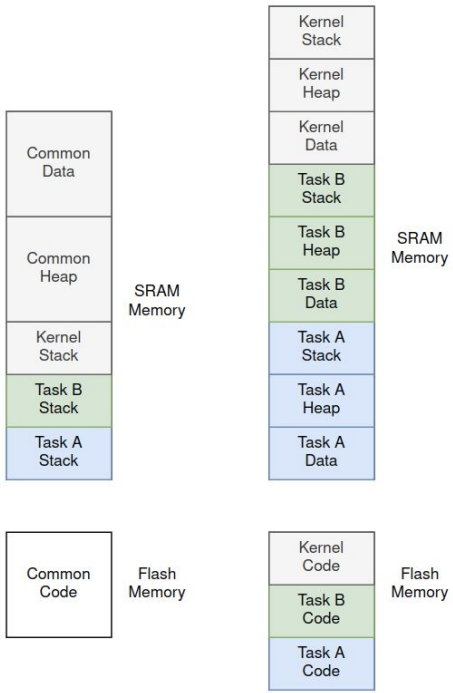
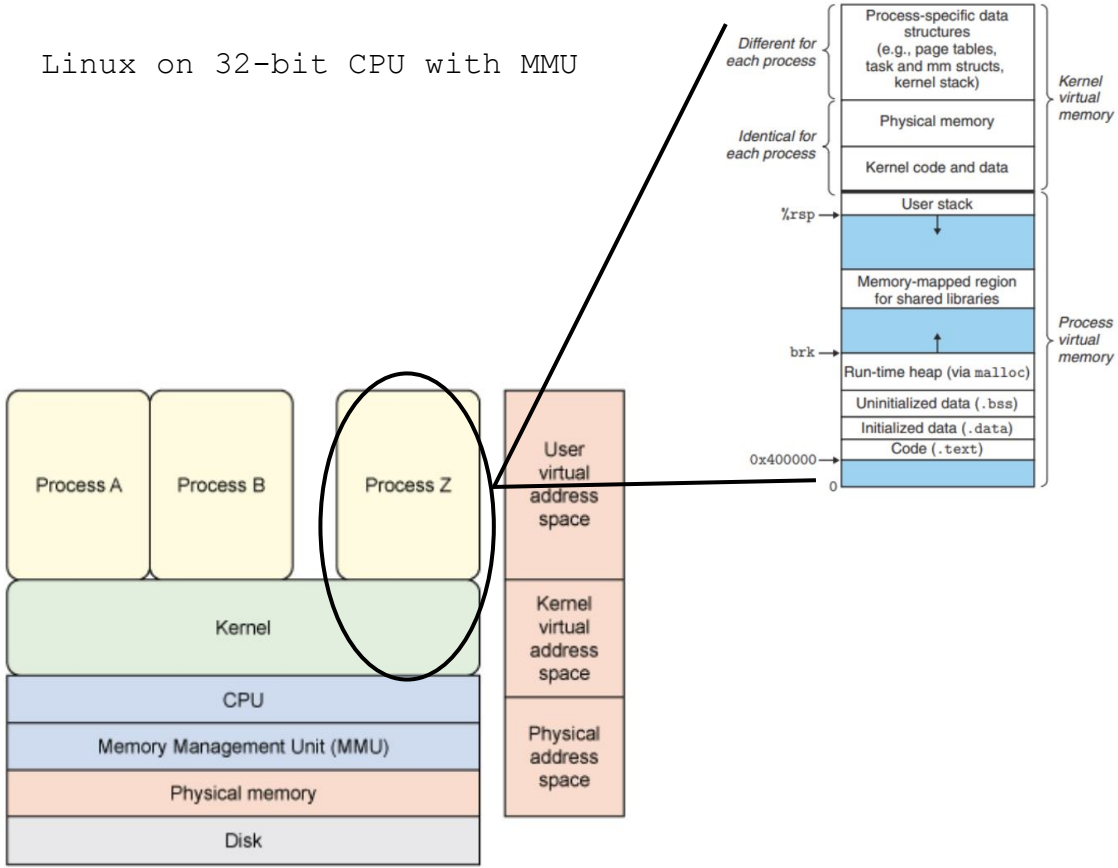
Virtual Memory

Embedded OS on Cortex-M (without MMU)



Linux on 32-bit CPU with MMU

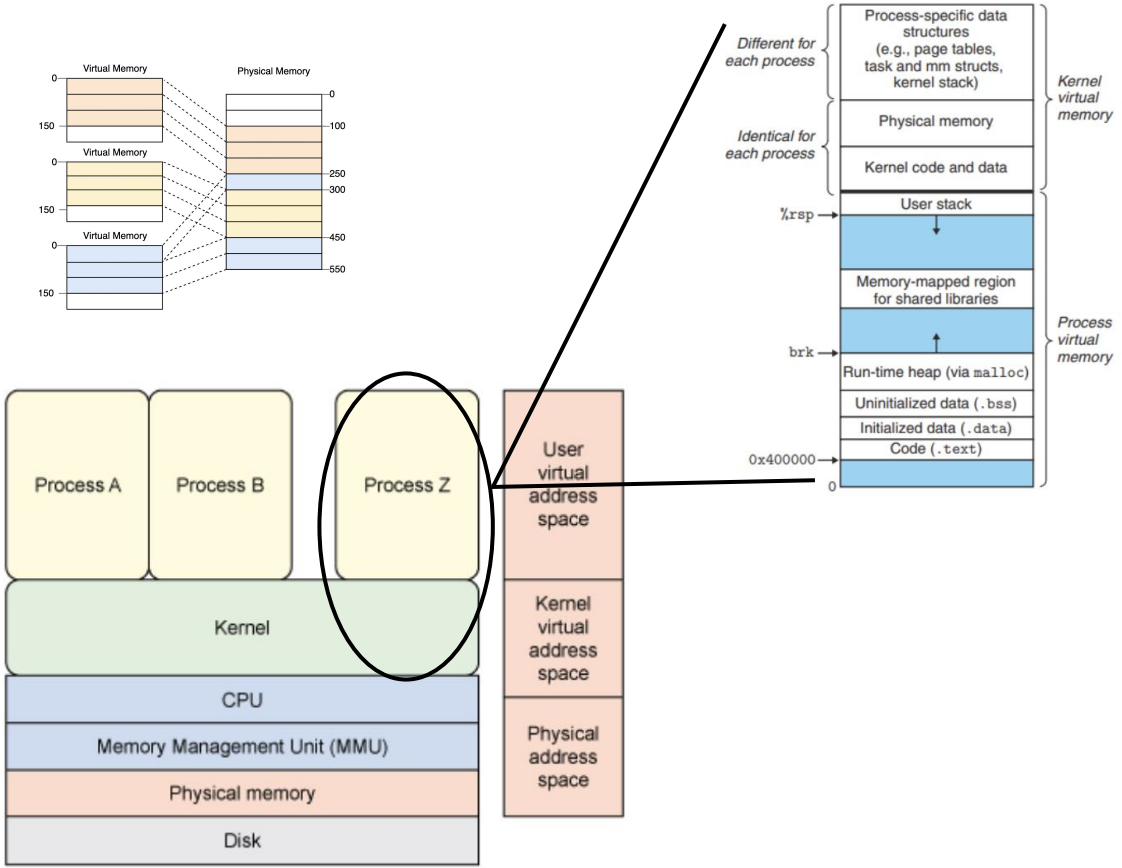


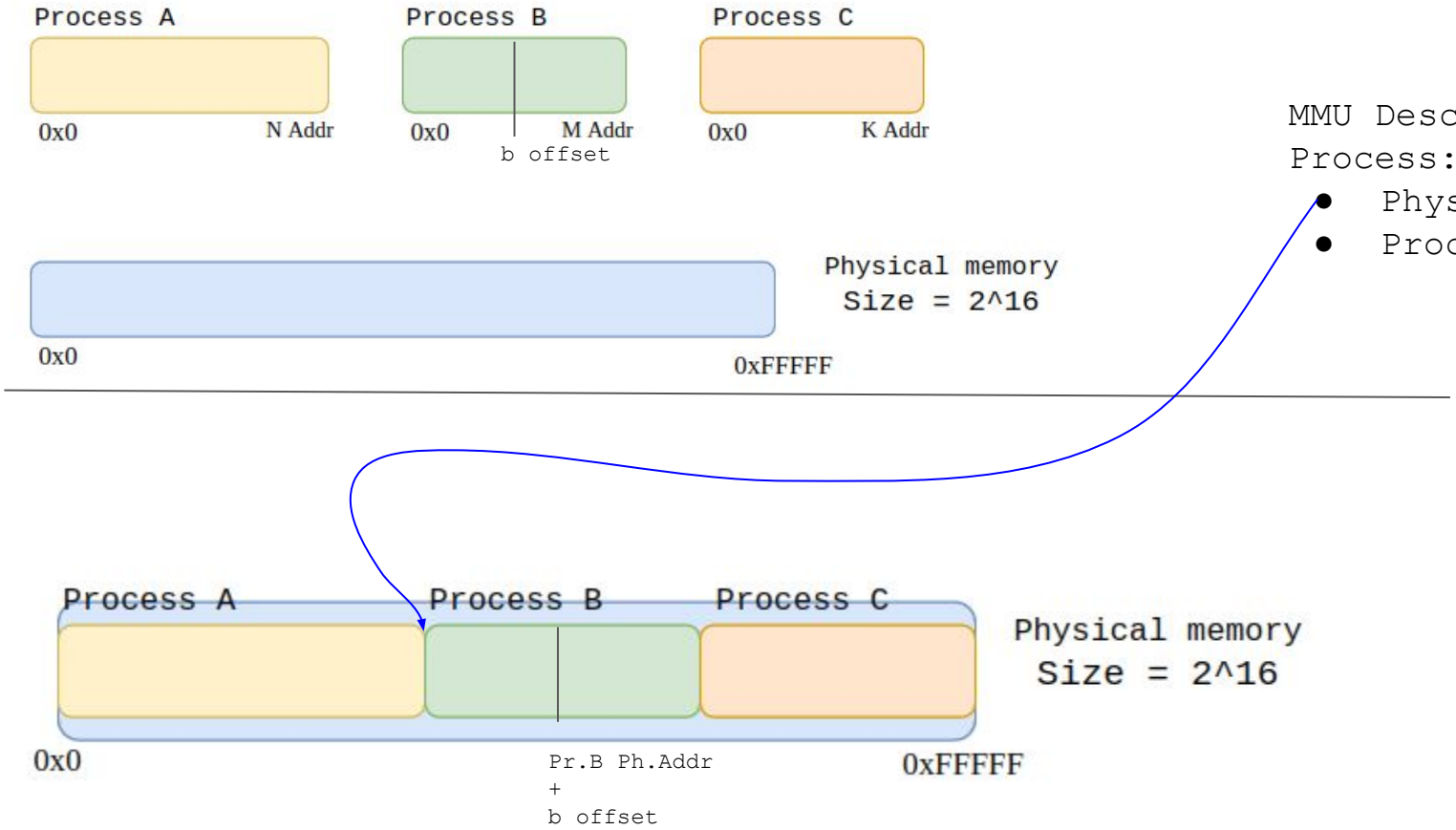
Преимущества виртуального адресного пространства:

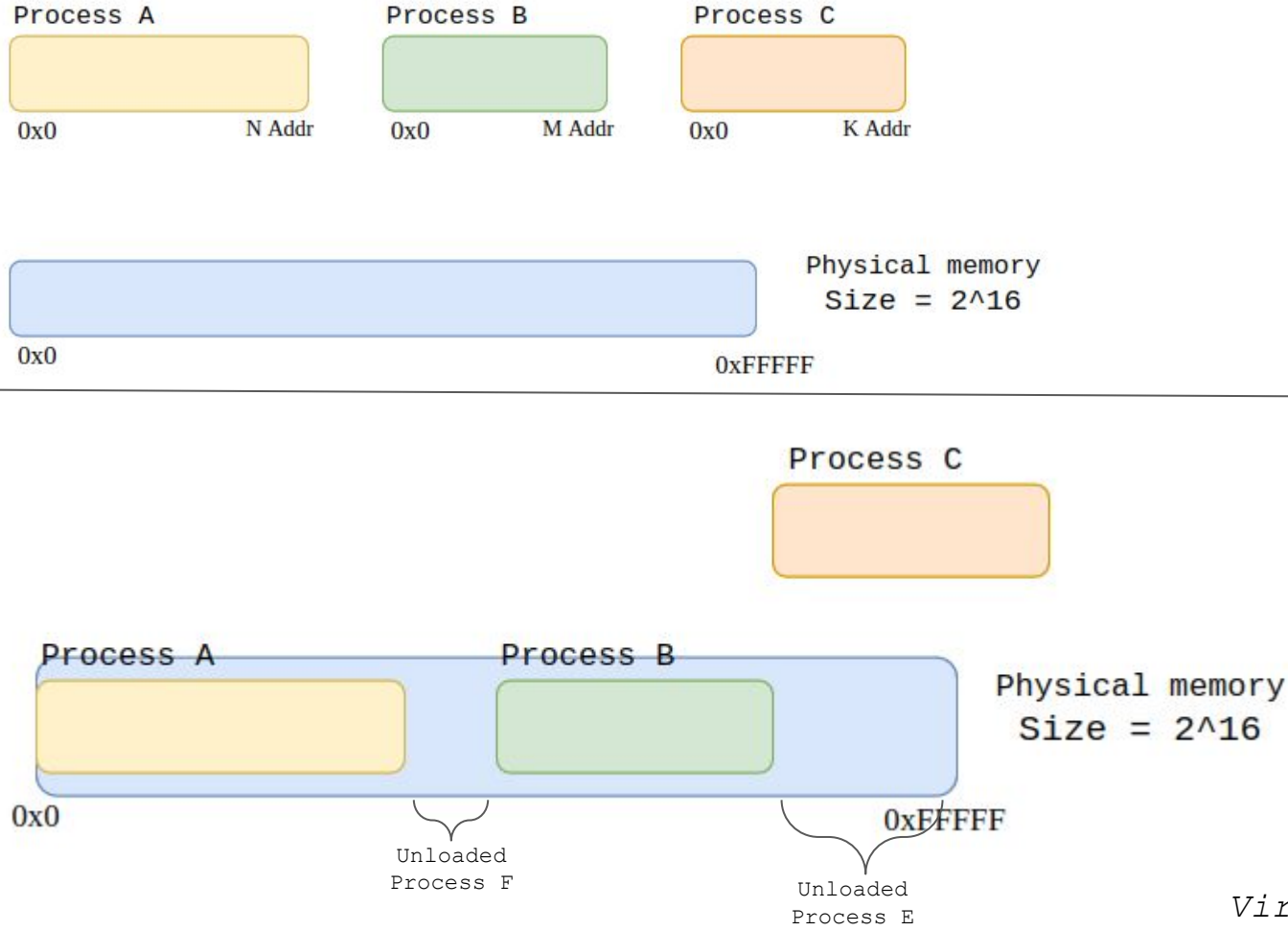
+ Все процессы изолированы друг от друга и не могут повредить данные другого процесса

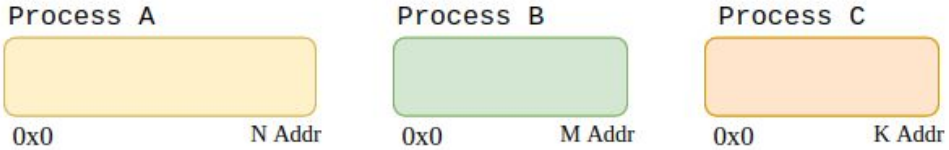
+ Программа всегда линкуется в одно адресное пространство (всегда по одним и тем же адресам)

Linux on 32-bit CPU with MMU

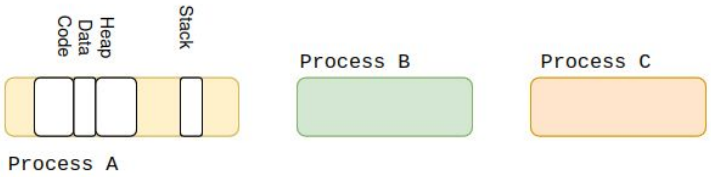



Process A
0x0
N Addr
Process B
0x0
M Addr
b offset
Process C
0x0
K Addr
Physical memory
Size = 2^16
0x0
0xFFFFF
MMU Descriptor Per Process:
Physical Address
Process Size
Process A
Process B
Process C
Physical memory
Size = 2^16
0x0
Pr.B Ph.Addr
+
b offset
0xFFFFF

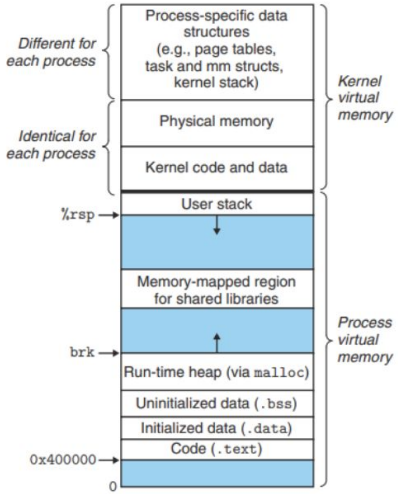

Process A
0x0
N Addr
Process B
0x0
M Addr
Process C
0x0
K Addr
Physical memory
Size = 2^16
0x0
0xFFFFF
Process C
Process A
Process B
Physical memory
Size = 2^16
0x0
0xFFFFF
Unloaded
Process F
Unloaded
Process E

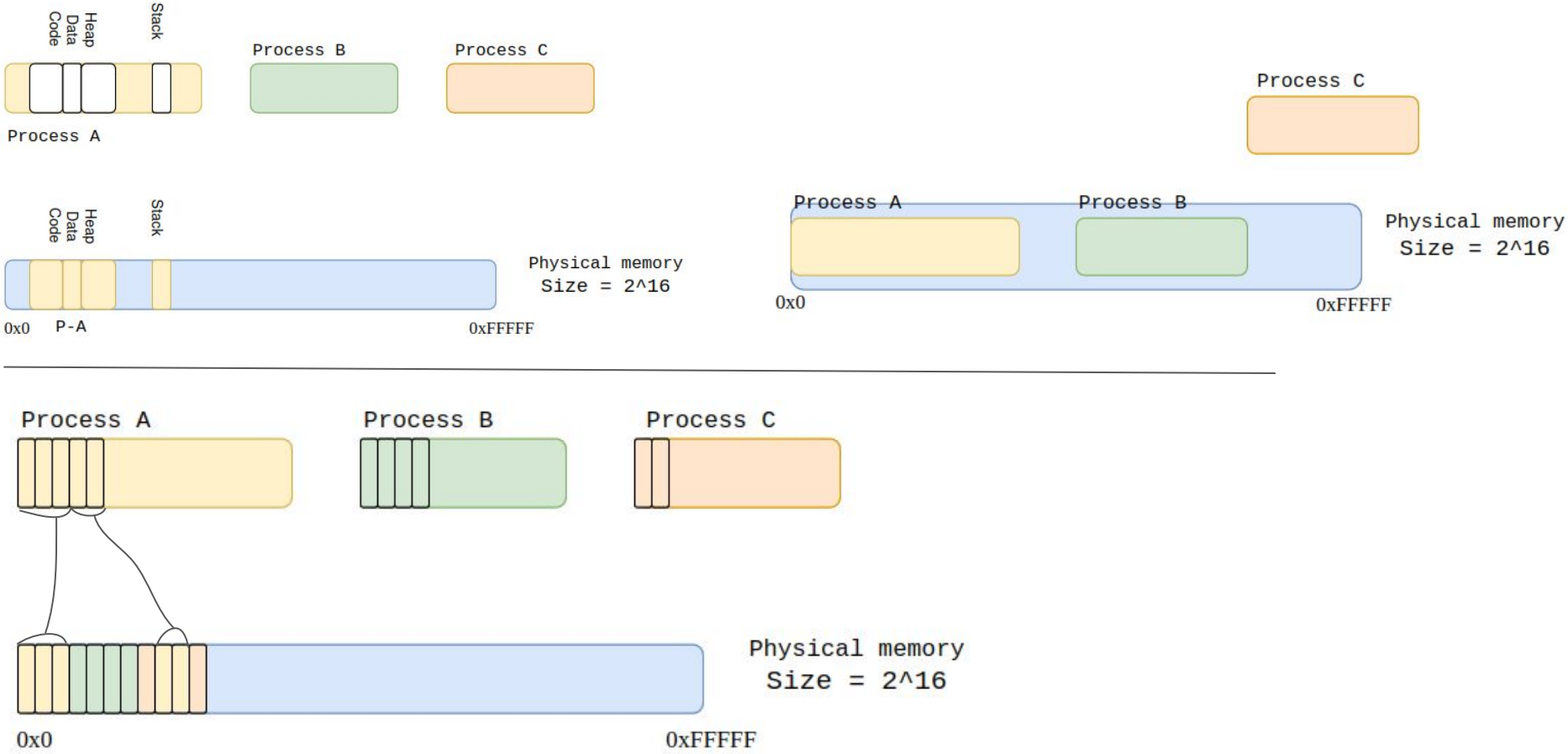

Process A
Process B
Process C
0x0
N Addr
0x0
M Addr
0x0
K Addr


Physical memory
Size = 2^16
0x0
0xFFFFF


Code
Data
Heap
Stack
Process A
Process B
Process C


Code
Data
Heap
Stack
Physical memory
Size = 2^16
0x0
P-A
0xFFFFF


Different for each process
Identical for each process
Process-specific data structures (e.g., page tables, task and mm structs, kernel stack)
Physical memory
Kernel code and data
Kernel virtual memory
%rsp
User stack
Memory-mapped region for shared libraries
brk
Run-time heap (via malloc)
Uninitialized data (.bss)
Initialized data (.data)
Code (.text)
0x400000
0
Process virtual memory


Code
Data
Heap
Stack
Process A
Process B
Process C
Process C
Code
Data
Heap
Stack
Physical memory
Size = 2^16
0x0
P-A
0xFFFFF
Process A
Process B
Physical memory
Size = 2^16
0x0
0xFFFFF
Process A
Process B
Process C
Physical memory
Size = 2^16
0x0
0xFFFFF

Process A

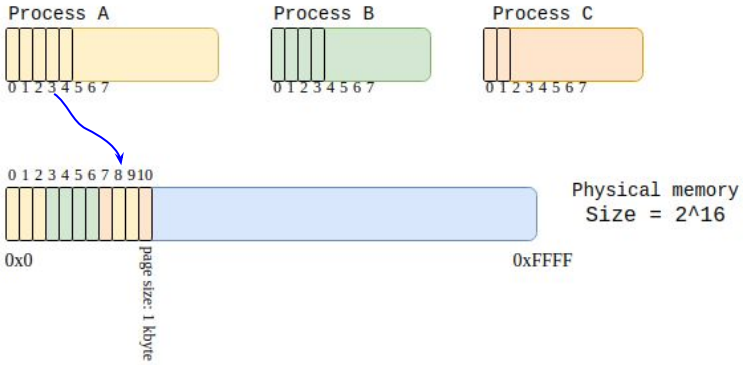
V-Page	Ph-Page	Valid
0	0	1
1	1	1
2	2	1
3	8	1
4	9	1
5	0	0
6	0	0
...	...	...

Process B

V-Page	Ph-Page	Valid
0	3	1
1	4	1
2	5	1
3	6	1
4	0	0
5	0	0
6	0	0
...	...	...

Process C

V-Page	Ph-Page	Valid
0	7	1
1	10	1
2	0	0
3	0	0
4	0	0
5	0	0
6	0	0
...	...	...

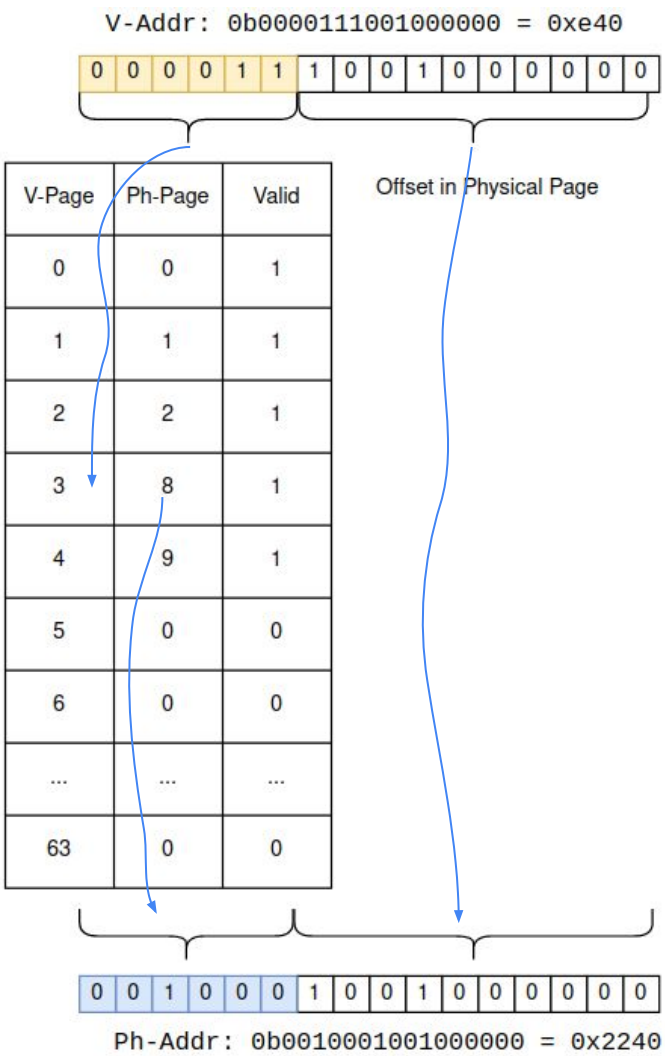


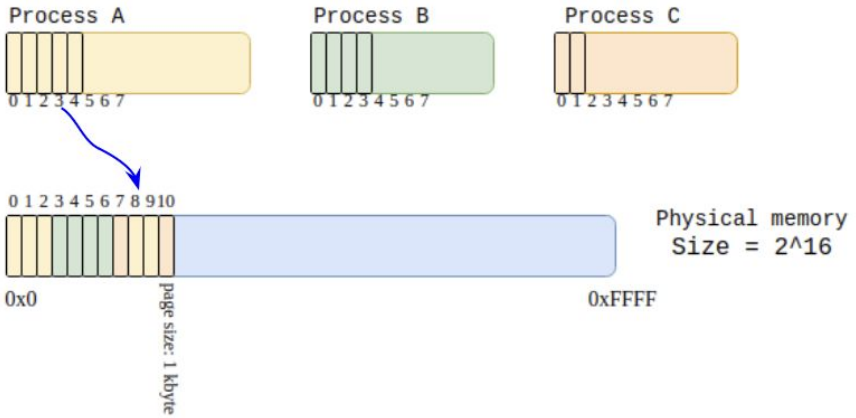
16-bit address: 0b1111 1111 1111 1111

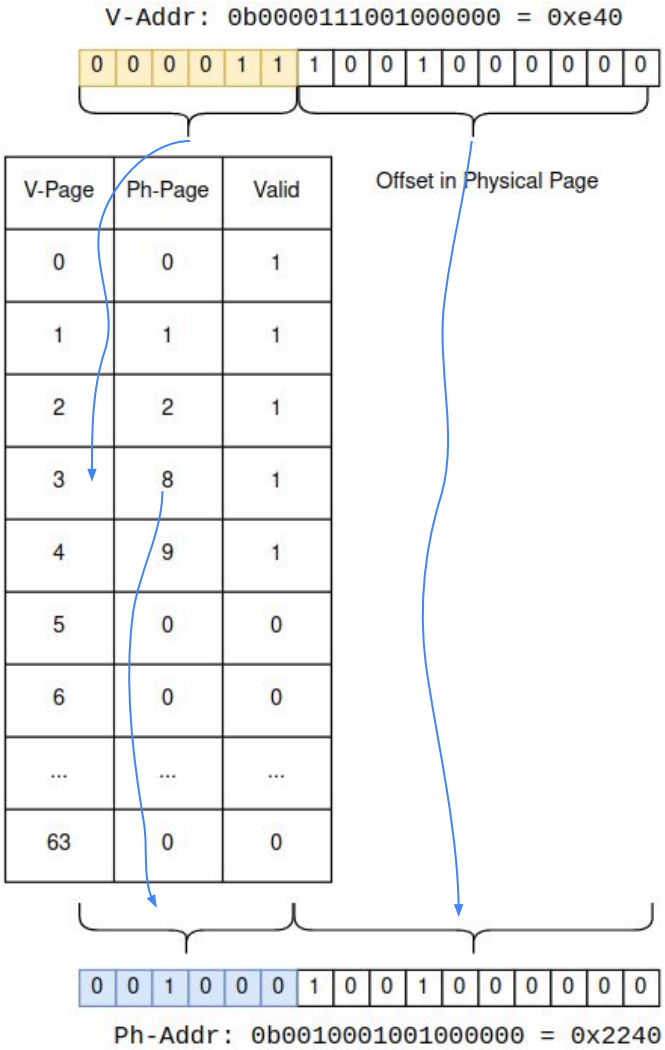
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Offset in Physical Page

V-Page	Ph-Page	Valid
0	0	1
1	1	1
2	2	1
3	8	1
4	9	1
5	0	0
6	0	0
...	...	...


V-Addr: 0b0000111001000000 = 0xe40
0 0 0 0 1 1 1 0 0 1 0 0 0 0 0 0
Offset in Physical Page
V-Page
Ph-Page
Valid
0 0 1
1 1 1
2 2 1
3 8 1
4 9 1
5 0 0
6 0 0
... ...
63 0 0
0 0 1 0 0 0 1 0 0 1 0 0 0 0 0 0
Ph-Addr: 0b0010001001000000 = 0x2240


Process A
Process B
Process C
0 1 2 3 4 5 6 7
0 1 2 3 4 5 6 7 8 9 10
Physical memory
Size = 2^16
0x0
page size: 1 kbyte
0xFFFF



- 16 bit (64 kBytes) Physical Memory
- 16 bit Virtual Memory
- 1 kBytes virtual page
- Table size = 64 kBytes / 1 kBytes = 64 entries
- 1 entry size = 4 bytes
- Total Table size = 64 entries * 4 bytes = 256 bytes

- 32 bit (4 GBytes) Physical Memory
- 32 bit (4 GBytes) Virtual Memory
- 4 kBytes virtual page
- Table size = 4 GBytes / 4 kBytes = 1 000 000 entry
- 1 entry size = 4 bytes
- Total Table size = 1 000 000 * 4 bytes = 4 MBytes

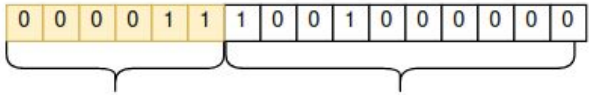
Проблема:

4 Mbytes per Process - должны быть выделены всегда. Это много.

Как уменьшить размер таблицы ??

Хранить в памяти только часть “таблицы страниц” - для используемых (Valid) страниц.

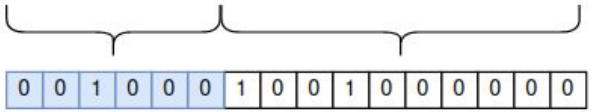
V-Addr: 0b0000111001000000 = 0xe40



Offset in Physical Page

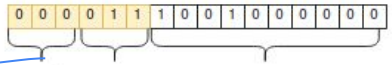
V-Page	Ph-Page	Valid
0	0	1
1	1	1
2	2	1
3	8	1
4	9	1
5	0	0
6	0	0
...	...	...
63	0	0

Total Table size = 64 entries * 4 bytes = **256 bytes**



Ph-Addr: 0b0010001001000000 = 0x2240

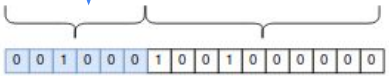
16-bit address: 0b1111 1111 1111 1111



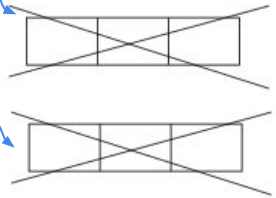
V-Page	Next Level Table Addr	Valid
0	0xXXXX	1
1	0	0
2	0	0
3	0	0
4	0	0
5	0	0
6	0	0
7	0	0

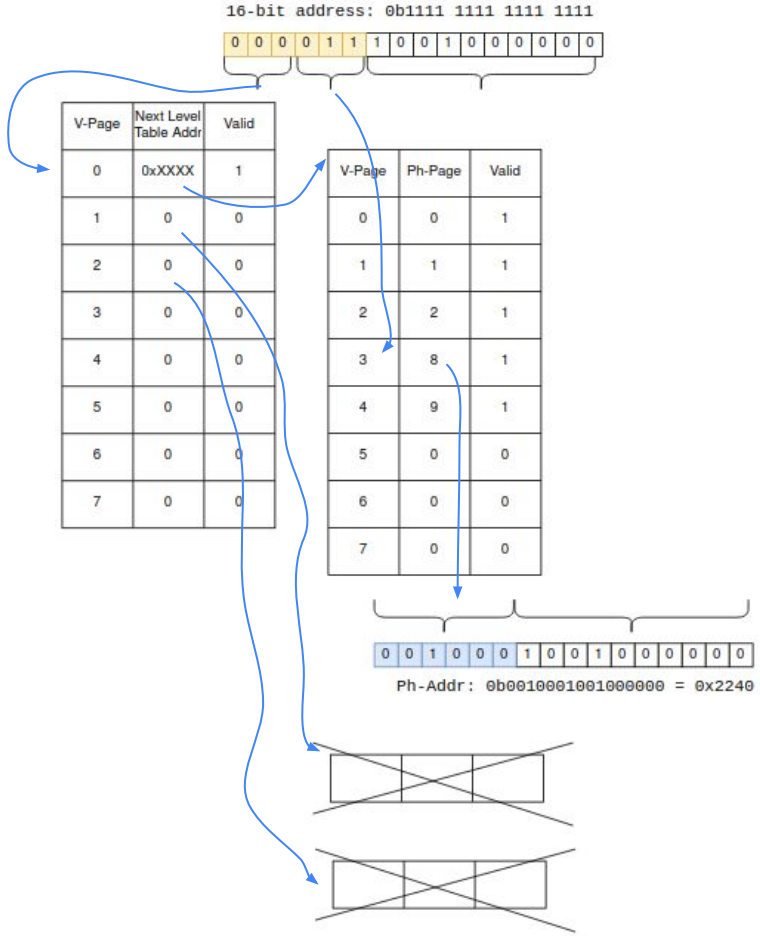
V-Page	Ph-Page	Valid
0	0	1
1	1	1
2	2	1
3	8	1
4	9	1
5	0	0
6	0	0
7	0	0

Total Table size = 8 entries * 2 table * 4 bytes = **64 bytes**



Ph-Addr: 0b0010001001000000 = 0x2240

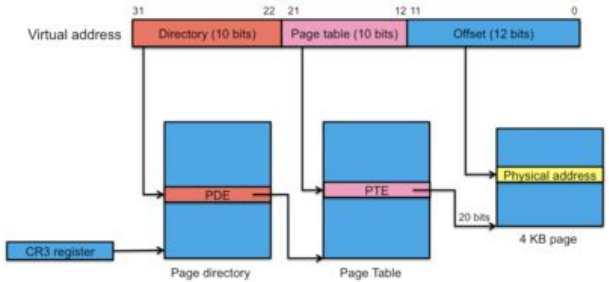


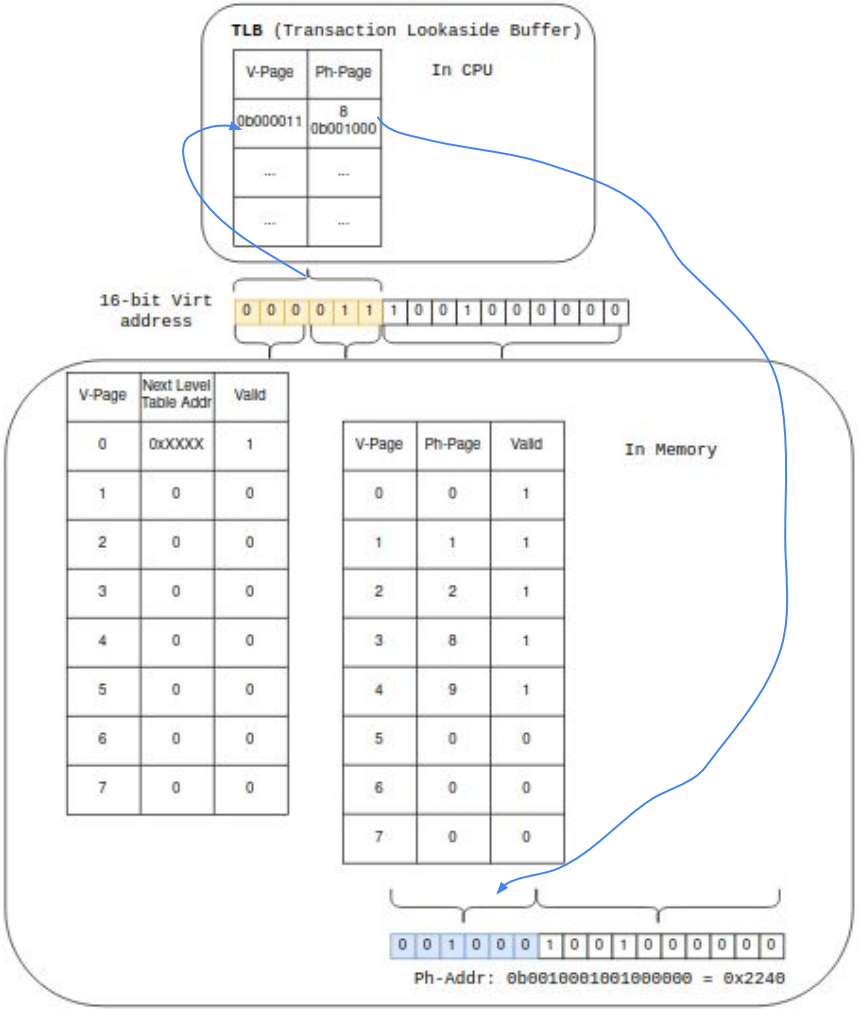


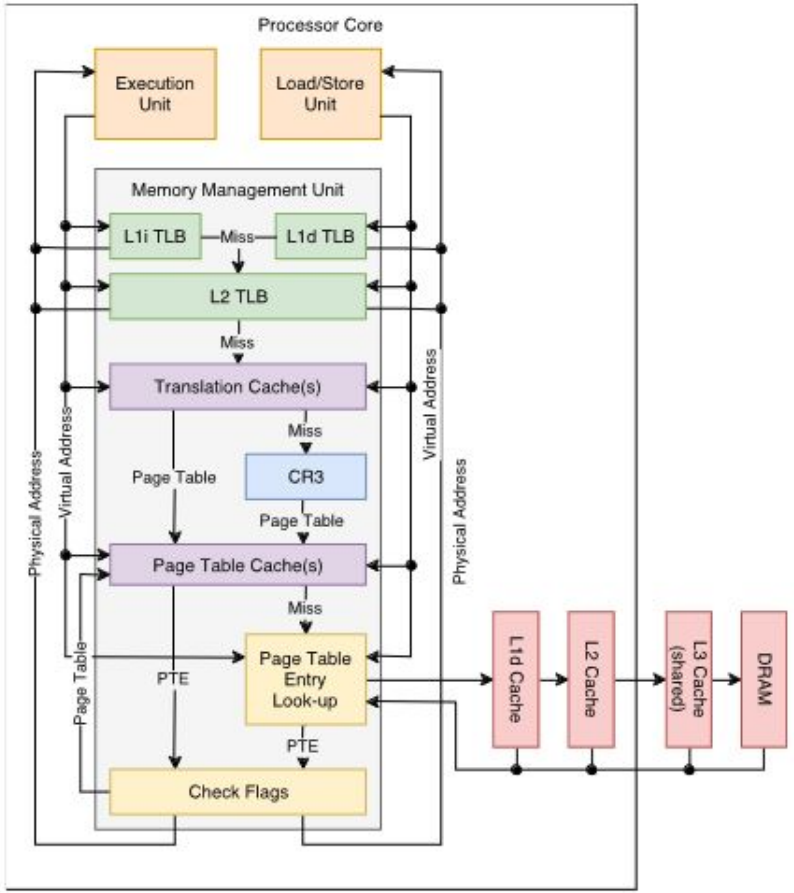
Обычно делают так чтобы одна таблица была по размеру равна одной MMU page.

Например:

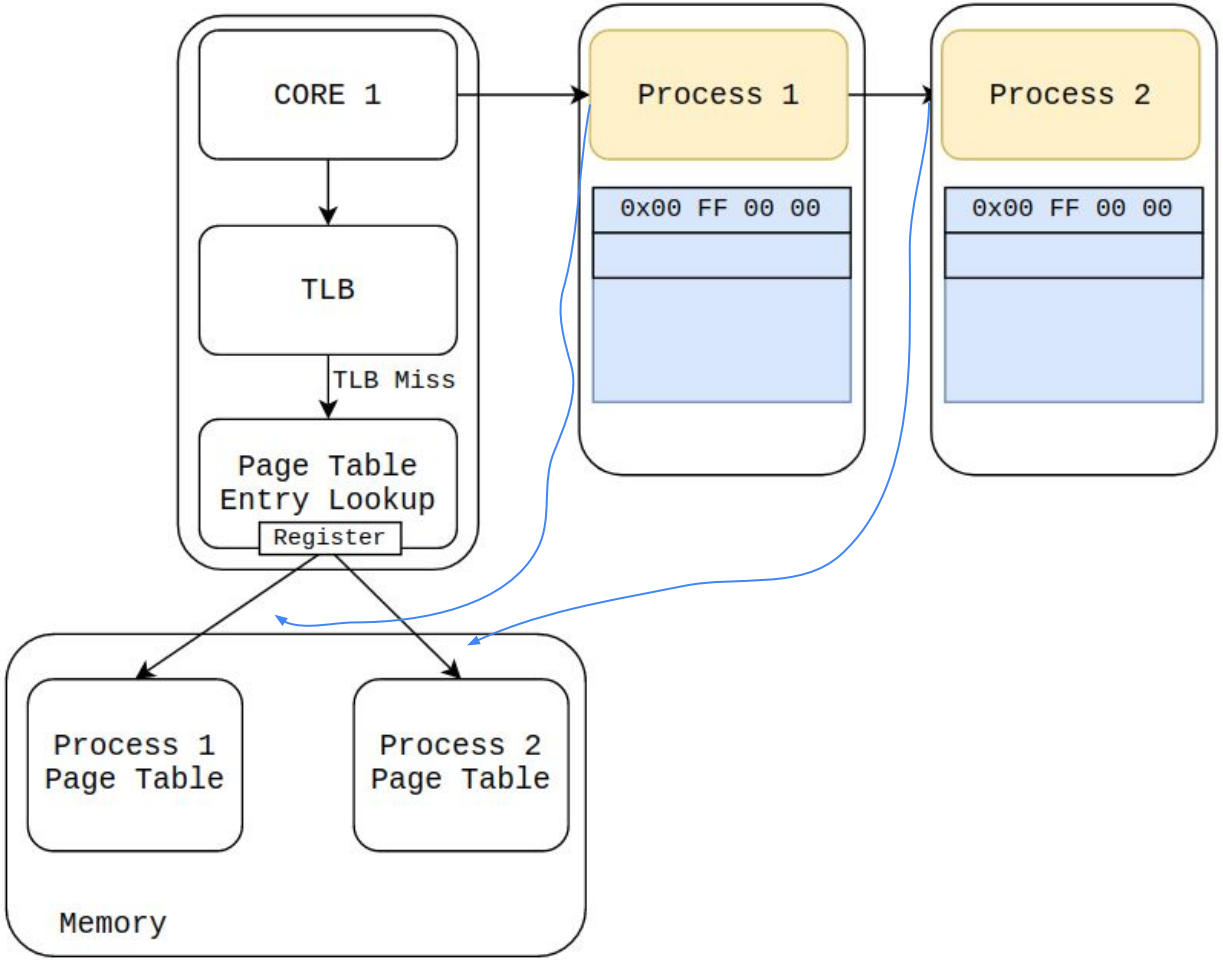
- 32 bit (4 GBytes) Physical Memory
- 32 bit (4 GBytes) Virtual Memory
- 4 kBytes virtual page
- 1 entry size = 4 bytes
- 1 table size = 4 kBytes = 1024 entries
- "First page offset" Bits Number = 10 (2**10 = 1024)
- "Second page offset" Bits Number = 10 (2**10 = 1024)
- Offset in Physical Page = 12 bits







https://download.vusec.net/papers/revanc_ir-cs-77.pdf

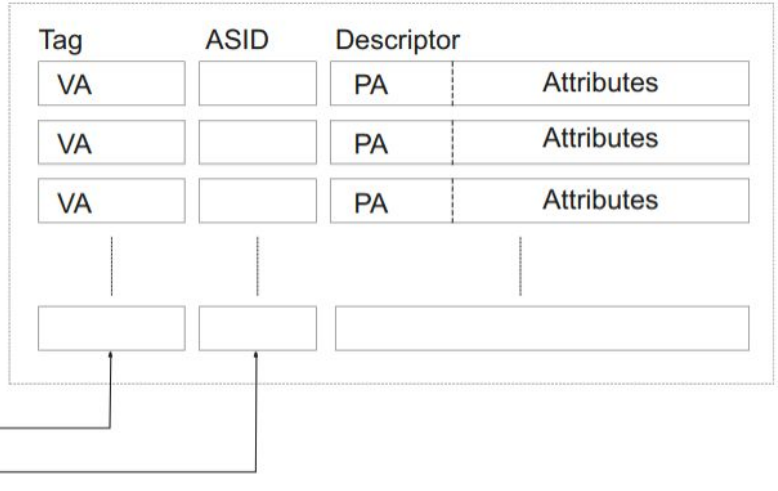


2 процесса используют один и тот же виртуальный адрес 0x00FF0000.

Каждый процесс имеет свои таблицы преобразования. ОС при смене контекста переключает данные таблицы, записывая в “специальный регистр MMU” адрес, где искать корневую таблицу преобразования для данного процесса.

ПРОБЛЕМА:
TLB не переключается - это общий кэш. В TLB мы не можем отличить 0x00ff0000 1го процесса и 0x00ff0000 2го процесса.

Самое простое решение:
Сбрасывать весь TLB при переключении контекста.



Attributes:

- User/SuperVisor R/W Permission flags - доступ на запись/чтение из (не)/привилегированного режима
- Execute Disable - отключить доступ на выполнение кода из этой страницы
- Cache Attributes - настройка параметров кэширования страницы
- Present - страница находится в физической памяти в данный момент
- Dirty - была запись в страницу

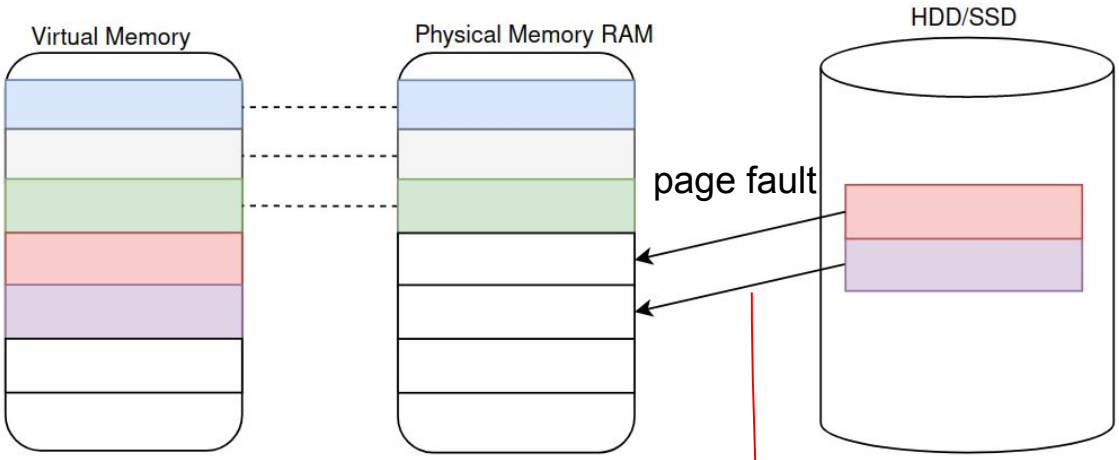
2 процесса используют один и тот же виртуальный адрес 0x00FF0000.

Каждый процесс имеет свои таблицы преобразования. ОС при смене контекста переключает данные таблицы, записывая в “специальный регистр MMU” адрес, где искать корневую таблицу преобразования для данного процесса.

ПРОБЛЕМА:
TLB не переключается - это общий кэш. В TLB мы не можем отличить 0x00ff0000 1го процесса и 0x00ff0000 2го процесса.

Решение:
Добавить некий уникальный номер каждому процессу и хранить его в TLB вместе с адресом:
Process 1 Unique ID, 0x00ff0000
Process 2 Unique ID, 0x00ff0000

https://www.macs.hw.ac.uk/~hwloidl/Courses/F28HS/Docu/DEN0013D_cortex_a_series_PG.pdf




```c
const int const_data = 10;

int main(void)
{
    int *p_data = 0;
    *p_data = 10;  // ???

    p_data = &const_data;
    *p_data = 4; // ???

    p_data = malloc(4097);
}
```

Present Attribute - страница находится в физической памяти в данный момент

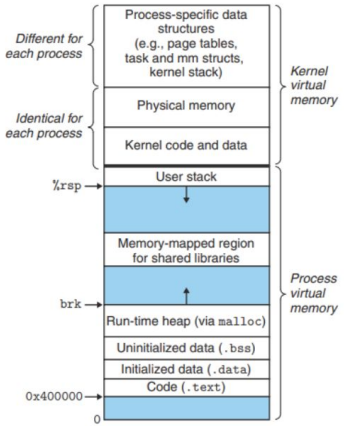
Consider how an executable program might be loaded from disk into memory. One option is to load the entire program in physical memory at program execution time. However, a problem with this approach is that we may not initially need the entire program in memory. Suppose a program starts with a list of available options from which the user is to select. Loading the entire program into memory results in loading the executable code for all options, regardless of whether or not an option is ultimately selected by the user. An alternative strategy is to load pages only as they are needed. This technique is known as **demand paging**.

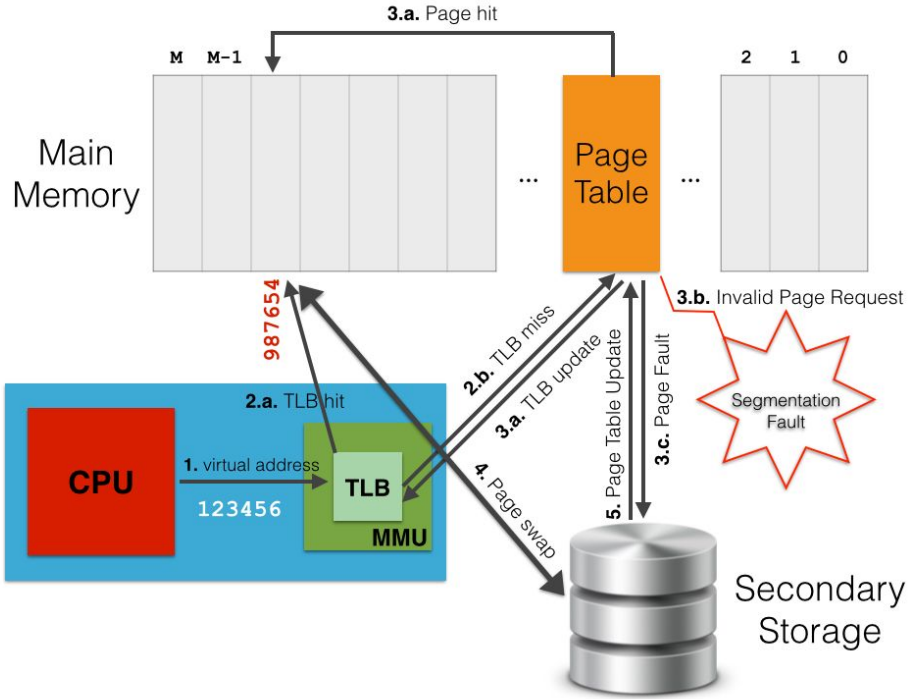
```c
const int const_data = 10;

int main(void)
{
    int *p_data = 0;
    *p_data = 10;  // ???

    p_data = &const_data;
    *p_data = 4; // ???

    p_data = malloc(4097);
}
```

- Что будет при разыменование указателя на нулевой адрес ??
- Что будет если мы попытаемся изменить константу через указатель на нее ??

```c
const int const_data = 10;

int main(void)
{
    int *p_data = 0;
    *p_data = 10;  // ???

    p_data = &const_data;
    *p_data = 4; // ???

    p_data = malloc(4097);
}
```

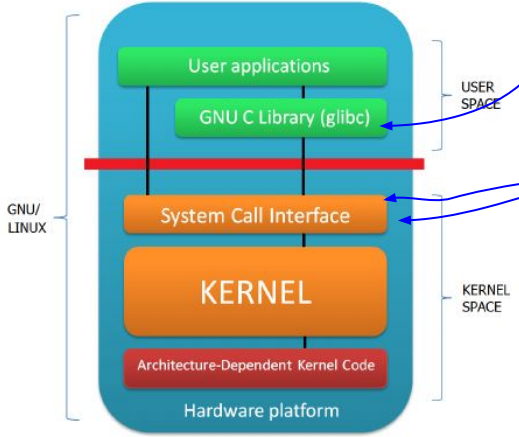
- Что будет при разыменование указателя на нулевой адрес ??
- Что будет если мы попытаемся изменить константу через указатель на нее ??

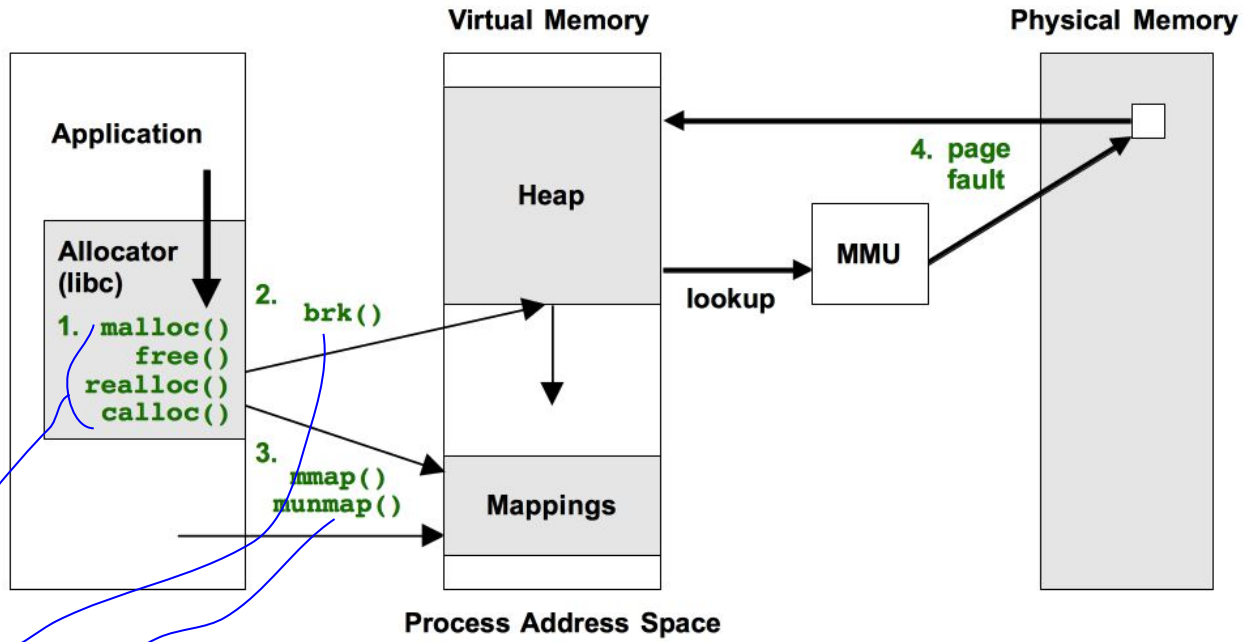


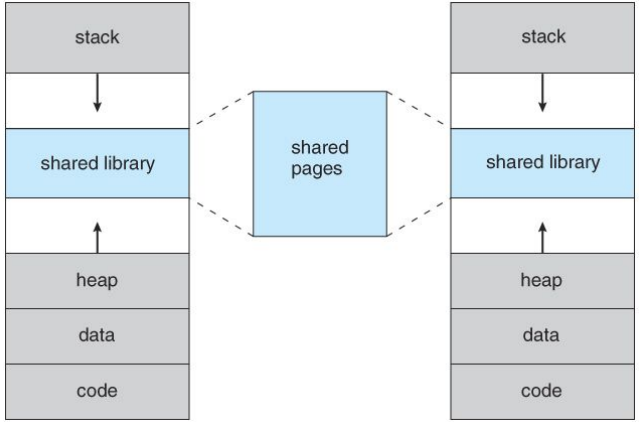


- Что происходит при выделение памяти через malloc ??

```c
const int const_data = 10;

int main(void)
{
    int *p_data = 0;
    *p_data = 10;  // ???

    p_data = &const_data;
    *p_data = 4; // ???

    p_data = malloc(4097);
}
```



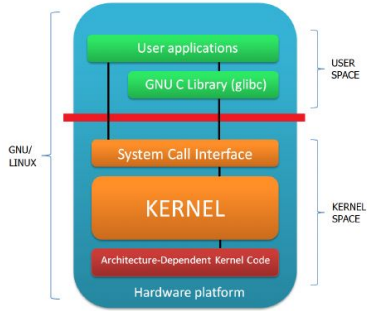


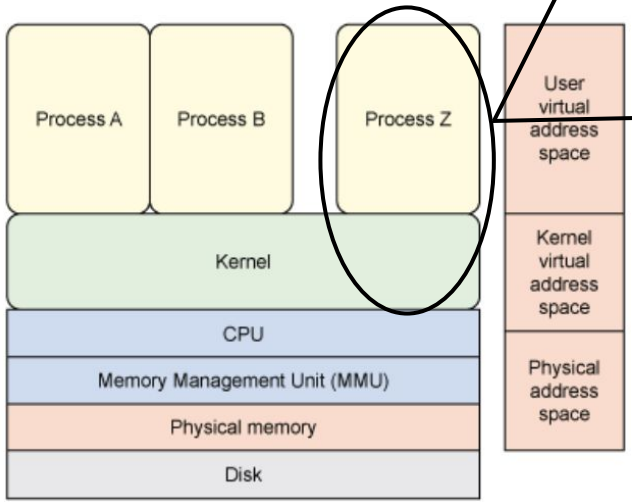


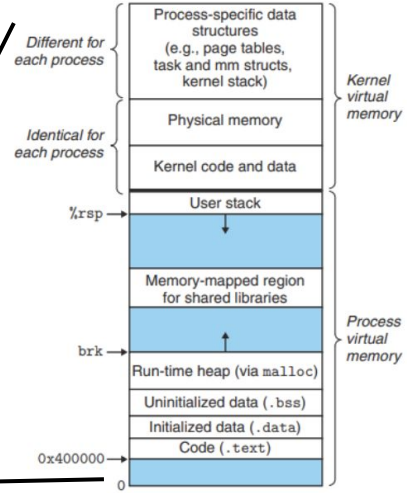

```c
const int const_data = 10;

int main(void)
{
    int *p_data = 0;
    *p_data = 10;  // ???

    p_data = &const_data;
    *p_data = 4; // ???

    p_data = malloc(4097);
}
```

nm ./a.out

```
0000000000001149 T main
                 U malloc@@GLIBC_2.2.5
```







https://cseweb.ucsd.edu//~ricko/CSE131/the%20inside%20story%20on%20shared%20libraries%20and%20dynamic%20loading.pdf

```c
#include <stdio.h>

int my_shared_var = 10;

void shared_function(void)
{
  printf("shared function called: %d\n", my_shared_var);
}
```

```c
#include <math.h>
#include <stdio.h>

extern int my_shared_var;
void shared_function(void);

int main(void)
{
  double pi = 3.14 / 2;
  printf("sin pi = %f\n", sin(pi));

  my_shared_var = 42;
  shared_function();
}
```

```
gcc -c -fpic shared.c
gcc -shared -o libmyshared.so ./shared.o

#move libmyshared.so in /lib/x86_64-linux-gnu/

gcc ./main.c -lm -lmyshared
```

nm ./a.out

```
0000000000001189 T main
0000000000004010 B my_shared_var
                 U printf@@GLIBC_2.2.5
0000000000001100 t register_tm_clones
                 U shared_function
                 U sin@@GLIBC_2.2.5
```

```
root@alexPC:/home/alex/lectures/shared# ldd ./a.out
        linux-vdso.so.1 (0x00007ffe789ef000)
        libm.so.6 => /lib/x86_64-linux-gnu/libm.so.6 (0x00007f0d93662000)
        libmyshared.so => /lib/x86_64-linux-gnu/libmyshared.so (0x00007f0d9365d000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f0d9346b000)
        /lib64/ld-linux-x86-64.so.2 (0x00007f0d937cf000)
```

https://eli.thegreenplace.net/2011/11/03/position-independent-code-pic-in-shared-libraries

```c
extern int my_shared_var1;
extern int my_shared_var2;
void shared_function(void);

int main(void)
{
  my_shared_var1 = 42;
  my_shared_var2 = 84;
  shared_function();
}
```

```c
int my_shared_var1 = 10;
int my_shared_var2 = 20;

void shared_function(void)
{
  my_shared_var1 *= 47;
  my_shared_var2 *= 31;
}
```

```
arm-none-eabi-gcc -mcpu=cortex-m7 -mlittle-endian -mthumb -c -fpic -g -O0
./main.c

arm-none-eabi-gcc ./shared.c -mcpu=cortex-m7 -mlittle-endian -mthumb -g -O0
-shared -fpic -o libtest.so

arm-none-eabi-gcc ./main.o -L. -ltest -mcpu=cortex-m7 -mlittle-endian -mthumb
-T./STM32F746NGHx_FLASH.ld --specs=nosys.specs

arm-none-eabi-objdump -Dz ./a.out
arm-none-eabi-objdump -R ./a.out
```

```
080000bc <main>:
 80000bc:       b580            push    {r7, lr}
 80000be:       af00            add     r7, sp, #0
 80000c0:       4b07            ldr     r3, [pc, #28]   ; (80000e0 <main+0x24>)
 80000c2:       447b            add     r3, pc
 80000c4:       4a07            ldr     r2, [pc, #28]   ; (80000e4 <main+0x28>)
 80000c6:       589a            ldr     r2, [r3, r2]
 80000c8:       212a            movs    r1, #42 ; 0x2a
 80000ca:       6011            str     r1, [r2, #0]
 80000cc:       4a06            ldr     r2, [pc, #24]   ; (80000e8 <main+0x2c>)
 80000ce:       589b            ldr     r3, [r3, r2]
 80000d0:       2254            movs    r2, #84 ; 0x54
 80000d2:       601a            str     r2, [r3, #0]
 80000d4:       f000 f99a       bl      800040c <_etext+0x30>
 80000d8:       2300            movs    r3, #0
 80000da:       4618            mov     r0, r3
 80000dc:       bd80            pop     {r7, pc}
 80000de:       bf00            nop
 80000e0:       1800043e        stmdane r0, {r1, r2, r3, r4, r5, sl}
 80000e4:       00000004        andeq   r0, r0, r4
 80000e8:       00000000        andeq   r0, r0, r0
```

R3 = 0x1800043E
R3/.GOT = R3 + PC(Current instruction address + 4)
R3/.GOT = 0x1800043E + 0x80000C6 = 0x20000504
R2 = offset for **my_shared_var1** addr in .GOT
R2 = Read address of **my_shared_var1** from .GOT
Write 42 in **my_shared_var1**

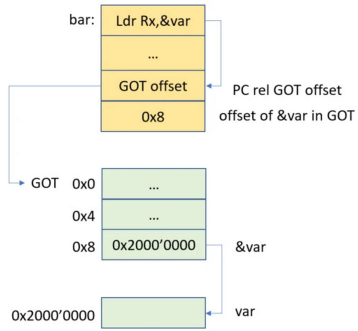
тоже самое для **my_shared_var2**

```
800040c:       f240 1c08       movw    ip, #264        ; 0x108
8000410:       f6c1 0c00       movt    ip, #6144       ; 0x1800
8000414:       44fc            add     ip, pc
8000416:       f8dc f000       ldr.w   pc, [ip]
800041a:       e7fc            b.n     8000416 <.plt+0x3a>
```

IP - Inter-Procedural register - R12
IP = 0x18000108
IP = 0x18000108 + 0x8000418 = 0x20000520 = .GOT.PLT for shared_function
Branch на shared_function через .GOT.PLT

```
DYNAMIC RELOCATION RECORDS
OFFSET   TYPE              VALUE
20000504 R_ARM_GLOB_DAT    my_shared_var2
20000508 R_ARM_GLOB_DAT    my_shared_var1
20000518 R_ARM_JUMP_SLOT   __deregister_frame_info
2000051c R_ARM_JUMP_SLOT   __register_frame_info
20000520 R_ARM_JUMP_SLOT   shared_function
```

```
Disassembly of section .got:

20000504 <.got>:
20000504:       00000000        an
20000508:       00000000        an

Disassembly of section .got.plt:

2000050c <_GLOBAL_OFFSET_TABLE_>:
2000050c:       20000434        an
20000510:       00000000        an
20000514:       00000000        an
20000518:       080003dd        st
2000051c:       080003dd        st
20000520:       080003dd        st
```

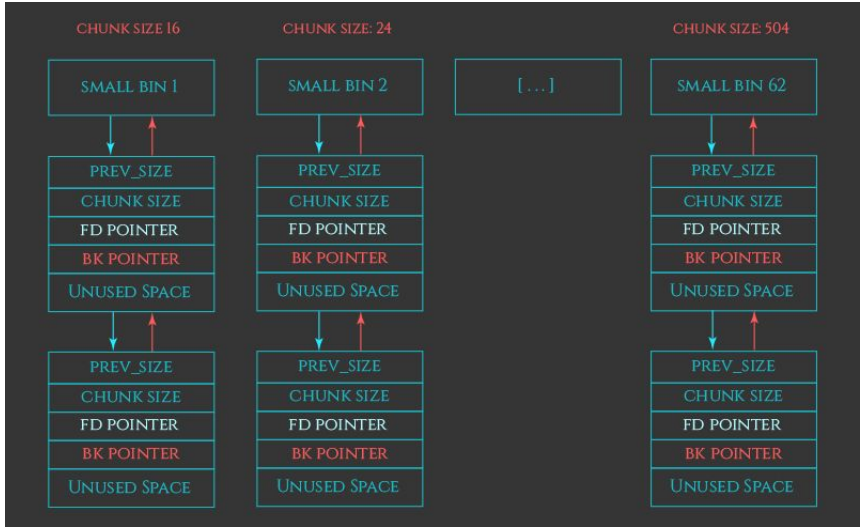


https://github.com/rqujju/STM32-projects/tree/master/got_plt

There are 5 type of bins:

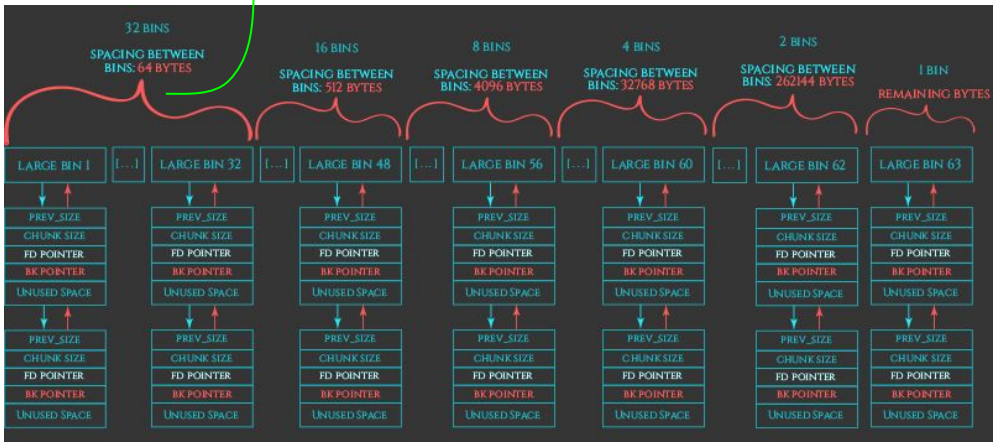
- *62 small bins - stores chunks that are all the **same fixed size** (before 512 bytes);*
- *63 large bins - stores chunks **within a size range;***
- *1 unsorted bin - optimization is based on the observation that often frees are clustered together, and frees are often immediately followed by allocations of similarly sized chunks*
- *10 fast bins - optimization is based on the observation that often frees are clustered together, but for small bins*
- *64 tcache bins per thread - per thread bin queue*

small bins

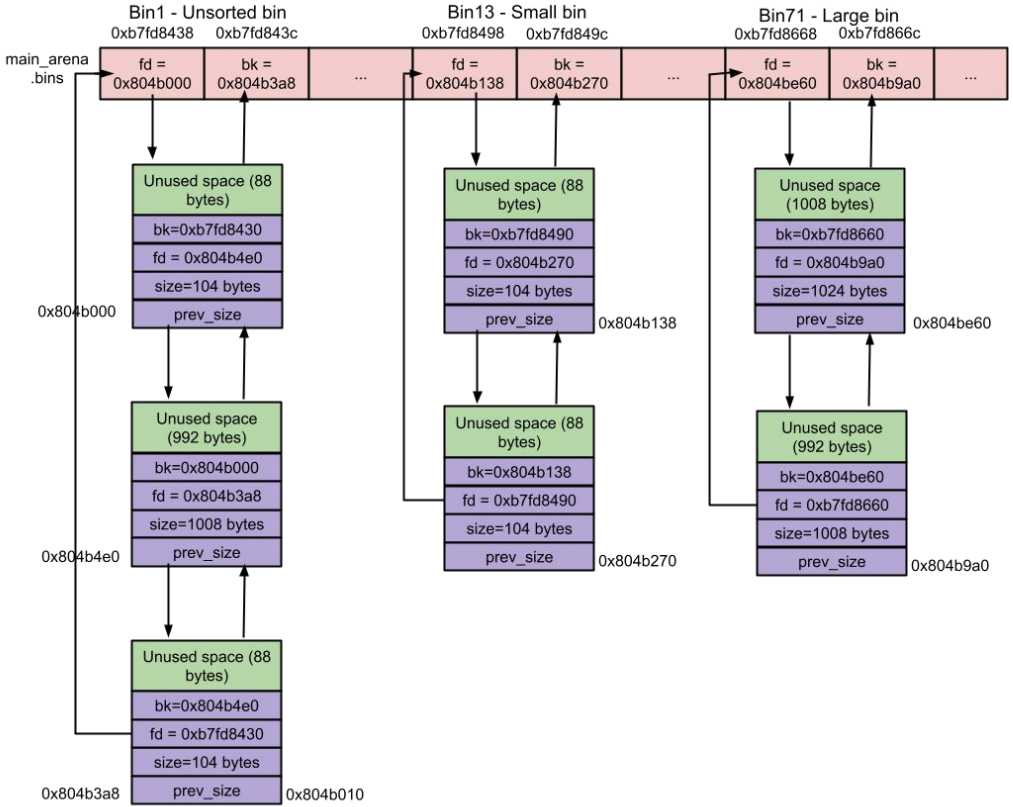


Large BIN-1 chunk size range:
512 bytes - 675 bytes (+64)
Large BIN-2 chunk size range:
675 bytes - 739 bytes (+64)
etc…

large bins



https://azeria-labs.com/heap-exploitation-part-2-glibc-heap-free-bins/



Unsorted, Small and Large Bin Snapshot

Appendix: glibc malloc

https://sploitfun.wordpress.com/2015/02/10/understanding-glibc-malloc/