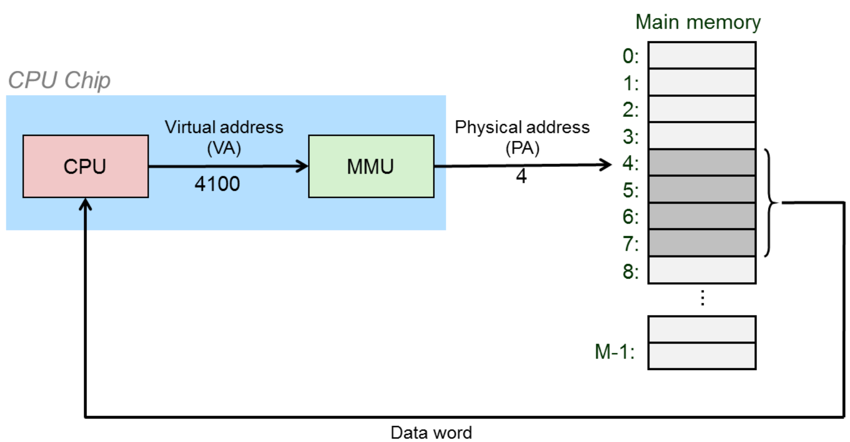
The Storage Hierarchy

Principles of Virtual Memory



Module Outline

- Where do Functions & Variables go?
- Motivation
- The Basics
- Features of Virtual Memory
- Back to Our Example Program
- On-Demand Paging
- Module Summary

```
$ ./layout
[7375] Memory addresses
  &foo():         0x5621d0284400
  &bar():         0x5621d0284410
  &main():        0x5621d02840f0

  &global_int:    0x5621d0287080
  &global_arr:    0x5621d02870c0
  &global_ptr:    0x5621d06870c0

  &local_int:     0x7fff202b87c0
  &local_arr:     0x7fff202b87b8
  &local_ptr:     0x7fff202b87bc

   global_ptr:    0x7fe24e283010
```

Where do Functions & Variables go?

Where do Functions & Variables go?

```c
…

int global_int = 0x1;
int global_arr[N];
int *global_ptr;
int *shared_int;

void foo(void) { … }

void bar(unsigned int key, int val) { … }

int main(int argc, char *argv[])
{
  int local_arr[1024];
  int local_k, local_v;

  …
```

```c
  printf("[%d] Memory addresses\n"
         "  &foo():        %p\n"
         "  &bar():        %p\n"
         "  &main():       %p\n"
         "\n"
         "  &global_int:   %p\n"
         "  &global_arr:   %p\n"
         "  &global_ptr:   %p\n"
         "\n"
         "  &local_int:    %p\n"
         "  &local_arr:    %p\n"
         "  &local_ptr:    %p\n"
         "\n"
         "   global_ptr:   %p\n"
         "   shared_ptr:   %p\n“
         "\n\n",
         pid,
         &foo, &bar, &main,
         &global_int, &global_arr, &global_ptr,
         &local_arr, &local_k, &local_v,
         global_ptr, &shared->shared_int);
  …
}
```

layout.c

CSE 컴퓨터공학부 Department of Computer Science & Engineering

Where do Functions & Variables go?

- Older Linux kernels without address space layout randomization or address space layout randomization disabled

```
$ make layout
gcc -Wall -O2 -o layout layout.c –lpthread
$ ./layout
[ 4914] Memory addresses
  &foo():         0x4007c0
  &bar():         0x4007d0
  &main():        0x4005a0

  &global_int:    0x601060
  &global_arr:    0x6010c0
  &global_ptr:    0xa010c0

  &local_int:     0x7fffe4b12780
  &local_arr:     0x7fffe4b12920
  &local_ptr:     0x7fffe4b12924

   global_ptr:    0x7efedd940010
   shared_ptr:    0x7efeddf7e020

…
```

```
$ ./layout
[ 4919] Memory addresses
  &foo():         0x4007c0
  &bar():         0x4007d0
  &main():        0x4005a0

  &global_int:    0x601060
  &global_arr:    0x6010c0
  &global_ptr:    0xa010c0

  &local_int:     0x7ffcf8698640
  &local_arr:     0x7ffcf86987e0
  &local_ptr:     0x7ffcf86987e4

   global_ptr:    0x7f6c0ce11010
   shared_ptr:    0x7f6c0d44f020

…
```


CSE 컴퓨터공학부 Department of Computer Science & Engineering

Where do Functions & Variables go?

- Recent Linux kernels with address space layout randomization (ASLR)

different address (security)

```
$ make layout
gcc -Wall -O2 -o layout layout.c -lpthread
$ ./layout
[16072] Memory addresses
  &foo():        0x56272dead430
  &bar():        0x56272dead440
  &main():       0x56272dead100

  &global_int:   0x56272e2b00e0
  &global_arr:   0x56272deb00e0
  &global_ptr:   0x56272deb00c8

  &local_int:    0x7fff0c9334b0
  &local_arr:    0x7fff0c9334a8
  &local_ptr:    0x7fff0c9334ac

   global_ptr:   0x7f70fc85c010
   shared_int:   0x7f70fce9a020

…
```

```
$ ./layout
[16073] Memory addresses
  &foo():        0x5576a2907430
  &bar():        0x5576a2907440
  &main():       0x5576a2907100

  &global_int:   0x5576a2d0a0e0
  &global_arr:   0x5576a290a0e0
  &global_ptr:   0x5576a290a0c8

  &local_int:    0x7ffe2ad75630
  &local_arr:    0x7ffe2ad75628
  &local_ptr:    0x7ffe2ad7562c

   global_ptr:   0x7f96662bf010
   shared_int:   0x7f96668fd020

…
```


CSE 컴퓨터공학부 Department of Computer Science & Engineering

Where do Functions & Variables go?

- Multi-process version
  - creates *n* clones of itself
  - all clones repeatedly increment two variables `global_int` and `shared_int`

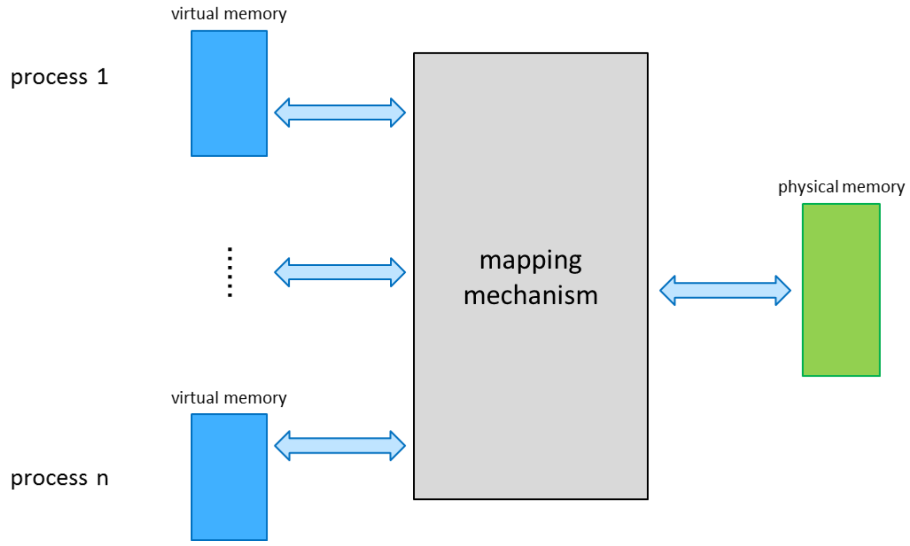
```
$ ./layout 2
[16074] Memory addresses
  &foo():        0x55d2942ea430
  &bar():        0x55d2942ea440
  &main():       0x55d2942ea100

  &global_int:   0x55d2946ed0e0
  &global_arr:   0x55d2942ed0e0
  &global_ptr:   0x55d2942ed0c8

  &local_int:    0x7ffd6a6ad640
  &local_arr:    0x7ffd6a6ad638
  &local_ptr:    0x7ffd6a6ad63c

   global_ptr:   0x7f8e5c5a1010
   shared_int:   0x7f8e5cbdf020

               mory addresses
                 0x55d2942ea430
                 0x55d2942ea440
                 0x55d2942ea100

              int:   0x55d2946ed0e0
              arr:   0x55d2942ed0e0
              ptr:   0x55d2942ed0c8

              nt:    0x7ffd6a6ad640
              r:     0x7ffd6a6ad638
              tr:    0x7ffd6a6ad63c

   global_ptr:   0x7f8e5c5a1010
   shared_int:   0x7f8e5cbdf020
…
```

Same address for global variable (sharing)

```
…
[16074] global_int = mem[0x55d2946ed0e0] = 0; shared_int = mem[0x7f8e5cbdf020] = 0
[16075] global_int = mem[0x55d2946ed0e0] = 0; shared_int = mem[0x7f8e5cbdf020] = 1
[16074] global_int = mem[0x55d2946ed0e0] = 1; shared_int = mem[0x7f8e5cbdf020] = 2
[16074] global_int = mem[0x55d2946ed0e0] = 2; shared_int = mem[0x7f8e5cbdf020] = 3
[16075] global_int = mem[0x55d2946ed0e0] = 1; shared_int = mem[0x7f8e5cbdf020] = 4
[16074] global_int = mem[0x55d2946ed0e0] = 3; shared_int = mem[0x7f8e5cbdf020] = 5
[16075] global_int = mem[0x55d2946ed0e0] = 2; shared_int = mem[0x7f8e5cbdf020] = 6
[16074] global_int = mem[0x55d2946ed0e0] = 4; shared_int = mem[0x7f8e5cbdf020] = 7
…
```

How is that possible?

CSE 컴퓨터공학부 Department of Computer Science & Engineering

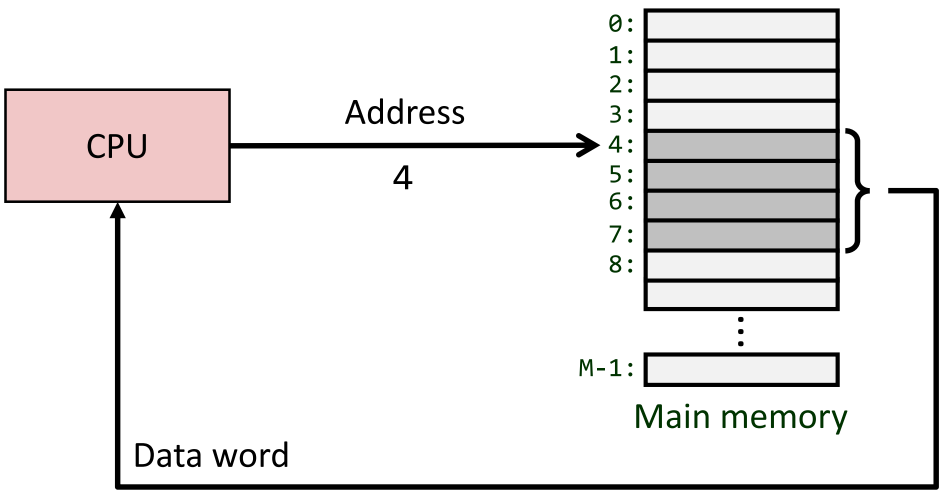

virtual memory
process 1
mapping
mechanism
physical memory
virtual memory
process n

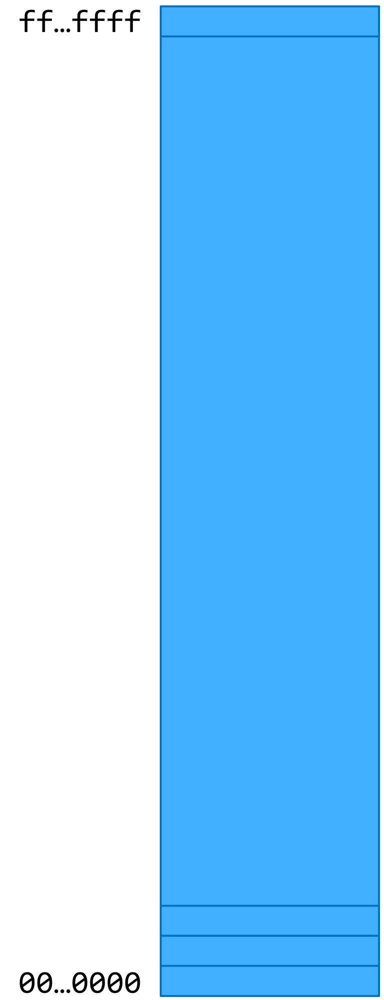
Motivation

Memory Up Until Now

- Programs access data through memory addresses

```
ld x10, 16(x2)
sd x11, 24(x2)
```





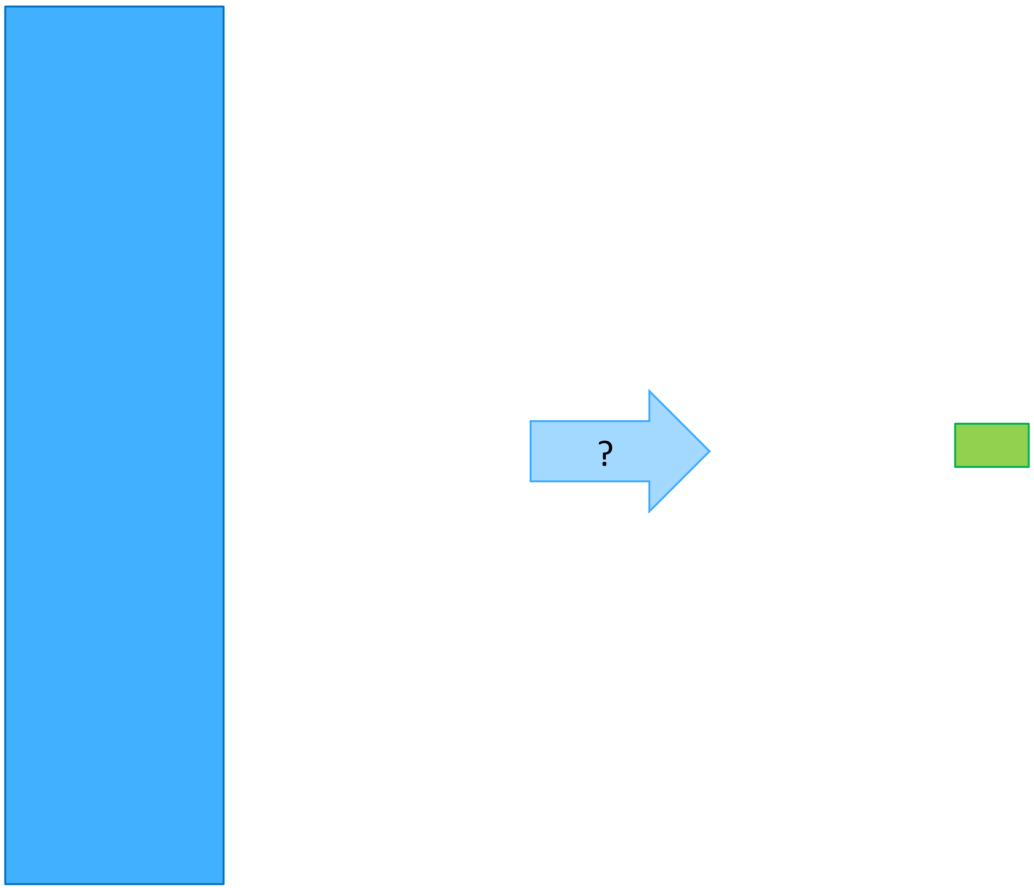
  - conceptually a very large array of bytes

CSE 컴퓨터공학부 Department of Computer Science & Engineering

Problem 1: How does everything fit?

virtual memory:
64-bit addresses,
16 exabytes

physical memory:
a few gigabytes



…and there are many processes!

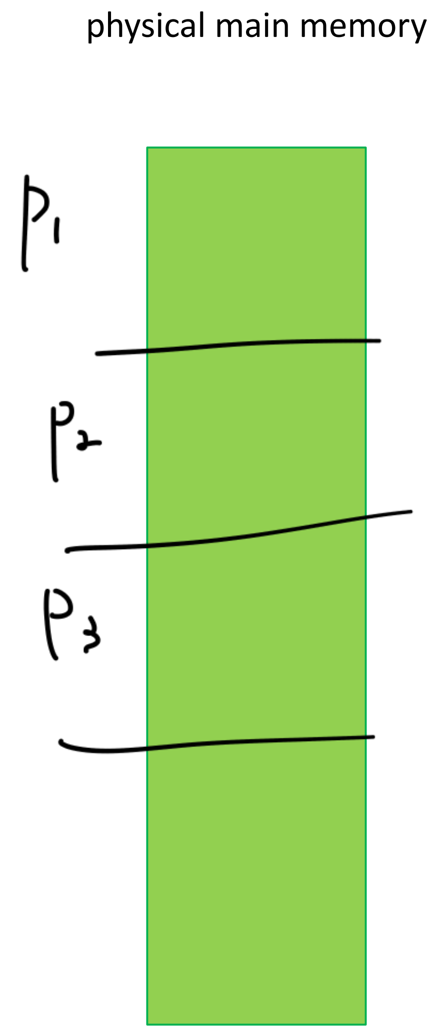
Problem 2: Memory Management (where)

process 1
process 2
process 3
…
process n

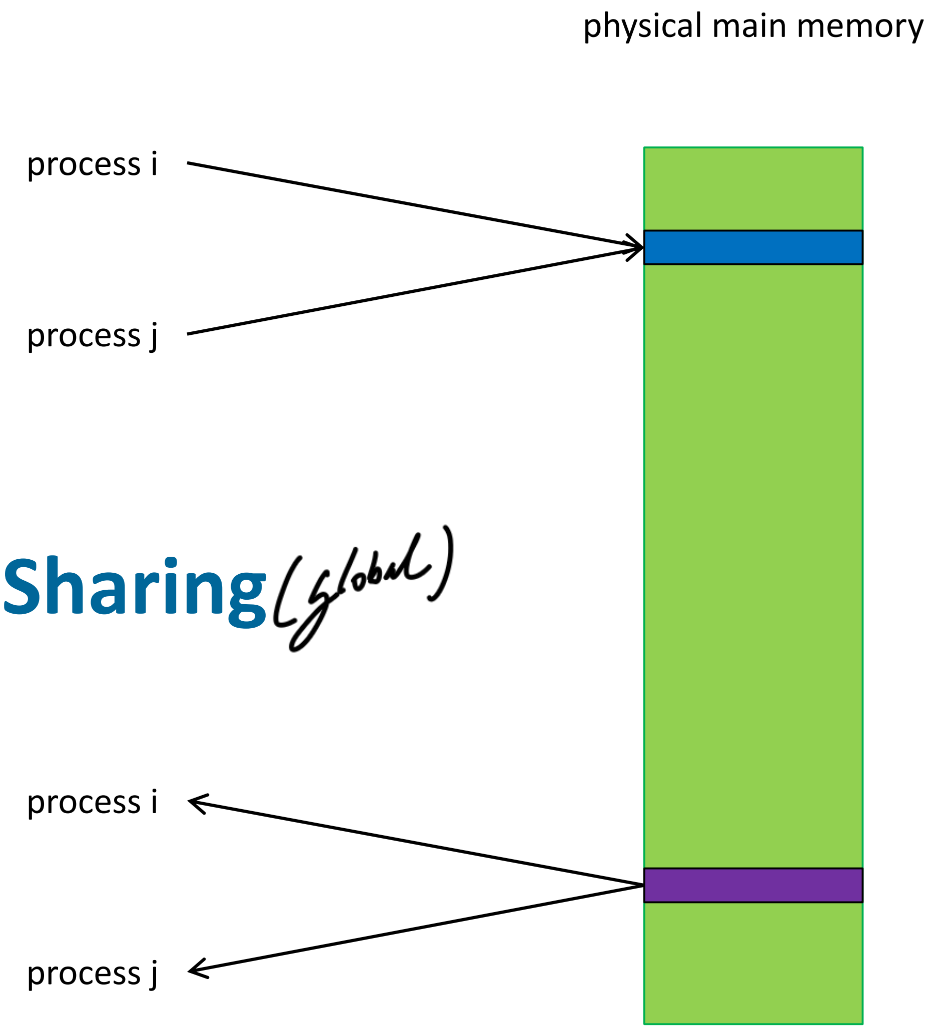
x

`.text`
`.data`
heap
stack

what goes where?



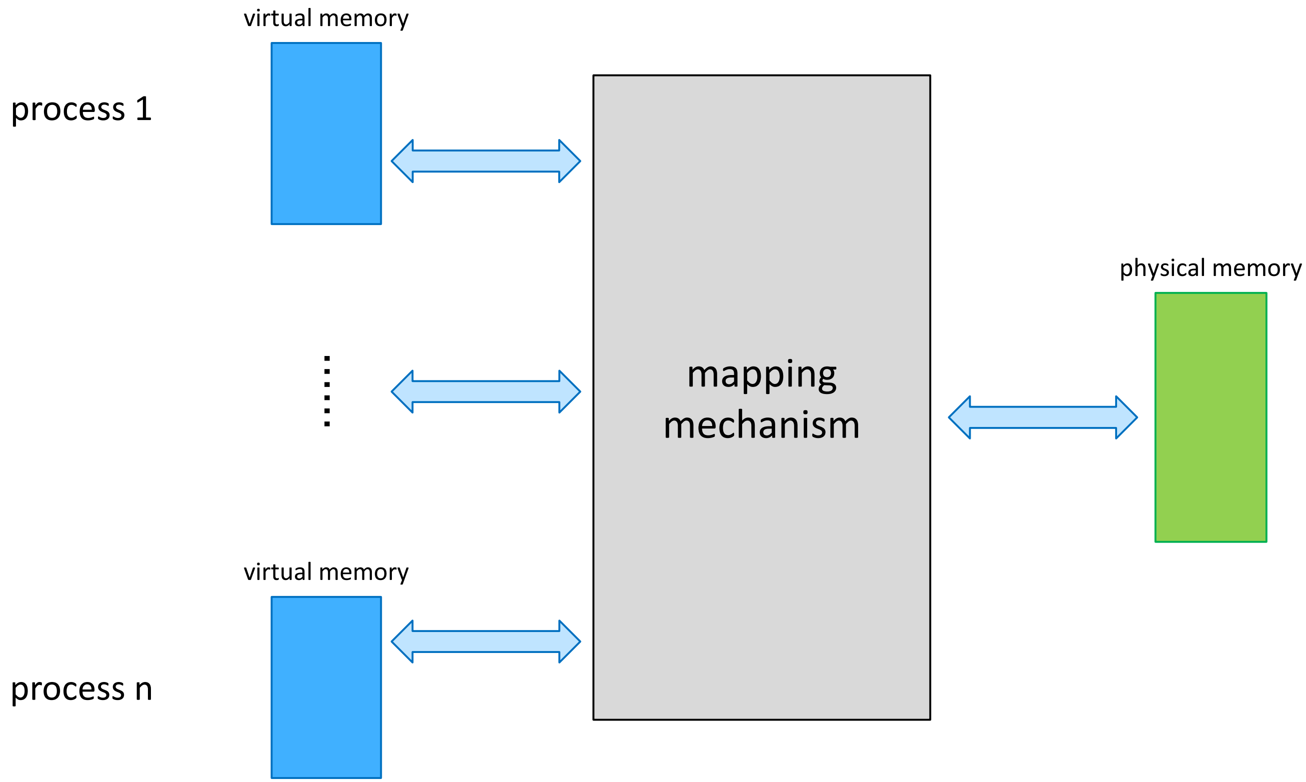
Problem 3: Protection



Problem 4: Sharing (global)

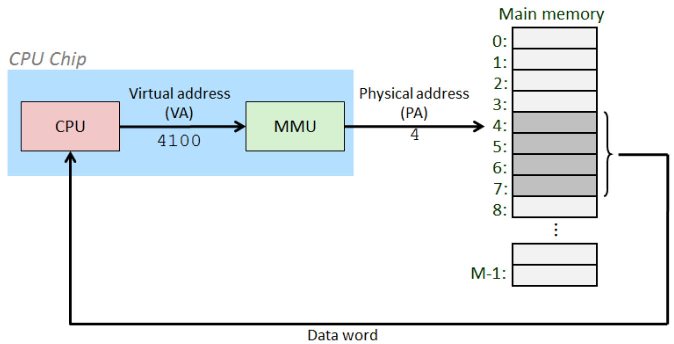
CSE 컴퓨터공학부 Department of Computer Science & Engineering

Solution: Indirection (MMU)



- Each process gets its own private memory space
- **Solves all our previous problems!**

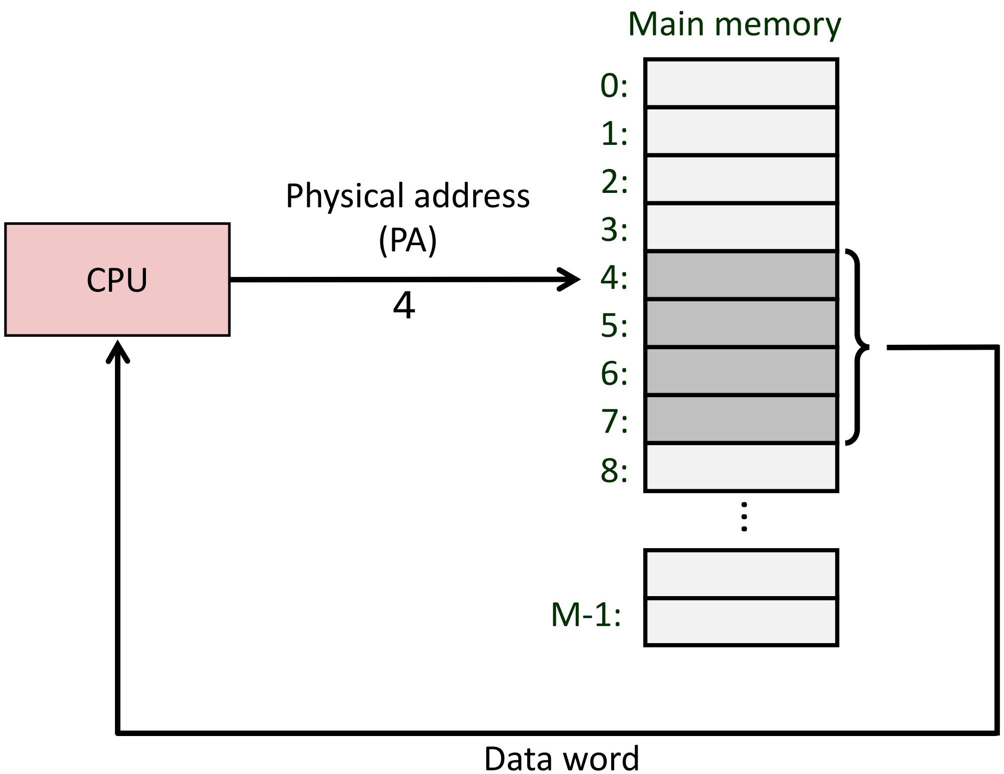
CSE 컴퓨터공학부 Department of Computer Science & Engineering



Main memory
0:
1:
2:
3:
4:
5:
6:
7:
8:
M-1:
CPU Chip
CPU
Virtual address
(VA)
4100
MMU
Physical address
(PA)
4
Data word

The Basics

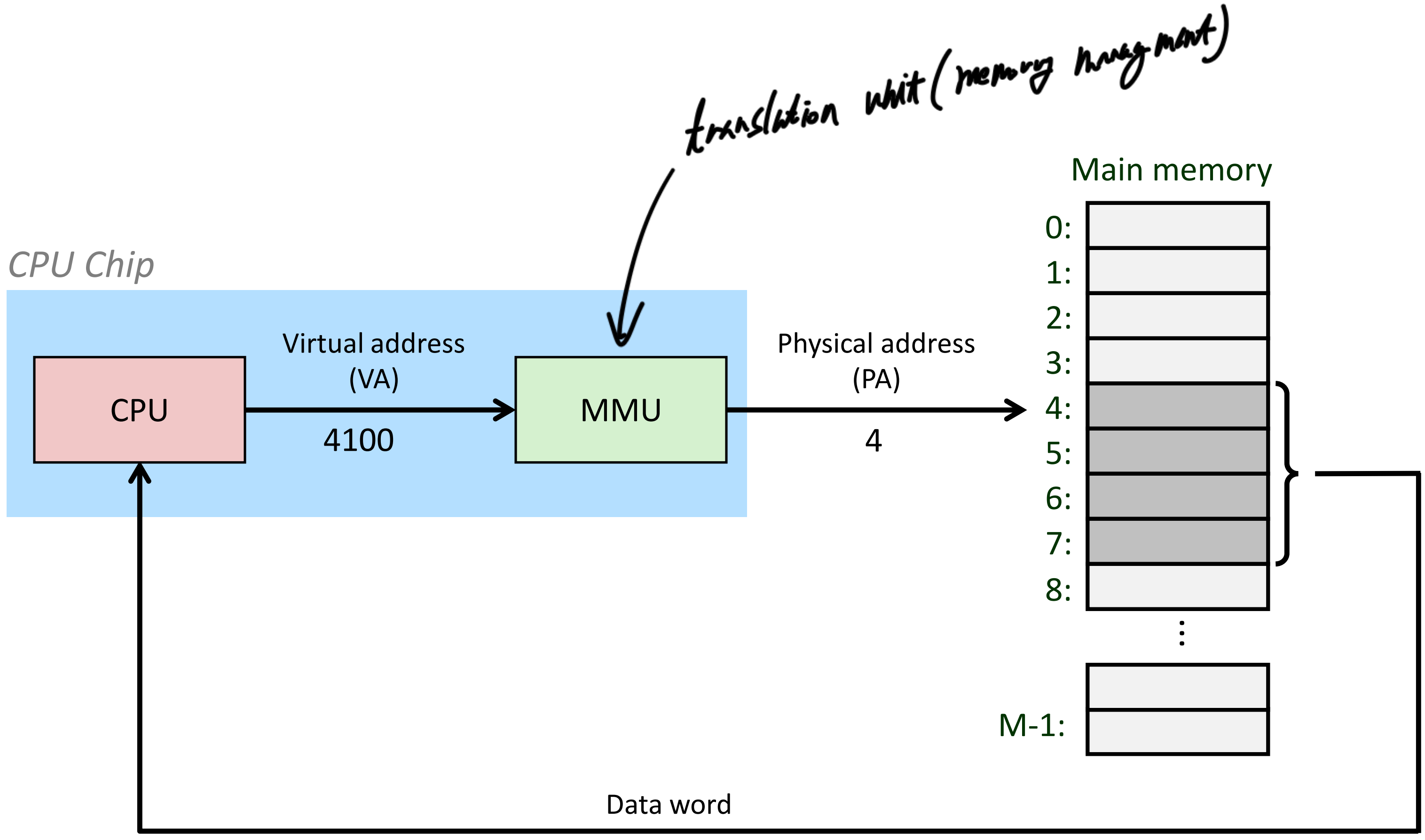
A System Using Physical Addressing



- Still used in simple systems like embedded microcontrollers:
  IoT devices, embedded controllers in elevators, digital picture frames, …

CSE 컴퓨터공학부
Department of Computer Science & Engineering

A System Using Virtual Addressing



- Used in all modern servers, desktops, and laptops
- One of the great ideas in computer science

The Basics

- **Virtual address (VA) = abstraction of physical address (PA)**

- **Address spaces**
  - Linear address space: Ordered set of contiguous non-negative integer addresses:

    $\{0,1,2,3,\ldots\}$

  - Virtual address space: Set of $N=2^n$ virtual addresses

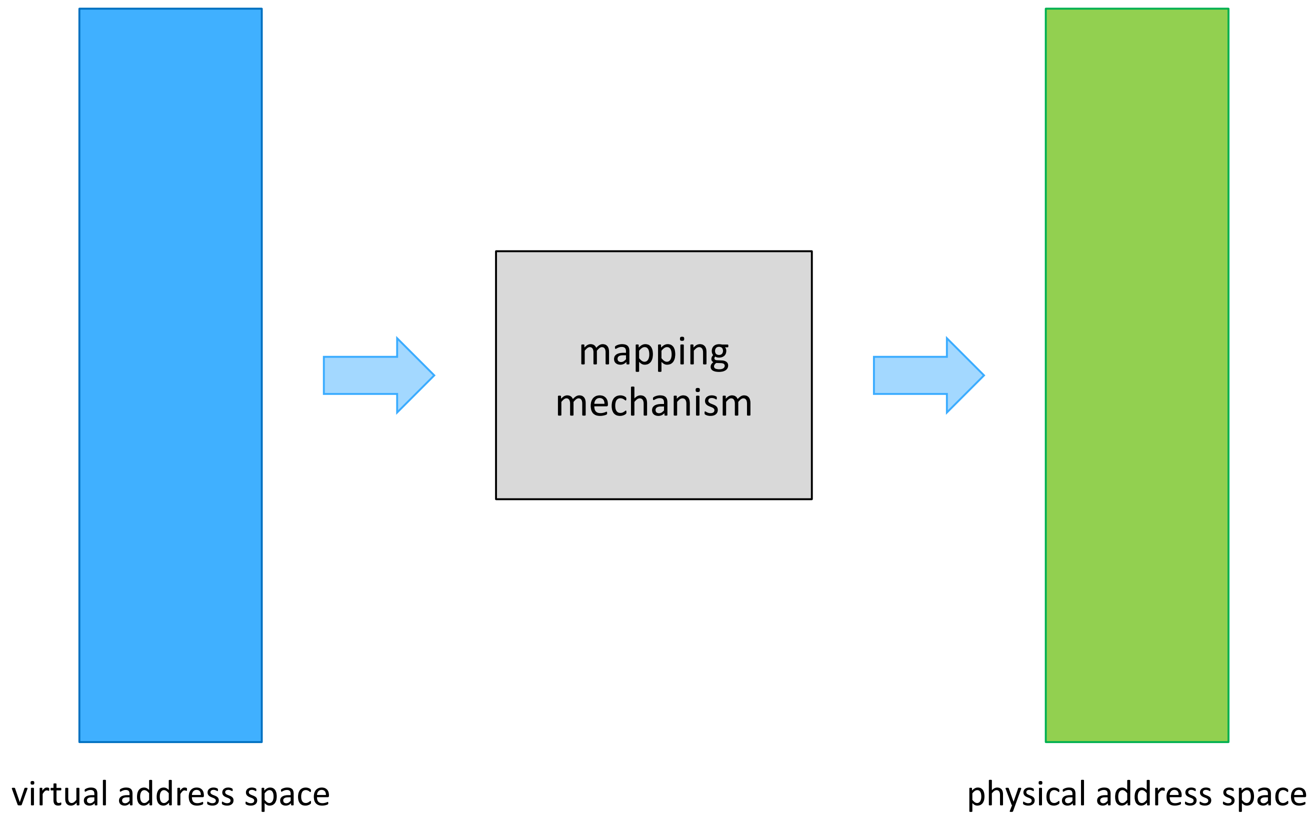
    $\{0,1,2,3,\ldots,N-1\}$

  - Physical address space: Set of $M=2^m$ physical addresses

    $\{0,1,2,3,\ldots,M-1\}$

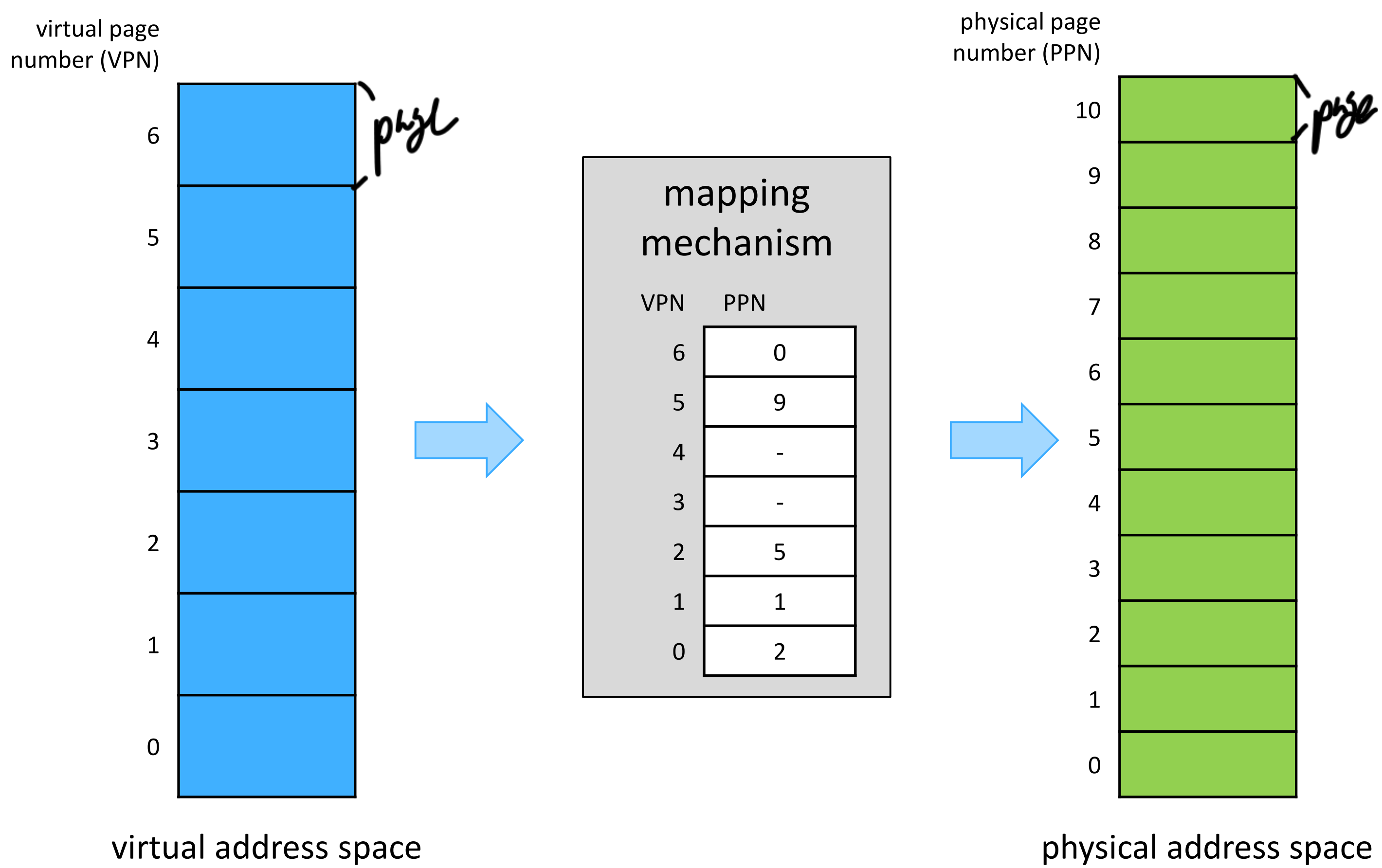
The Basics

- Implementation: indirection between VA and PA



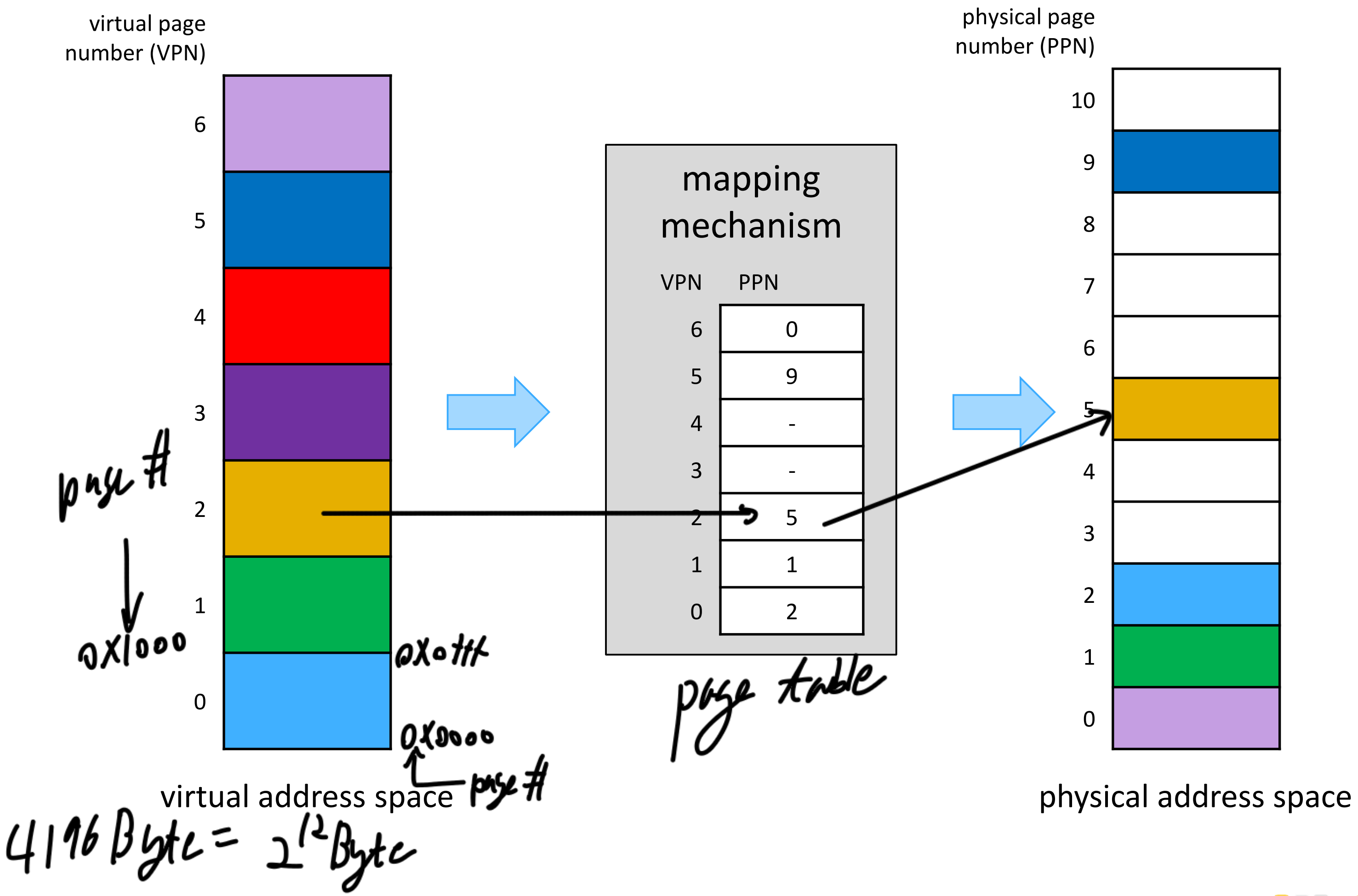
The Basics

- Implementation: indirection between VA and PA



Translation of VA to PA

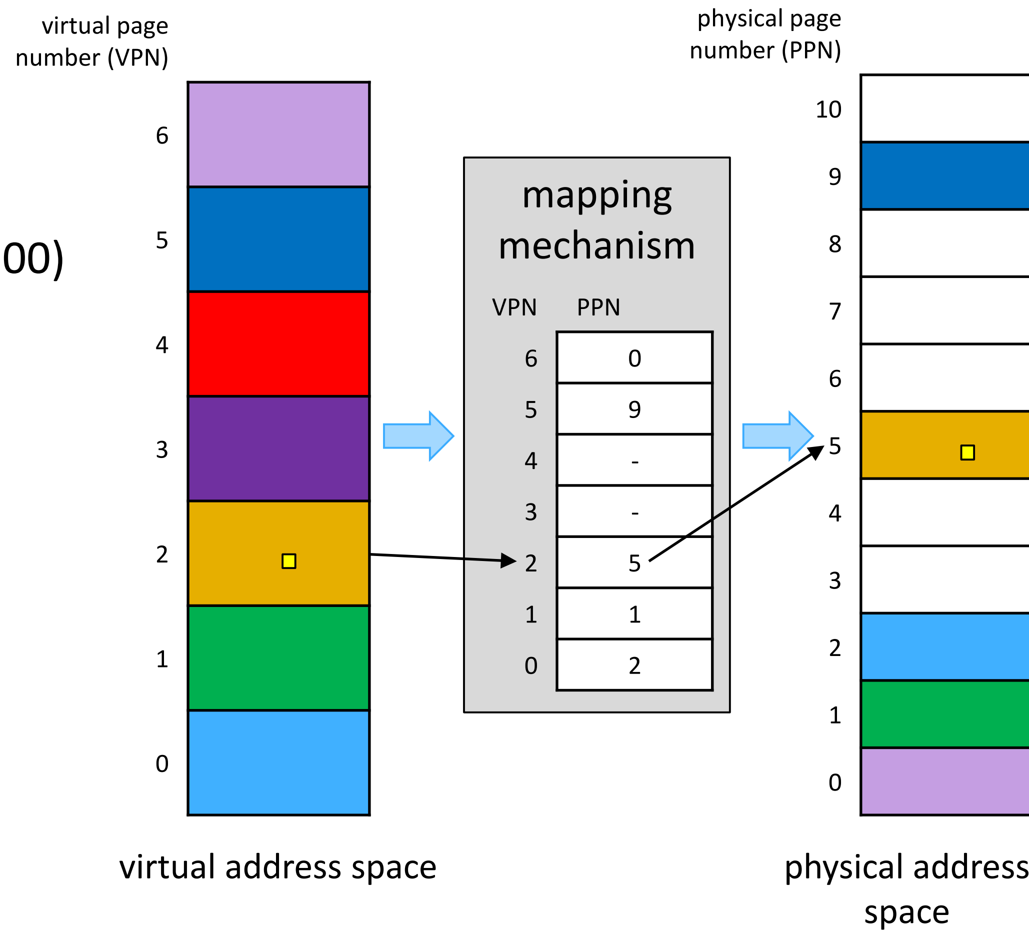
- Implementation: indirection between VA and PA



CSE 컴퓨터공학부 Department of Computer Science & Engineering

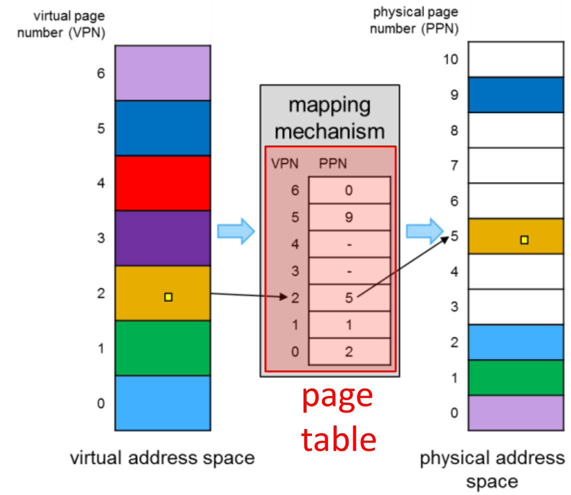
Translation of VA to PA

- Translation of VA 0x233 to PA
  - page size = 0x100
  - VA 0x233 =
    virtual page: 2 (= 0x233/0x100)
    virtual offset: 0x33 (= 0x233%0x100)
  - VPN 2 maps to PPN 5
    physical page 5
  - PA = 0x533
    physical page address: 0x500
    physical offset: 0x33



CSE 컴퓨터공학부 Department of Computer Science & Engineering

Translation of VA to PA

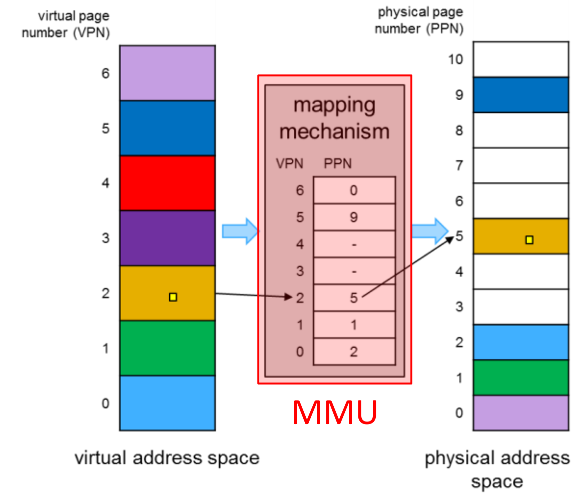


- Given
  - page size (PS)
    (always virtual page size == physical page size!)
  - page table (PT)
    translation table VPN→PPN
    one entry in the table is called a page table entry (PTE)

- Translation
  - **PA = PT[VA / PS] + VA % PS**

CSE 컴퓨터공학부 Department of Computer Science & Engineering

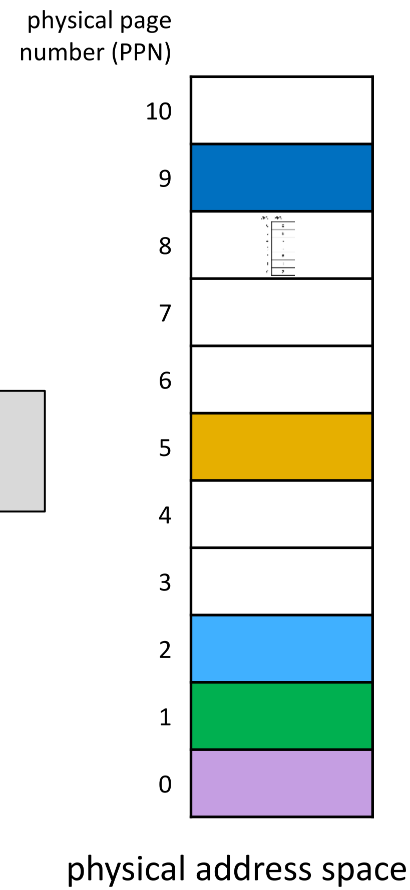
Memory Management Unit



- Memory Management Unit (MMU)
  - hardware unit performing VA to PA translation
  - translation is time critical, has to be fast

- The PT is stored in main memory, the MMU only holds a pointer to it
  - page table base register (PTBR)
    - on Intel architectures, the PTBR is named register CR3

MMU
PTBR = 0x800



CSE 컴퓨터공학부 Department of Computer Science & Engineering

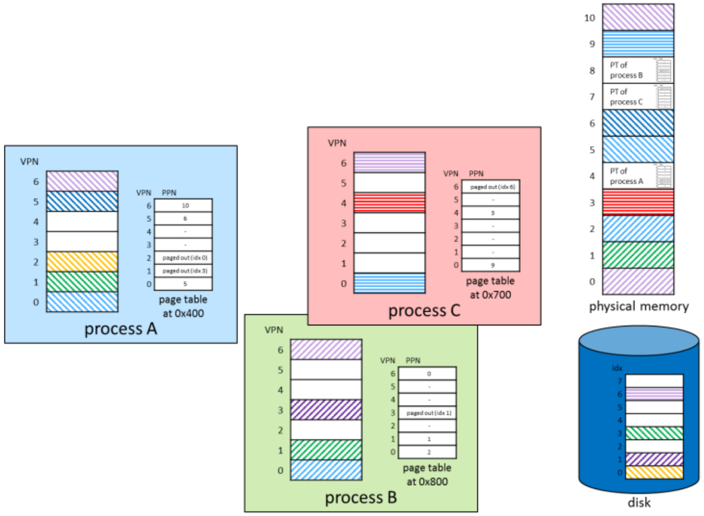
Advantages of Virtual Memory (VM)

- **Efficient use of limited main memory**
  - uses DRAM as a cache for the parts of a virtual address space
    - some non-cached parts stored on disk
    - some (unallocated) and non-cached parts stored nowhere
  - keep only active areas of virtual address space in memory
    - transfer data back and forth as needed

- **Simplified memory management for programmers**
  - Each process gets the same full, uniform linear address space

- **Isolated address spaces**
  - One process can't interfere with another's memory
    - because they operate in different address spaces
  - User program cannot access privileged kernel information
    - different sections of address spaces have different permissions

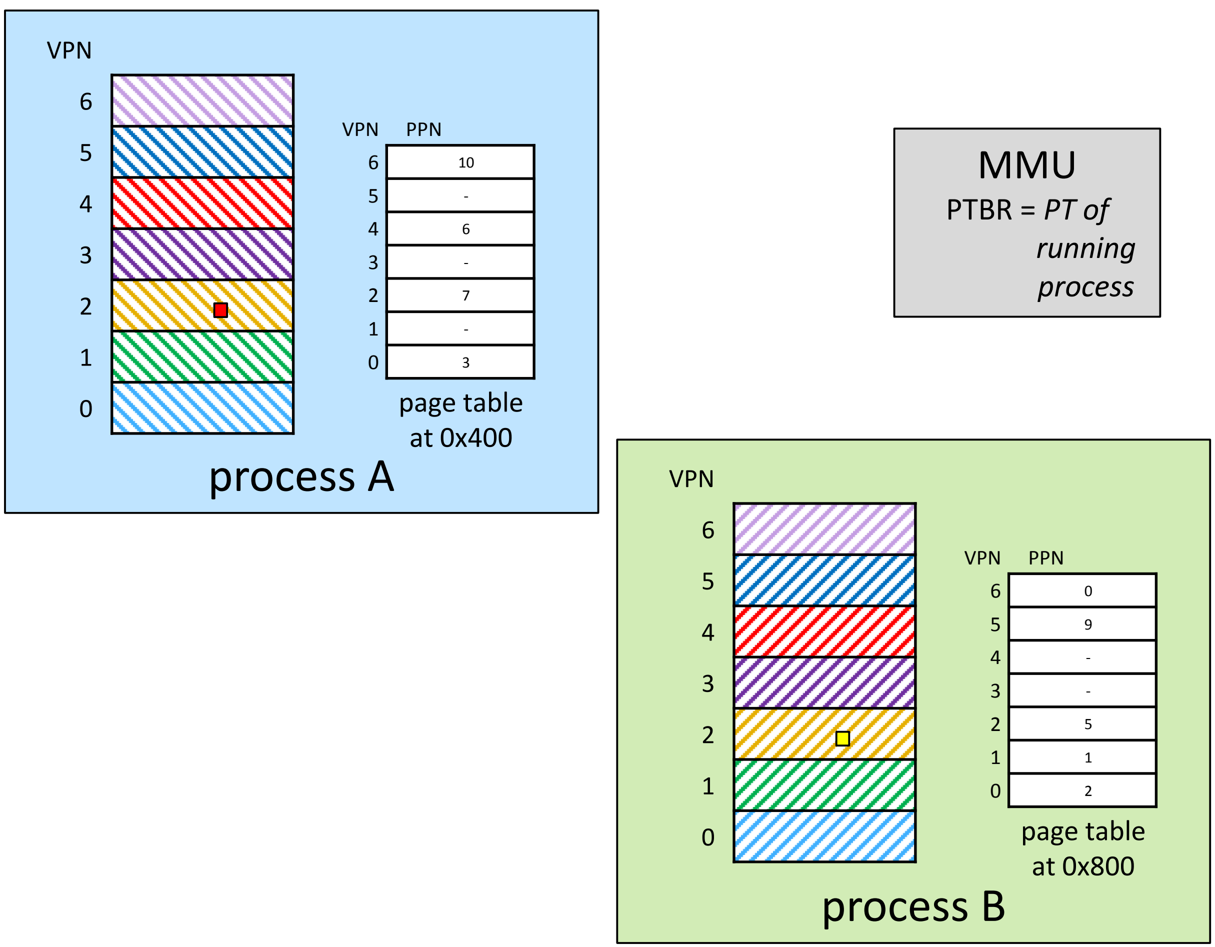
CSE 컴퓨터공학부 Department of Computer Science & Engineering

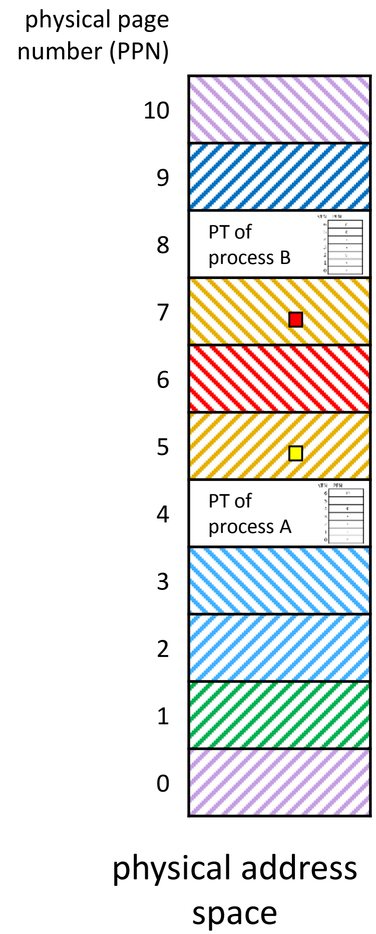


Features of Virtual Memory

Address Space Isolation

- Each process has its own address space
  → each process has its own page table

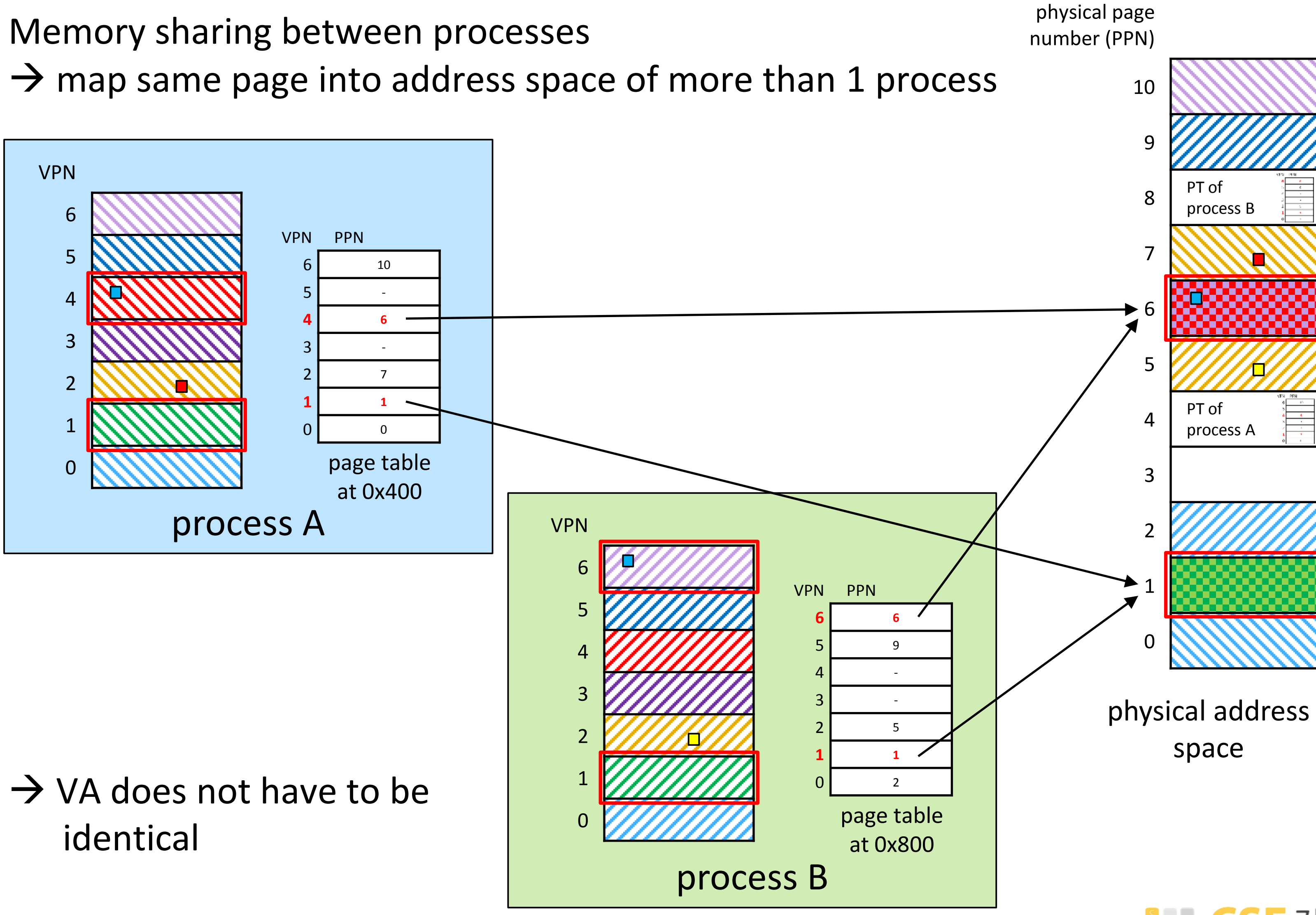




CSE 컴퓨터공학부 Department of Computer Science & Engineering

Shared Memory

- Memory sharing between processes
  → map same page into address space of more than 1 process

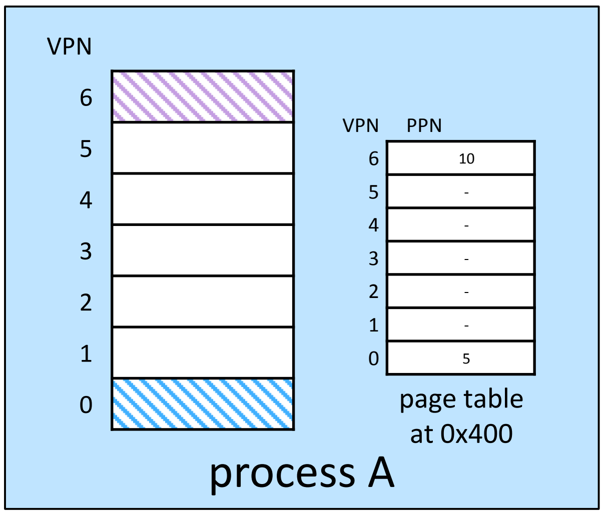


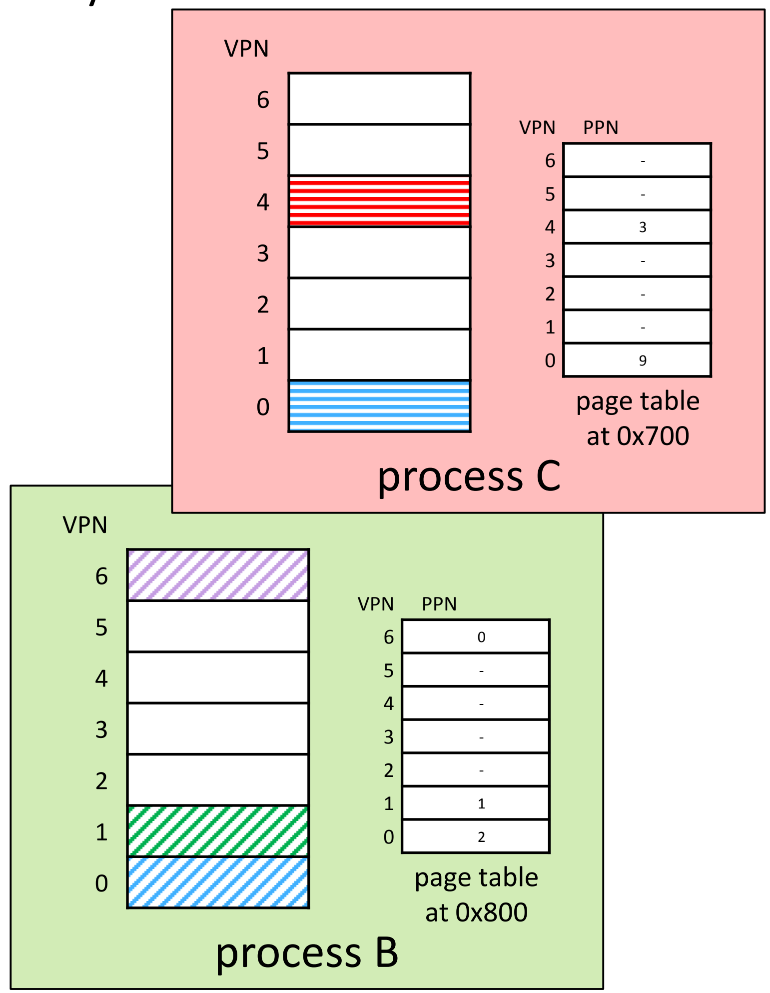
→ VA does not have to be identical

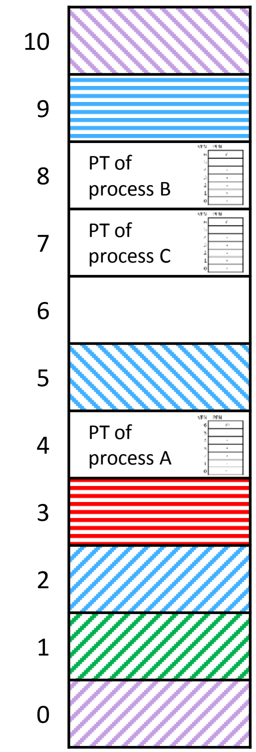
CSE 컴퓨터공학부 Department of Computer Science & Engineering

Decoupling of Virtual from Physical Address Space

- Run several processes with a large virtual address space on a much smaller physical memory







physical memory
11*0x100 = 0xB00
(11 pages)

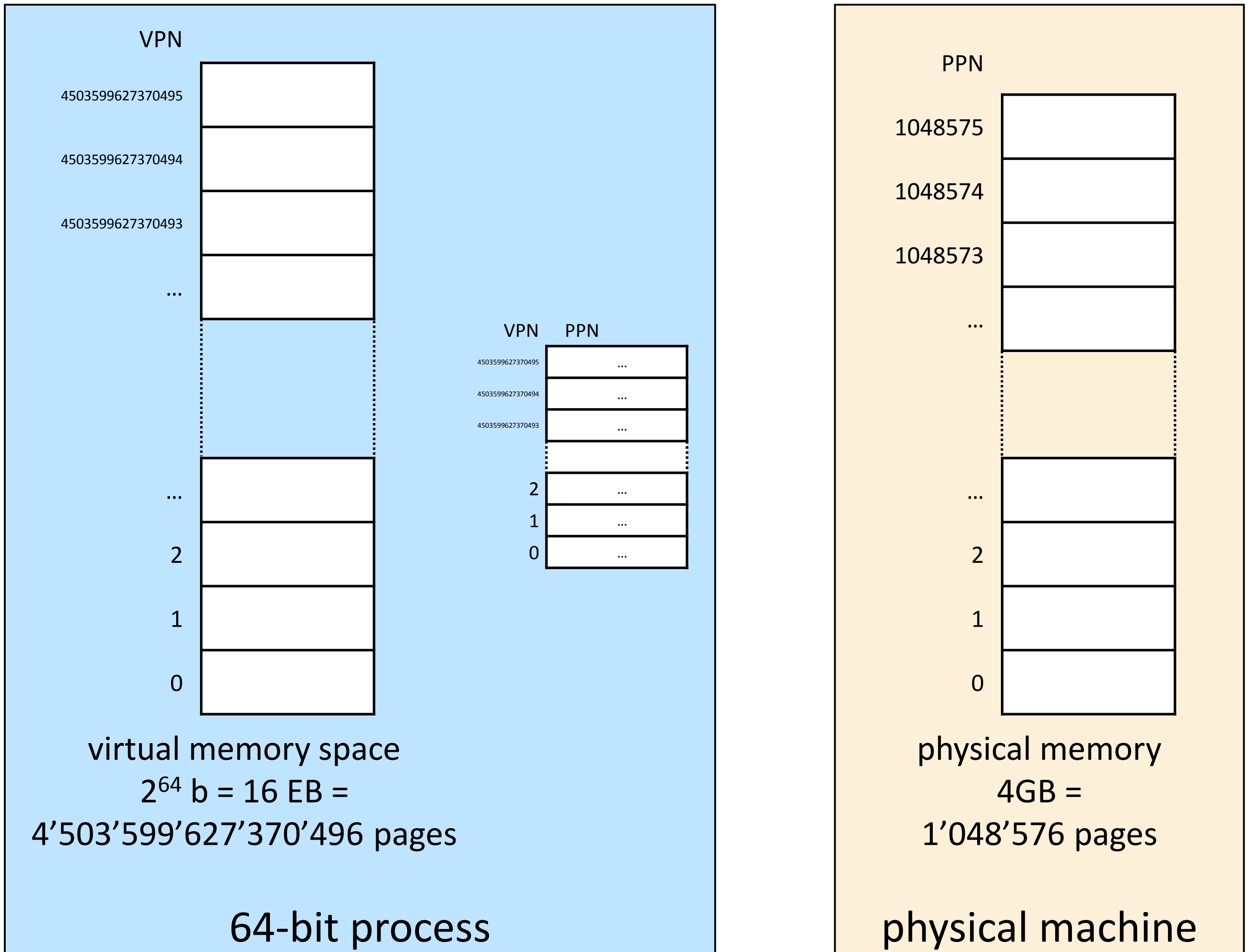
each process "sees"
7*0x100 = 0x700 bytes
(7 pages) of virtual memory

3 processes "see" 0x1500
bytes (21 pages) of
virtual memory

CSE 컴퓨터공학부 Department of Computer Science & Engineering

Decoupling of Virtual from Physical Address Space

- 64-bit process on much smaller physical memory (PS=4KB)



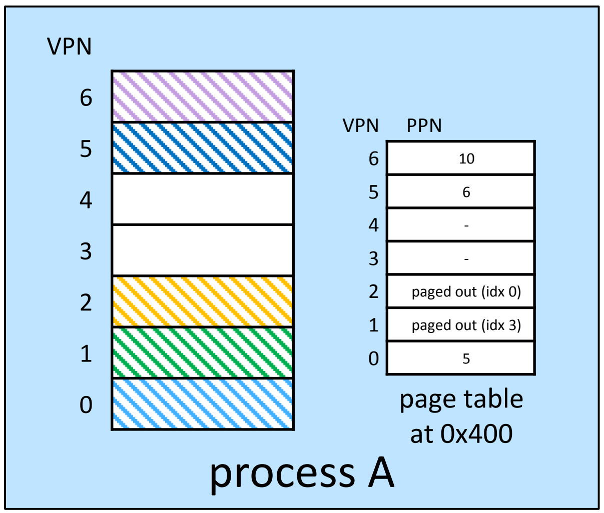
Lots of engineering problems:

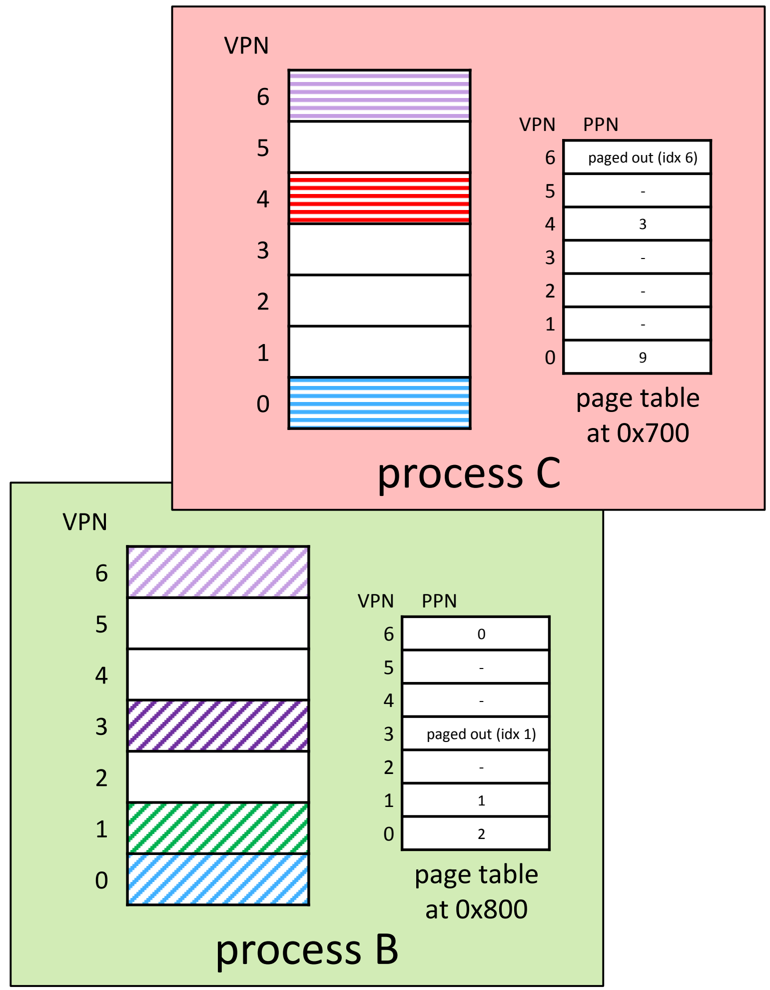
- how big is the page table?
- multi-level page tables
- security
- isolation
- sharing
- avoid duplication
- lazy copy

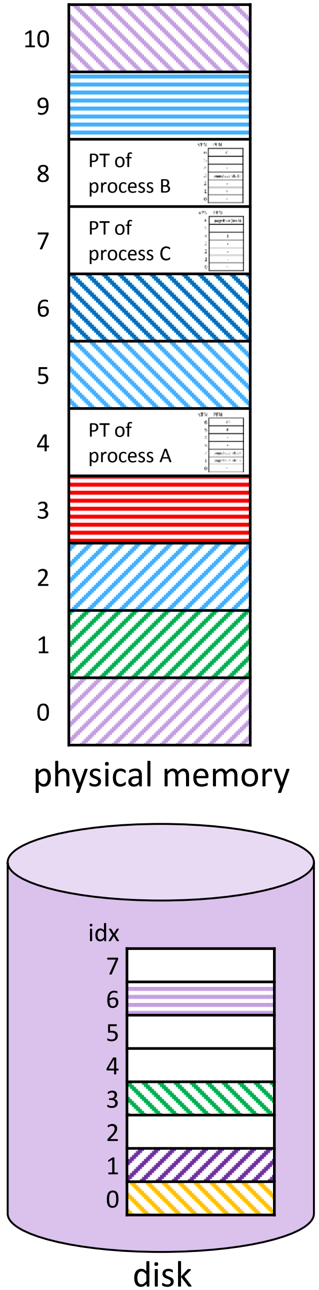
CSE 컴퓨터공학부 Department of Computer Science & Engineering

Physical Memory as a Cache

- Use the disk for swapping ("on-demand paging")





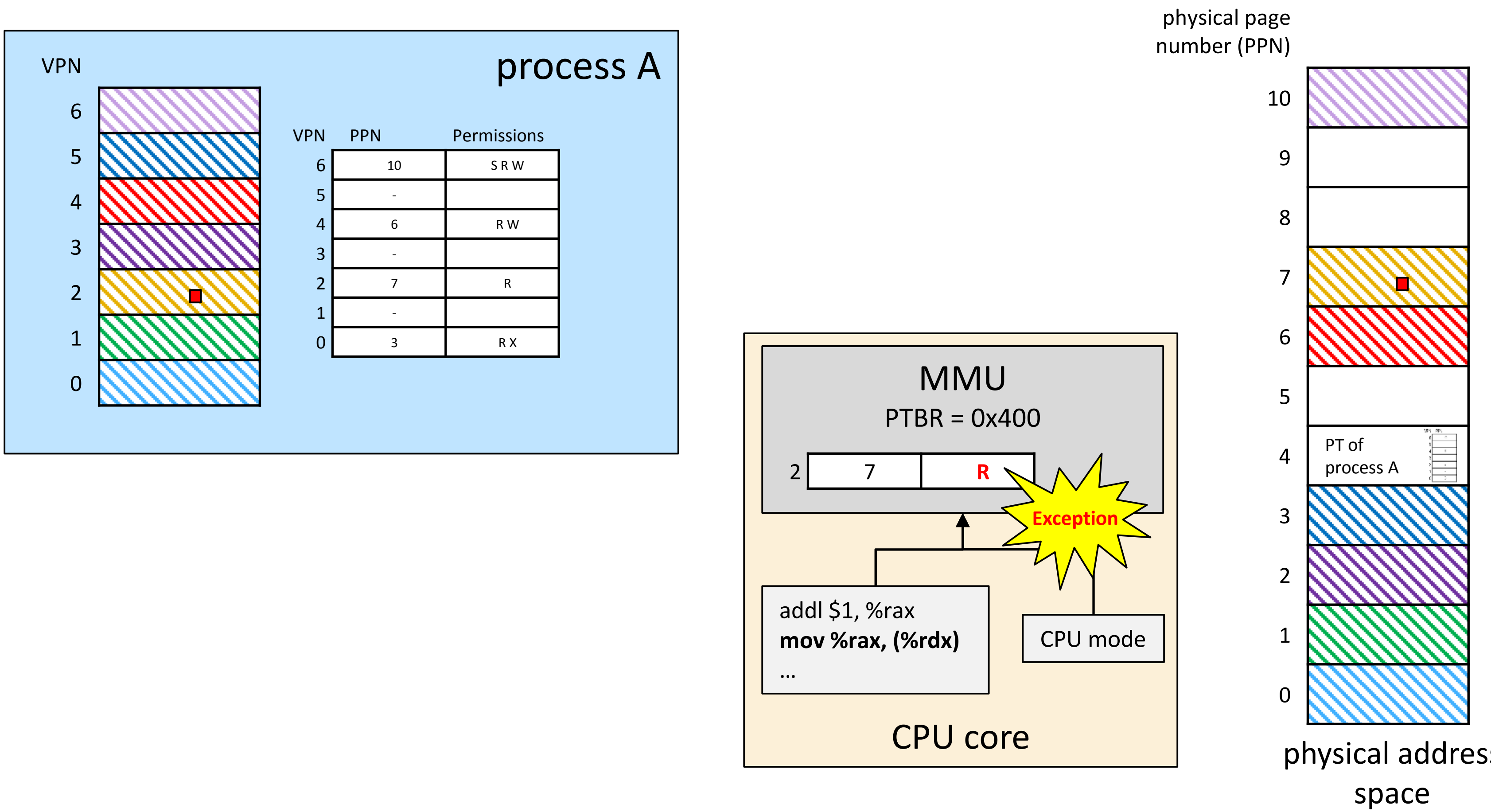


Lots of engineering problems:

- what happens when accessing address 0x233 in process A?
- which page to replace when physical memory is full?
- …

Memory Access Permissions

- Define access permissions for each page to prevent from bugs / attacks



```
$ ./layout
[7375] Memory addresses
  &foo():         0x5621d0284400
  &bar():         0x5621d0284410
  &main():        0x5621d02840f0

  &global_int:    0x5621d0287080
  &global_arr:    0x5621d02870c0
  &global_ptr:    0x5621d06870c0

  &local_int:     0x7fff202b87c0
  &local_arr:     0x7fff202b87b8
  &local_ptr:     0x7fff202b87bc

   global_ptr:    0x7fe24e283010
```

Back to Our Example Program

CSE 컴퓨터공학부 Department of Computer Science & Engineering

Motivating Example

- Where do functions & variables go?

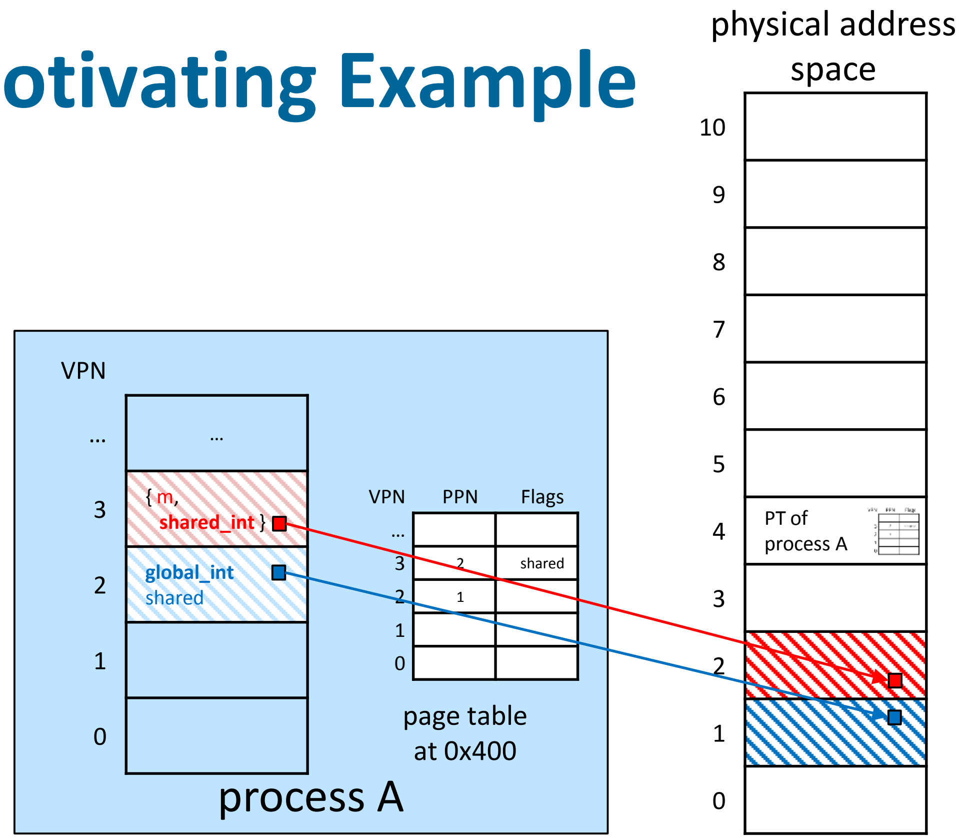
```c
…
#define N 1024*1024

int global_int = 0;
…
struct __shared {
  sem_t m;
  int shared_int;
} *shared;
…
int main(int argc, char *argv[])
{
  …
  //
  // create & initialize shared memory area
  //
  shared = (struct __shared*)mmap(NULL, sizeof(struct __shared),
           PROT_READ|PROT_WRITE, MAP_ANONYMOUS|MAP_SHARED, 0, 0);
  if (shared == MAP_FAILED) {
    perror("Cannot map shared memory");
    return EXIT_FAILURE;
  }
  sem_init(&shared->m, 1, 1);
  shared->shared_int = 0;
  …
```

```c
  …
  //
  // create child processes
  // each process has a different 'delay' value
  //
  while (nproc > 1) {
    if (fork() == 0) break;
    nproc--;
  }
  delay = nproc;
  …
  //
  // endless loop increasing global_int / shared_int
  //
  while (1) {
    …
    // increase variables
    global_int++;
    sem_wait(&shared->m);
    shared->shared_int++;
    sem_post(&shared->m);

    sleep(delay);
  }

  return EXIT_SUCCESS;
}
```

layout.c

CSE 컴퓨터공학부 Department of Computer Science & Engineering

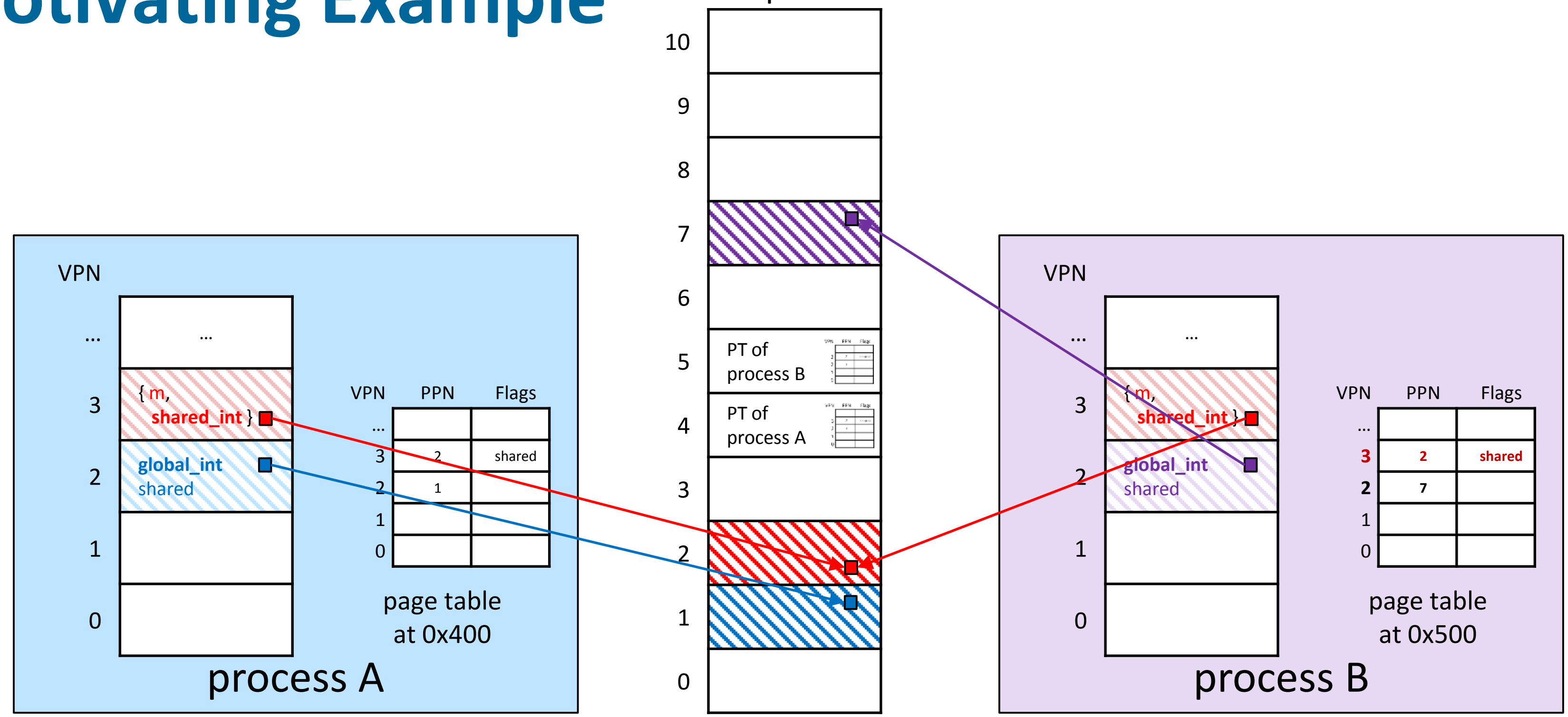
Motivating Example




```
…
int global_int = 0;
struct __shared {
  sem_t m;
  int shared_int;
} *shared;
…
int main(int argc, char *argv[])
{ …
  // create & initialize shared memory area
  shared = (struct __shared*)mmap(NULL, sizeof(struct __shared),
           PROT_READ|PROT_WRITE, MAP_ANONYMOUS|MAP_SHARED, 0, 0);
…
```


CSE 컴퓨터공학부 Department of Computer Science & Engineering

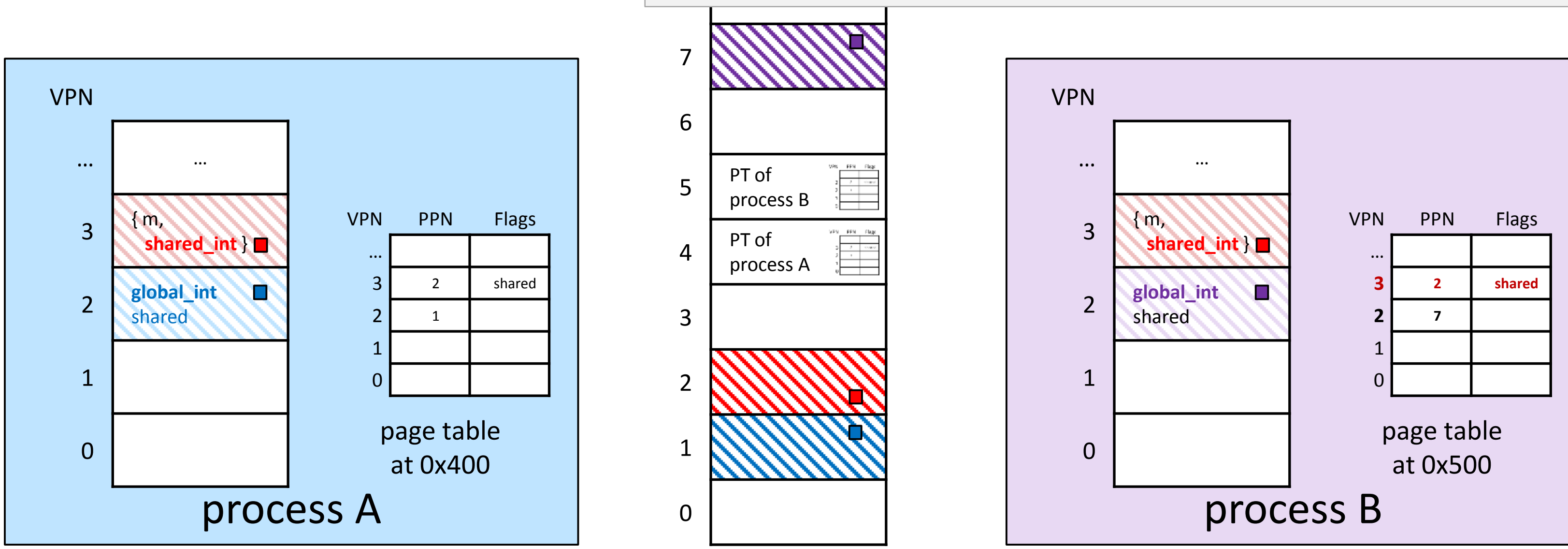
Motivating Example




```
…
while (nproc > 1) {
    if (fork() == 0) break;
    nproc--;
 }
…
```

Motivating Example

```
…
[16074] global_int = mem[0x55d2946ed0e0] = 0; shared_int = mem[0x7f8e5cbdf020] = 0
[16075] global_int = mem[0x55d2946ed0e0] = 0; shared_int = mem[0x7f8e5cbdf020] = 1
[16074] global_int = mem[0x55d2946ed0e0] = 1; shared_int = mem[0x7f8e5cbdf020] = 2
[16074] global_int = mem[0x55d2946ed0e0] = 2; shared_int = mem[0x7f8e5cbdf020] = 3
[16075] global_int = mem[0x55d2946ed0e0] = 1; shared_int = mem[0x7f8e5cbdf020] = 4
[16074] global_int = mem[0x55d2946ed0e0] = 3; shared_int = mem[0x7f8e5cbdf020] = 5
[16075] global_int = mem[0x55d2946ed0e0] = 2; shared_int = mem[0x7f8e5cbdf020] = 6
[16074] global_int = mem[0x55d2946ed0e0] = 4; shared_int = mem[0x7f8e5cbdf020] = 7
…
```




```
…
while (1) {
   …
   // increase variables;
   global_int++;

   sem_wait(&shared->m);
   shared->shared_int++;
   sem_post(&shared->m);

   sleep(delay);
 }
```

global_int	shared_int	global_int
1	1	
	2	1
2	3	
3	4	
	5	2
4	6	
	7	3

```
…
while (1) {
   …
   // increase variables;
   global_int++;

   sem_wait(&shared->m);
   shared->shared_int++;
   sem_post(&shared->m);

   sleep(delay);
 }
```

On-Demand Paging

On-Demand Paging

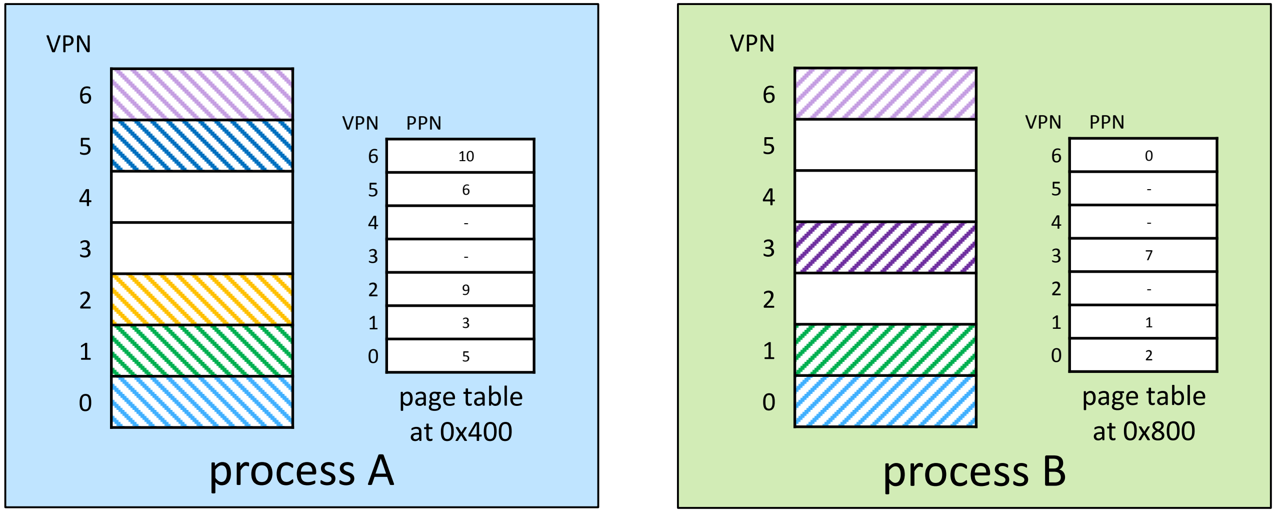
- **Key idea: use the disk as “cold storage” if the physical memory gets full**
  - Store (“page out”) memory pages to the disk
  - Load them back when needed

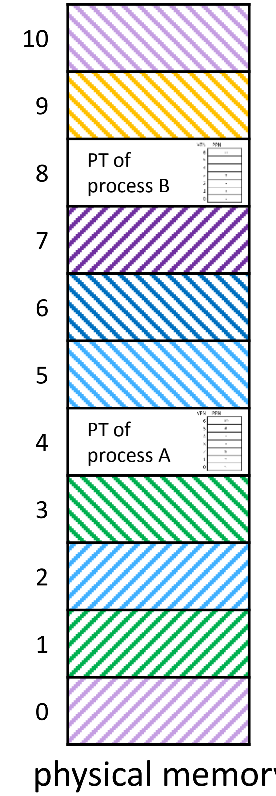
- Reverse view: physical memory is a cache for the disk
  - Not quite true in the same sense as for a cache – there are data pages that are created and initially only exist in DRAM and not on disk (stack, heap)

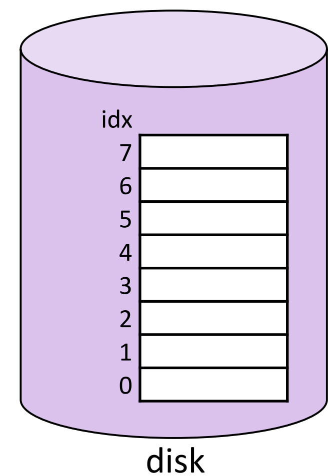
- Only possible thanks to virtual memory
  - Before virtual memory, processes were **swapped** out to make room for other processes
  - Process swapping (swap in/out) loads/saves the *entire memory* of a process from/to the disk

CSE 컴퓨터공학부 Department of Computer Science & Engineering

On-Demand Paging



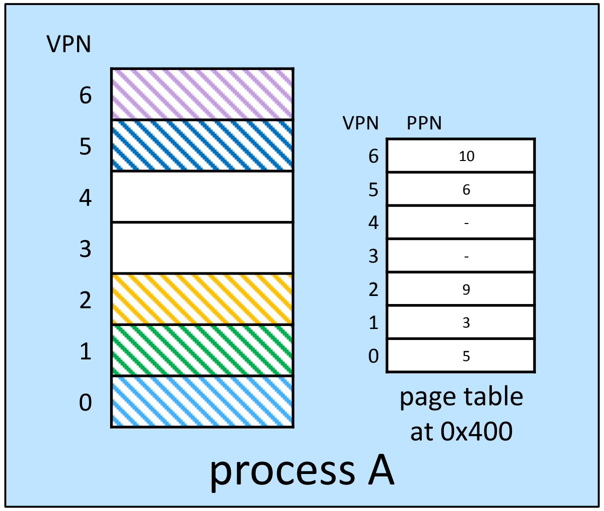


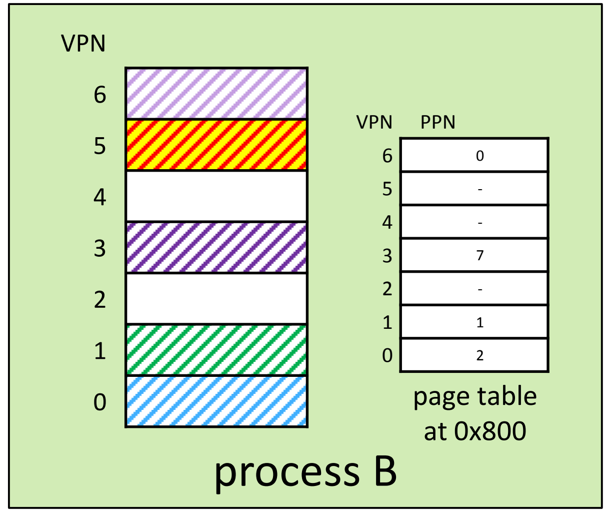


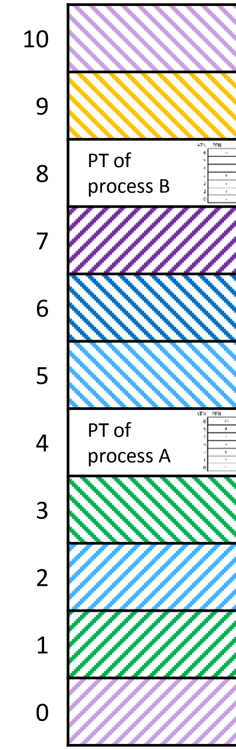
- Physical memory completely full
- Process B needs more memory on the stack

CSE 컴퓨터공학부
Department of Computer Science & Engineering

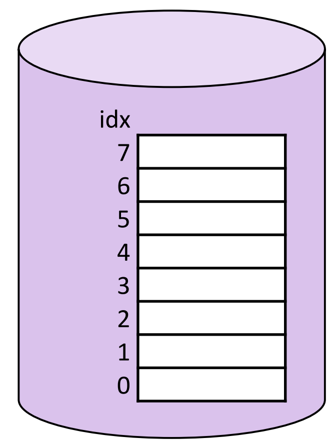
On-Demand Paging







physical memory

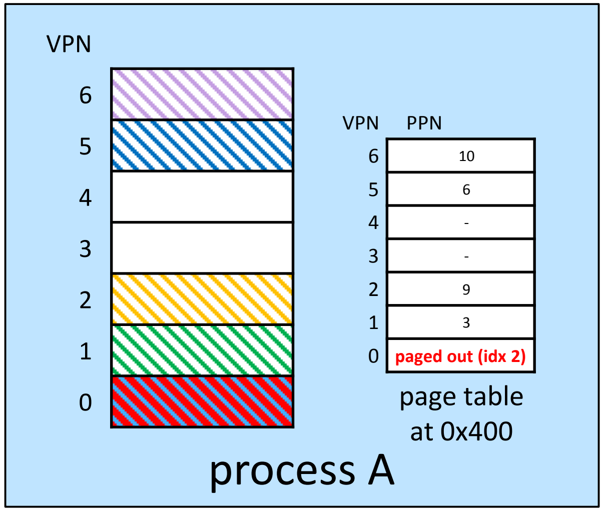


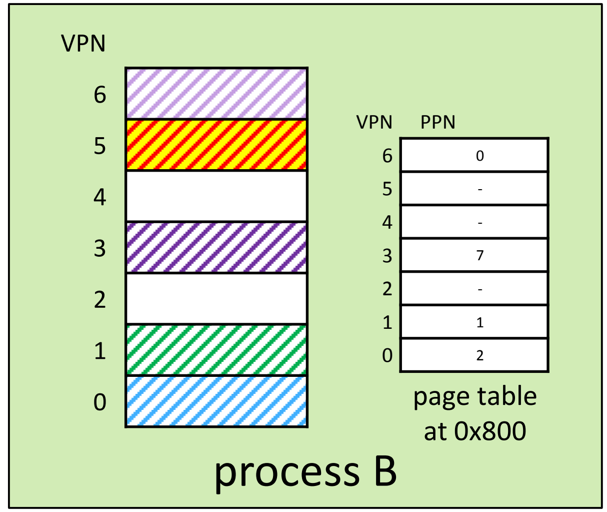
disk

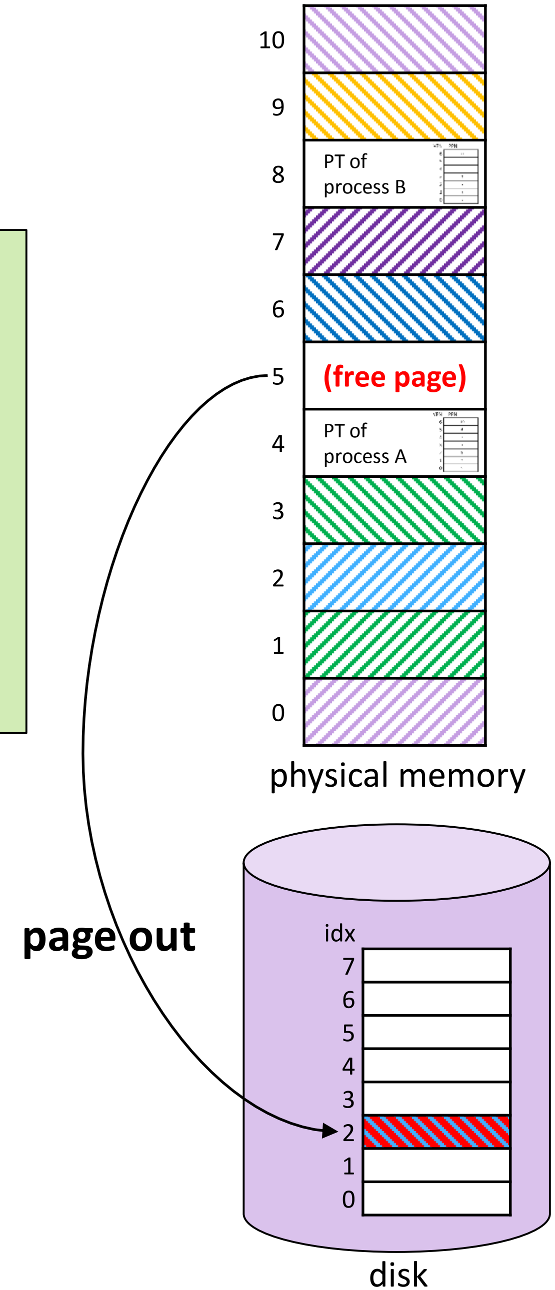
- Physical memory completely full
- Process B needs more memory on the stack
  - additional memory always requested from the kernel
  - kernels realizes that there are no free pages left
  - selects one page to be paged out to disk

CSE 컴퓨터공학부 Department of Computer Science & Engineering

On-Demand Paging



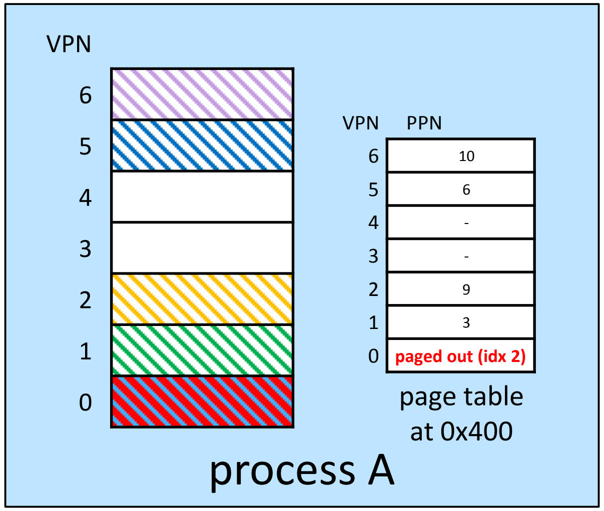


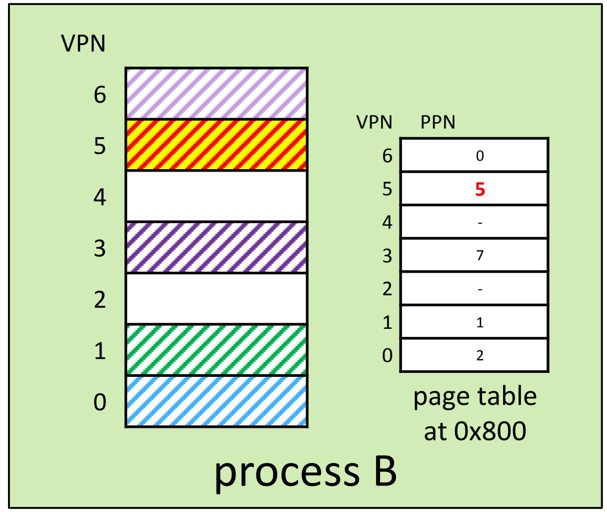


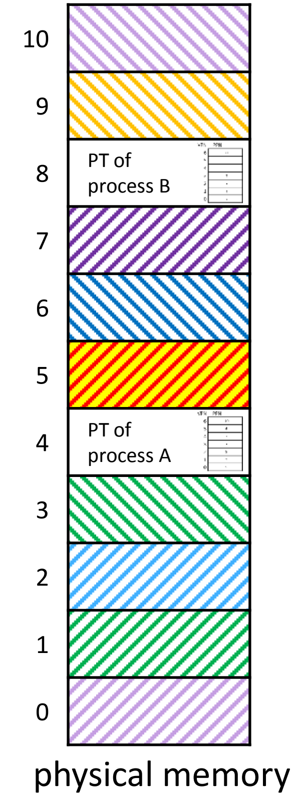
- Physical memory completely full
- Process B needs more memory on the stack
  - additional memory always requested from the kernel
  - kernels realizes that there are no free pages left
  - selects one page to be paged out to disk

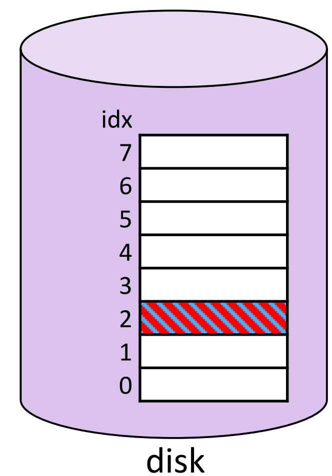
CSE 컴퓨터공학부 Department of Computer Science & Engineering

On-Demand Paging





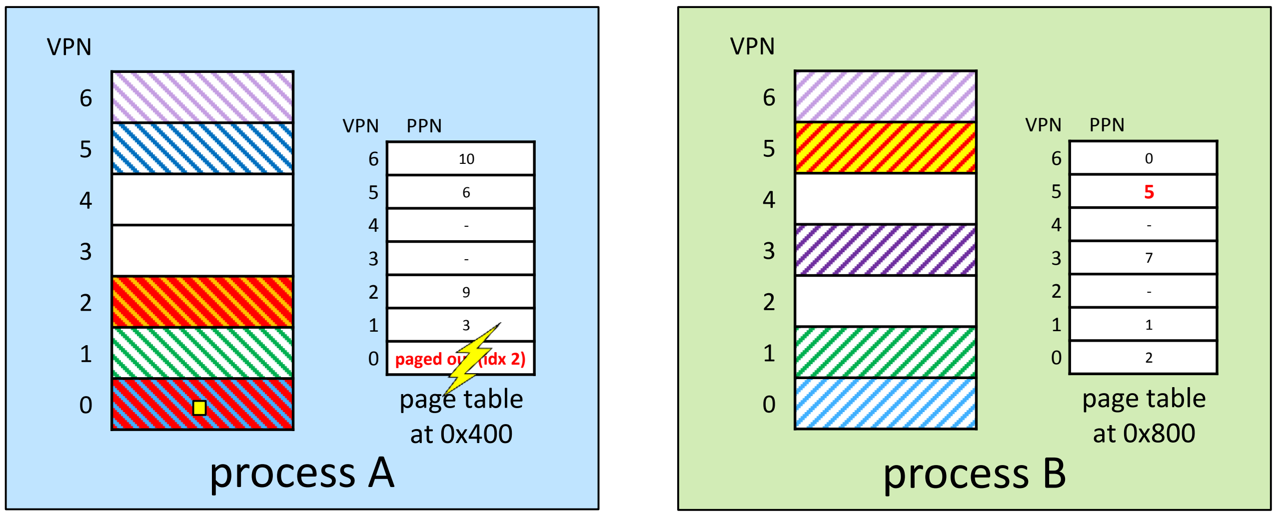


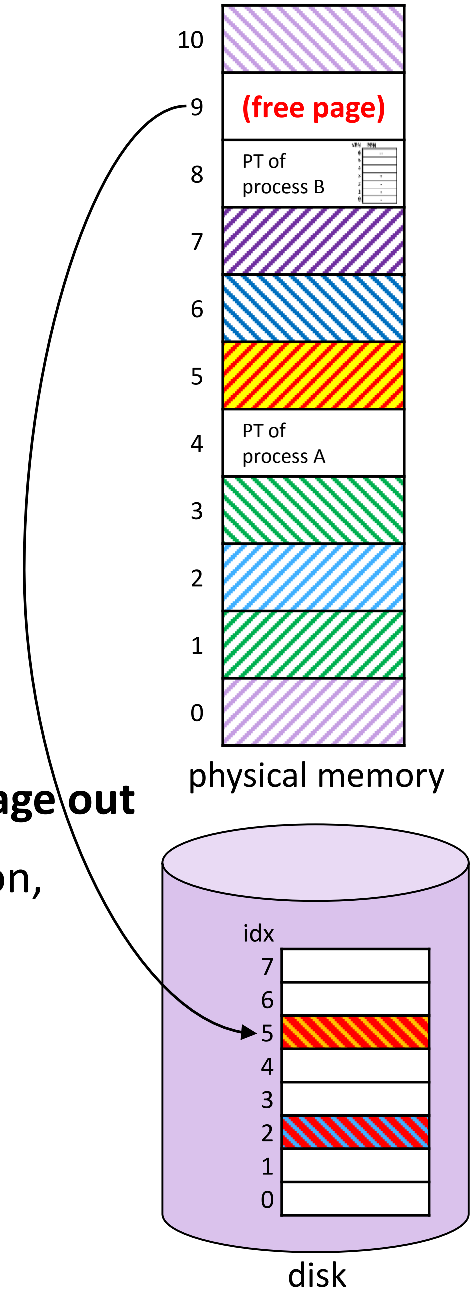


- Physical memory completely full
- Process B needs more memory on the stack
  - additional memory always requested from the kernel
  - kernels realizes that there are no free pages left
  - selects one page to be paged out to disk (aka victim page)
  - new page of process B is allocated in place of the paged out page

CSE 컴퓨터공학부 Department of Computer Science & Engineering

On-Demand Paging

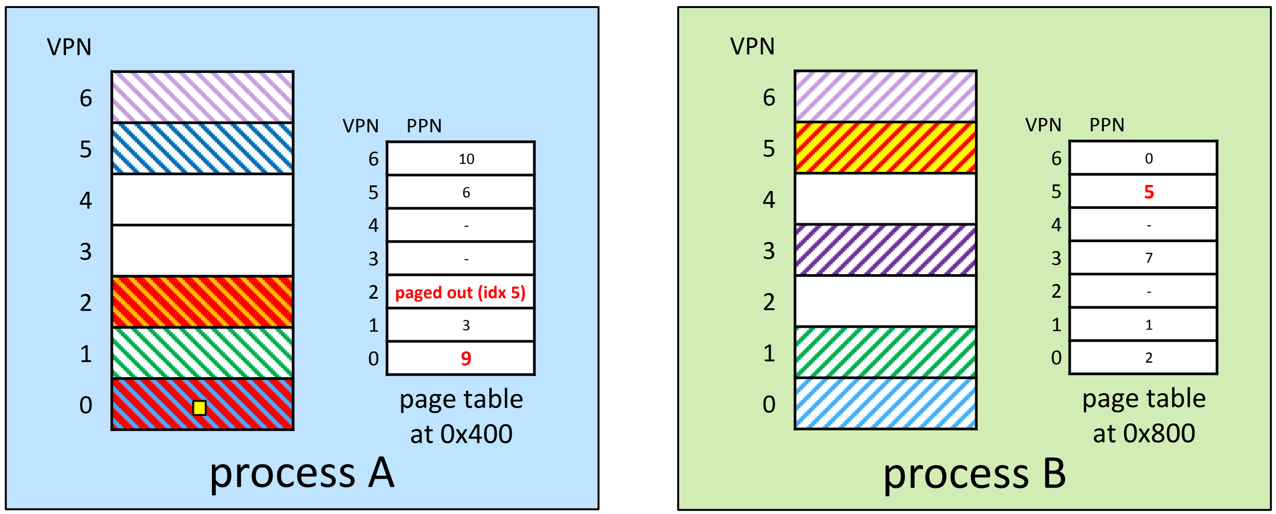


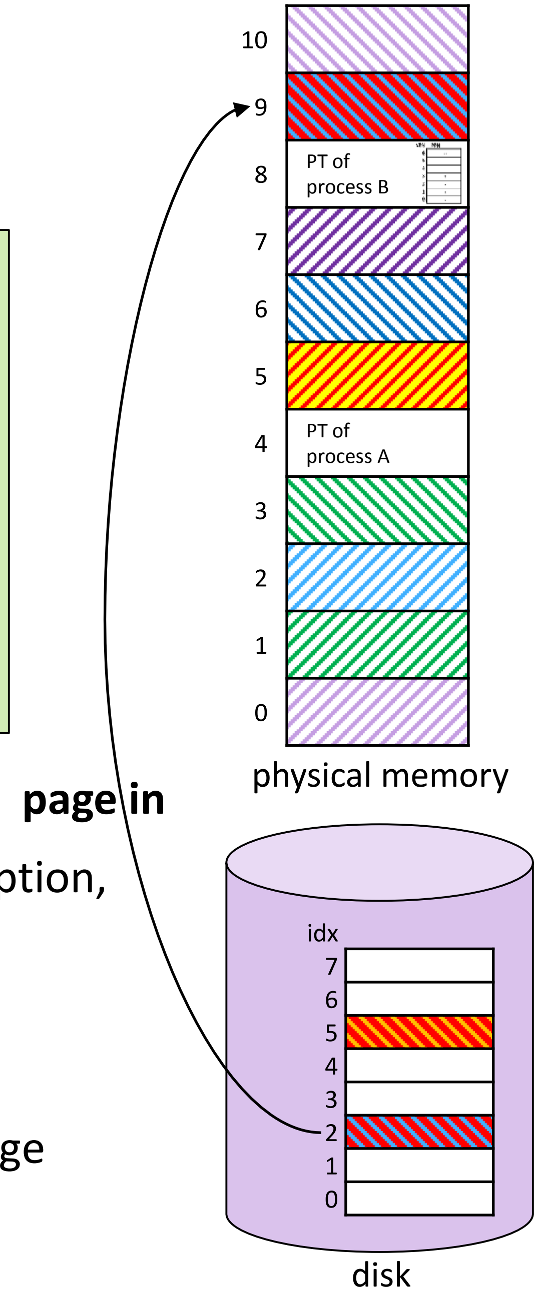


- What happens when process A accesses virtual page 0?
  - MMU: VA → PA translation fails, generates page fault exception, kernel intercepts and calls page fault handler
  - Page fault handler inspects faulted address, must bring page into memory (page in)
  - If no free page available: first select and page out victim page

CSE 컴퓨터공학부
Department of Computer Science & Engineering

On-Demand Paging



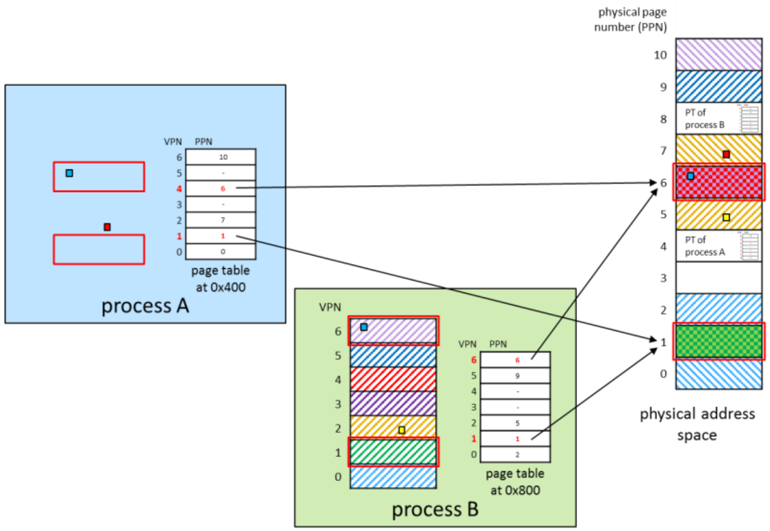


- What happens when process A accesses virtual page 0?
  - MMU: VA → PA translation fails, generates page fault exception, kernel intercepts and calls page fault handler
  - Page fault handler inspects faulted address, must bring page into memory (page in)
  - If no free page available: first select and page out victim page
  - Page in requested page, fix PTE, restart memory operation

CSE 컴퓨터공학부 Department of Computer Science & Engineering

Page Replacement Algorithms

- **Best case**: select page that will not be accessed for the longest in the future
  - Bélády's algorithm, impossible to implement
- **Random**
  - simple, but may accidentally make bad choices
- **FIFO**
  - pick page that was in memory the longest
  - simple, but may make bad choices, may suffers from Bélády's anomaly
- **LRU, MRU, LFU** (l = least, m = most, r = recently, f = frequently, u = used)
  - exploit locality
  - require keeping track of when pages are accessed
  - clock algorithm (2^{nd} chance algorithm): approximates LRU using FIFO & access bit
- **Thrashing**
  - set of regularly accessed pages > number of available physical pages
  - would bring the system to a halt
  - remedy: select a process and kill it (Linux: Out-of-Memory killer)



Module Summary

CSE 컴퓨터공학부
Department of Computer Science & Engineering

Module Summary

- Programmer's view of virtual memory
  - Each process has its own private linear address space
  - Cannot be corrupted by other processes
- System view of virtual memory
  - Uses memory efficiently by caching virtual memory pages
    - Efficient only because of locality
  - Simplifies memory management and programming
  - Simplifies protection by providing a convenient interpositioning point to check permissions