Виртуальная память

Источники: [1].

Виртуальная память — это метод управления памятью, который обеспечивает идеализированную абстракцию физической памяти, доступной на данном компьютере.

Операционная система, используя комбинацию программного и аппаратного обеспечения, отображает виртуальные адреса в физические. При этом основное хранилище, с точки зрения пользовательской программы, выглядит как непрерывное адресное пространство или набор непрерывных сегментов. Операционная система управляет отображением виртуальных адресов в физические, в то время как часть процессора, называемая Memory Management Unit (MMU) автоматически транслирует эти адреса.

Преимущества виртуальной памяти

- Освобождение пользовательских программ от необходимости управления общей памятью: приложения запускаются каждое в своём адресном пространстве, позволяя использовать его полностью, как если бы они запускались на отдельных машинах. В частности, можно использовать абсолютную адресацию.
- Возможность совместного использования динамических библиотек.
- Повышение безопасности за счёт изоляции памяти между процессами.
- Возможность концептуально использовать больше памяти, чем доступно физически: например, если нужно аллоцировать 100 мегабайт, а доступно 200 кусков физической памяти по 1 мегабайту — виртуальное адресное пространство больше физического, поэтому такая ситуация маловероятна.
- Возможность ленивого выделения памяти операционной системой.
- Возможность использовать дисковое хранилище, повышая объём доступной физической памяти.

Недостатки виртуальной памяти

- Системы виртуальной памяти запускают непредсказуемые ловушки (trap), вызывая задержки в ответ на ввод, что нежелательно для компьютерных систем, которым требуется очень быстрое или очень стабильное время отклика. (Не критично для операционных систем общего назначения)
- Аппаратное обеспечение для преобразования виртуальных адресов в физические требует значительной площади чипа для реализации, поэтому не все чипы его содержат. (Большинство современных процессоров, рассчитанных на массового пользователя предоставляют поддержку виртуальной памяти)
- Чуть более сложное устройство операционной системы. (Учитывая сложность устройства современных операционных систем, такое усложнение кажется несущественным)
- Замедление доступа к памяти. (Влияет ли это на эффективность программ? Может быть, этот недостаток нивелируется преимуществами виртуальной памяти?) (В 1960-х годах были опасения, что новые общесистемные алгоритмы будут менее эффективными, чем ранее используемые алгоритмы, специфические для приложений. Однако к 1969 году спор о виртуальной памяти для коммерческих компьютеров был окончен: исследовательская группа IBM под руководством Дэвида Сейра показала, что их система наложения виртуальной памяти постоянно работала лучше, чем лучшие системы с ручным управлением)

Мне стало интересно, может ли использование физической памяти быть полезным для производительности хотя бы некоторых приложений, которым не нужны преимущества использования виртуальной памяти. Поэтому я решил написать свой загрузчик, который будет измерять время работы некоторых алгоритмов с активным использованием памяти для физического и виртуального режимов.

Устройство виртуальной памяти в x86

Виртуальная память была введена в процессор Intel 80286, но она использовала сегментацию памяти: каждый сегмент (с произвольным началом и концом) виртуальной памяти транслировался в соответствующий сегмент физической памяти. Такая система плохо масштабировалась для больших размеров сегментов.

Поэтому современные системы виртуальной памяти используют страничную адресацию. Это означает, что вирутальное и физическое адресные пространства делятся на большое количество небольших страниц (от 4 килобайт, но бывают и больше) и каждая (используемая) виртуальная страница отображается на физическую.

При обращении к памяти (с выключенным кэшом) процессор смотрит на регистр CR3, в котором хранится физический адрес таблицы страниц. Каждый уровень таблицы страниц состоит из PTE (Page Table Entry), которые хранят флаги (например, является ли страница валидной) и, если страница валидна, то старшие биты её физического адреса. Каждой PTE соответствует определённый диапазон виртуальных адресов и каждый уровень таблицы страниц ссылается на следующий, в котором диапазон виртуальных адресов меньше. А последний уровень ссылается непосредственно на соответствующие физические страницы. Такое древовидное устройство таблицы страниц позволяет найти баланс между производительностью и потреблением памяти, так как предоставляет возможность отсекать большие диапазоны неиспользуемой виртуальной памяти уже на первом уровне.

Перейдём от теории к практике, а именно — напишем самый простой загрузчик.

Загрузчики (bootloaders)

Источники: [2], [3].

При включении компьютера процессор сразу начинает исполнять код так называемого BIOS, который находится по адресу, зашитому в процессор. Таким образом, он знает, где начинать. Этот BIOS передаёт управление загрузчику, который должен быть найден на диске, выбранном для загрузки в меню BIOS, с помощью определённого алгоритма. Загрузчик обычно настраивает конфигурацию процессора и памяти, необходимую для запуска операционной системы, но мне нужно просто запустить код.

На самом деле бывают разные виды “BIOS”: на достаточно старых компьютерах эта программа действительно называется BIOS (Basic Input/Output System), но в 2006 году появился более современный аналог, называемый UEFI (Unified Extensible Firmware Interface), который подвергся критике за то, что усложнил систему, не давая существенных преимуществ.

Поэтому написание загрузчика для BIOS сильно отличается от написания его для UEFI. Например, загрузчик, совместимый с BIOS, пишется на ассемблере и загружается по адресу `0x7C00`, в то время как у UEFI есть библиотека, которая позволяет обращаться к встроенным функциям для работы с внешними устройствами, поэтому его пишут на C. Также различается способ хранения загрузчика: для BIOS он хранится в сыром виде на специальном разделе MBR (Master Boot Record), а для UEFI — в файле `EFI/BOOT/BOOT<ARCH>.EFI` в файловой системе FAT. Бывают техники, с помощью которых на диск сохраняются сразу и BIOS, и UEFI загрузчики.

Для своей цели я выбрал загрузчик для BIOS. Во-первых, когда я искал в интернете гайды и примеры по написанию загрузчика, я постоянно находил именно этот вариант, поэтому не знал о существовании UEFI. Во-вторых, код на ассемблере, вероятно, позволяет лучше контролировать систему, чем высокоуровневый код на C.

Hello world загрузчик для BIOS

Источники: [4], [5], [6], [7].

Вот шаблон самого простого загрузчика для BIOS:

```
org 0x7c00          ; Start address
[bits 16]           ; Use 16-bit registers and instructions

; Write code here

    times 510-($-$$) db 0 ; Number of bytes must be 512
    dw 0xAA55       ; Last 2 bytes must be such
```

Здесь `org 0x7c00` — директива ассемблера, которая используется, чтобы обозначить адрес в памяти, по которому будет загружен код программы. Позволяет ассемблеру корректно посчитать абсолютный адрес каждой метки.

`[bits 16]` — тоже директива ассемблера, которая не компилируется ни в какой код, но управляет компиляцией остального кода: все инструкции, следующие после неё компилируются для 16-битного режима. Она нужна конкретно в начале загрузчика, потому что при включении процессор находится в Real Mode (16-битном режиме с физической адресацией).

`times 510-($-$$) db 0` — заполняет нулями исполняемый файл до 510 байта.

`dw 0xAA55` — магическое число, которым должны заканчиваться первые 512 байт загрузчика (те, что загружаются BIOS).

Далее, на месте комментария `;Write code here` напишем следующий код:

```
_start:
    xor ax, ax
    mov ds, ax      ; Data segment
    mov es, ax      ; Extra segment
    mov ss, ax      ; Stack segment
    mov sp, 0x8000  ; Initialize stack
```

Этот код инициализирует регистры сегментов, которые используются для перевода логических (не путать с виртуальными) адресов в физические. В зависимости от инструкции процессор выбирает соответствующий регистр сегмента, умножает его значение на 16 и прибавляет логический адрес. Это позволяет в том числе использовать 20-битные адреса в 16-битном режиме. Нам это не нужно, поэтому присваиваем им нули. Последняя инструкция инициализирует стек, присваивая Stack Pointer значение конца стека (стек будет расти вниз).

Напишем функцию печати:

```
_print:
.loop:
    lodsb           ; mov al, [si]; inc si
    cmp al, 0
    je .done
    mov ah, 0x0E    ; Write Character in TTY Mode
    int 0x10        ; BIOS video services
```

```
    jmp .loop
.done:
    ret
```

При вызове в регистре `si` должен быть адрес строки, которую необходимо напечатать.

`lodsb` — специальная инструкция x86, которая коипрует в `al` один байт из памяти, на которую указывает `si`, и увеличивает указатель.

`cmp al, 0` — если мы дошли до 0, то выйти из функции.

Чтобы напечатать символ, а также для других полезных операций, BIOS предоставляет прерывания (инструкция `int`). Например, для видео сервисов номер прерывания — `0x10` [7]. В регистре `ah` обычно записывается номер функции, которую нужно выполнить. В нашем случае — напечатать символ. Для этой функции аргумент должен находиться в регистре `al`, где он и находится. Таким образом, мы можем напечатать строку.

Код, вызывающий функцию:

```
    mov si, _message
    call _print
    jmp $           ; Hang
```

Пишем после инициализации стека, `jmp $` — инструкция, которая позволяет процессору зависнуть после исполнения.

Сообщение, которое мы хотим напечатать:

```
_message:
    db 'Hello world!', 0xA, 0xD, 0
```

`0xA, 0xD` — символы перевода строки.

Самый простой загрузчик готов. Компилируем его с помощью команды:

`nasm -Werror hello_world.asm -f bin -o hello_world.bin`. Запускаем на виртуальной машине (вдруг там какая-то ошибка, которая сломает компьютер) с помощью команды:

`qemu-system-x86_64 -nographic -drive format=raw,file=hello_world.bin`. Его вывод:

```
SeaBIOS (version 1.16.3-debian-1.16.3-2)


iPXE (https://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+06FCAF60+06F0AF60 CA00



Booting from Hard Disk...
Hello world!
```

Для сравнения, Hello world для UEFI пишется в одну инструкцию [3]:

```
#include <Uefi.h>
#include <Library/UefiLib.h>
#include <Library/ShellCEntryLib.h>

EFI_STATUS EFIAPI ShellAppMain(IN UINTN Argc, IN CHAR16 **Argv)
{
    Print(L"hello, world\n");
    return EFI_SUCCESS;
}
```

echo загрузчик для BIOS

Источники: [8].

Давайте попробуем сделать простейшее приложение с текстовым интерфейсом. Сейчас нажатия клавиатуры игнорируются. Попробуем их обработать. Для этого нам понадобится прерывание `0x16` (сервисы клавиатуры). Нажатия клавиатуры записываются BIOS в специальный буфер. Чтобы загрузить значение из этого буфера в `al`, нужно вызвать `int 0x16` с `0` в `ah`.

Попробуем прочитать символ из буфера и сразу же его вывести. В этот раз функция `print` не нужна. Код после инициализации стека выглядит следующим образом:

```
.loop:
    mov ah, 0x00    ; Read character
    int 0x16        ; Keyboard services
    mov ah, 0x0E    ; Write Character in TTY Mode
    int 0x10        ; BIOS video services
    jmp .loop
```

Запустив программу, можно убедиться, что она действительно работает. При этом в результате нажатия специальных клавиш выводятся символы, соответствующие их кодам, которые никак не связаны с этими клавишами. Таким образом, BIOS может считывать не только клавиши ASCII, но и стрелки, и комбинации клавиш.

Многостадийный загрузчик

Источники: [9], [10].

При включении компьютера, BIOS загружает только первые 512 байт загрузчика. Этого может не хватить как для моей цели, так и для запуска ОС. Поэтому существует техника, называемая многостадийным загрузчиком. Как понятно из называния, он состоит из нескольких стадий, в данном случае — двух: первая занимает 512 байт и загружает код для второй, в которой уже выполняется полезная работа.

Напишем первую стадию. Для начала (после инициализации стека) нужно инициализировать диск:

```
    xor ax, ax      ; Reset Disk Drives
    int 13h         ; Low Level Disk Services
    mov si, disk_reset_error_message
    jc panic
```

При запуске загрузчика в регистре `dl` уже установлен номер загрузочного диска. При желании его можно поменять. Прерывание `0x13` — сервис работы с дисками, `ah`, равное `0`, означает сброс диска.

Также, некоторые прерывания BIOS поддерживают код возврата (0 или 1), который хранится в Carry флаге. Поэтому можно добавить обработку ошибок.

Загрузим следующие сектора до конца программы:

```
    mov ah, 0x02    ; Read Sectors
    mov al, (end - $$ + 511) / 512 - 1 ; Number of sectors to read
    mov ch, 0       ; Cylinder number
    mov cl, 2       ; Sector number
    mov dh, 0       ; Head number
    mov bx, 0x7E00  ; Buffer address in memory (where to load)
    int 0x13        ; Low Level Disk Services
```

Прерывание BIOS `0x13` с `ah`, равным `0x02`, отвечает за чтение секторов с диска. Что удобно — можно прочитать сразу несколько секторов (`al`), что неудобно — нужно указывать Cylinder number, Sector number и Head number, которые непонятно что означают для обычной USB флешки. Этот кусок кода загружает сектора, начиная со второго по адресу `0x7E00`, который следует сразу за первой стадией.

Это прерывание тоже поддерживает код возврата, так что ошибку можно обработать:

```
    mov si, disk_read_error_message
    jc panic
```

Загрузив сектора в оперативную память, прыгаем на вторую стадию:

```
    jmp second_stage
```

Добавим в самый конец файла сообщение и код, который его выведет:

```
second_stage:       ; Second stage
    mov si, message
    call println
    jmp $           ; Hang

message:
    db 'Hello world!', 0
```

Вывод полностью совпадает с обычным Hello world.

Переход в unreal mode

Источники: [11], [12], [13], [14].

При включении компьютера, процессор находится в реальном режиме (real mode), особенностями которого являются 16-битный набор инструкций и прямой доступ (без трансляции виртуальных адресов в физические) к 1 мегабайту оперативной памяти. Для некоторых алгоритмов, которые будут использоваться для тестирования, может оказаться недостаточно такого объёма памяти (не говоря уже о том, что вся память в таком случае поместится в кэш 3 уровня, а большая часть и в кэш второго уровня, что сделает исполнение программы нереалистичным).

Чтобы запустить более реалистичную программу, которая будет иметь доступ к 2^{32} байтам, то есть 4 гигабайтам, но при этом сохранить прямой доступ к памяти, можно воспользоваться техникой, которая называется нереальным режимом (unreal mode).

Для начала, мы войдём в защищённый 32-битный режим, переключив специальный флаг. В 32-битном режиме значение регистров сегментов немного изменено: теперь они хранят смещение дескриптора сегмента относительно начала глобальной (или локальной) таблицы дескрипторов сегментов (Global Descriptor Table, GDT), указатель на которую можно загрузить с помощью инструкции `lgdt`. Такое устройство позволяет разделять память на сегменты с любыми началом и концом, а также флагами доступа. Когда мы обновим значения регистров сегментов, процессор сохранит их значения в специальном кэше и автоматически перейдёт в 32-битный режим. При этом, после возвращения в реальный режим значения сегментных регистров останутся 32-битными и мы получим возможность обращаться к памяти по адресам от 1 мегабайта до 4 гигабайт.

Напишем отдельный загрузчик для нереального режима, который можно будет потом использовать без изменений в наших программах.

Для начала, нам нужно создать GDT. Для этого вставим в программу следующий код:

```
; GDT (Global Descriptor Table)
align 8
gdt_start:
    ; Null descriptor
    dd 0x00000000, 0x00000000
gdt_code:
    ; Code segment descriptor
    dw 0xFFFF       ; Limit 0:15
    dw 0x0000       ; Base 0:15
    db 0x00         ; Base 16:23
     db 0x9A          ; Access byte: 7 - present, 4 - not system, 3 - executable, 1
- readable
    db 0xCF         ; Flag (page granularity and 32-bit mode) and limit 16:19
    db 0x00         ; Base 24:31
gdt_data:
    ; Data segment descriptor
    dw 0xFFFF       ; Limit 0:15
    dw 0x0000       ; Base 0:15
    db 0x00         ; Base 16:23
    db 0x92         ; Access byte: 7 - present, 4 - not system, 3 - not executable, 1
- writable
    db 0xCF         ; Flag (page granularity and 32-bit mode) and limit 16:19
    db 0x00         ; Base 24:31
gdt_end:

gdt_descriptor:
    dw gdt_end - gdt_start - 1 ; Limit of GDT
    dd gdt_start    ; Base address of GDT
```

Здесь `align 8` выравнивает код следующей инструкции, чтобы его адрес делился на 8. Производители процессоров рекомендуют делать это выравнивание, чтобы таблица дескрипторов лучше помещалась в кэш.

Первые 8 байт таблицы дескрипторов — null descriptor. По какой-то причине он должен состоять из 8 нулевых байт. Далее размещаем свои дескрипторы. Так как мы полностью контролируем исполнение, нам будет удобно делать дескрипторы сегментов, заполняющих всё адресное пространство.

Вторые 8 байт таблицы дескрипторов — дескриптор кодового сегмента. Он отличается от дескриптора сегмента данных флагом, который позволяет исполнять код, записанный в нём. Дескриптор содержит 32 бита начала сегмента и 20 бит максимального смещения относительно начала. Так как нам надо 4 гигабайта памяти, нужно установить в 1 первый бит предпоследнего байта, чтобы включить страничную гранулярность и умножить смещение на 4096. Второй бит этого байта отвечает за битность: если он 0, то дескриптор 16-битный, иначе — 32-битный. У 16-битного режима есть преимущество: в нём мы можем вызвать прерывания BIOS без дополнительных действий. Но в 32-битном режиме доступны более эффективные инструкции, так что выберем его. Также есть ещё один байт с флагами: access byte. Первый бит означает, что сегмент валиден, четвёртый — что сегмент не системный, пятый позволяет исполнять код, а седьмой в контексте исполняемого сегмента позволяет читать содержимое сегмента. Биты базы установим в 0, а лимита — в 1.

Третьи 8 байт таблицы дескрипторов — дескриптор сегмента данных. В нём, в отличие от дескриптора кодового сегмента должен не быть установлен бит исполнения. Тогда седьмой бит откроет сегмент для записи.

Также нам нужен дескриптор для самой таблицы, который состоит из её максимального смещения и начала.

После первой фазы загрузчика, которая описывалась в предыдущем разделе, загрузим таблицу дескрипторов:

```
    lgdt [gdt_descriptor] ; Load GDT (Global Descriptor Table)
```

Далее нужно выключить прерывания с помощью инструкции `cli`. Это необходимо сделать, если мы собираемся перейти в 32-битный режим, так как встроенные в BIOS обработчики прерываний его не поддерживают.

Включаем защищённый режим:

```
    mov eax, cr0
    or eax, 1
    mov cr0, eax    ; Enable protected mode
```

И начинаем загружать дескрипторы с кодового сегмента:

```
    jmp word (gdt_code - gdt_start):unreal_start ; Far jump to flush prefetch queue
```

Такой `jmp` называется far jump и позволяет загрузить значение до двоеточия в регистр кодового сегмента и сразу перейти к адресу после двоеточия.

Далее выбираем 32-битный режим и загружаем другие регистры сегментов:

```
[bits 32]           ; Use 32-bit registers and instructions
unreal_start:
    mov ax, (gdt_data - gdt_start)
    mov ds, ax
    mov es, ax
    mov fs, ax
    mov gs, ax
    mov ss, ax
```

После этого можно вернуться в реальный режим, который теперь будет позволять исполнять 32-битные инструкции и обращаться к адресам до 4 гигабайт:

```
    mov eax, cr0
    and eax, 0xFFFFFFFE
    mov cr0, eax    ; Return to real mode
```

Очень важно, что после far jump мы оказались в 32-битном режиме и не можем использовать код для 16-битного, то есть весь предыдущий код, а также прерывания BIOS. Поэтому я решил вынести загрузчик в отдельный файл, чтобы его отдельно компилировать, а потом соединять с основным файлом, в котором будут функции, например, для печати, для 32-битного режима.

Также важно правильно инициализировать стек: теперь мы можем сделать длинную программу и нужно проследить, чтобы стек не перекрывал ни программный код, ни память, которую мы будем использовать для алгоритмов.

Если не выполнить эти два условия, можно нарушить нормальное исполнение программы, получив ошибку, которую трудно будет исправить.

Инициализируем стек и прыгаем в конец файла — после `0xAA55` — к следующему файлу:

```
    mov esp, 0x800000; Initialize stack
    jmp done
```

Hello world в нереальном режиме

Источники: [15].

Будем начинать каждый файл с этих строк:

```
org 0x7e00          ; Start address
[bits 32]           ; Use 32-bit registers and instructions
```

Так как с `0x7c00` по `0x7e00` был загрузчик, программа начинается с `0x7e00`.

Так как теперь мы не можем использовать прерывания BIOS, придётся придумывать новый способ печати. Для этого можно просто записать нужные значения по адресу видеопамяти: `0xb8000`. Выделим место для хранения курсора: заполним 4 байта нулями.

```
cursor:
    dd 0
```

Вообще, правильнее было бы создать секцию `.bss` и выделять память в ней, но мне было удобнее сделать это прямо в коде программы.

Сделаем функцию очистки экрана, которая заполняет нулями всю видеопамять с адреса `0xb8000` до `0xb8000 + 25 * 80 * 2` (25 строк по 80 символов по 2 байта).

Для того, чтобы вывести символ, достаточно скопировать его код в первый байт, соответствующий позиции, и, опционально, его цвет во второй байт. Например, `0x07` — серый символ на чёрном фоне. Отдельно стоит обработать символы перевода строки и возврата каретки, так как не печатаются в обычном смысле, но изменяют позицию курсора. После печати строки сохраняем курсор для дальнейшего использования.

В самой программе копируем сообщение на большой адрес `0x200000`:

```
    mov esi, message
    mov edi, 0x200000
    mov ecx, (end_message - message)
    rep movsb       ; Move message to large address
```

Здесь `rep` повторяет определённую инструкцию количество раз, специфицированное . В данном случае, копирует символ из адреса `esi` в адрес `edi`.

Теперь очищаем экран и выводим сообщение с этого адреса:

```
    call clear_screen
    mov esi, 0x200000
    call println
    jmp $           ; Hang
```

Для запуска программы компилируем загрузчик:

```
nasm -Werror unreal_bootloader.asm -f bin -o unreal_bootloader.bin
```

Компилируем программу:

```
nasm -Werror unreal_hello_world.asm -f bin -o unreal_hello_world_raw.bin
```

Конкатенируем машинные коды из этих двух файлов:

```
cat unreal_bootloader.bin unreal_hello_world_raw.bin > unreal_hello_world.bin
```

Удаляем артефакты сборки:

```
rm unreal_hello_world_raw.bin
```

Запускаем с помощью qemu:

```
qemu-system-x86_64 -drive format=raw,file=unreal_hello_world.bin
```

Вывод программы:

```
Hello world from 0x200000!
```

Это значит, что адреса выше 1 мегабайта работают на чтение и на запись.

Тестирование RDTSC

Источники: [16].

Чтобы начать создавать тесты, нужно написать и протестировать ещё две вещи: вывод числа и замер времени. Для этого я написал программу которая в цикле выводит время, прошедшее с включения компьютера.

Чтобы вывести число, я завёл буфер на 8 цифр, и сделал функцию, которая в него записывает шестнадцатеричное представление `eax`.

Чтобы получать время, я использовал инструкцию `rdtsc`, которая записывает в регистры и количество циклов процессора, прошедших с момента включения компьютера. Мне кажется, это самый точный способ измерить время.

Вывод программы:

`xxxxxxxxxxxxxxxx tacts of processor since reset`, где `x` постоянно увеличиваются.

Memory access test

Источники: [17], [18], [16], [19].

Я хочу измерить время работы реального алгоритма — merge sort — и просто случайного доступа к памяти в нереальном и в 32-битном защищённом режимах.

32-битный защищённый режим — режим процессора, который исполняет 32-битный набор инструкций с обращением к 4 гигабайтам памяти через таблицу страниц, состоящую из двух уровней: page directory и page table.

Для того, чтобы начать сам тест, нужно сначала инициализировать таблицу страниц. Для этого выделяем место под page directory и заполняем её алгебраической прогрессией с началом `адрес page directory + 4096 + 3` и шагом `4096`, где `4096` — размер страницы. Каждая запись в page directory или page table — page table entry (PTE) — адрес следующего уровня или самой страницы без нижних 12 бит и некоторые флаги. Таким образом, на данный момент в page directory каждая запись указывает на одну из 1024 следующих за ней страниц с флагами записи (второй бит с конца) и валидности (последний бит). Теперь нужно следующие 1024 страницы заполнить алгебраической прогрессией с началом 3 и шагом 4096. Таким образом, получается, что каждый виртуальный адрес транслируется в такой же физический и всего адресуемо $1024 * 1024 * 4096 = 2^{32}$ байт, то есть вся доступная память. После инициализации таблицы страниц нужно загрузить указатель на page table в регистр `cr3`.

В качестве генератора случайных чисел, чтобы заполнять неотсортированный массив или случайно обращаться к памяти, я использовал реализацию из glibc. Сначала я загружаю в `eax` сид генератора — 42 (чтобы программа работала детерминированно) — потом вызываю функцию `rand_r`, которая умножает его на `1103515245` и прибавляет `12345` по модулю 2^{31}. Эта функция выдаёт не очень случайные числа, но работает быстро, так что в самый раз.

В функции `test`, которая запускает тесты по очереди в обоих режимах, я сначала включаю нереальный режим и выключаю страничную адресацию, обнуляя первый и последний биты `cr0`. Потом запускаю тесты merge sort и random access, вывожу результаты. Дальше включаю

защищённый режим со страничной адресацией, устанавливая первый и последний биты `cr0`, и запускаю те же самые тесты в нём.

Merge sort заполняет массив в определённом участке памяти случайными числами, потом вызывает функцию `start_test`, которая очищает обычный кэш с помощью инструкции `wbinvd` и TLB-кэш с помощью перезаписи `cr3` и записывает время начала. Очищение кэшей нужно, чтобы результат измерений меньше зависел от внешних условий. Затем выполняет саму сортировку, которая написана достаточно просто, так как работает только с массивами длины степени двойки, сохраняет время конца и выполняет проверку корректности алгоритма: как упорядоченность массива, так и контрольную сумму. Таким образом, результаты измерений максимально точны, а алгоритм точно работает без ошибок.

Random access просто в цикле вычисляет случайный адрес массива и пишет в него случайное число. Эта функция слишком проста и бесполезна, чтобы проверять корректность её работы.

Запустив тесты на qemu, можно убедиться, что при реализации этой виртуальной машины не задумывались о реалистичном счётчике времени. Чтобы убедиться в этом ещё сильнее, я добавил опцию компиляции для тестирования каждого случая с включённым и выключенным кэшом. Чтобы отключить кэш, я устанавливаю второй старший бит `cr0`, а чтобы включить опять — обнуляю. Теперь тесты должны работать сильно дольше с отключённым кэшом, но в qemu выводится то же самое время. Поэтому стоит запустить на реальном компьютере.

Запуск загрузчика на настоящем компьютере

Источники: [20].

Чтобы загрузить загрузчик на USB флешку, нужно его скомпилировать — так же, как и для qemu. Теперь нужно просто подключить флешку и ввести команду в терминале:

```
sudo dd if=/path/to/bootloader.bin of=/path/to/drive
```

Где `/path/to/drive` — обычно `/dev/sda`.

Изначально я не знал, что бывают разные виды “BIOS”. Поэтому нашёл в интернете первый попавшийся гайд по написанию загрузчика — и он оказался для BIOS. Но потом я узнал, что на моём ноутбуке вместо BIOS стоит UEFI. Поэтому я решил протестировать загрузчик на старом компьютере с уже установленным BIOS версии 0804 2009 года на процессоре `AMD Phenom(tm) II X4 945 Processor` с частотой 3000MHz и оперативной памятью 4 гигабайта.

Но прежде, чем мы разберём результаты, я расскажу об ещё двух опциях.

64-битный режим

Источники: [21], [22], [19].

Так как обычно программы пишут для 64-битной архитектуры, я решил протестировать, как изменится для неё время работы программы.

Так как в 64-битном режиме (long mode) обязательна страничная адресация [21], мы можем запустить тесты только с ней.

Для начала инициализируем таблицу страниц. Тут нет ничего нового, кроме того, что теперь адрес занимает 8 байт, следовательно, PTE занимает 8 байт. Таким образом, на странице помещается в два раза меньше PTE (512). Я заметил, что использую только четверть 4 гигабайт, поэтому в два раза меньше PTE мне хватит, чтобы использовать только одну page directory. Другое нововведение длинного режима — 4 уровня таблицы страниц. Таким образом, мне нужно заполнить первый уровень (page map level 4) нулями, кроме первых 4 байт, где нужно

разместить адрес второго уровня + 3. Второй уровень (page directory pointer) — аналогично. А третий и четвёртый отличаются от 32-битного режима тем, что PTE находятся в нечётных четвёрках байт (Little Endian).

Чтобы перейти в 64-битный режим, нужно создать 64-битную таблицу дескрипторов. Она выглядит так же, как 32-битная, только limit каждого дескриптора, кроме нулевого, должен быть равен 0, а предпоследний байт — `0x20` — флаг 64-битного сегмента.

Установка этого бита в `cr4` позволяет использовать 64-битные PTE:

```
mov eax, 00100000b ; Enable Physical Address Extension
mov cr4, eax
```

Эти инструкции включают длинный режим:

```
mov ecx, 0xc0000080 ; Extended Feature Enable Register
rdmsr               ; Read from MSR
or eax, 0x00000100 ; Enable longmode
wrmsr               ; Write to MSR
```

Здесь `rdmsr` и `wrmsr` — прочитать и записать соответственно model-specific register, указанный в `ecx`, в/из `eax`.

На данный момент страничная адресация не включена, поэтому процессор находится в 32-битном режиме. Включаем бит защищённого режима и страничную адресацию:

```
mov eax, cr0
or eax, 0x80000001 ; Enable protected mode with page addressing
mov cr0, eax
```

Теперь процессор находится в 64-битном режиме, а это значит, что весь код, скомпилированный для 32-битного режима не может быть исполнен. Именно поэтому я вынес этот код в отдельный файл. Все последующие инструкции находятся под директивой `[bits 64]`.

Загружаем новую 64-битную таблицу дескрипторов. Так как мы в 64-битном режиме, far jump выполняется в 3 приёма: кладём в стек смещение дескриптора кодового сегмента и адрес следующей инструкции. Теперь выполняем far return — и мы в полноценном длинном режиме.

```
lgdt [gdt_descriptor] ; Load Global Descriptor Table
push (gdt_code - gdt_start) ; Segment
push start          ; Address
retf                ; Far return
```

Остальной код аналогичен 32-битному за исключением того, что некоторые инструкции требуют 64-битные регистры.

Использование PMC регистров для диагностики кэш-промахов

Источники: [23].

На процессоре, с которым я работал, AMD Phenom II, есть 4 Performance-Monitoring Counters регистра. Их можно настроить, чтобы они увеличивались на 1 каждый раз, когда происходит определённое событие, например, доступ к кэшу или кэш-промах.

Я сделал ещё одну опцию для своей программы: подсчёт доступов к DTLB кэшу и DTLB кэш-промахов. Для этого перед измерением времени начала нужно настроить эти регистры.

У процессора AMD Phenom II регистры MSR C0010000, C0010001, C0010002, C0010003 — Performance Event Select Register — регистры для настройки счётчиков. В них можно записать идентификатор события, флаги, специфичные для конкретного события, и просто флаги.

Например, `045h` — событие попадания в DTLB второго уровня, `046h` — кэш-промахи DTLB второго уровня, а `04Dh` — попадания в DTLB первого уровня.

Таким образом, в первый регистр пишем `0x00430745`, где `43` — флаги (os mode, user mode, enable), а `07` — флаги, специфичные для этого события — включаем все случаи:

```
mov ecx, 0xc0010000 ; First Performance Event Select Register
mov eax, 0x00430740 ; Data Cache Accesses
mov edx, 0x00000000 ; Data Cache Accesses
wrmsr
```

`edx` — старшие 32 бита регистра.

Во второй регистр пишем `0x00430746`, в третий — `0x0043074d`.

Далее неплохо сбросить счётчики. У AMD Phenom II — это Performance Event Counter Registers — MSR C0010004, C0010005, C0010006, C0010007, которые соответствуют регистрам на 4 меньше. Для этого просто записываем в них нули.

После замера времени конца теста читаем эти регистры с помощью `rdmsr` и выводим их значения.

Если теперь попробовать запустить на qemu, станет понятно, что поддержки этих регистров на qemu нет: прочитанные значения — нули.

Проведение измерений и обработка результатов

Источники: [5].

Для тестирования на компьютере, я сделал загрузчик с возможностью выбора одного из четырёх вариантов программы. Во время тестирования выяснилось, что qemu работает медленнее, чем нативное исполнение, как и ожидалось, поэтому при подборе длины массива учитывается не только опция тестирования с выключенным кэшом, но и новая опция запуска в qemu, чтобы ожидание вывода не было таким утомительным.

Немного обработанные результаты тестирования на больших объёмах данных (массив из 67′108′864 элементов):

```
Merge sort:    0000000588B8AC05 tacts in unreal mode
DTLB1/DTLB2/Memory 0000000126426CCD/0000000000000786/0000000000340036
Random access: 00000004C0F6A2E6 tacts in unreal mode
DTLB1/DTLB2/Memory 00000000569EA5DD/00000000001FBE24/000000000FDFBCEF
Merge sort:    0000000590A736E8 tacts in protected mode with page addressing
DTLB1/DTLB2/Memory 00000001267E3D2F/0000000000000AF2/000000000033FA5A
Random access: 00000005B5321A7A tacts in protected mode with page addressing
DTLB1/DTLB2/Memory 00000000511623ED/00000000001D770B/000000000FDFB12B
Merge sort:    0000000569A8C9CC tacts in long mode with page addressing
DTLB1/DTLB2/Memory 0000000147E5BE05/0000000000000845/000000000033F955
Random access: 00000006DAFD5A19 tacts in long mode with page addressing
DTLB1/DTLB2/Memory 0000000050332869/00000000001D5F37/000000000FDFB200
```

Здесь время измерялось для точности без включённых счётчиков, а кэш-промахи — отдельно.

Увеличение времени работы для 32-битного защищённого режима:

```
Protected mode penalty:
Merge sort:    0.56%
Random access: 20%
```

Увеличение времени работы для длинного режима:

```
Long mode penalty:
Merge sort:    -2.2%
Random access: 44.2%
```

Можно заметить, что сортировка меньше зависит от страничной адресации. Такой эффект можно объяснить тем, что в сортировке мы последовательно обращаемся к памяти, поэтому MMU (memory management unit) чаще попадает в DTLB1 кэш, по сравнению с DTLB2 и памятью, что видно в результатах измерений. Хоть event и спекулятивный, то есть зависит от эвристик процессора, всё равно можно заметить разницу.

Результаты тестирования на небольших объёмах данных с кэшом и без кэша (массив из 4′194′304 элементов):

```
Enabled cache:
Merge sort:    000000004F8ED9B1 tacts in unreal mode
DTLB1/DTLB2/Memory 0000000011596E89/00000000000005CC/000000000002BAD2
Random access: 00000000356C5D79 tacts in unreal mode
DTLB1/DTLB2/Memory 0000000005D55AF6/00000000001E7509/0000000000DFC3F2
Merge sort:    000000004FF295BE tacts in protected mode with page addressing
DTLB1/DTLB2/Memory 00000000115C6BC6/0000000000000806/000000000002B887
Random access: 000000003B631221 tacts in protected mode with page addressing
DTLB1/DTLB2/Memory 0000000005A11B44/00000000001DE6E5/0000000000DFBFBB
Merge sort:    000000005054E7DC tacts in long mode with page addressing
DTLB1/DTLB2/Memory 00000000106A3EBC/000000000000060D/000000000002BA34
Random access: 000000003D91DBC4 tacts in long mode with page addressing
DTLB1/DTLB2/Memory 0000000005495B32/00000000001D8605/0000000000DFBB04
Disabled cache:
Merge sort:    000000163072067D tacts in unreal mode
DTLB1/DTLB2/Memory 000000000EFADEA3/0000000000000766/000000000002B8B3
Random access: 0000000AC10E4B6F tacts in unreal mode
DTLB1/DTLB2/Memory 00000000119BF91E/000000000007A854/0000000002F5492B
Merge sort:    0000001630AF91D4 tacts in protected mode with page addressing
DTLB1/DTLB2/Memory 000000000EFA68B5/0000000000000783/000000000002B889
Random access: 0000000ADF34B1F7 tacts in protected mode with page addressing
DTLB1/DTLB2/Memory 000000000CD04E2D/00000000001D8E20/0000000000E195A7
Merge sort:    000000162C7A48C8 tacts in long mode with page addressing
DTLB1/DTLB2/Memory 00000000137D9ABB/0000000000000768/000000000002B89F
Random access: 00000008CB2748DE tacts in long mode with page addressing
DTLB1/DTLB2/Memory 00000000073578AC/00000000001D4DF8/0000000000E1D6CC
```

Увеличение времени работы для 32-битного защищённого режима с включённым и выключенным кэшом:

```
With enabled cache protected mode penalty:
Merge sort:    0.5%
Random access: 11%
With disabled cache protected mode penalty:
Merge sort:    0%
Random access: 1%
```

Увеличение времени работы для длинного режима с включённым и выключенным кэшом:

```
With enabled cache long mode penalty:
Merge sort:    1%
Random access: 15%
With disabled cache long mode penalty:
Merge sort:    -0.07%
Random access: -18.2%
```

Опять же, можно заметить, что с включённым кэшом время работы сортировки меньше зависит от страничной адресации. Опять же из-за TLB-кэша.

Также random access увеличился не так сильно, потому что разброс адресов оказался не таким большим и много адресов попало в DTLB1 и DTLB2 кэши.

Поведение защищённого режима с отключённым кэшом было тоже предсказуемо: время, затраченное на преобразование адреса не так велико по сравнению с временем доступа к памяти.

Гораздо сложнее объяснить, почему random access в длинном режиме работает быстрее, даже чем в нереальном. Скорее всего, это связано с более оптимальной работой процессора в длинном режиме. У меня есть предположение, откуда мог быть настолько сильный прирост производительности: возможно, процессор догадался, что случайные записи в память чаще всего независимы и исполняет их параллельно.

Прирост производительности merge sort не настолько большой. Он, скорее всего связан с тем, что в длинном режиме решили отказаться от сегментов. Конечно, нам пришлось загрузить глобальную таблицу десрипторов, но она была нужна только для того, чтобы процессор перешёл в 64-битный режим и по факту не используется. Это могло незначительно повлиять на производительность длинного режима.

Выводы

Как можно заметить, для стандартных алгоритмов, которые последовательно обращаются к памяти, оверхед страничной адресации достаточно мал. Но если запустить искусственный алгоритм, который написан не очень эффективно, страничная адресация, скорее всего будет сильно влиять на время исполнения. Хотя более современные режимы содержат оптимизации, которые могут даже ускорить неэффективный алгоритм.

На самом деле разработчики процессоров позаботились о том, чтобы увеличение времени исполнения при страничной адресации не чувствовалось. Так, TLB кэш позволяет искать нужную страницу в разы эффективнее, а кэш первого уровня начинает поиск кэш-линии ещё до трансляции адресов — по виртуальному адресу [24].

Также в реальных программах неоценимы преимущества страничной адресации — те, о которых я писал в самом начале. С появлением 64-битного режима процессоры стали ещё производительнее: стало можно работать с 64-битными числами, появились специализированные инструкции, отменили сегменты.

Таким образом, уже в 32-битном режиме страничная адресация почти не делает разницы для обычных программ — а в 64-битном режиме они будут работать ещё быстрее.

Так что, со всеми оптимизациями, страничная адресация — очень полезная вещь, которую вряд ли захочется когда-нибудь отключить.

Bibliography

[1] “Virtual memory - Wikipedia,” [Online]. Available: https://en.wikipedia.org/wiki/Virtual_memory

[2] “BIOS - Wikipedia,” [Online]. Available: https://en.wikipedia.org/wiki/BIOS

[3] “UEFI - Wikipedia,” [Online]. Available: https://en.wikipedia.org/wiki/UEFI

[4] “hello world without an operating system - Youtube,” [Online]. Available: https://www.youtube.com/watch?v=bpa5H6tnYZA

[5] “X86 memory segmentation - Wikipedia,” [Online]. Available: https://en.wikipedia.org/wiki/X86_memory_segmentation

[6] “BIOS interrupt call - Wikipedia,” [Online]. Available: https://en.wikipedia.org/wiki/BIOS_interrupt_call

[7] “INT 10H - Wikipedia,” [Online]. Available: https://en.wikipedia.org/wiki/INT_10H

[8] “INT 16H - Wikipedia,” [Online]. Available: https://en.wikipedia.org/wiki/INT_16H

[9] “floppybird boot.asm by icebreaker - Github,” [Online]. Available: https://github.com/icebreaker/floppybird/blob/master/src/boot.asm

[10] “INT 13H - Wikipedia,” [Online]. Available: https://en.wikipedia.org/wiki/INT_13H

[11] “Real mode - Wikipedia,” [Online]. Available: https://en.wikipedia.org/wiki/Real_mode

[12] “Unreal mode - Wikipedia,” [Online]. Available: https://en.wikipedia.org/wiki/Unreal_mode

[13] “Unreal Mode in x86,” [Online]. Available: https://thejat.in/learn/unreal-mode-in-x86

[14] “What is Global Descriptor Table - GeeksforGeeks,” [Online]. Available: https://www.geeksforgeeks.org/what-is-global-descriptor-table/

[15] “address of video memory - StackOverflow,” [Online]. Available: https://stackoverflow.com/questions/17367618/address-of-video-memory

[16] “x86 and amd64 instruction reference,” [Online]. Available: https://www.felixcloutier.com/x86/

[17] “page_table.asm - Github,” [Online]. Available: https://github.com/HadeelMabrouk/Bootloader/blob/master/sources/includes/second_stage/page_table.asm

[18] “Linear congruential generator - Wikipedia,” [Online]. Available: https://en.wikipedia.org/wiki/Linear_congruential_generator

[19] “Control register - Wikipedia,” [Online]. Available: https://en.wikipedia.org/wiki/Control_register

[20] “how to burn a boot loader onto a usb drive boot loader as a bin - SuperUser,” [Online]. Available: https://superuser.com/questions/1645622/how-to-burn-a-boot-loader-onto-a-usb-drive-boot-loader-as-a-bin

[21] “is it possible to enter long mode without setting up paging - StackOverflow,” [Online]. Available: https://stackoverflow.com/questions/70609634/is-it-possible-to-enter-long-mode-without-setting-up-paging

[22] “triple fault when jumping to 64 bit longmode - StackOverflow,” [Online]. Available: https://stackoverflow.com/questions/43438363/triple-fault-when-jumping-to-64-bit-longmode

[23] “Phenom II documentation - AMD,” [Online]. Available: https://www.amd.com/content/dam/amd/en/documents/archived-tech-docs/programmer-references/31116.pdf

[24] “CPU cache - Wikipedia,” [Online]. Available: https://en.wikipedia.org/wiki/CPU_cache