**Лекция 7**

**Виды памяти**

Для программы в оперативной памяти выделяется отрезок, в котором будет находиться ее память. Он делится на три части: data (bss, data, rodata), text, stack.

**bss** — существует для статических переменных, не инициализированных в начале работы программы. То есть они определяются уже в рантайме, но место под них выделено заранее.

**data** — статическая память, хранит глобальные, статические переменные, литералы, таблицы виртуальных функций.

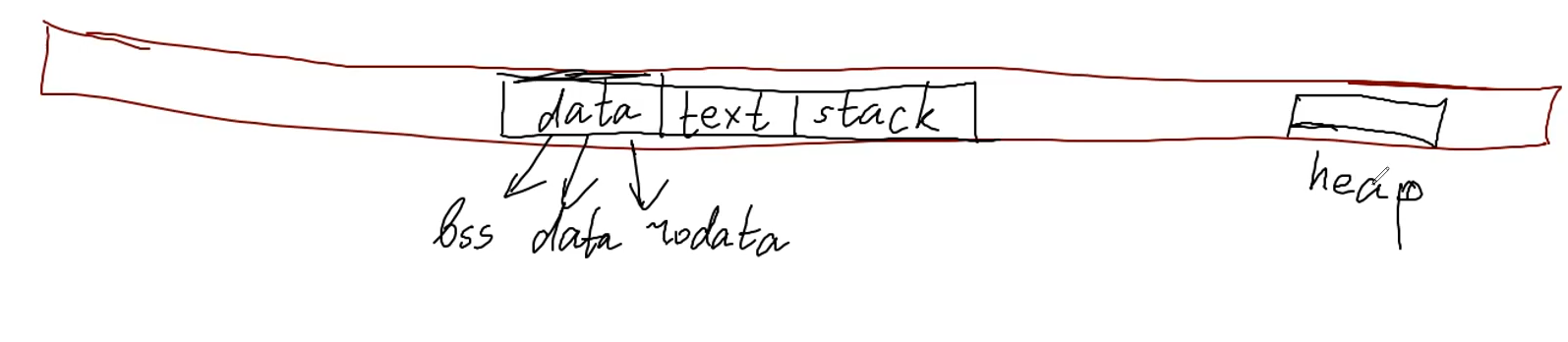
**rodata** — read only data.

**text** – в этой секции хранится бинарный код.

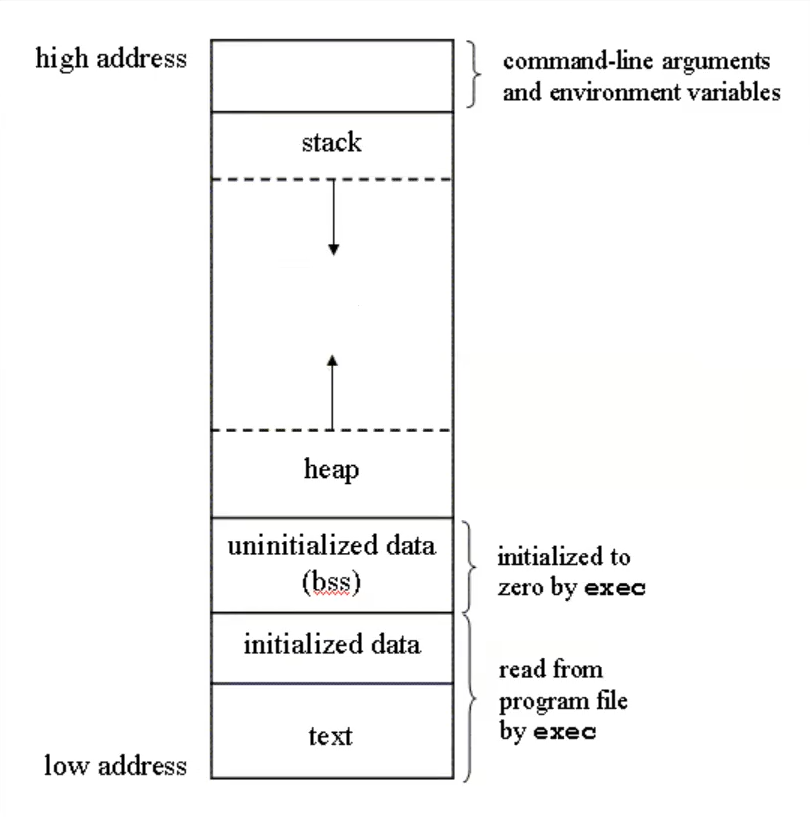
**stack** — автоматическая память, в ней хранятся локальные переменные и адреса возвратов.

**heap** — динамическая память.

Эти куски памяти, на самом деле, не выделяются непрерывно, и могут лежать в любом порядке относительно друг друга.



Операционная система создает для пользователя иллюзию, что адреса расположены в таком порядке и непрерывно.

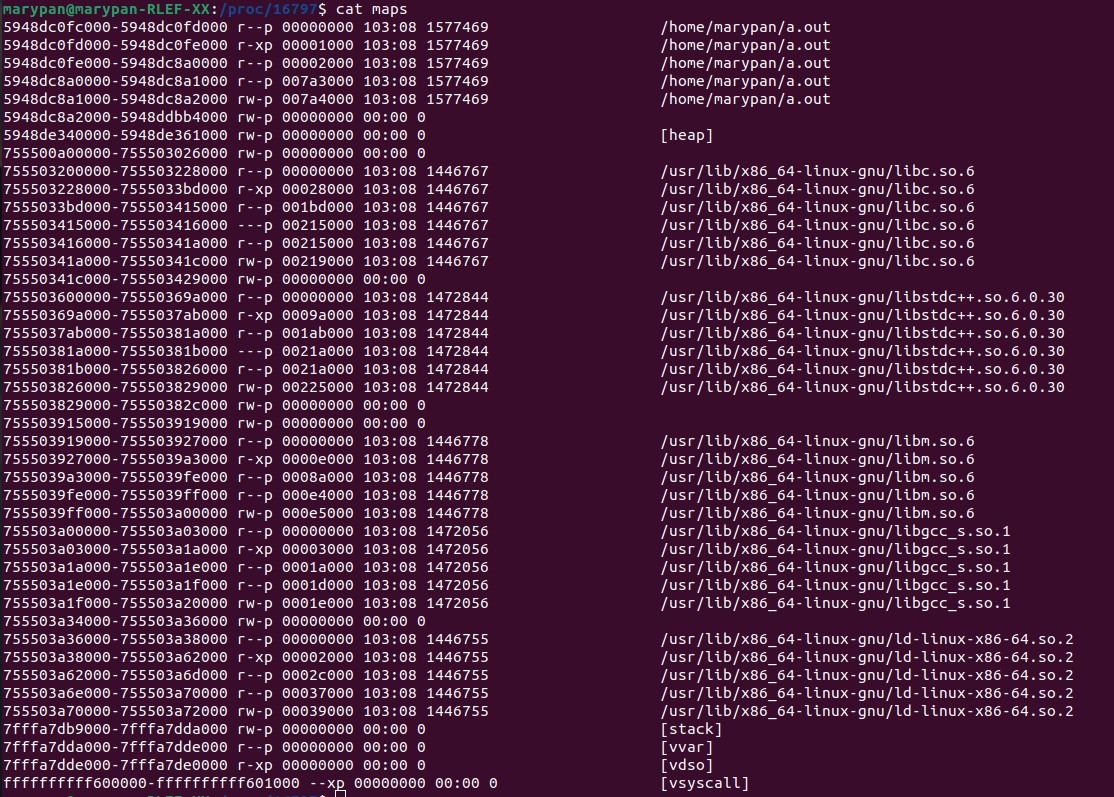


Адреса в программе устроены примерно таким образом: есть stack, который растет вниз от какого-то фиксированного адреса сверху. Когда кладем что-то на стек, адреса на самом деле уменьшаются. Heap же по мере выделения новых адресов растет вверх от некоторой границы адресов. Ниже будут расположены bss, data и text.

Продемонстрируем, как это работает:

#include <iostream>  
#include <vector>  
  
const char\* str = "A quick brown fox jumps over the lazy dog"; //rodata  
  
const int a[2'000'000]{}; //rodata  
  
void f() {} //text  
  
int main() {  
 static int b[5'000'000]; //bss  
 std::vector<int> v(10'000'000); //heap  
  
 int x = 0; //stack  
  
 std::cout << "str: " << (void\*)str << '\n';  
 std::cout << "global array: " << (void\*)a << '\n';  
 std::cout << "static local array: " << (void\*)b << '\n';  
 std::cout << "vector data: " << &v[0] << '\n';  
 std::cout << "stack variable: " << &x << '\n';  
 std::cout << "function: " << (void\*)&f << '\n';  
  
 getchar();  
}

str: 0x5a4b1ed31020  
global array: 0x5a4b1ed31060  
static local array: 0x5a4b1f4d4180  
vector data: 0x7dff98c00010  
stack variable: 0x7ffef5f2d14c  
function: 0x5a4b1ed302e9

Выведем таблицу всех адресов процесса

**pgrep $a.out** — узнать id процесса

**cd /proc/id/** - здесь содержится информация о процессе, в том числе файл maps, который содержит информацию о диапазонах памяти, которые выделены данному процессу и что им соответствует.

**cat /proc/$(pgrep a.out)/maps**

Каждому диапазону соответствуют определенные права. Если по какому-то адресу диапазона попытаться сделать что-то, что запрещено для него, будет segfault.

**Пример segfault:**

#include <iostream>  
void f() {std::cout << 1;} //text  
  
int main() {  
  
 int x = 0; //stack  
  
 void (\*p)() = &f;  
 p();  
  
 p = (void(\*)()) &x;  
 p();  
}

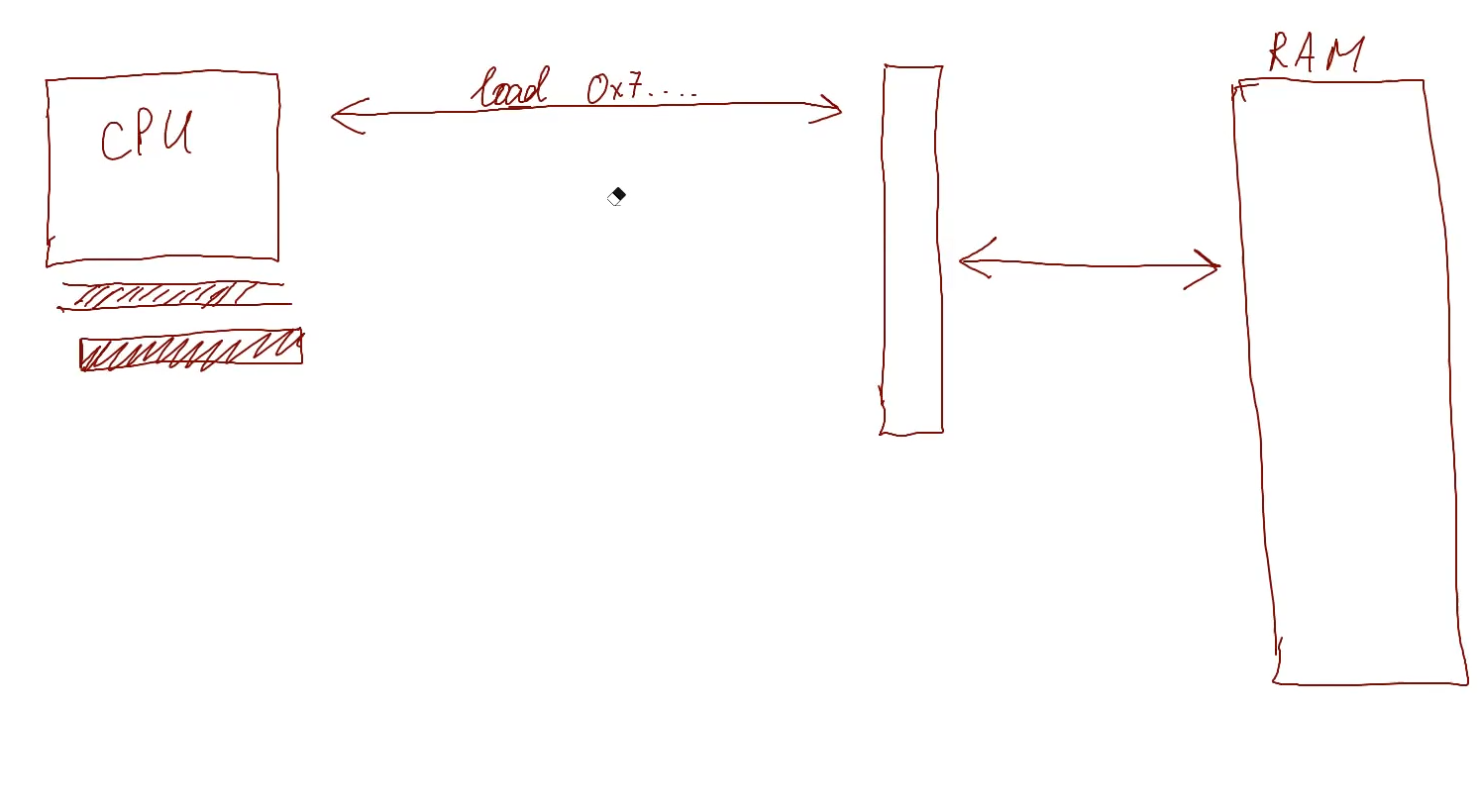
Все эти наблюдаемые нами адреса — не настоящие, это виртуальная память. Но существует некоторое отображение из виртуальных адресов в реальные.

**Что происходит, когда мы обращаемся к какому-то адресу по какому-то указателю?**

Есть центральный процессор. У него есть поток инструкций, который он выполняет. В какой-то момент он встречает инструкцию: разыменование указателя.

У процессора есть некоторое устройство, которое позволяет ему коммуницировать с оперативной памятью — **шина данных**. Он посылает на него запрос: load 0x7...(какой-то адрес). Эта шина может передать такой запрос в RAM, а потом результат передать обратно процессору.

У процессора есть кэш, куда он записывает полученную информацию, чтобы быстро ее доставать оттуда при дальнейших запросах.



В этой схеме есть промежуточное звено, которое называется **memory management unit**.

Это микросхема, являющееся частью процессора. MMU — это устройство, которое хранит кэш того, что недавно запросили из оперативной памяти. Самая важная функция, которую он выполняет, это трансляция виртуальных адресов в настоящие физические адреса. Он как раз содержит отображение из виртуальных адресов в настоящие. Как оно там появилось? ОС для каждого запускаемого процесса создает отображение и частично загружает его в MMU, при надобности дополнительно подгружая туда части этого отображения.

Когда от процессора в MMU идет запрос load address, он смотрит, есть ли у него информация о том, какому физическому адресу соответствует этот виртуальный адрес. Если есть, то он это число, полученное согласно отображению, и передает в оперативную память. Если нет, то MMU просит ОС либо подгрузить соответствующее число, либо случается segfault.

**Как ОС поддерживает эти отображения, где она их хранит?**

У ОС есть определенная область оперативной памяти, которой она пользуется для себя, и там хранятся эти отображения в виде деревьев. Но хранить информацию про каждый байт памяти, чему он соответствует, было бы слишком расточительно по памяти, поэтому память поделена на некоторые участки, которые называются страницами. Обычно они имеют размер 4 килобайта. Вся оперативная память, которая не используется непосредственно ОС, и которая доступна процессам, поделена на страницы (**pages**). Поэтому в маппингах достаточно хранить информацию для каждой страницы.

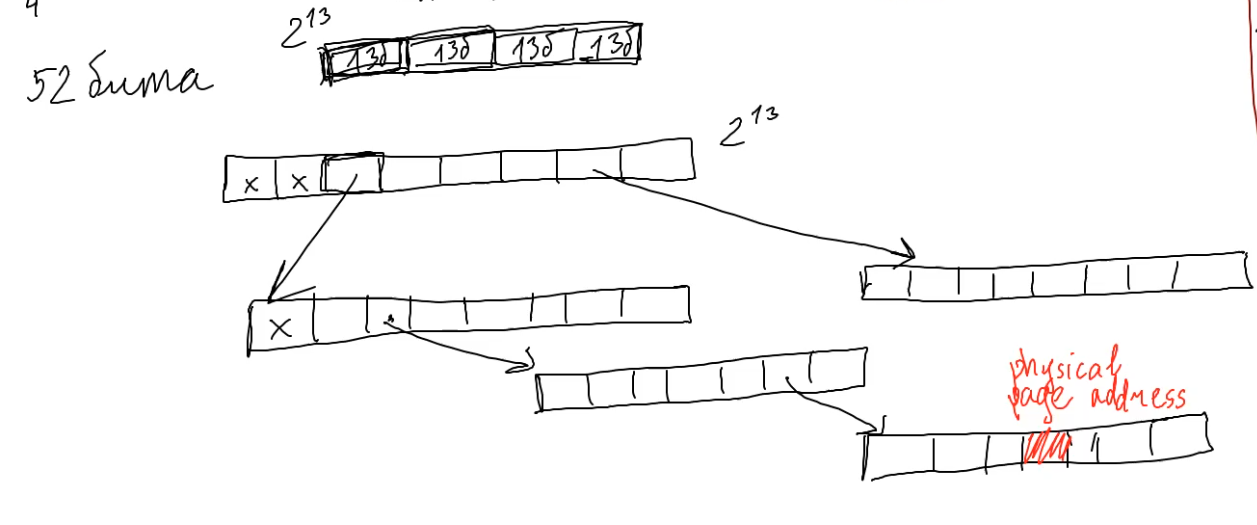
**Как устроена структура maps, хранящаяся в оперативной памяти?**

Она хранится в виде B-дерева.

Номера страниц кодируются 52 битами (так как адреса 64-х битные, а последние три цифры адреса это нули). Адрес страницы разбивается на 4 куска по 13 бит.

Есть массив размера 213 из ячеек, в каждой из которых написано либо nullptr (означает, что таких виртуальных адресов нет), либо указатель на следующую вершину дерева.

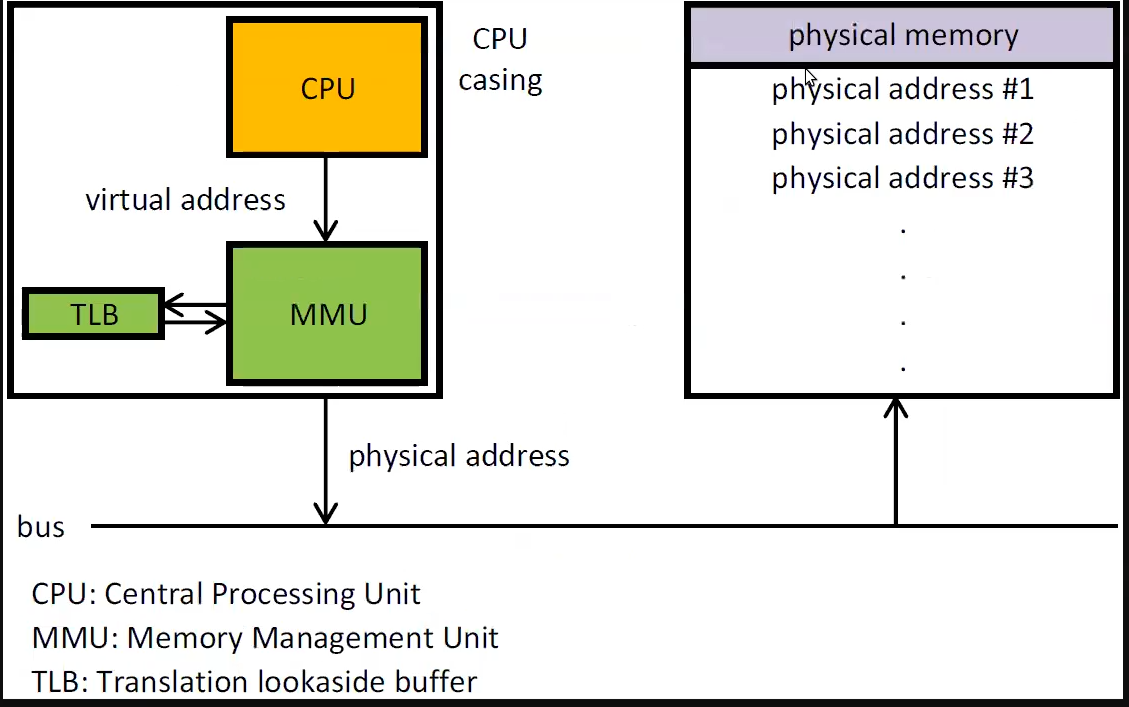
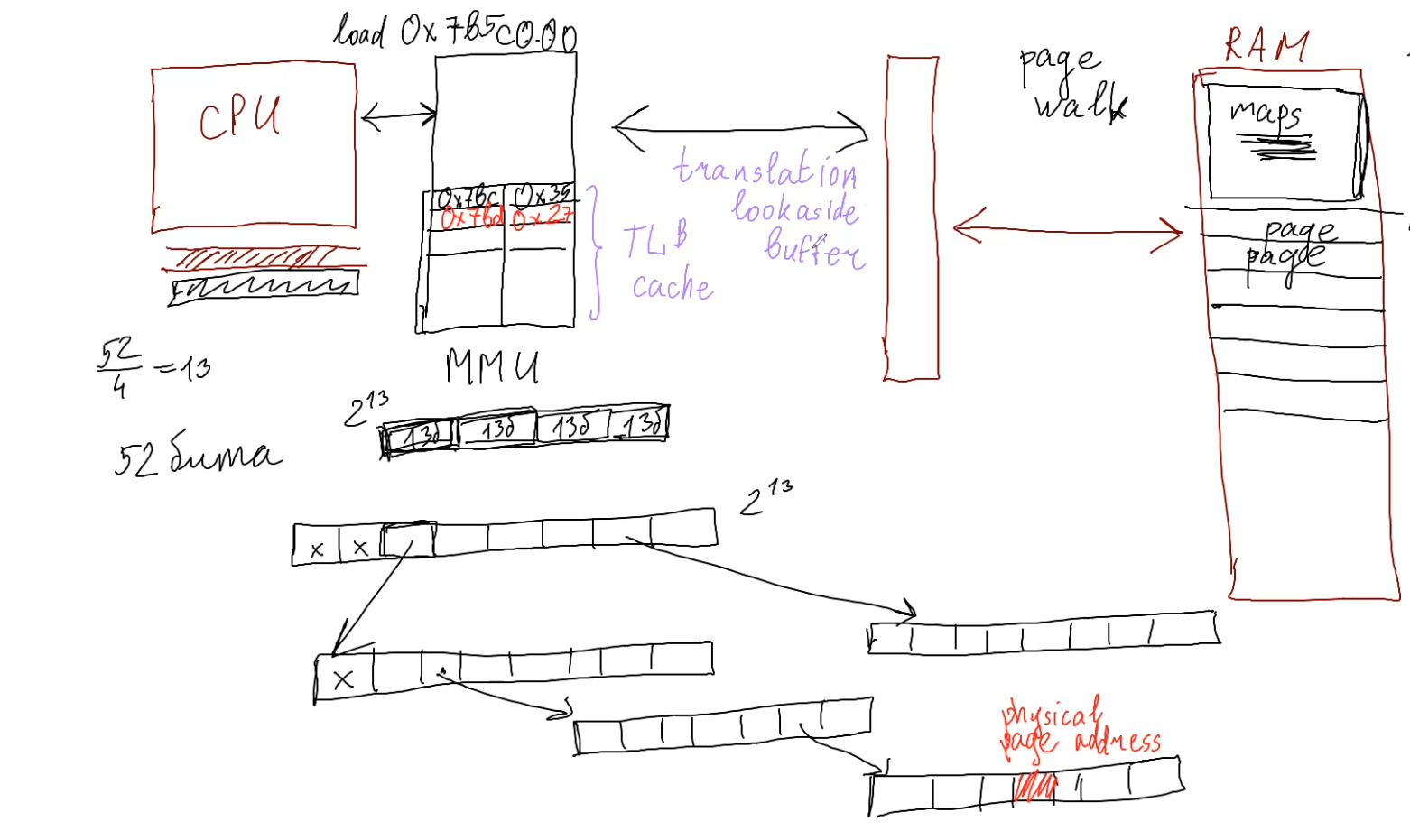
Благодаря такому хранению maps занимает не так много памяти.

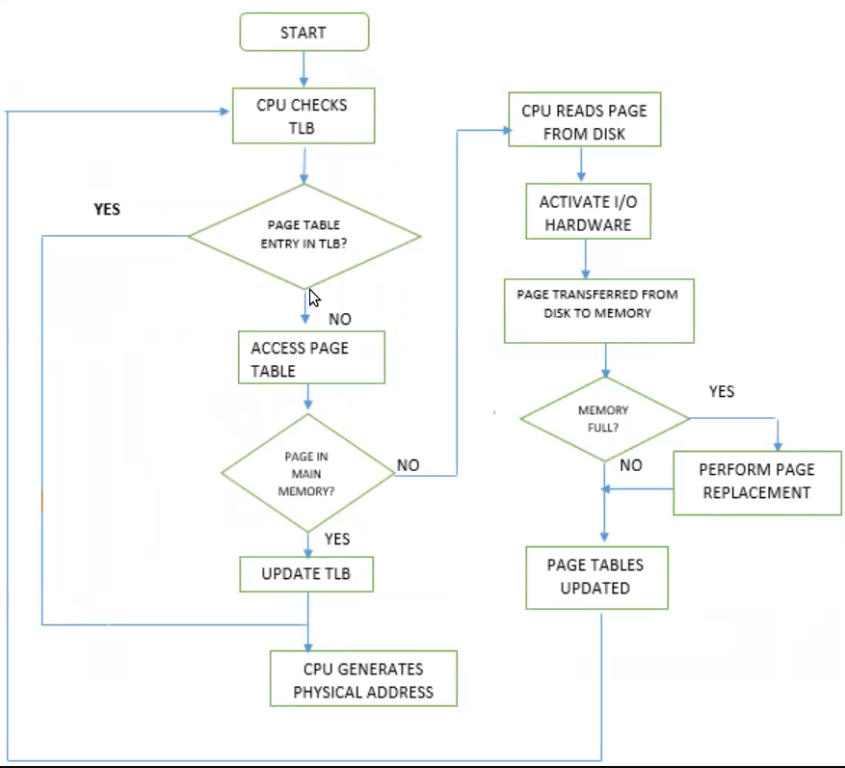


Тогда MMU, на самом деле, оперирует страницами. У него есть своя память, в которой он хранит информацию о том, какие виртуальные адреса он уже запрашивал и что им соответствует. Хранит в виде хэш-таблицы, которая называется **TLB cache**. (TLB — translation lookaside buffer). В случае переполнения таблицы старые записи перезатираются.

Когда к нему приходит запрос адреса, он смотрит в этой хэш-таблице, нет ли в ней страницы, куда попадает этот адрес. Если есть, то MMU уже по настоящему физическому адресу обращается через шину к оперативной памяти, и передает результат обратно процессору, который в свою очередь записывает его к себе в кэш.

Если же виртуальный адрес не попадает ни в одну страницу, которую хранит MMU, тогда MMU должен совершить «**page walk**». Он должен зайти в RAM с запросом: найди такой-то адрес в дереве страниц. И делаем поиск по этому B-дереву. Когда physical page была найдена, MMU записывает себе в таблицу результат поиска.





Что происходит, если в дереве мы не находим нужную страницу? Возникает ситуация, которая называется **pagefault**. Возможно одно из двух: либо этой страницы там нет, потому что ОС для нее еще не замапила физическую страницу (хотя должна была), либо действительно обратились туда, куда нам нельзя. В случае pagefault в дело вступает ядро ОС: процесс прерывается, исполняется код ядра ОС. ОС либо выдаст физическую страницу и обновит структуру maps, либо, если мы запросили что-то, чего нам ОС не обещала, случится segfault.

У каждой страницы также есть свои права, с которыми они доступны, и их также хранят MMU и maps.

**Зачем вообще нужна виртуальная память?** Почему сразу нельзя обращаться к физической памяти? Во-первых, чтобы изолировать процессы друг от друга, и невозможно было бы залезть в чужое пространство и испортить там что-либо. Во-вторых, память может быть выделена фрагментированно, потому что неэффективно выдавать каждый раз непрерывные куски памяти, а виртуальная память как раз помогает создавать иллюзию непрерывности памяти.

**Примечания:**

perf - утилита для просмотра того, что происходит во время исполнения программы. У неё можно смотреть sudo perf stat -e dtlb\_load\_misses.miss\_causes\_a\_walk ./a.out.

Как узнать размер tlb кэша? cat /proc/cpuinfo. Есть также утилита cpuid.

Есть itlb (instruction, когда процессор просит считать следующую инструкцию) и dtlb (data).

В htop можно смотреть RES (реальная память) и VIRT (виртуальная память) для процессов. Что ОС делать, если она обещала память, но памяти стало не хватать? Есть файл-подкачки (swapfile). Весит он довольно много. ОС засовывает спящие процессы на жесткий диск в swapfile, а освободившуюся память выдадим нуждающемуся процессу.

Ситуация, когда надо делать свопинг называется major pagefault. Все остальные pagefault'ы - minor. Major pagefault - очень долгая операция.

* load-miss - не нашли адрес в кеше.
* page-fault - не нашли адрес в структуре page table. Вызываем ОС.

**Ссылки:**

https://habr.com/ru/articles/211150/

https://en.wikipedia.org/wiki/Memory\_management\_unit

https://en.wikipedia.org/wiki/Translation\_lookaside\_buffer

https://en.wikipedia.org/wiki/Page\_fault

https://en.wikipedia.org/wiki/Page\_table