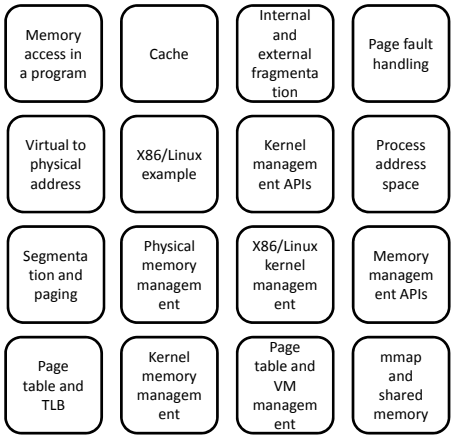
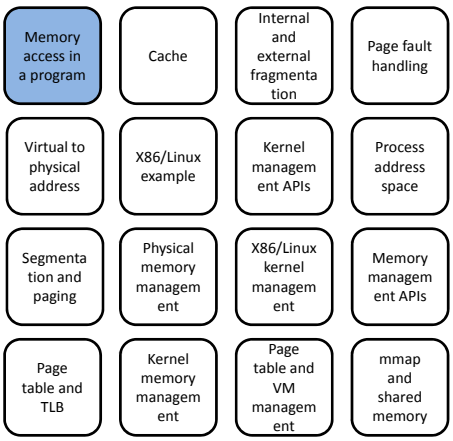
Operating System Design and Implementation

Memory Management – Part I

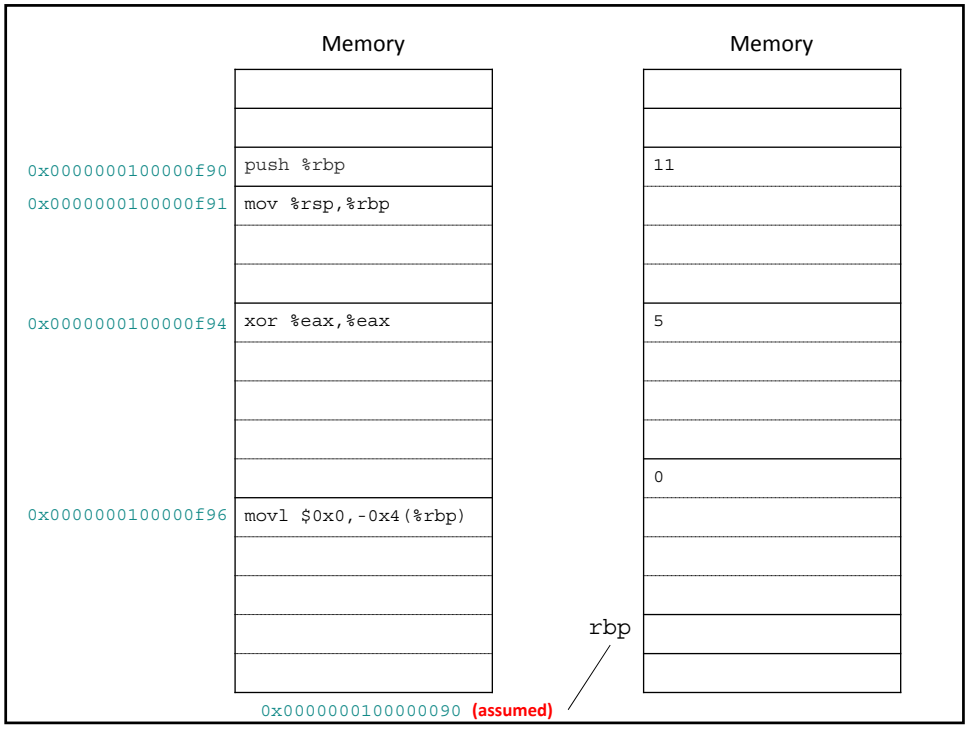
Shiao-Li Tsao

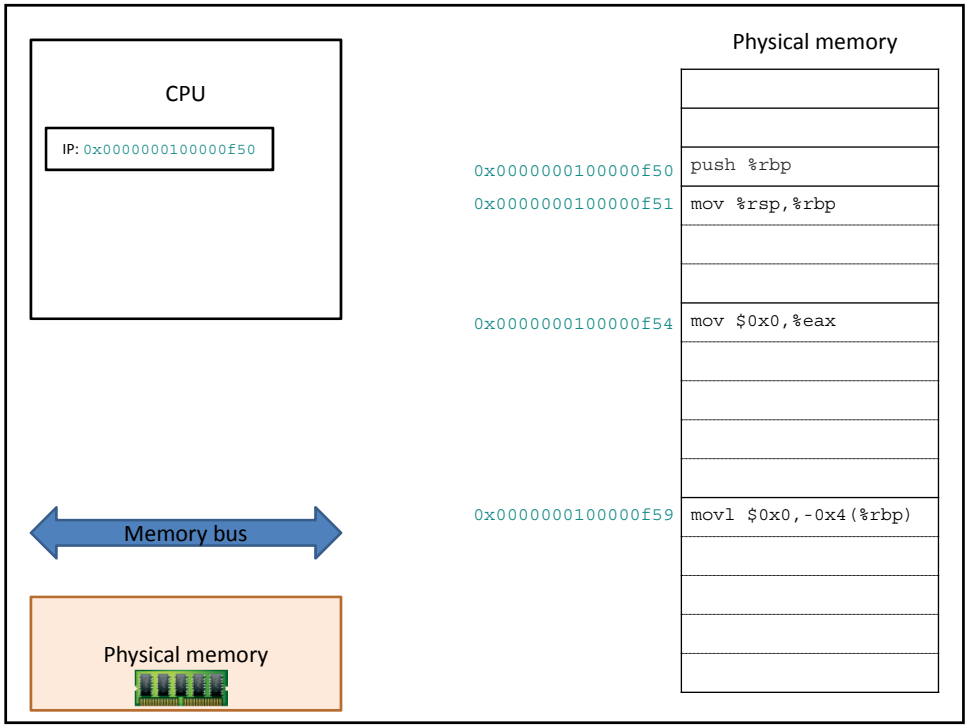





```c
int main()
{
    int a = 5;
    int b = a + 6;
    return 0;
}
```

```
0000000100000f90 <_main>:
   100000f90:   55                      push   %rbp
   100000f91:   48 89 e5                mov    %rsp,%rbp
   100000f94:   31 c0                   xor    %eax,%eax
   100000f96:   c7 45 fc 00 00 00 00    movl   $0x0,-0x4(%rbp)
   100000f9d:   c7 45 f8 05 00 00 00    movl   $0x5,-0x8(%rbp)
   100000fa4:   8b 4d f8                mov    -0x8(%rbp),%ecx
   100000fa7:   83 c1 06                add    $0x6,%ecx
   100000faa:   89 4d f4                mov    %ecx,-0xc(%rbp)
   100000fad:   5d                      pop    %rbp
   100000fae:   c3                      retq
```


Memory
Memory
0x0000000100000f90
push %rbp
0x0000000100000f91
mov %rsp,%rbp
0x0000000100000f94
xor %eax,%eax
0x0000000100000f96
movl $0x0,-0x4(%rbp)
11
5
0
rbp
0x0000000100000090 (assumed)

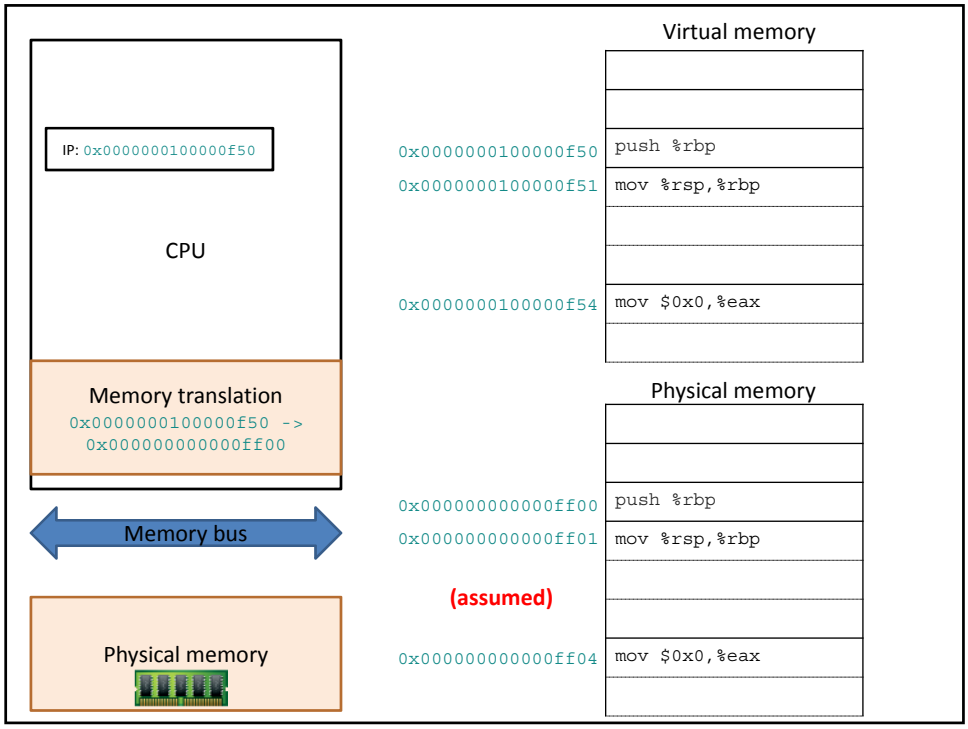

CPU
IP: 0x0000000100000f50
Memory bus
Physical memory
Physical memory
0x0000000100000f50
push %rbp
0x0000000100000f51
mov %rsp,%rbp
0x0000000100000f54
mov $0x0,%eax
0x0000000100000f59
movl $0x0,-0x4(%rbp)

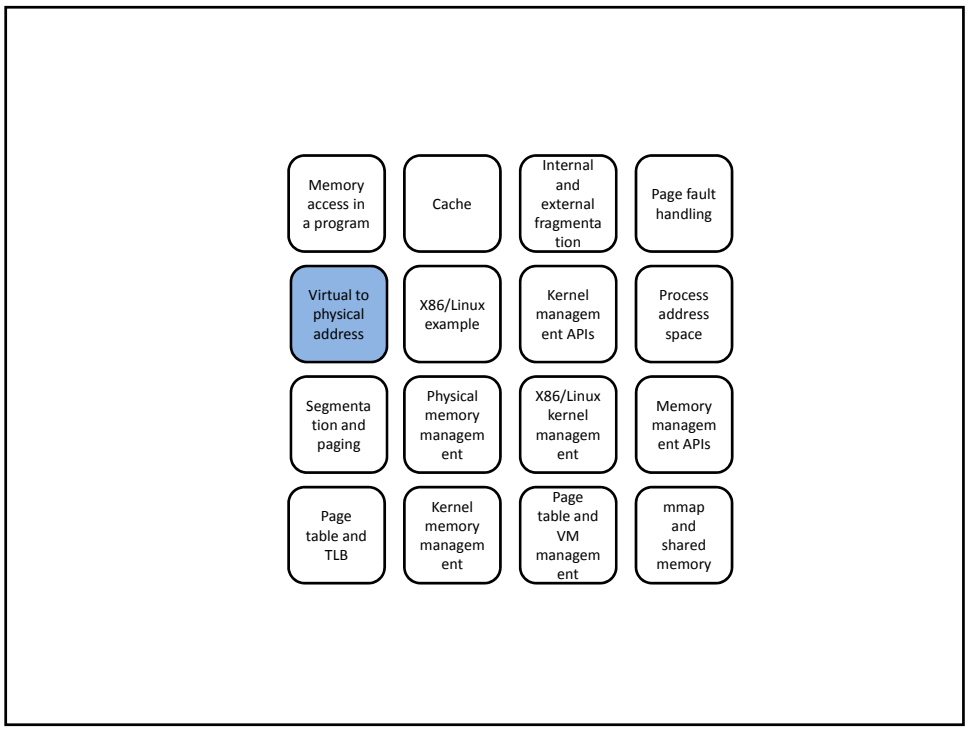
Physical Memory vs. Virtual Memory

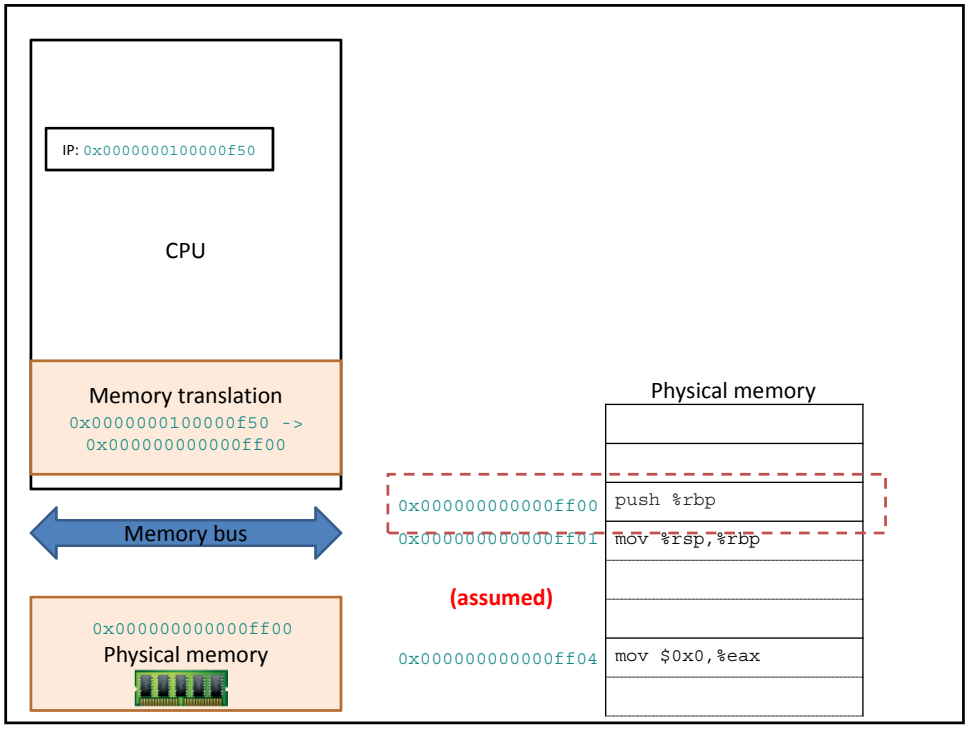
- Why not physical memory?
- Benefits for using virtual memory
  - Physical memory shared by multiple processes
  - Isolate processes from each other
  - Large/contiguous per process address space
  - Use memory more efficiently
- Disadvantage for using virtual memory
  - Expensive and slow
- *What kind of memory will you use if we only have single process, or you are the only developer of all processes, or you need very fast memory accesses*

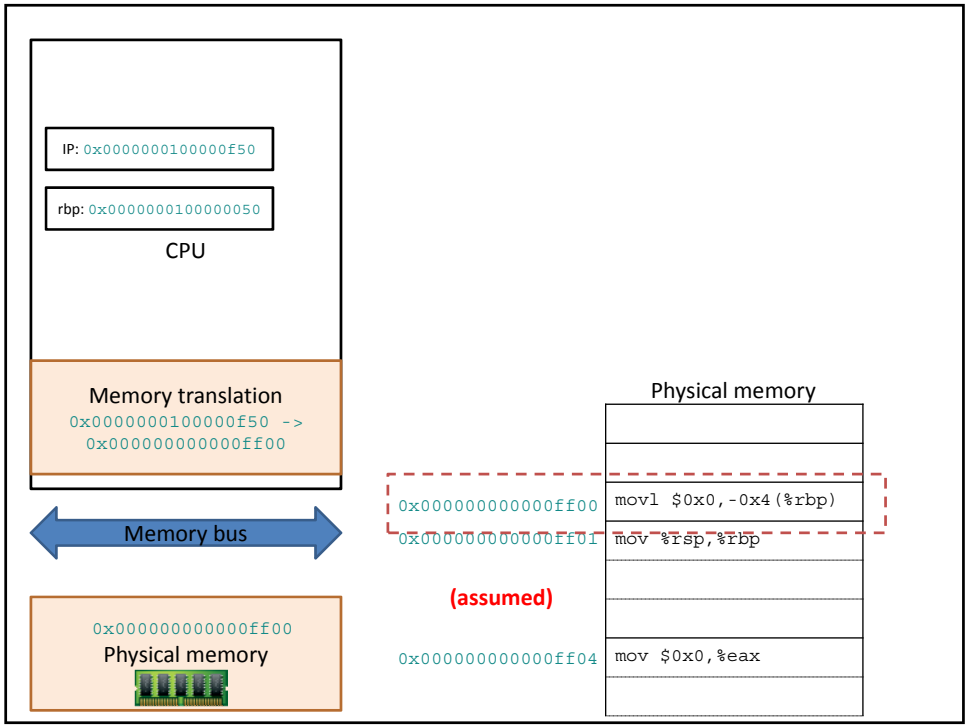
Virtual addresses

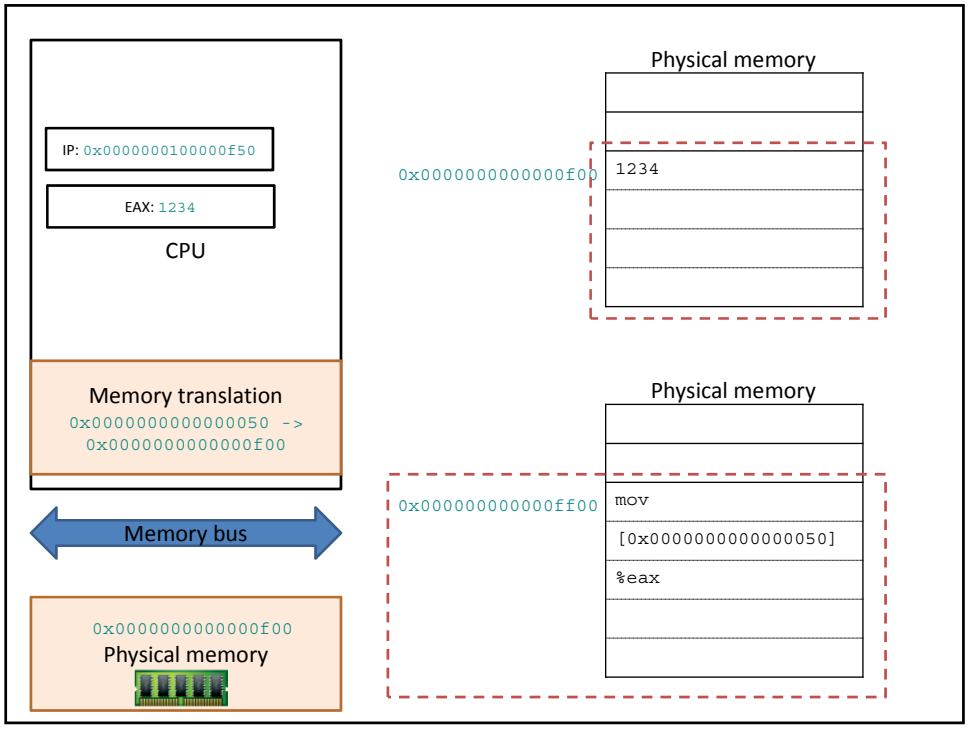
- The memory addresses users and compiler believed
- The code/data are stored in physical memory
- The virtual memory addresses are translated into physical memory addresses (by CPU) and then CPU can access the code/data
- How can a CPU translate the addresses?
  - Offset? Table lookup?

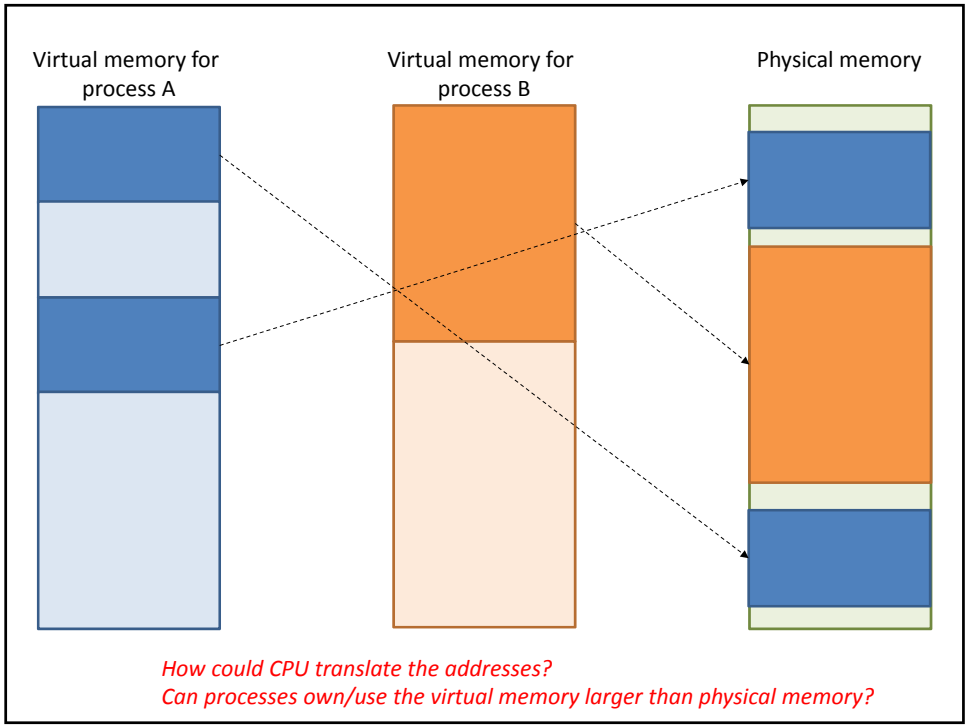

Virtual memory
IP: 0x0000000100000f50
CPU
0x0000000100000f50
push %rbp
0x0000000100000f51
mov %rsp,%rbp
0x0000000100000f54
mov $0x0,%eax
Memory translation
0x0000000100000f50 ->
0x000000000000ff00
Physical memory
Memory bus
0x000000000000ff00
push %rbp
0x000000000000ff01
mov %rsp,%rbp
(assumed)
Physical memory
0x000000000000ff04
mov $0x0,%eax

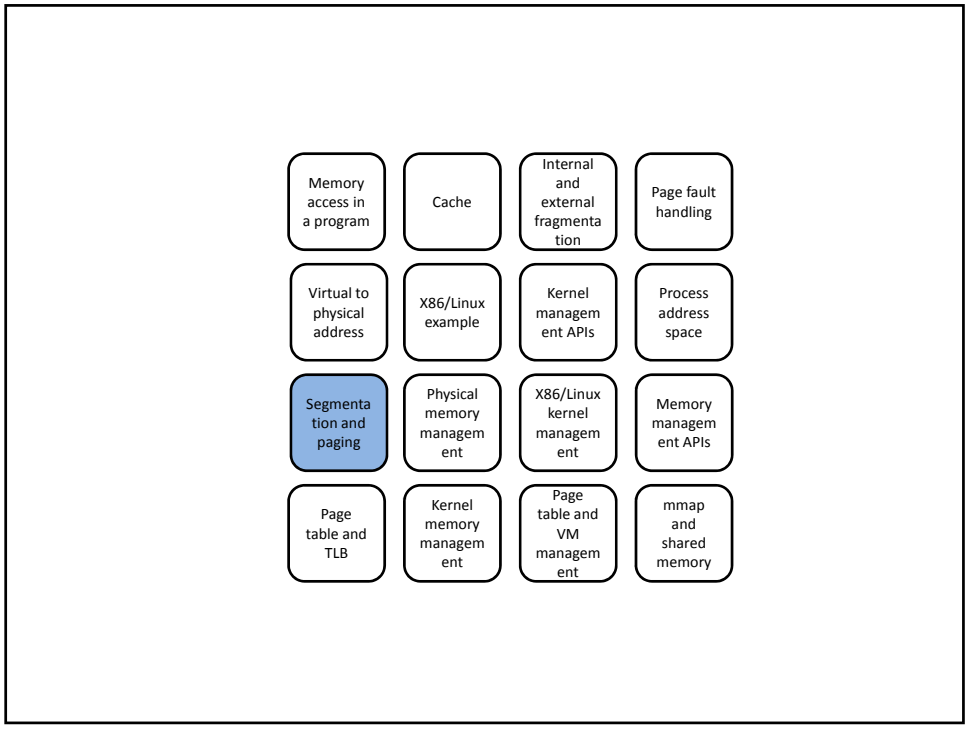

Memory access in a program
Cache
Internal and external fragmentation
Page fault handling
Virtual to physical address
X86/Linux example
Kernel management APIs
Process address space
Segmentation and paging
Physical memory management
X86/Linux kernel management
Memory management APIs
Page table and TLB
Kernel memory management
Page table and VM management
mmap and shared memory

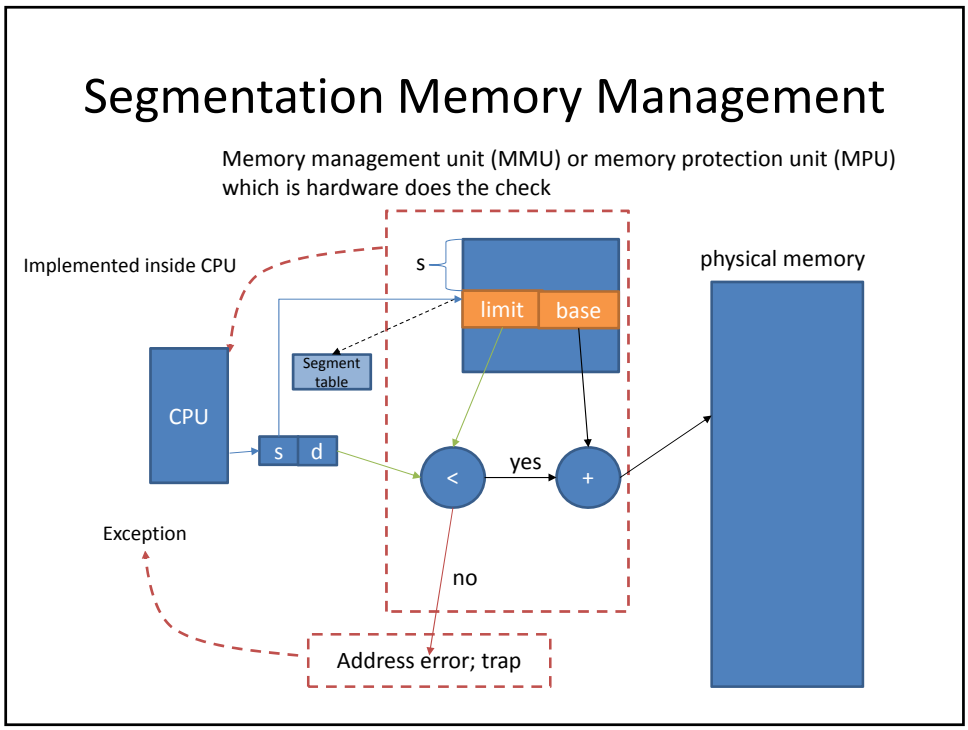

IP: 0x0000000100000f50
CPU
Memory translation
0x0000000100000f50 ->
0x000000000000ff00
Memory bus
0x000000000000ff00
Physical memory
Physical memory
0x000000000000ff00
push %rbp
0x000000000000ff01
mov %rsp,%rbp
(assumed)
0x000000000000ff04
mov $0x0,%eax


IP: 0x0000000100000f50
rbp: 0x0000000100000050
CPU
Memory translation
0x0000000100000f50 ->
0x000000000000ff00
Memory bus
0x000000000000ff00
Physical memory
Physical memory
0x000000000000ff00
movl $0x0,-0x4(%rbp)
0x000000000000ff01
mov %rsp,%rbp
(assumed)
0x000000000000ff04
mov $0x0,%eax


IP: 0x0000000100000f50
EAX: 1234
CPU
Memory translation
0x0000000000000050 ->
0x0000000000000f00
Memory bus
0x0000000000000f00
Physical memory
Physical memory
0x0000000000000f00
1234
Physical memory
0x0000000000000ff00
mov
[0x0000000000000050]
%eax


Virtual memory for process A
Virtual memory for process B
Physical memory
How could CPU translate the addresses?
Can processes own/use the virtual memory larger than physical memory?


Memory access in a program
Cache
Internal and external fragmenta tion
Page fault handling
Virtual to physical address
X86/Linux example
Kernel managem ent APIs
Process address space
Segmenta tion and paging
Physical memory managem ent
X86/Linux kernel managem ent
Memory managem ent APIs
Page table and TLB
Kernel memory managem ent
Page table and VM managem ent
mmap and shared memory


Segmentation Memory Management
Memory management unit (MMU) or memory protection unit (MPU) which is hardware does the check
Implemented inside CPU
s
limit
base
physical memory
Segment table
CPU
s
d
<
yes
+
Exception
no
Address error; trap

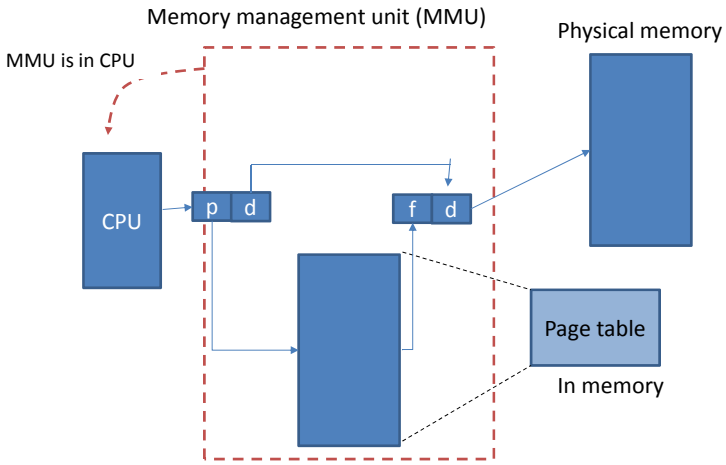
Segmentation Memory Management (example)

segment	limit	base
3	0x2000	0x1234
1	0x1234	0x1200

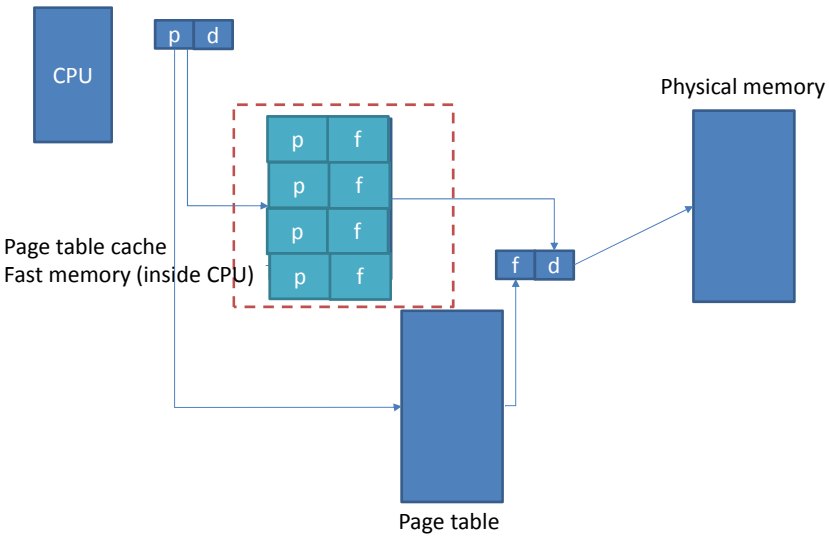
Read/write, … (such as ring level)

Read only

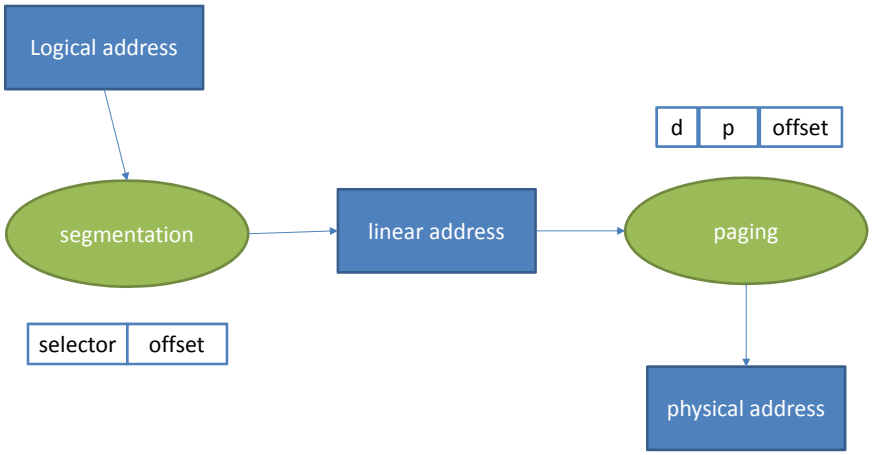
Page memory management

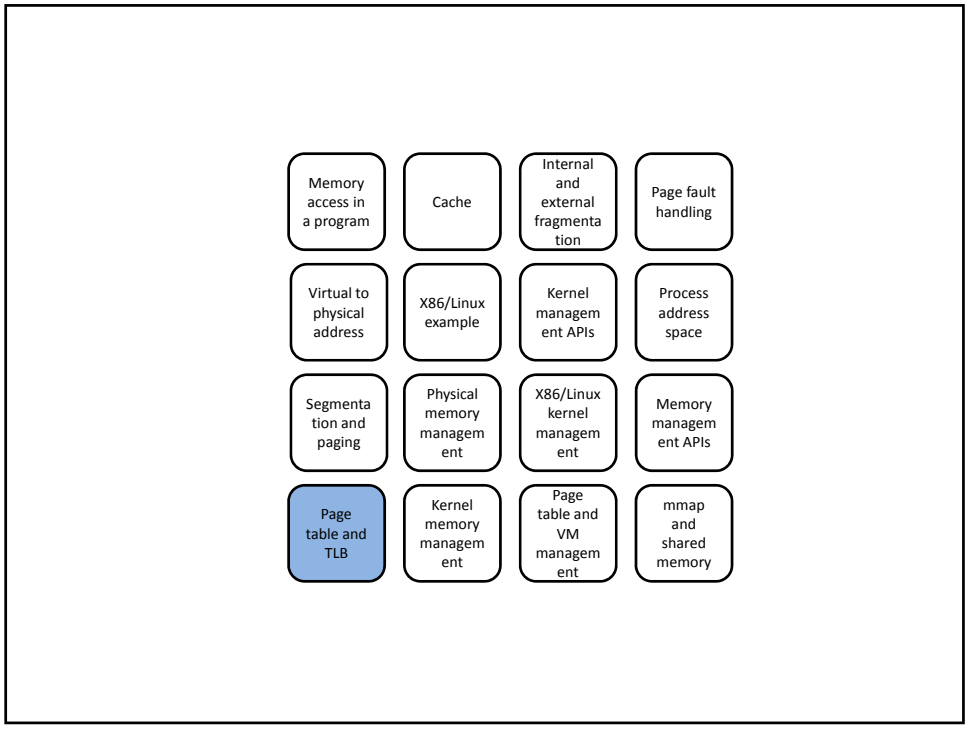


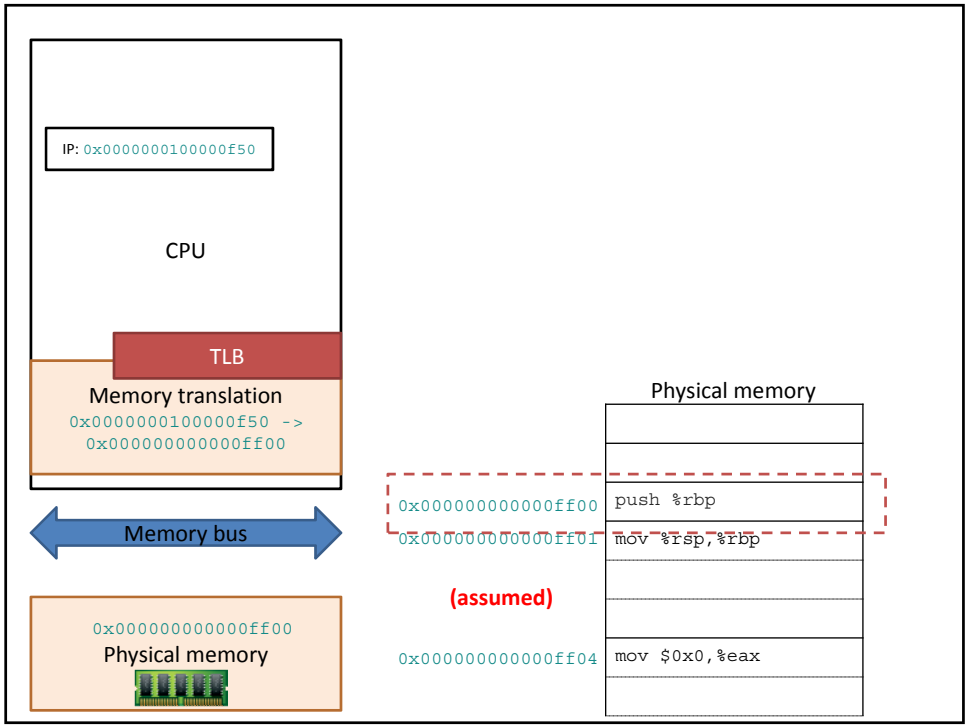
Paging with TLB



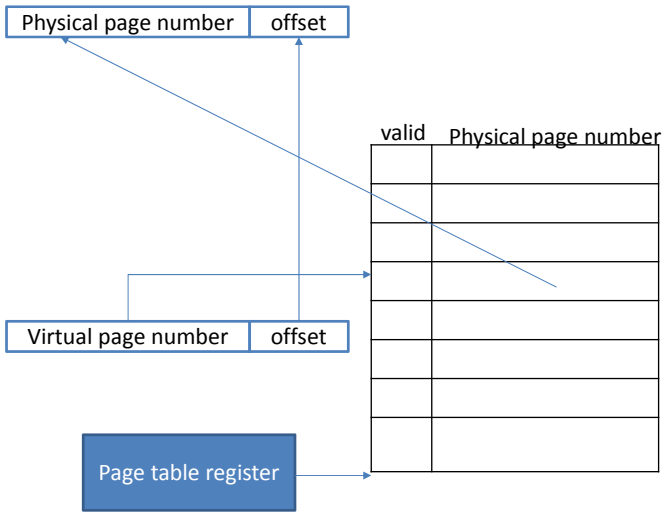
Segmentation with paging (x86 example)



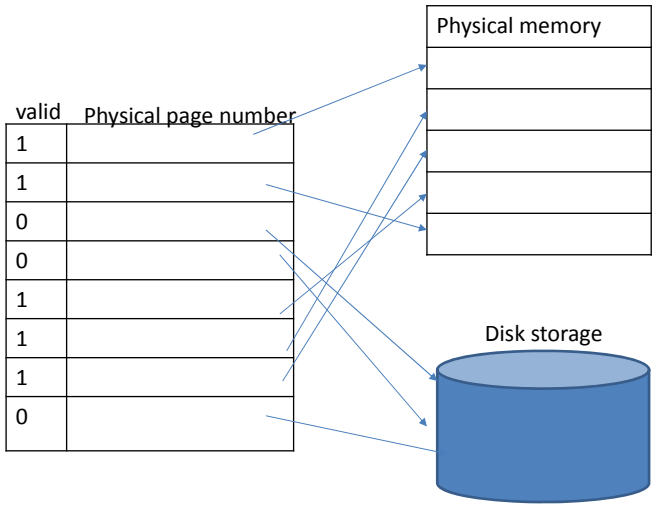

Memory access in a program
Cache
Internal and external fragmenta tion
Page fault handling
Virtual to physical address
X86/Linux example
Kernel managem ent APIs
Process address space
Segmenta tion and paging
Physical memory managem ent
X86/Linux kernel managem ent
Memory managem ent APIs
Page table and TLB
Kernel memory managem ent
Page table and VM managem ent
mmap and shared memory

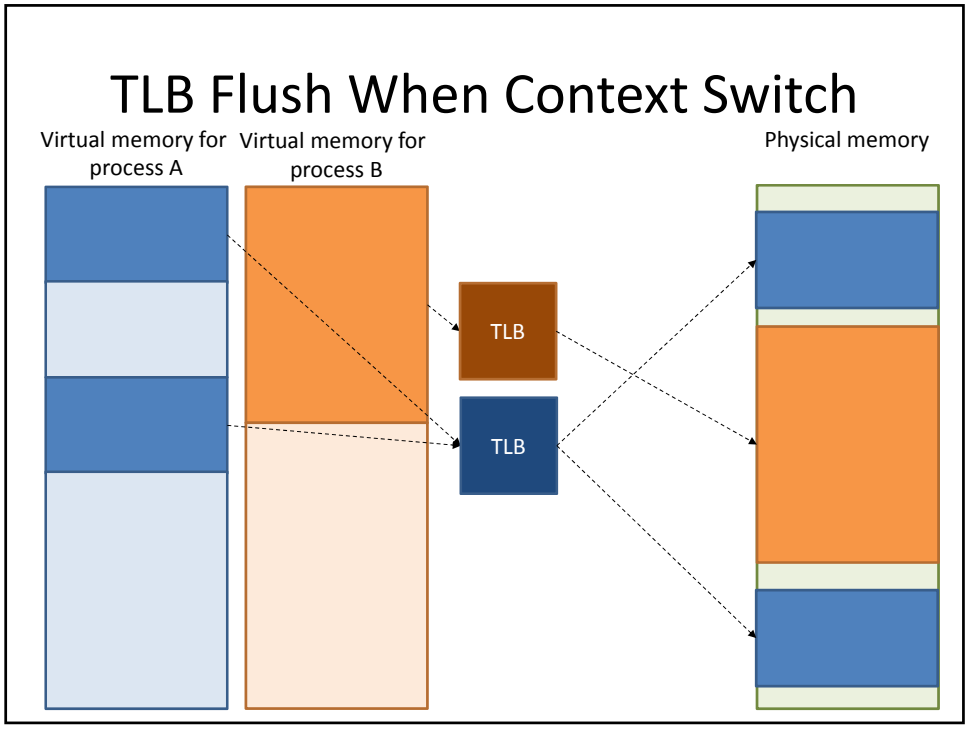

IP: 0x0000000100000f50
CPU
TLB
Memory translation
0x0000000100000f50 ->
0x000000000000ff00
Memory bus
0x000000000000ff00
Physical memory
Physical memory
0x000000000000ff00
push %rbp
0x000000000000ff01
mov %rsp,%rbp
(assumed)
0x000000000000ff04
mov $0x0,%eax

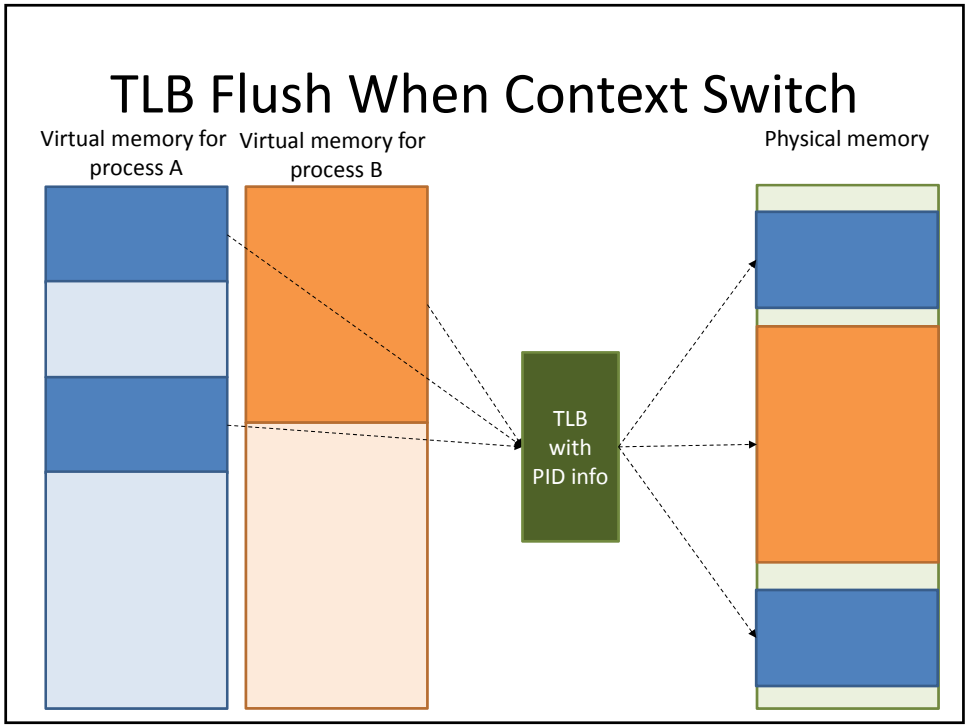
Page Table

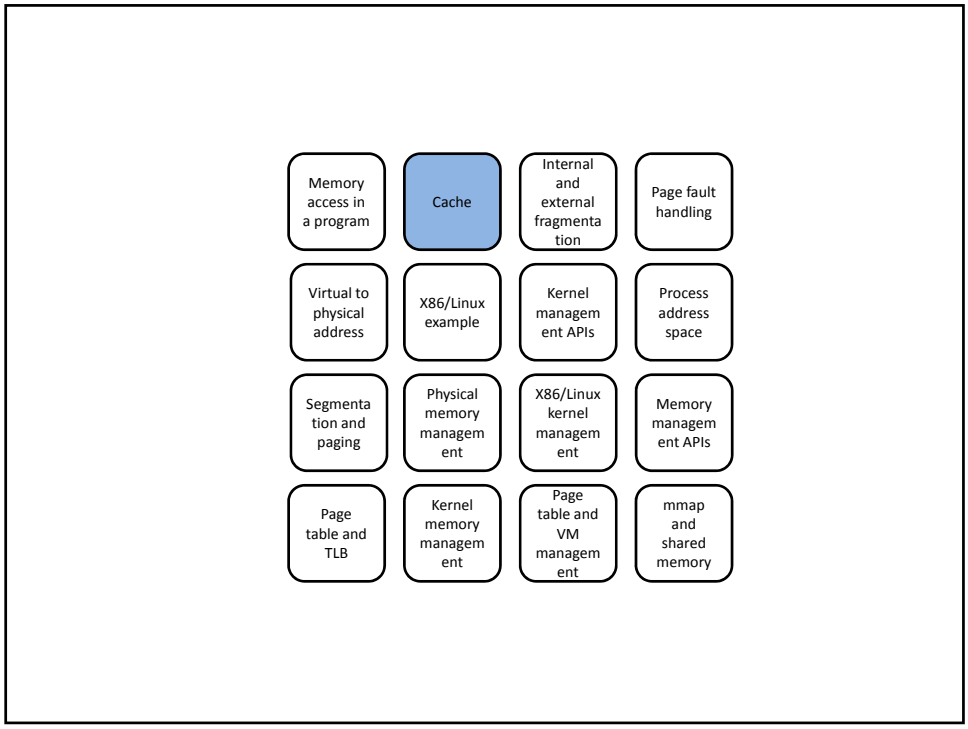


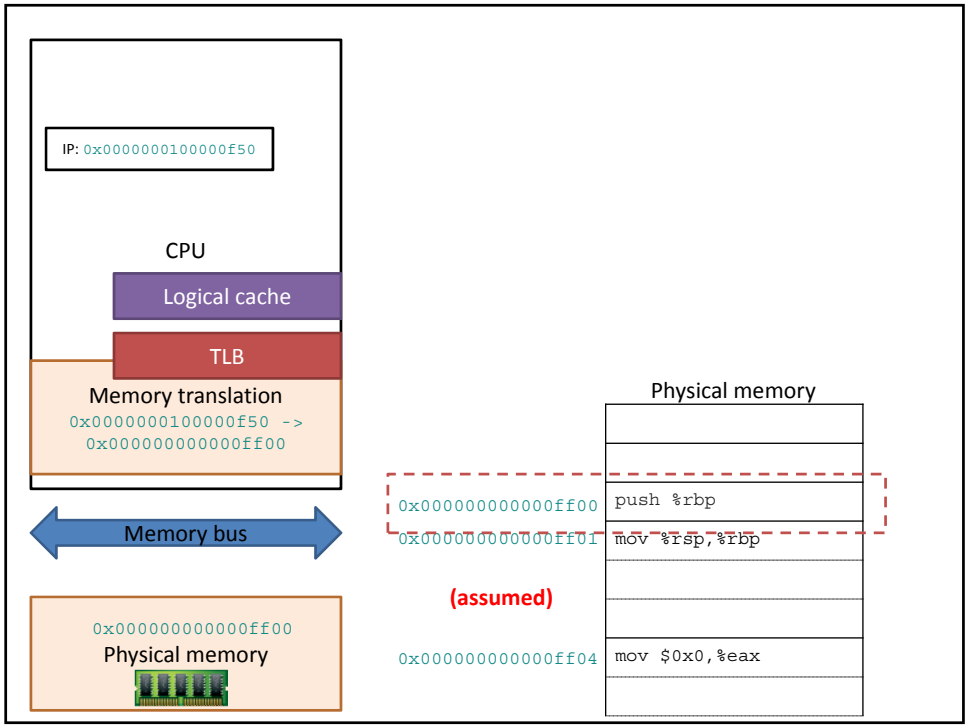
Page Table and Virtual Address

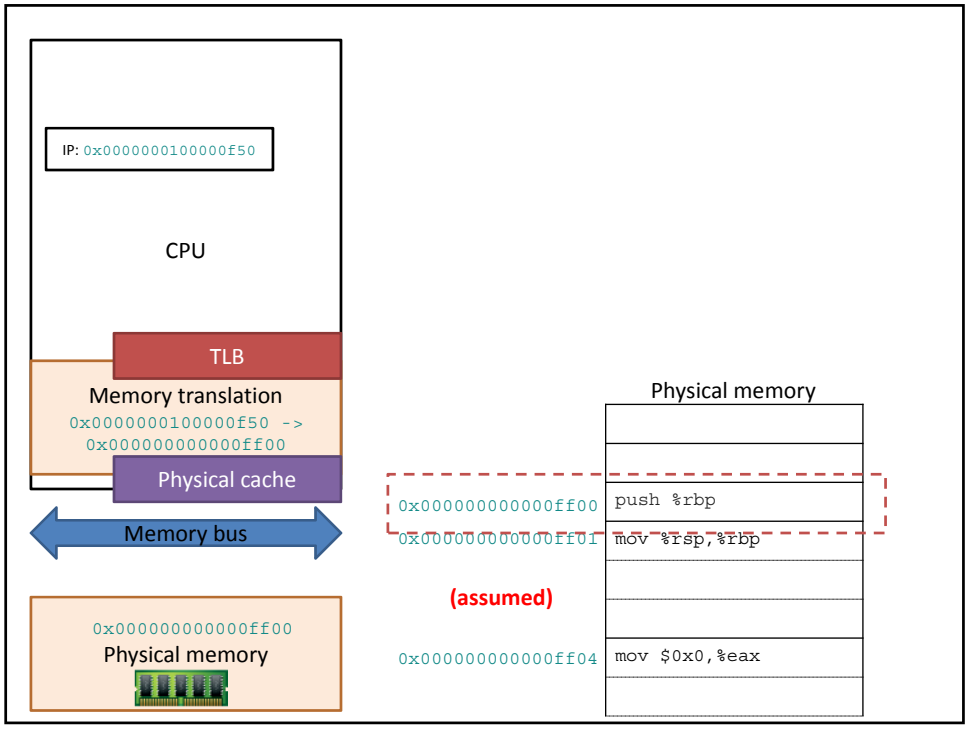



TLB Flush When Context Switch
Virtual memory for process A
Virtual memory for process B
Physical memory
TLB
TLB

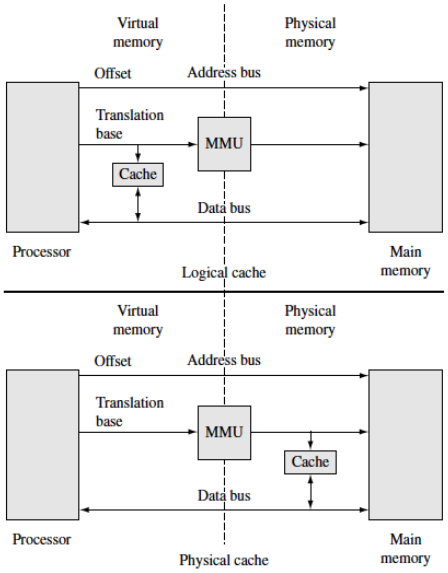

TLB Flush When Context Switch
Virtual memory for process A
Virtual memory for process B
Physical memory
TLB with PID info


Memory access in a program
Cache
Internal and external fragmenta tion
Page fault handling
Virtual to physical address
X86/Linux example
Kernel managem ent APIs
Process address space
Segmenta tion and paging
Physical memory managem ent
X86/Linux kernel managem ent
Memory managem ent APIs
Page table and TLB
Kernel memory managem ent
Page table and VM managem ent
mmap and shared memory


IP: 0x0000000100000f50
CPU
Logical cache
TLB
Memory translation
0x0000000100000f50 ->
0x000000000000ff00
Memory bus
0x000000000000ff00
Physical memory
Physical memory
0x000000000000ff00
push %rbp
0x000000000000ff01
mov %rsp,%rbp
(assumed)
0x000000000000ff04
mov $0x0,%eax



Logical and Physical Memory



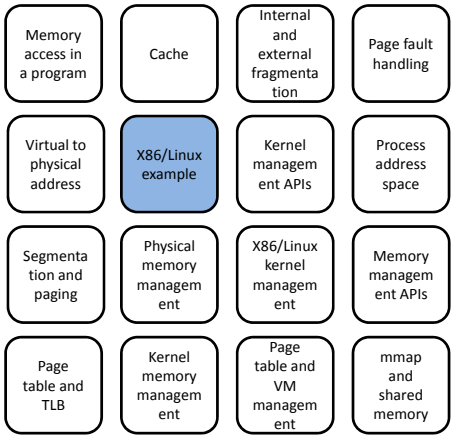
ARM7-ARM10

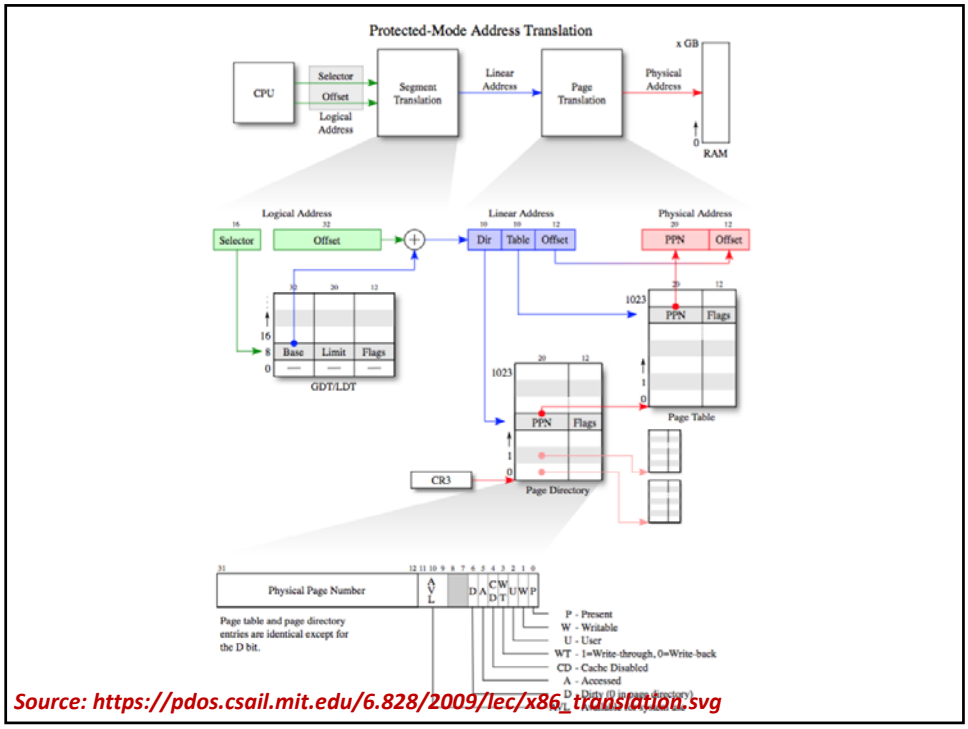
ARM11

Logical and Physical Memory

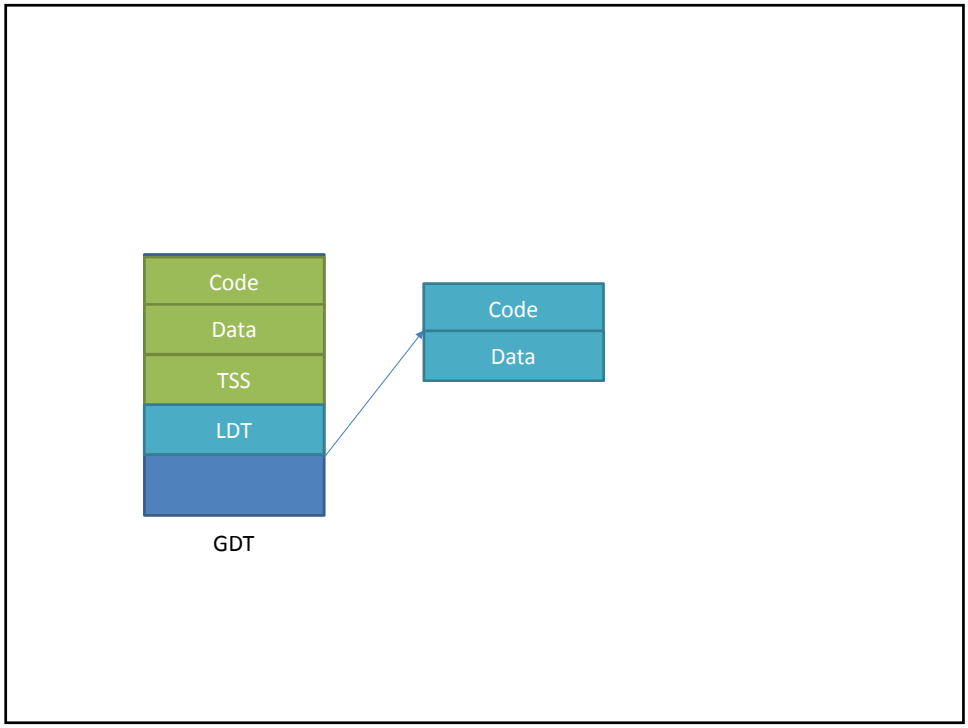
- Physical Caches
  - Slow but share without flush
- Logical Caches
  - Pid or flush while context switch

31



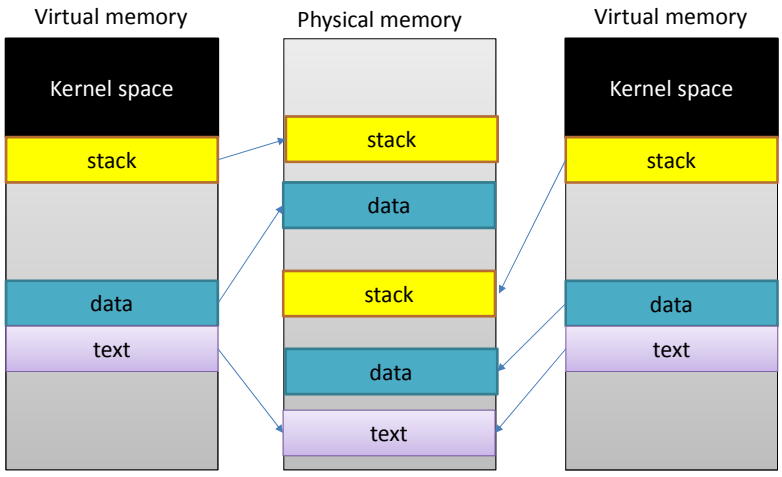


Source: https://pdos.csail.mit.edu/6.828/2009/lec/x86_translation.svg

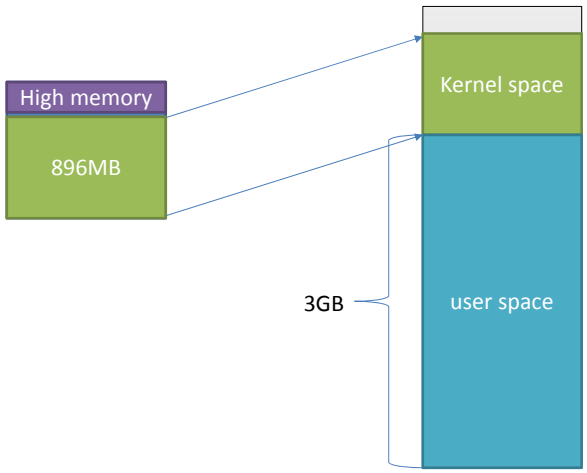


User Address Space

- 4G for 32 bits processor
- How about kernel?
  - Kernel uses another 4G address space ?
  - Context switch for system calls?
- How about user and kernel share the same 4G spaces?
  - User program can use *X*
  - Kernel program can use *4G-X*
- Can user accesses the kernel data?
  - Limited by user data segment
- Can kernel access the user data?
  - Kernel data segment covers 4G

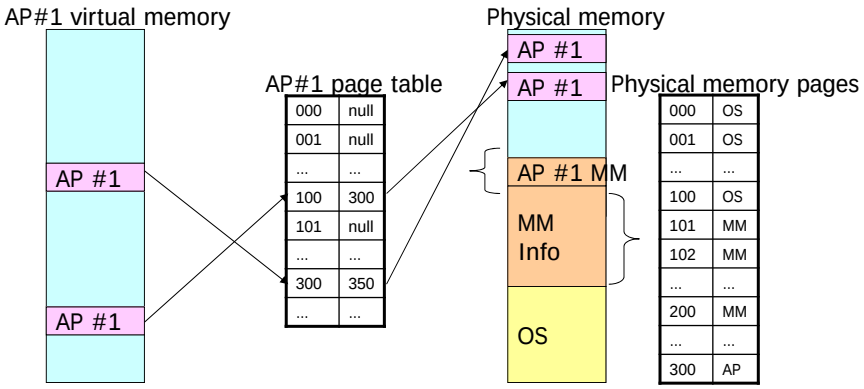


Direct linear mapping



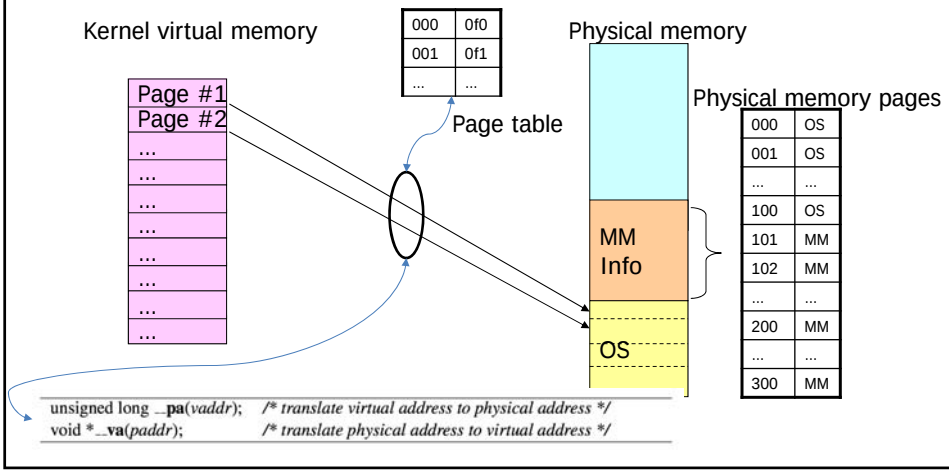
MM Basics (Cont.)

- Management physical memory (page)



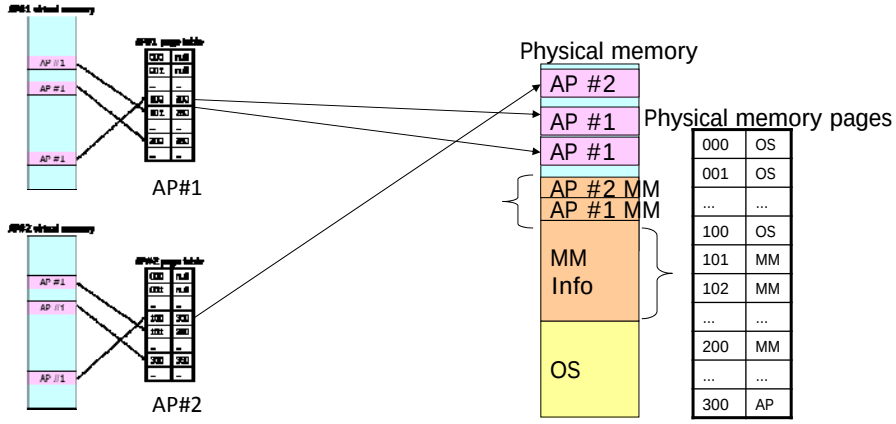
MM Basics (Cont.)

- Kernel memory allocation/release

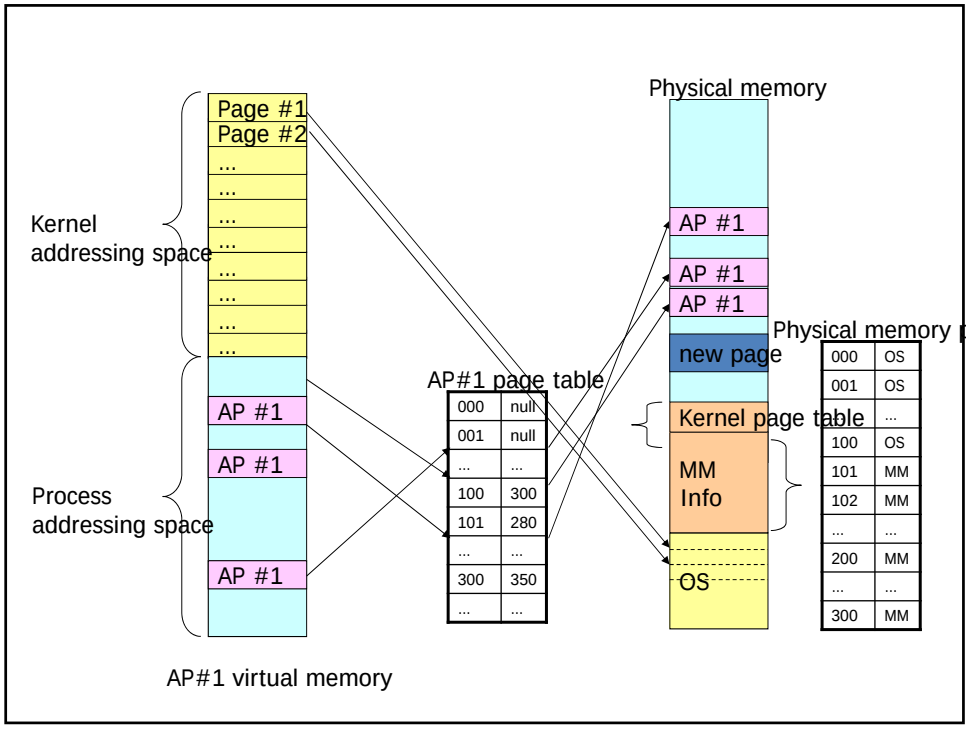


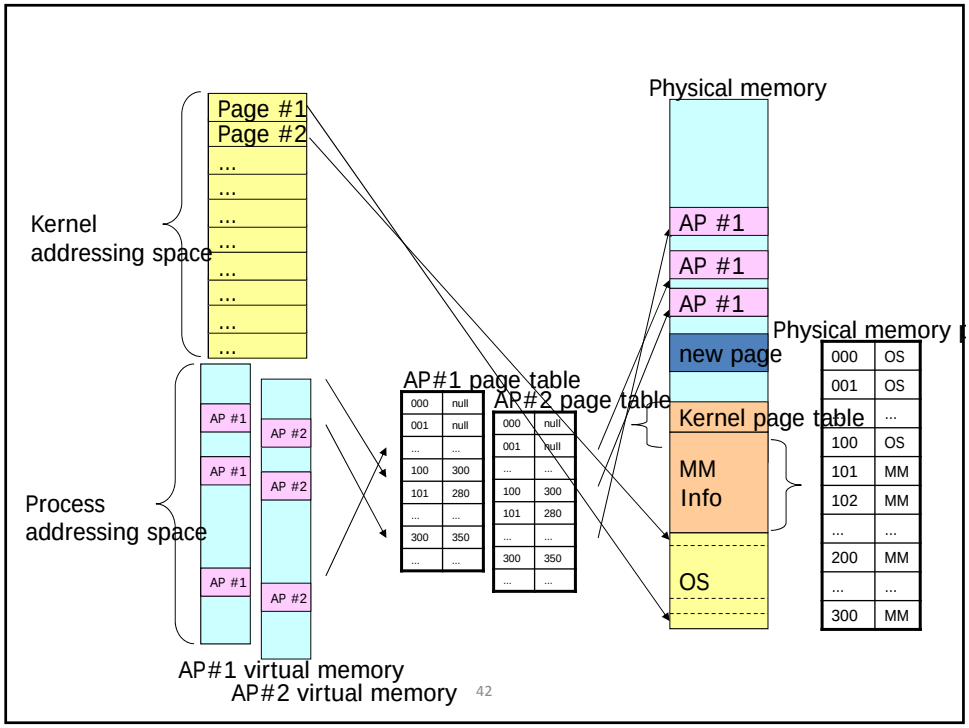
MM Basics (Cont.)

- application memory allocation/release



40


Physical memory
Page #1
Page #2
...
Kernel
addressing space
Process
addressing space
AP #1
AP #1
AP #1
AP#1 page table
000 null
001 null
... ...
100 300
101 280
... ...
300 350
... ...
AP #1
AP #1
AP #1
new page
Kernel page table
MM
Info
OS
Physical memory
000 OS
001 OS
... ...
100 OS
101 MM
102 MM
... ...
200 MM
... ...
300 MM
AP#1 virtual memory


Physical memory
Page #1
Page #2
...
Kernel
addressing space
Process
addressing space
AP #1
AP #1
AP #2
AP #2
AP #1
AP #2
AP#1 page table
AP#2 page table
000 null
001 null
... ...
100 300
101 280
... ...
300 350
... ...
AP #1
AP #1
AP #1
new page
Kernel page table
MM
Info
OS
Physical memory
000 OS
001 OS
... ...
100 OS
101 MM
102 MM
... ...
200 MM
... ...
300 MM
AP#1 virtual memory
AP#2 virtual memory
42