# Лекция 4.3 Реализация вызова mmap

Разработал: Максимов А.Н.

# Способы взаимодействия

- **Драйвер в ядре**
- **Драйвер в пользовательском пространстве**

# Драйвер в пользовательском пространстве

Ядро Linux предоставляет определенные механизмы для доступа к оборудованию непосредственно из пользовательского пространства:

• Устройства, подключенные к памяти с UIO, включая обработку прерываний, драйвер-api/uio-howto

• При помощи mmap или /dev/mem

• USB-устройства с libusb, https://libusb.info/

• Устройства SPI с поддержкой spidev, spi/spidev

• Устройства I2C с i2c dev, i2c/dev-интерфейсом

Эти решения имеет смысл использовать использовать в том случае, если:

-Нет необходимости использовать существующую подсистему ядра, такую как сетевойстек или файловые системы.

-Ядру нет необходимости выступать в качестве “мультиплексора” для устройства: к устройству обращается только одно приложение.

-Некоторые классы устройств, такие как принтеры и сканеры, не имеют никакой поддержки ядра, они всегда обрабатывались в пользовательском пространстве по историческим причинам.

- В противном случае это предпочтительно использовать ядерные драйверы.

# Реализация  вызова mmap в драйверах.

Информация по системе управлению памятью

Назначение mmap

Реализация в ядре

Реализация в пользовательском приложении

Пример

# Типы адресов

User virtual addresses – обычный адрес использующийся пользоватьскими программами 32 или 64 бита длины. Каждый процесс имеет свое адресное пространство.

Physical addresses

Адрес который используется между процессором и системной памятью.The addresses used between the processor and the system's memory. Physical addresses 32 или 64-bit. 32-ные системы могут иметь большее физическое пространство в ряде случаев.

Bus addresses

Адрес используемы между перефирийной шиной и памятью. Обычно такой же как физический. Некоторые процессоры имею I/O memory management unit (IOMMU) который транслирует адреса между шиной и основной памятью.

Логический адрес ядра

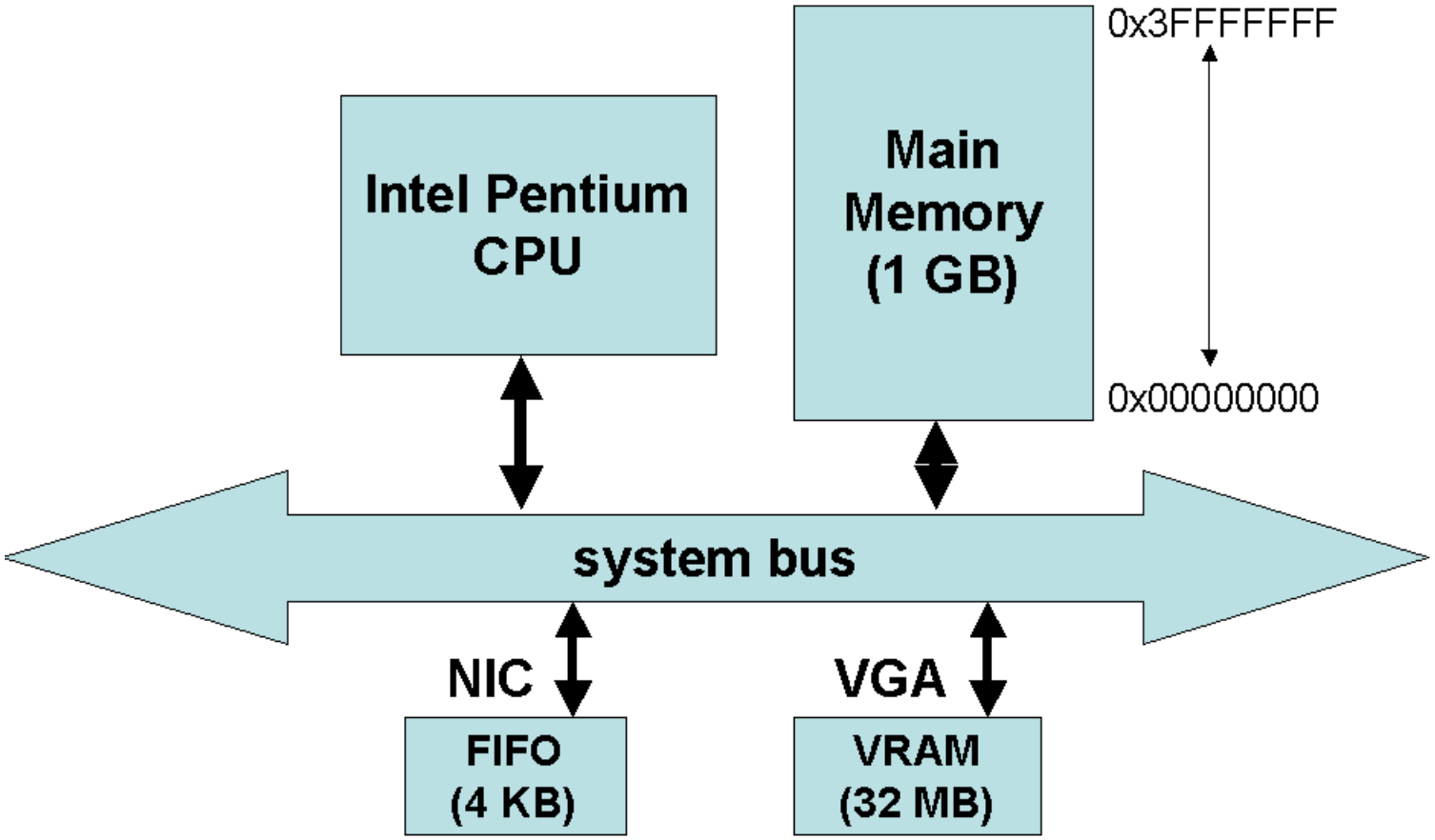
Обычный адрес в пространстве ядра. В большинстве архитектур логический адрес и физический адрес различаются только на константу. В случае наличия большого количества физической памяти логических адресов для доступа к ней может нахватать. kmalloc возвращает логический адрес.

<asm/page.h>

__pa( )  logical ->physical

__va( )  physical -> logical

# Архитектура памяти системы




Allan B. Cruse Advanced Systems Programming. Lecture 7. University of San Francisco. (lect_usfca1)

# Инициализация памяти в процессе загрузки

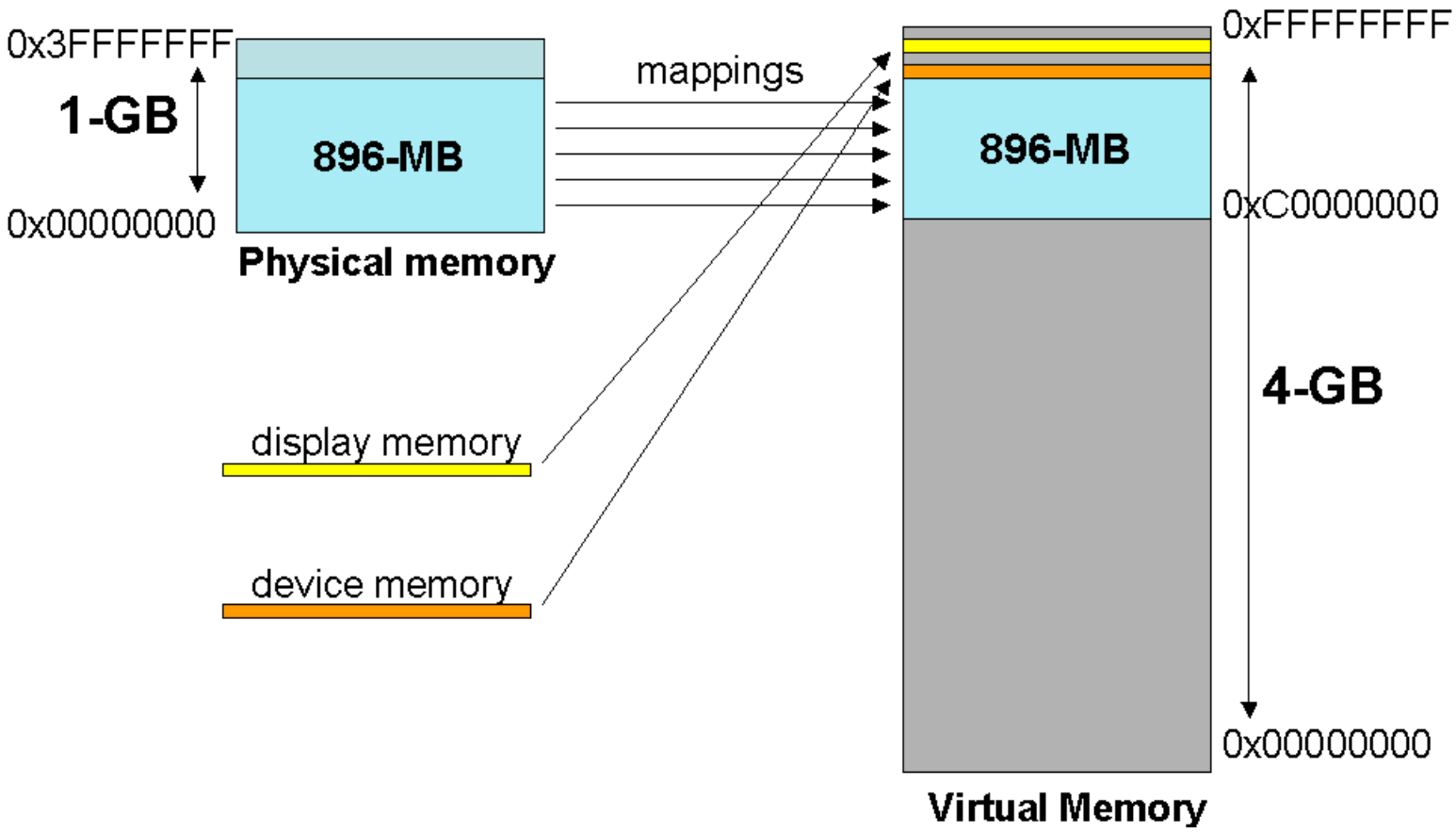
После загрузки CPU x86 системы видит только1-MB физической памяти

Код инициализации linux активирует "protected-mode” так чтобы вся память, включая память устройств стала доступной

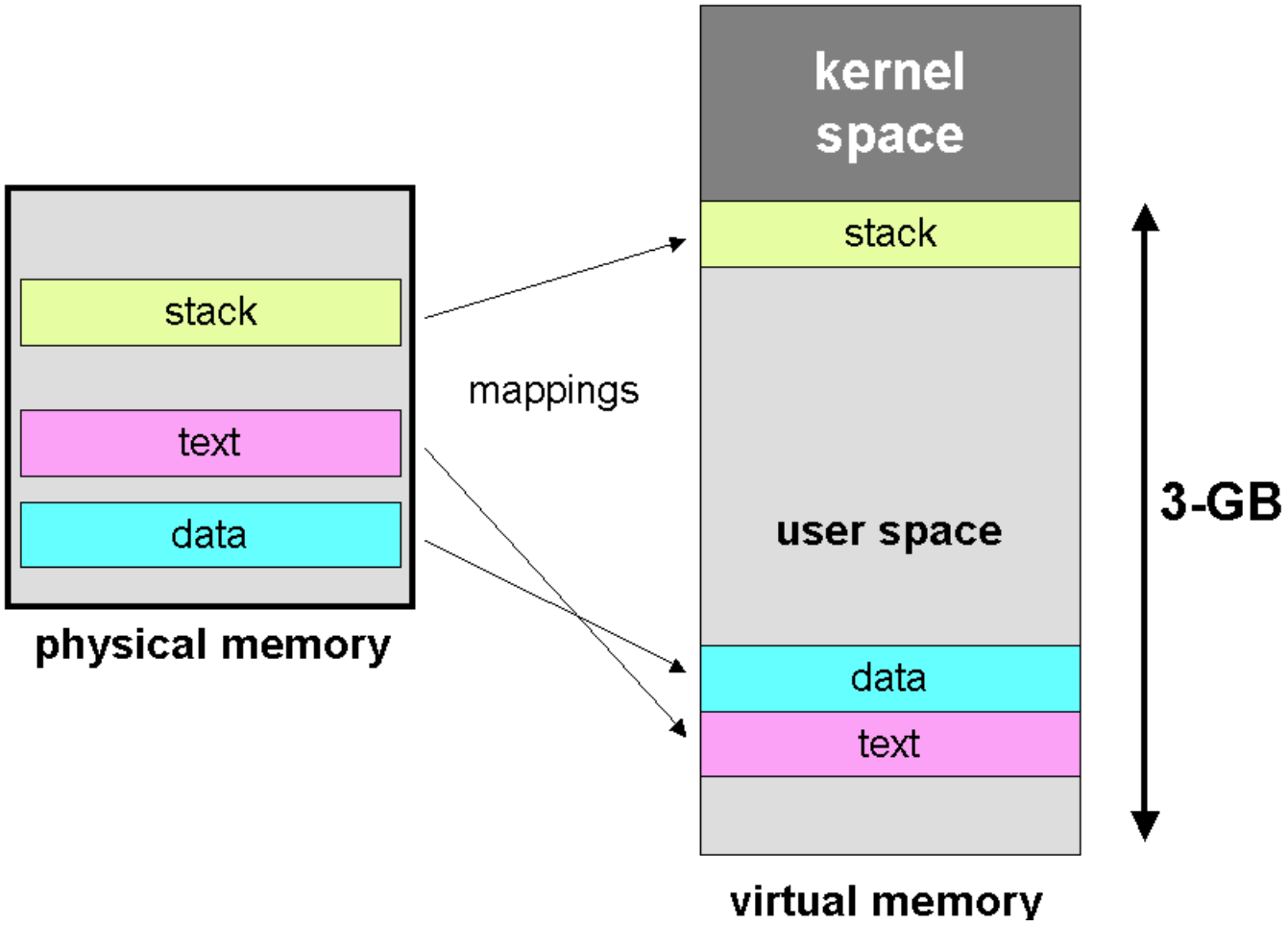
Linux инициализирует виртуальную память "virtual memory" становится доступно 4-GB адресное пространство

Allan B. Cruse Advanced Systems Programming. Lecture 7. University of San Francisco. (lect_usfca1)

# Адресное пространство ядра

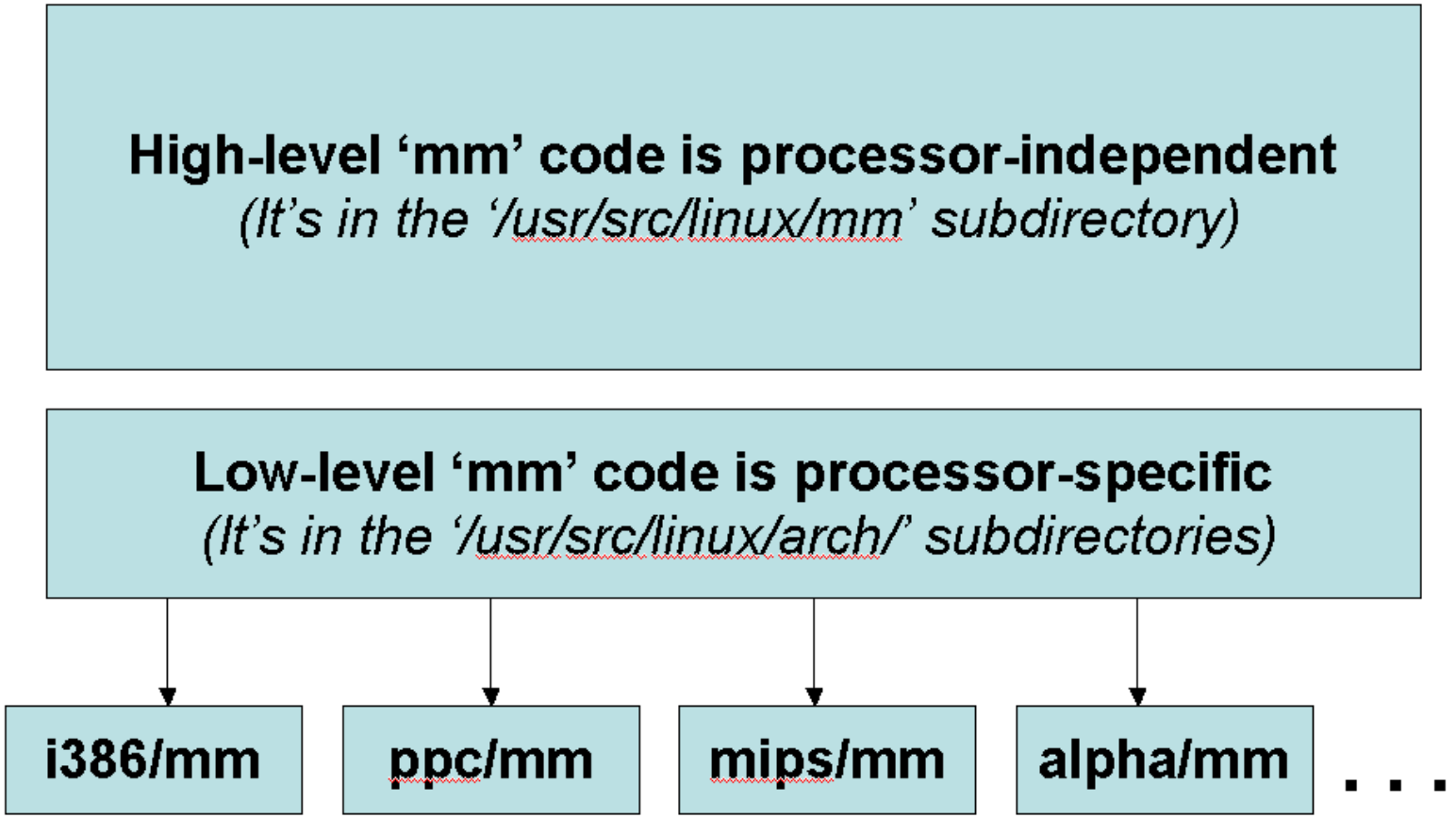


# Адресное пространство задачи




Allan B. Cruse Advanced Systems Programming. Lecture 7. University of San Francisco. (lect_usfca1)

# Исходный код mm



# mmap

Существует возможность отображать части виртуальной памяти на содержимое файлов.

Для представления области виртуальной памяти с однородными свойствами используется структура Virtual Memery Area (VMA).

VMA обладает следующими свойствами:

- последовательный диапазон виртуальных адресов;
- Одинаковые права доступа
- Сохраняются в одном объекте (файл или область swap)

# VMA пользовательского процесса

Пользовательский процесс имеет несколько областей VMA :

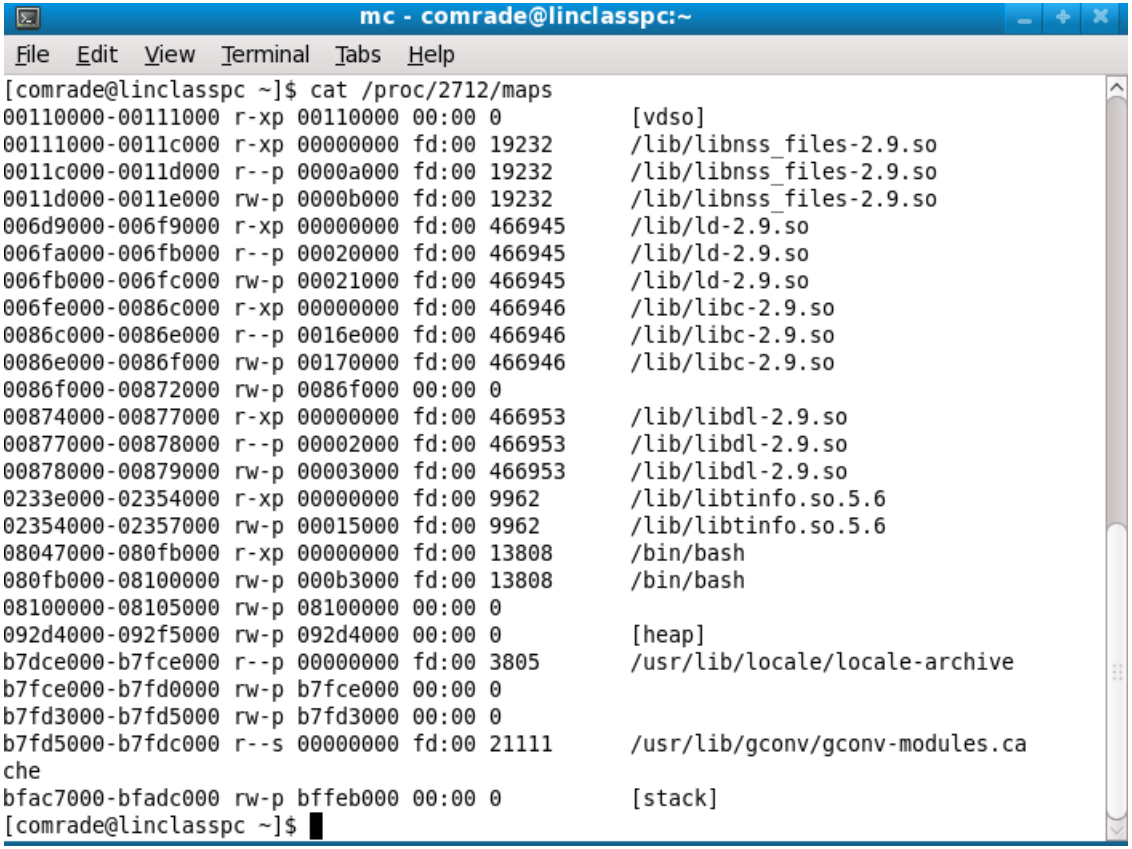
- Область выполняемого кода (text)
- Областей данных
- Область стека
- Неинициализированные данные (BSS)
- Области отображений (одна область на отображение)

(иногда области vma рассматриваются как сегменты)

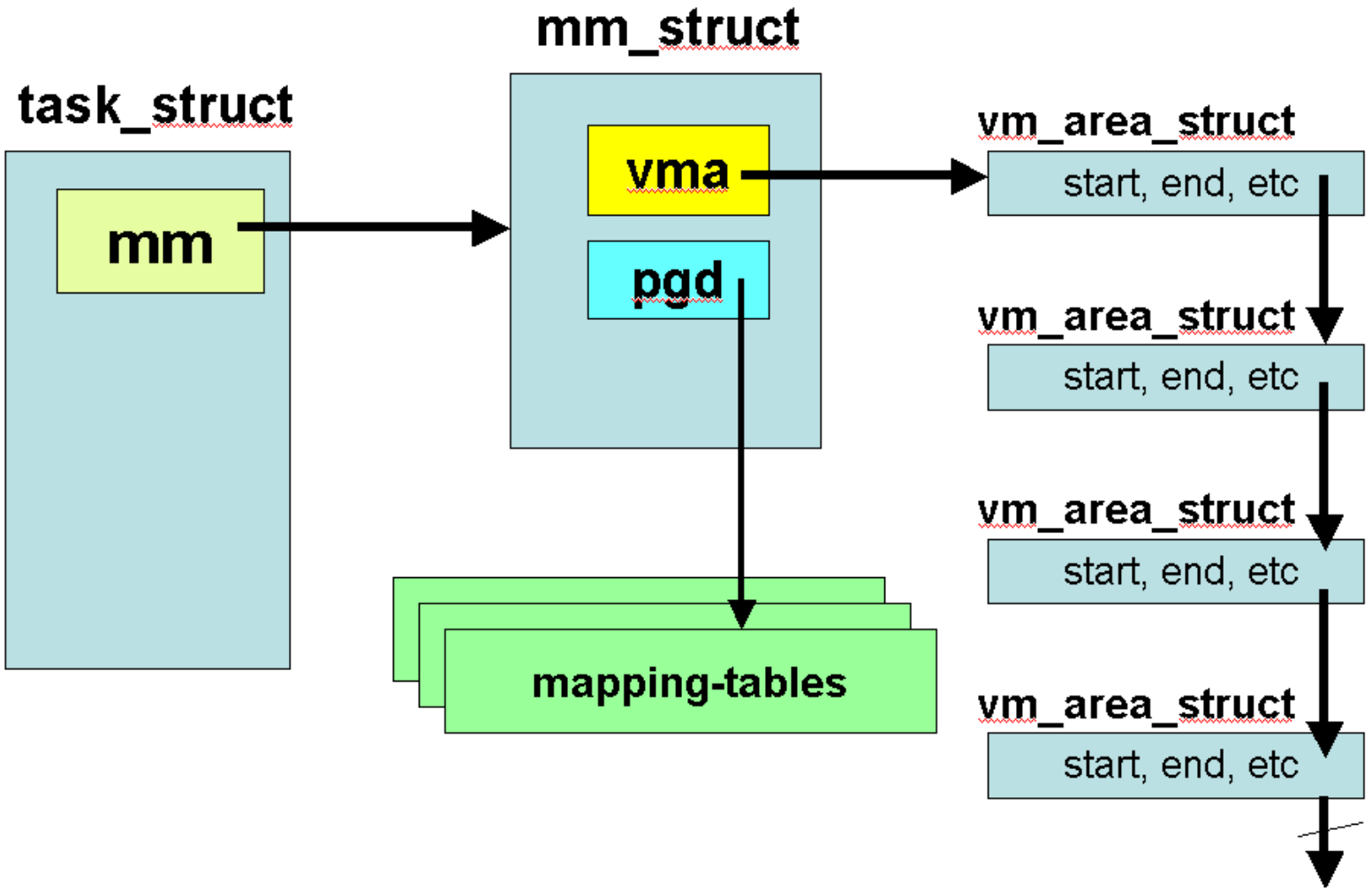
# Карта vma процесса

Можно посмотреть карты процесса через /proc/<pid>/maps или pmap <pid>

Назначение полей следующее: start-end permision offset major-minor inode image



# Структуры данных для организации отображения памяти




Allan B. Cruse Advanced Systems Programming. Lecture 7. University of San Francisco. (lect_usfca1)

# STRUCT VM_AREA_STRUCT (linux\mm_types.h)

```
struct vm_area_struct {
    struct mm_struct * vm_mm;     /* The address space we belong to. */
    unsigned long vm_start;       /* Our start address within vm_mm. */
    unsigned long vm_end;         /* The first byte after our end address       within vm_mm. */
    struct vm_area_struct *vm_next;       /* linked list of VM areas per task, sorted by address */
    pgprot_t vm_page_prot;        /* Access permissions of this VMA. */
    unsigned long vm_flags;       /* Flags, see mm.h. */
    struct rb_node vm_rb;
    union {           struct {
              struct list_head list;
              void *parent;     /* aligns with prio_tree_node parent */
              struct vm_area_struct *head;
          } vm_set;
          struct raw_prio_tree_node prio_tree_node;
    } shared;
    struct list_head anon_vma_node;     /* Serialized by anon_vma->lock */
    struct anon_vma *anon_vma;    /* Serialized by page_table_lock */
    struct vm_operations_struct * vm_ops;       /* Function pointers to deal with this struct. */
    /* Information about our backing store: */
    unsigned long vm_pgoff;       /* Offset (within vm_file) in PAGE_SIZE units, *not* PAGE_CACHE_SIZE */
    struct file * vm_file;        /* File we map to (can be NULL). */
    void * vm_private_data;       /* was vm_pte (shared mem) */
    unsigned long vm_truncate_count;/* truncate_count or restart_addr */
};
```

# Вызов mmap

mmap отображает часть виртуального пространства процесса на виртуальное пространство ядра.

Пользовательская задача:

1) Открыть устройство

2) Вызвать mmap :

```
void * mmap(  void *start,    /* Начальный адресOften 0, preferred starting address */
              size_t length, /* Размер области */
              int prot ,      /* Доступ: read, write, execute */
              int flags,      /* Опции: shared mapping, private copy... */
              int fd,         /* Дескриптор файла */
              off_t offset   /*  Смещение от начала файла*/
);
```

3) Получен виртуальные адрес.

# Пример пользовательской программы mmap

```
#include <stdio.h>// for printf(), perror()
#include <fcntl.h> // for open()
#include <stdlib.h>         // for exit()
#include <sys/mman.h>  // for mmap(), munmap()
int main( int argc, char **argv ) {
      int       fd = open("/dev/nic", O_RDONLY ); // open the device-file
      if ( fd < 0 ) { printf(“error open /dev/nic \n" ); exit(1); }
      int       size = 0x100; int    base = 0;
      void      *vm = mmap( NULL, size, PROT_READ, MAP_SHARED, fd, 0 );  // Отобразить io-memory
                                                                          // устройства в user-space
      if ( vm == MAP_FAILED ) { perror( " error mmap /dev/nic " ); exit(1); }
      printf( "\n\nRealTek 8139 Registers mapped at %p \n", vm );// Отобразить регистры
      unsigned long          *lp = (unsigned long *)vm;
      for (int i = 0; i < 64; i++) {
        if ( ( i % 8 ) == 0 ) printf( "\n%02X: ", i*4 );          printf( "%08lX ", lp[ i ] );
      }
      printf( "\n\n\n" );
}
```

# Реализация mmap в ядре

Реализуем вызов mmap из file_operation:

```
int (*mmap) ( struct file *, /* Open file structure */
                    struct vm_area_struct * /* Kernel VMA structure */
);
```

Запрос на отображение памяти устройство в адресное пространство пользовательского процесса.

Существуют несколько способов реализовать отображение

Простейший способ включает использование вызова remap_pfn_range(). Эта функция выполняет большенство работы.

# remap_pfn_range

Функция отражает участок памяти ядра в адресное пространство пользователя.

int remap_pfn_range(struct vm_area_struct * vma, unsigned long addr, unsigned long pfn, unsigned long size, pgprot_t prot);

vma структура vma используемая в отображении

addr адрес начала в пространстве пользователя

pfn (page frame number) Старшиие байты адреса физической памяти

(без байтов относящихся к размеру страницы page size)

size разрмер области памяти

prot флаги защиты страницы для данного отображения

(дополнительная информация http://lwn.net/Articles/104333/)

# Пример реализации отображения mmap

```
int my_mmap( struct file *file, struct vm_area_struct *vma ){
unsigned long     region_origin = vma->vm_pgoff * PAGE_SIZE;
unsigned long     region_length = vma->vm_end - vma->vm_start;
unsigned long     physical_addr = mmio_base + region_origin;
unsigned long     user_virtaddr = vma->vm_start;
unsigned long     phys_page_off = physical_addr & ~PAGE_MASK; // determine page-offset

if ( region_length > PAGE_SIZE ) return -EINVAL; // sanity check: mapped region is confined to just one
      page
vma->vm_flags |= VM_RESERVED;                    // tell the kernel not to try swapping out this region
vma->vm_flags |= VM_IO ;                         // tell the kernel that this region maps io memory
// ask the kernel to set up the required page-tables
//     if ( io_remap_page_range( vma, user_virtaddr, physical_addr>>PAGE_SHIFT,
if ( io_remap_pfn_range( vma, user_virtaddr, physical_addr>>PAGE_SHIFT,region_length,
       vma->vm_page_prot ) ) return -EAGAIN;
vma->vm_start += phys_page_off; // add page offset to virtual page start
 return       0;  // SUCCESS
}
```

# Пример реализации отображения mmap (2) Изменения API

Существует тенденция к замене вызова io_remap_page_range на io_remap_pfn_range

Дополнительная информация

http://www.mjmwired.net/kernel/Documentation/x86/pat.txt

http://lwn.net/Articles/104333/

# Пример реализации отображения mmap (3)

```
#include <linux/init.h>
#include <linux/module.h>    // for init_module()
#include <linux/fs.h>
#include <linux/kernel.h>
#include <linux/pci.h>                  // for pci_find_device()
#define VENDOR_ID  0x10EC       // RealTek Semiconductor Corp
#define      DEVICE_ID  0x8139    // RTL-8139 Network Processor
char modname[] = "mmap8139";    // for displaying module name
char devname[] = "nic";                 // for registering the driver
int my_major = 98;                      // static major-ID assignment
unsigned long mmio_base;                // registers physical address
unsigned long mmio_size;                // size of the registers bank
MODULE_LICENSE("GPL");

static struct file_operations
my_fops = {
   owner:            THIS_MODULE,
   mmap:             my_mmap,
};
void cleanup_module( void )
{
   unregister_chrdev( my_major, devname );
   printk( "<1>Removing \'%s\' module\n", modname );
}
```

# Практическое задание

Отразить область ввода-вывода Realtek 8139 в пространство пользователя

# Отображение памяти ядра. Пример

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/fs.h>
#include <linux/debugfs.h>
#include <linux/slab.h>
#include <linux/mm.h>
#ifndef VM_RESERVED
# define  VM_RESERVED   (VM_DONTEXPAND | VM_DONTDUMP)
#endif
struct dentry  *file;
struct mmap_info {
    char *data;
    int reference;
};
void mmap_open(struct vm_area_struct *vma) {
    struct mmap_info *info = (struct mmap_info *)vma->vm_private_data;
    info->reference++;
}
```

# Отображение памяти ядра. Пример

```
void mmap_close(struct vm_area_struct *vma) {
    struct mmap_info *info = (struct mmap_info *)vma->vm_private_data;
    info->reference--;
}
static int mmap_fault(struct vm_area_struct *vma, struct vm_fault *vmf) {
    struct page *page;
    struct mmap_info *info;
    info = (struct mmap_info *)vma->vm_private_data;
    if (!info->data)     {
        printk("No data\n");
        return 0;
    }
    page = virt_to_page(info->data);
    get_page(page);
    vmf->page = page;
    return 0;
}
```

# Отображение памяти ядра. Пример

```
struct vm_operations_struct mmap_vm_ops = {
    .open =     mmap_open,
    .close =    mmap_close,
    .fault =    mmap_fault,
};
int op_mmap(struct file *filp, struct vm_area_struct *vma) {
    vma->vm_ops = &mmap_vm_ops;
    vma->vm_flags |= VM_RESERVED;
    vma->vm_private_data = filp->private_data;
    mmap_open(vma);
    return 0;
}
int mmapfop_close(struct inode *inode, struct file *filp) {
    struct mmap_info *info = filp->private_data;
    free_page((unsigned long)info->data);
    kfree(info);
    filp->private_data = NULL;
    return 0;
}
```

# Отображение памяти ядра. Пример

```
int mmapfop_open(struct inode *inode, struct file *filp) {
    struct mmap_info *info = kmalloc(sizeof(struct mmap_info), GFP_KERNEL);
    info->data = (char *)get_zeroed_page(GFP_KERNEL);
    memcpy(info->data, "hello from kernel this is file: ", 32);
    memcpy(info->data + 32, filp->f_dentry->d_name.name, strlen(filp->f_dentry->d_name.name));
    /* assign this info struct to the file */
    filp->private_data = info; return 0;
}
static const struct file_operations mmap_fops = {
    .open = mmapfop_open,
    .release = mmapfop_close,
    .mmap = op_mmap,
};
static int __init mmapexample_module_init(void){
    file = debugfs_create_file("mmap_example", 0644, NULL, NULL, &mmap_fops); return 0;
}
static void __exit mmapexample_module_exit(void){
    debugfs_remove(file);
}
```

# Отображение памяти ядра. Пример

```
#include <fcntl.h>
#include <sys/mman.h>
#define PAGE_SIZE     4096
int main ( int argc, char **argv ) {
    int configfd;
    char * address = NULL;
    configfd = open("/sys/kernel/debug/mmap_example", O_RDWR);
    if(configfd < 0) {
        perror("Open call failed"); return -1;
    }
    address = mmap(NULL, PAGE_SIZE, PROT_READ|PROT_WRITE, MAP_SHARED, configfd, 0);
    if (address == MAP_FAILED) {
        perror("mmap operation failed");  return -1;
    }
    printf("Initial message: %s\n", address);
    memcpy(address + 11 , "*user*", 6);
    printf("Changed message: %s\n", address);
    close(configfd);
    return 0;
}
```

# Дополнительная информация.

http://www.mjmwired.net/kernel/Documentation/x86/pat.txt

http://lwn.net/Articles/104333/

https://coherentmusings.wordpress.com/2014/06/10/implementing-mmap-for-transferring-data-from-user-space-to-kernel-space/

http://stackoverflow.com/questions/10760479/mmap-kernel-buffer-to-user-space

# Отображение памяти ядра. Пример

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/fs.h>
#include <linux/debugfs.h>
#include <linux/slab.h>
#include <linux/mm.h>
#ifndef VM_RESERVED
# define  VM_RESERVED   (VM_DONTEXPAND | VM_DONTDUMP)
#endif
struct dentry  *file;
struct mmap_info {
    char *data;
    int reference;
};
void mmap_open(struct vm_area_struct *vma) {
    struct mmap_info *info = (struct mmap_info *)vma->vm_private_data;
    info->reference++;
}
```

# Отображение памяти ядра. Пример

```
void mmap_close(struct vm_area_struct *vma) {
    struct mmap_info *info = (struct mmap_info *)vma->vm_private_data;
    info->reference--;
}
static int mmap_fault(struct vm_area_struct *vma, struct vm_fault *vmf) {
    struct page *page;
    struct mmap_info *info;
    info = (struct mmap_info *)vma->vm_private_data;
    if (!info->data)     {
        printk("No data\n");
        return 0;
    }
    page = virt_to_page(info->data);
    get_page(page);
    vmf->page = page;
    return 0;
}
```

# Отображение памяти ядра. Пример

```
struct vm_operations_struct mmap_vm_ops = {
    .open =     mmap_open,
    .close =    mmap_close,
    .fault =    mmap_fault,
};
int op_mmap(struct file *filp, struct vm_area_struct *vma) {
    vma->vm_ops = &mmap_vm_ops;
    vma->vm_flags |= VM_RESERVED;
    vma->vm_private_data = filp->private_data;
    mmap_open(vma);
    return 0;
}
int mmapfop_close(struct inode *inode, struct file *filp) {
    struct mmap_info *info = filp->private_data;
    free_page((unsigned long)info->data);
    kfree(info);
    filp->private_data = NULL;
    return 0;
}
```

# Отображение памяти ядра. Пример

```
int mmapfop_open(struct inode *inode, struct file *filp) {
    struct mmap_info *info = kmalloc(sizeof(struct mmap_info), GFP_KERNEL);
    info->data = (char *)get_zeroed_page(GFP_KERNEL);
    memcpy(info->data, "hello from kernel this is file: ", 32);
    memcpy(info->data + 32, filp->f_dentry->d_name.name, strlen(filp->f_dentry->d_name.name));
    /* assign this info struct to the file */
    filp->private_data = info; return 0;
}
static const struct file_operations mmap_fops = {
    .open = mmapfop_open,
    .release = mmapfop_close,
    .mmap = op_mmap,
};
static int __init mmapexample_module_init(void){
    file = debugfs_create_file("mmap_example", 0644, NULL, NULL, &mmap_fops); return 0;
}
static void __exit mmapexample_module_exit(void){
    debugfs_remove(file);
}
```

# Отображение памяти ядра. Пример

```
#include <fcntl.h>
#include <sys/mman.h>
#define PAGE_SIZE     4096
int main ( int argc, char **argv ) {
    int configfd;
    char * address = NULL;
    configfd = open("/sys/kernel/debug/mmap_example", O_RDWR);
    if(configfd < 0) {
        perror("Open call failed"); return -1;
    }
    address = mmap(NULL, PAGE_SIZE, PROT_READ|PROT_WRITE, MAP_SHARED, configfd, 0);
    if (address == MAP_FAILED) {
        perror("mmap operation failed");  return -1;
    }
    printf("Initial message: %s\n", address);
    memcpy(address + 11 , "*user*", 6);
    printf("Changed message: %s\n", address);
    close(configfd);
    return 0;
}
```

# Дополнительная информация.

http://www.mjmwired.net/kernel/Documentation/x86/pat.txt

http://lwn.net/Articles/104333/

https://coherentmusings.wordpress.com/2014/06/10/implementing-mmap-for-transferring-data-from-user-space-to-kernel-space/

http://stackoverflow.com/questions/10760479/mmap-kernel-buffer-to-user-space