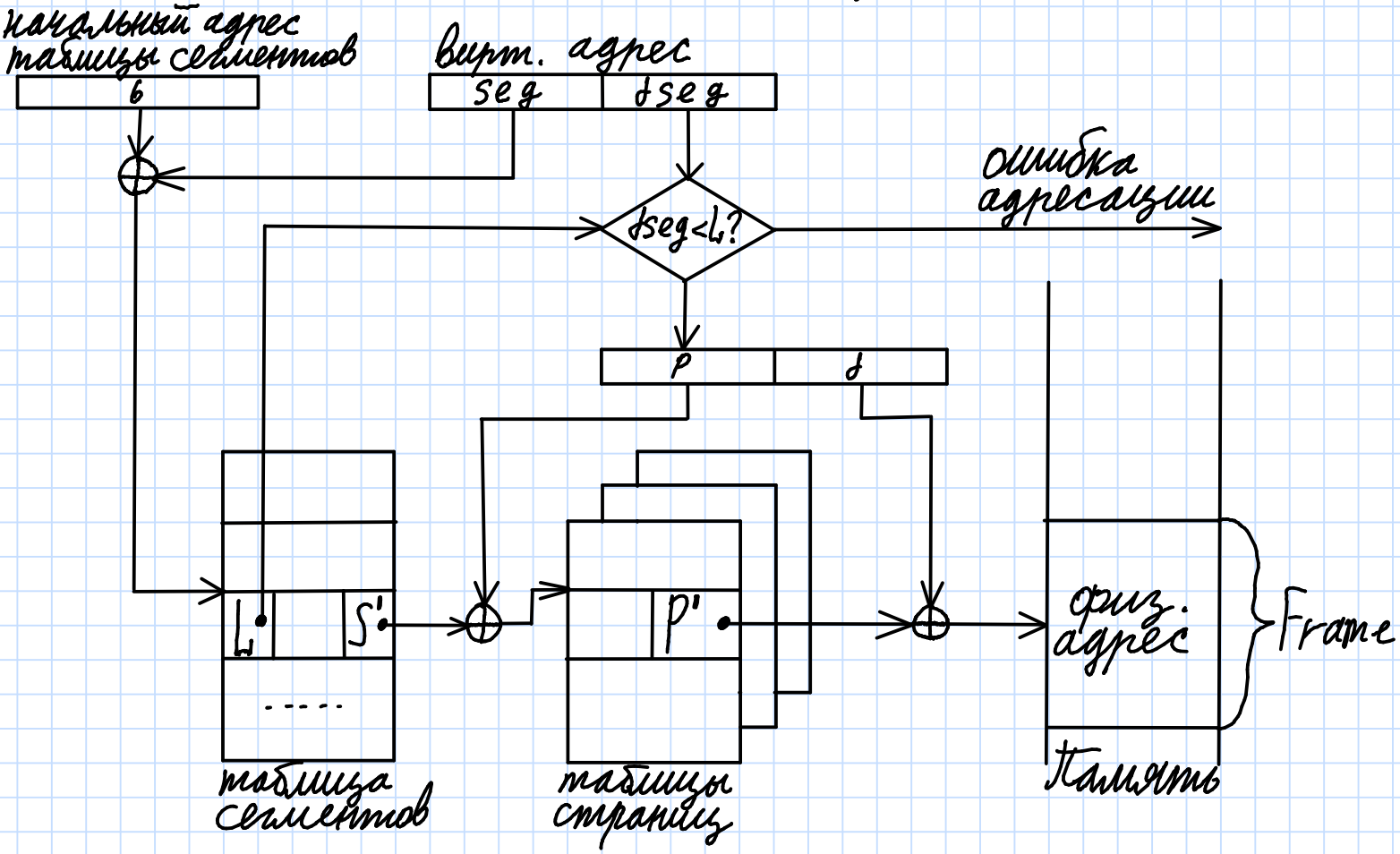
Сегментно-страничное преобразование



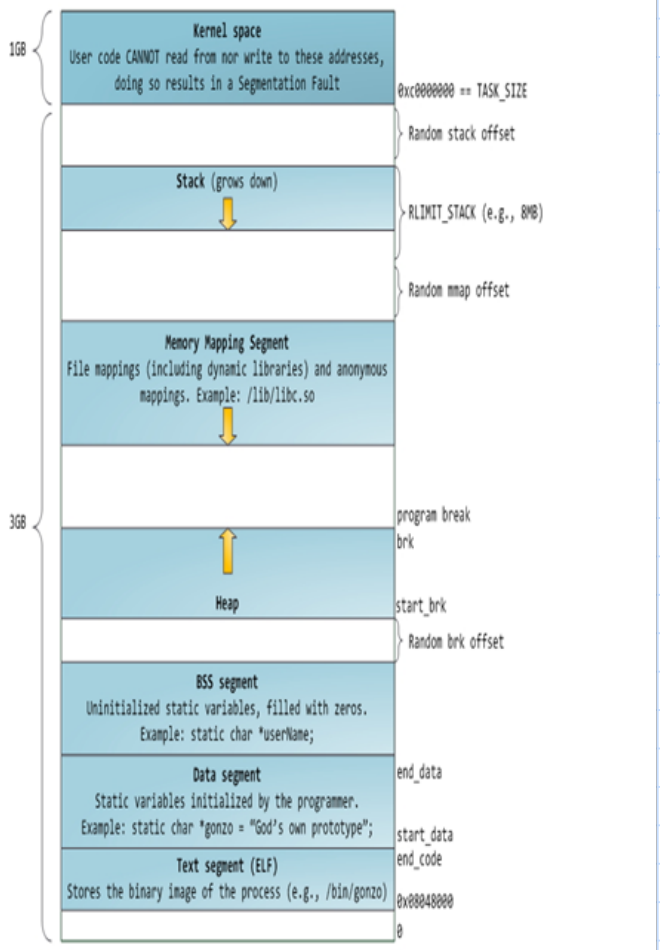
?Рассм. преобр-е PAE в режиме Long. (см. далее)

Здесь идея вирт. памяти: в памяти находятся только сегменты, к которым обращается процессор, и возникает задача перемещения сегментов (разделов);

По схеме: есть таблица сегментов, каждый из которых делится на страницы, т.е. будет много таблиц страниц;
Кол-во таблиц страниц будет определяться кол-вом сегментов программы;

Таблица сегментов - локальная, описывает виртуальное адресное пр-во процесса, т.е. содержит информацию о сегментах, с которыми

работает программа;
При этом сегмент кратен странице
⇒ до числа страничного преобр-я остался
(от схемы выше)
один шаг. Но понятие сегмента остаётся:



У нас есть сегменты
- данных (статические данные (их м.б. много ~ много модулей));
- кода;
- стека;
Но все эти сегменты находятся в адр.пр-ве процесса

Итоге, можно отказаться от идеи с таблицей сегментов (на схеме в начале лекции);
Но в совр. системах имеется тенденция к увеличению быстродействия процессоров, объёмов памяти ЗУ и увеличению объёмов прогр. кода ⇒ к увеличению адр. пр-в программ;
У программ большого объёма большое адр. пр-во. В таком случае таблица страниц (она системная, значит она находит-

ся в об-ти данных ядра) становится больше. При этом большие программы могут сильно ветвиться.

Так и возникла идея DLL: чтобы не компоновать большие библиотеки с программами, можно сделать их динамически подгружаемыми;

Поэтому пришли к следующей идее:
многоуровневые таблицы страниц, т.е. в системной памяти находятся только актуальные таблицы страниц.
актуальные — те, в которых находятся дескрипторы страниц, с которыми в текущий момент времени работает процессор;
При этом, если процессор обращается к странице, отсутствующей в памяти, возникает страничное прерывание, „но теперь это многократно расширенный процесс";

Наши комп-ры 64-разрядные (режим Long), но есть режим compatibility (совместимости), позволяющий запускать 32-разр. приложения;

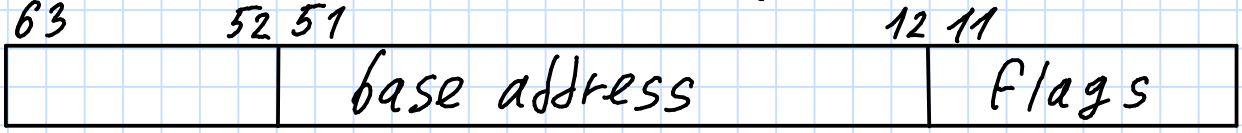
Рассм. схему для x64
(не IA64, она отличается)
регистры (в т.ч. адресные) 64-разрядные;
Размер страницы - 4 Кб (2^{12} байт);

63 48	47 9	39	38 9	30	29 9	21	20 9	12	11 12	0
Зарезерв. вирт.	page map level 4 pointer offset		page directory pointer offset		page directory offset		page table offset		offset	

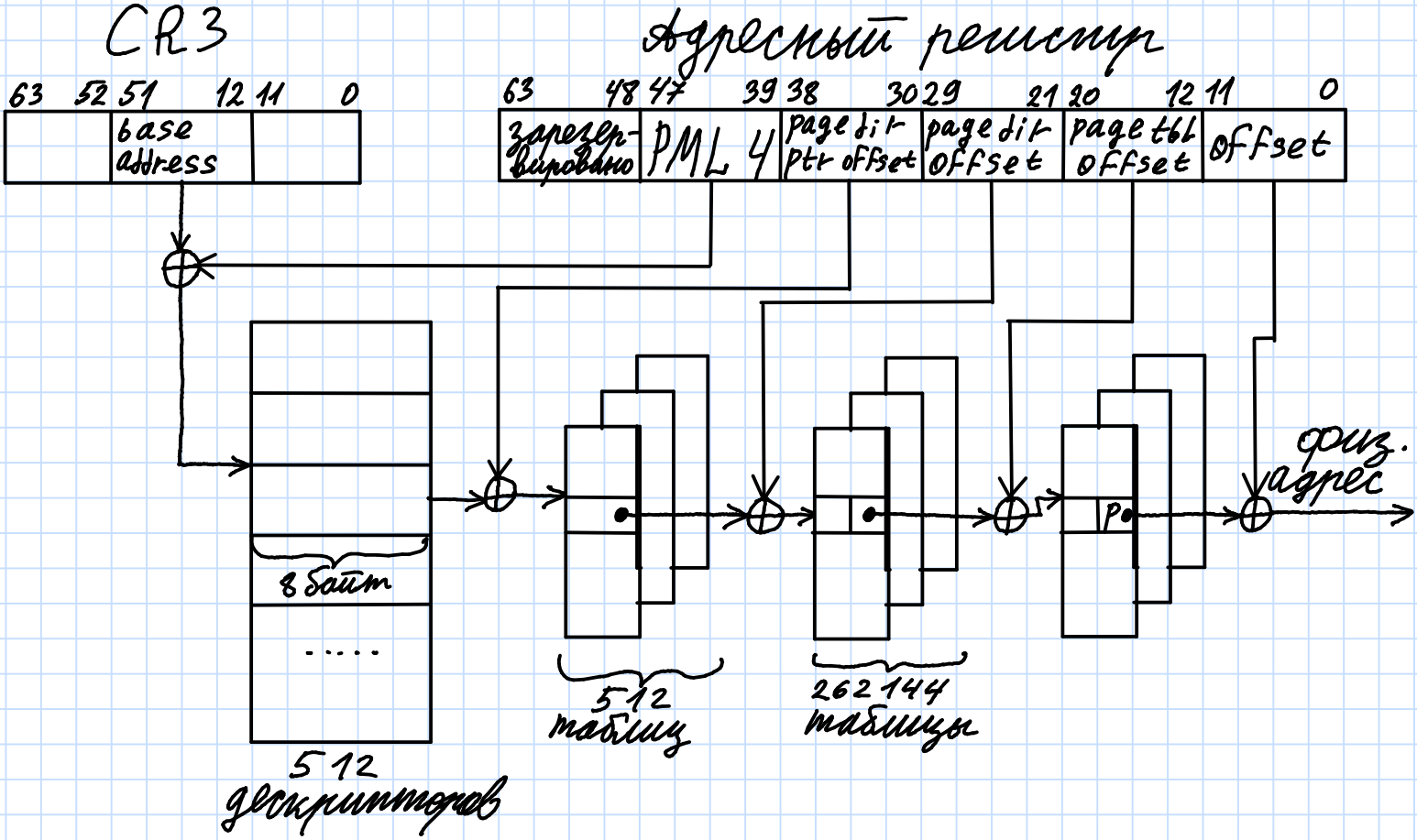
(PML4) | по сути pointer на pointer

physical page offset (это смещ. в странице, т.е. ≤ 4096)

В процессорах Intel есть 64-раз. р-р CR3, содержащий начальный адрес таблицы 4 уровня;



9 разрядов ⟹ 512 дескрипторов могут содержать 1-я таблица 4 уровня;
Каждый дескриптор занимает 8 байт и содержит начальный адрес таблиц следующего уровня (pointer)
⟹ их м.б. 512;



Все дескрипторы во всех таблицах по 8 байт, тогда размер таблицы $2^9 \cdot 8 = 2^{12}$ байт
⟹ все таблицы размером в 1 страницу;
Это очень важно, т.к. в системах передача страниц оптимизирована;

512 таблиц, в каждой 512 дескрипторов ⟹

⟹ 262 144 таблицы след. уровня могут быть (page directory);

262 144 · 512 = 134 217 528 — кол-во страниц, которое м.б. создано для процесса;

↳ это не значит, что все страницы создаются одномоментно!

Более того, не все таблицы создаются сразу; Они создаются тогда, когда в них возникает необходимость, т.е. в рез-те прерывания

Например, процессор мог обратиться к несуществующей странице, у которой не существует далее таблицы страниц, в которой она находится; Далее может оказаться, что не существует каталога таблиц страниц, который ссылается на эту часть адр. пр-ва и т.д. (таблица pointer'ов на эти каталоги) ⟹

⟹ лавина прерываний ⟹ время (колоссальные накладные расходы);

Но есть выигрыш — увеличение мультизадачности;

Для этого всё делается.

Вирт. указатель (адрес) занимает 48 бит, что позволяет адресовать 2^{48} байт или 256 ТБ;

Сокращение возможной адресации связано с аппаратными ограничениями, т.е. 64-разр. адресации разработчики пока не смогли достичь; (уверен?)

Для 4 таблиц размещение указателей

занимает много времени (с т.з. тактовой частоты процессора);
Поэтому в совр. процессорах, чтобы сократить время преобразования, стали использовать соотв. кэши;

Кэш TLB (386+)
Translation Lookaside Buffer
В нём находятся адреса физ. страниц, к которым были последние обращения;

Раньше назывался 4-направленным ассоциативным по мн-ву кэшем;

Управляющие регистры:

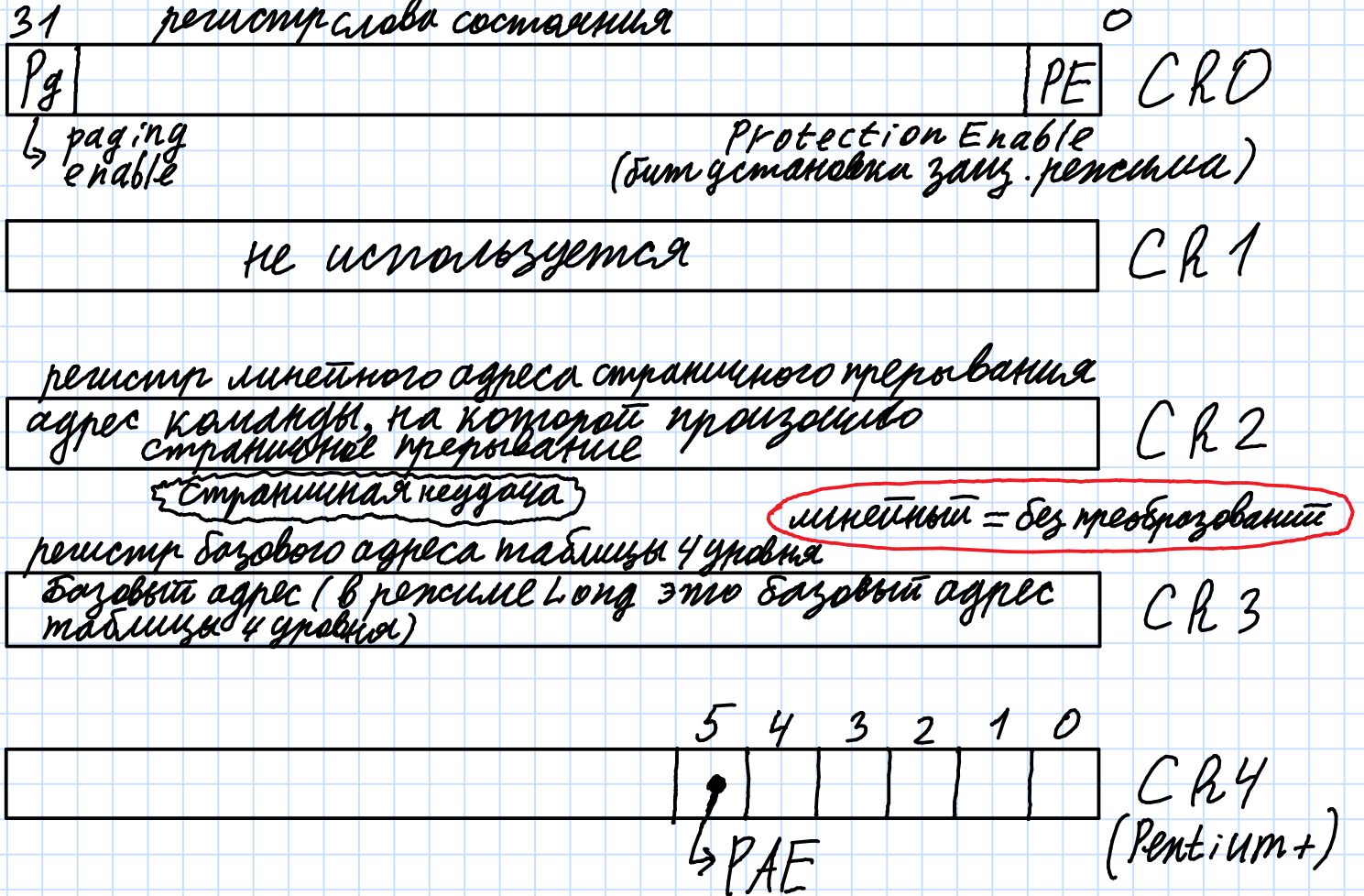


Схема PAE (Physical Address Extension)

Существуют архитектурно-зависимые регистры,

которые находятся в оперативной памяти по известному адресу

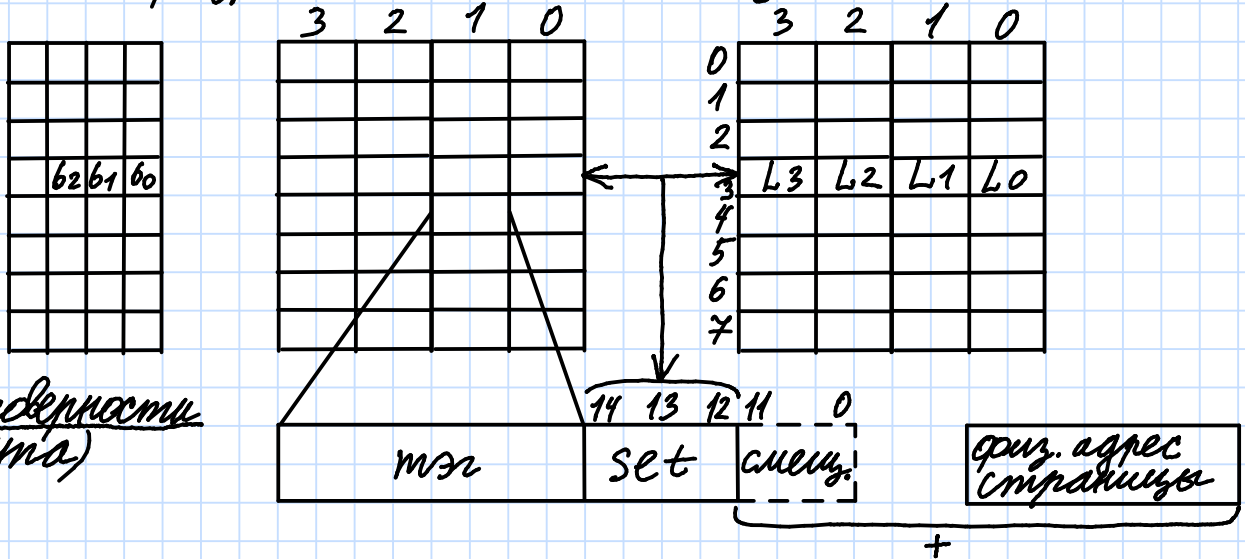
В Long много флагов, т.к. идея Intel и Microsoft – поддержка аппаратной совместимости;
Компьютер стартует в реальном режиме (16-разр.), потом программно переводится в защищённый, а затем в Long;

При этом для адресации в режиме Long создаются таблицы всех 4-х уровней (по одной каждого уровня);
Дальше – по мере необходимости (см. ранее);

Продолжение про TLB.
Здесь реализован алгоритм псевдо-LRU
(аппроксимация, т.к. алгоритм LRU в чистом виде очень затратный)
(программы обладают св-вом локальности ⟹ надо держать в памяти информацию, к которой были посл. обращения)

Кроме TLB каждый процессор имеет кеш 1-го и 2-го уровня (L1 и L2). Все процессоры могут обращаться к кешу 3 уровня (общий доступ);

TLB (486)
↳ 32-разр. ⟹ смещение 3 2-разр.



Бит достоверности (доступа)

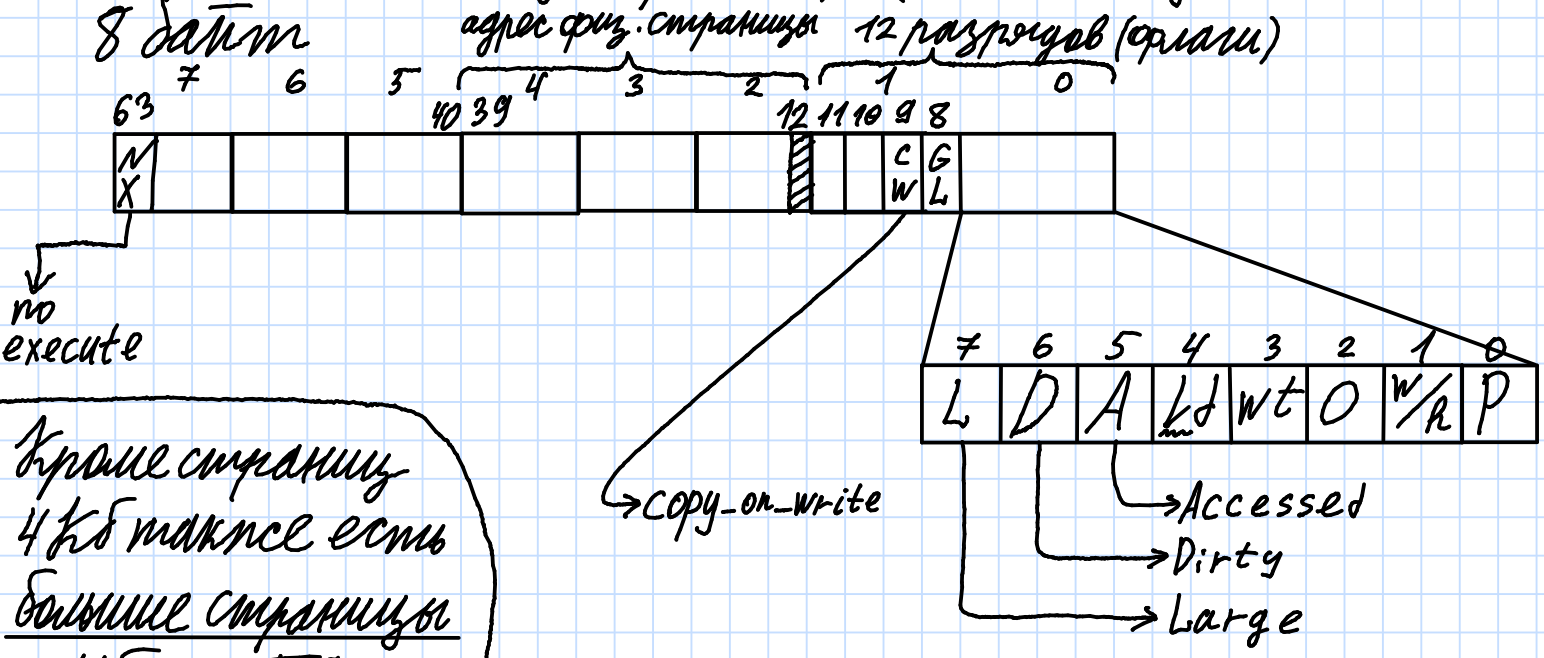
TLB - 4-напр. acc. полн-ву ⟹ частично-ассоциативный кэш (как и все кэши в чипе процессора)

Для реализации алгоритма псевдо-LRU в кэше имеется блок достоверности и LRU;
Бит достоверности (бит обращения) уст. в 1 при загрузке кэша и при каждом обращении к соотв. странице;
Также на каждую страницу определены b_0, b_1, b_2:

$b_0 \to 1$, если было обращение к паре L_0, L_1;
0, если было обращение к паре L_2, L_3;
$b_1 \to 1$, если в паре L_0, L_1 было обращение к L_0;
0, если в паре L_0, L_1 было обращение к L_1;
$b_2 \to 1$, если в паре L_2, L_3 было обращение к L_2;
0, если в паре L_2, L_3 было обращение к L_3;

Благодаря алгоритму псевдо-LRU TLB эффективнее по времени, чем полностью ассоциативный кэш;

Страничный дескриптор (PTE - Page Table Entry)
8 байт



Кроме страниц 4 Кб также есть большие страницы 2 Мб для БД

Ленивая аллокация
(ленивое выделение памяти)

В Linux определён набор сист. вызовов, которые работают с вирт. памятью, т.е. для процесса создаётся вирт. память;

Т.е. большую часть времени процессор работает с вирт. адресами;

Выделение памяти (сист. вызов mmap) происходит при запросе, но на самом деле память не выделяется, а только добавляется соотв. запись в структуры ядра ОС;

При первом обращении процесса к фактически не выделенной памяти происходит страничное исключение (page fault). При обработке этого исключения система выполнит выделение памяти;

Зачем так сделано?

Для экономии памяти, т.к. процесс, запрашивая вирт. память, может её никогда не использовать;

Поэтому на 1 этапе выполняется т.н. ленивый, т.е. относительно быстрый этап выделения;

Архитектура ядер современных операционных систем.

Существует 2 архитектуры вычисл. систем:
1) с монолитным ядром;
2) с микроядром (микроядерная архитектура);

Windows и UNIX/Linux - системы с монолитным ядром, хотя UNIX всегда имела минимизированное ядро

Монолитное ядро - программа, имеющая модульную структуру, т.е. состоящее из подпрограмм;

Микроядро (Microkernel architechture) -
- модуль ядра, реализующий набор низкоуровневых ф-ций, т.е. непосредственное взаимодействие с аппаратной частью и обеспечение взаимодействия между процессами ОС, которые, возможно, имеют собств. адр. пр-ва и выполняются на уровне пользователя;

Модуль микроядра обеспечивает взаимодействие между процессами ОС и между процессами пользователей и процессами ОС с помощью передачи сообщений;

Конец 60-х: Мединк и Донован сформулировали идею иерархической структуры ОС (иерархической машины);

Иерархия была построена по отн-ю к процессу (единица декомпозиции системы);

Уровни этой иерархической машины располагаются в зав-ти от близости к аппаратному обеспечению системы;

Идея микроядерной архитектуры базируется на иерархической машине;