Управление памятью

Recap: виртуальная память

- Вся физическая память разделена на ***фреймы*** — куски размером 4096 байт
- Вся виртуальная память аналогично разделена на ***страницы***
- Трансляцией виртуальной памяти в физическую занимается ***memory management unit*** (MMU)
- Операционная система хранит четырёх-уровневые таблицы страниц для каждого процесса
- Таблица страниц переключается каждый раз при context switch
- Физический адрес текущей P4 хранит специальный регистр CR3

Higher half kernel

Higher half kernel

- x86-64 требует присутсвия IDT и interrupt handlers в текущем виртуальном AS
- Проще всего всё ядро поместить в память
- Ядерные адреса находятся выше `0xffff800000000000`
- Вся физическая память (точнее, первые 64 Тб) замаплена с `0xffff888000000000`

Немного про боль в 32-ух битном режиме

- Всю физическую память замапить было нельзя
- 3/1 split: 3 Гб под юзерспейс, 1 Гб под ядро
- Прямой маппинг первых 896 Мб физической памяти
- Остальные 128 Мб отведены для перевыделений через `kmap` / `kunmap`

Kernel remapping

- ОС стартует в первых мегабайтах памяти (HellOS стартует по адресу `0x200000`)
- Higher half вообще может не существовать как физическая память
- Идея: давайте ядро разделим на early-часть и основную
- Early часть будет запускаться загрузчиком, подготавливать нужные структуры процессора и запускать основную часть
- Early часть также будет ответственна за инициализацию аллокатора и сбор правильного memory layout (мы будем использовать похожий на тот, что используется в Linux)

Kernel remapping

- VMA = virtual memory address, адрес после загрузки — с ними работает линкер
- LMA = load memory address, *физический* адрес, по которому загрузчик поместит ядро
- Обычно LMA и VMA совпадают, но не в этот раз!
- Новые секции для early-кода: `.early.text`, `.early.data` и `.early.bss`
- Эти секции будут иметь LMA

linker.ld

```
SECTIONS
{
    . = 2M;

    .multiboot : { KEEP(*(.multiboot)) }

    // Секции для early части

    .early.text : { *(.boot.text) }
    .early.rodata : { *(.boot.rodata) }
    // ...

    _early_end = .;

    . = KERNEL_HIGH;

    // Основные секции ядра

    .text : AT(_early_end + ADDR(.text) - KERNEL_HIGH) { *(.text) }
    .rodata : AT(_early_end + ADDR(.rodata) - KERNEL_HIGH) { *(.rodata) }

    // ...
}
```

Как поместить функцию в другую секцию?

- ASM:

```
.section .early.text
func:
    ...
```

- C:

```
__attribute__((section(".early.text"))) void func() {
    ...
}
```

Выделение памяти в ОС

- Выделение физических фреймов
- Выделение виртуальных адресов для процессов
- Выделение памяти для нужд ядра

Выделение фреймов: linked-list allocator

- Хранит связный список свободных фреймов
- Указатель на следующий свободный фрейм хранится в первых восьми байтах предыдущего фрейма
- Прост в реализации
- Выделение/освобождение страниц за `O(1)`
- Если нужно выделять несколько страниц непрерывных физически, то сложность резко становится `O(N)`

Выделение фреймов: buddy allocator

- Статья
- Хранит списки непрерывных блоков страниц
- Размер блока кратен степени двойки
- Аллокатор ищет список с запрошенным размером и выдаёт страницы оттуда
- Если список пуст (нет блоков с нужным размером), то рекурсивно делится самый наибольший кусок, способный удовлетворить запрос
- Выделение/освобождение страниц за `O(1)`

Выделение виртуальной памяти для процессов

- С точки зрения процесса выделением виртуальной памяти занимается `mmap`
- Процесс просит память, ОС ему выдаёт нужный регион (или сама решает, где этот регион поставить)
- Однако, современные ОС не выделяют всю запрошенную память сразу
- Вместо этого они лишь сохраняют пометку о выделенном регионе памяти внутри специальной структуры
- При первом обращении к одной из страниц в регионе, происходит `#PF`
- Во время `#PF` выделяется новый физический фрейм и подмапливается в указанный виртуальный адрес
- После `iretq` прерванная инструкция повторяется и уже успешно читает/пишет в память
- Это называется *on-demand paging*

Выделение памяти для нужд ядра

- Выделение фреймов или нескольких фреймов подряд (например, для копирования данных с переферии при помощи DMA или для поддержания page tables): buddy allocator, фреймы уже замаплены => дешёвые аллокации
- Аллокация многих маленьких объектов (например, `struct task`, сокеты, файловые дескрипторы, итд): обычно для каждого типа объектов заводится отдельный аллокатор, который может эти объекты выделять (slab allocator)
- Выделение непрерывного виртуального, но не физически, региона памяти

Meltdown и Spectre

- Meltdown и Spectre — уязвимости **процессоров**, опубликованные в 2017 году
- Основаны на спекулятивном исполнении и кэше процессора
- Первоначальные уязвимости были опубликованы для Intel'овских процессоров
- Затем найдены аналогичные уязвимости для AMD и ARM

Как работает Meltdown?

```
char probes[256*64]; // 64 — размер кэш-линии
const char* ptr = ...; // любой адрес внутри ядра
char b = *ptr; // эта инструкция провоцирует page fault
char b2 = probes[b * 64]; // процессор спекулятивно читает эту ячейку

for (int i = 0; i < 256; i++) {
    start = hr_timer();
    char b = probes[i * 64];
    end = hr_timer();
    if (end - start <= CACHE_READ_TIME) {
        // pwned!
    }
}
```

Page table isolation

- PTI или KAISER
- Нет маппинга — нечего читать!
- Давайте просто хранить отображения только нужных частей — IDT и interrupt handlers — всё остальное будем перемапливать каждый раз при входе в ядро

Process-Context Identifiers (PCIDs)

- Смывать TLB на каждое переключение адресного пространства — очень дорого
- PCID — 12 битное число, "тэг" для TLB кэша
- Для включения нужно выставить `CR4.PCIDE = 1`
- Сам PCID указывается в нижних 12-ти битах `CR3`
- При трансляции адресов, MMU будет использовать только те записи, у которых PCID совпадает с текущим
- При смене `CR3` записи с другими PCID не будут инвалидироваться
- Ручная инвалидация TLB: `invlpg` и `invpcid`

Вопросы?