

Лекция 36

Управление памятью в ядре ОС

Виртуальная адресация

- Для многопроцессной обработки требуется защита памяти: процесс не должен иметь неавторизованный доступ к памяти других процессов и ядра
- Адреса ячеек памяти данных и программы, используемые в процессе, не обязаны совпадать с адресами в физической памяти (ОЗУ)
- Адреса ячеек памяти для процесса — **виртуальные адреса**
- Адреса ячеек памяти в оперативной памяти — **физические адреса**

Виртуальная адресация (память)

- Программно-аппаратный механизм трансляции виртуальных адресов в физические
- Аппаратная часть — отображение виртуальных адресов в физические в «обычной» ситуации — должно быть очень быстрой, так как необходимо для выполнения каждой инструкции
- Программная часть — подготовка отображения к работе, обработка исключительных ситуаций

Модели виртуальной адресации

- Модель база+смещение
 - Два регистра для процесса: регистр базы (B), регистр размера (Z)
 - Пусть V — виртуальный адрес (беззнаковое значение), если $V \geq Z$ — ошибка доступа к памяти, иначе
 - P — физический адрес, $P = B + V$

Сегментная адресация

- Каждый процесс состоит из нескольких сегментов: сегмент кода, сегмент стека, сегмент данных1, сегмент данных2
- Для каждого сегмента хранятся свои базовый адрес и размер
- У каждого сегмента свои права доступа, например:
 - Код: чтение + выполнение
 - Стек: чтение + запись
- Сегмент может отсутствовать в оперативной памяти и подгружаться по требованию

Страничная адресация

- Все пространство виртуальных адресов разбивается на страницы **равного размера**
- Каждая страница виртуальной памяти отображается на физическую память независимо от других
- Каждая страница имеет права доступа независимо от других страниц
- Страница может быть отмечена как неотображенная или отсутствующая в памяти
- При невозможности аппаратно отобразить виртуальную страницу в физическую — Page Fault

Отображение страниц

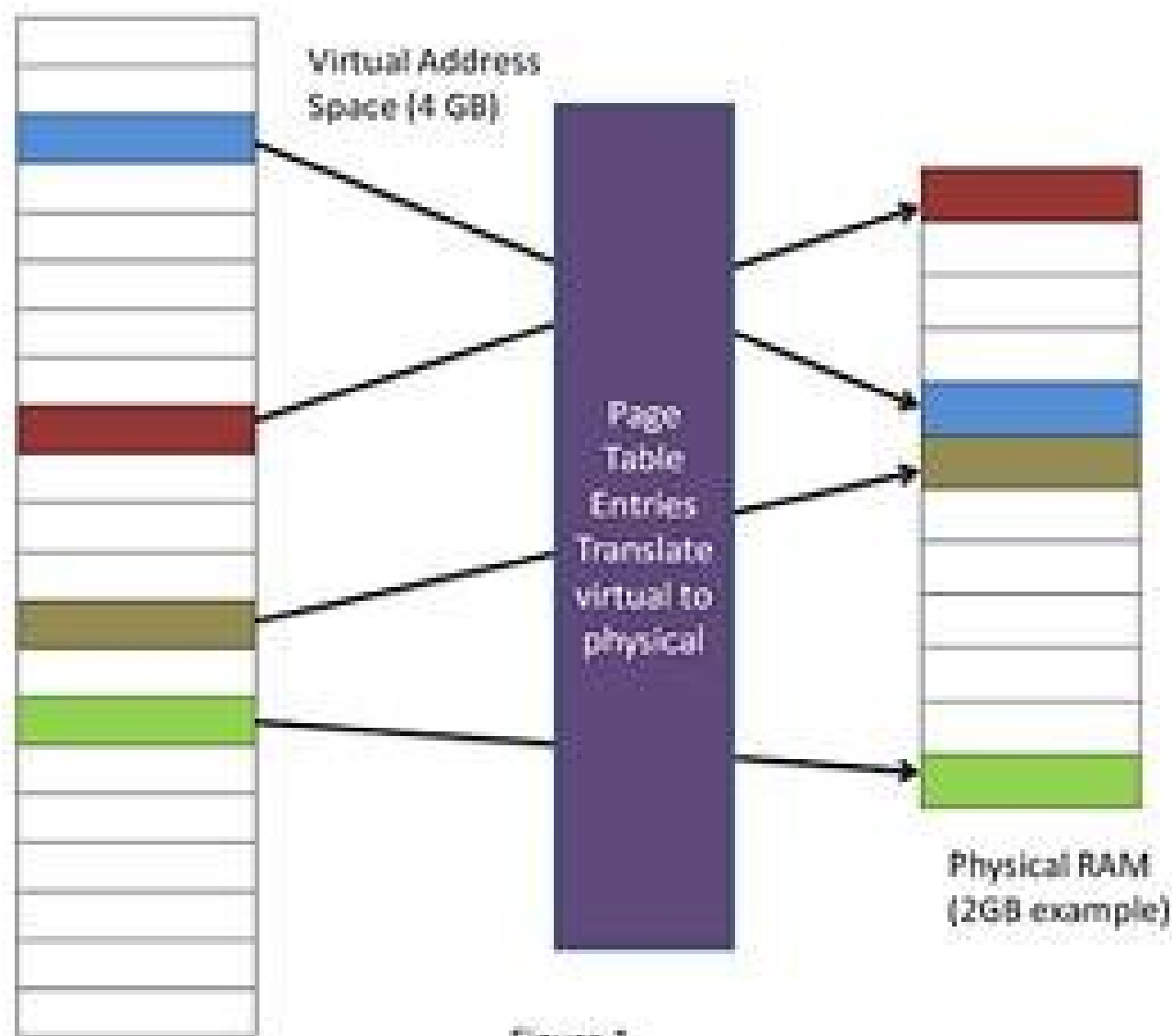
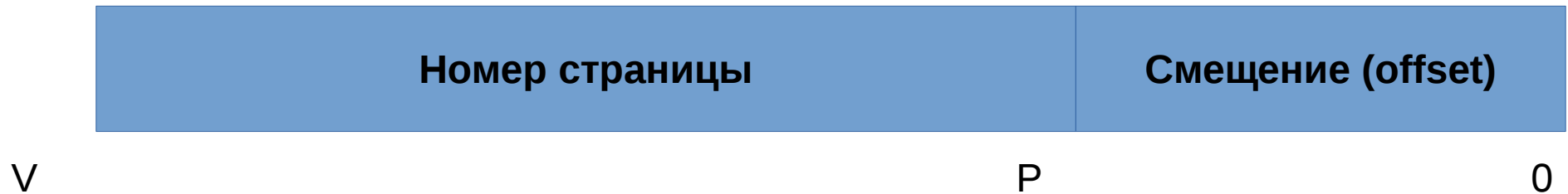


Figure 1

Виртуальный адрес



- V — количество бит виртуального адреса
- P — количество бит на смещение в страницу
- $(V - P)$ — количество бит на номер страницы
- Для x86: $V = 32$, $P = 12$, $V - P = 20$
 - Виртуальное адресное пространство 4GiB
 - Размер страницы — 4096 байт (4KiB)
 - 2^{20} (~1 Mi виртуальных страниц)

Двухуровневая таблица страниц (x86)

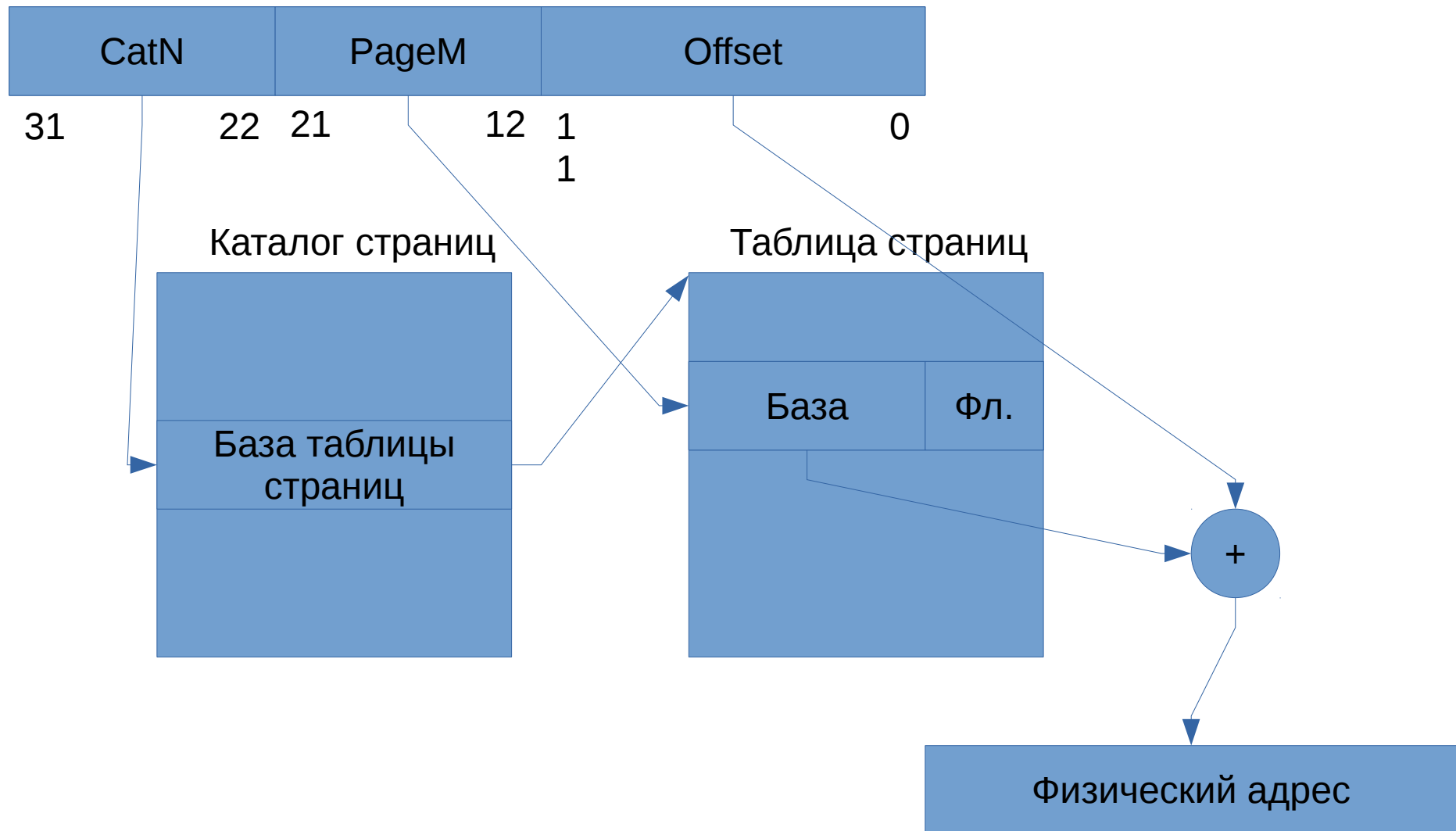


Таблица страниц

- Регистр процессора CR2 указывает на начало каталога страниц
- X86 — двухуровневая таблица страниц, размер страницы — 4KiB, в каталоге страниц 1024 записи, в каждой таблице страниц 1024 записи, одна запись — 4 байта
- X64 — четырехуровневая таблица страниц, размер страницы — 4KiB, в таблице каждого уровня 512 записей, одна запись — 8 байт.

Элемент таблицы страниц (x86)

Адрес физической страницы												Avail.	G	0	D	A	C	W	U	R	P
31												12	9								0

- P — страница присутствует в ОЗУ
- R — право на запись в страницу
- U — доступна из user-space
- C — кеширование страницы запрещено
- W — разрешена сквозная (write-through) запись
- A — к странице было обращение
- D — (dirty) страница была модифицирована
- G — страница глобальная

Трансляция адресов x86

```
#define PAGE_SIZE 4096
#define TABLE_SIZE 1024
unsigned translate(unsigned va)
{
    unsigned *catalog = CR2;
    unsigned *table = catalog[va >> 22] & -PAGE_SIZE;
    unsigned phys = table[(va >> 12) & (TABLE_SIZE - 1)]
& -PAGE_SIZE;
    return phys + (va & (PAGE_SIZE - 1));
}
```

Доступ к странице

- Если страница отсутствует в ОЗУ ($P == 0$), обращение к странице \rightarrow Page Fault
- Если в user-space и $U == 0 \rightarrow$ Page Fault
- Если записываем в страницу и $R == 0 \rightarrow$ Page Fault
- Устанавливаем флаг «accessed» ($A = 1$)
- Если записываем, устанавливаем флаг «dirty» ($D = 1$)

TLB (Translation Lookaside Buffer)

- Двухуровневая таблица страниц может потребовать 2 вспомогательных обращения к памяти!
- TLB — кэш-память для отображения виртуального адреса в физический
- TLB может быть многоуровневым и разделенным:
для Intel Nehalem:
 - 64 записи в L1 DTLB
 - 128 записей в L1 ITLB
 - 512 записей в L2 TLB

PageFault

- Исключение PageFault не обязательно ошибка в программе
- Допустимые ситуации:
 - Страница данных откачана в swap ($P == 0$)
 - Страница кода не загружена из файла ($P == 0$)
 - Запись в страницу созданную для copy-on-write ($R == 0$)
- Обработчик исключения определяет причину PageFault. Если PageFault произошел из-за ошибки, ошибка передается в программу

3G/1G virtual memory split

- Верхний 1GiB адресного пространства процесса содержит страницы, отображенные в режиме $U == 0$
- Страницы недоступны из user-space, но как только процесс переключается в kernel-space они становятся доступны
- Код и данные ядра отображаются в эту часть адресного пространства
- При переключении контекста (при входе в системный вызов) нет накладных расходов на перезагрузку таблицы страниц
- Из кода ядра непосредственно доступна память процесса (copy_from_user, copy_to_user)

Управление памятью в ядре Linux

Управление физической памятью

- `struct page`; - дескриптор физической страницы — создается для каждой страницы
- `sizeof(struct page) == 32` на x86
- `struct page *mem_map`; - массив всех физических страниц
- Если размер ОЗУ 4GiB, то массив `mem_map` займет 32MiB пространства ядра
- Определена в `include/linux/mm_types.h`

Struct page

- unsigned long flags;
 - PG_locked; - не может быть выгружена
 - PG_error; - I/O error на этой странице
 - PG_referenced; - было обращение к странице
 - PG_uptodate; - содержимое корректно
 - PG_dirty; - требует записи на диск
- struct address_space *mapping;
 - Либо указатель на пространство inode,
 - Либо на anon_vma (для анонимных страниц)

Struct page

- `atomic_t _mapcount;`
 - Счетчик использований данной физической страницы в PTE (page table entry) виртуального адресного пространства страниц
- `atomic_t _count;`
 - Счетчик ссылок
- `struct list_head lru;`
 - Поля `prev`, `next`, которые организуют страницы в список LRU
- `pgoff_t index;`
 - Смещение в отображении `mmap`
- `struct address_space *mapping;`
 - Структура, описывающее отображение страницы (inode)

Зоны физической памяти

- Области физической памяти
 - `ZONE_DMA` — пригодные для DMA
 - `ZONE_NORMAL` — непосредственно отображенные в виртуальное адресное пространство ядра
 - `ZONE_HIGH` — динамически отображаемые в виртуальное адресное пространство ядра
- `struct zone;` - структура, описывающая зоны
 - `zone_lru` — голова LRU списка страниц зоны
- `struct zone *node_zones;` - массив зон памяти

Дескриптор виртуальной памяти процесса

- `struct mm_struct;`
 - `struct vm_area_struct *mmap;`
 - Список областей памяти процесса
 - `pgd_t *pgd;`
 - Каталог виртуальных страниц верхнего уровня
 - `atomic_t mm_users;`
 - Счетчик использования из других процессов
 - `atomic_t mm_count;`
 - Основной счетчик использования

Дескриптор области виртуальной памяти

```
struct vm_area_struct {
    unsigned long vm_start; /* Start address within */
    unsigned long vm_end;   /* The first byte after end */
    struct vm_area_struct *vm_next, *vm_prev;
    /* linked list of VM areas per task, sorted by address */
    struct mm_struct *vm_mm; /* The address space we belong */
    pgprot_t vm_page_prot; /* Access permissions of VMA */
    unsigned long vm_flags; /* Flags, see mm.h. */
    struct anon_vma *anon_vma; /* анонимное отображение */
    const struct vm_operations_struct *vm_ops;
    unsigned long vm_pgoff; /* Offset (within vm_file) */
    struct file * vm_file; /* File map to (can be NULL) */
    // ...
};
```

Структура адресного пространства

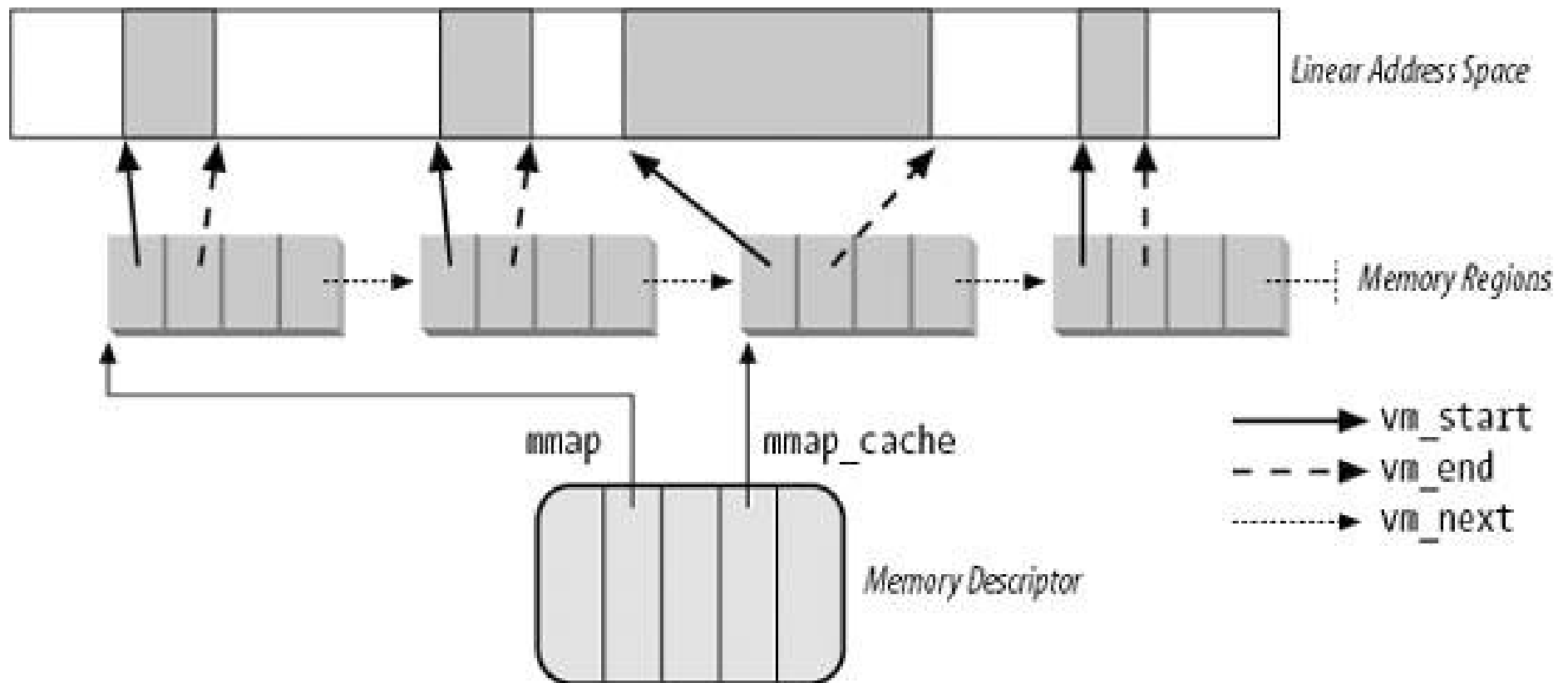
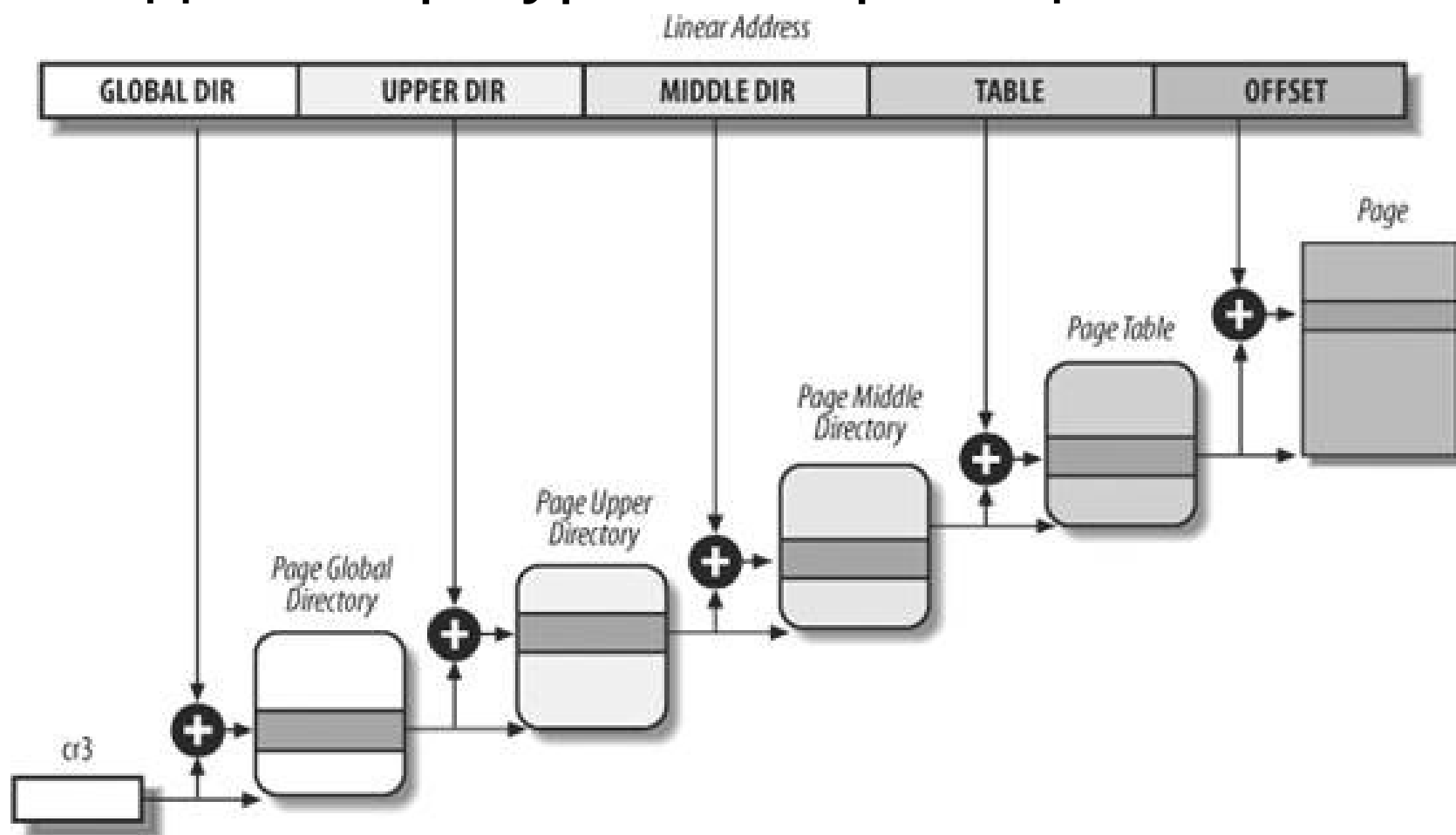


Таблица страниц в процессе

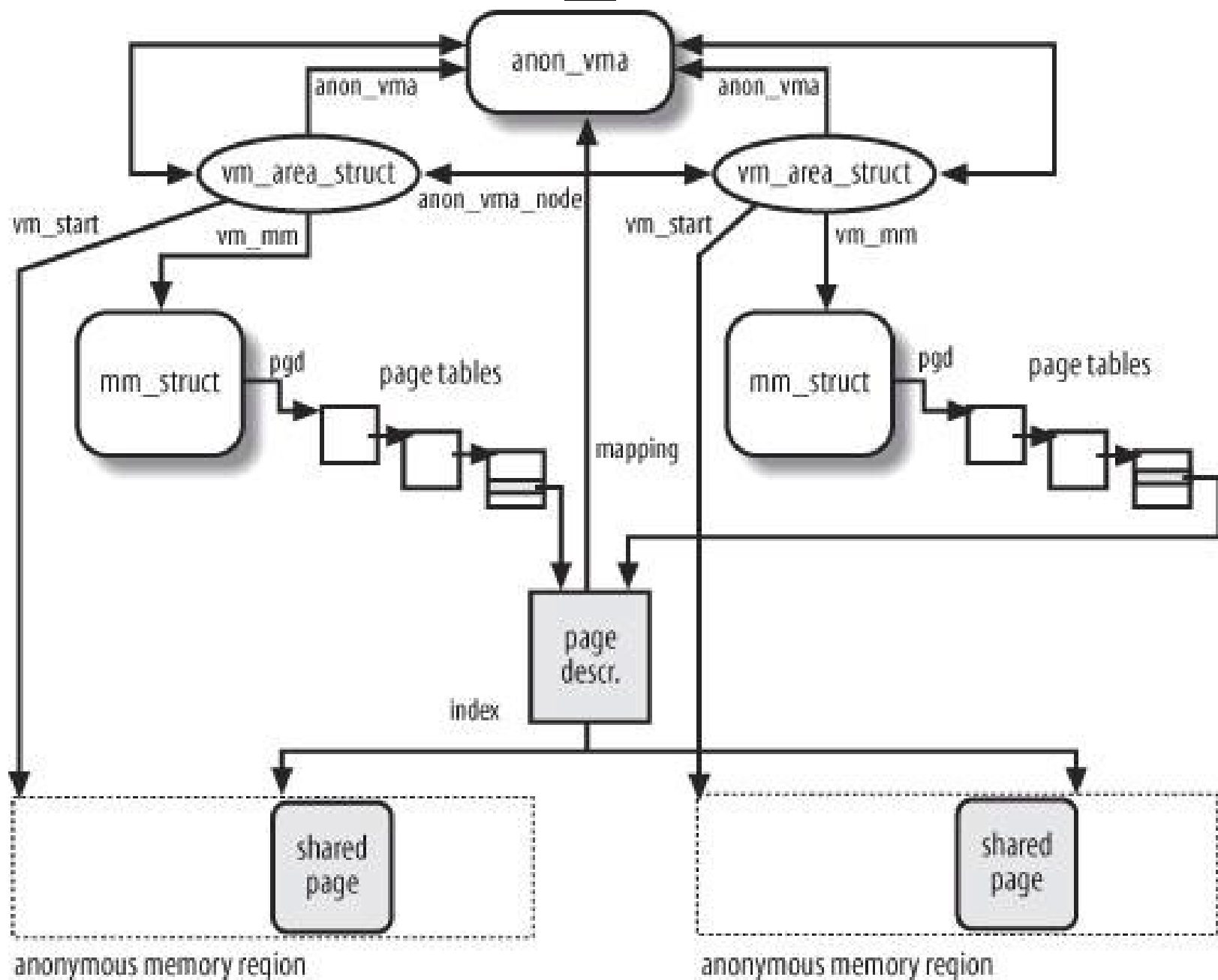
- Поле pgd в struct mm_struct
- Всегда четыре уровня страниц



Обратное отображение

- Задача: как внести изменения в структуры процессов при изменениях в состоянии физических страниц?
- У нас есть `struct page`, требуется найти все PTE процессов, которые ссылаются на эту физическую страницу
- Для анонимных отображений: `anon_vma`
- Для отображений из файла (`mmap`): `struct address_space`

anon_vma



Struct address_space

```
struct address_space {
    struct inode          *host;          /* owning inode
*/
    struct radix_tree_root page_tree; /* radix tree of
pages */
    struct prio_tree_root i_mmap;        /* list of all
mappings*/
    unsigned long          nrpages;       /* total number
of pages */
    pgoff_t                writeback_index; /* WB
start off */
    struct address_space_operations *a_ops; /* op
table */
    struct backing_dev_info *backing_dev_info; /* RA
info*/
};
```

Замещение страниц

- Алгоритм замещения страниц должен решить, какую из используемых страниц физической памяти освободить
- Может потребоваться запись в отображаемый файл или swar (для dirty страниц)
- Вызывается при Page Fault, когда нет свободных страниц или их меньше некоторого порога

Состояние страницы

- Для каждой страницы хранится два бита
 - R — из данной страницы было чтение
 - M — страница была модифицирована
- Бит R периодически (например, по таймерному прерыванию) очищается ядром ОС
- Бит M очищается только после записи страницы

Теоретически-оптимальный алг.

- (Алгоритм прорицателя :)
- Выгружается та страница памяти, которая не потребуется в будущем дольше всего
- Например, лучше выгрузить страницу, которая не будет нужна 5 секунд, чем страницу, которая не будет нужна 1 секунду
- Аккуратное предсказание в реальных условиях практически невозможно
- Можно собирать и накапливать информацию о предыдущих запусках, при последующих запусках поведение алгоритма будет приближаться к оптимальному

Not Recently Used

- Пытается удалять неиспользуемую в последнее время страницу (not recently used)
- Страница выбирается случайным образом из множества страниц наименьшего класса
 - (0) $R = 0, M = 0$
 - (1) $R = 1, M = 0$
 - (2) $R = 0, M = 1$
 - (3) $R = 1, M = 1$

Алгоритм FIFO

- Страницы, загружаемые в память, добавляются в конец очереди
- Выгружается страница из начала очереди
- Алгоритм второй попытки: если к первой в очереди странице было обращение, время загрузки обновляется, страница переставляется в конец списка на удаление

Алгоритм часов (clock)

- Физические страницы организованы в кольцевой список
- «Стрелка» (итератор) указывает на некоторую страницу
- При Page Fault:
 - Если у текущей страницы $R == 0$, она замещается, стрелка продвигается
 - Если $R == 1$, R очищается, стрелка продвигается пока не найдется страница с $R == 0$

Least Recently Used (LRU)

- Выталкивается страница, которая не использовалась дольше всего
- Демонстрирует производительность, близкую к идеальной
- «Настоящий LRU»:
 - при каждом обращении к странице она переставляется в конец списка страниц на выталкивание
 - при выталкивании страница берется из начала
 - очень дорогой и практически не реализуемый!

Two-list LRU (Linux)

- Два списка: активный и неактивный
- Если у страницы $R == 1$ она перемещается в конец активного списка
- Если активный список становится слишком большим, страницы из начала активного списка перемещаются в конец неактивного списка
- Для вытеснения страницы берутся из головы неактивного списка

Not Frequently Used (NFU)

- При каждом прерывании по таймеру к счетчику использования страницы прибавляется значение R
- Алгоритм старения: предположим, что под счетчик отводится K бит
- Значение счетчика для страницы пересчитывается по формуле:
$$\text{count} = (\text{count} \gg 1) \mid (1 \ll (K-1))$$