table entries corresponding to the linear addresses of the memory region (prefaulting). Mainly used for non-linear file memory mappings.

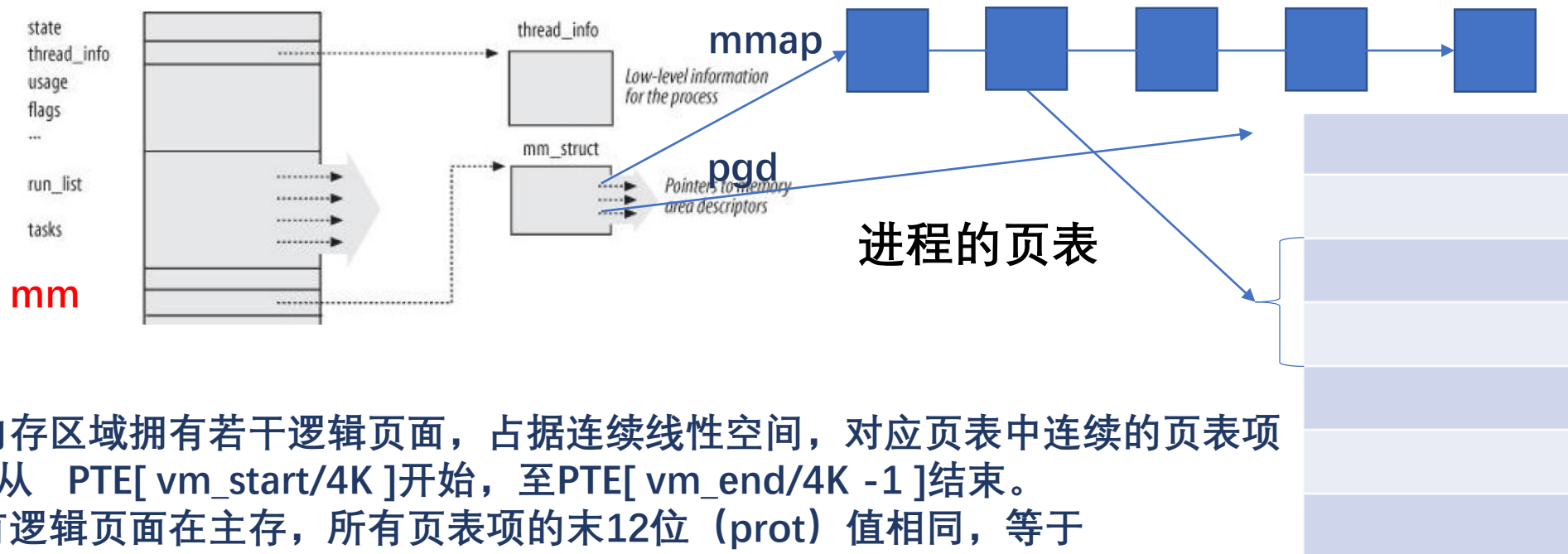
关键数据结构 mmap



vm_area_struct按内存区域的起始地址递增排序。 所有vm_area_struct同时挂在vm_area_struct 链表 和 红黑树上

发生缺页时，系统用产生缺页的线性地址搜索红黑树定位包含所缺地址的内存区域
进程运行时会插入、删除内存区域。系统会搜索红黑树，定位

关键数据结构 内存区域和页表的关系



- 1、每个内存区域拥有若干逻辑页面，占据连续线性空间，对应页表中连续的页表项
从 $PTE[vm_start/4K]$ 开始，至 $PTE[vm_end/4K - 1]$ 结束。
- 2、若所有逻辑页面在主存，所有页表项的末12位 (prot) 值相同，等于
vm_area_struct 中的 vm_page_prot 分量：

- 代码 U RO X P
- 变量 U RW \bar{X} P (数据/堆栈/mmap的文件/共享内存)
- 常量 U RO \bar{X} P

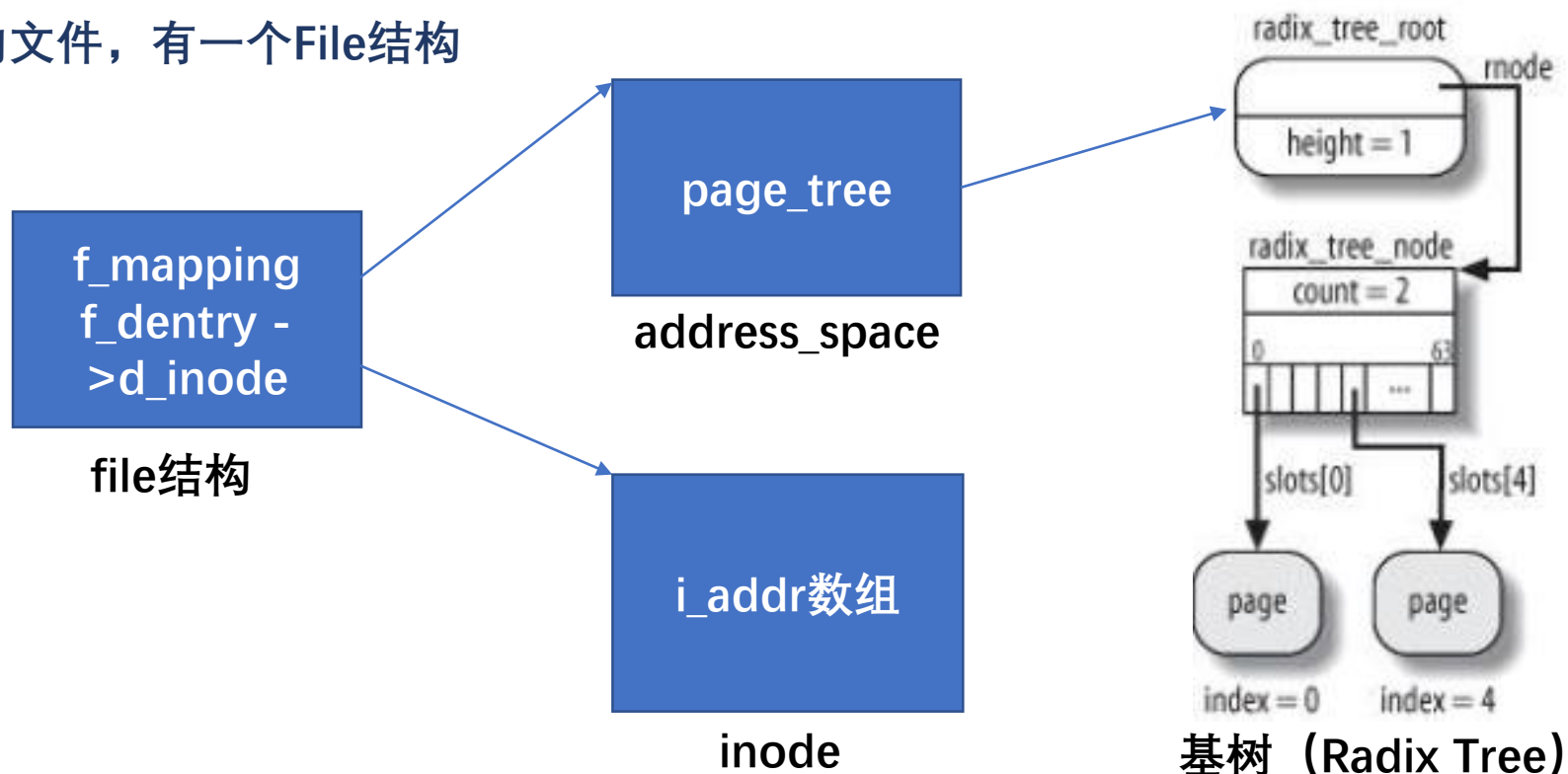
补：Linux文件系统

1、每个正在使用的文件有一个内存inode 和一个address_space

- 内存inode 记录存放文件数据的磁盘地址。系统用文件中的偏移量搜索inode，得数据的磁盘地址。
- address_space 中有一棵radix树，树上非空的叶子节点表示缓存着文件数据的主存页框。

例：若叶子节点 j 指向 $\text{page}[i]$ ，表示 i 号主存页框中装着该文件的 j 号逻辑页面： $[j*4K, (j+1)*4K]$

2、每个使用中的文件，有一个File结构

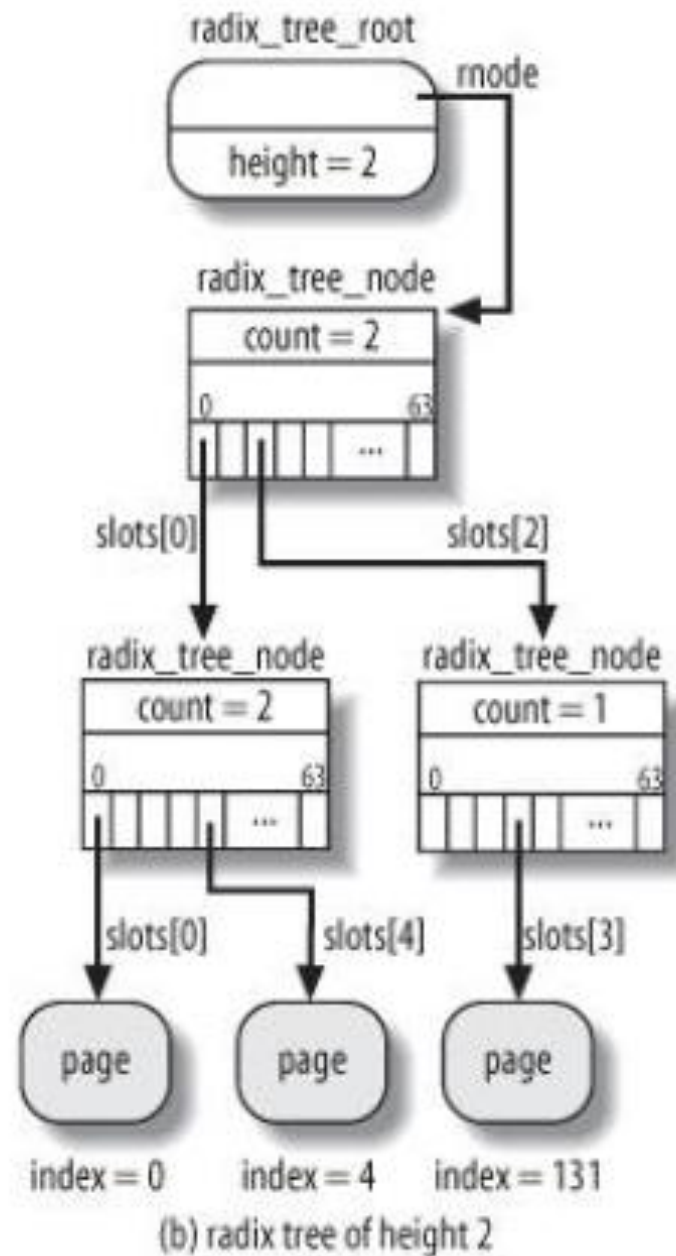
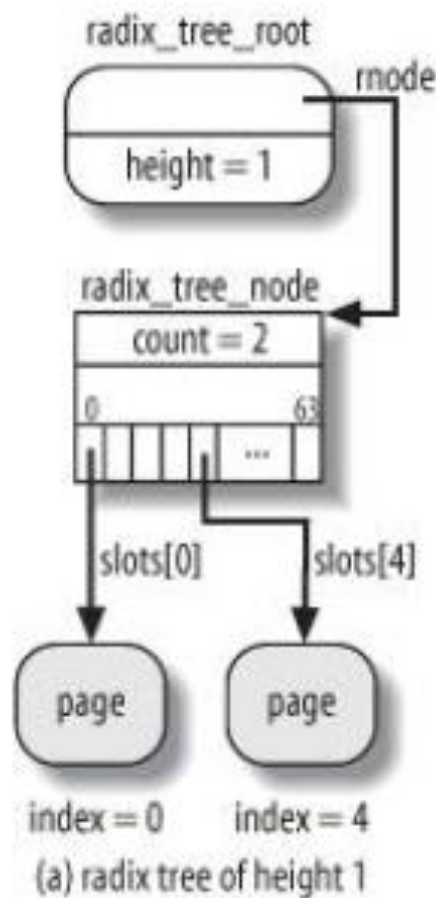


15.1.2 radix tree

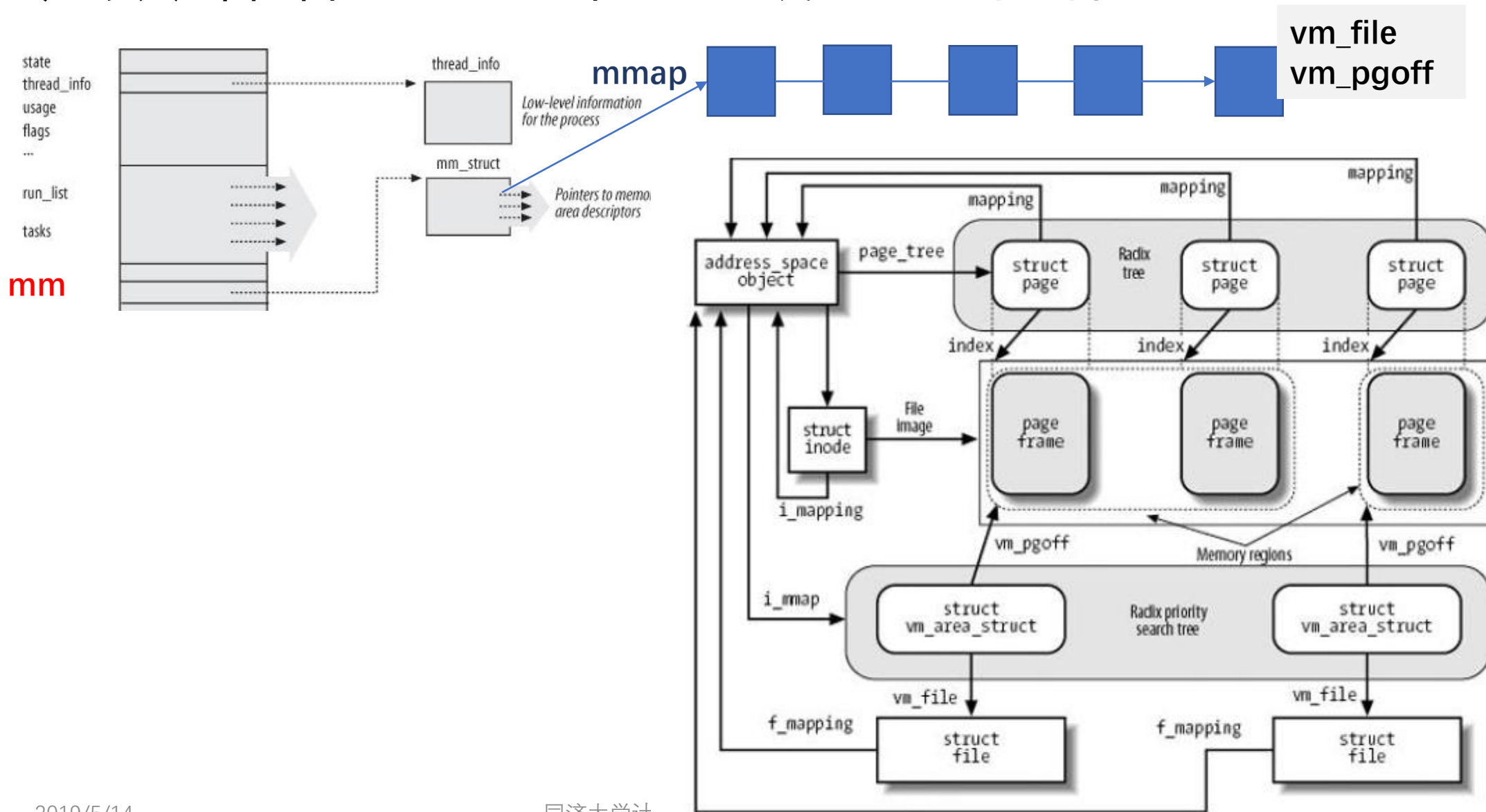
基树有多高，有多少个slot，是由文件的大小决定的。

基树有多少张叶子（非空slot），是由分配给文件的主存页框数决定的。

系统使用一个新文件，刚开始的时候，基树没叶子。随着文件读写操作，基树的叶子会多起来。



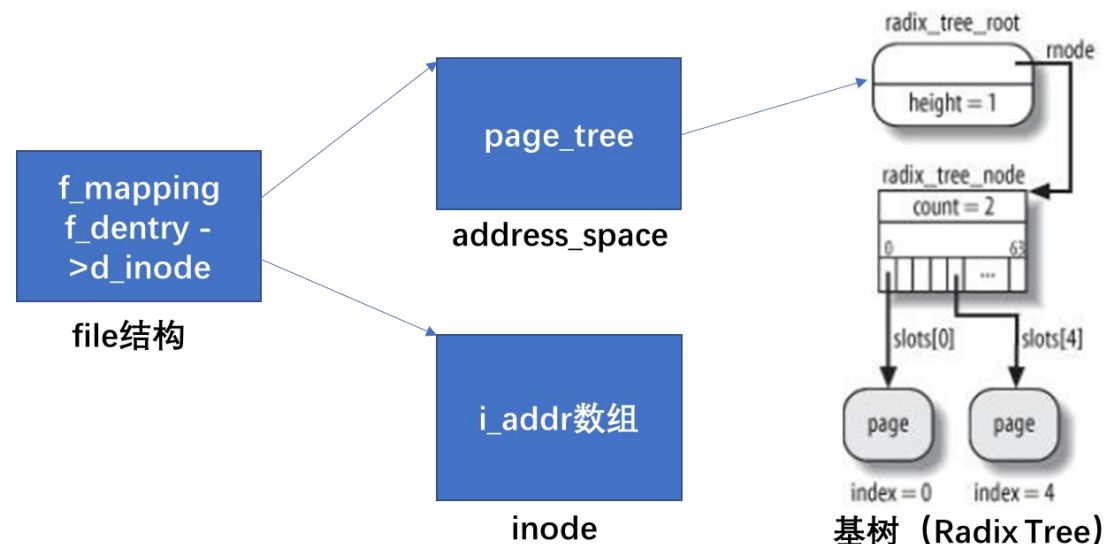
关键数据结构 内存区域和文件的关系



Linux文件读操作例题：读文件A的第5000字节

- 1、 $5000/4096 == 1$ ，读操作的语义是：读文件A的1#逻辑页面中的904字节
- 2、用file结构找到文件A的address_space和radix树
- 3、用逻辑页号1作为索引，查radix树。找1#叶子节点
 - 非空（2345）。2345#页框中就是A的1#逻辑页面
 - 空。
 - 为文件A追加一个主存页框9876，把9876号页框挂在Radix树上。
 - 查内存inode，读i_addr[1]，得1#逻辑页面在磁盘上的地址：物理块号Y
 - 发DMA命令：读磁盘物理块Y，写内存物理地址 $9876 \times 4K$ ，传4096个字节
 - 入睡，等待IO结束
- 4、读物理页框（2356或9876）中的第904字节，送用户空间变量

PS：不考虑STDIO用户空间缓存



16.2.1. Memory Mapping Data Structures

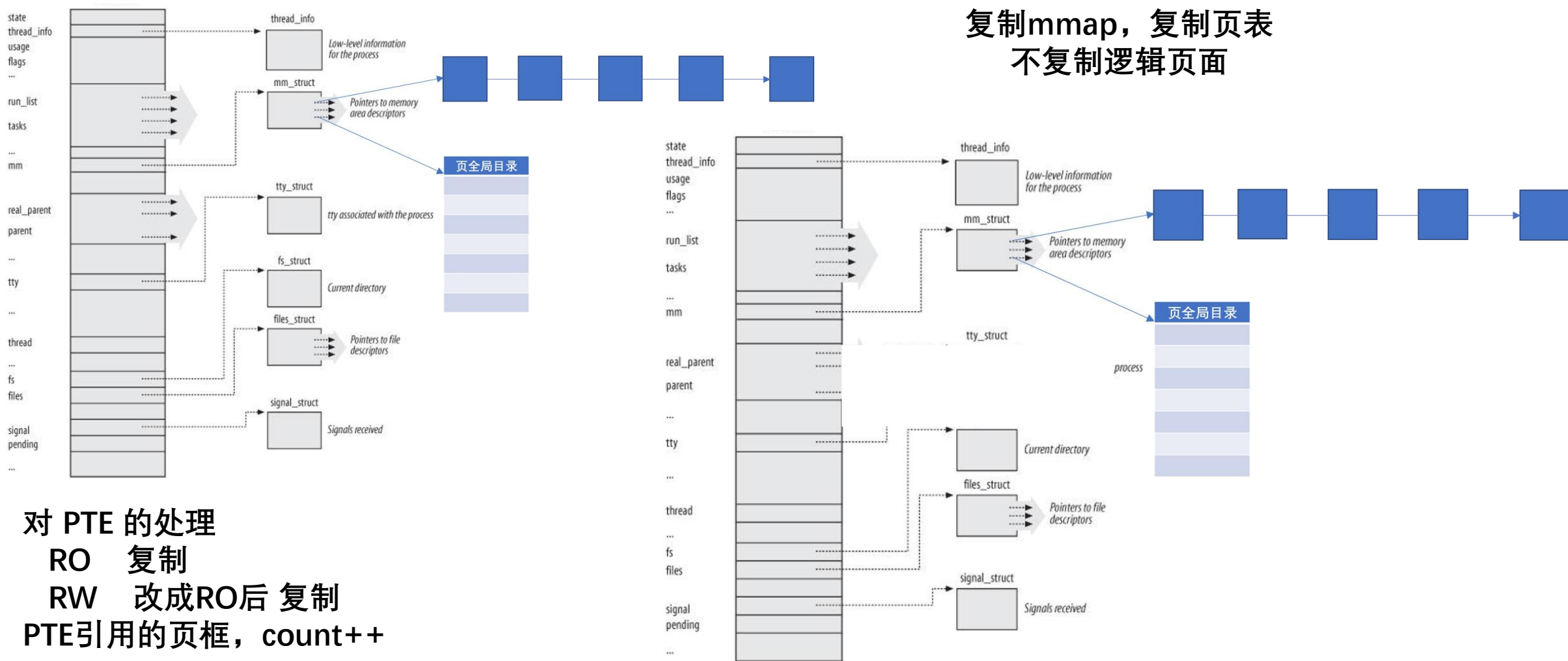
A memory mapping is represented by a combination of the following data structures :

- The inode object associated with the mapped file
- The `address_space` object of the mapped file
- A file object for each different mapping performed on the file by different processes
- A `vm_area_struct` descriptor for each different mapping on the file
- A page descriptor for each page frame assigned to a memory region that maps the file

[Figure 16-2](#) illustrates how the data structures are linked. On the left side of the image we show the inode, which identifies the file. The `i_mapping` field of each inode object points to the `address_space` object of the file. In turn, the `page_tree` field of each `address_space` object points to the radix tree of pages belonging to the address space (see the section "[The Radix Tree](#)" in [Chapter 15](#)), while the `i_mmap` field points to a second tree called the radix priority search tree (PST) of memory regions belonging to the address space. The main use of PST is for performing "reverse mapping," that is, for identifying quickly all processes that share a given page. We'll cover in detail PSTs in the next chapter, because they are used for page frame reclaiming. The link between file objects relative to the same file and the inode is established by means of the `f_mapping` field.

Each memory region descriptor has a `vm_file` field that links it to the file object of the mapped file (if that field is null, the memory region is not used in a memory mapping). The position of the first mapped location is stored into the `vm_pgoff` field of the memory region descriptor; it represents the file offset as a number of page-size units. The length of the mapped file portion is simply the length of the memory region, which can be computed from the `vm_start` and `vm_end` fields.

进程创建 fork, 对mm_struct的影响



对 PTE 的处理
RO 复制
RW 改成RO后 复制
PTE引用的页框, count++

缺页异常处理程序（父进程所有页面在内存，不考虑非法内存访问的情况）

- COW，写时复制（Copy On Write）
- 访问代码页面或只读数据页面，不会引发缺页，进程正常运行
- 访问同一张数据页面，逻辑页号是Y。父进程和子进程都会因为写RO的页面引发缺页。也就是会触发2次缺页。

- 第一次: Page[old] 的count>1, 触发COW, 缺页异常处理程序执行: count--, 分配新主存页框new, 将页框old中的数据复制到new中, 修改PTE[Y].base = Y; PTE[Y].prot 改回RW。刷快表。

缺页异常返回, 重新访问引发缺页的数据。这一次, check通过, 进程正常运行。

总结: 第一个访问数据页面的进程, 得到新页框, 触发复制操作。但是, 另一个进程的数据页面仍然是RO属性, 这需要第二次缺页异常处理程序来改正。

- 第二次: Page[old] 的count==1, 不会触发COW和主存页框分配。缺页异常处理程序简单将 PTE[Y].prot 改回RW。刷快表。就可以返回。

COW 细节: 复制 页框 中的 数据

return VM_FAULT_MINOR;
(2) If the page is shared among several processes by means of COW, the function copies the content of the old page frame (old_page) into the newly allocated one (new_page).
(3) To avoid race conditions, get_page() is invoked to increase the usage counter of old_page before starting the copy operation:

```
old_page = pte_page(pte);  
pte_unmap(page_table);  
get_page(old_page);  
spin_unlock(&mm->page_table_lock);  
if (old_page == virt_to_page(empty_zero_page))  
    new_page = alloc_page(GFP_HIGHUSER | __GFP_ZERO);  
} else {  
    new_page = alloc_page(GFP_HIGHUSER);  
    vfrom = kmap_atomic(old_page, KM_USER0);  
    vto = kmap_atomic(new_page, KM_USER1);  
    copy_page(vto, vfrom);  
    kunmap_atomic(vfrom, KM_USER0);  
    kunmap_atomic(vto, KM_USER1);  
}
```

老的页框内为物理框号

如果复制的是zero页面
不复制了, 新页面填0.

使用虚拟地址常量
KM_USER0 和
KM_USER1.

建立映射, 之后实施复制

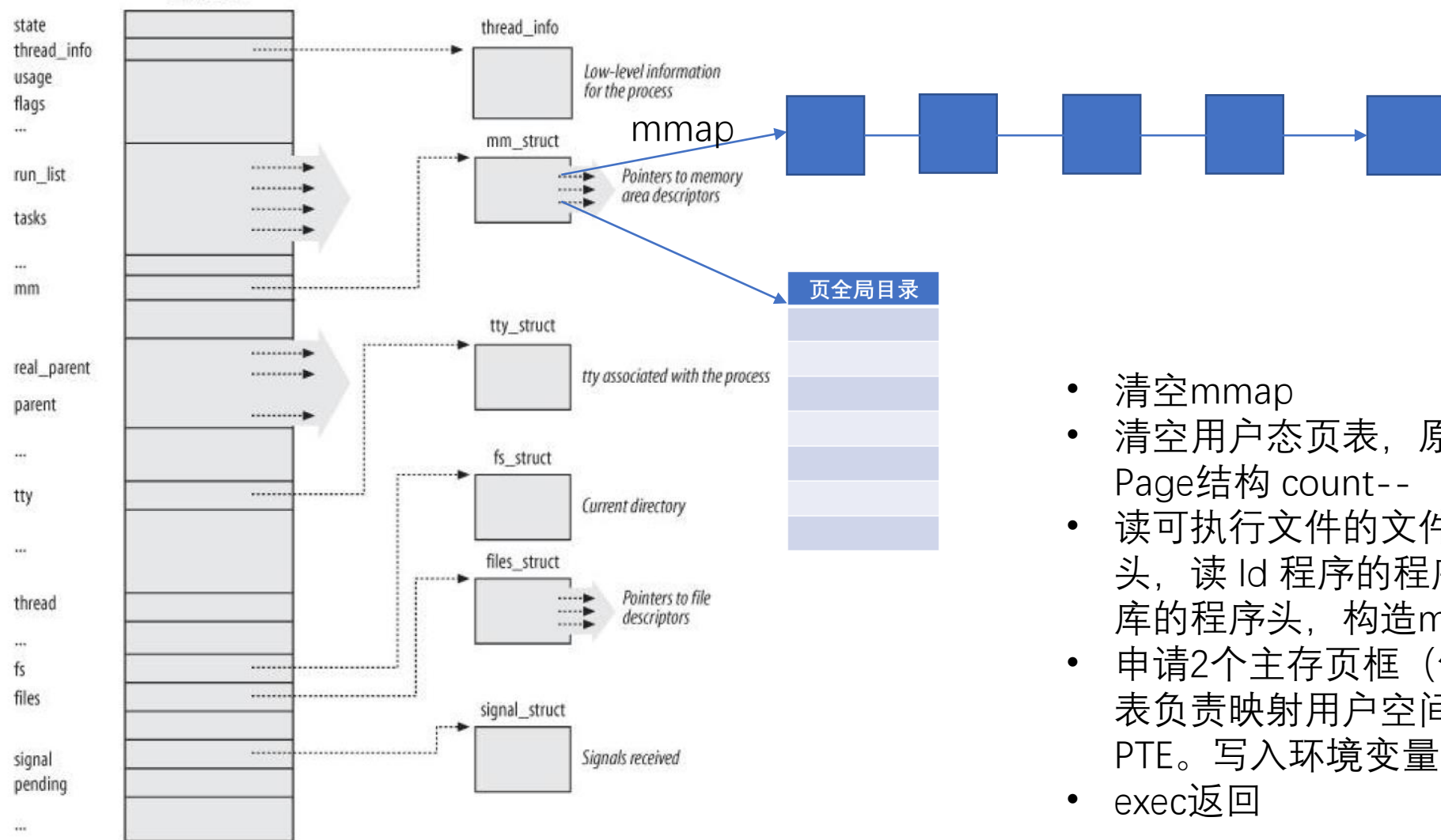
If the old page is the zero page, the new frame is efficiently filled with zeros when it is allocated (__GFP_ZERO flag). Otherwise, the page frame content is copied using the copy_page() macro. Special handling for the zero page is not strictly required, but it improves the system performance, because it preserves the microprocessor hardware cache by making fewer address references.

会改善硬件的性能

block the process, the function check

exec

清空、重建mmap
清空用户态页表，所有PTE是0



- 清空mmap
- 清空用户态页表，原来页表映射的所有Page结构 count--
- 读可执行文件的文件头，读标准C库的程序头，读ld程序的程序头，读所有动态链接库的程序头，构造mmap链表
- 申请2个主存页框（做初始用户栈），写页表负责映射用户空间最后2个逻辑页面的PTE。写入环境变量和命令行参数
- `exec`返回

可执行程序 and mmap 重建过程 1

程序头

程序头
第1个逻辑段，代码段
第2个逻辑段，只读数据段
第3个逻辑段，数据段
其它逻辑段 ~m (不太有需要装到 进程图像中的段了)
符号表

a
b
c

sth about 整个程序，比如有几个段头，程序的入口地址 entry point 初始堆栈大小，代码段第一条指令的线性地址 imageBegin
1# 段头 section header
2# 段头 section header
3# 段头 section header
4# 段头 section header
~ n# 段头

有几个段头，程序的入口地址 entry point
初始堆栈大小，代码段第一条指令的线性地址 imageBegin

代码段

只读数据段

数据段

BSS段

有没有堆栈段？？

section header:

vm_start,
vm_end,
vm_pgoff (文件中的起始地址)
文件中段的长度,
prot

求所有段头

可执行程序 and mmap 重建过程 2

程序头

sth about 整个程序, 比如有几个段头, 程序的入口地址 entry point
初始堆栈大小, 代码段第一条指令的线性地址 imageBegin

1# 段头 section header	代码段
2# 段头 section header	只读数据段
3# 段头 section header	数据段
4# 段头 section header	BSS段
~ n# 段头	堆栈段

section header:
vm_start,
vm_end,
vm_pgoff ,
文件中段的长度,
prot

- 读 可执行程序 的每个段头, 构造vm_area_struct, 插入mmap红黑树 和 vm_area_struct列表
- 以3G为vm_end, 3G-初始栈大小为vm_begin, 为stack构造vm_area_struct
- 读 ld 程序的.....
- 确定 所有 include 的动态链接库 (.so程序), 确定线性空间中每个逻辑段的起始地址。结合动态链接库程序头中的段头, 做出来vm_area_struct, 插入mmap红黑树 和 vm_area_struct列表

请求调页 硬件基础

- Page fault 发生时，硬件会
- CR2：产生缺页异常的线性地址
- 压栈 error code：
 - bit0 是 0，异常是因为present bit
1，异常是因为访问权限不对
 - bit1 是 0，异常发生时，CPU执行读操作或执行指令
1，~写
 - bit2 是 0，异常发生时，核心态运行
1，~ 用户态运行

第一步：定位缺页所在的 memory region

- 取CR2值，查红黑树（略去禁止用户态代码访问内核空间的判断）
- 存在 $CR2 \in [vm_start, vm_end)$ 的节点。看error code和引发缺页的PTE，是不是写了RO页面，是
 - memory region是可写的嘛？ 是的，COW。 否则，段错误。
- 不存在 $CR2 \in [vm_start, vm_end)$ 的节点，是要扩展堆栈嘛？
 - 是的， 合法内存访问。请求调页。
 - 不是的，访问非法内存区域，给现运行进程送 SIGSEGV 信号。段错误，现运行进程被杀死。


PS：系统会调用这个memory region负责缺页异常处理的的 no_page函数。
下面看Linux针对不同情况执行的异常处理过程。记下来定位得到的memory region。

案例：进程PA， exec (example) 之后立即fork子进程PB。之后，这两个进程共同承担example程序的任务

- 缺代码或只读数据页面 (memory region 是RO的)

- 所缺地址在文件中的偏移量 =
 $CR2 - vm_start + vm_pgoff$

叶子原本就存在，不会IO磁盘， minor page fault
叶子原本不存在， IO磁盘， 进程会睡， major page fault

- 用计算所得文件中的偏移量，查radix树~~~具体过程见PPT14。
- 文件中数据内存就绪后，记radix树上与该地址对应的叶子节点是page[j]，
 $PTE[CR2/4K].base = j$
 $PTE[CR2/4K].prot = vm_area_struct$ 中的prot
- 刷快表， page[j].count++，缺页异常返回

案例：进程PA， exec (example) 之后立即fork子进程PB。之后，这两个进程共同承担example程序的任务

- 缺数据页面 (memory region RW, 有文件)
 - PB缺，PA访问过的页面
PTE非空，页框 count>1, present是1 COW
 - PB缺，PA没访问过的页面，分配主存页框，填充不同的数据
 - 第1次缺 PTE空 去radix树，读可执行文件，用全局变量的初值fill
 - 第2次缺 PTE不空 去swap分区，读曾经swap out的页面，用磁盘上保存的页面fill
 - PA随后访问这张页面。 同PB，第一次缺页和第二次缺页的情况
- 缺页异常返回

案例：进程PA， exec (example) 之后立即fork子进程PB。之后，这两个进程共同承担example程序的任务

- 缺BSS页面，（memory region RW，没有文件，匿名）。和上页PPT类似。唯一区别红色笔迹标识
 - PB缺，PA访问过的页面
 - PB缺，PA没访问过的页面
 - 第1次缺 用0填充分配得到的页框（这里Linux用Zero页，进一步推迟了物理内存的分配，稍后介绍）
 - 第2次缺
 - PA随后访问这张页面
 - 缺页异常返回

案例：进程PA， exec (example) 之后立即fork子进程PB。之后，这两个进程共同承担example程序的任务

- 缺堆栈页面

- CR2在堆栈内存区域

- COW父进程的堆栈页面

- 不在堆栈内存区域，需要扩充堆栈。判断条件 $CR2 + 32 < regs->esp$

- 修改memory region: $vm_start -= 4K$

- 分配一个新主存页框 k,

- $PTE[CR2/4K].base = k$

- $PTE[CR2/4K].prot = vm_area_struct \text{ 中的 } prot$

- 缺页异常返回

案例：进程PA malloc申请动态内存

- 只修改memory region `vm_end += 新申请的动态内存的大小`
- 系统调用返回
- PS：此时，系统并没有为动态内存分配主存页框。也没有写与这块虚空间对应的PTEs。这些与内存分配相关的操作**延迟到**进程写这段内存区域的时刻。

Linux延迟内存访问技术

- 尽量延迟为用户数据（逻辑页面）分配主存页框，直至进程真正读写逻辑页面的最后时刻。COW、malloc操作皆如此。
- 另外，在delay方面，Linux走的更远。
 - 进程第一次访问BSS页面，并且是读操作。延迟分配页框。
这是因为不管读哪个变量，读出来的值都是0。 `PTE[CR2/4K].base = ZeroPage` RW
∴ 为BSS分配主存页框，会推迟到第一次写操作的时刻。此时，分配空页框Y，填全0。 `PTE[CR2/4K].base = Y`。

系统的 0 页，页框号是ZeroPage

RAM	内核静态区域	全0	
-----	--------	----	--

最后，有关系统安全

- 系统分配给APP的主存页框，一定要清0！！
里面会有，系统或其它应用的重要数据