BAN CƠ YẾU CHÍNH PHỦ

HỌC VIỆN KỸ THUẬT MẬT MÃ

CHƯƠNG 7. QUẢN LÝ BỘ NHỚ

Ngành: Công nghệ thông tin

Chuyên ngành: Kỹ thuật phần mềm nhúng và di động

Mã số: 52.48.02.01

Nội dung

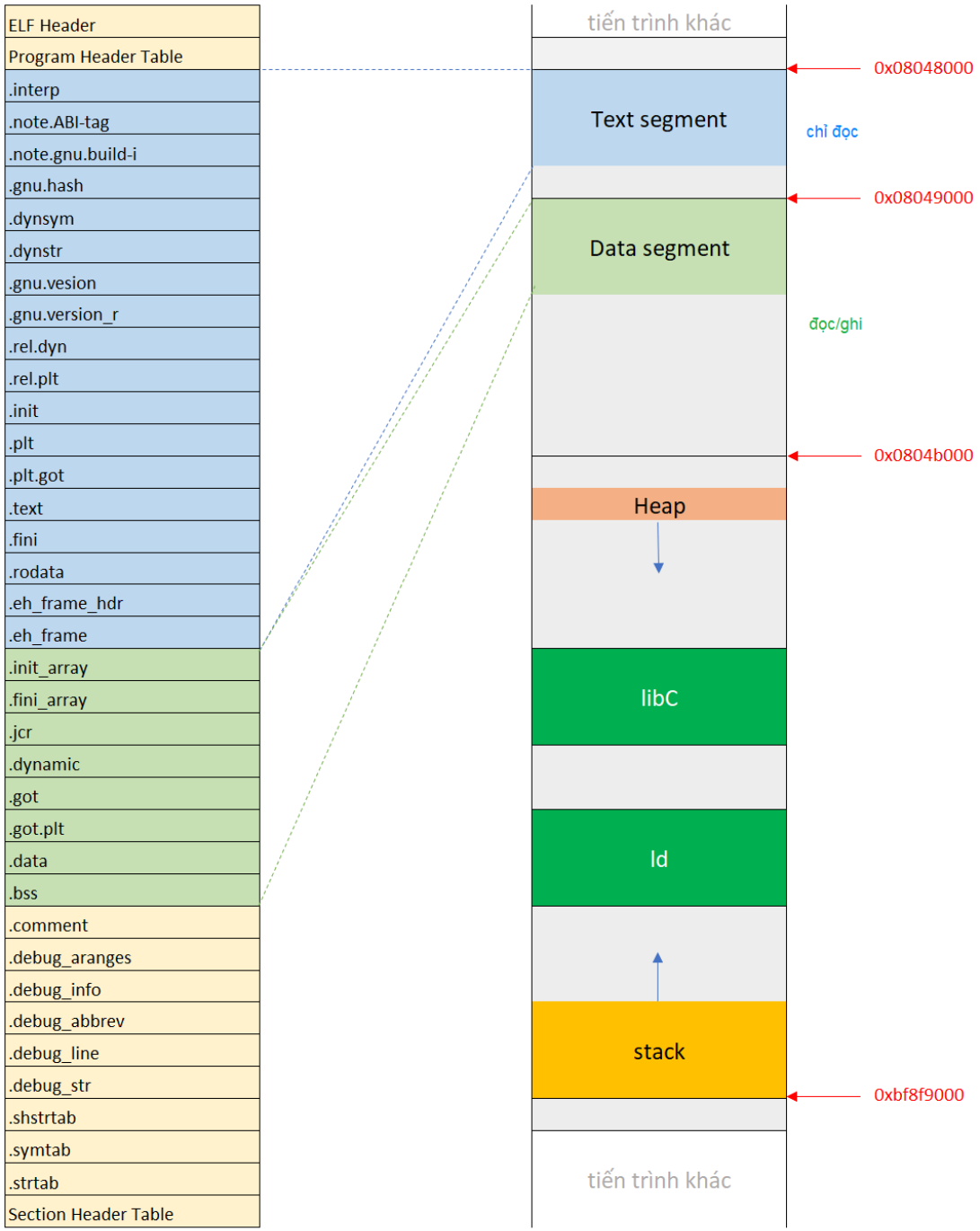
- Mô hình bộ nhớ ảo
- Bộ nhớ ảo trong Linux
- Cấp phát theo byte
- Cấp phát theo page-frame
- Cấp phát theo mem pool
- Truy cập dữ lieu trên mô-đun IO
- Ánh xạ bộ nhớ

Mô hình bộ nhớ ảo

- File thực thi gồm một số section quan trọng
  - Section **.text**: chứa các lệnh của chương trình.
  - Section **.rodata**: chứa các biến chỉ đọc (ví dụ được khai báo với từ khóa const).
  - Section **.data**: chứa các biến toàn cục và biến tĩnh đã được khởi tạo.
  - Section **.bss**: chứa các biến toàn cục và biến tĩnh chưa được khởi tạo

Mô hình bộ nhớ ảo

- Hình bên gồm
  - File thực thi trên ổ cứng
  - Ảnh tiến trình trong RAM



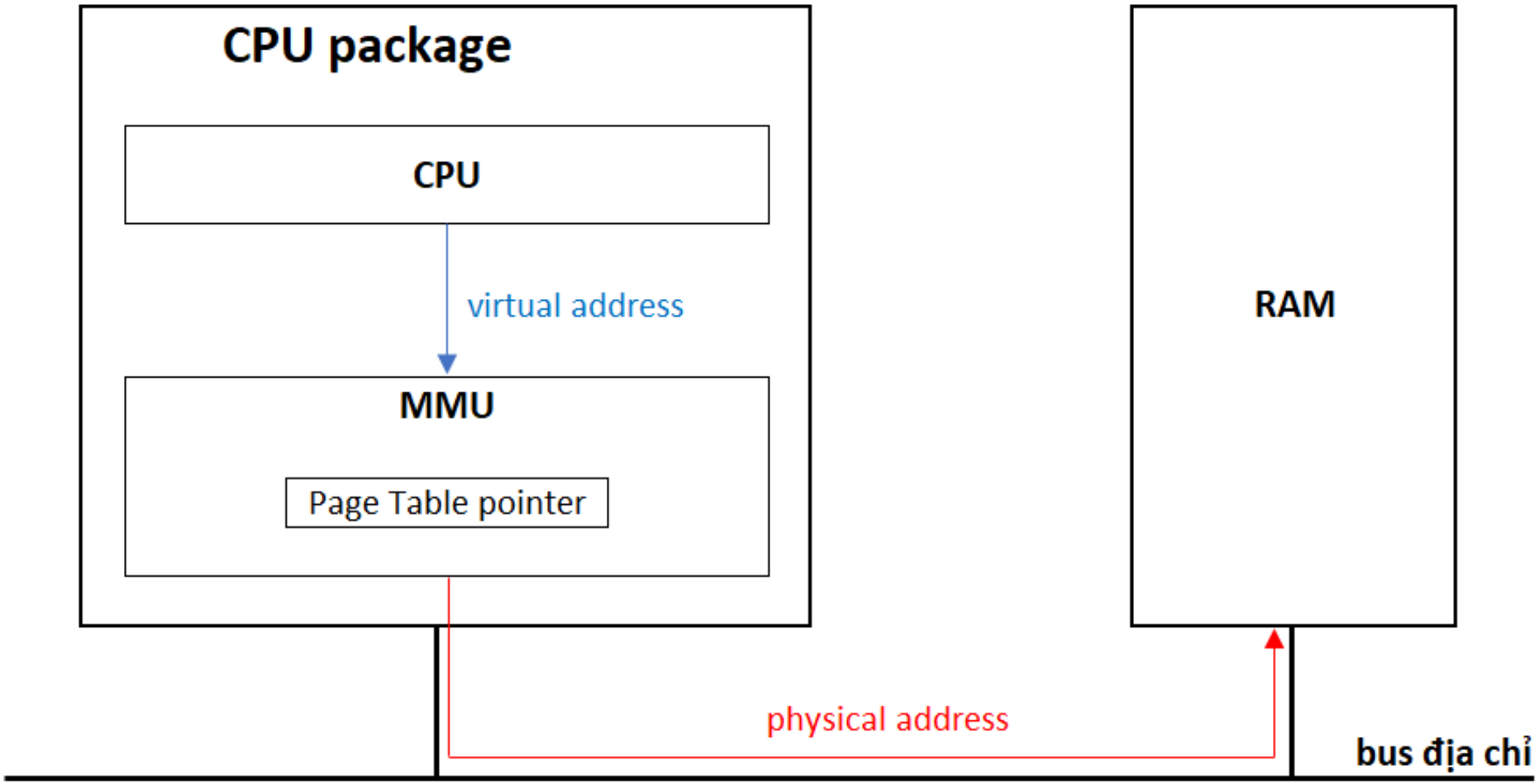
File thực thi trên ổ cứng

Ảnh của tiến trình trên RAM

Mô hình bộ nhớ ảo

- **Cơ chế virtual memory được triển khai**
  - Cơ chế virtual memory được triển khai trên cả phần cứng và phần mềm
  - Về mặt phần cứng, khối MMU (**M**emory **M**anagement **U**nit) được thêm vào hệ thống
    - Khối MMU được đặt giữa CPU và RAM, có nhiệm vụ dịch địa chỉ do CPU phát ra (gọi là địa chỉ ảo hay virtual address) sang địa chỉ để truy cập RAM (gọi là địa chỉ vật lý hay physical address)

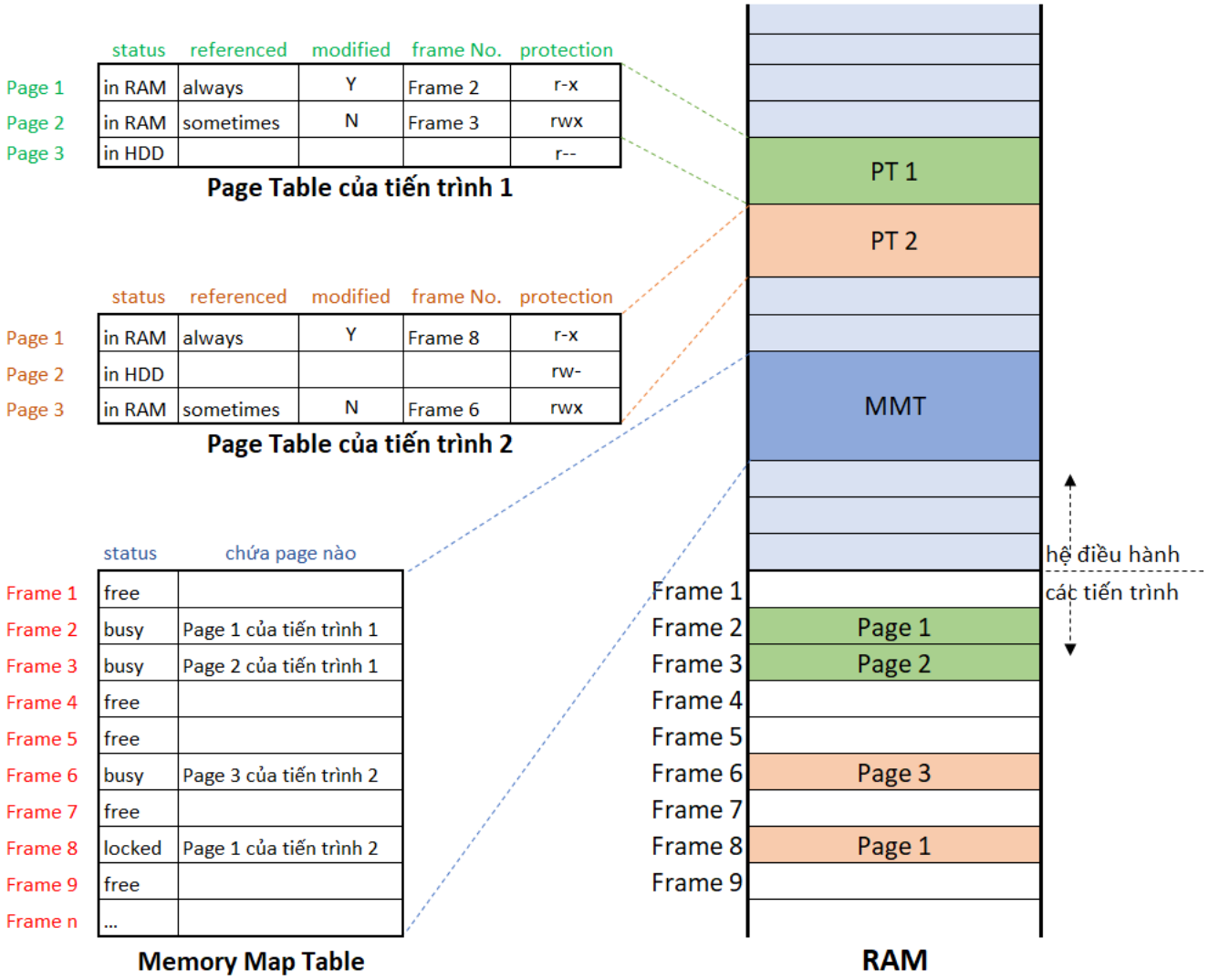
Mô hình bộ nhớ ảo



Mô hình bộ nhớ ảo

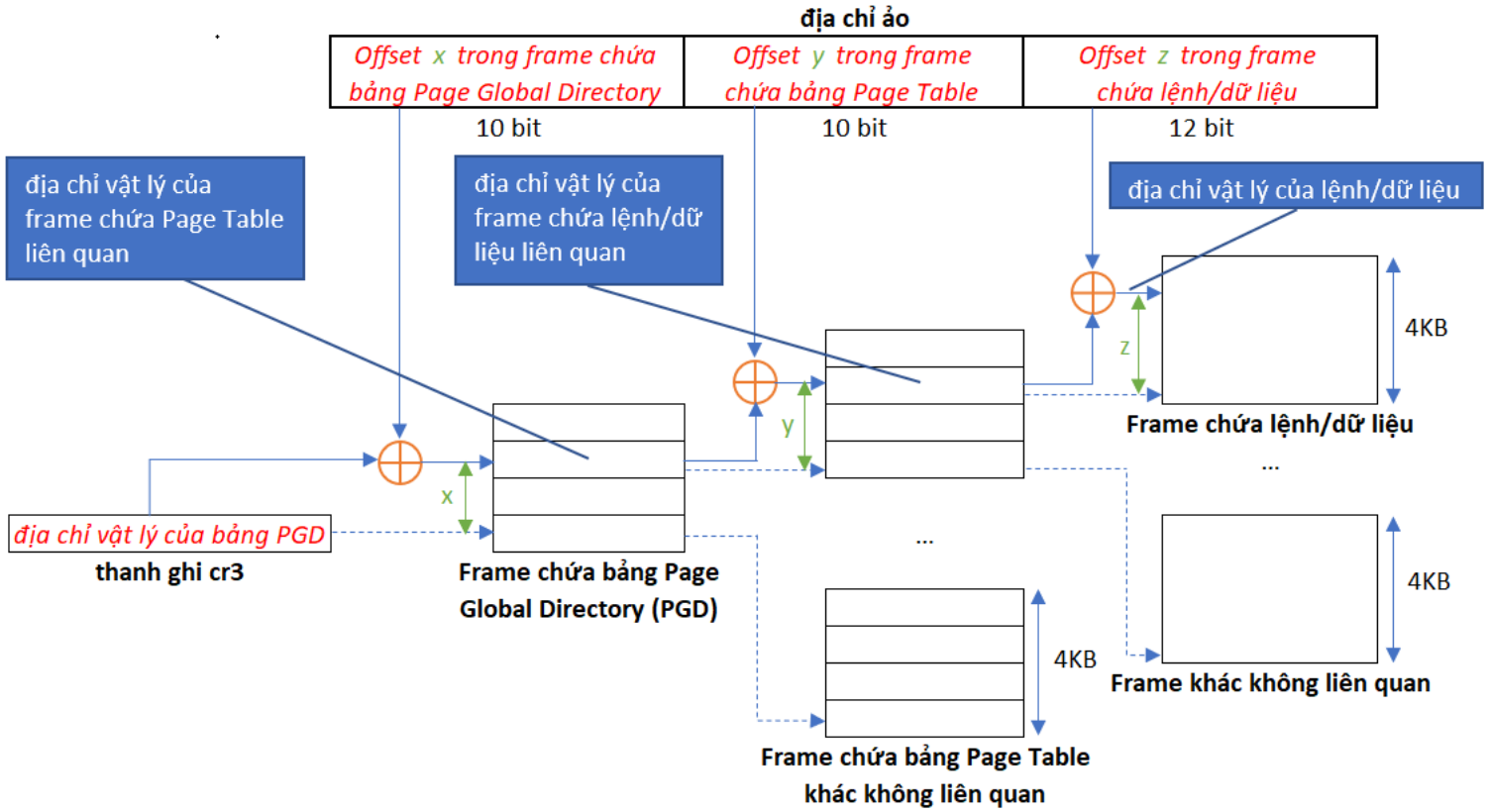
- Về mặt phần mềm, hệ điều hành sẽ:
  - Chia RAM thành nhiều mảnh nhỏ, mỗi mảnh gọi là một frame. Các frame có kích thước giống nhau. Kích thước này tùy từng hệ thống, nhưng thường là 4KB
  - Hệ điều hành tạo ra bảng MMT (**M**emory **M**ap **T**able) để quản lý tất cả các frame trên RAM. Bảng MMT được lưu trên RAM.
  - mỗi tiến trình có một không gian địa chỉ riêng
    - ✓ Không gian địa chỉ của mỗi tiến trình được chia thành nhiều mảnh nhỏ, mỗi mảnh gọi là một page. Các page có kích thước giống nhau và giống với kích thước của frame.
    - ✓ Tương ứng với mỗi tiến trình, hệ điều hành tạo ra một bảng Page Table để quản lý tất cả các page của một tiến trình. Các bảng Page Table cũng được lưu trên RAM.
    - ✓ cập nhật lại thanh ghi Page Table pointer của MMU mỗi khi xảy ra context switch

Mô hình bộ nhớ ảo



Bộ nhớ ảo trong Linux

- Cơ chế dịch địa chỉ ảo sang địa chỉ vật lý trong các hệ thống 32 bit
  - 2-level paging



Địa chỉ vật lý

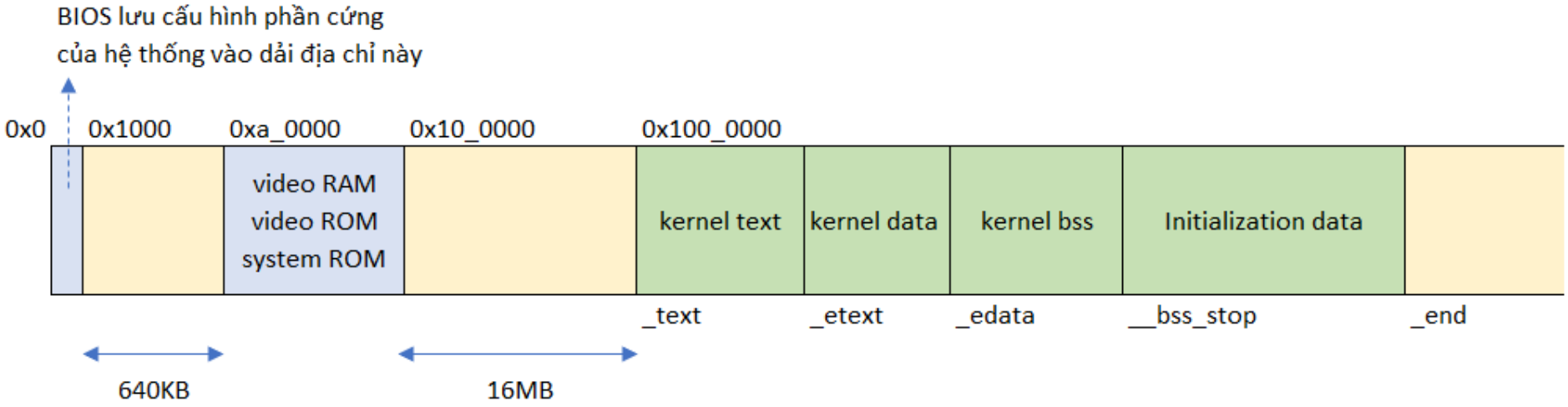
- **Không gian địa chỉ vật lý**
  - Nếu bus địa chỉ gồm 32 đường dây, thì không gian địa chỉ vật lý sẽ là 4GB, kéo dài từ 0x0000_0000 đến 0xFFFF_FFFF. Mỗi số giúp xác định vị trí của một lệnh hoặc dữ liệu trên bộ nhớ vật lý. Bộ nhớ vật lý có thể là:
    - ✓bộ nhớ của hệ thống, bao gồm RAM (bộ nhớ đọc ghi) và ROM (bộ nhớ chỉ đọc).
    - ✓bộ nhớ của một số thiết bị ngoại vi khác (ví dụ card đồ họa).
- Sử dụng lệnh ***cat /proc/iomem***, ta sẽ biết không gian địa chỉ vật lý được quy hoạch như thế nào

Không gian địa chỉ vật lý

- Xét về mặt nội dung, Linux kernel được chia thành 4 vùng:
  - Vùng kernel text chứa các lệnh của kernel (hay kernel code). Vùng này kéo dài từ vị trí _***text*** đến _***etext***.
  - Vùng kernel data chứa các dữ liệu đã được khởi tạo của kernel. Vùng này kéo dài từ vị trí _***etext*** đến _***edata***.
  - Vùng kernel bss chứa các dữ liệu chưa được khởi tạo của kernel. Vùng này kéo dài từ vị trí _***edata*** đến __***bss_stop***.
  - Vùng initialization data chứa các dữ liệu cần thiết cho quá trình khởi động của kernel. Vùng này kéo dài từ __***bss_stop*** đến _***end***. Khi kernel khởi động thành công, vùng này sẽ được giải phóng.
- Giá trị của _***text***, _***etext***, _***edata***, __***bss_stop*** và _***end*** có thể tìm thấy trong file */boot/System.map-$(uname -r)*.
  - Tuy nhiên, các giá trị này đều là địa chỉ ảo, không phải địa chỉ vật lý.

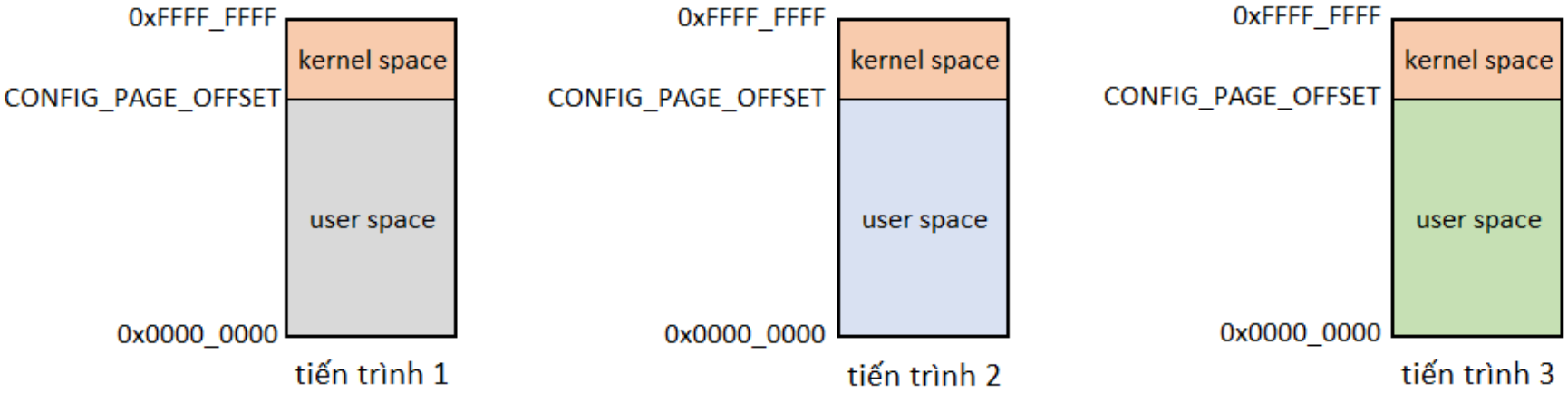
Không gian địa chỉ vật lý

- *Cách bố trí Linux kernel trên không gian địa chỉ vật lý*



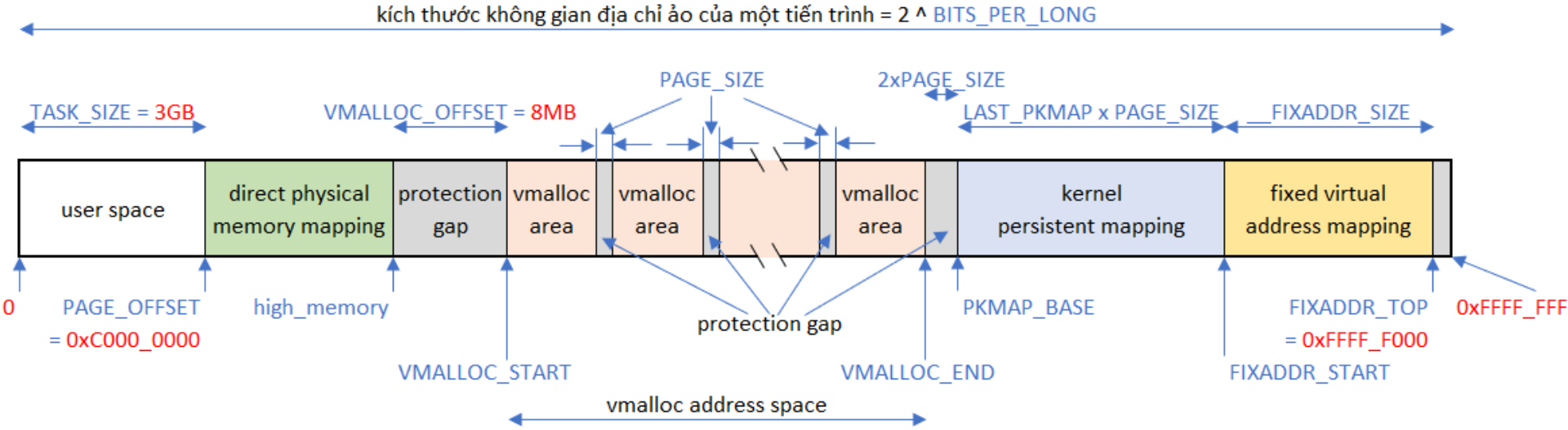
Không gian địa chỉ ảo

- **Linux chia không gian địa chỉ ảo thành 2 phần**
  - Phần **user space**: chứa địa chỉ ảo của các lệnh và dữ liệu trong tiến trình.
  - Phần **kernel space**: một phần kernel space chứa địa chỉ ảo của các lệnh và dữ liệu trong Linux kernel



Không gian nhân

- Kernel space được chia làm 4 vùng:
  - Vùng direct physical memory mapping (gọi ngắn gọn là vùng direct mapping)
  - Vùng vmalloc address space (gọi ngắn gọn là vùng vmalloc)
  - Vùng kernel persistent mapping (gọi ngắn gọn là vùng kmap)
  - Vùng fixed virtual address mapping (gọi ngắn gọn là vùng fixmaps)

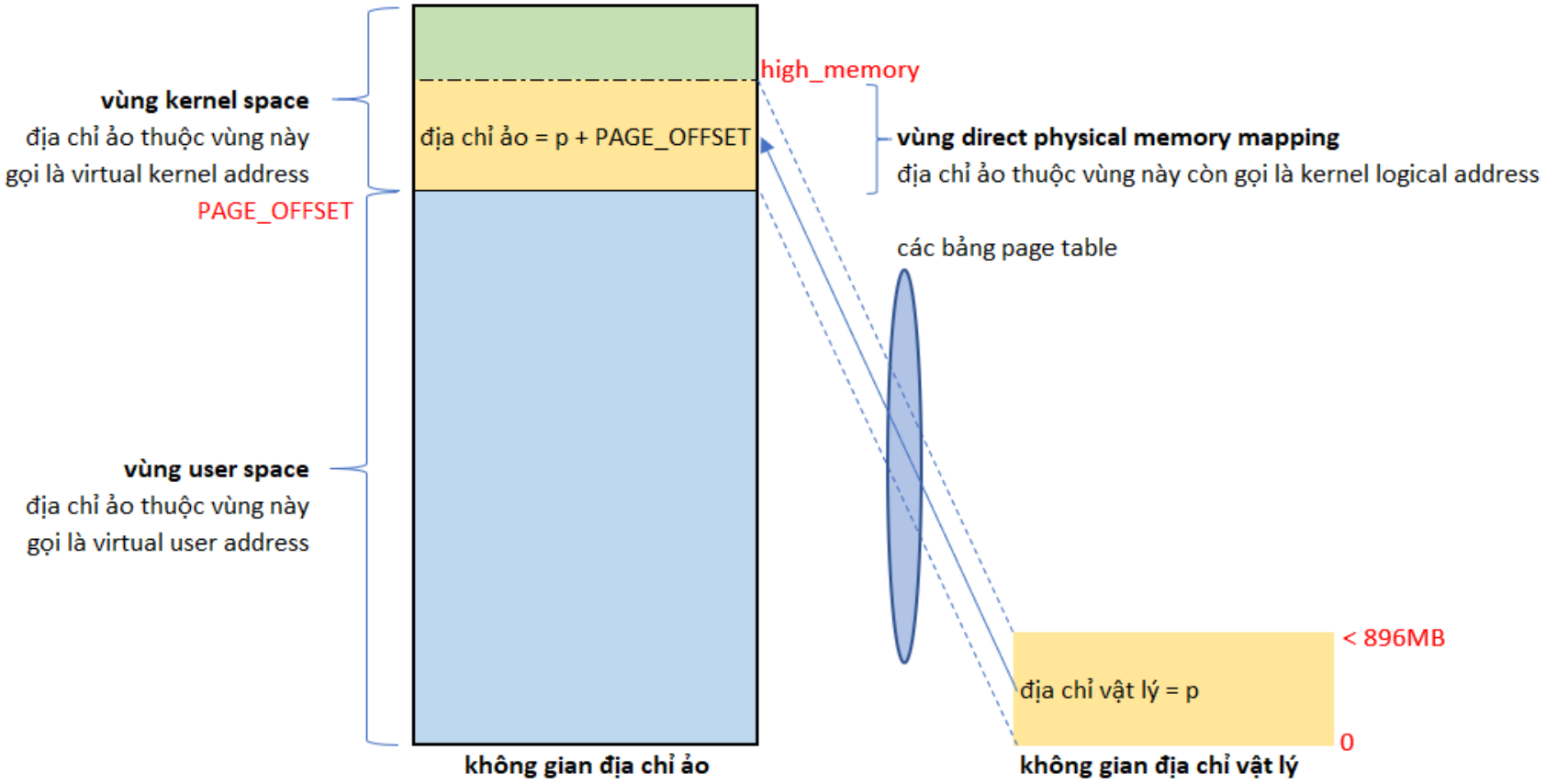


Vùng direct physical memory mapping

- Nếu kích thước của không gian địa chỉ vật lý không vượt quá 896MB, thì toàn bộ không gian địa chỉ vật lý được ánh xạ vào vùng địa chỉ ảo này. Việc ánh xạ được thiết lập ngay từ lúc khởi động, và được duy trì liên tục trong lúc hệ thống hoạt động
  - Vùng này kéo dài từ vị trí **PAGE_OFFSET** cho đến vị trí high_memory. Kích thước cực đại của vùng này là 896MB.
  - Bất kì địa chỉ ảo nào thuộc vùng này đều hơn địa chỉ vật lý tương ứng một lượng bằng **PAGE_OFFSET**
- cấp phát bộ nhớ vật lý liên tục trong một thời gian ngắn. Vì bộ nhớ vật lý đã được ánh xạ sẵn vào vùng này rồi, nên việc cấp phát diễn ra nhanh chóng

Vùng direct physical memory mapping

- Ánh xạ không gian địa chỉ vật lý vào vùng direct mapping trong hệ thống 32 bit



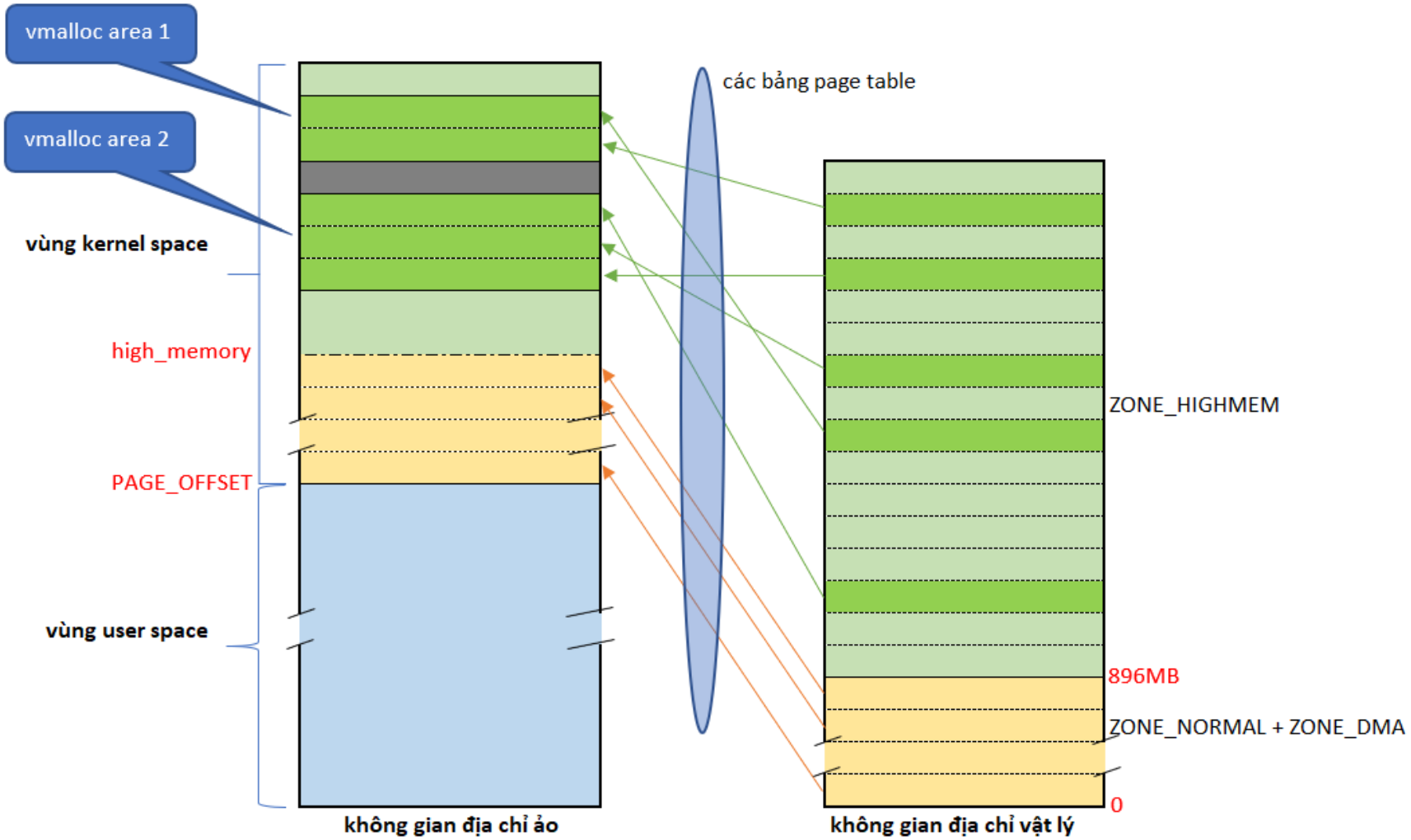
Vùng vmalloc address space

- Khi viết device driver, lập trình viên thường sử dụng hàm **kmalloc** để yêu cầu Linux kernel cấp phát một vùng nhớ liên tục trên ZONE_NORMAL.
  - Lý do là vì ZONE_NORMAL đã được ánh xạ sẵn vào kernel space, nên việc cấp phát diễn ra nhanh chóng
  - khi hệ thống đã hoạt động trong một thời gian dài, ZONE_NORMAL có thể bị phân mảnh
- có thể xem xét việc cấp phát bộ nhớ trên ZONE_HIGHMEM
  - để cấp phát được một vùng nhớ trên ZONE_HIGHMEM, thì vùng đó cần được ánh xạ vào kernel space
  - Linux cung cấp 2 kỹ thuật ánh xạ là **vmalloc** và **kmap** (được chia thành **permanent kmap** và **temporary kmap**)

Vùng vmalloc address space

- Kỹ thuật vmalloc giúp ánh xạ một số frame trên ZONE_HIGHMEM vào các page trên vùng vmalloc address space.
  - Vùng vmalloc address space nằm cách vùng direct physical memory mapping 8MB, bắt đầu từ VMALLOC_START cho tới vùng VMALLOC_END.
  - Vùng này gồm nhiều mảnh nhỏ, cách nhau 4KB. Mỗi mảnh được gọi là một **vmalloc area** hay **noncontiguous memory area**
- Nhằm phục vụ mục đích quản lý, Linux kernel sử dụng cấu trúc vm_struct để mô tả một vmalloc area.
  - Tất cả các cấu trúc **vm_struct** được chứa trong danh sách vmlist

Vùng vmalloc address space



Vùng kernel persistent mapping

- Trong cơ chế vmalloc, Linux kernel tạo ra một vmalloc area trên không gian địa chỉ ảo, rồi ánh xạ bất kì frame nào còn trống trên ZONE_HIGHMEM vào vmalloc area đó.
- Khác với vmalloc, cơ chế permanent kmap ánh xạ một frame cho trước vào một page trên vùng kernel persistent mapping.
  - frame này có thể nằm ở bất cứ đâu trên không gian địa chỉ vật lý, chứ không nhất thiết phải trên ZONE_HIGHMEM.
  - Thuật ngữ “permanent kmap” hay “persistent mapping” được sử dụng để chỉ các ánh xạ có thể tồn tại lâu dài

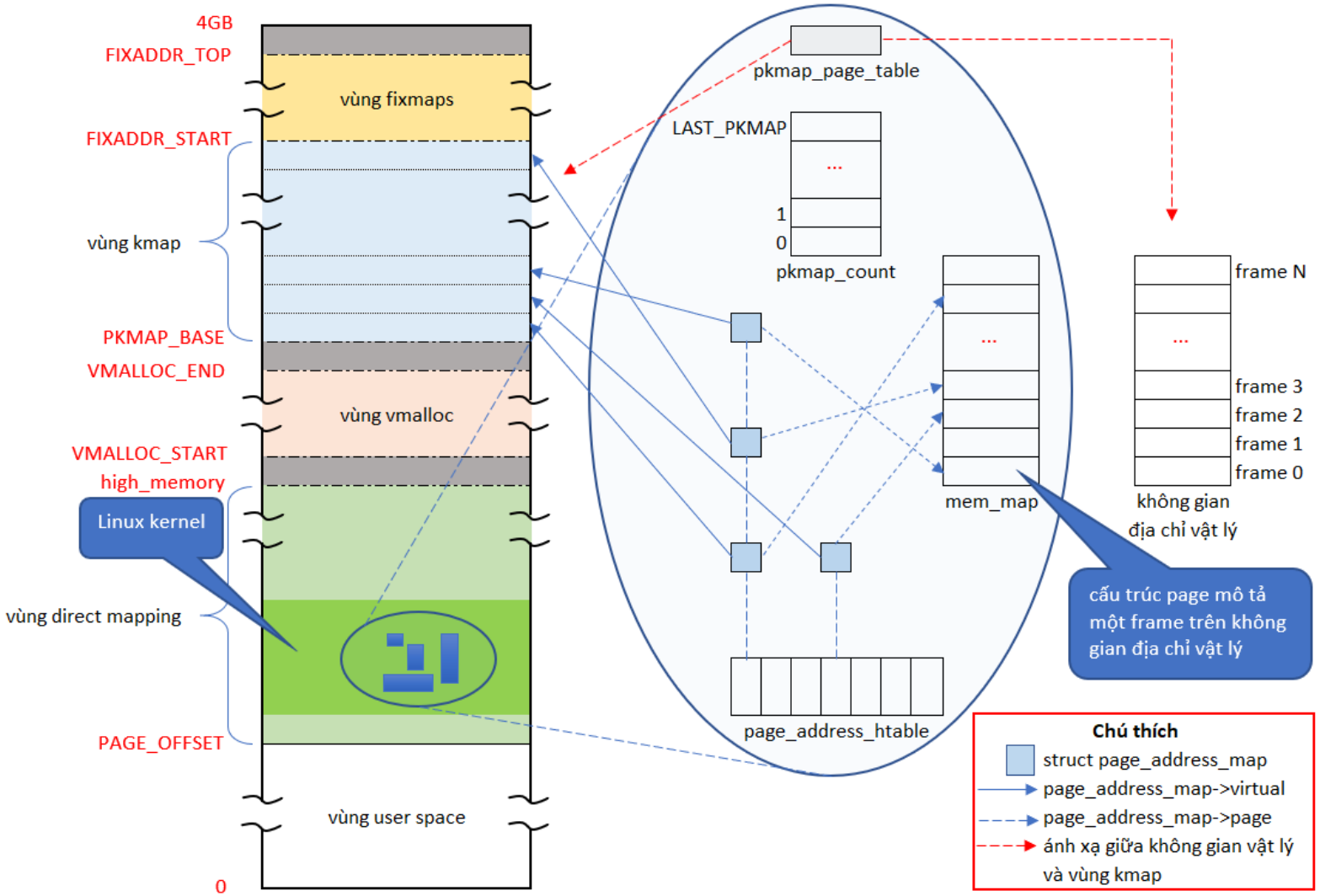
Vùng kernel persistent mapping

- Vùng kernel persistent mapping dài **LAST_PKMAP** page, bắt đầu từ địa chỉ PKMAP_BASE cho tới FIXADDR_START.
  - Trong Linux 32 bit, **LAST_PKMAP** có thể là 512 page hoặc 1024 page.
- Nhằm cho phép ánh xạ một frame vào một page trên vùng này, Linux cung cấp hàm kmap:
- Nếu frame này nằm trong ZONE_NORMAL hoặc ZONE_DMA, thì hàm này sẽ trả luôn về địa chỉ logical kernel address tương ứng với frame đó.
- Nếu frame này nằm trong ZONE_HIGHMEM, thì hàm này sẽ gọi hàm kmap_high thiết lập một ánh xạ mới.

Vùng kernel persistent mapping

- Để ánh xạ một frame vào một page trong vùng kernel persistent mapping, Linux kernel cần phải thiết lập một dòng trong một bảng Page Table.
  - Địa chỉ của bảng Page Table đó được lưu trong con trỏ pkmap_page_table, còn số dòng của bảng được xác định bởi macro **LAST_PKMAP**.
  - Linux cũng xây dựng mảng pkmap_count gồm **LAST_PKMAP** phần tử để kiểm soát xem dòng nào của bảng Page Table chưa được sử dụng, nếu đã được sử dụng rồi thì có bao nhiêu kernel thread đang sử dụng
  - Dựa vào đó, Linux sẽ biết được toàn bộ vùng này đã được ánh xạ hết hay chưa

Vùng kernel persistent mapping



Vùng fixed virtual address mapping

- Hệ thống có một số vùng nhớ cần được truy cập nhanh, ví dụ như vùng nhớ chứa bảng IDT mô tả ngắt.
- Nếu dùng các biến con trỏ thông thường để lưu địa chỉ ảo của các vùng nhớ nói trên, thì sẽ mất nhiều thời gian để tìm ra được địa chỉ vật lý của các vùng nhớ đó.
- Giải pháp ở đây là chọn ra một số địa chỉ ảo cố định, liên kết với địa chỉ vật lý của các vùng nhớ. Do đó, những địa chỉ ảo này được gọi là fixed virtual address.
- Tập hợp các fixed virtual address trên kernel space được gọi là vùng fixed virtual address mapping hay vùng fixmaps.
  - Dải địa chỉ này bắt đầu từ FIXADR_TOP và kết thúc ở FIXADDR_START. Kích thước của dải này là một số nguyên lần **PAGE_SIZE**.

Vùng fixed virtual address mapping

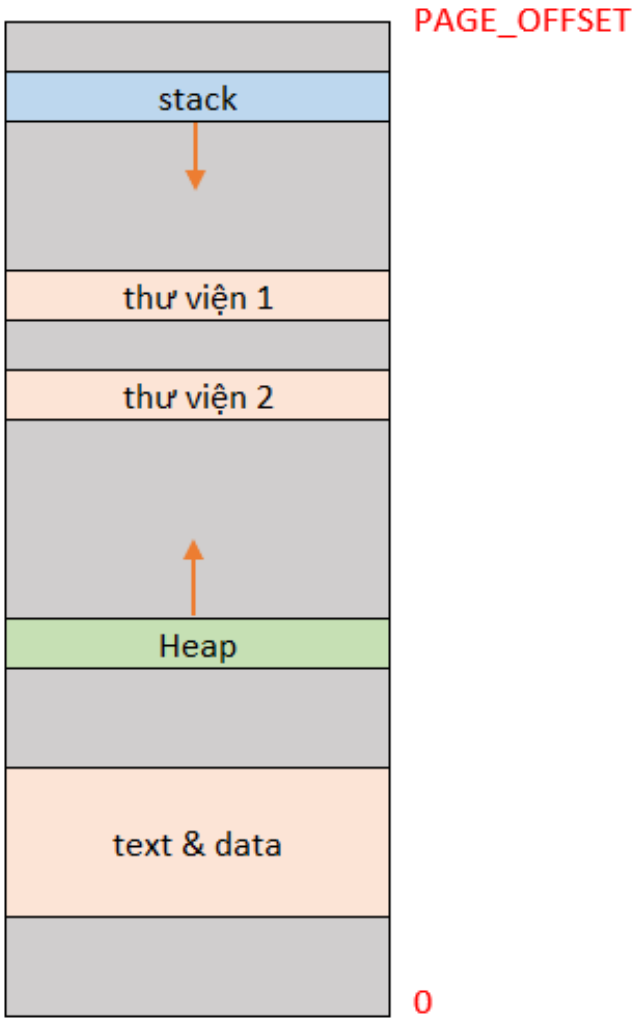
- Linux kernel sử dụng cấu trúc dữ liệu enum fixed_addresses để mô tả vùng fixmaps.
  - Mỗi số nguyên trong cấu trúc sẽ tương ứng với một fixed virtual address.
  - Để xác định fixed virtual address thứ n, Linux cung cấp hàm fix_to_virt.

```
/include/asm-generic/fixmap.h
#define __fix_to_virt(x)        (FIXADDR_TOP - ((x) << PAGE_SHIFT))

static __always_inline unsigned long fix_to_virt(const unsigned int idx)
{
        BUILD_BUG_ON(idx >= __end_of_fixed_addresses);
        return __fix_to_virt(idx);
}
```

User space

- Trong hệ thống sử dụng Linux 32 bit, user space của một tiến trình rộng mặc định 3GB. Tuy nhiên, không phải toàn bộ 3GB user space được sử dụng hết.
  - Trên thực tế, chỉ có một số dải địa chỉ được sử dụng. Mỗi dải này được gọi là một memory region hay virtual memory area
- Hình bên: *Quy hoạch vùng user space*



User space

- có thể đọc thông tin từ file */proc/<PID của tiến trình>/maps* để biết dải địa chỉ của các memory region trong vùng user space của một tiến trình

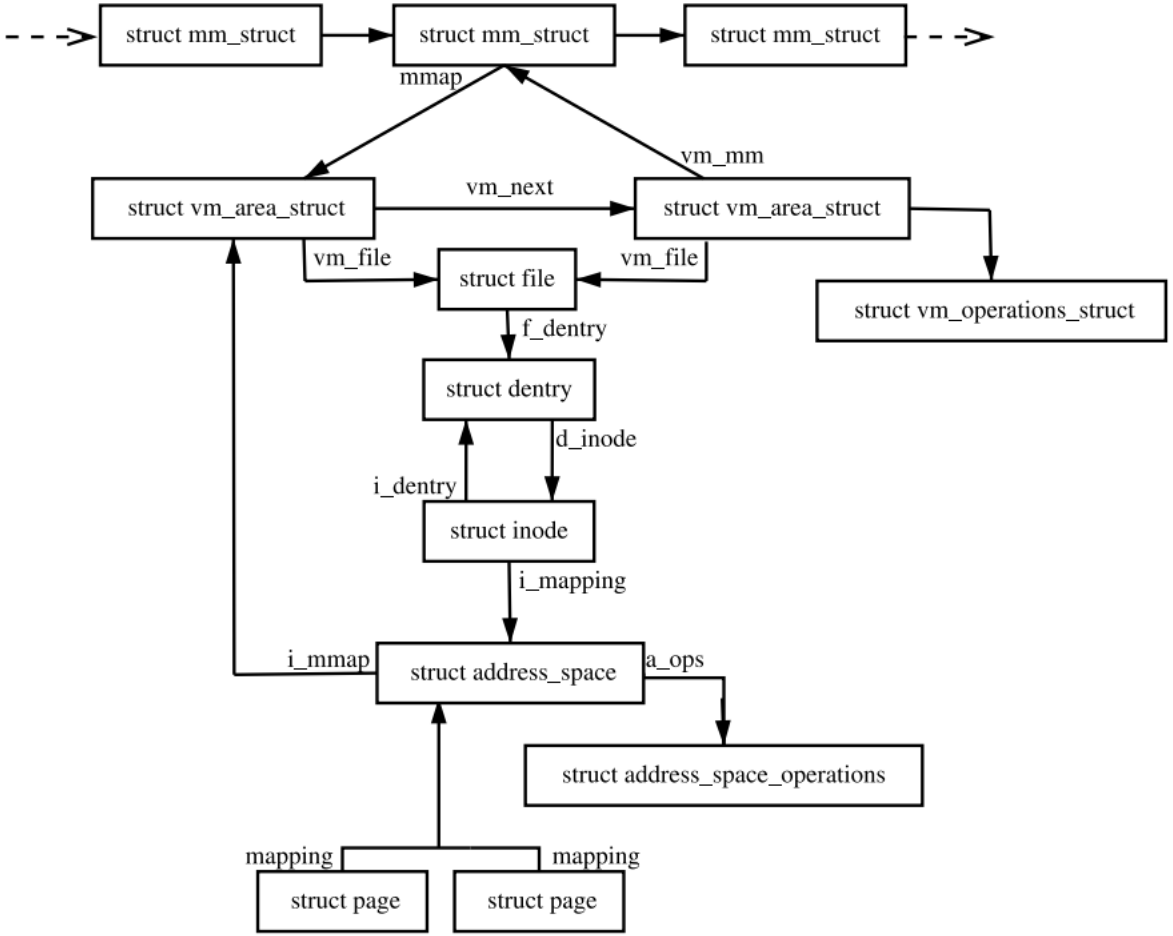
```
datnt@ubuntu32:~/helloWorld$ cat /proc/3205/maps
08048000-08049000 r-xp 00000000 08:01 656557     /home/datnt/helloWorld/hello
08049000-0804a000 r--p 00000000 08:01 656557     /home/datnt/helloWorld/hello
0804a000-0804b000 rw-p 00001000 08:01 656557     /home/datnt/helloWorld/hello
09319000-0933a000 rw-p 00000000 00:00 0          [heap]
b7d31000-b7d32000 rw-p 00000000 00:00 0
b7d32000-b7ee2000 r-xp 00000000 08:01 915828     /lib/i386-linux-gnu/libc-2.23.s
o
b7ee2000-b7ee4000 r--p 001af000 08:01 915828     /lib/i386-linux-gnu/libc-2.23.s
o
b7ee4000-b7ee5000 rw-p 001b1000 08:01 915828     /lib/i386-linux-gnu/libc-2.23.s
o
b7ee5000-b7ee8000 rw-p 00000000 00:00 0
b7efe000-b7eff000 rw-p 00000000 00:00 0
b7eff000-b7f02000 r--p 00000000 00:00 0          [vvar]
b7f02000-b7f04000 r-xp 00000000 00:00 0          [vdso]
b7f04000-b7f27000 r-xp 00000000 08:01 915800     /lib/i386-linux-gnu/ld-2.23.so
b7f27000-b7f28000 r--p 00022000 08:01 915800     /lib/i386-linux-gnu/ld-2.23.so
b7f28000-b7f29000 rw-p 00023000 08:01 915800     /lib/i386-linux-gnu/ld-2.23.so
bfe6e000-bfe8f000 rw-p 00000000 00:00 0          [stack]
datnt@ubuntu32:~/helloWorld$
```

User space

- Mỗi một tiến trình được Linux kernel mô tả bằng một cấu trúc task_struct.
  - Trường mm của cấu trúc **task_struct** trỏ tới cấu trúc mm_struct mô tả vùng user space của tiến trình đó
  - struct mm_struct { struct vm_area_struct * mmap; //danh sách các memory region pgd_t * pgd; //địa chỉ bảng PGD của tiến trình … }
    - Trường mmap của cấu trúc **mm_struct** là một danh sách các memory region của tiến trình.
    - Mỗi memory region được Linux kernel mô tả bằng một cấu trúc vm_area_struct

User space

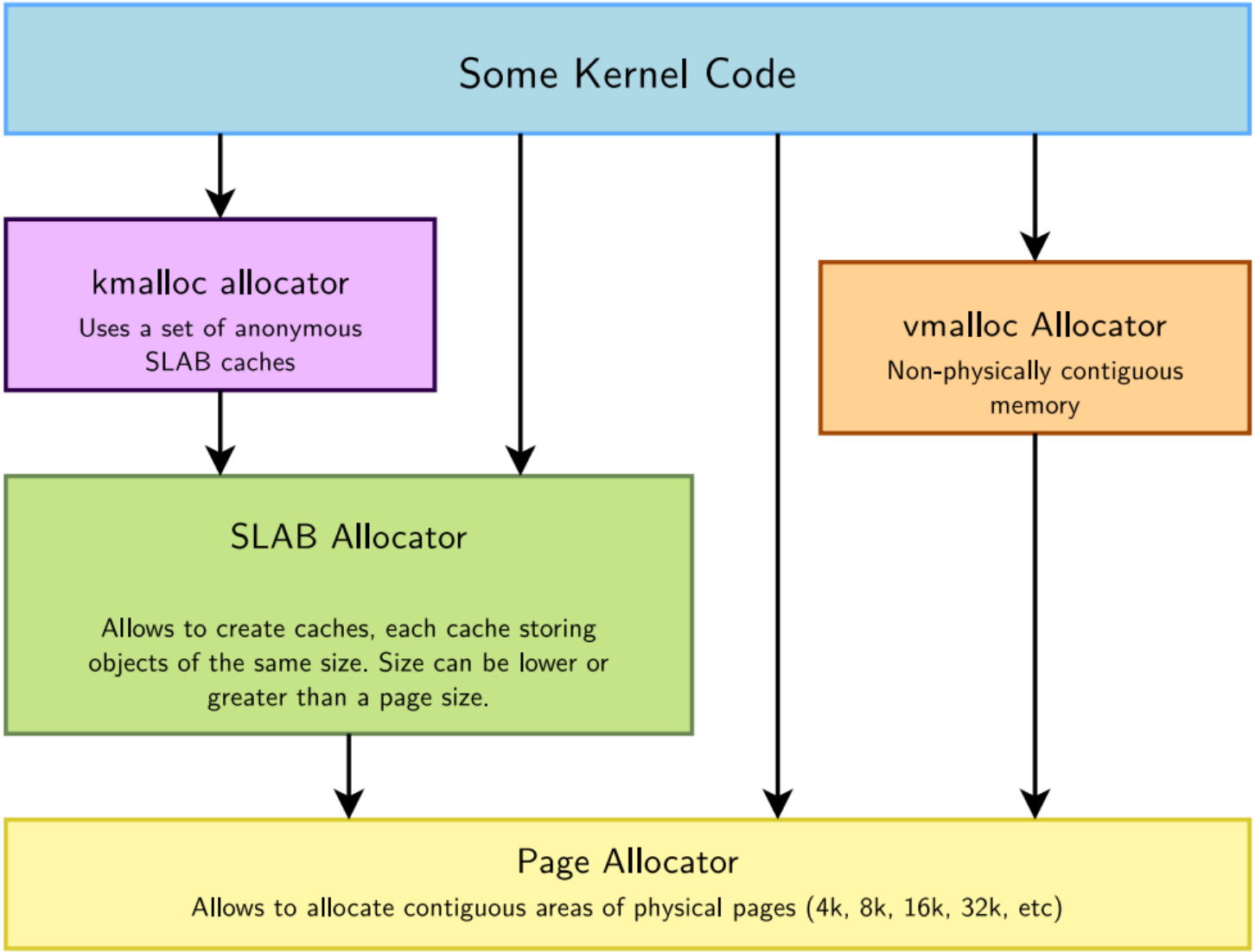
- mối quan hệ giữa cấu trúc **mm_struct** và các **vm_area_struct** trong mỗi tiến trình



Cấp phát theo byte

- Khi lập trình ứng dụng (application programming), chúng ta thường sử dụng cặp hàm **malloc/free** để cấp phát/giải phóng bộ nhớ cho các tiến trình trên user space.
- thường sử dụng cặp hàm **kmalloc/kfree** để cấp phát/giải phóng bộ nhớ cho các module dưới kernel space.
  - Phương pháp cấp phát này có tên là kmalloc allocator.
- Ngoài kmalloc allocator, ta còn có các phương pháp cấp phát bộ nhớ khác: vmalloc allocator, slab allocator, và page allocator

Cấp phát theo byte



Cách sử dụng hàm kmalloc()

```
/*
 * hàm kmalloc
 * chức năng: cấp phát một vùng nhớ liên tục trên RAM.
 *
 * tham số đầu vào:
 *     size  [I]: kích thước vùng nhớ muốn được cấp phát (tính theo byte).
 *     flags [I]: cờ điều khiển quá trình cấp phát.
 *
 * giá trị trả về:
 *    Nếu thành công, hàm này sẽ trả về kernel logical address của vùng nhớ đã
được cấp phát.
 *    Nếu thất bại, hàm này sẽ trả về NULL.
 *
 * chú ý:
 *     1. Trong quá trình lập trình, ta nên kiểm tra giá trị trả về và xử lý
nếu cần.
 *     2. Hàm này chỉ nên dùng để cấp phát các vùng nhớ có kích thước nhỏ hơn
4MB.
 *     Nếu cần vùng nhớ có kích thước lớn hơn, ta nên sử dụng các hàm cấp phát
bộ nhớ
 *     theo page (xem bài học sau).
 */
void *kmalloc(size_t size, gfp_t flags);

/*
 * hàm kzalloc
 * chức năng: giống như hàm kmalloc. Ngoài ra, hàm này còn thực hiện
 *            xóa các bit của vùng nhớ sau khi cấp phát về 0.
 */
Hàm void *kzalloc(size_t size, gfp_t flags);

/*
 * hàm kfree
 * chức năng: giải phóng bộ nhớ có địa chỉ @ptr.
 *            đã được cấp phát trước đó bởi kmalloc hoặc kzalloc.
 */
void kfree(void *ptr);
```

Cách sử dụng hàm vmalloc()
cấp phát bộ nhớ trên ZONE_HIGHMEM

```
/*
 * hàm vmalloc
 * chức năng: cấp phát một vùng nhớ liên tục trên kernel space.
 *
 * tham số đầu vào:
 *     size  [I]: kích thước vùng nhớ muốn được cấp phát (tính theo byte).
 *
 * giá trị trả về:
 *     Nếu thành công, hàm này sẽ trả về địa chỉ ảo của vùng nhớ đã được cấp
phát.
 *     Nếu thất bại, hàm này sẽ trả về NULL.
 *
 * chú ý:
 *     1. Trong quá trình lập trình, ta nên kiểm tra giá trị trả về và
 *     xử lý nếu cần.
 *     2. Các frame trên ZONE_HIGHMEM không nhất thiết phải nằm cạnh nhau.
 *     Do đó, vùng nhớ được cấp phát bởi vmalloc liên tục trên không gian
 *     địa chỉ ảo, nhưng chưa chắc liên tục trên không gian địa chỉ vật lý.
 *     3. Kernel thread gọi hàm này có thể bị đưa vào trạng thái ngủ,
 *     do đó không sử dụng hàm này trong interrupt handler.
 *     4. Mặc dù ta yêu cầu cấp phát size byte, nhưng số lượng byte thực tế
 *     được cấp phát có thể lớn hơn, và là bội số của PAGE_SIZE.
 */
void *vmalloc(unsigned long size);

/*
 * hàm vfree
 * chức năng: giải phóng bộ nhớ có địa chỉ @ptr
 *            đã được cấp phát trước đó bởi vmalloc.
 */
void vfree(const void *ptr);
```

Cấp phát theo page-frame

- Trong quá trình viết device driver, ta có thể dùng hàm alloc_pages để yêu cầu Linux kernel cấp phát cho một vài page frame trên RAM.
  - khi không sử dụng nữa, ta dùng hàm __free_pages để giải phóng các page frame đã được cấp phát
- sử dụng hàm page_address để lấy ra được địa chỉ kernel virtual address từ cấu trúc page
- có thể sử dụng hàm __get_free_pages để lấy được luôn địa chỉ kernel virtual address của page frame.
  - khi không sử dụng nữa, ta dùng hàm free_pages để giải phóng các frame đã cấp phát

Cấp phát theo page-frame

```
/*
 * hàm alloc_pages
 * chức năng: cấp phát một vài page frame trên RAM.
 *
 * tham số đầu vào:
 *     gfp_mask [I]: cờ điều khiển quá trình cấp phát, tương tự như
 *                   tham số flags trong hàm kmalloc.
 *     order    [I]: (2^order) page frame sẽ được cấp phát.
 *
 * giá trị trả về:
 *     Nếu thành công, hàm này trả về địa chỉ của cấu trúc page mô tả frame
 *     đầu tiên trong chuỗi các frame được cấp phát.
 *     Nếu thất bại, hàm này sẽ trả về NULL.
 *
 * chú ý:
 *     1. Trong quá trình lập trình, ta nên kiểm tra giá trị trả về và xử lý nếu
cần.
 *     2. Nếu muốn cấp phát một page frame, ta có 2 cách:
 *     - cách 1: sử dụng hàm alloc_pages với tham số @order = 0
 *     - cách 2: sử dụng hàm alloc_page(gfp_mask)
 */
struct page *alloc_pages(gfp_t gfp_mask, unsigned int order);
```

Cấp phát theo page-frame

```
/*
 * hàm __free_pages
 * chức năng: giải phóng 2^order page frame, bắt đầu từ page frame
 *            tương ứng với cấu trúc page có địa chỉ @page.
 *
 * tham số đầu vào:
 *     page [I]: địa chỉ của cấu trúc page mô tả frame
 *               sẽ bị giải phóng bộ nhớ.
 *     order[I]: (2^order) page frame sẽ bị giải phóng.
 *
 * giá trị trả về: không có
 *
 * chú ý:
 *     Nếu muốn giải phóng một page frame, ta có 2 cách:
 *     - cách 1: sử dụng hàm __free_pages với tham số @order = 0
 *     - cách 2: sử dụng hàm __free_page(page)
 */
void __free_pages (struct page *page, unsigned int order);
```

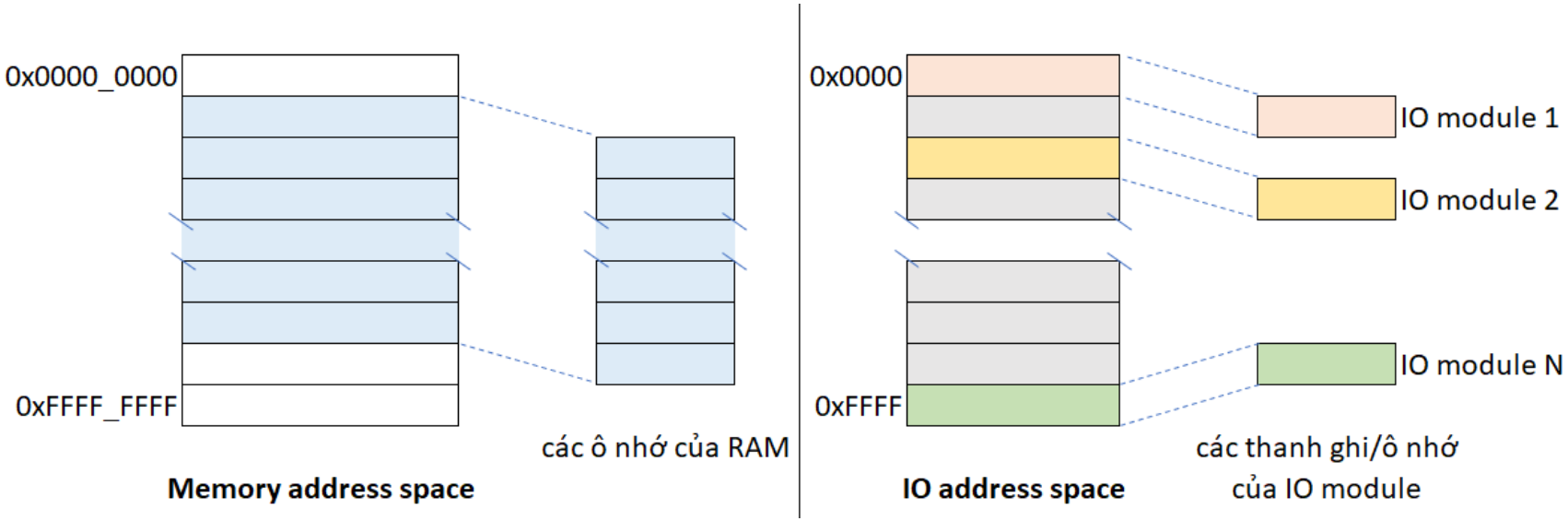
Truy cập dữ lieu trên mô-đun IO

- hệ thống máy tính gồm 4 thành phần chính: CPU, RAM, các IO module, và hệ thống bus.
  - CPU lần lượt lấy các lệnh trên RAM vào, giải mã, rồi dựa vào đó mà xử lý dữ liệu.
  - Dữ liệu có thể nằm trên RAM hoặc trên các IO module.
  - Để CPU truy cập được dữ liệu trên IO module, thì các thanh ghi/ô nhớ của IO module cũng cần phải có địa chỉ.
  - Có 2 phương pháp đánh địa chỉ cho các thanh ghi/ô nhớ của IO module là **PIO** và **MMIO**

Phương pháp PIO

- Trong phương pháp PIO (**P**ort **I**nput **O**utput), dải số dùng để đánh địa chỉ cho các thanh ghi/ô nhớ của IO module và dải số dùng để đánh địa chỉ cho các ô nhớ của RAM thuộc 2 không gian khác nhau: **IO address space** và **memory address space**.
- Khi cần truy cập dữ liệu trên IO module, CPU sẽ phát địa chỉ thuộc IO address space lên một bus riêng.
  - Lệnh truy cập dữ liệu trên IO module cũng khác với lệnh truy cập dữ liệu trên RAM.
  - Hệ thống dùng bộ xử lý x86 sử dụng phương pháp này

Phương pháp PIO



- Mỗi địa chỉ trong IO address space còn được gọi là một **port**. Do vậy, IO address space còn được gọi là **port address space**, hay ngắn gọn là **IO ports**. Không gian IO ports rộng 16 bit, và được chia làm nhiều vùng, mỗi vùng được gọi là một port region
- Để biết được IO ports gồm những vùng nào, ta đọc file */proc/ioports*

PIO: Truy cập từ kernel space

- Linux cung cấp một số hàm truy cập dữ liệu trên IO module trong trường hợp hệ thống sử dụng phương pháp PIO.
  - Trước hết, cần gọi hàm request_region để yêu cầu kernel cho phép ta truy cập vào một port region.
  - Sau khi được kernel cho phép, sẽ sử dụng các hàm inb, inw, inl để đọc dữ liệu từ một port trong port region đó, hoặc sử dụng hàm outb, outw, outl để ghi dữ liệu ra một port.
  - Khi không cần dùng port region đó nữa, gọi hàm release_region để thông báo cho kernel biết

PIO: Truy cập từ kernel space

```
/*
 * hàm: request_region
 * chức năng: yêu cầu Linux kernel cho phép ta truy cập vào một port region.
 * tham số đầu vào:
 *    start [I]: là địa chỉ vật lý của port region.
 *    len   [I]: là kích thước của port region (hay số lượng port).
 *    *name [I]: là tên của port region, sẽ xuất hiện trong file /proc/ioports.
 *
 * giá trị trả về:
 *    Trả về một con trỏ kiểu resource nếu thành công.
 *    Trả về NULL nếu đã có kernel module khác đang sử dụng port region này rồi.
 */
struct resource *request_region(unsigned long start,
                                unsigned long len, char *name);

/*
 * hàm: release_region
 * chức năng: thông báo ngừng sử dụng một port region cho Linux kernel biết.
 * tham số đầu vào:
 *    start [I]: là địa chỉ vật lý của port region.
 *    len   [I]: là kích thước của port region (hay số lượng port).
 *
 * giá trị trả về: Không có
 */
void release_region(unsigned long start, unsigned long len);
```

PIO: Truy cập từ kernel space

```
/*
 * hàm: inb, inw, inl
 * chức năng: đọc dữ liệu có kích thước 8/16/32 bit từ một port.
 * tham số đầu vào:
 *    addr [I]: dữ liệu được đọc từ port có địa chỉ vật lý này.
 *
 * giá trị trả về: dữ liệu trong port
 *
 * chú ý:
 *    Giá trị @addr nên nằm trong dải từ @start đến @start + @len.
 */
u8 inb(unsigned long addr);
u16 inw(unsigned long addr);
u32 inl(unsigned long addr);

/*
 * hàm: outb, outw, outl
 * chức năng: ghi dữ liệu có kích thước 8/16/32 bit vào một port.
 * tham số đầu vào:
 *    b    [I]: dữ liệu cần ghi vào port
 *    addr [I]: @b được ghi vào port có địa chỉ vật lý này.
 *
 * giá trị trả về: không có giá trị trả về
 *
 * chú ý:
 *    Giá trị @addr nên nằm trong dải từ @start đến @start + @len.
 */
void outb(u8 b, unsigned long addr);
void outw(u16 b, unsigned long addr);
void outl(u32 b, unsigned long addr);
```

PIO: Truy cập từ user space

- Các tiến trình trên user space cũng có thể truy cập trực tiếp vào các port. Để làm được điều này, ta có 2 cách:
  - Sử dụng system call ioperm hoặc iopl.
  - Truy cập thông qua */dev/port*.

PIO: Truy cập từ user space

```c
/*
 * chương trình 1:
 * đọc số giây của RTC sử dụng ioperm hoặc iopl
 */
#include <stdio.h>
#include <stdlib.h>
#include <sys/io.h>

void dump_port(unsigned char addr_port, unsigned char data_port,
               unsigned short offset, unsigned short length)
{
    unsigned char i, *data;
    data = (unsigned char *)malloc(length);

    /* Ghi giá trị offset vào index port và đọc dữ liệu từ data port */
    for(i = offset; i < offset + length; i++) {
        outb(i, addr_port );
        data[i - offset] = inb(data_port);
    }

    for(i = 0; i < length; i++)
    printf("%02X ", data[i]);
    free(data);
}

int main(int argc, char *argv[])
{
    /* sử dụng ioperm hoặc iopl */
    iopl(3);
    //ioperm(0x70, 2, 1);

    dump_port(0x70, 0x71, 0x0, 1);
}
```

Phương pháp MMIO

- Trong phương pháp MMIO (**M**emory **M**apped **I**nput **O**utput), dải số dùng để đánh địa chỉ cho các thanh ghi/ô nhớ của IO module và dải số dùng để đánh địa chỉ cho các ô nhớ của RAM thuộc cùng một không gian địa chỉ.
  - hệ thống không cần phải thiết kế một bus riêng, và CPU cũng không cần một tập lệnh riêng để truy cập dữ liệu trên IO module.
  - Việc truy cập dữ liệu trên IO module tượng tự như truy cập dữ liệu trên RAM. Các hệ thống dùng bộ xử lý ARM sử dụng phương pháp này.
  - Hệ thống x86, bên cạnh dùng phương pháp PIO, cũng sử dụng phương pháp MMIO.
- Trong phương pháp MMIO, không gian địa chỉ vật lý được chia thành các vùng, mỗi vùng được gọi là một IO memory region.
  - Mỗi IO memory region là một dải địa chỉ của các ô nhớ trên RAM, hoặc là các thanh ghi/ô nhớ trên IO module.
  - Để biết được không gian địa chỉ vật lý của cả hệ thống đã được quy hoạch như nào, đọc file */proc/iomem*

MMIO: Truy cập từ kernel space

- Linux cung cấp một số hàm giúp ta truy cập dữ liệu trên IO module trong trường hợp hệ thống sử dụng phương pháp MMIO.
  - Trước hết, cần gọi hàm request_mem_region để yêu cầu kernel cho phép ta truy cập vào dải địa chỉ vật lý của IO module.
  - Sau đó, gọi hàm ioremap để ánh xạ dải địa chỉ vật lý của IO module vào kernel space.
  - Sau khi đã ánh xạ xong, sẽ sử dụng các hàm ioread8, ioread16, ioread32 để đọc dữ liệu, hoặc sử dụng hàm iowrite8, iowrite16, iowrite32 để ghi dữ liệu ra một thanh ghi/ô nhớ nào đó của IO module.
  - Khi không cần dùng nữa, gọi hàm iounmap và release_mem_region để thông báo cho kernel biết

MMIO: Truy cập từ kernel space

```
/*
 * hàm: request_mem_region
 * chức năng: yêu cầu Linux kernel cho phép ta truy cập vào một IO memory
region
 *            chứa dải địa chỉ vật lý của IO module.
 * tham số đầu vào:
 *    start [I]: là địa chỉ vật lý của IO memory region.
 *    len   [I]: là kích thước của IO memory region.
 *    *name [I]: là tên của IO memory region, sẽ xuất hiện trong file
/proc/iomem.
 *
 * giá trị trả về:
 *    Trả về một con trỏ kiểu resource nếu thành công.
 *    Trả về NULL nếu đã có kernel module khác đang sử dụng IO memory region
này rồi.
 */
struct resource* request_mem_region(unsigned long start, unsigned long len,
char *name);

/*
 * hàm: release_mem_region
 * chức năng: thông báo cho Linux kernel biết rằng ta ngừng sử dụng IO memory
region
 *            chứa dải địa chỉ vật lý của IO module.
 * tham số đầu vào:
 *    start [I]: là địa chỉ vật lý của IO memory region.
 *    len   [I]: là kích thước của IO memory region.
 *
 * giá trị trả về: Không có
 */
void release_mem_region(unsigned long start, unsigned long len);
```

MMIO: Truy cập từ kernel space

```
/*
 * hàm: ioremap
 * chức năng: ánh xạ IO memory region vào kernel space.
 * tham số đầu vào:
 *    phys_add [I]: là địa chỉ vật lý của IO memory region.
 *    size     [I]: là kích thước của IO memory region.
 *
 * giá trị trả về:
 *    Trả về địa chỉ ảo của IO memory region nếu quá trình ánh xạ thành công
 *    Trả về NULL nếu quá trình ánh xạ thất bại
 * chú ý: gọi sau hàm request_mem_region
 */
void __iomem *ioremap(unsigned long phys_add, unsigned long size);

/*
 * hàm: iounmap
 * chức năng: ngừng ánh xạ IO memory region vào kernel space.
 * tham số đầu vào:
 *    addr [I]: là địa chỉ ảo của IO memory region trong kernel space.
 *
 * giá trị trả về: Không có
 * chú ý: gọi trước hàm release_mem_region
 */
void iounmap(void __iomem *addr);

/*
 * hàm: ioread8, ioread16, ioread32
 * chức năng: đọc dữ liệu có kích thước 8/16/32 bit từ một thanh ghi/ô nhớ của
IO module.
 * tham số đầu vào:
 *    addr [I]: địa chỉ ảo của thanh ghi/ô nhớ cần đọc dữ liệu.
 *
 * giá trị trả về: dữ liệu trong thanh ghi/ô nhớ của IO module
 *
 * chú ý:
 *    Giá trị @addr nên nằm trong dải từ @start đến @start + @len.
 */
unsigned int ioread8(void __iomem *addr);
unsigned int ioread16(void __iomem *addr);
unsigned int ioread32(void __iomem *addr);
```

MMIO: Truy cập từ kernel space

```
/*
 * hàm: iowrite8, iowrite16, iowrite32
 * chức năng: ghi dữ liệu có kích thước 8/16/32 bit vào một thanh ghi/ô nhớ của
IO module.
 * tham số đầu vào:
 *    value[I]: dữ liệu cần ghi vào thanh ghi/ô nhớ
 *    addr [I]: địa chỉ ảo của thanh ghi/ô nhớ cần ghi dữ liệu.
 *
 * giá trị trả về: không có giá trị trả về
 *
 * chú ý:
 *    Giá trị @addr nên nằm trong dải từ @start đến @start + @len.
 */
void iowrite8(u8 value, void __iomem *addr);
void iowrite16(u16 value, void __iomem *addr);
void iowrite32(u32 value, void __iomem *addr);
```

MMIO: **Truy cập từ user space**

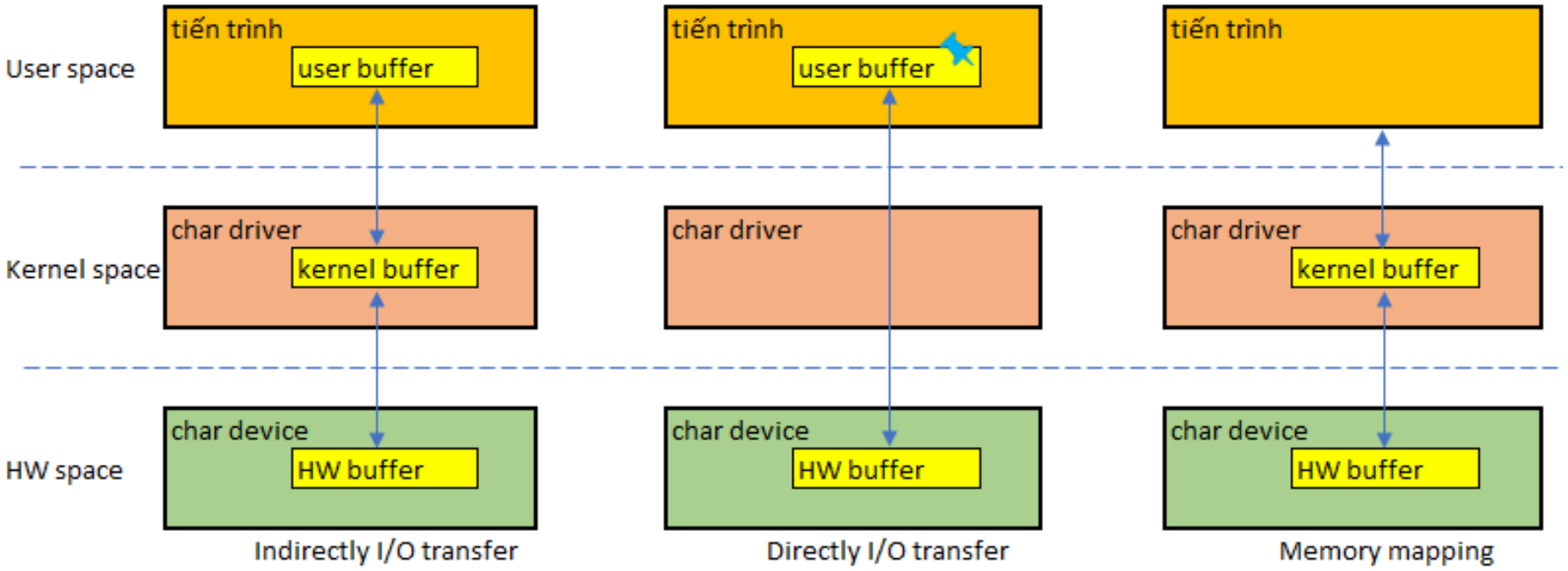
- Các tiến trình trên user space cũng có thể truy cập trực tiếp vào các IO memory region.
- Để làm được điều này, ta sử dụng kỹ thuật memory mapping

Ánh xạ bộ nhớ

- Char driver sử dụng
  các hàm **copy_to_user** hoặc **copy_from_user** để sao chép dữ liệu giữa kernel buffer và user buffer.
- muốn giảm số lần sao chép dữ liệu giữa tiến trình và char device để tăng tốc độ trao đổi dữ liệu. Linux kernel cung cấp 2 phương pháp:
  - Phương pháp **directly I/O transfer**: char driver sẽ thực hiện sao chép dữ liệu từ HW buffer vào thẳng user buffer (hoặc ngược lại).
    - ✓Trong quá trình trao đổi dữ liệu với char device, các user buffer không được phép đưa ra ngoài ổ cứng.
  - Phương pháp **memory mapping**: tiến trình sẽ được cấp quyền truy cập thẳng vào kernel buffer

Ánh xạ bộ nhớ

- *Các phương pháp trao đổi dữ liệu giữa tiến trình và char driver*



Ánh xạ bộ nhớ

- Phương pháp memory mapping cho phép một tiến trình truy cập trực tiếp vào kernel buffer.
  - Kernel buffer này có thể liên kết với một vùng nhớ trên RAM, hoặc một phần của file (ví dụ regular file, device file).
  - Vì được phép truy cập trực tiếp, nên tiến trình đọc/ghi dữ liệu thẳng vào kernel buffer mà không cần phải thông qua system call **read** hay **write**, từ đó loại bỏ việc sao chép dữ liệu giữa user buffer và kernel buffer.
  - Điều này giúp cải thiện tốc độ trao đổi dữ liệu so với phương pháp Indirectly IO transfer, nhất là trong trường hợp khối lượng dữ liệu lớn, hoặc phải truy cập nhiều lần.
  - Tuy vậy, nhược điểm của kỹ thuật này là thời gian thiết lập và giải phóng lâu hơn

Ánh xạ bộ nhớ

- Để triển khai phương pháp memory mapping, ta cần thực hiện cả ở tiến trình trên user space và ở driver dưới kernel space
- Trên user space, tiến trình cần gọi system call mmap để yêu cầu Linux tạo ra một vùng virtual memory area (VMA) trong user space của tiến trình. Vùng VMA này được gọi là vùng **memory mapping** hay **mapped area**. Vị trí và kích thước của vùng này là bội của PAGE_SIZE. Ta phân loại các vùng memory mapping như sau:
  - ***Dựa vào loại kernel buffer*** tương ứng với vùng memory mapping, ta có 2 kiểu:
    - ✓**anonymous mapping**: kernel buffer liên kết với một vùng nhớ vật lý trên RAM.
    - ✓**file mapping** (hay **memory mapped file**): kernel buffer liên kết với một phần của file, hoặc một vùng nhớ vật lý trên thiết bị.

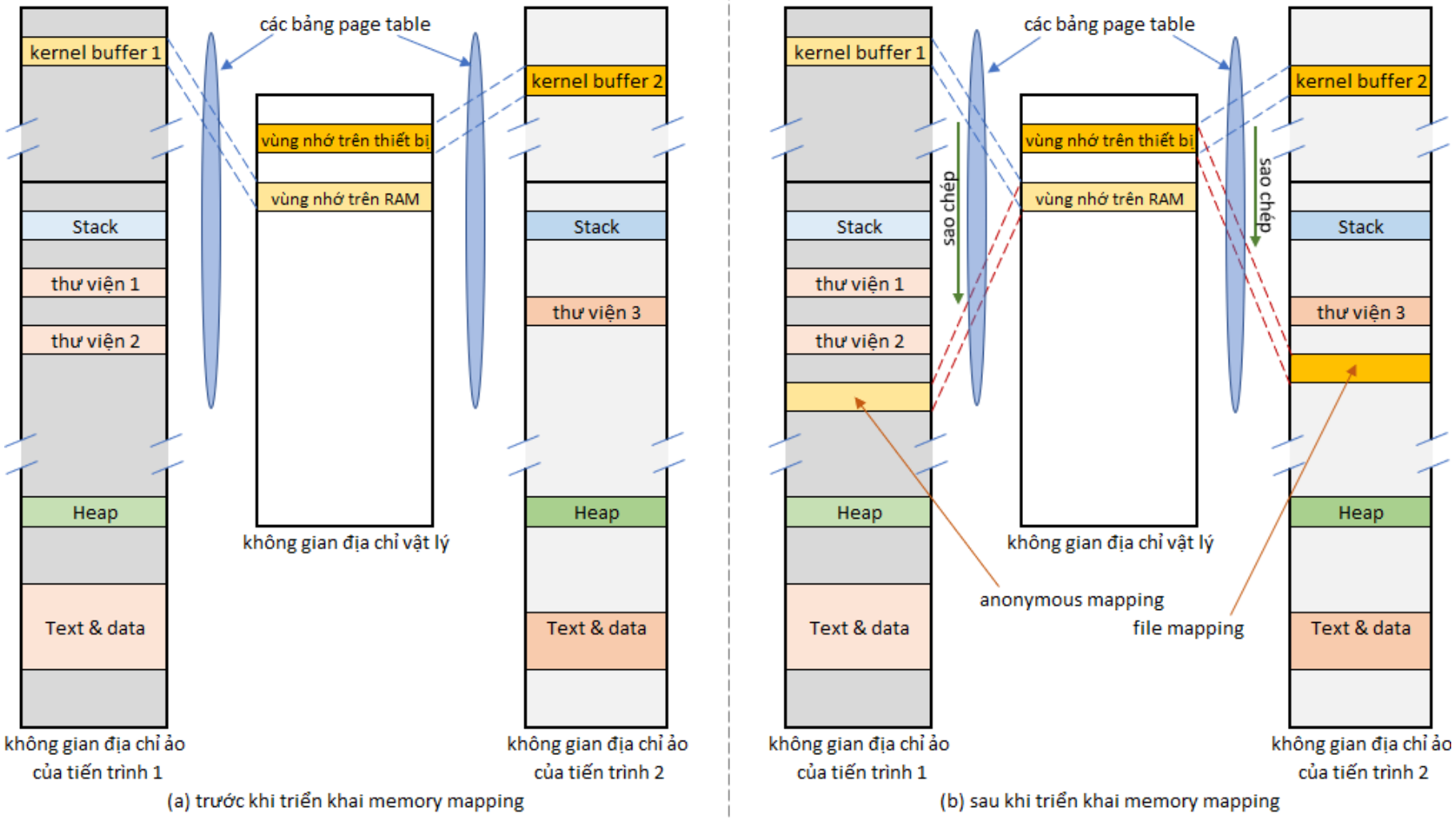
Ánh xạ bộ nhớ

- ***Dựa vào khả năng chia sẻ dữ liệu*** giữa các tiến trình trên vùng memory mapping, có 2 kiểu:
  - **shared mapping**: khi một tiến trình ghi dữ liệu vào vùng nhớ này thì các tiến trình khác cũng biết được.
    - ✓Nếu vùng này cũng là file mapping, thì sự thay đổi dữ liệu sẽ được cập nhật trên file hoặc thiết bị.
  - **private mapping**: khi một tiến trình ghi dữ liệu vào vùng nhớ này thì các tiến trình khác không biết được.
    - ✓Nếu vùng này cũng là file mapping, thì sự thay đổi dữ liệu sẽ không được cập nhật trên file hoặc thiết bị.
    - ✓Mọi thay đổi sẽ bị mất khi tiến trình kết thúc

Ánh xạ bộ nhớ

- Dưới kernel space, driver cần triển khai entry point **mmap** để ánh xạ vùng nhớ vật lý vào vùng mapped area của tiến trình.
  - Khái niệm "ánh xạ" ở đây được hiểu là thiết lập bảng page table của tiến trình để tạo liên kết giữa vùng nhớ vật lý và vùng mapped area.
  - Để thực hiện việc này, một số dòng của bảng kernel page table được sao chép cho user page table

Cơ chế memory mapping: tạo một vùng mapped area rồi ánh xạ vùng nhớ vật lý vào mapped area đó



Triển khai ánh xạ bộ nhớ

- Trong user space
- Trong kernel space

Triển khai memory mapping trên user space

- Để tạo ra một vùng memory mapping trên user space của tiến trình, gọi system call **mmap** để tạo ra một trong bốn loại memory mapping sau:
  - Private file mapping: ta thường tạo ra loại này khi cần ánh xạ share library file vào trong vùng user space của tiến trình.
  - Private anonymous mapping: ta thường tạo ra loại này khi cần cấp phát một lượng lớn bộ nhớ, và khởi tạo vùng này bằng 0.
  - Shared file mapping: ta thường tạo ra loại này khi cần truy cập trực tiếp vào file. Ngoài ra, ta cũng có thể sử dụng loại này nếu muốn thực hiện truyền thông liên tiến trình.
  - Shared anonymous mapping: ta thường tạo ra loại này khi cần trao đổi thông tin giữa tiến trình cha và tiến trình con.
- Tuy nhiên, trong các ứng dụng điều khiển thiết bị, ta thường sử dụng loại shared file mapping

Triển khai memory mapping trên user space

```
/*
 * System call mmap
 * chức năng: tạo ra một vùng memory mapping trên user space của tiến trình.
 *
 * tham số đầu vào:
 *     addr  [I]: địa chỉ mong muốn của vùng memory mapping trên user space.
 *                - Nếu tham số này bằng NULL, thì kernel sẽ tự chọn ra một
 *                địa chỉ (là bội của PAGE_SIZE).
 *                - Nếu tham số này khác NULL, thì kernel coi đây là một gợi ý
 *                để chọn vị trí của vùng memory mapping.
 *     length[I]: kích thước của vùng memory mapping, tính theo byte.
 *     prot  [I]: dùng để xác định quyền truy cập vào vùng memory mapping.
 *                Tham số này không được xung đột với chế độ mở file.
 *                Tham số này có thể nhận một trong số các giá trị sau:
 *                - PROT_EXEC: vùng này được phép thực thi.
 *                - PROT_READ: vùng này được phép đọc.
 *                - PROT_WRITE: vùng này được phép ghi.
 *                - PROT_NONE: vùng này không được phép truy cập.
 *     flag  [I]: dùng để xác định kiểu memory mapping. Tham số này phải nhận
 *                một trong hai giá trị sau:
 *                - MAP_SHARED: tạo ra shared mapping.
 *                - MAP_PRIVATE: tạo ra private mapping.
 *               Ngoài ra, ta có thể dùng phép OR để kết hợp với một số cờ
sau:
 *                - MAP_ANONYMOUS: tạo ra anonymous mapping. Toàn bộ ô nhớ trên
 *                vùng này được khởi tạo là 0.
 *                - MAP_UNINITIALIZED: không khởi tạo vùng anonymous mapping.
 *                - MAP_FIXED: bắt buộc phải tạo ra vùng memory mapping tại
 *                vị trí @addr. Trong trường hợp này, @addr phải là bội của
 *                PAGE_SIZE. Ta nên tránh sử dụng cờ này, bởi vì có thể làm
 *                tổn thương một vùng memory mapping khác.
 *                ...
 *     fd    [I]: cấu trúc dữ liệu đại diện cho một file.
 *     offset[I]: Một phần của file, có kích thước @length, tính từ vị trí
 *                @offset sẽ được ánh xạ lên vùng memory mapping. Giá trị
 *                truyền cho @offset phải là bội của PAGE_SIZE.
```

Triển khai memory mapping trên user space

```
 * giá trị trả về:
 *    - Nếu thành công, một vùng memory mapping sẽ được tạo ra, và hàm này sẽ
 *      trả về địa chỉ ảo của vùng đó. Địa chỉ ảo này luôn lớn hơn giá trị
 *      trong /proc/sys/vm/mmap_min_addr. Thông tin của vùng này sẽ xuất hiện
 *      trong /proc/<PID>/maps.
 *    - Nếu thất bại, hàm này sẽ trả về NULL.
 * chú ý:
 *    1. Nếu muốn tạo ra file mapping, ta phải gọi system call open trước khi
 *    gọi mmap. Giá trị trả về của open sẽ được truyền cho tham số @fd.
 *    2. Nếu muốn tạo ra anonymous mapping, ta không quan tâm tới @fd và
@offset.
 *    3. Sau khi mmap trả về, ta có thể đóng file mà không làm ảnh hưởng tới
 *    vùng memory mapping.
 *    4. Giá trị trả về của mmap có thể khác với tham số @addr.
 *    5. Kích thước thật sự của vùng memory mapping là bội của PAGE_SIZE,
 *    và có thể khác với tham số @length.
 */
#include <sys/mman.h>
void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t
offset);
```

Triển khai memory mapping trên user space

- Khi không cần dùng đến vùng memory mapping nữa, gọi system call **munmap** để xóa vùng đó đi

```
/*
 * System call munmmap
 * chức năng: xóa bỏ một vùng memory mapping trên user space của tiến trình.
 *
 * tham số đầu vào:
 *     addr  [I]: địa chỉ của vùng memory mapping trên user space.
 *                Tham số này phải là bội của PAGE_SIZE.
 *     length[I]: kích thước của vùng memory mapping, tính theo byte.
 * giá trị trả về:
 *    Nếu thành công, hàm này sẽ trả về 0.
 *    Nếu thất bại, hàm này sẽ trả về -1. Mã lỗi cụ thể được đặt trong biến
errno
 */

#include <sys/mman.h>
int munmap(void *addr, size_t length);
```

Triển khai memory mapping dưới kernel space

- Khi tiến trình gọi system call **mmap** trên một device file, thì Linux kernel sẽ kích hoạt một hàm của device driver.
- Hàm này được gọi là entry point mmap, có nhiệm vụ ánh xạ một dải địa chỉ vật lý vào vùng memory mapping trên user space của tiến trình. Dải địa chỉ vật lý có thể là địa chỉ của các ô nhớ trên RAM hoặc các thanh ghi/ô nhớ trên IO module

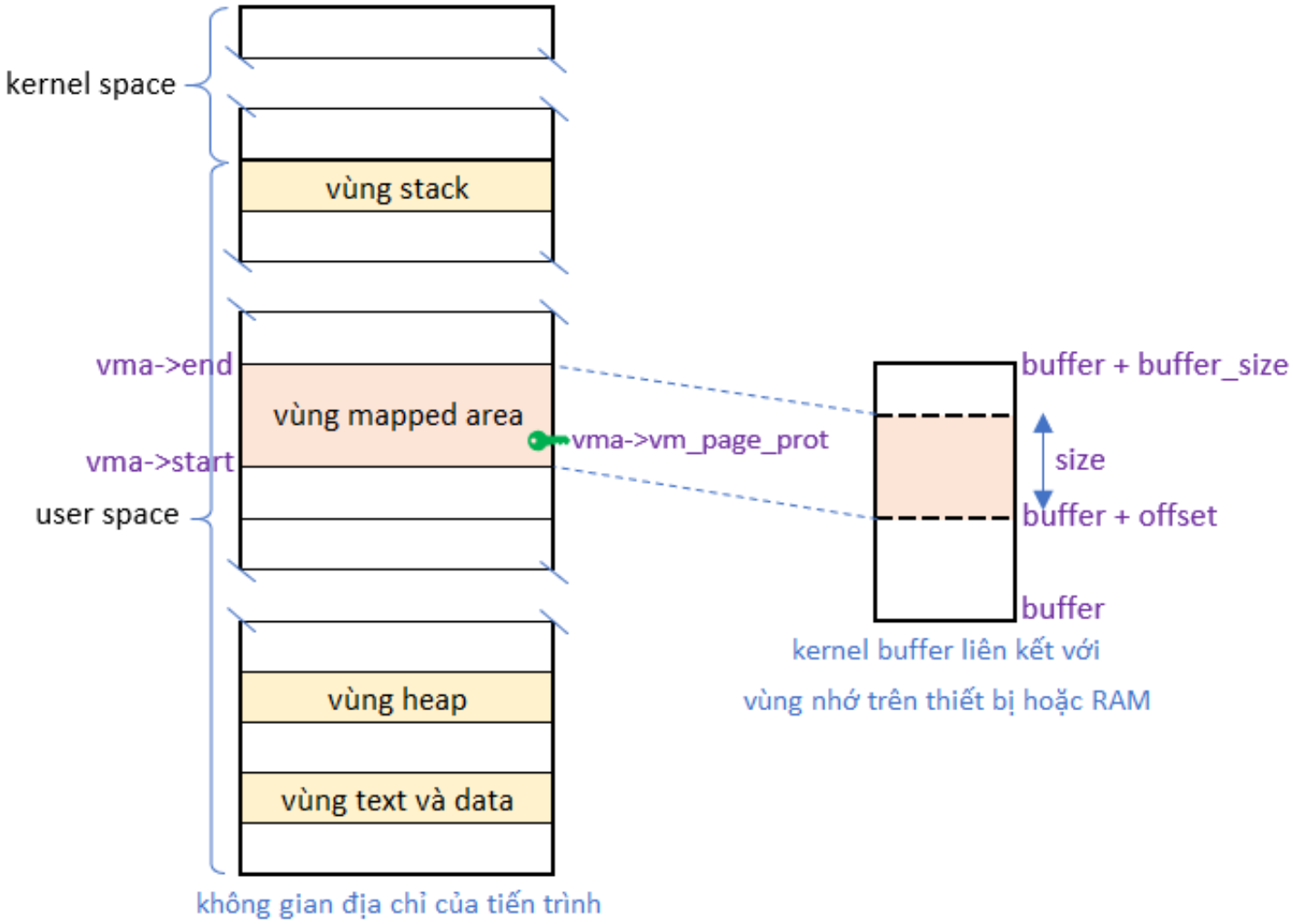
```
/*
 * entry point mmap
 * chức năng: ánh xạ một dải địa chỉ vật lý vào vùng memory mapping
 *            trên user space của tiến trình.
 * tham số đầu vào:
 *     *filp [I]: địa chỉ của cấu trúc file. Cấu trúc này mô tả một
 *                device file đang mở.
 *     *vma  [I]: địa chỉ của cấu trúc vm_ara_struct. Cấu trúc này mô tả
 *                vùng memory mapping trên user space của tiến trình.
 *                Linux kernel đã khởi tạo biến cấu trúc này trước khi
 *                gọi entry point mmap.
 * giá trị trả về:
 *      Trả về 0 thể hiện rằng quá trình ánh xạ thành công.
 *      Trả về một số âm nếu có lỗi.
 */
int (*mmap) (struct file *filp, struct vm_area_struct *vma);
```

Triển khai memory mapping dưới kernel space

- Để ánh xạ một dải địa chỉ vật lý vào vùng memory mapping, entry point **mmap** cần phải thiết lập bảng page table một cách thích hợp
  - Việc này có thể đạt được nhờ hàm remap_pfn_range hoặc hàm io_remap_pfn_range.
  - Hàm **remap_pfn_range** được sử dụng trong trường hợp vùng nhớ vật lý nằm trên RAM, còn hàm **io_remap_pfn_range** được sử dụng trong trường hợp vùng nhớ vật lý nằm trên thiết bị

```
/*
 * hàm remap_pfn_range và io_remap_pfn_range
 * chức năng:  ánh xạ một dải địa chỉ vật lý (đã liên kết với kernel space)
 *             vào vùng memory mapping của tiến trình. Dải địa chỉ vật lý
 *             có thể là địa chỉ dành cho RAM, hoặc địa chỉ dành cho IO module.
 * tham số đầu vào:
 *     *vma [I]: địa chỉ của cấu trúc vm_ara_struct. Cấu trúc này mô tả
 *               vùng memory mapping trên user space. Ta thường gán
 *               con trỏ vma của entry point mmap cho tham số này.
 *     addr [I]: địa chỉ user virtual address của vùng memory mapping.
 *               Ta thường gán vma->start cho tham số này.
 *     pfn  [I]: là số hiệu của page frame trên không gian địa chỉ vật lý cần
được
 *               ánh xạ vào vùng memory mapping.
 *               - Nếu sử dụng hàm remap_pfn_range, giá trị của tham số này
thường
 *                 được tính từ vma->vm_pgoff (page offset) như sau:
 *                 offset = vma->vm_pgoff << PAGE_SHIFT
 *                 Nếu (buffer được cấp phát bằng kmalloc hoặc
__get_free_pages)
 *                    thì pfn = virt_to_phys(buffer + offset) >> PAGE_SHIFT
 *                    hoặc pfn = virt_to_pfn(buffer + offset)
 *                 Nếu (buffer được cấp phát bằng vmalloc)
 *                    thì pfn = vmalloc_to_pfn(buffer + offset)
 *                 Nếu (buffer được cấp phát bằng alloc_pages)
 *                    thì pfn = page_to_pfn(page)
 *               - Nếu sử dụng hàm io_remap_pfn_range, giá trị của tham số này
thường
 *                 được tính như sau:
 *                 pfn = phy_addr >> PAGE_SHIFT + vma->vm_pgoff
 *     size [I]: là kích thước của vùng memory mapping.
 *               Ta thường gán (vma->end - vma->start) cho tham số này.
 *     prot [I]: là cờ điều khiển quyền truy cập vùng memory mapping.
 *               Ta thường gán vma->vm_page_prot cho tham số này.
 * giá trị trả về:
 *     Trả về 0 thể hiện rằng quá trình ánh xạ thành công.
 *     Trả về một số âm nếu có lỗi.
 * chú ý:
 *     1. Hàm này ánh xạ dải địa chỉ vật lý [@pfn<<PAGE_SHIFT ;
@pfn<<PAGE_SHIFT + @size]
 *     vào vùng memory mapping [@addr ; @addr + @size]
 *     2. size < kích thước dải địa chỉ vật lý - offset
 */
int remap_pfn_range(structure vm_area_struct *vma, unsigned long addr,
                    unsigned long pfn, unsigned long size, pgprot_t prot);
int io_remap_pfn_range(structure vm_area_struct *vma, unsigned long addr,
                    unsigned long pfn, unsigned long size, pgprot_t prot);
```

Hoạt động ánh xạ bộ nhớ của thiết bị vào không gian địa chỉ của tiến trình



Tóm lược các bước triển khai entry point **mmap** của một device driver

```
static int my_mmap(struct file *filp, struct vm_area_struct *vma)
{
    /* bước 1: xác định vị trí trên vùng nhớ vật lý bắt đầu được ánh xạ */
    unsigned long offset = vma->vm_pgoff << PAGE_SHIFT;
    if (offset >= buffer_size)
        return -EINVAL;

    /* bước 2: xác định kích thước vùng nhớ vật lý dùng để ánh xạ */
    unsigned long size = vma->vm_end - vma->vm_start;
    if (size > (buffer_size - offset))
        return -EINVAL;

    /* bước 3: xác định PFN của page frame tương ứng chứa @offset */
    //Giả sử buffer do kmalloc tạo ra
    pfn = virt_to_phys(buffer + offset) >> PAGE_SHIFT;

    /* bước 4: Nếu ánh xạ bộ nhớ trên thiết bị thì điều chỉnh cờ của vma */
    //không sử dụng bộ nhớ cache
    vma->vm_page_prot = pgprot_noncached(vma->vm_page_prot);
    //Nếu không muốn swap out vùng memory mapping ra ổ cứng (Tùy chọn)
    vma->vm_flags |= VM_RESERVED;

    /*
     * bước 5: gọi hàm remap_pfn_range nếu ánh xạ RAM vào memory mapping
     * hoặc gọi hàm io_remap_pfn_range nếu ánh xạ IO memory region vào.
     */
    if (remap_pfn_range(vma, vma->vm_start, pfn, size, vma->vm_page_prot)) {
        return -EAGAIN;
    }
    return 0;
}

static const struct file_operations my_fops = {
   .owner = THIS_MODULE,
   ...
   .mmap = my_mmap,
   ...
};
```

HỎI - ĐÁP