Processes and Addresses

```
#include "csapp.h"

int main() {
  int *x = malloc(sizeof(int));
  *x = 10;
  if (Fork() == 0)
    printf("%d\n", *x);
  else {
    *x = 20;
    printf("%d\n", *x);
  }

  return 1;
}
```

Copy

Processes and Addresses

```
#include "csapp.h"

int main() {
  int *x = malloc(sizeof(int));
  *x = 10;
  if (Fork() == 0)
    printf("%d\n", *x);
  else {
    *x = 20;
    printf("%d\n", *x);
  }

  return 1;
}
```

Copy

Prints 10 and 20 in either order

Processes and Addresses

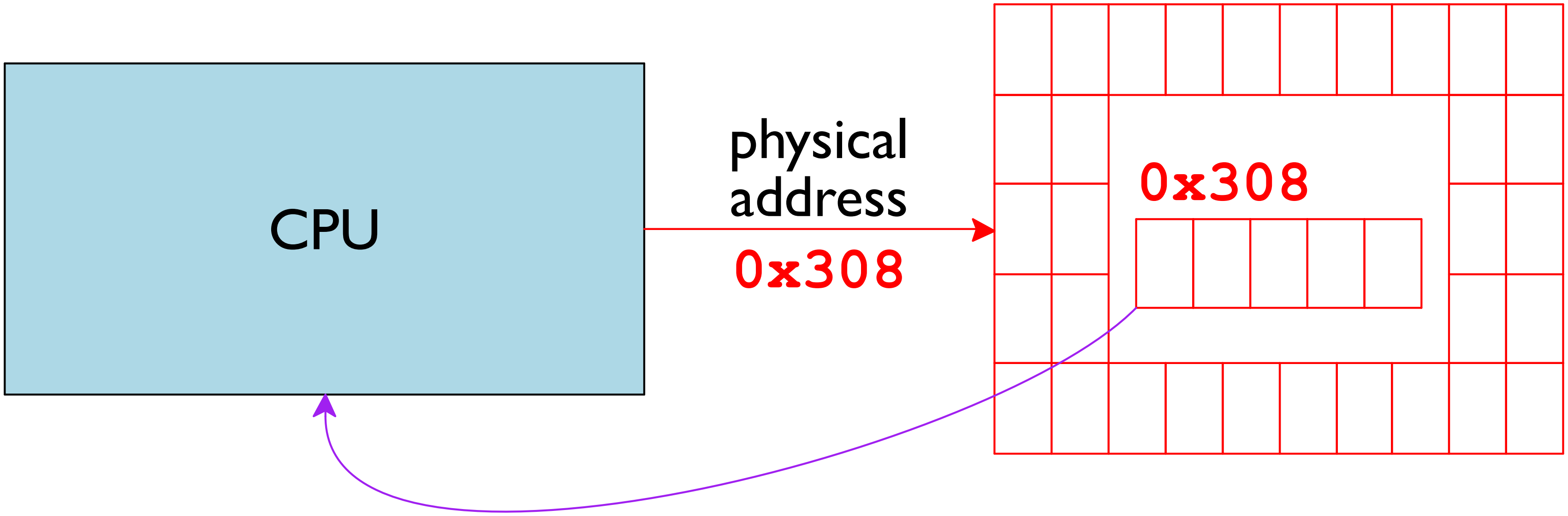
```
#include "csapp.h"

int main() {
  int *x = malloc(sizeof(int));
  *x = 10;
  if (Fork() == 0)
    printf("%p\n", x);
  else {
    *x = 20;
    printf("%p\n", x);
  }

  return 1;
}
```

Processes and Addresses

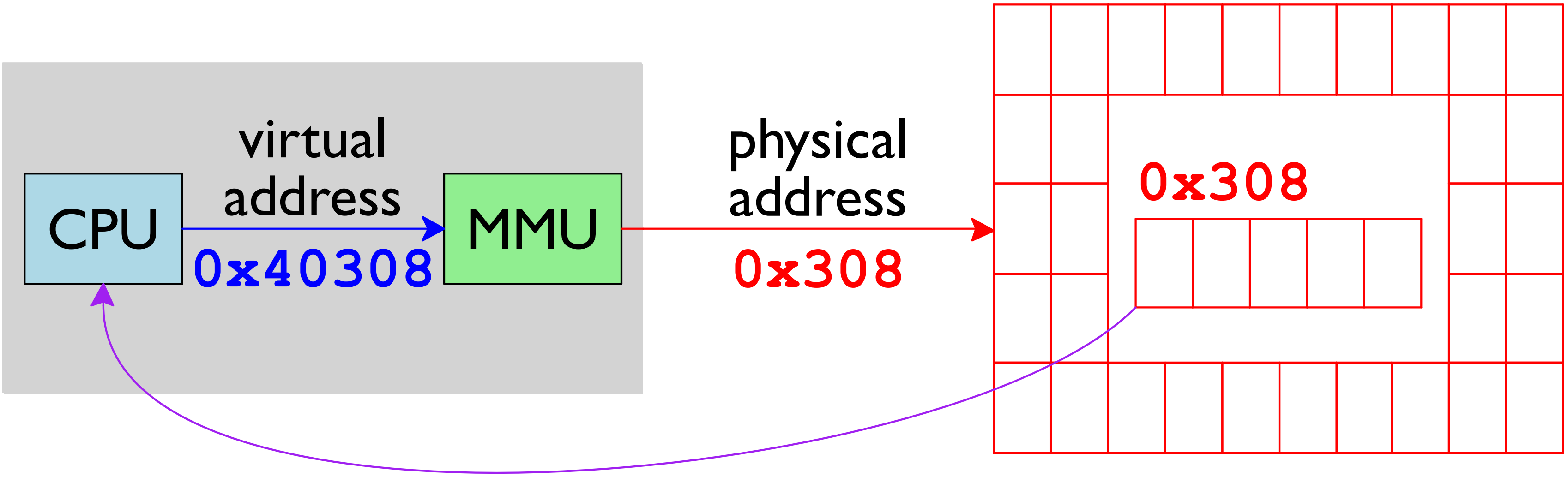
```
#include "csapp.h"

int main() {
  int *x = malloc(sizeof(int));
  *x = 10;
  if (Fork() == 0)
    printf("%p\n", x);
  else {
    *x = 20;
    printf("%p\n", x);
  }

  return 1;
}
```

Prints the same address twice

Physical vs. Virtual Addresses



Physical vs. Virtual Addresses



Virtual Memory Benefits

✓ Isolates processes

Virtual Memory Benefits

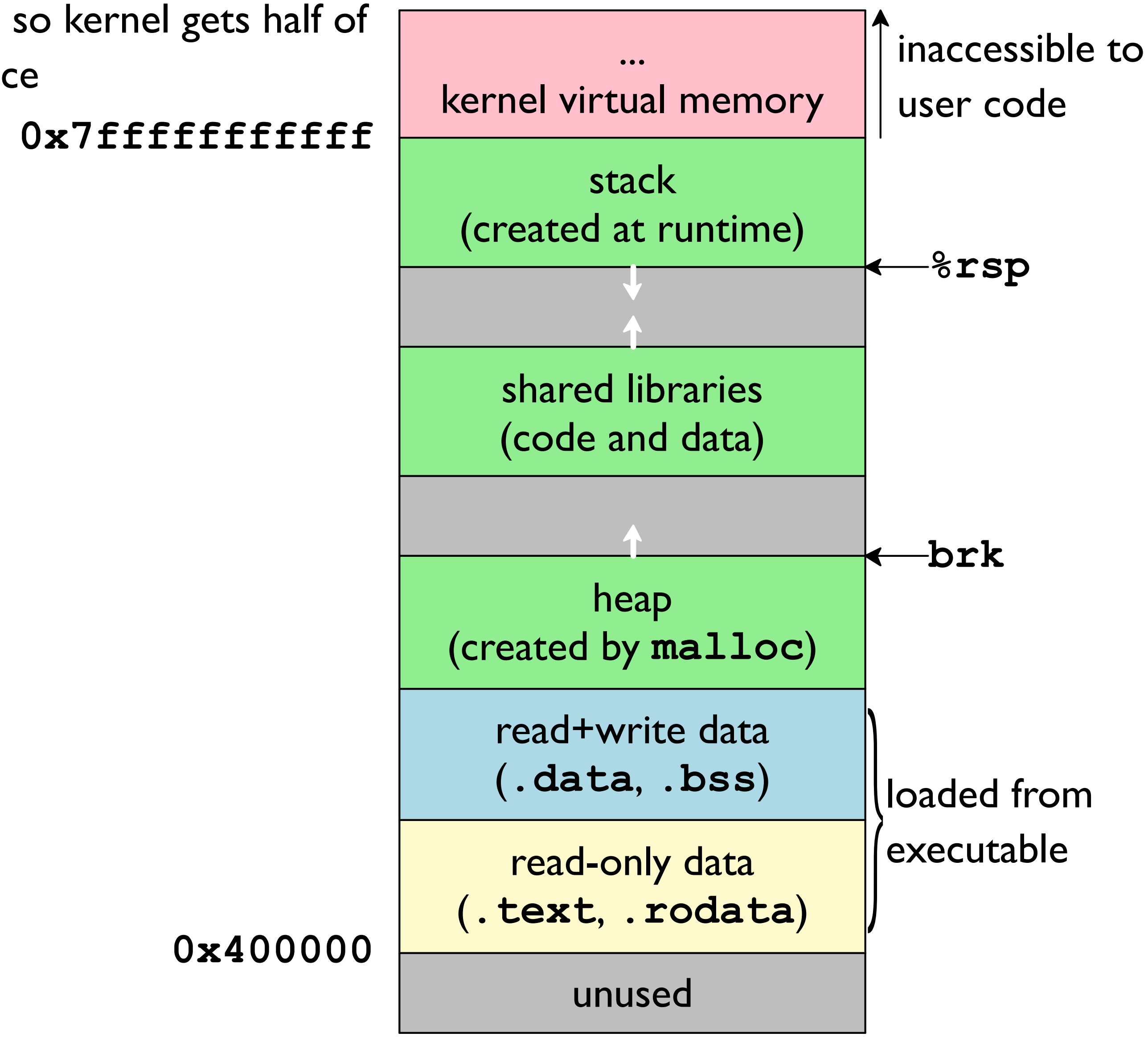
✓ Isolates processes

✓ Simplifies memory management

each process gets the uniform address space

Every Process's View of Memory

x86_64 supports only 48-bit addresses, so kernel gets half of virtual space



Bryant and O'Hallaron, *Computer Systems: A Programmer's Perspective*, Third Edition

Virtual Memory Benefits

✓ Isolates processes

✓ Simplifies memory management

each process gets the uniform address space

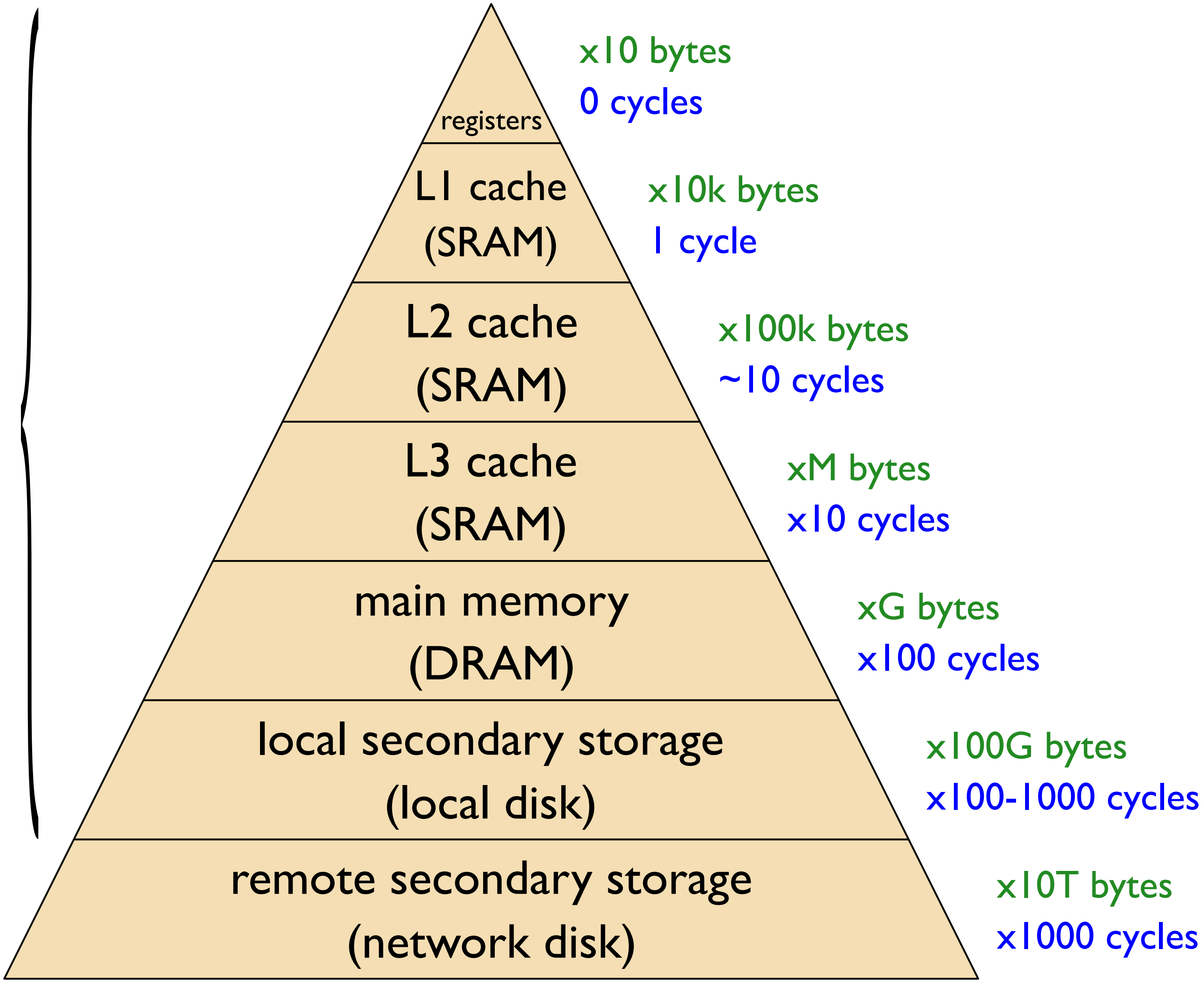
✓ Allows memory content to span devices

... especially main memory and disk

virtual address range ≫ physical memory

Memory Hierarchy

Uniform access via virtual addresses



registers — x10 bytes, 0 cycles

L1 cache (SRAM) — x10k bytes, 1 cycle

L2 cache (SRAM) — x100k bytes, ~10 cycles

L3 cache (SRAM) — xM bytes, x10 cycles

main memory (DRAM) — xG bytes, x100 cycles

local secondary storage (local disk) — x100G bytes, x100-1000 cycles

remote secondary storage (network disk) — x10T bytes, x1000 cycles

Bryant and O'Hallaron, *Computer Systems: A Programmer's Perspective*, Third Edition

Virtual Memory as a Cache

Page size typically 4k to 64k

“page” instead of “block”

“page fault” instead of “cache miss”

Fully associative

requires a large mapping

Complex replacement rules

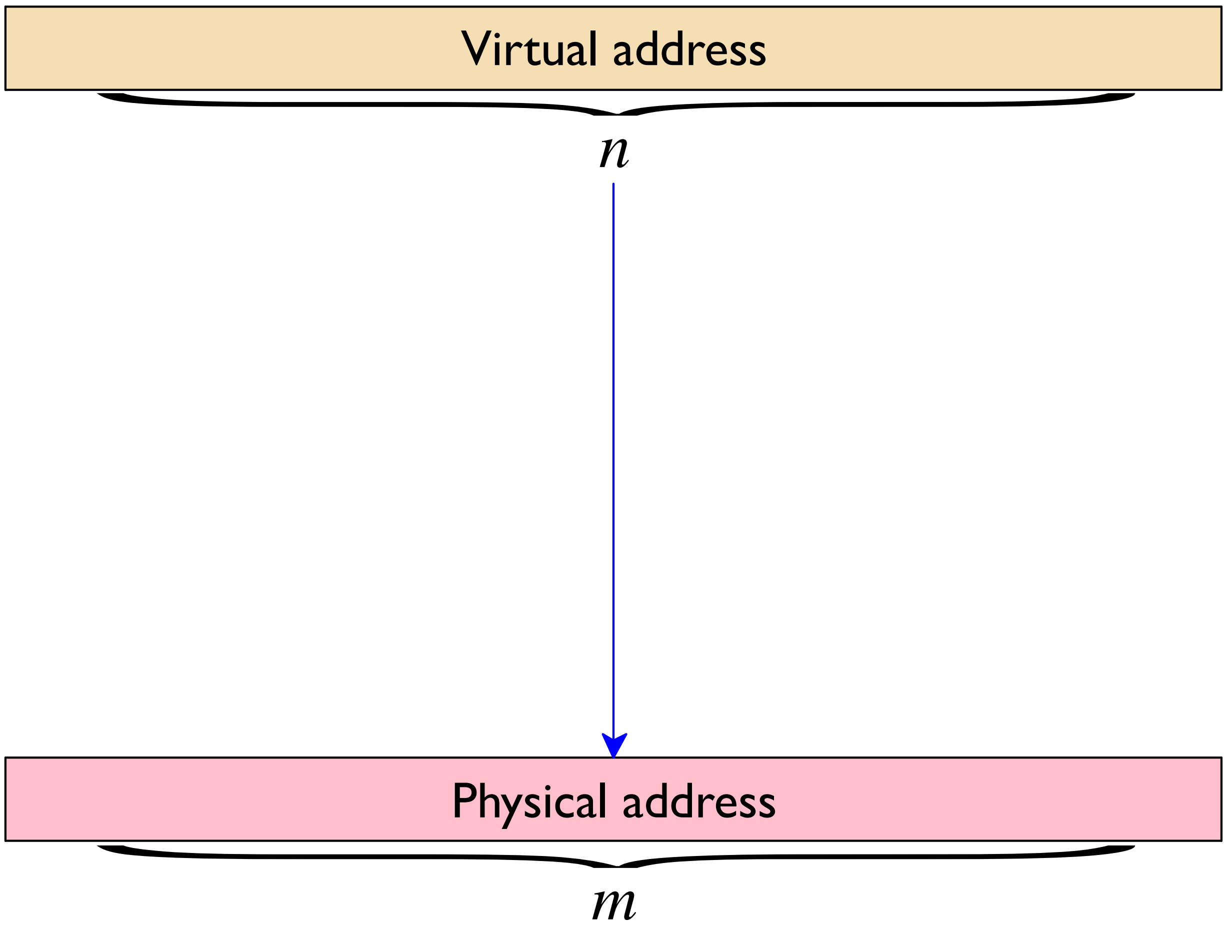
instead of just LRU

Write-back

as opposed to write-through

Address Translation with a Page Table

Address size = 2^n



Virtual address

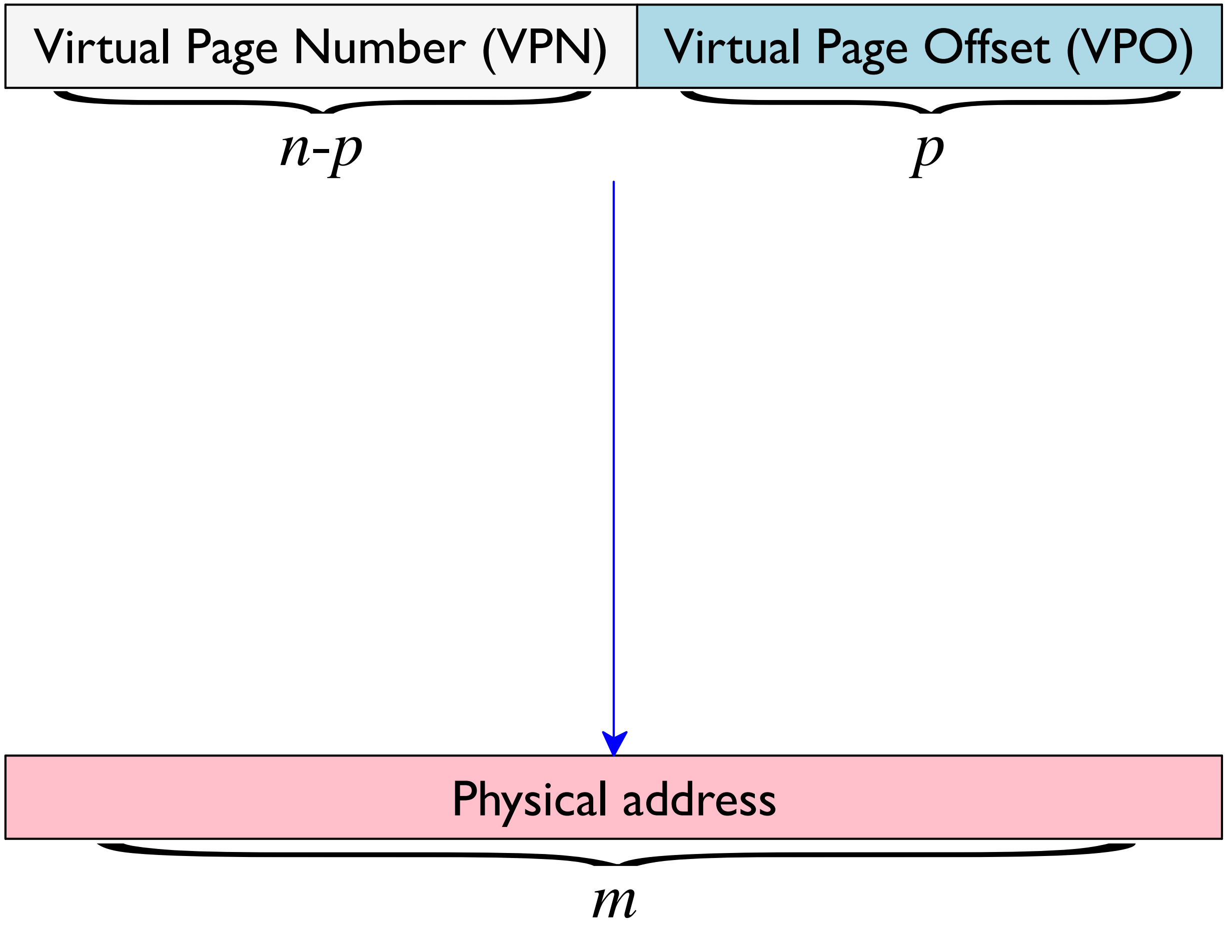
n

Physical address

m

Address Translation with a Page Table

Address size = 2^n Page size = 2^p



Virtual Page Number (VPN) | Virtual Page Offset (VPO)

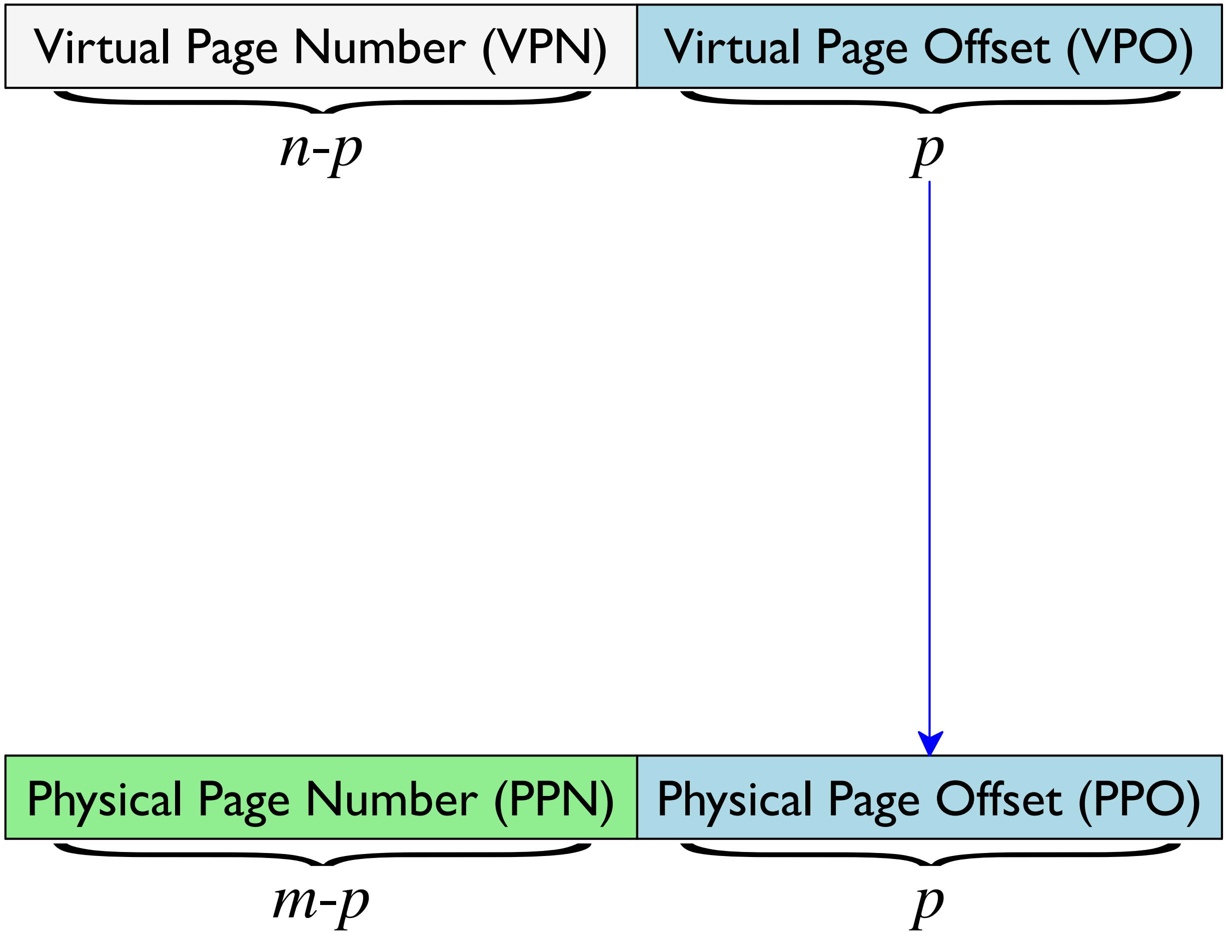
n-p | p

Physical address

m

Address Translation with a Page Table

Address size = 2^n Page size = 2^p



Virtual Page Number (VPN)

Virtual Page Offset (VPO)

n-p

p

Physical Page Number (PPN)

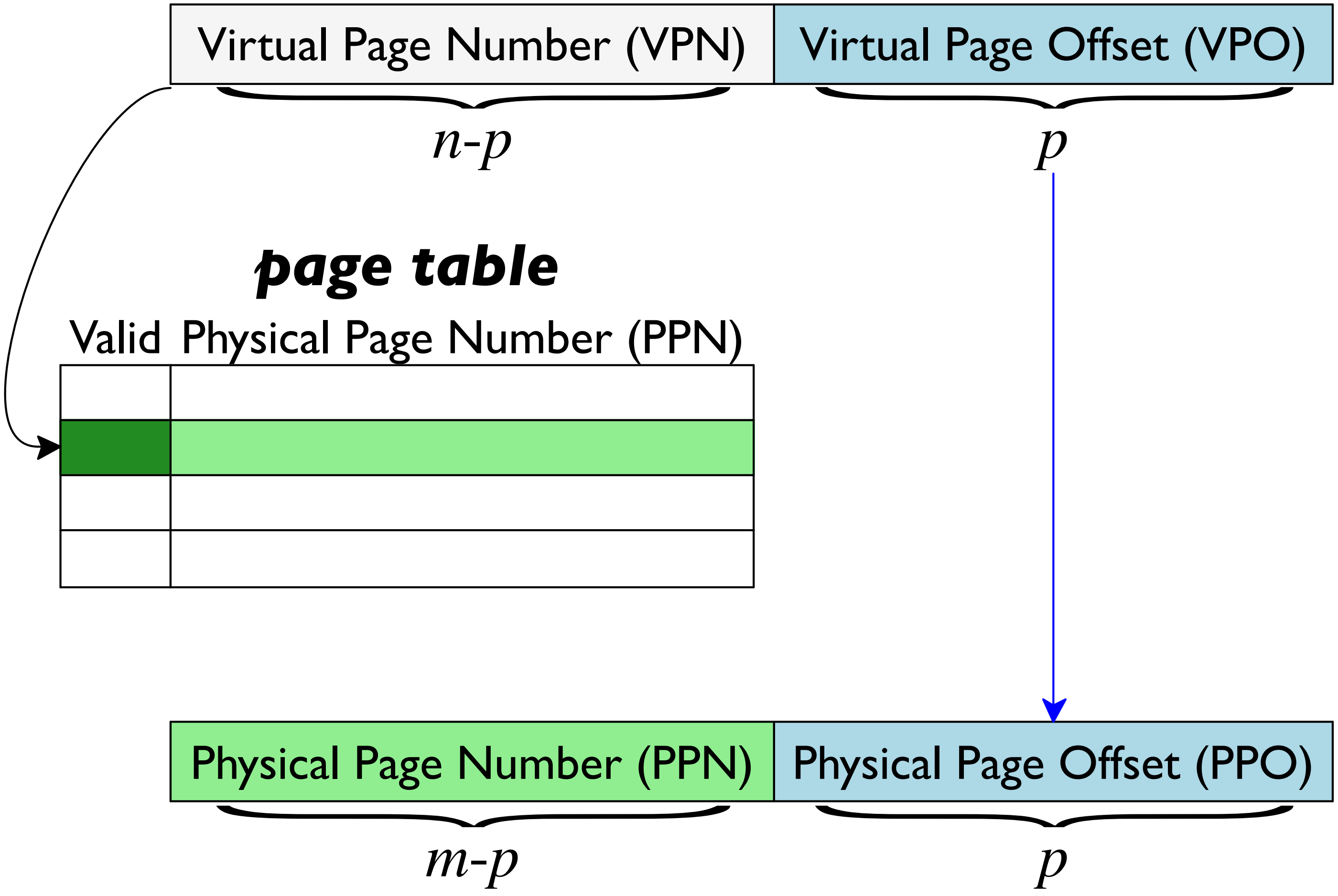
Physical Page Offset (PPO)

m-p

p

Address Translation with a Page Table

Address size = 2^n Page size = 2^p



Virtual Page Number (VPN)

Virtual Page Offset (VPO)

$n\text{-}p$

p

page table

Valid

Physical Page Number (PPN)

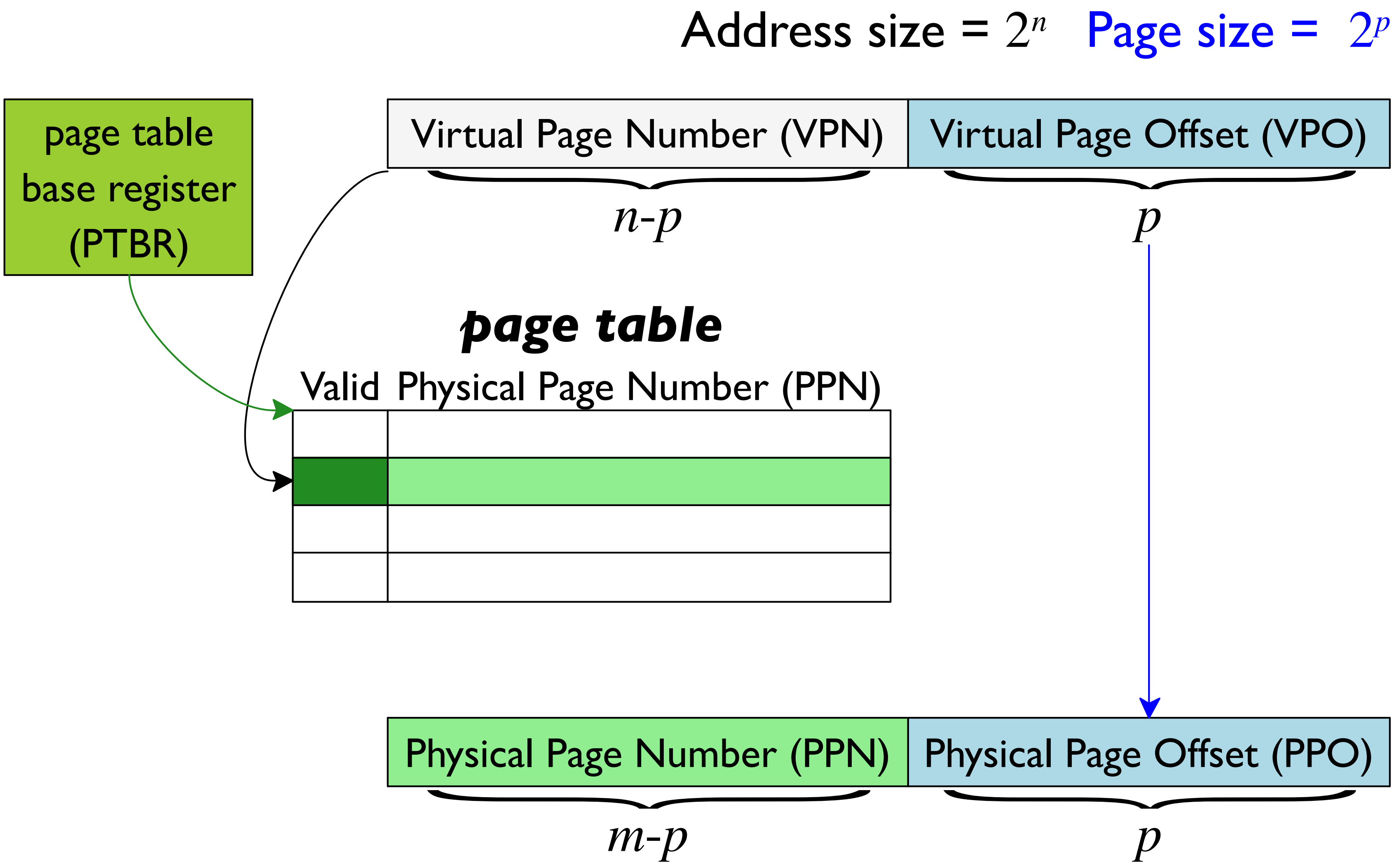
Physical Page Number (PPN)

Physical Page Offset (PPO)

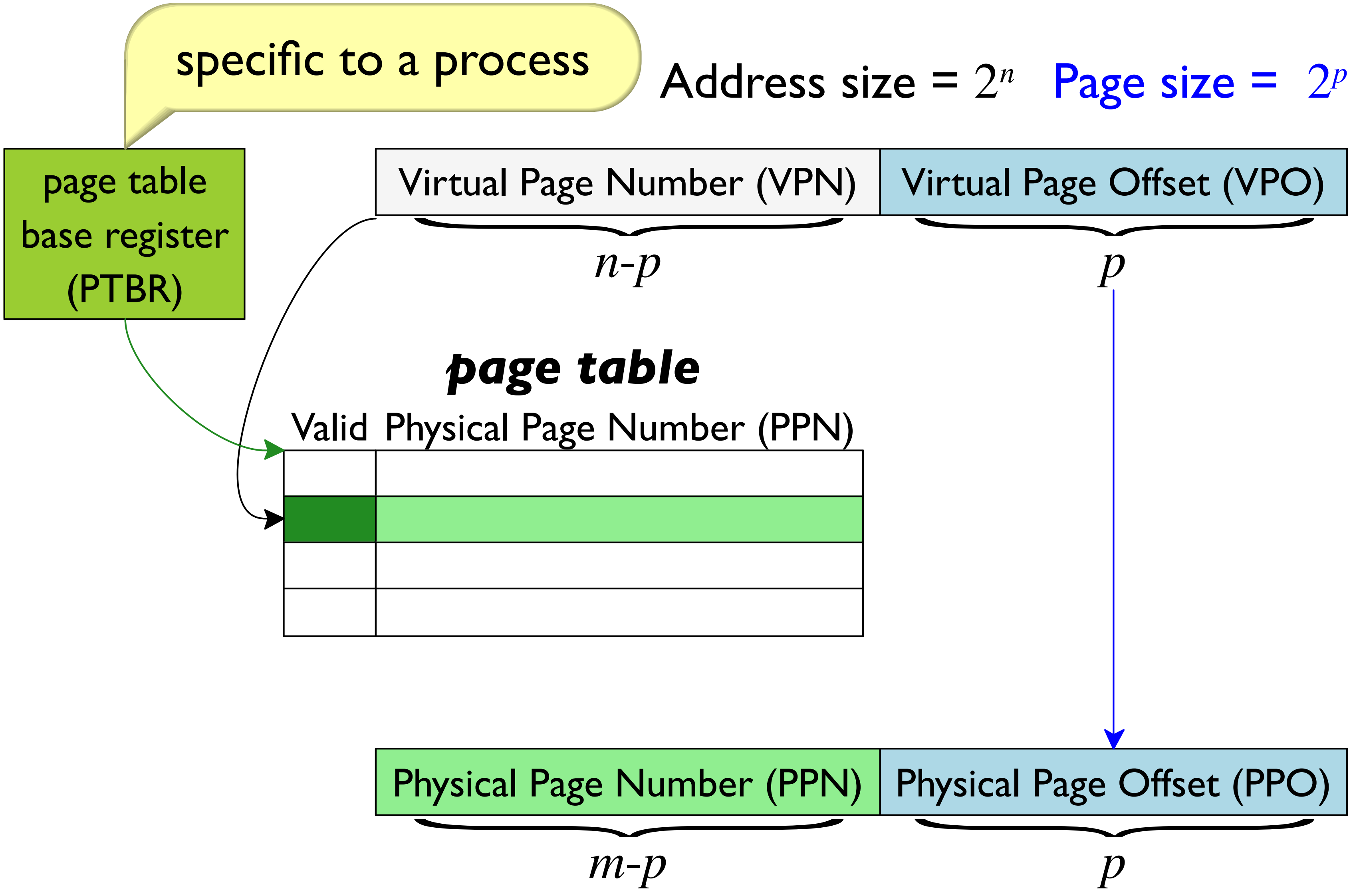
$m\text{-}p$

p

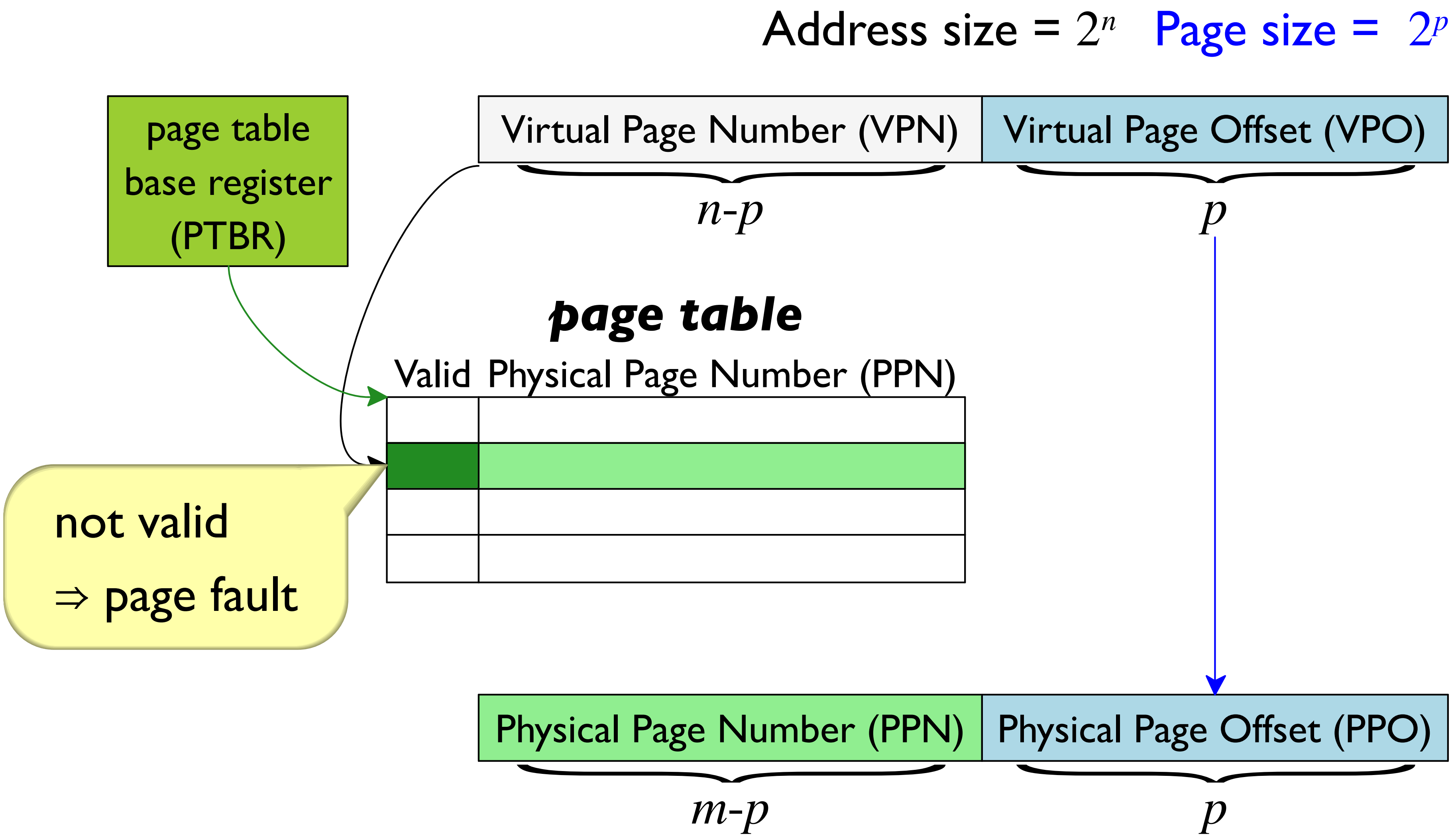
Address Translation with a Page Table


Address size = 2^n
Page size = 2^p
page table base register (PTBR)
Virtual Page Number (VPN)
Virtual Page Offset (VPO)
n-p
p
page table
Valid
Physical Page Number (PPN)
Physical Page Number (PPN)
Physical Page Offset (PPO)
m-p
p

Address Translation with a Page Table


specific to a process
Address size = 2^n
Page size = 2^p
page table base register (PTBR)
Virtual Page Number (VPN)
Virtual Page Offset (VPO)
n-p
p
page table
Valid
Physical Page Number (PPN)
Physical Page Number (PPN)
Physical Page Offset (PPO)
m-p
p

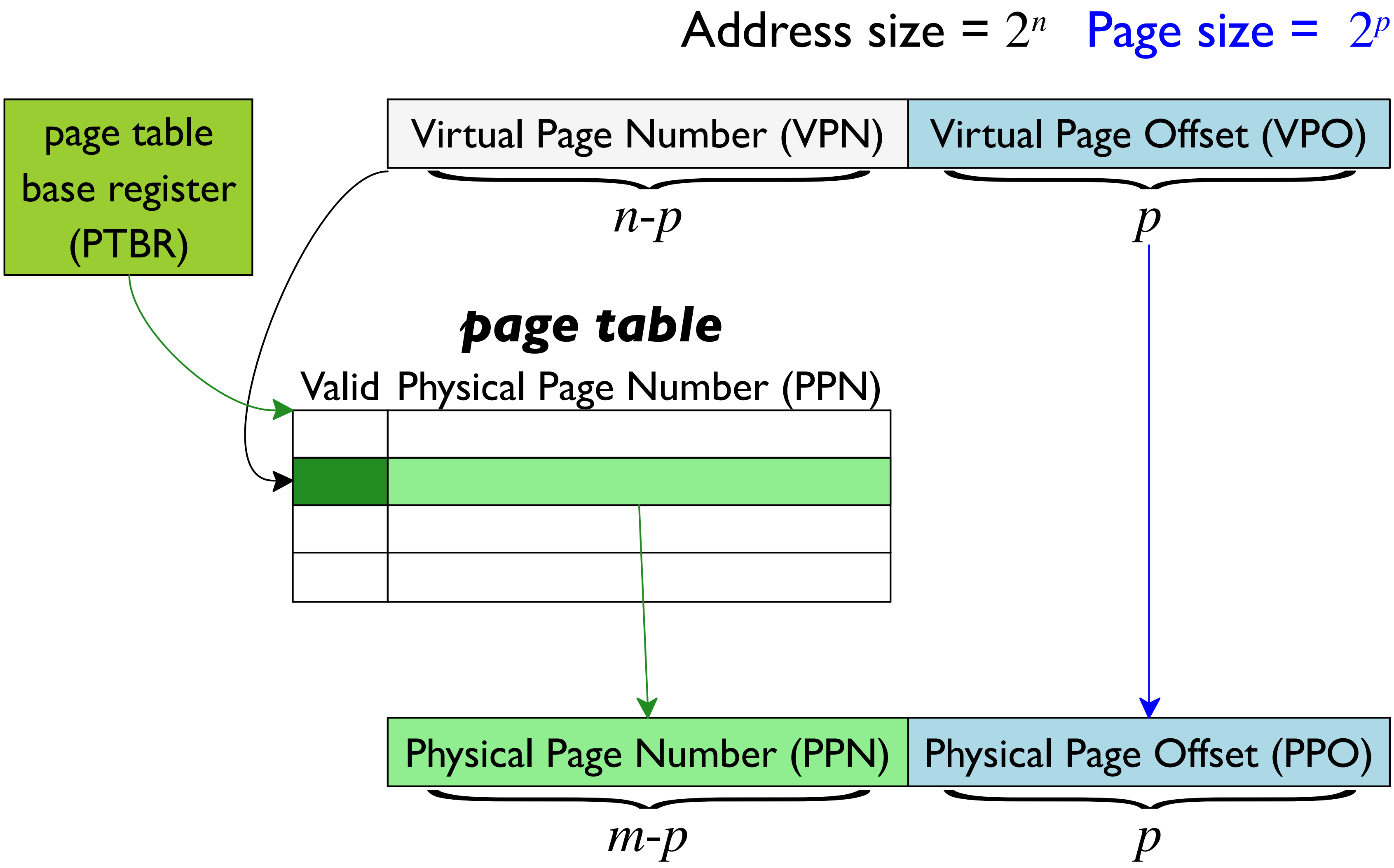
Address Translation with a Page Table



Address size = 2^n
Page size = 2^p
page table base register (PTBR)
Virtual Page Number (VPN)
Virtual Page Offset (VPO)
n-p
p
page table
Valid
Physical Page Number (PPN)
not valid
⇒ page fault
Physical Page Number (PPN)
Physical Page Offset (PPO)
m-p
p

Bryant and O'Hallaron, *Computer Systems: A Programmer's Perspective*, Third Edition

Address Translation with a Page Table

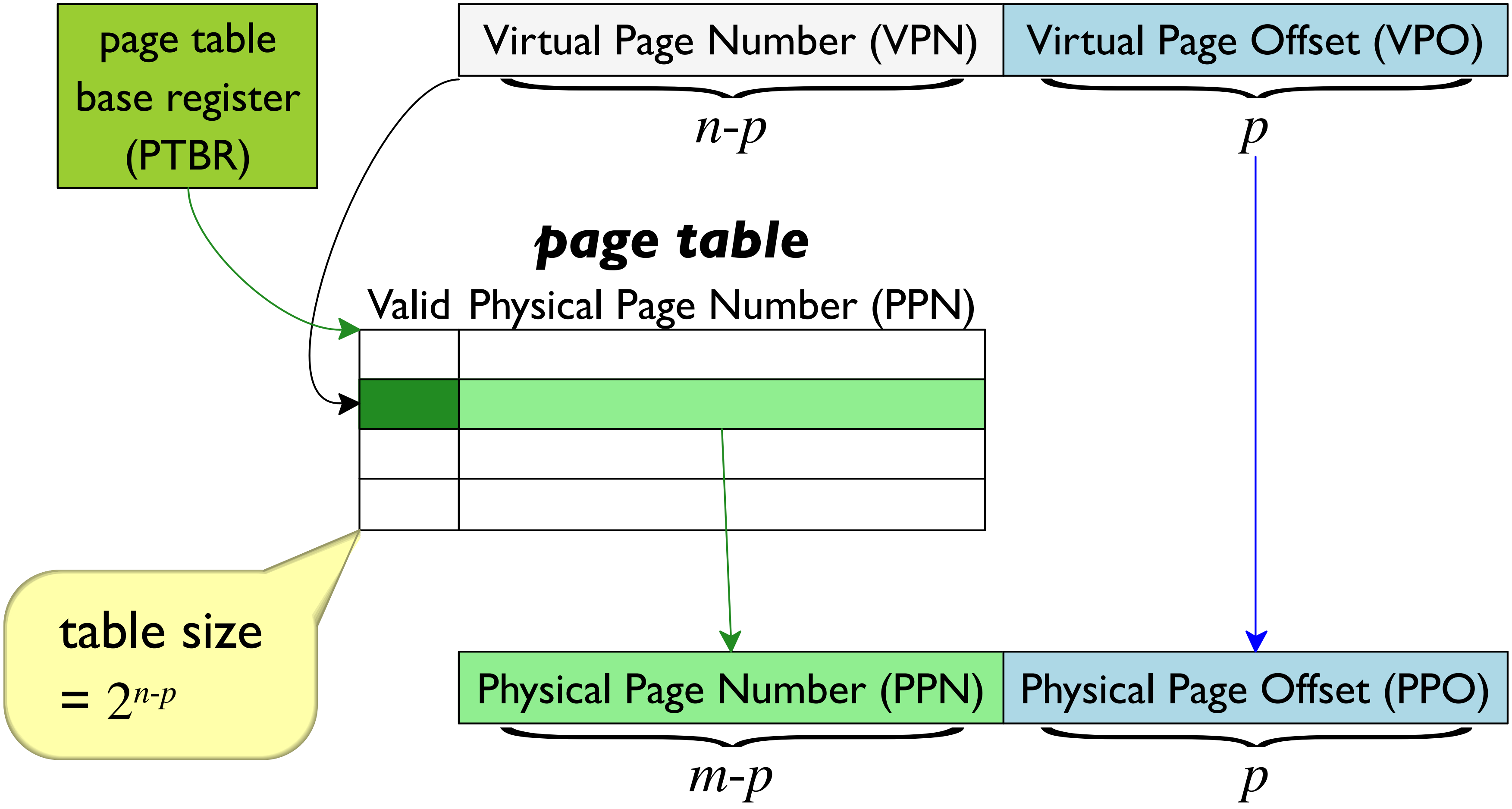


Address size = 2^n
Page size = 2^p
page table base register (PTBR)
Virtual Page Number (VPN)
Virtual Page Offset (VPO)
n-p
p
page table
Valid
Physical Page Number (PPN)
Physical Page Number (PPN)
Physical Page Offset (PPO)
m-p
p

Bryant and O'Hallaron, *Computer Systems: A Programmer's Perspective*, Third Edition

Address Translation with a Page Table

Address size = 2^n Page size = 2^p



page table base register (PTBR)

Virtual Page Number (VPN)

Virtual Page Offset (VPO)

$n\text{-}p$

p

page table

Valid

Physical Page Number (PPN)

table size = $2^{n\text{-}p}$

Physical Page Number (PPN)

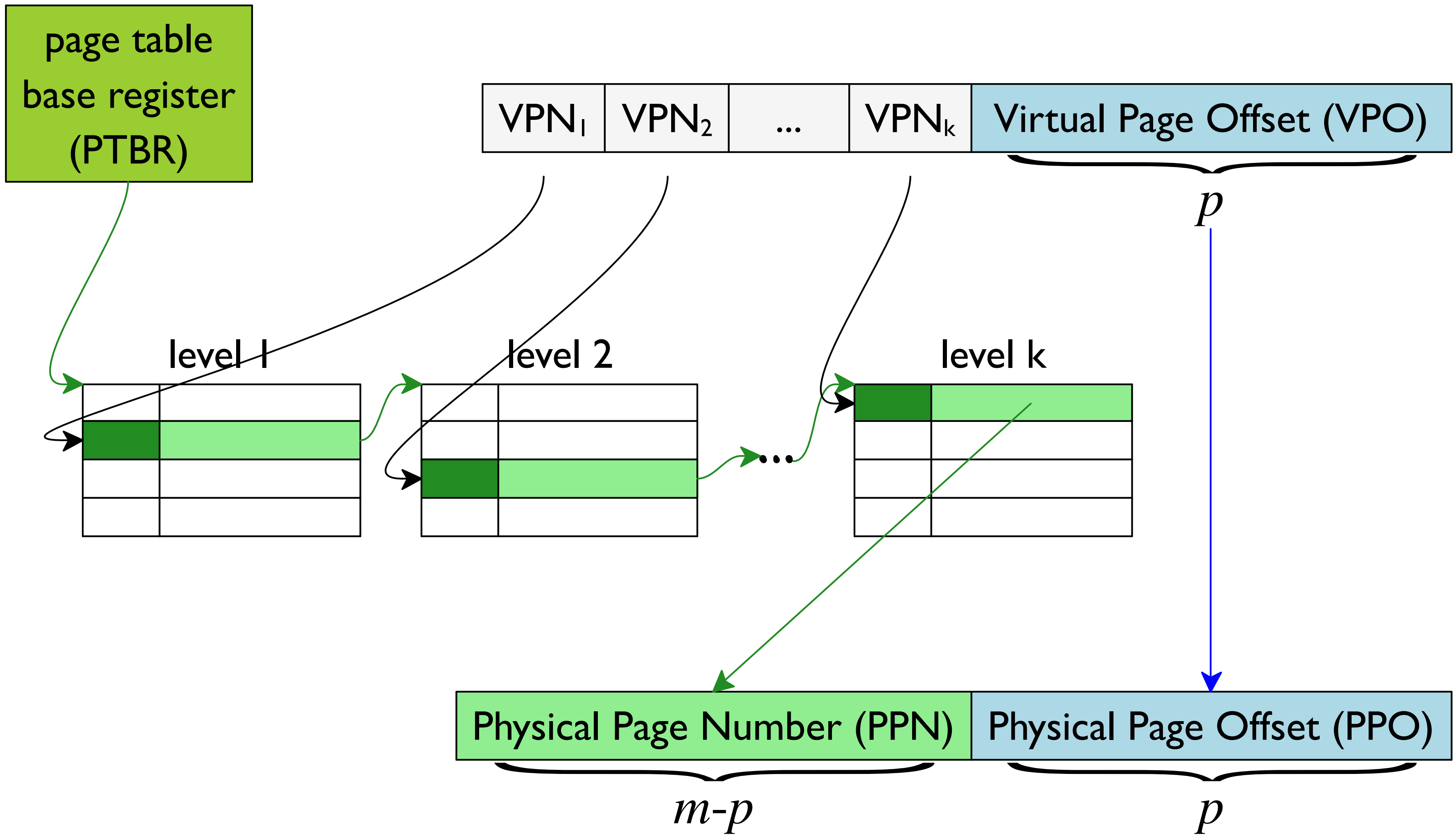
Physical Page Offset (PPO)

$m\text{-}p$

p

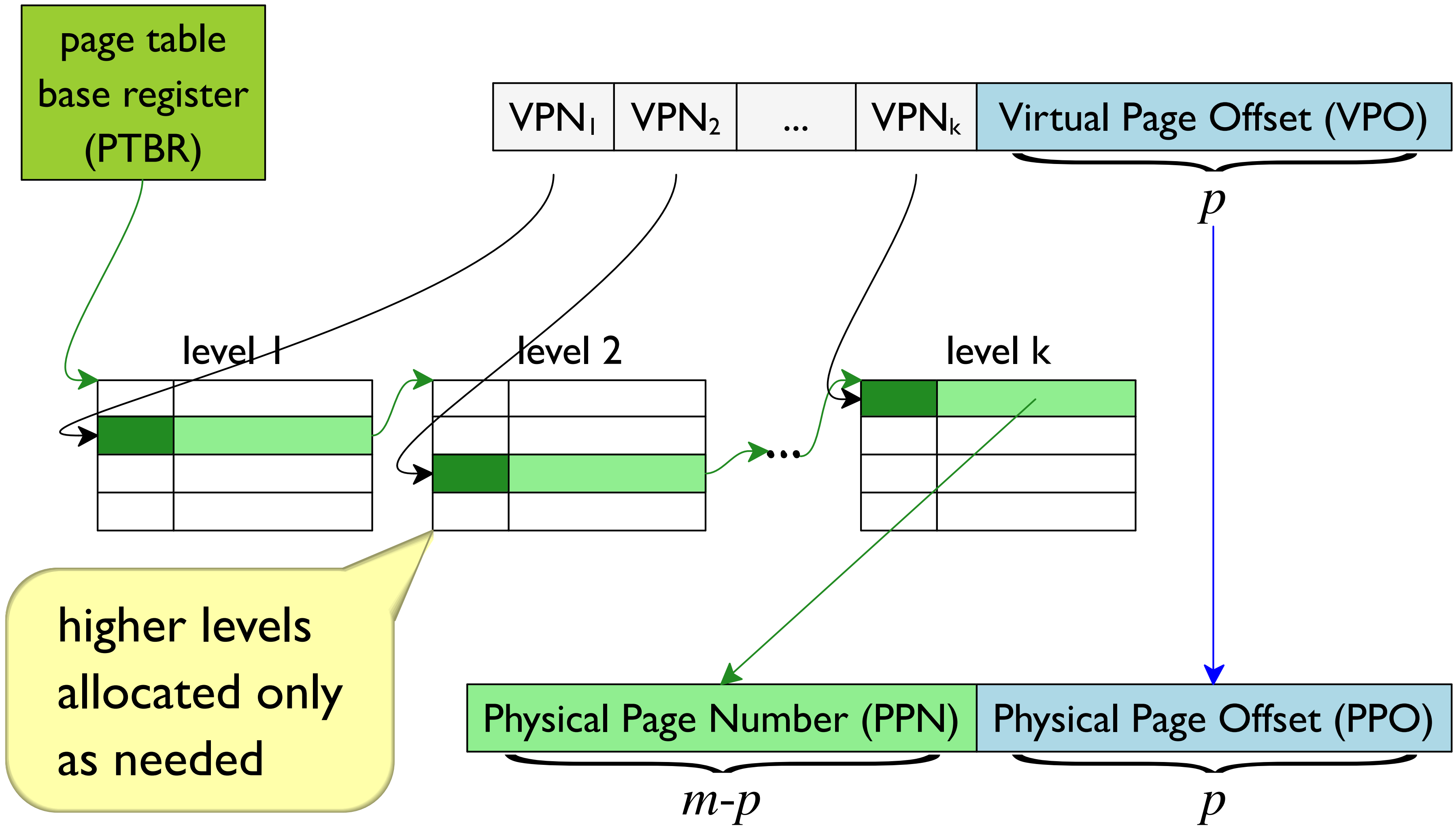
Bryant and O'Hallaron, *Computer Systems: A Programmer's Perspective*, Third Edition

Multi-Level Page Table

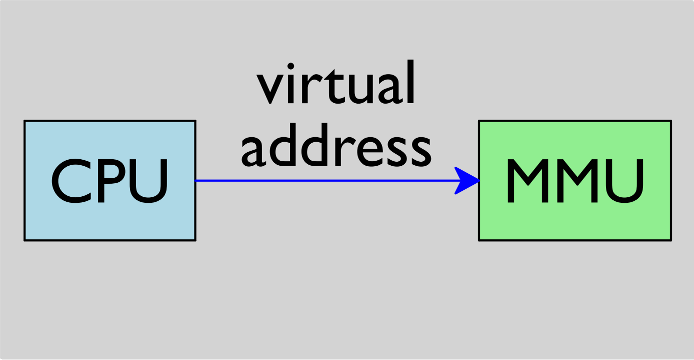

page table base register (PTBR)
VPN1
VPN2
...
VPNk
Virtual Page Offset (VPO)
p
level 1
level 2
level k
...
Physical Page Number (PPN)
Physical Page Offset (PPO)
m-p
p

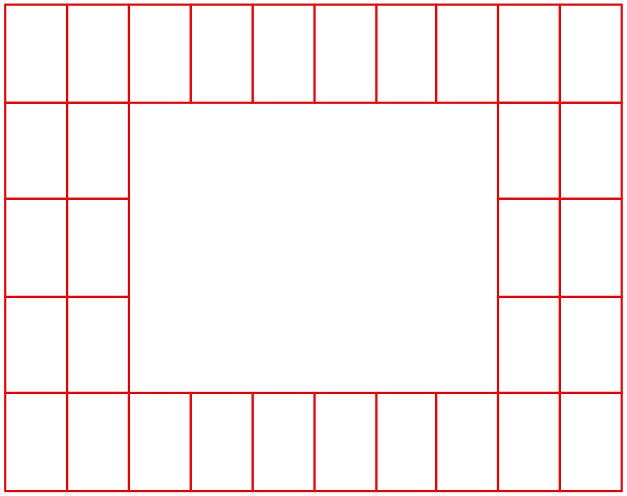
Bryant and O'Hallaron, *Computer Systems: A Programmer's Perspective*, Third Edition

Multi-Level Page Table

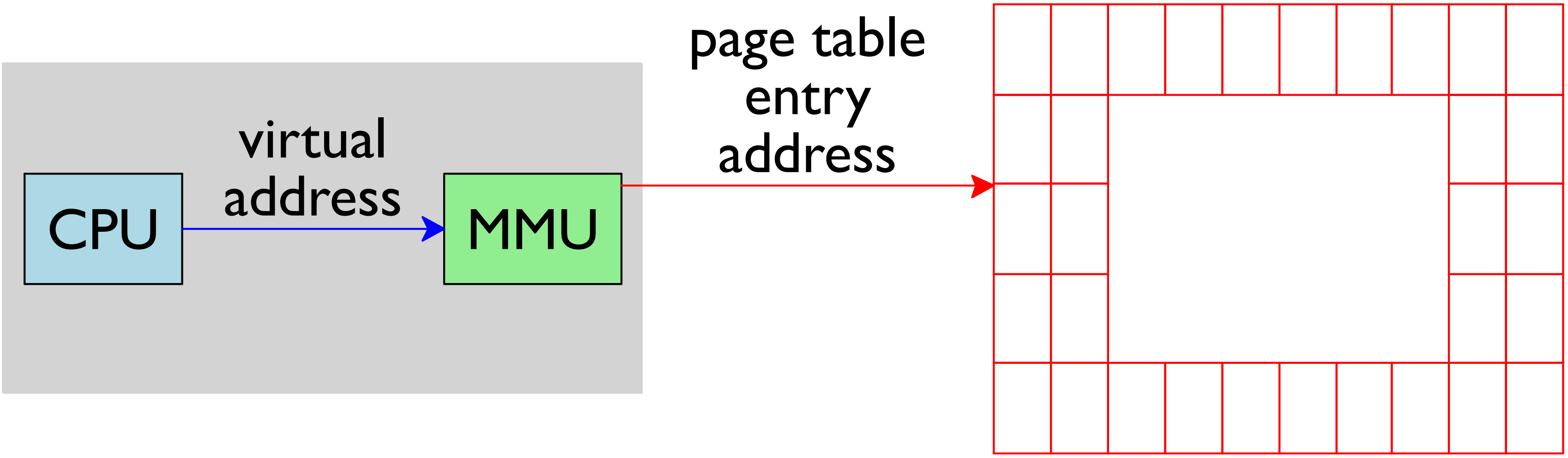

page table base register (PTBR)
VPN₁
VPN₂
...
VPNₖ
Virtual Page Offset (VPO)
p
level 1
level 2
...
level k
higher levels allocated only as needed
Physical Page Number (PPN)
Physical Page Offset (PPO)
m-p
p

Address Translation: Page Hit

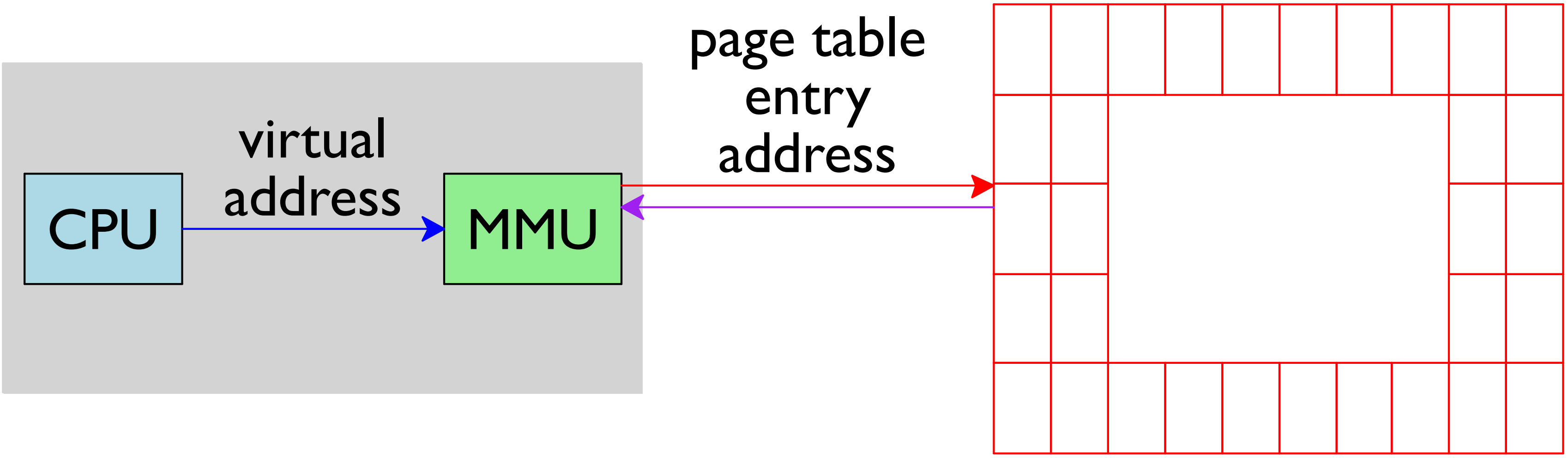




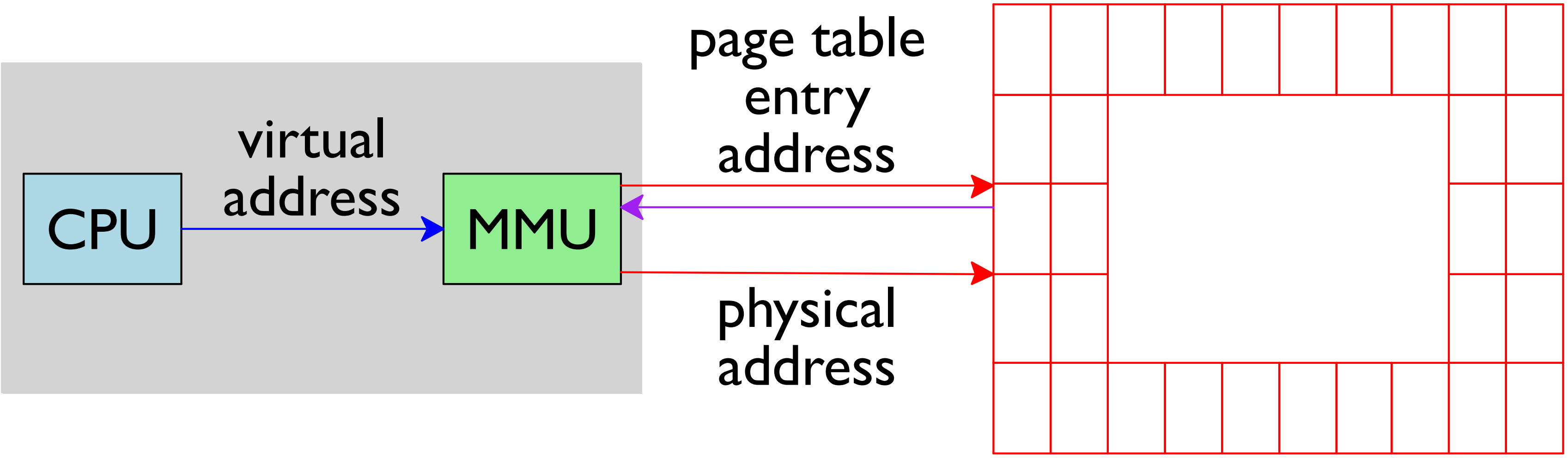
Address Translation: Page Hit



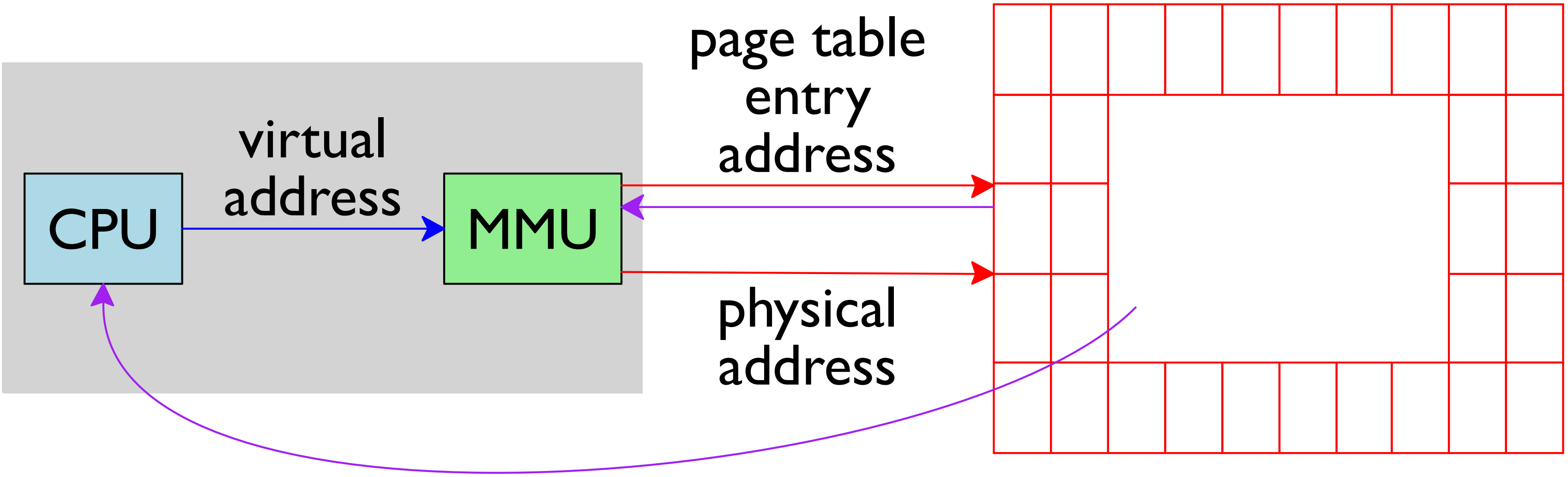
Address Translation: Page Hit



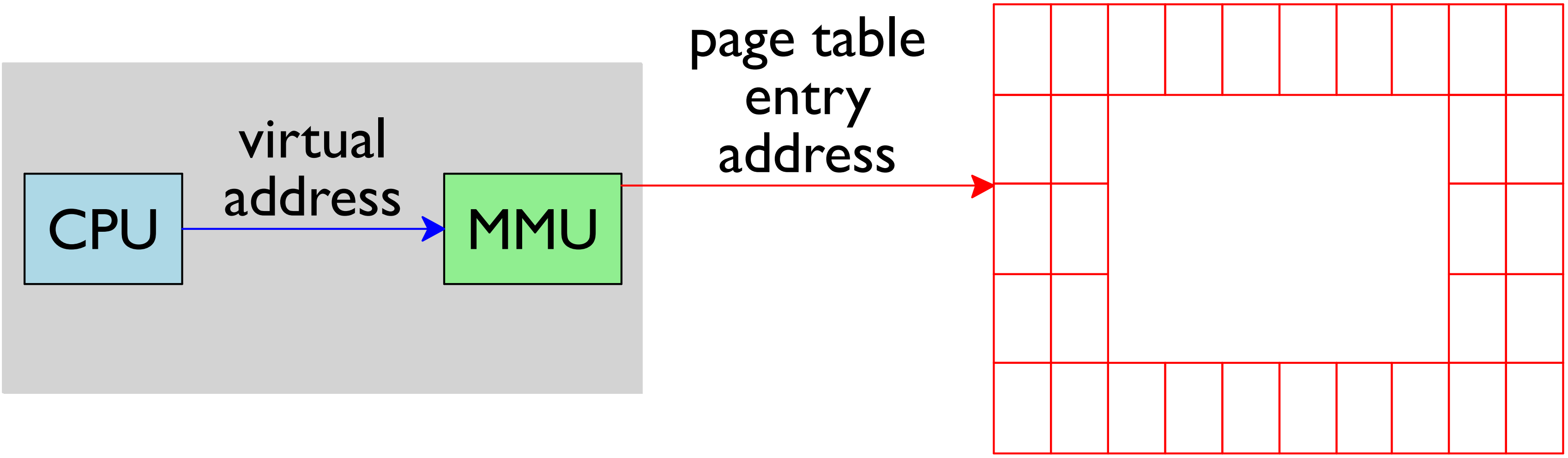
Address Translation: Page Hit



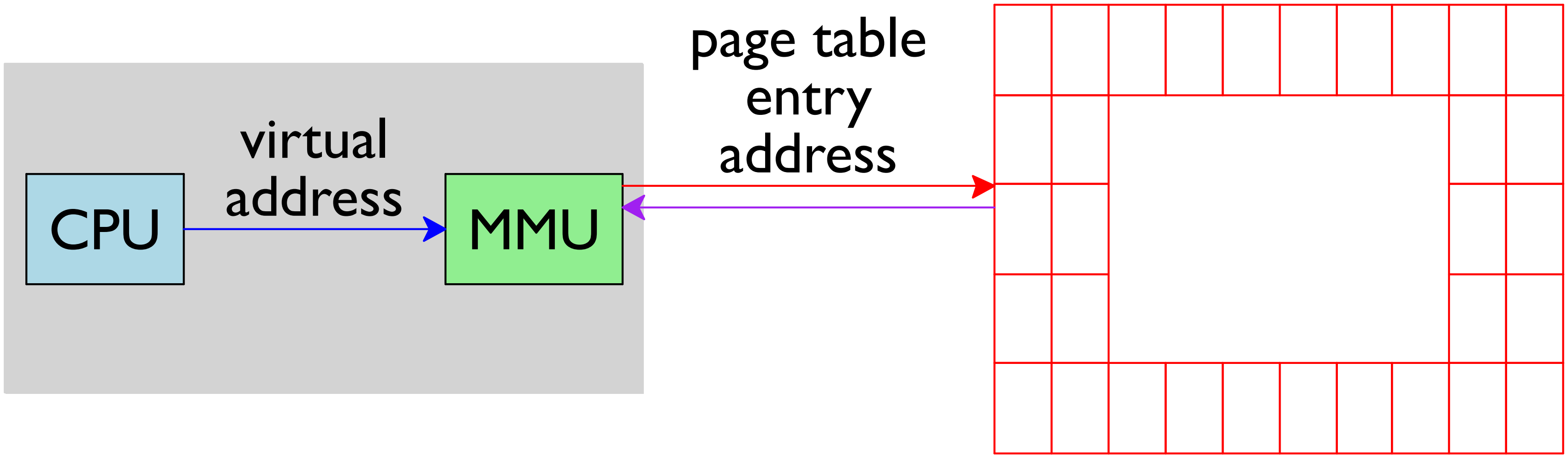
Address Translation: Page Hit



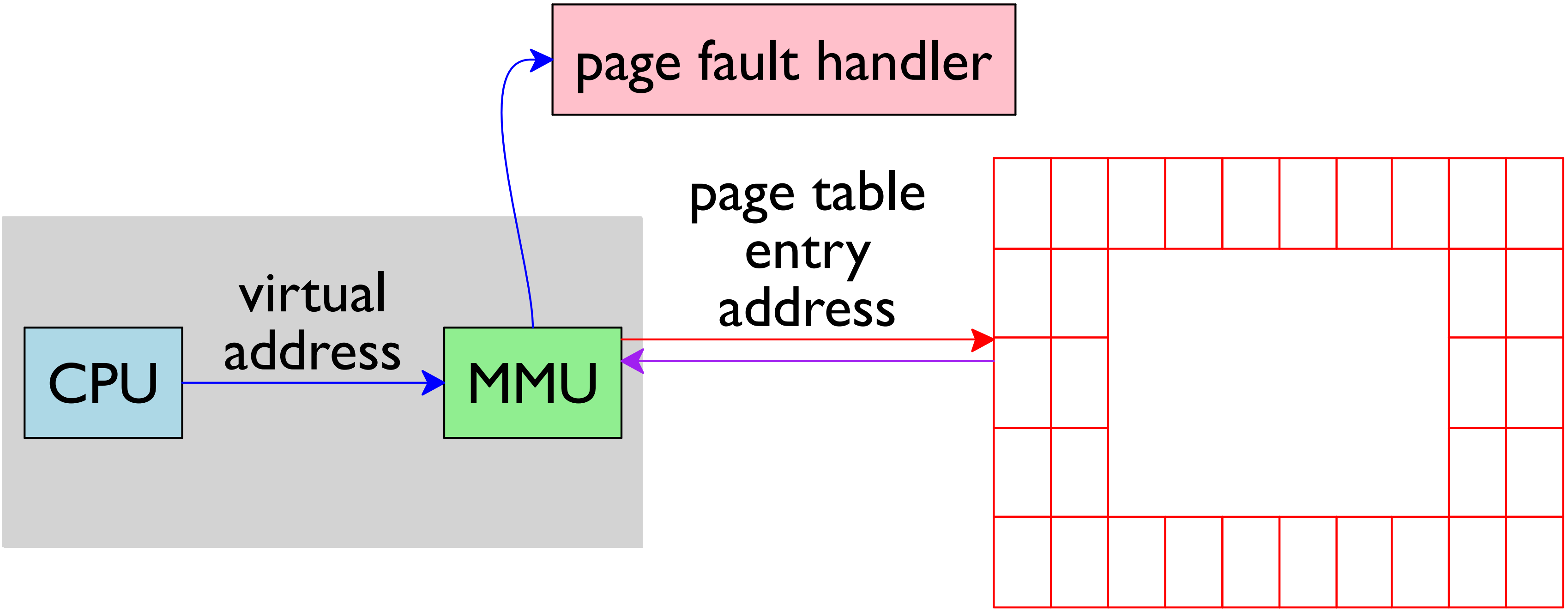
Address Translation: Page Fault



Address Translation: Page Fault

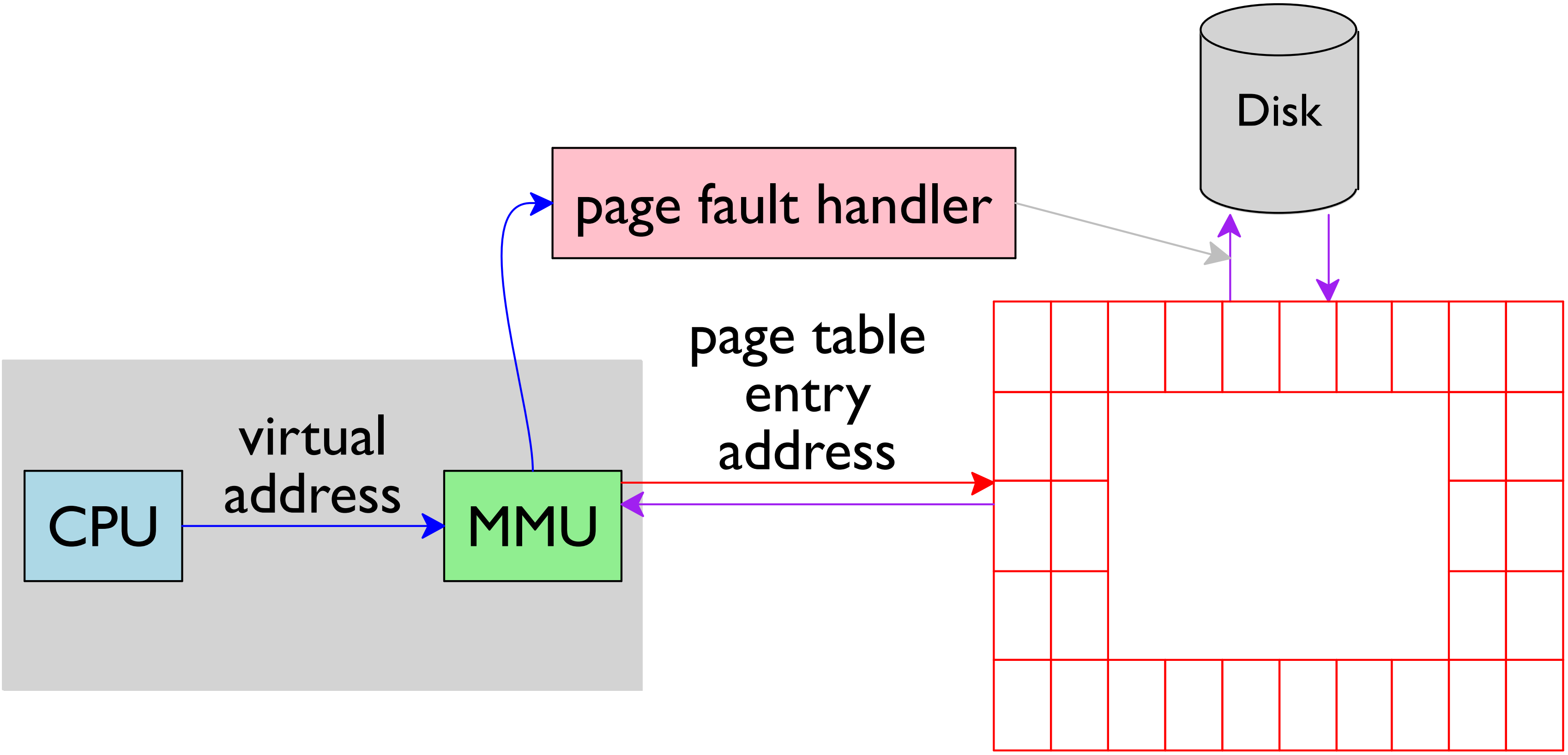


Address Translation: Page Fault



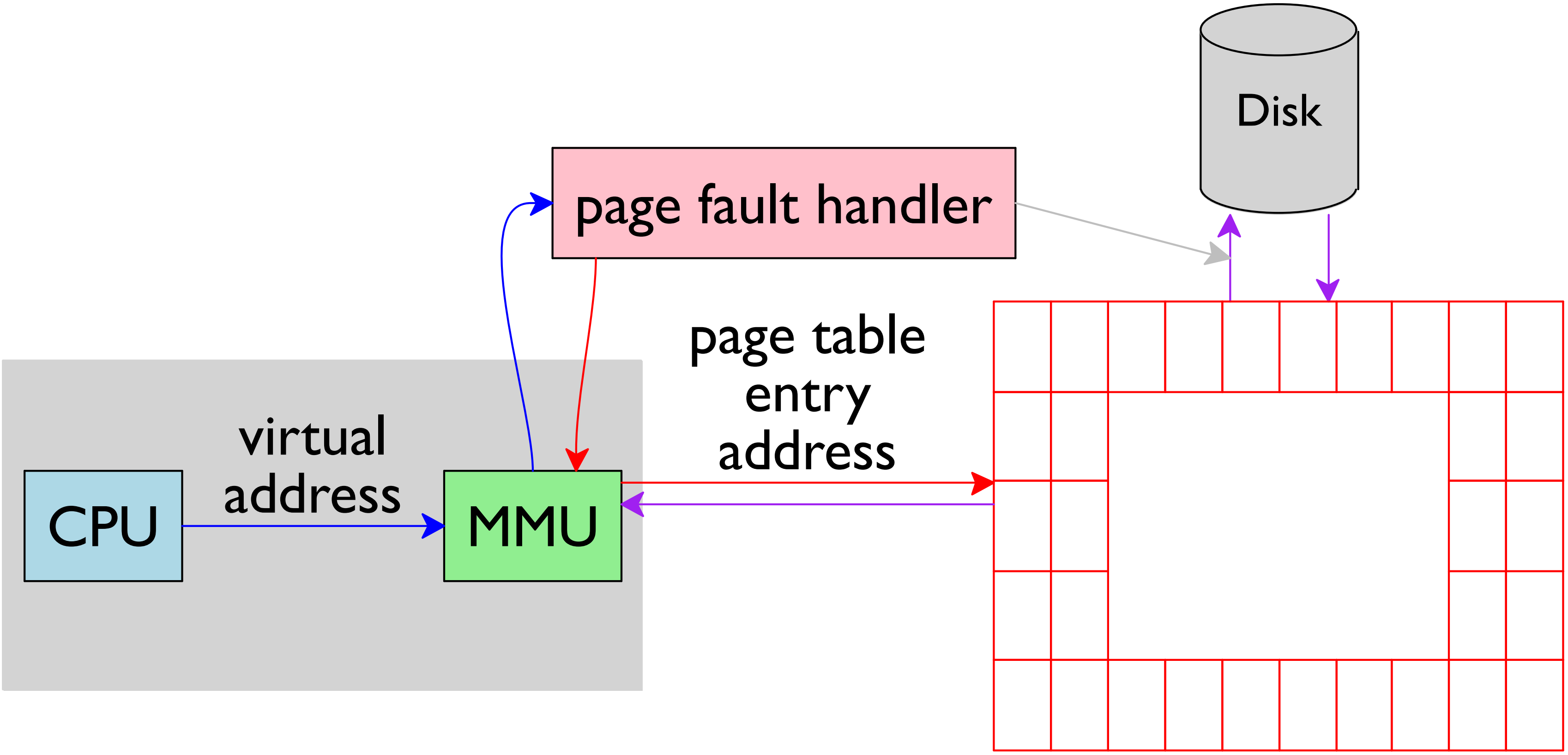
Bryant and O'Hallaron, *Computer Systems: A Programmer's Perspective*, Third Edition

Address Translation: Page Fault



Bryant and O'Hallaron, *Computer Systems: A Programmer's Perspective*, Third Edition

Address Translation: Page Fault



Disk

page fault handler

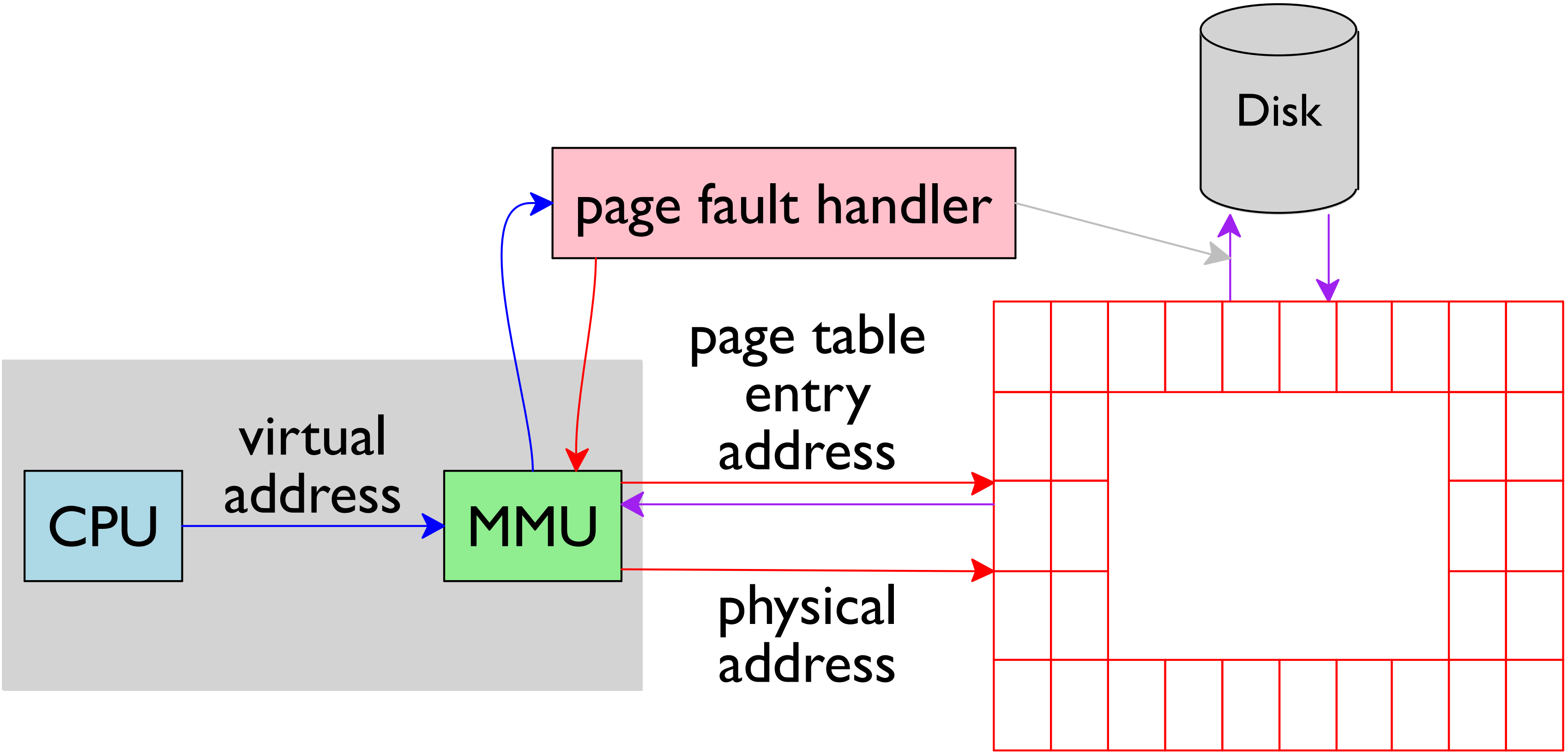
page table entry address

virtual address

CPU

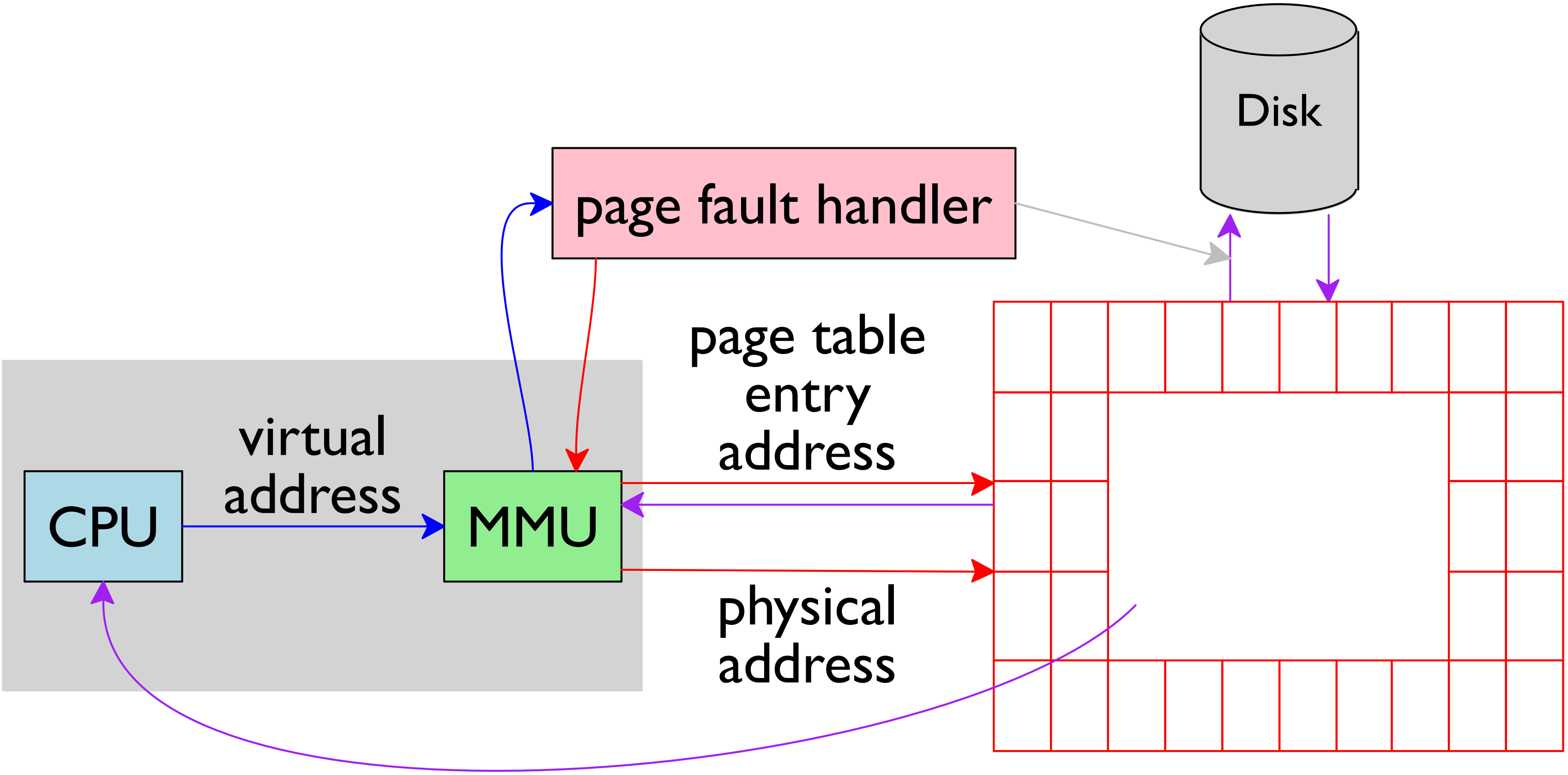
MMU

Address Translation: Page Fault



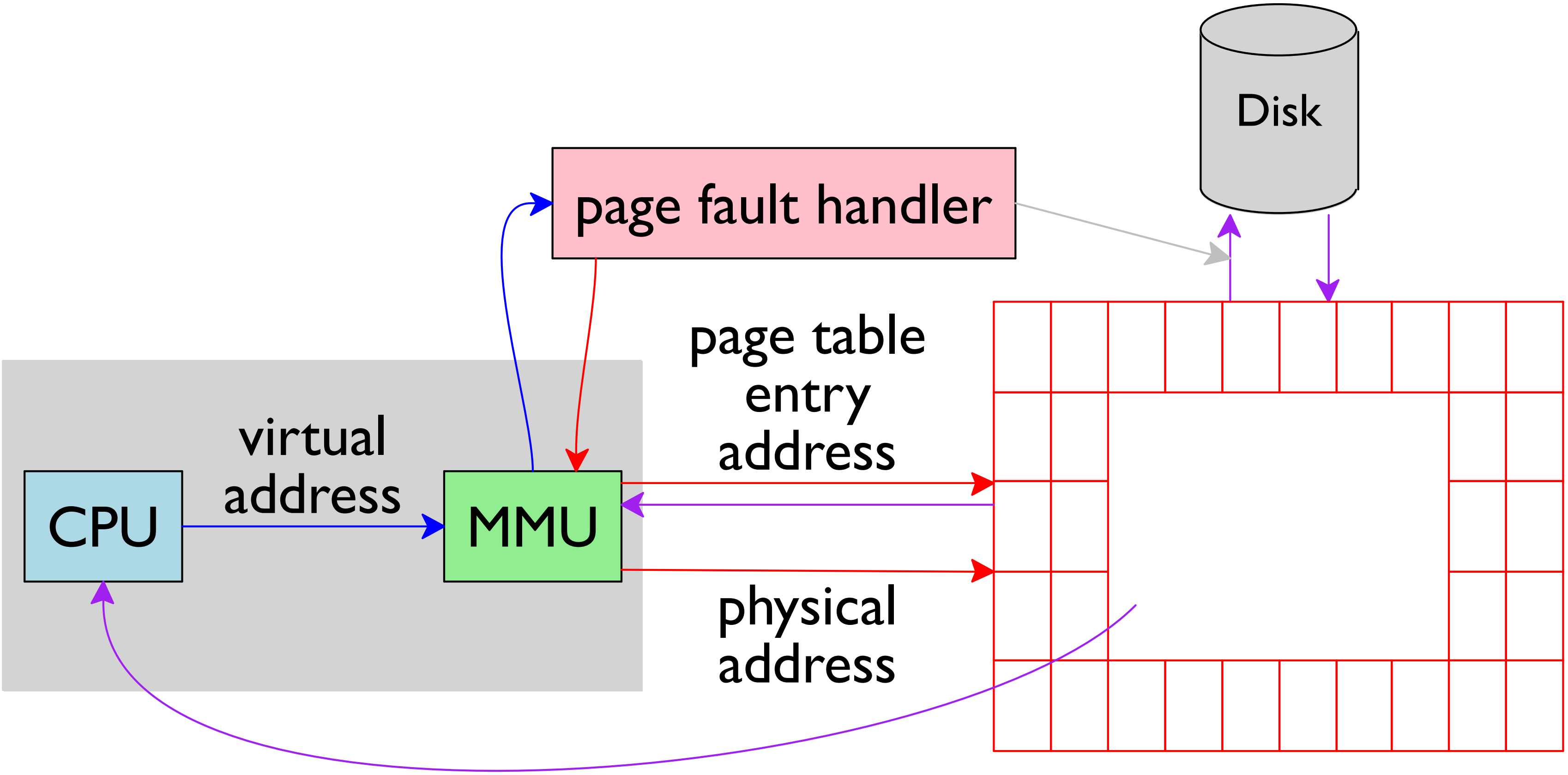
Bryant and O'Hallaron, *Computer Systems: A Programmer's Perspective*, Third Edition

Address Translation: Page Fault



Bryant and O'Hallaron, *Computer Systems: A Programmer's Perspective*, Third Edition

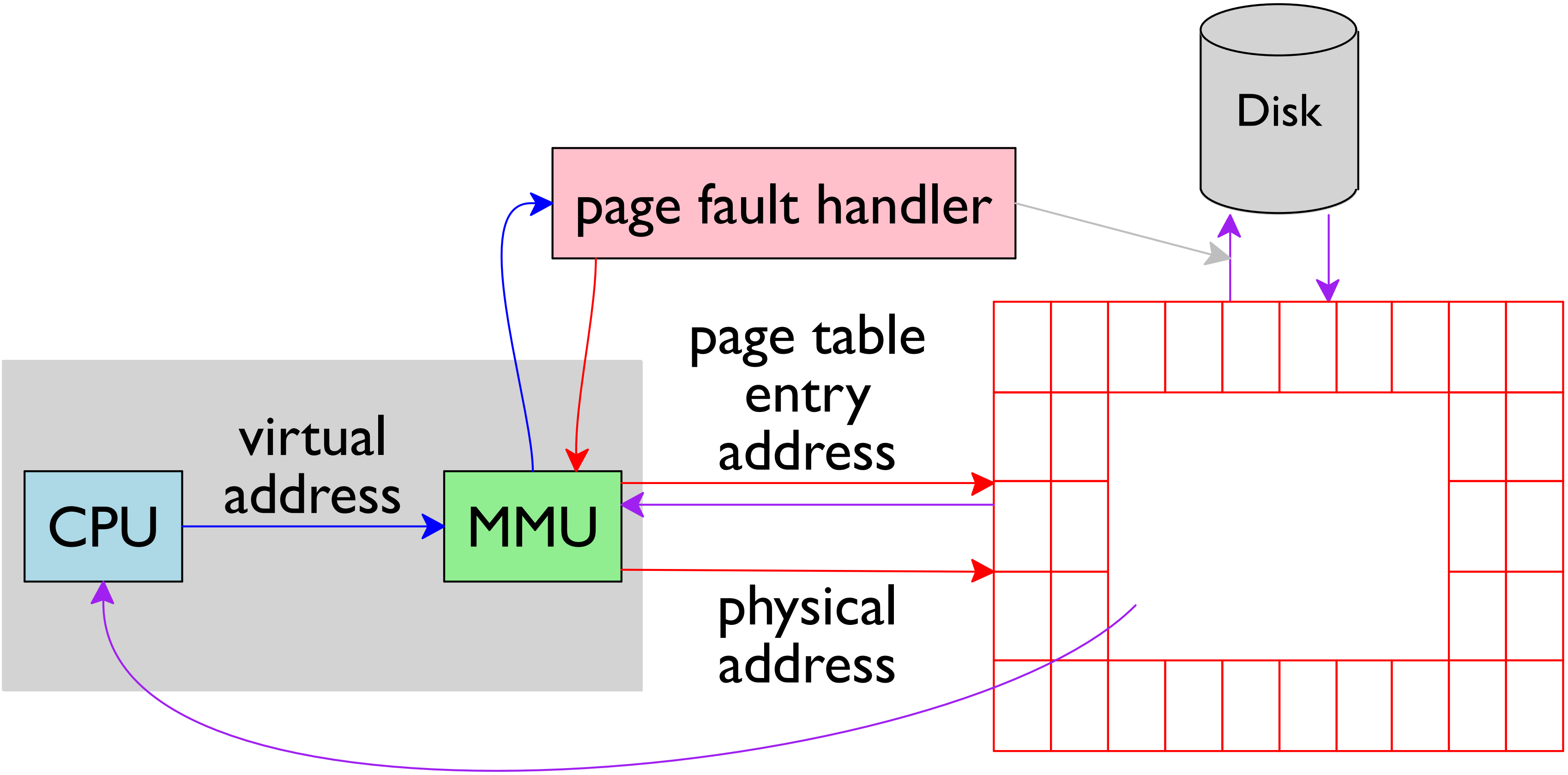
Address Translation: Page Fault



Moving data to and from disk — ok if good ***locality***

Bryant and O'Hallaron, *Computer Systems: A Programmer's Perspective*, Third Edition

Address Translation: Page Fault

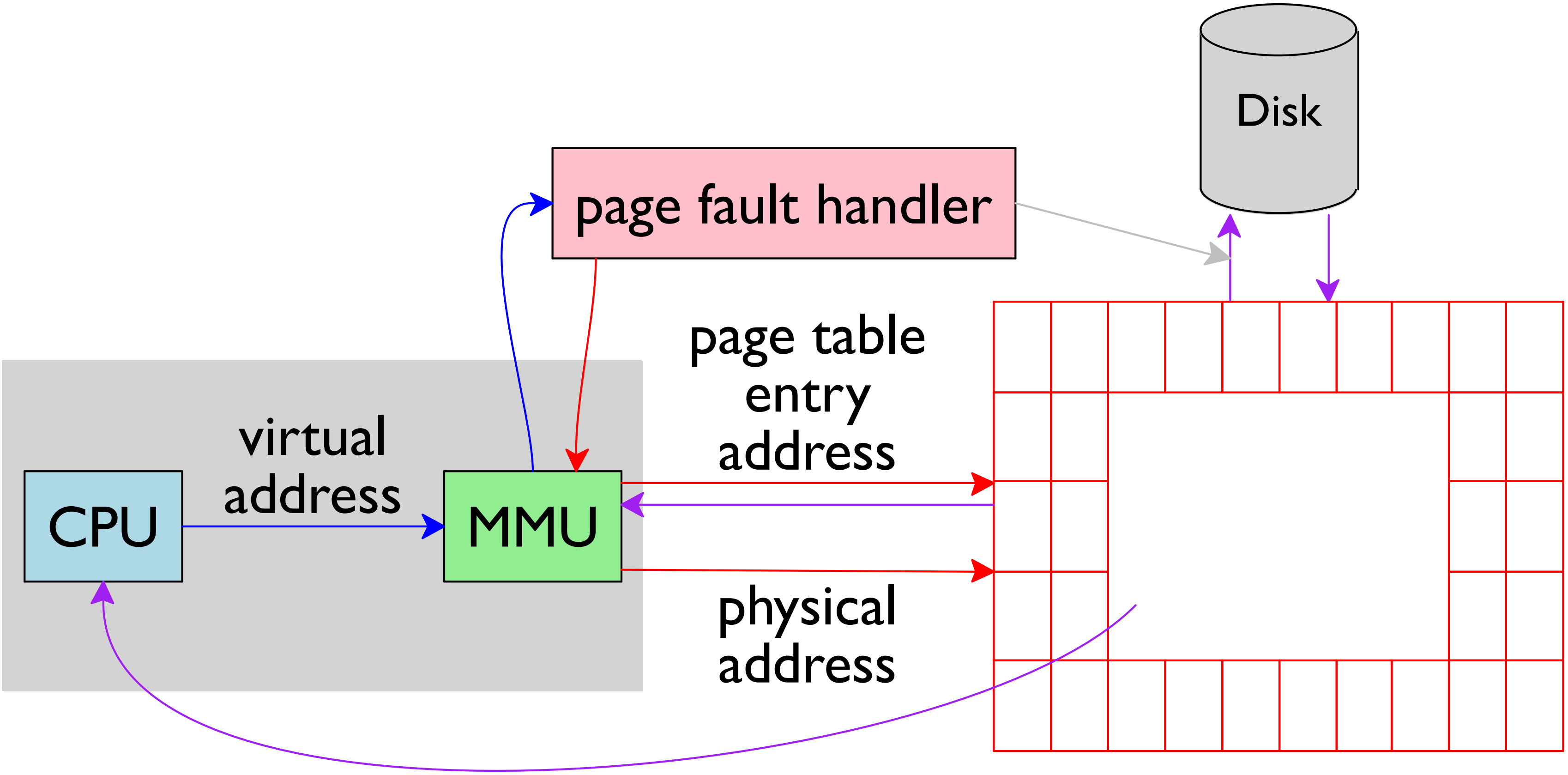


Moving data to and from disk — ok if good ***locality***

Working set = pages currently being used

Bryant and O'Hallaron, *Computer Systems: A Programmer's Perspective*, Third Edition

Address Translation: Page Fault



Moving data to and from disk — ok if good ***locality***

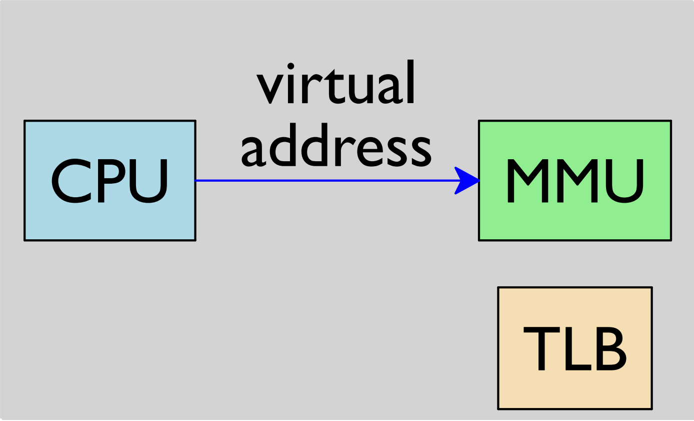
Working set = pages currently being used

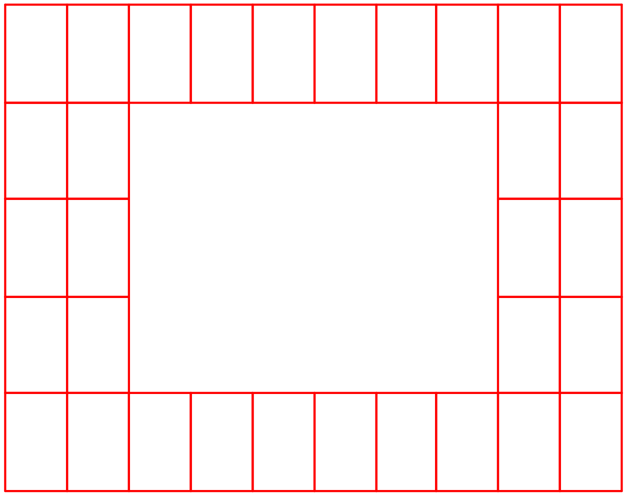
Working set > physcial memory ⇒ ***thrashing***

Bryant and O'Hallaron, *Computer Systems: A Programmer's Perspective*, Third Edition

Translation Lookaside Buffer

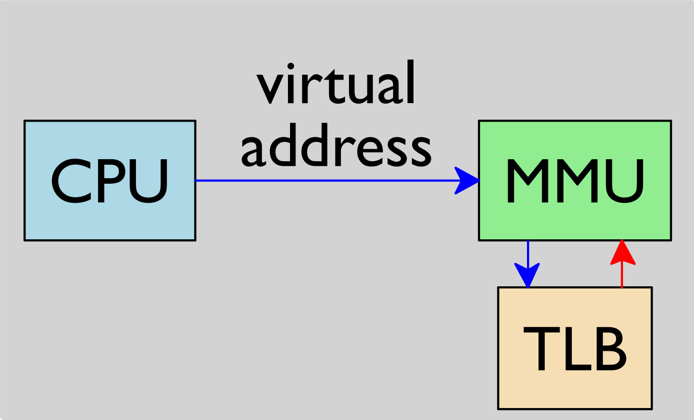
A ***translation lookaside buffer*** (TLB) is a custom cache for address translation

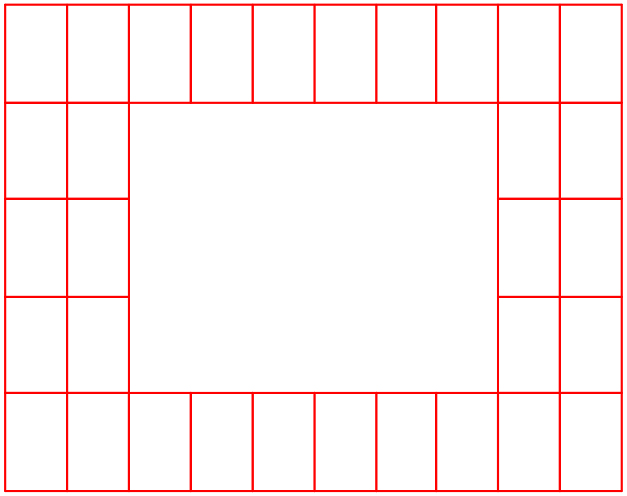




Translation Lookaside Buffer

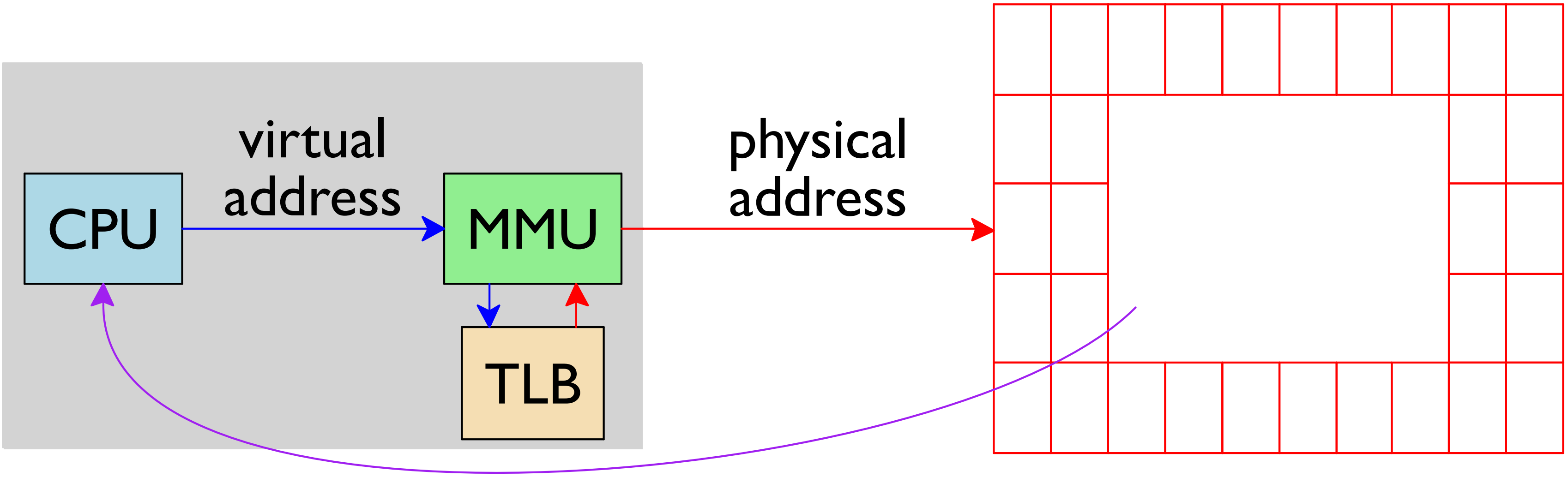
A ***translation lookaside buffer*** (TLB) is a custom cache for address translation



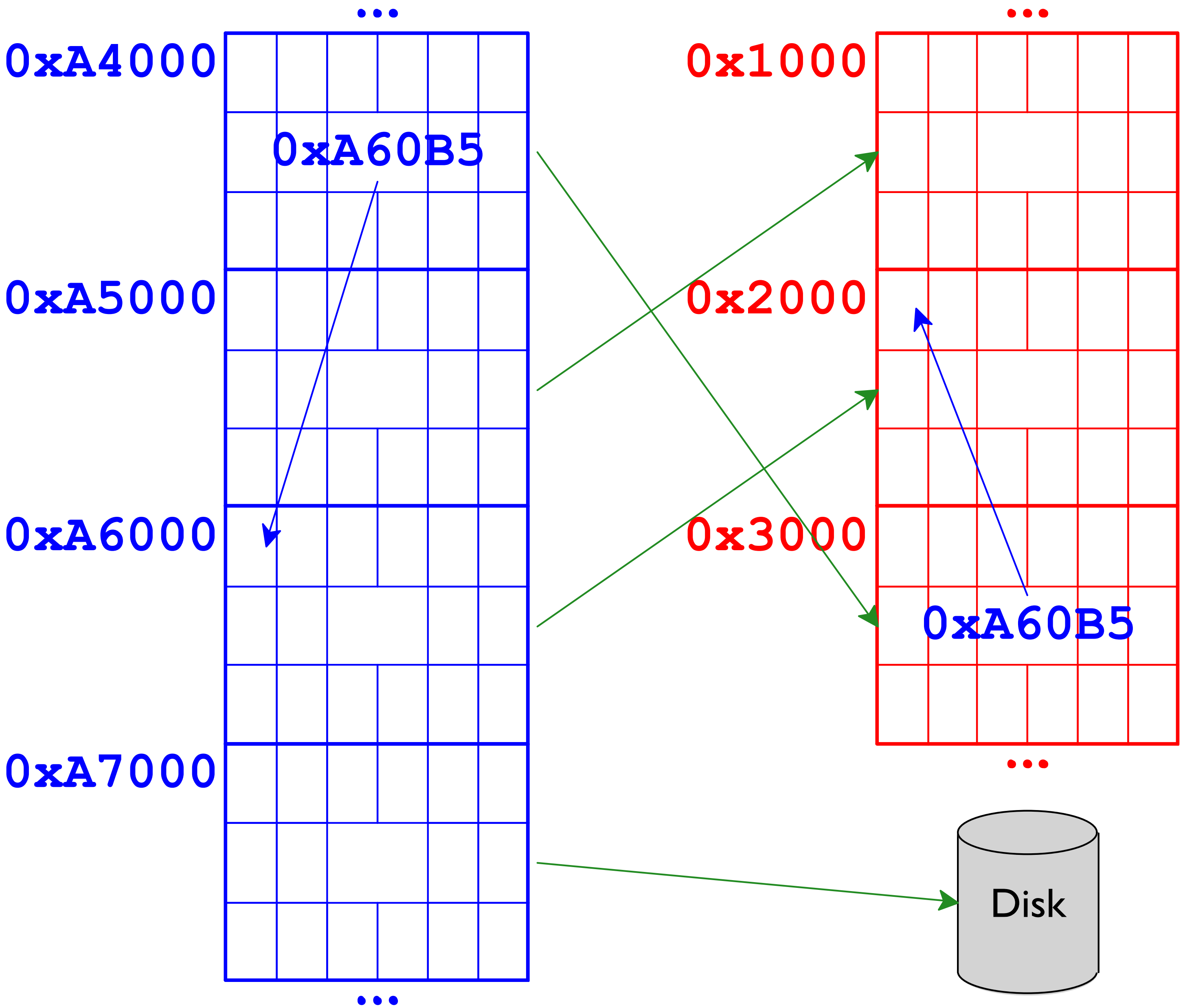


Translation Lookaside Buffer

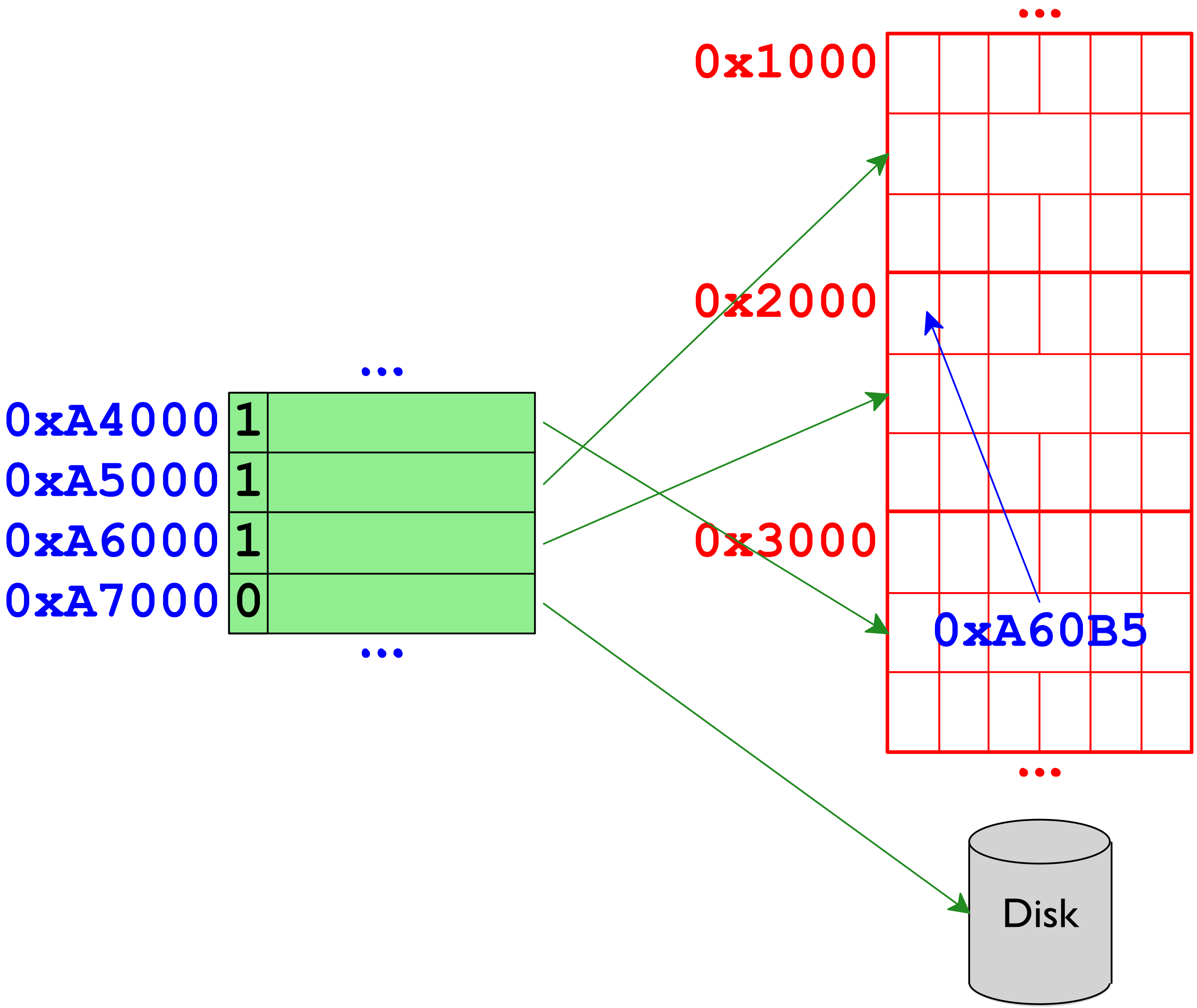
A ***translation lookaside buffer*** (TLB) is a custom cache for address translation



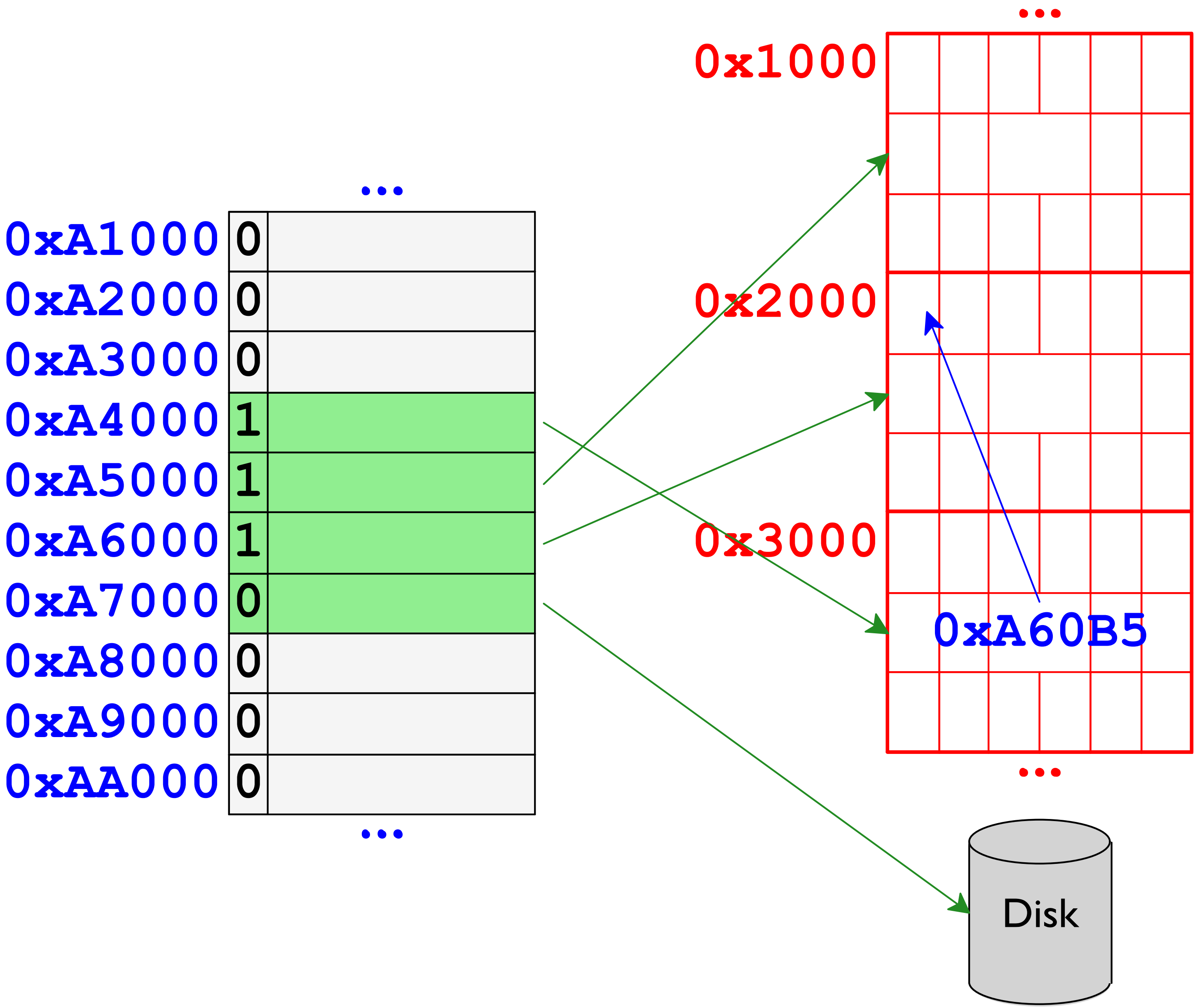
Virtual Memory: User vs. Kernel Views



Virtual Memory: User vs. Kernel Views



Virtual Memory: User vs. Kernel Views



...

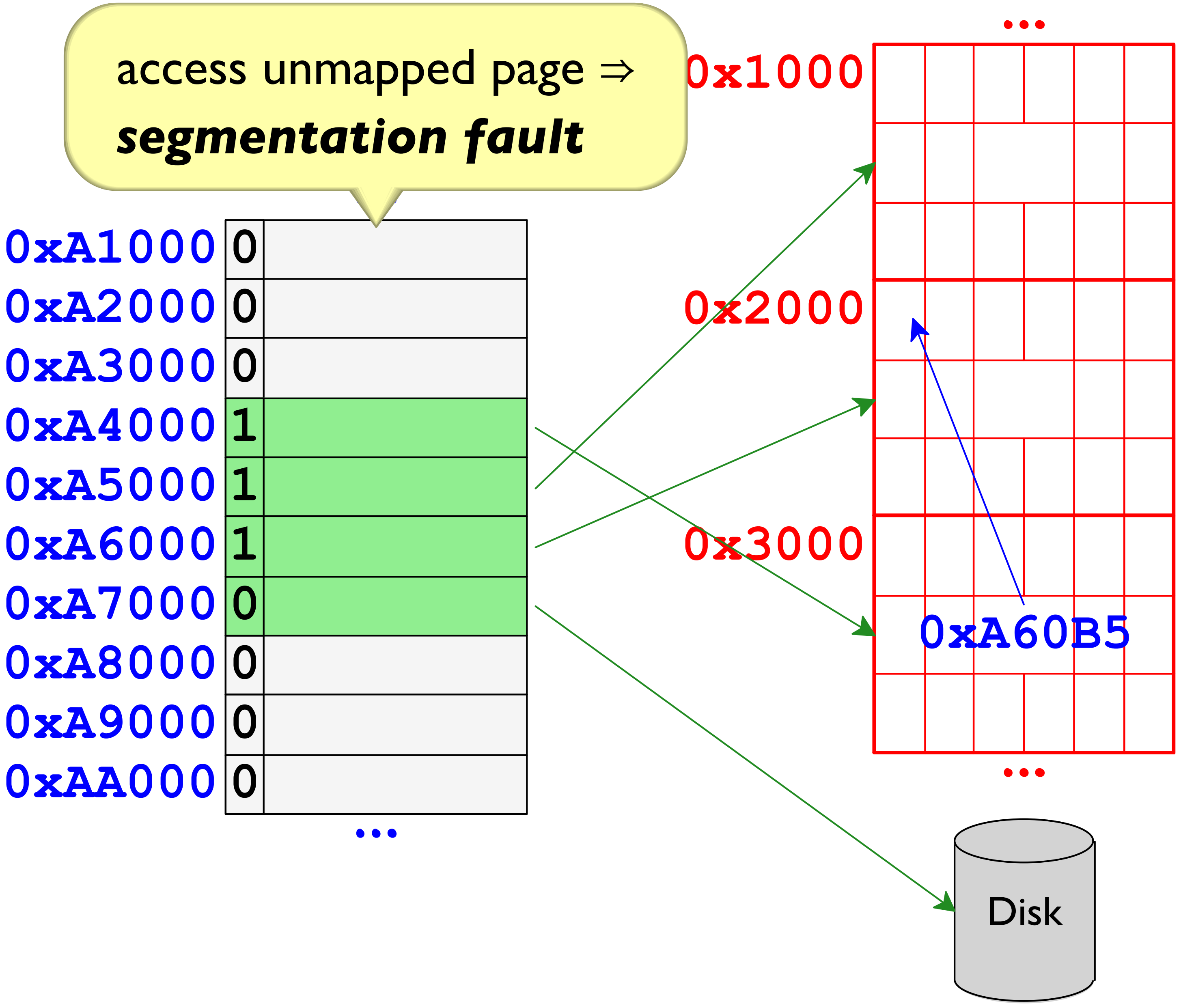
0xA1000	0
0xA2000	0
0xA3000	0
0xA4000	1
0xA5000	1
0xA6000	1
0xA7000	0
0xA8000	0
0xA9000	0
0xAA000	0

...

Virtual Memory: User vs. Kernel Views

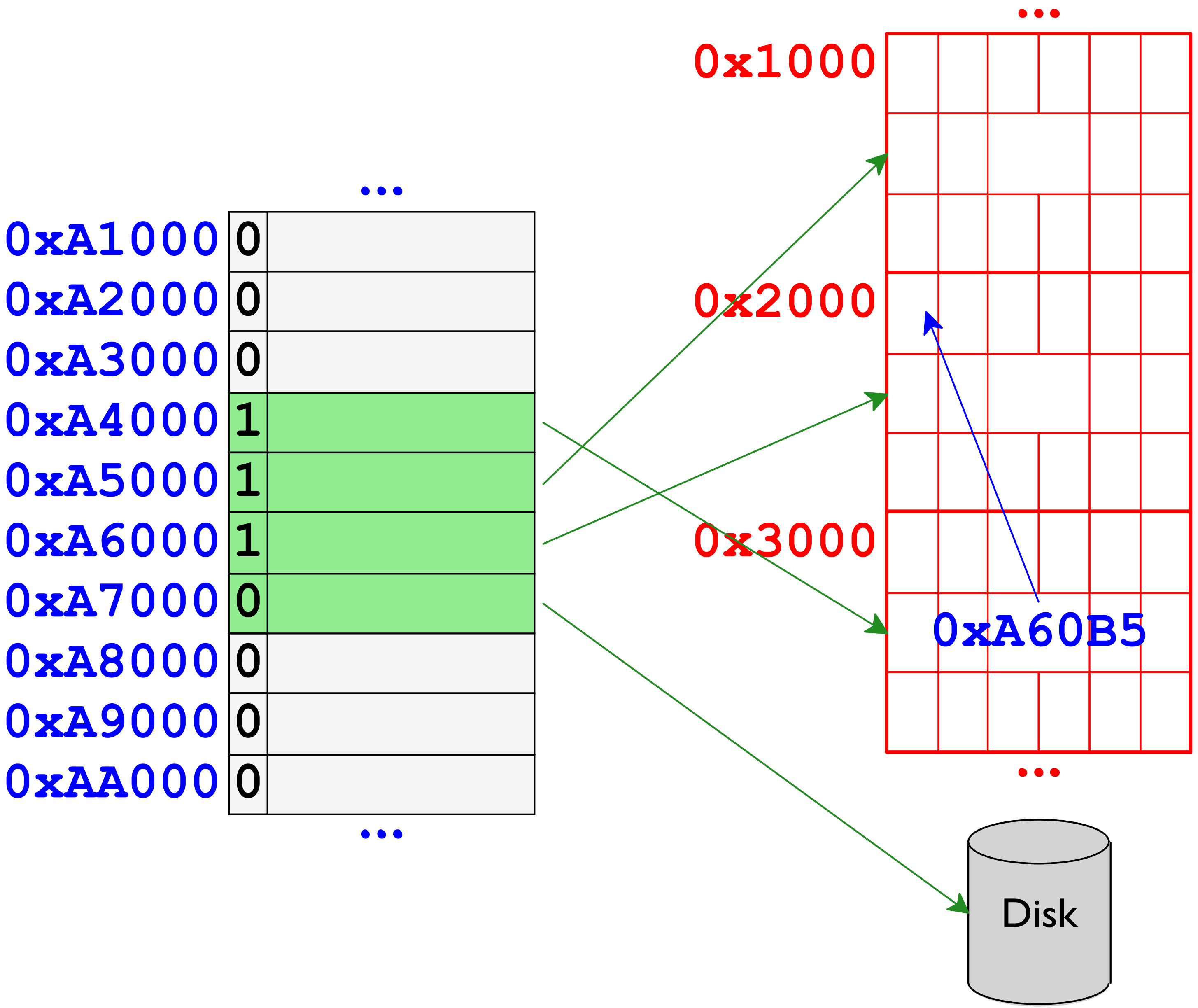
access unmapped page ⇒

segmentation fault

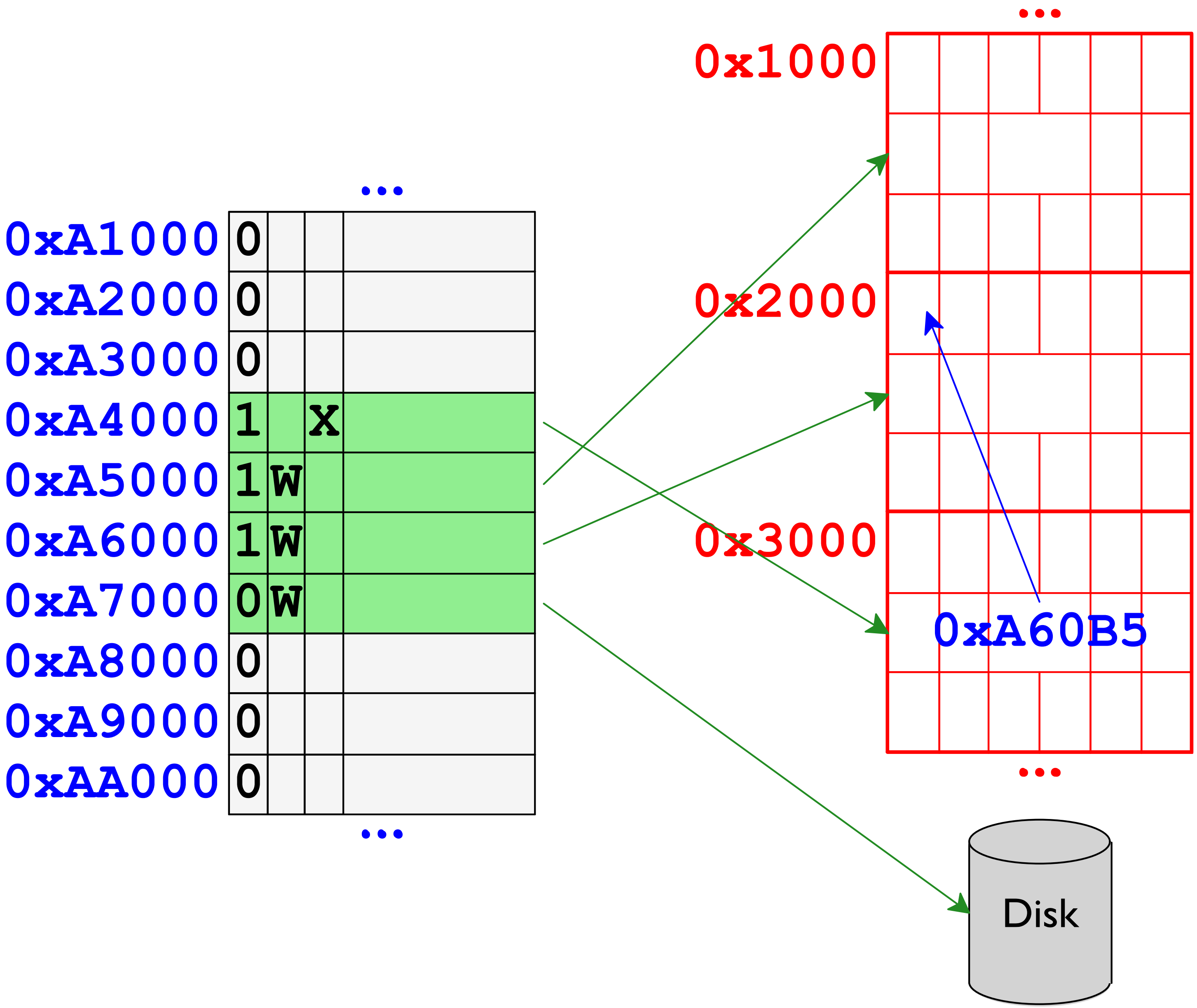


0xA1000	0
0xA2000	0
0xA3000	0
0xA4000	1
0xA5000	1
0xA6000	1
0xA7000	0
0xA8000	0
0xA9000	0
0xAA000	0

Virtual Memory: User vs. Kernel Views

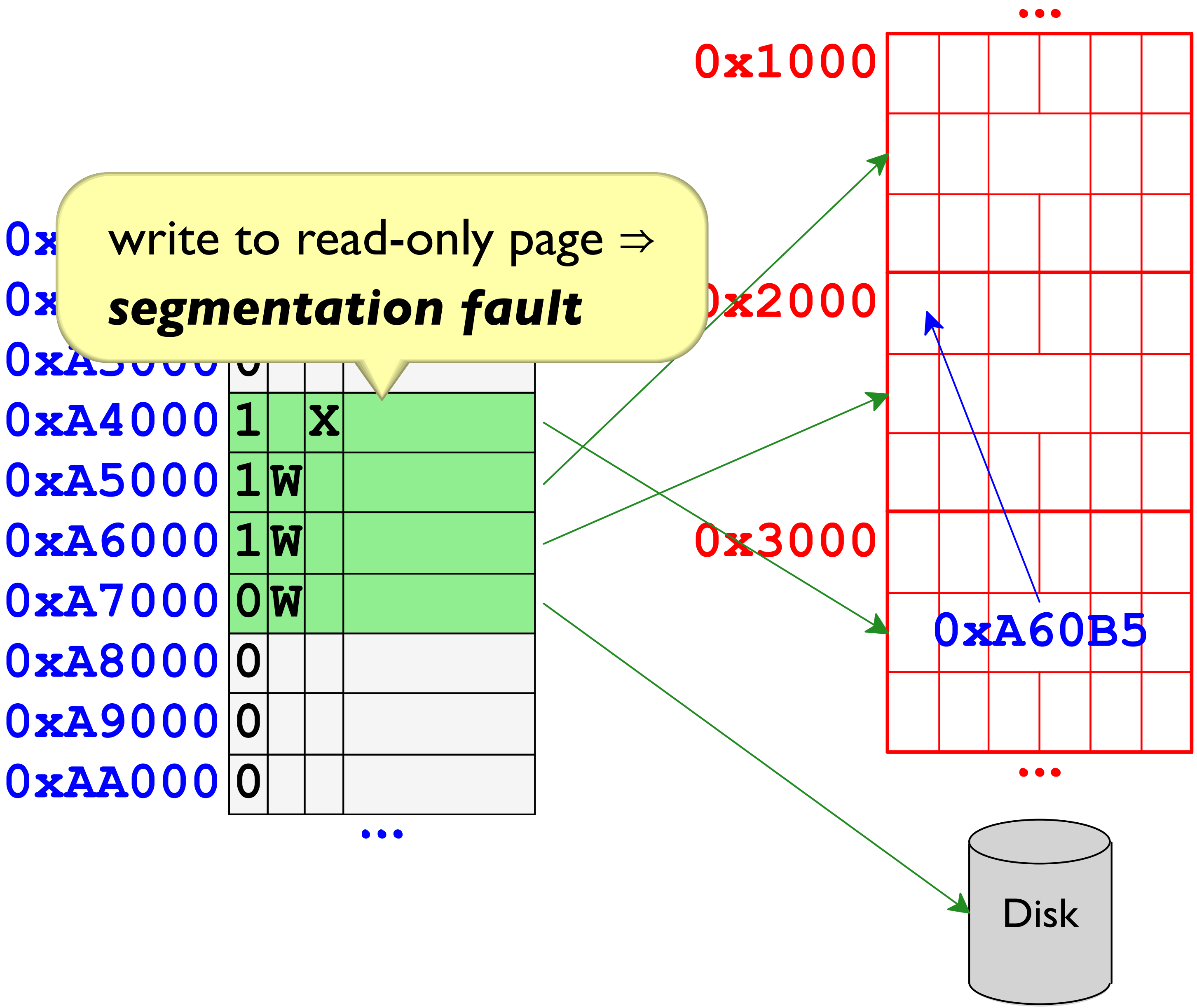


Virtual Memory: User vs. Kernel Views



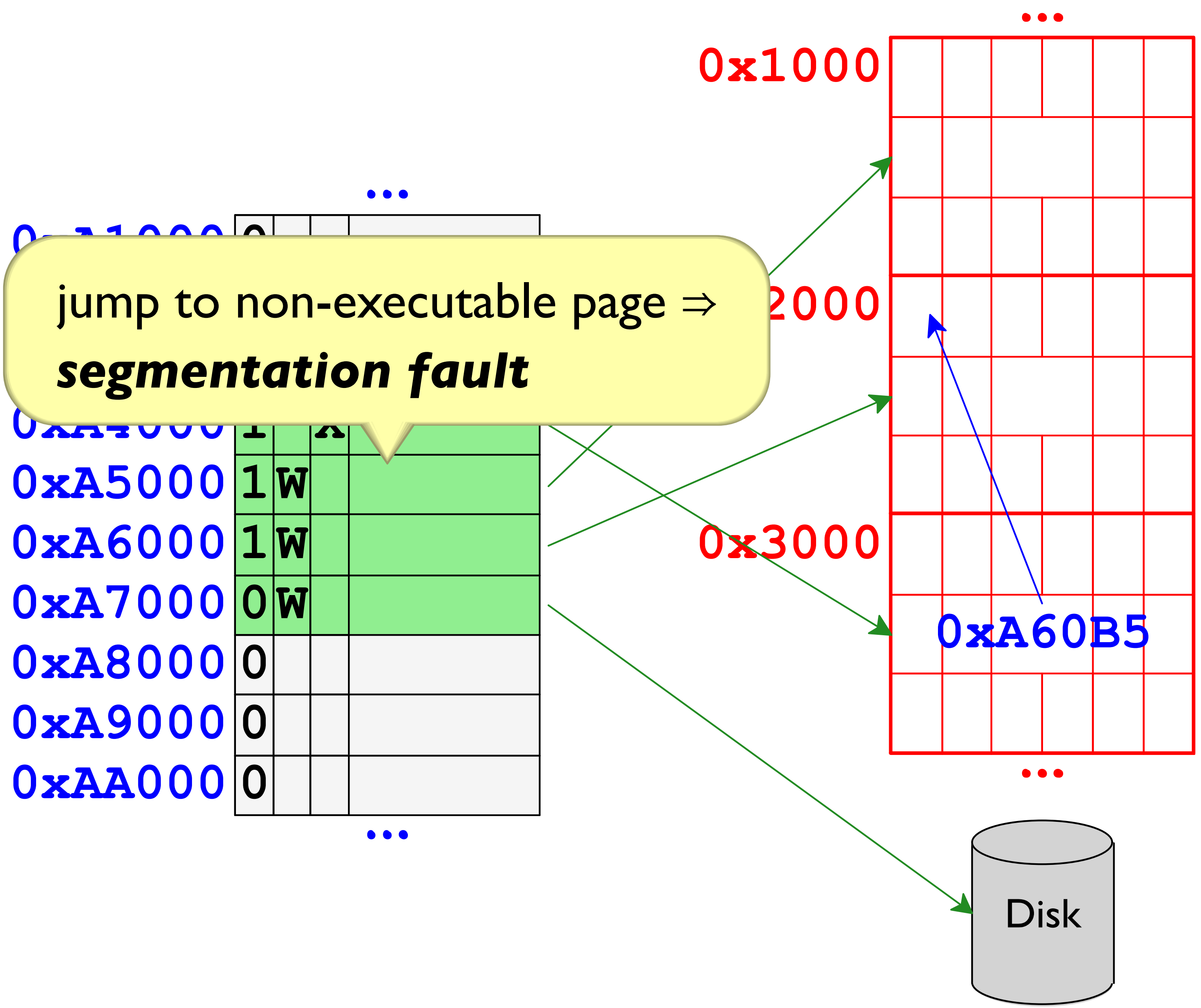
0xA1000	0			
0xA2000	0			
0xA3000	0			
0xA4000	1		X	
0xA5000	1	W		
0xA6000	1	W		
0xA7000	0	W		
0xA8000	0			
0xA9000	0			
0xAA000	0			

Virtual Memory: User vs. Kernel Views



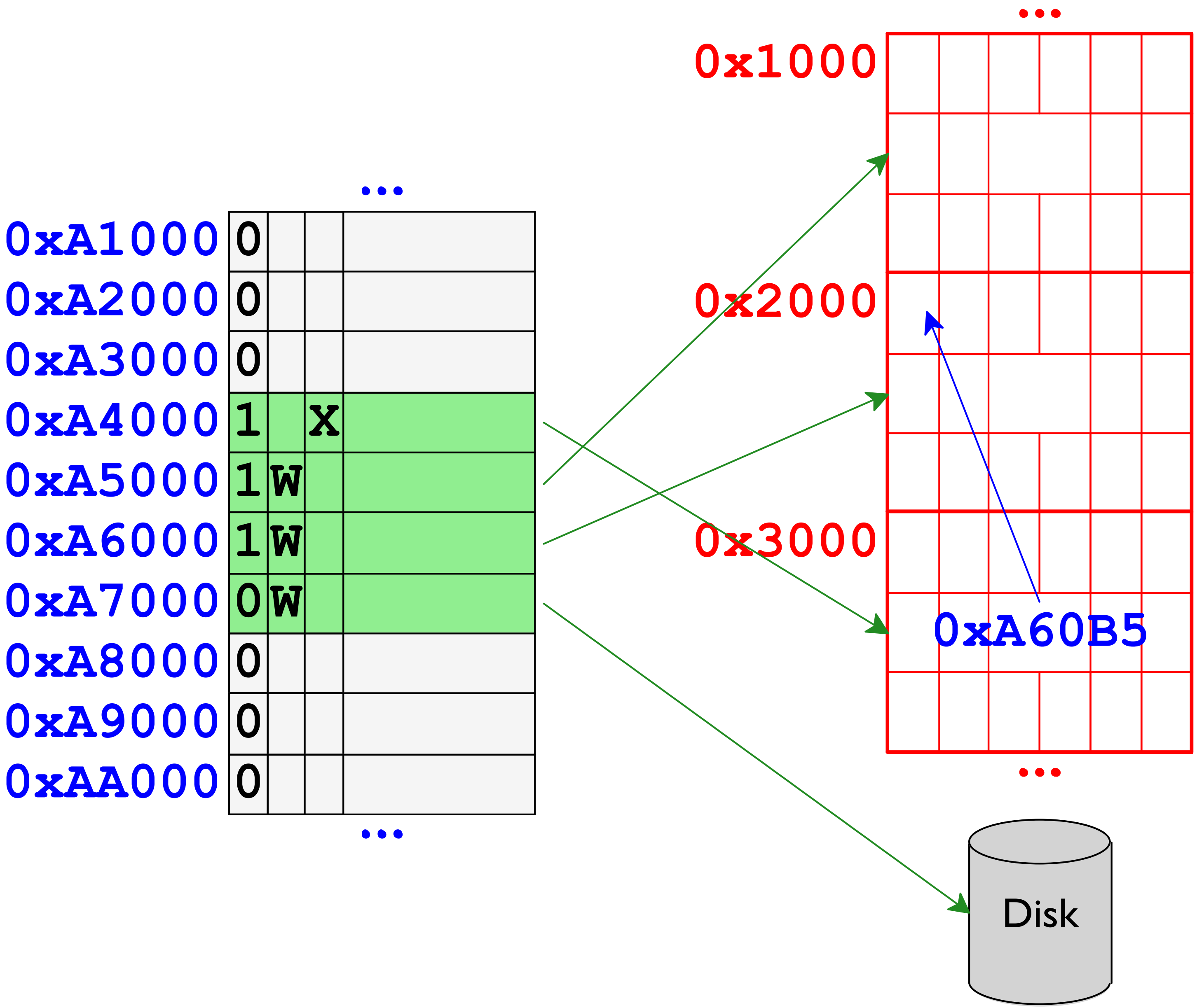
write to read-only page ⇒ ***segmentation fault***

Virtual Memory: User vs. Kernel Views



jump to non-executable page ⇒ ***segmentation fault***

Virtual Memory: User vs. Kernel Views



0xA1000	0			
0xA2000	0			
0xA3000	0			
0xA4000	1		X	
0xA5000	1	W		
0xA6000	1	W		
0xA7000	0	W		
0xA8000	0			
0xA9000	0			
0xAA000	0			

Trying to Write to Code Pages

```
#include "csapp.h"

int main() {
  int x = 8;

  *(int *)&x = 5;
  printf("ok\n");

  *(int *)main = 5;
  printf("not ok\n");

  return 0;
}
```

Copy

Trying to Write to Code Pages

```
#include "csapp.h"

int main() {
  int x = 8;

  *(int *)&x = 5;
  printf("ok\n");

  *(int *)main = 5;
  printf("not ok\n");

  return 0;
}
```

Copy

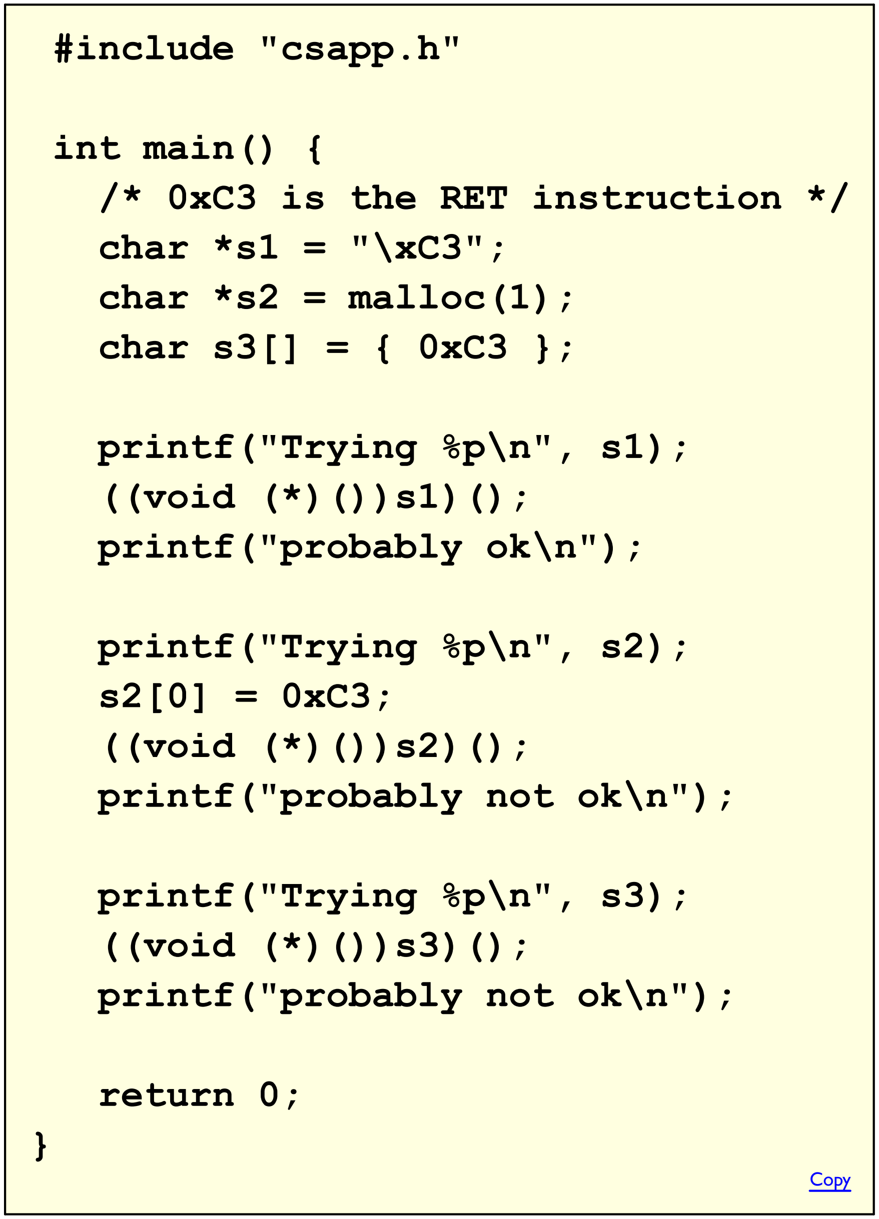
Fails, because page for **`main`** is not writable

Trying to Execute Other Memory

```
 #include "csapp.h"

 int main() {
   /* 0xC3 is the RET instruction */
   char *s1 = "\xC3";
   char *s2 = malloc(1);
   char s3[] = { 0xC3 };

   printf("Trying %p\n", s1);
   ((void (*)())s1)();
   printf("probably ok\n");

   printf("Trying %p\n", s2);
   s2[0] = 0xC3;
   ((void (*)())s2)();
   printf("probably not ok\n");

   printf("Trying %p\n", s3);
   ((void (*)())s3)();
   printf("probably not ok\n");

   return 0;
}
```

Copy

Trying to Execute Other Memory



```
 #include "csapp.h"

 int main() {
   /* 0xC3 is the RET instruction */
   char *s1 = "\xC3";
   char *s2 = malloc(1);
   char s3[] = { 0xC3 };

   printf("Trying %p\n", s1);
   ((void (*)())s1)();
   printf("probably ok\n");

   printf("Trying %p\n", s2);
   s2[0] = 0xC3;
   ((void (*)())s2)();
   printf("probably not ok\n");

   printf("Trying %p\n", s3);
   ((void (*)())s3)();
   printf("probably not ok\n");

   return 0;
}
```

Static data tends to be with executable code pages

Other memory is not executable by default

Syscall to Change the Page Table

```
#include <sys/mman.h>

void *mmap(void *addr, size_t length,
           int prot, int flags,
           int fd, off_t offset);

int munmap(void *addr, size_t length);
```

`mmap` changes the page table:

- **`addr`** — address to map or **`NULL`** for kernel choice
- **`length`** – bytes to map   rounded up to page size
- **`prot`** — bitwise **`PROT_{READ,WRITE,EXEC}`**
- **`flags`** — **`MAP_{PRIVATE,SHARED}`**, maybe **`MAP_ANON`**
- **`fd`** — file to map into memory if not **`MAP_ANON`**
- **`offset`** — offset into file

Syscall to Change the Page Table

```
#include <sys/mman.h>

void *mmap(void *addr, size_t length,
           int prot, int flags,
           int fd, off_t offset);

int munmap(void *addr, size_t length);
```

Read a file into memory (on demand):

```
fd = open(argv[1], O_RDONLY);
...
p = mmap(NULL, len,
         PROT_READ, MAP_PRIVATE,
         fd, 0);
```

Syscall to Change the Page Table

```
#include <sys/mman.h>

void *mmap(void *addr, size_t length,
           int prot, int flags,
           int fd, off_t offset);

int munmap(void *addr, size_t length);
```

Allocate a fresh page of memory:

```
p = mmap(NULL, getpagesize(),
         PROT_READ | PROT_WRITE,
         MAP_PRIVATE + MAP_ANON,
         -1, 0);
```

`MAP_ANON` with **`-1`** means “not from a file”

Using mmap

```
#include "csapp.h"

int main() {
  char *s;
  size_t sz = 1<<14;

  s = Mmap(0, sz,
           PROT_READ | PROT_WRITE | PROT_EXEC,
           MAP_PRIVATE | MAP_ANON,
           -1, 0);

  printf("Trying %p\n", s);
  s[0] = 0xC3;
  ((void (*)())s)();
  printf("ok\n");

  return 0;
}
```

Copy

Changing Page Protection

```
#include <sys/mman.h>

int mprotect(void *addr, size_t len, int prot);
```

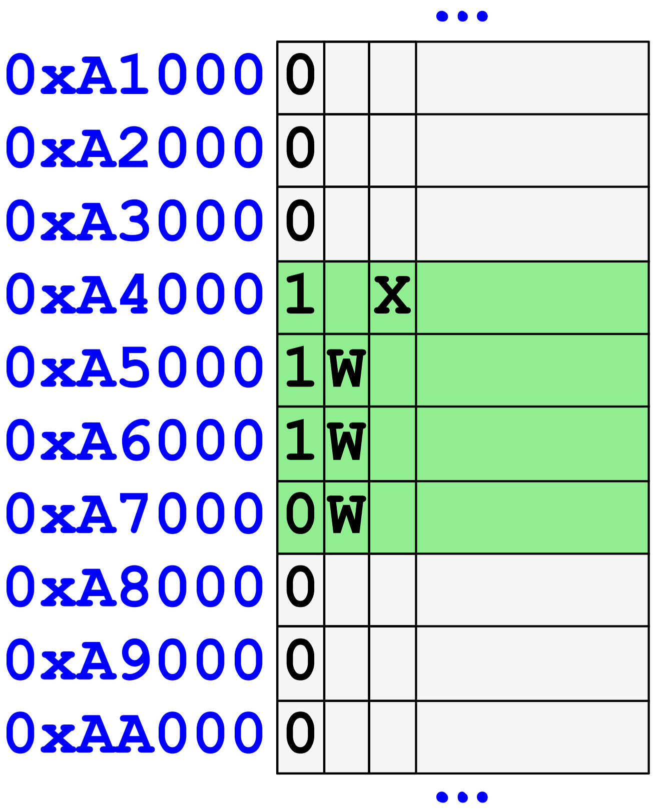
`mprotect` changes the protection of previously **`mmap`**ped pages

Using mprotect

```
#include "csapp.h"

int main() {
  char *s;
  size_t sz = 1<<14;

  s = Mmap(0, sz,
           PROT_READ | PROT_WRITE,
           MAP_PRIVATE | MAP_ANON,
           -1, 0);
  s[0] = 0xC3;

  Mprotect(s, sz, PROT_READ | PROT_EXEC);

  ((void (*)())s)();
  printf("ok\n");

  s[0] = 0x0;
  printf("not ok\n");

  return 0;
}
```

Copy

Segmentation Fault



…

0xA1000	0			
0xA2000	0			
0xA3000	0			
0xA4000	1		X	
0xA5000	1	W		
0xA6000	1	W		
0xA7000	0	W		
0xA8000	0			
0xA9000	0			
0xAA000	0			

…

Any of these trigger an exception:

- Read of unmapped page
- Write to read-only page
- Jump to non-executable page

Kernel handles the exception by sending a **`SIGSEGV`** signal

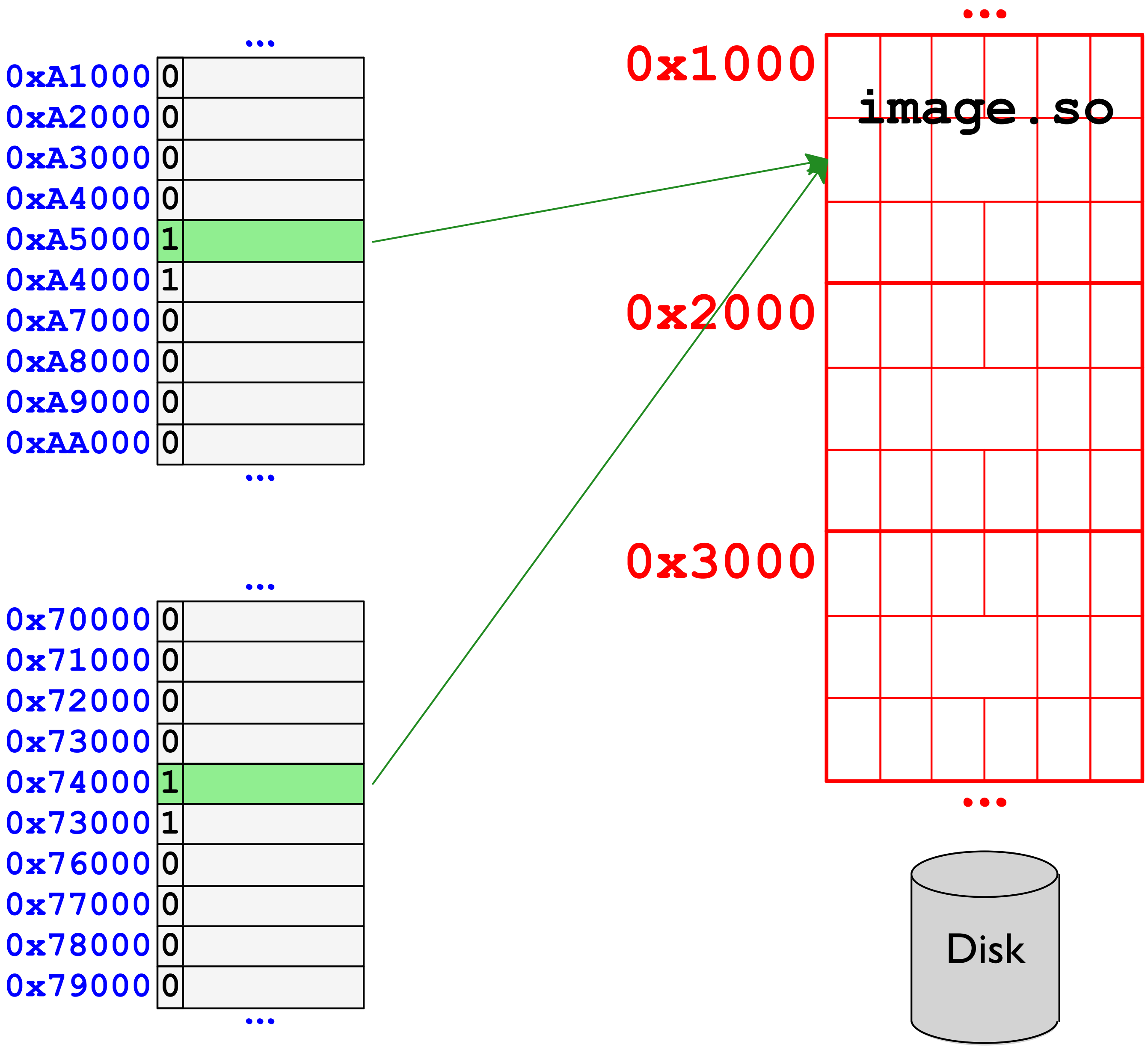
default handler prints “Segmentation Fault” and exits

Handling SIGSEGV

```
#include "csapp.h"

static char *s;
static size_t sz = 1<<14;

static void recover(int sig) {
  sio_puts("ouch...\n");
  Mprotect(s, sz, PROT_READ | PROT_WRITE);
}

int main() {
  s = Mmap(0, sz,
           PROT_READ | PROT_EXEC,
           MAP_PRIVATE | MAP_ANON,
           -1, 0);

  Signal(SIGSEGV, recover);

  s[0] = 0x0;
  printf("ok after all\n");

  return 0;
}
```

Copy

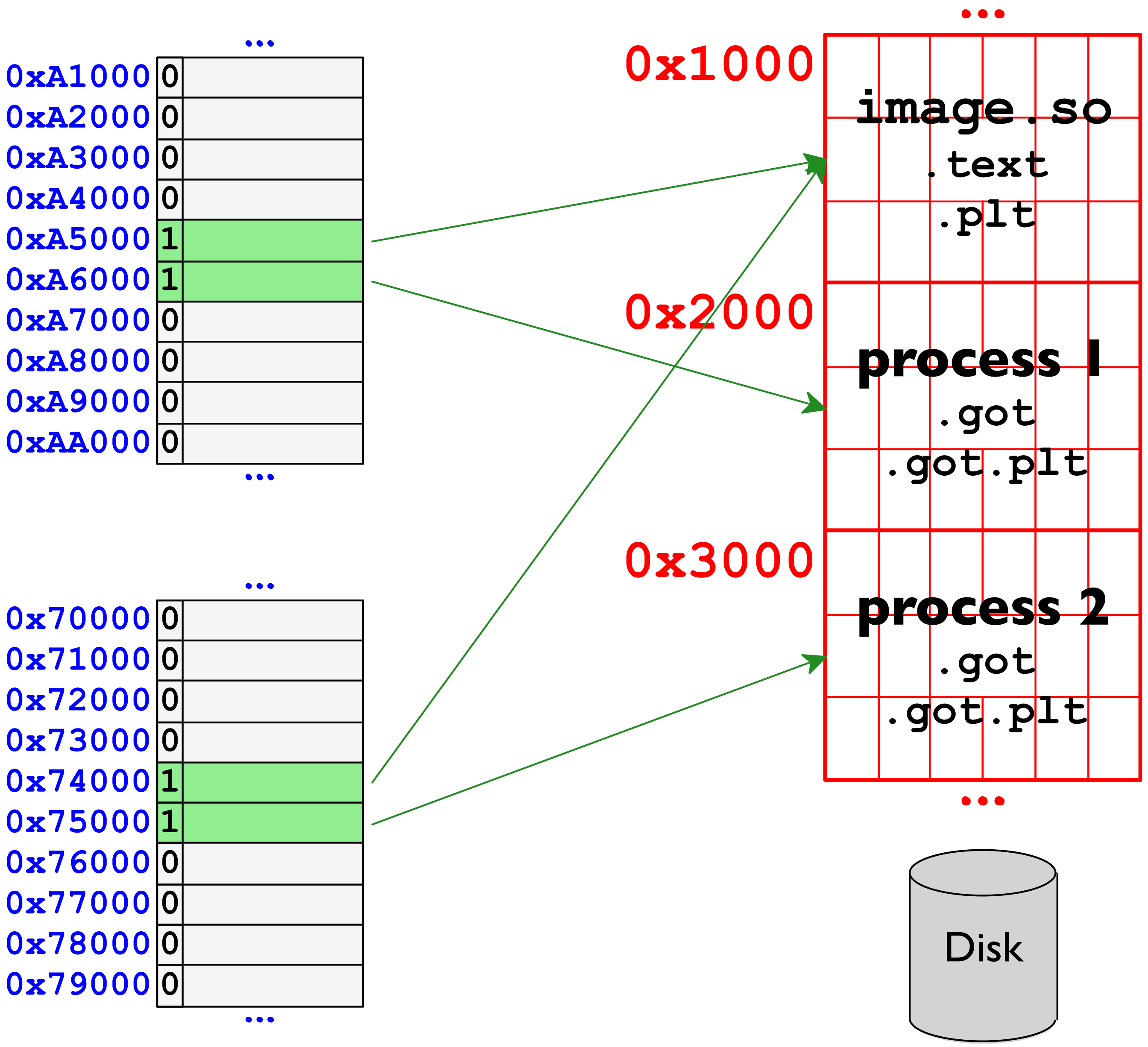
Sharing Position-Independent Code



…		
0xA1000	0	
0xA2000	0	
0xA3000	0	
0xA4000	0	
0xA5000	1	🟩
0xA4000	1	
0xA7000	0	
0xA8000	0	
0xA9000	0	
0xAA000	0	
…		

…		
0x70000	0	
0x71000	0	
0x72000	0	
0x73000	0	
0x74000	1	🟩
0x73000	1	
0x76000	0	
0x77000	0	
0x78000	0	
0x79000	0	
…		

Sharing Position-Independent Code



...

0xA1000	0	
0xA2000	0	
0xA3000	0	
0xA4000	0	
0xA5000	1	🟩
0xA6000	1	🟩
0xA7000	0	
0xA8000	0	
0xA9000	0	
0xAA000	0	

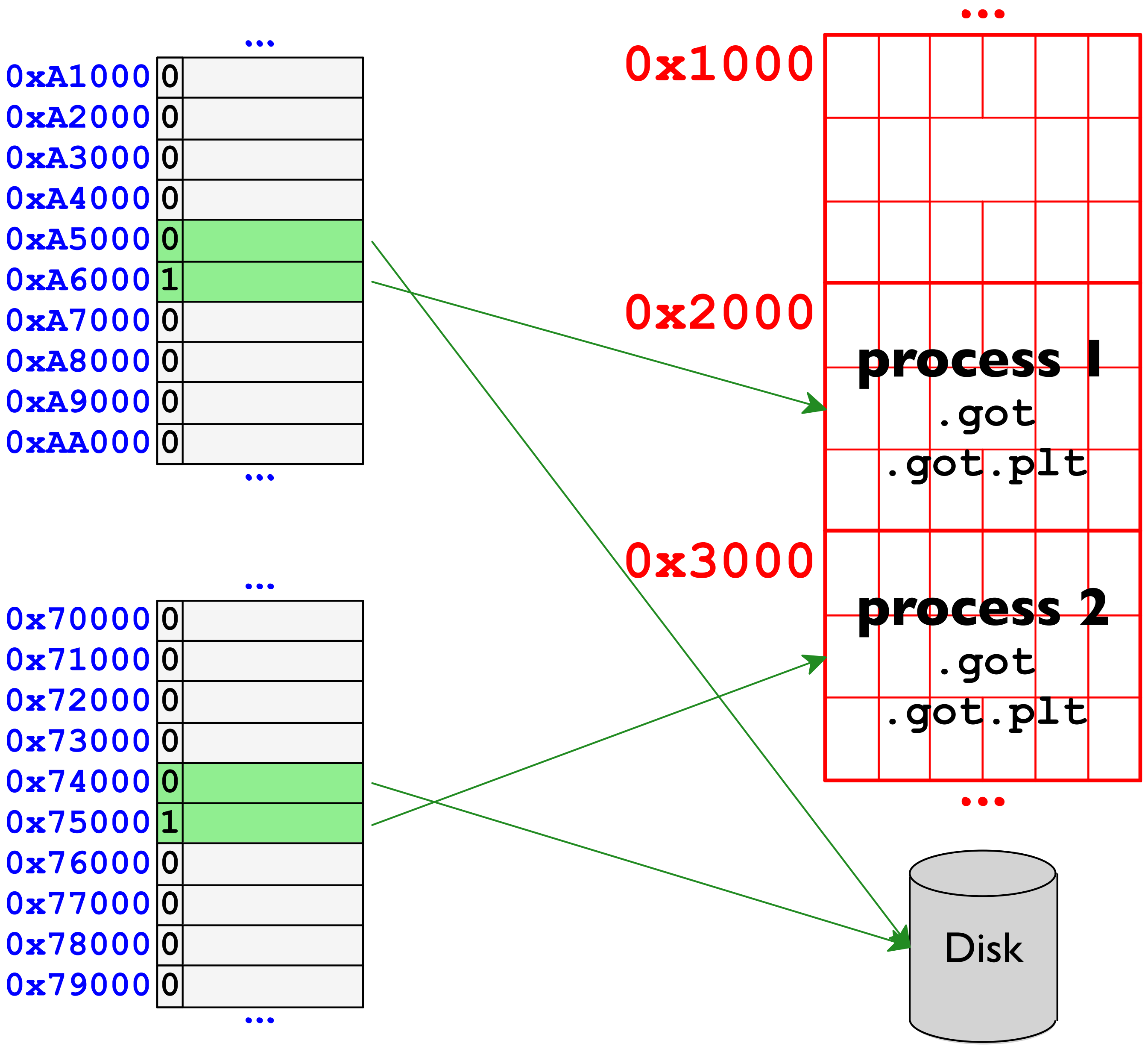
...

...

0x70000	0	
0x71000	0	
0x72000	0	
0x73000	0	
0x74000	1	🟩
0x75000	1	🟩
0x76000	0	
0x77000	0	
0x78000	0	
0x79000	0	

...

Sharing Position-Independent Code



…

0xA1000	0	
0xA2000	0	
0xA3000	0	
0xA4000	0	
0xA5000	0	🟩
0xA6000	1	🟩
0xA7000	0	
0xA8000	0	
0xA9000	0	
0xAA000	0	

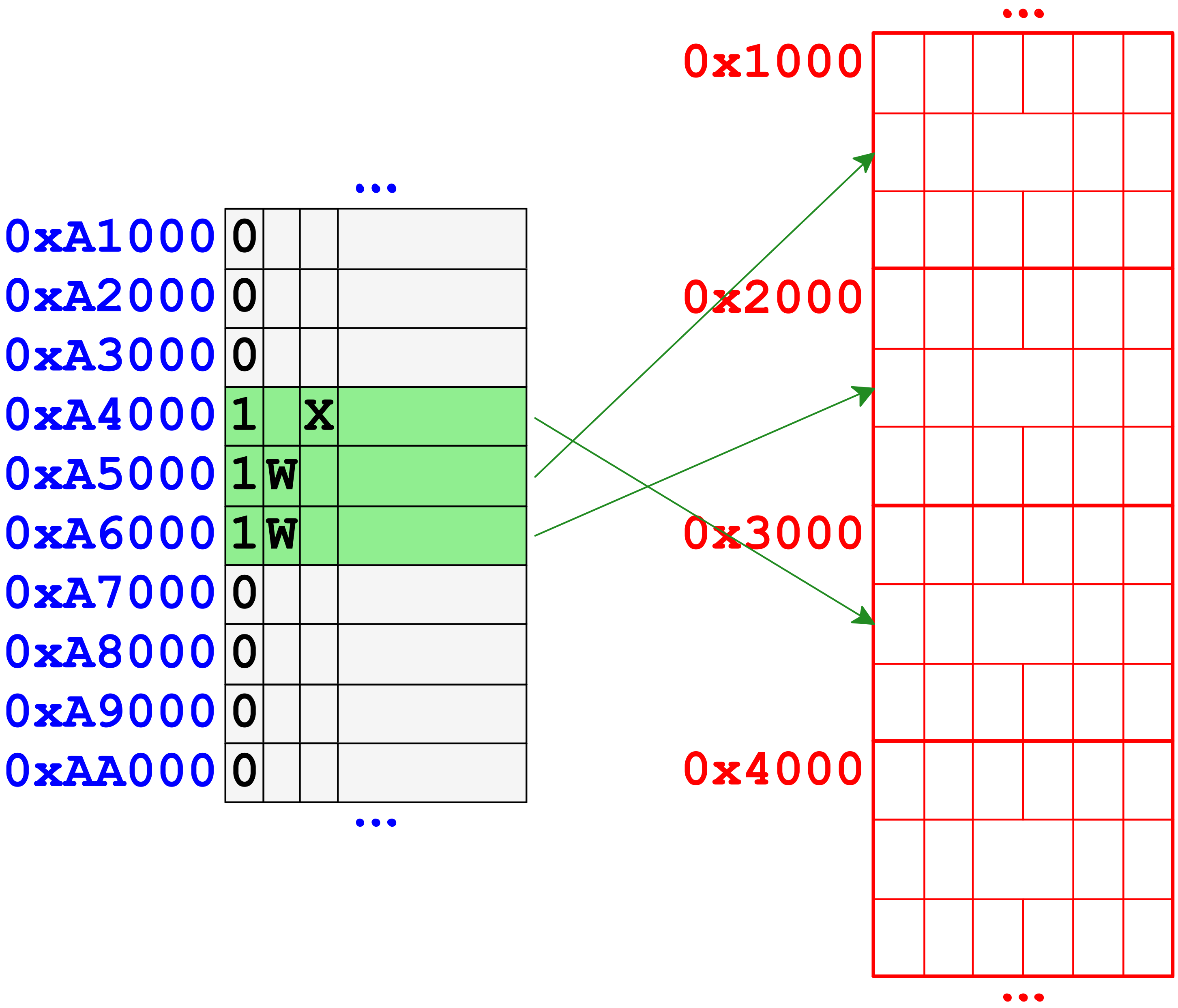
…

…

0x70000	0	
0x71000	0	
0x72000	0	
0x73000	0	
0x74000	0	🟩
0x75000	1	🟩
0x76000	0	
0x77000	0	
0x78000	0	
0x79000	0	

…

Virtual Memory and `fork`

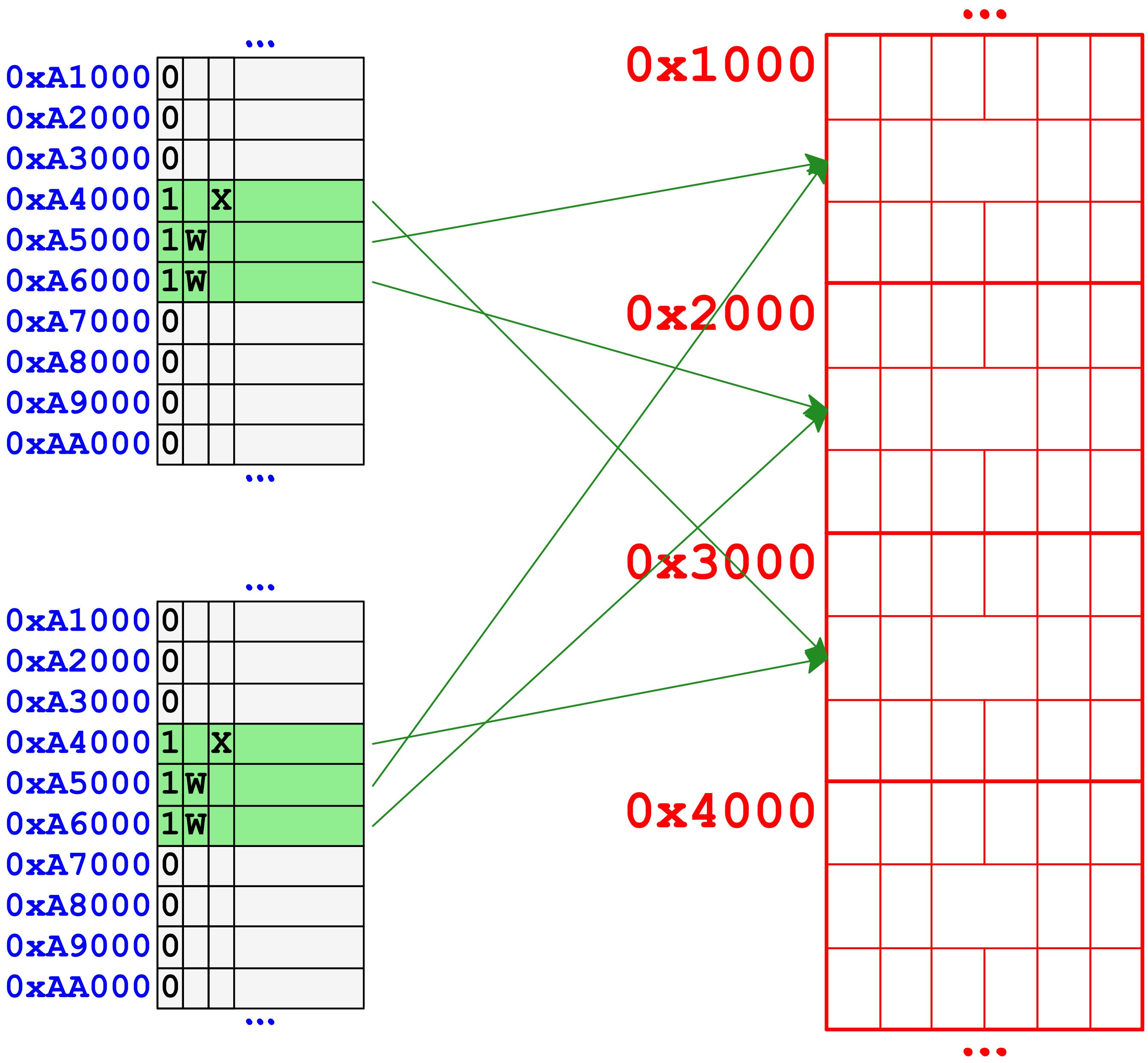


…

0xA1000	0			
0xA2000	0			
0xA3000	0			
0xA4000	1		X	
0xA5000	1	W		
0xA6000	1	W		
0xA7000	0			
0xA8000	0			
0xA9000	0			
0xAA000	0			

…

Virtual Memory and `fork`



…

0xA1000	0			
0xA2000	0			
0xA3000	0			
0xA4000	1		X	
0xA5000	1	W		
0xA6000	1	W		
0xA7000	0			
0xA8000	0			
0xA9000	0			
0xAA000	0			

…

…

0xA1000	0			
0xA2000	0			
0xA3000	0			
0xA4000	1		X	
0xA5000	1	W		
0xA6000	1	W		
0xA7000	0			
0xA8000	0			
0xA9000	0			
0xAA000	0			

…

…

0x1000

0x2000

0x3000

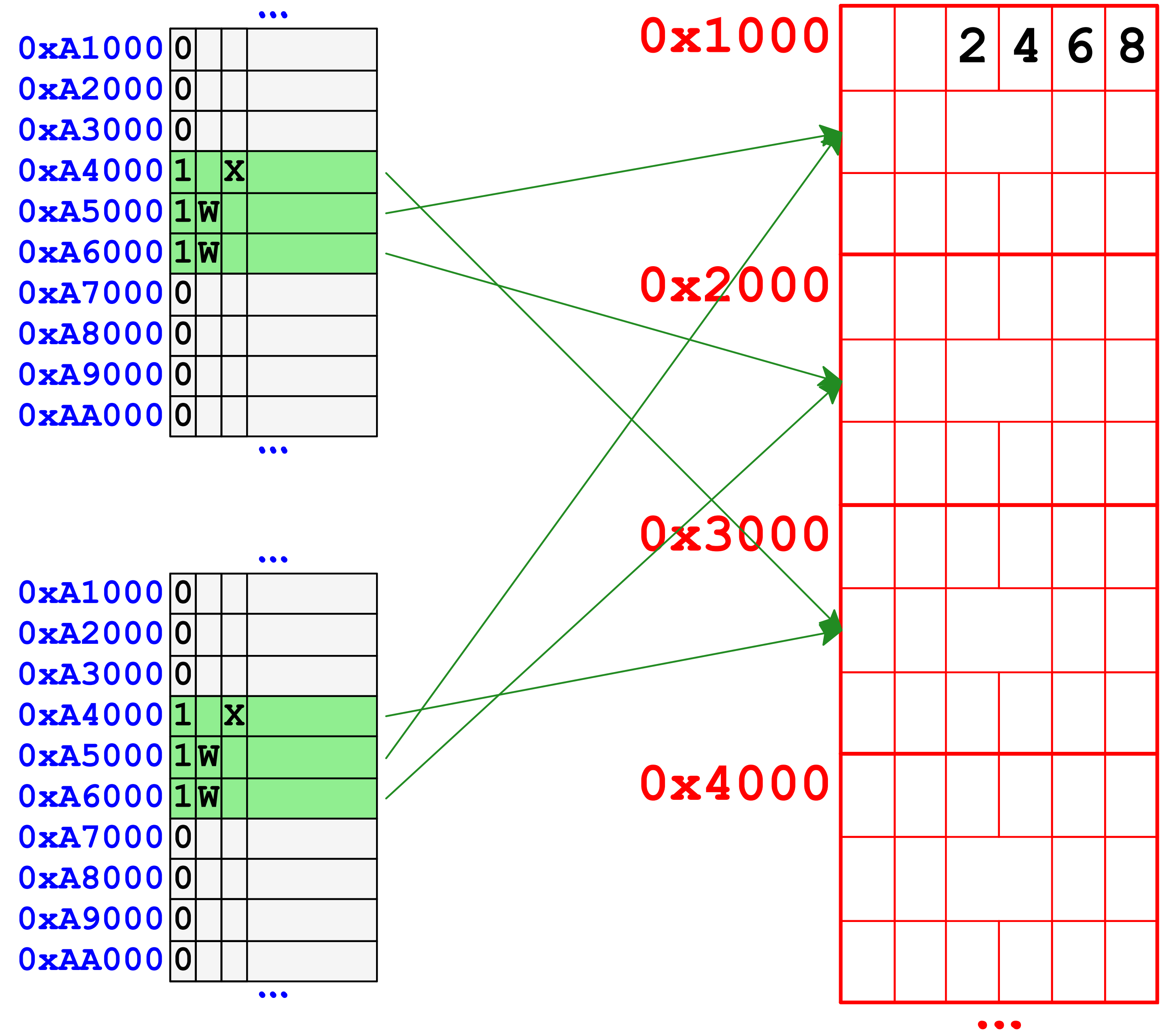
0x4000

…

Virtual Memory and `fork`

```
char a[] = {2, 4, 6, 8};
```

at 0xA5002



…

0xA1000	0			
0xA2000	0			
0xA3000	0			
0xA4000	1		X	
0xA5000	1	W		
0xA6000	1	W		
0xA7000	0			
0xA8000	0			
0xA9000	0			
0xAA000	0			

…

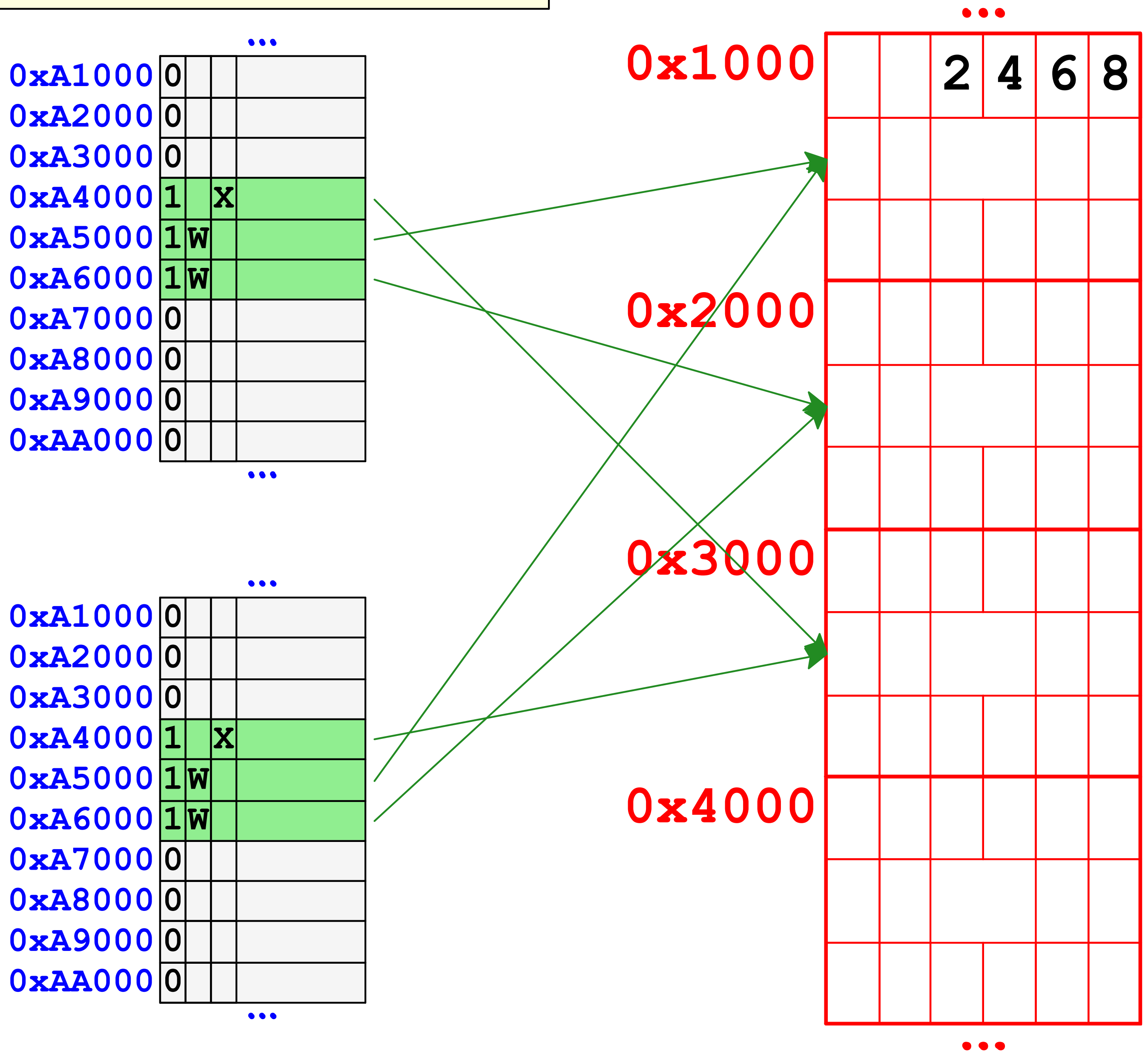
…

0xA1000	0			
0xA2000	0			
0xA3000	0			
0xA4000	1		X	
0xA5000	1	W		
0xA6000	1	W		
0xA7000	0			
0xA8000	0			
0xA9000	0			
0xAA000	0			

…

Virtual Memory and `fork`

`char a[] = {2, 4, 6, 8};` at 0xA5002



...

0xA1000	0			
0xA2000	0			
0xA3000	0			
0xA4000	1		X	
0xA5000	1	W		
0xA6000	1	W		
0xA7000	0			
0xA8000	0			
0xA9000	0			
0xAA000	0			

...

...

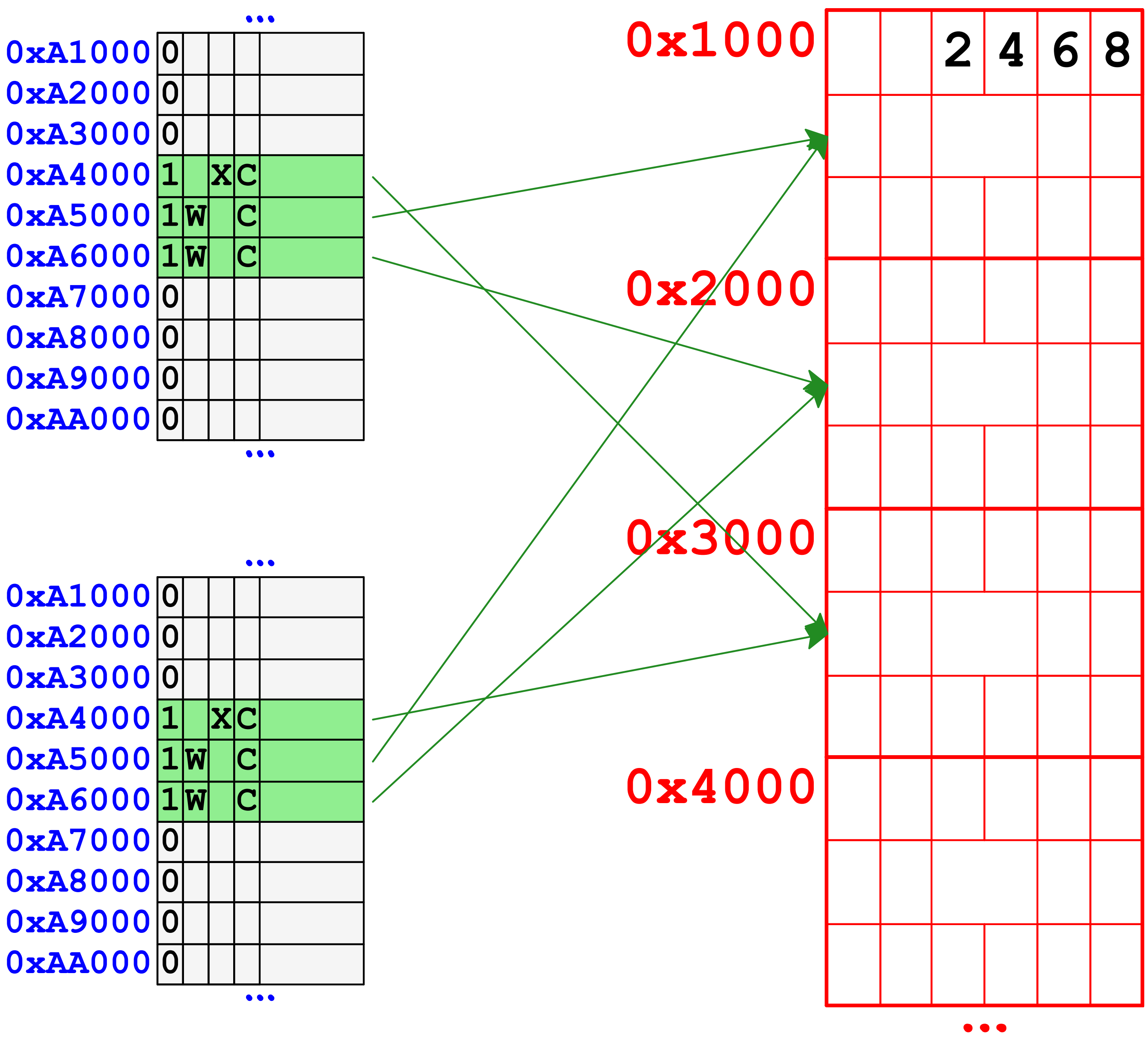
0xA1000	0			
0xA2000	0			
0xA3000	0			
0xA4000	1		X	
0xA5000	1	W		
0xA6000	1	W		
0xA7000	0			
0xA8000	0			
0xA9000	0			
0xAA000	0			

...

`a[2] = 7`

Virtual Memory and `fork`

`char a[] = {2, 4, 6, 8};` at 0xA5002



…

0xA1000	0			
0xA2000	0			
0xA3000	0			
0xA4000	1		X	C
0xA5000	1	W		C
0xA6000	1	W		C
0xA7000	0			
0xA8000	0			
0xA9000	0			
0xAA000	0			

…

…

0xA1000	0			
0xA2000	0			
0xA3000	0			
0xA4000	1		X	C
0xA5000	1	W		C
0xA6000	1	W		C
0xA7000	0			
0xA8000	0			
0xA9000	0			
0xAA000	0			

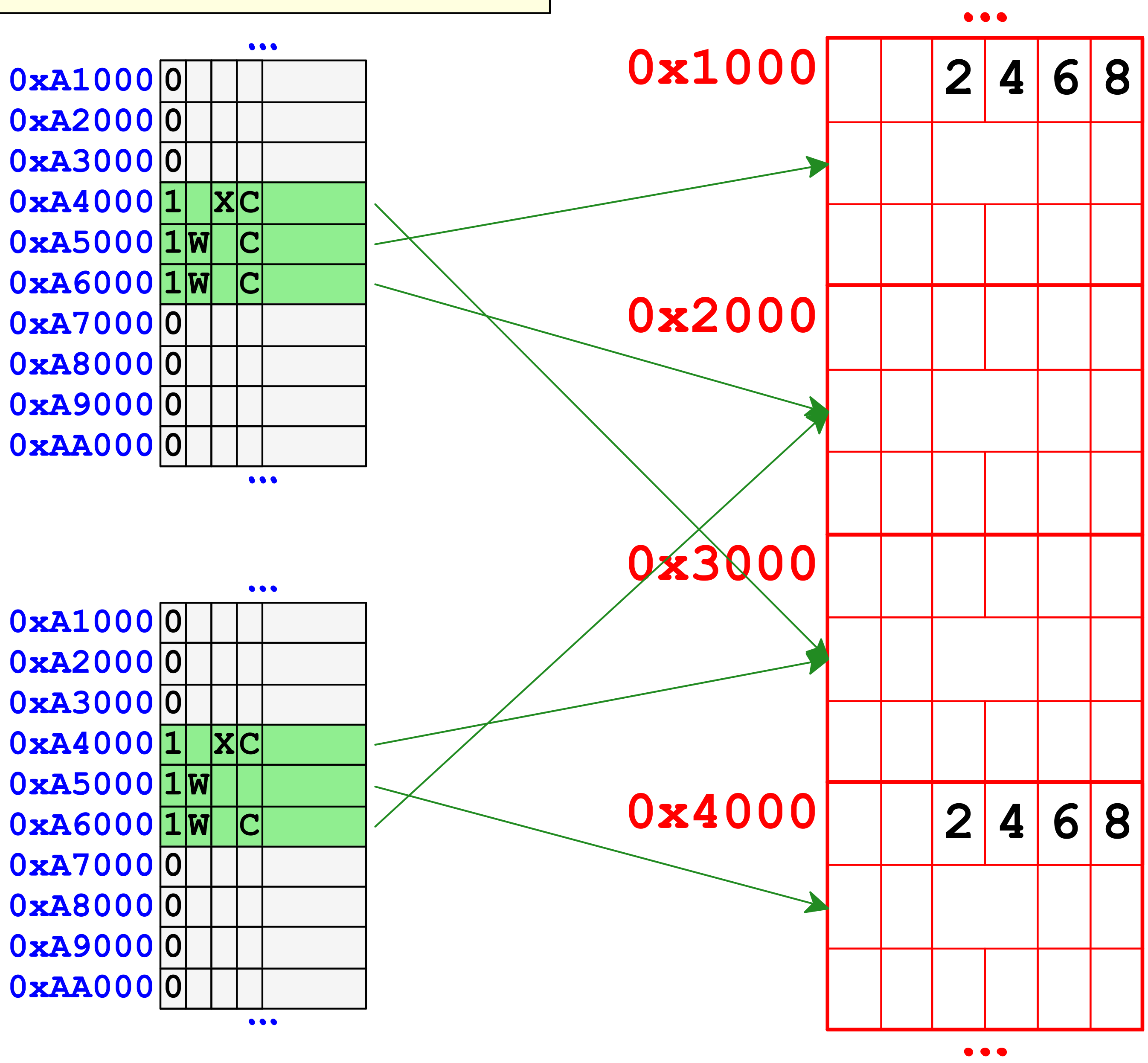
…

`a[2] = 7`

Virtual Memory and `fork`

```
char a[] = {2, 4, 6, 8};
```

at 0xA5002



…

0xA1000	0				
0xA2000	0				
0xA3000	0				
0xA4000	1		X	C	
0xA5000	1	W		C	
0xA6000	1	W		C	
0xA7000	0				
0xA8000	0				
0xA9000	0				
0xAA000	0				

…

```
a[2] = 7
```

…

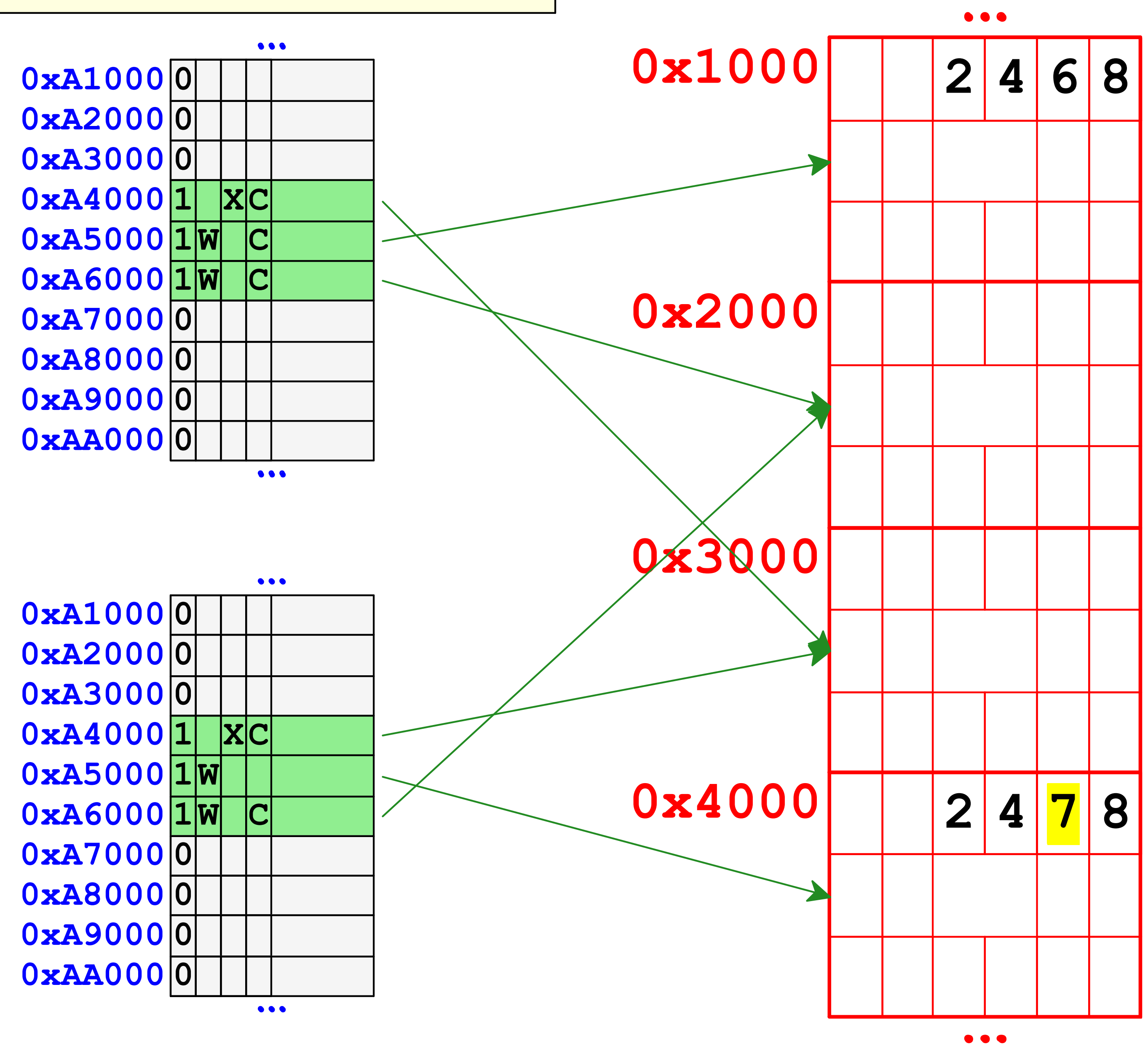
0xA1000	0				
0xA2000	0				
0xA3000	0				
0xA4000	1		X	C	
0xA5000	1	W			
0xA6000	1	W		C	
0xA7000	0				
0xA8000	0				
0xA9000	0				
0xAA000	0				

…

Virtual Memory and `fork`

```
char a[] = {2, 4, 6, 8};
```

at 0xA5002



...

0xA1000	0				
0xA2000	0				
0xA3000	0				
0xA4000	1		X	C	
0xA5000	1	W		C	
0xA6000	1	W		C	
0xA7000	0				
0xA8000	0				
0xA9000	0				
0xAA000	0				

...

...

0xA1000	0				
0xA2000	0				
0xA3000	0				
0xA4000	1		X	C	
0xA5000	1	W			
0xA6000	1	W		C	
0xA7000	0				
0xA8000	0				
0xA9000	0				
0xAA000	0				

...

```
a[2] = 7
```

Sharing Pages between Processes

```
#include "csapp.h"

int main() {
  char *s;
  size_t sz = 1<<14;

  s = Mmap(0, sz,
           PROT_READ | PROT_WRITE | PROT_EXEC,
           MAP_SHARED | MAP_ANON,
           -1, 0);
  s[0] = 1;

  if (Fork() == 0)
    s[0] = 2;
  else
    Wait(NULL);

  printf("%d at %p\n", s[0], s);

  return 0;
}
```

Copy

Sharing Pages between Processes

```
#include "csapp.h"

int main() {
  char *s;
  size_t sz = 1<<14;

  s = Mmap(0, sz,
           PROT_READ | PROT_WRITE | PROT_EXEC,
           MAP_SHARED | MAP_ANON,
           -1, 0);
  s[0] = 1;

  if (Fork() == 0)
    s[0] = 2;
  else
    Wait(NULL);

  printf("%d at %p\n", s[0], s);

  return 0;
}
```

Copy

Using **`MAP_SHARED`** effectively disables the copy-on-write flag that’s otherwise set by **`fork`**