NOTICE

- Deadlines
  - (4/13 11:59 PM, Today) Programming assignment 1
  - (4/20 11:59 PM) Midterm quiz 1

TOPICS FOR TODAY

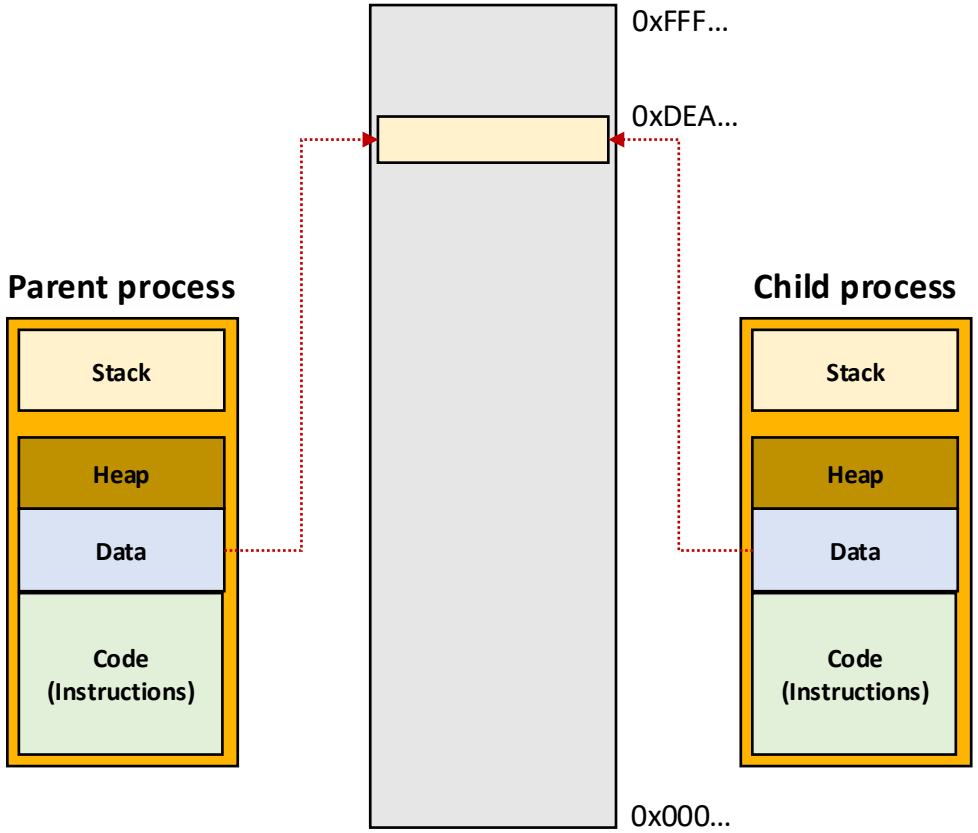
- Part I: Virtual memory
  - Provide abstraction
    - Why do we need virtual memory?
    - What is a virtual address space?
    - How does OS provide this abstraction?
  - Offer standard libraries
    - How do we observe virtual memory?
    - How do we interact with virtual memory?
    - How does malloc() work behind?
  - Manage resources
    - (OS II) Page replacement algorithms
    - (OS II) Shared memory between two processes

PROBLEM WITH PHYSICAL ADDRESSES

- Multiple processes run on a system
  - Process A uses 0xdeadbeef
  - Process B uses 0xdeadbeef
  - ?!

PROBLEM WITH PHYSICAL ADDRESSES

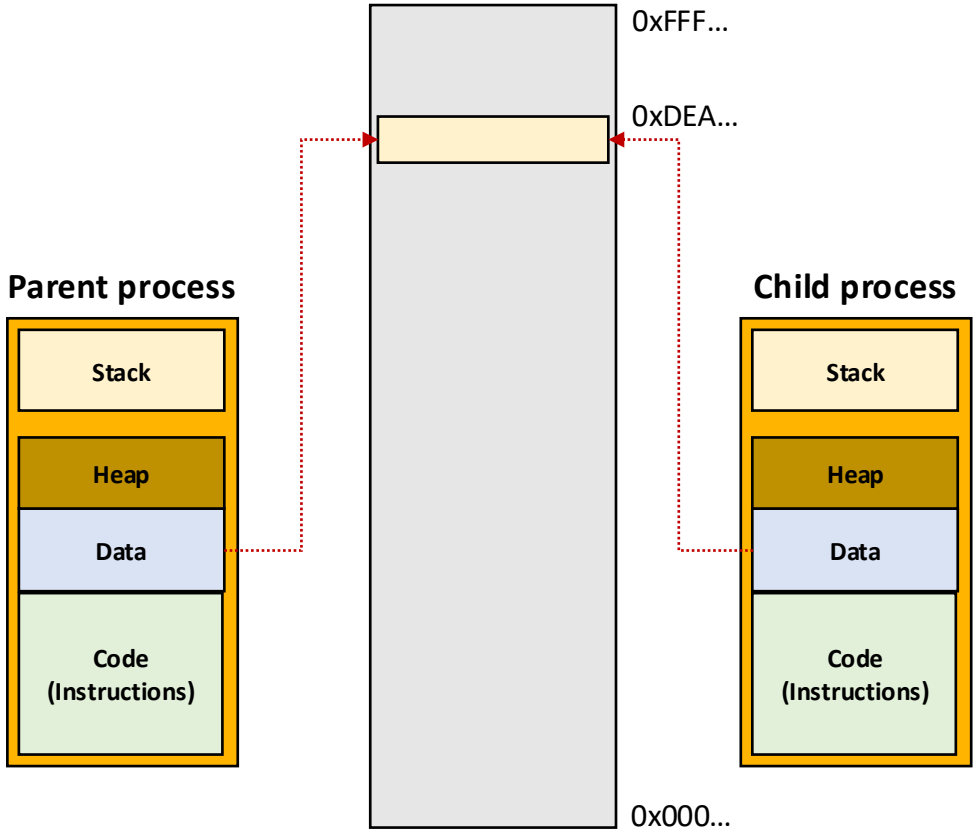
- Multiple processes run on a system
  - Process A uses 0xdeadbeef
  - Process B uses 0xdeadbeef
  - ?!



PROBLEM WITH PHYSICAL ADDRESSES

- Multiple processes run on a system
  - Process A uses 0xdeadbeef
  - Process B uses 0xdeadbeef

- (Security) Problems
  - A can read B's data
  - A can manipulate B's execution
  - A (or B) can hijack the kernel
  - … many more!



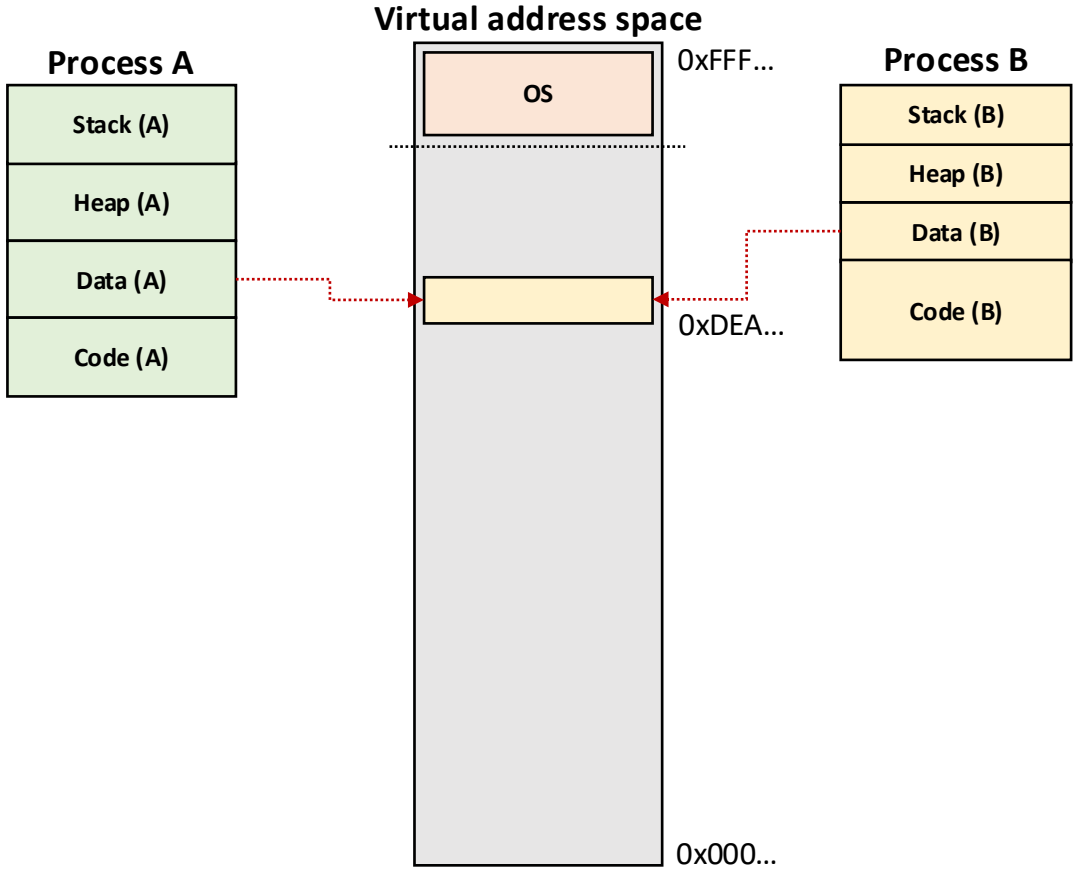
PROBLEM WITH PHYSICAL ADDRESSES

- Multiple processes run on a system
  - Process A uses 0xdeadbeef
  - Process B uses 0xdeadbeef
- (Security) Problems
  - A can read B's data
  - A can manipulate B's execution
  - A (or B) can hijack the kernel
  - … many more!
- But in reality

```
int global_var = 100;

int main() {
   pid_t pid = fork();

   if (pid == 0) {
      // Child (Process B)
      printf("[Child]  virtual address of global_var: %p\n", (void*)&global_var);
      printf("[Child]  value BEFORE write: %d\n", global_var);

      global_var = 999;       // write to "child" copy
      printf("[Child]  value AFTER  write: %d\n", global_var);

      exit(0);
   } else {
      // Parent (Process A)
      wait(NULL);  // let child finish first

      printf("[Parent] virtual address of global_var: %p\n", (void*)&global_var);
      printf("[Parent] value (unchanged): %d\n", global_var);
   }

   return 0;
}
```

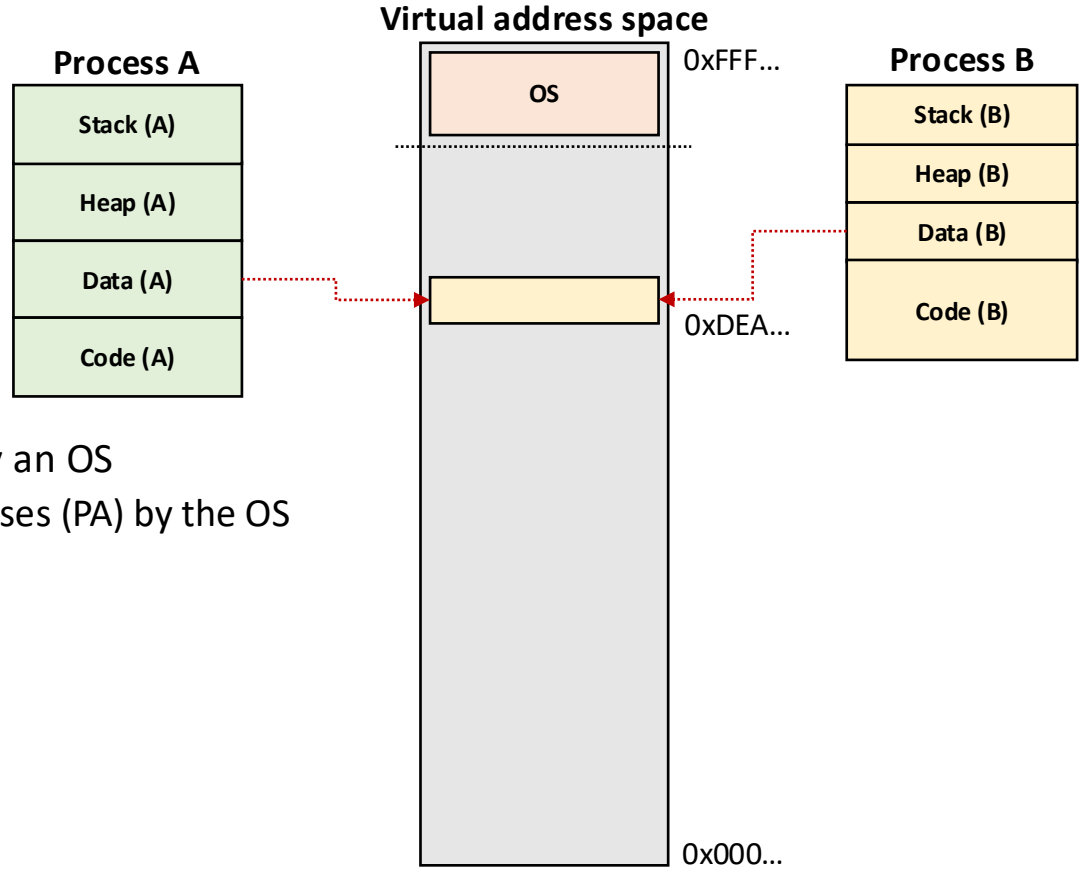
RECAP: PROCESS ISOLATION

- Process isolation
  - The same layout
    - Process A: 0x0.. – 0xFF..
    - Process B: 0x0.. – 0xFF..
  - But different phys addresses



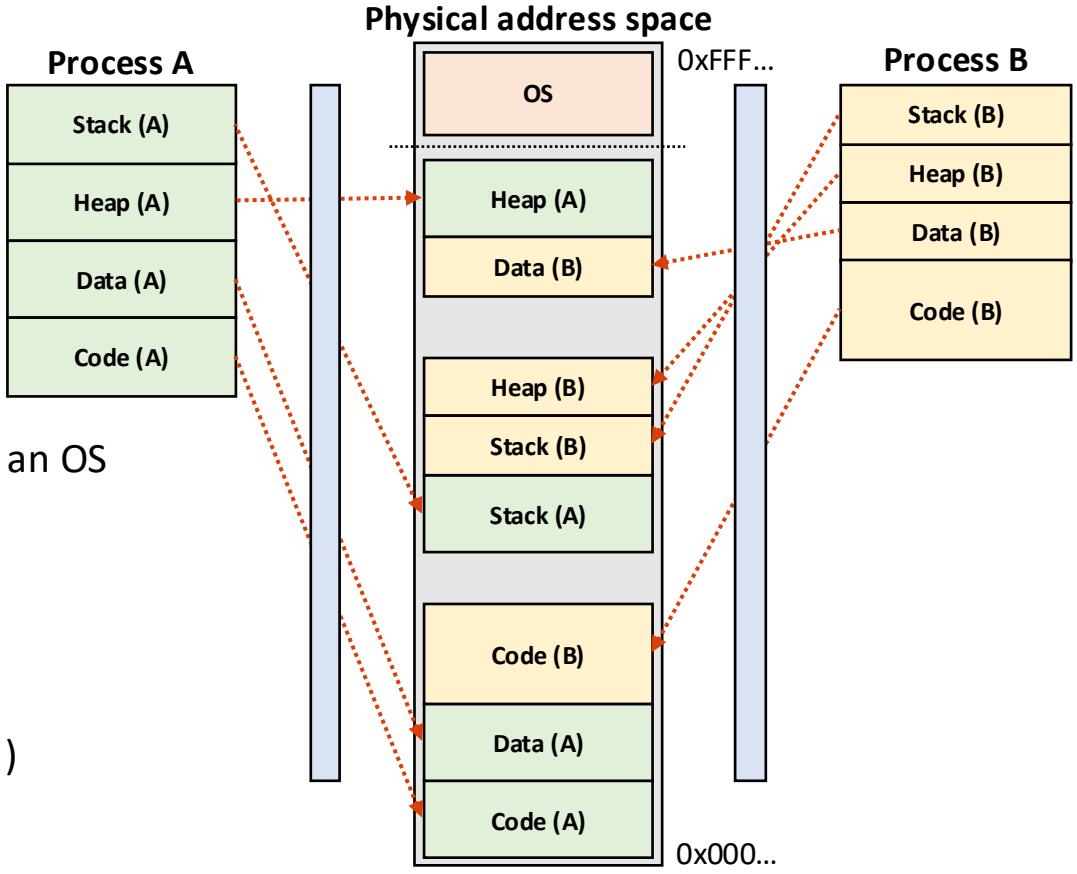
How to implement process isolation?

- Process isolation
  - The same layout
    - Process A: 0x0.. – 0xFF..
    - Process B: 0x0.. – 0xFF..
  - But different phys addresses
- Virtual address (VA)
  - A logical address generated by an OS
  - VA is mapped to physical addresses (PA) by the OS



How to enable process isolation?

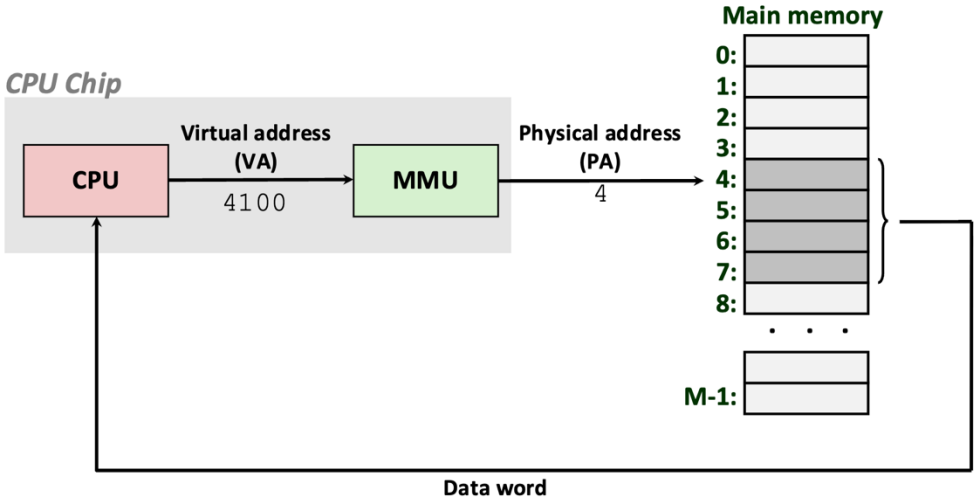
- Process isolation
  - The same layout
    - Process A: 0x0.. – 0xFF..
    - Process B: 0x0.. – 0xFF..
  - But different phys addresses
- Virtual address (VA)
  - A logical address generated by an OS
  - VA is mapped to PA by the OS
- Address translation
  - OS converts VA to PA
  - Supported by hardware (MMU)



ADDRESS TRANSLATION

- Address translation
  - OS converts VA to PA
  - OS sets the rules, *hardware does the work*

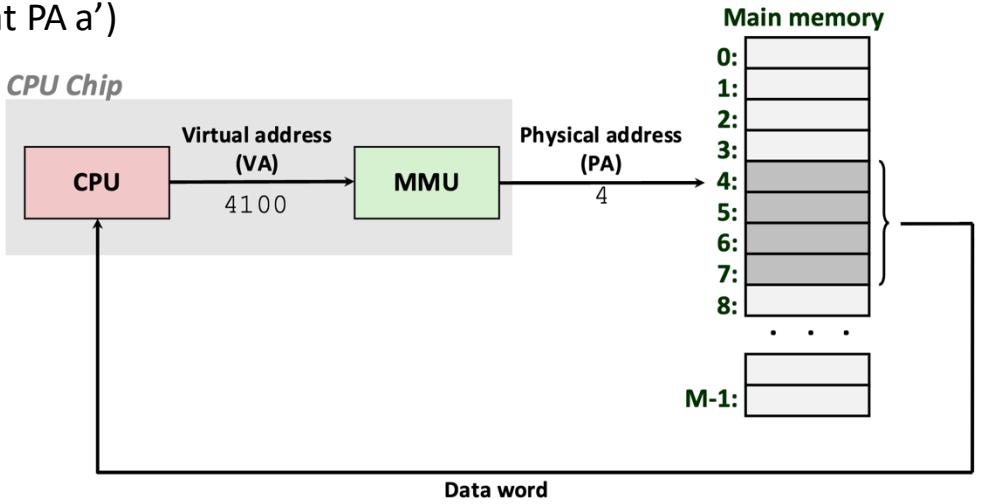
- Procedure
  - CPU accesses VA
  - MMU translates VA to PA
  - MMU works with OS
  - MMU accesses PA



[1]*Introduction to computer systems, CMU 213*

ADDRESS TRANSLATION – CONT'D

- Address translation
  - OS converts VA to PA
  - OS sets the rules, *hardware does the work*
  - Map: VA → PA U {Ø}
    - Map(a) = a' (if data at VA a is at PA a')
    - Map(a) = Ø (invalid or at disk)



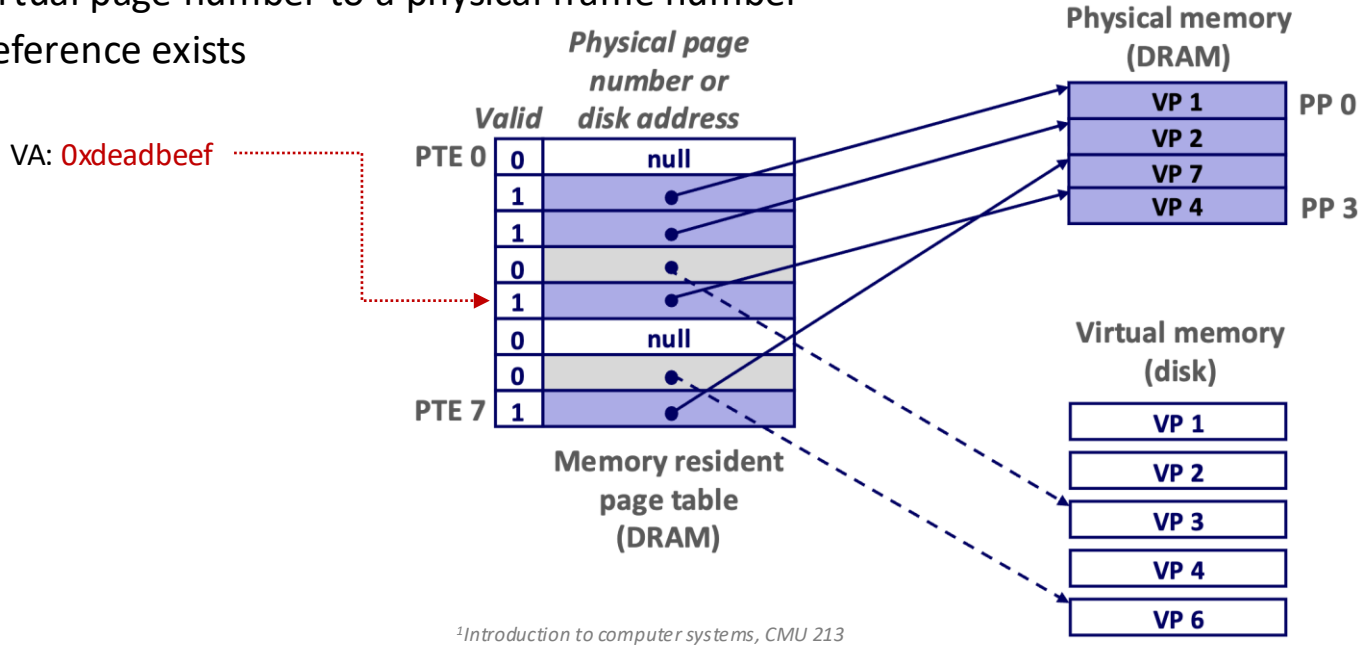
How to implement address translation?

- Paging
  - Memory management technique
  - Divide physical memory into fixed-size blocks (frames, 4kB)
  - Divide virtual memory into the same-size blocks (pages, 4kB)

- Page table
  - *Per-process*, *kernel data structure* in memory
  - An array of page table entries (PTE) that maps VA to PA
  - PTE is a 4- to 8-byte data structure
  - PTE maps a virtual page number to a physical frame number

How to implement address translation?

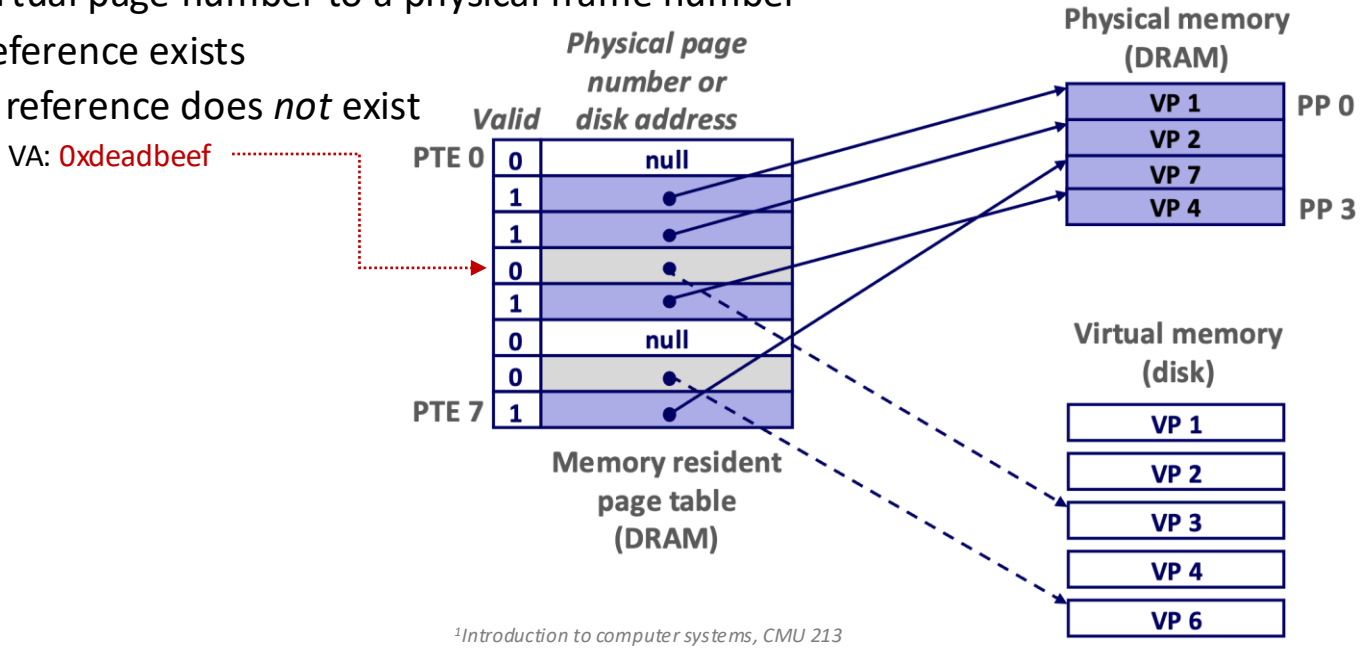
- Page table
  - *Per-process*, *kernel data structure* in memory
  - An array of PTE that maps VA to PA
  - PTE maps a virtual page number to a physical frame number
  - Page hit: PA reference exists



[1]Introduction to computer systems, CMU 213

How to implement address translation?

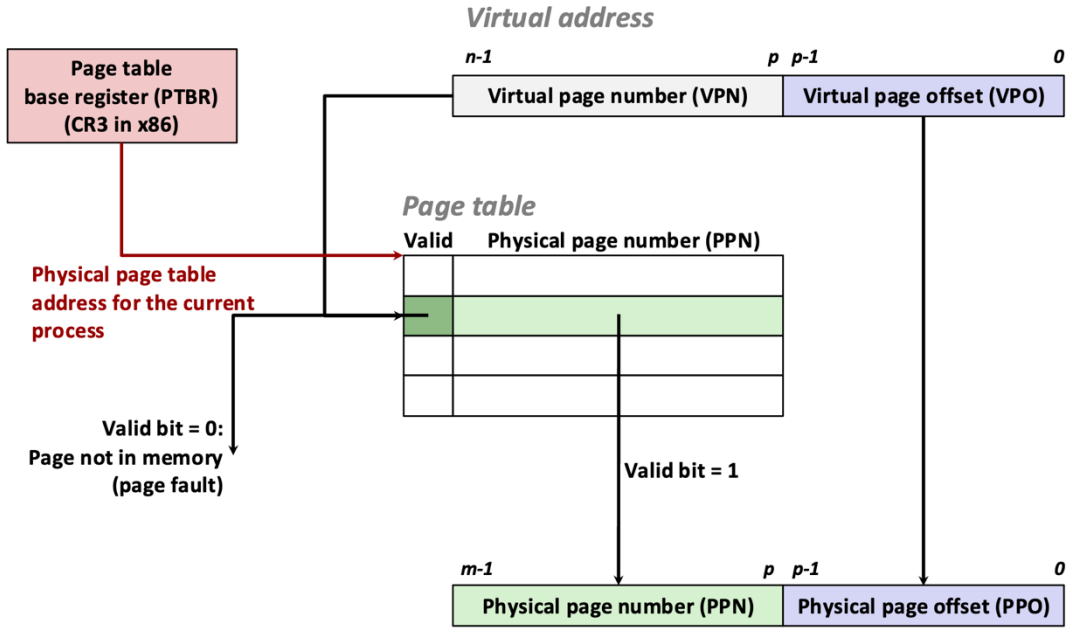
- Page table
  - *Per-process*, *kernel data structure* in memory
  - An array of PTE that maps VA to PA
  - PTE maps a virtual page number to a physical frame number
  - Page hit: PA reference exists
  - Page miss: PA reference does *not* exist



[1]Introduction to computer systems, CMU 213

How to implement address translation?

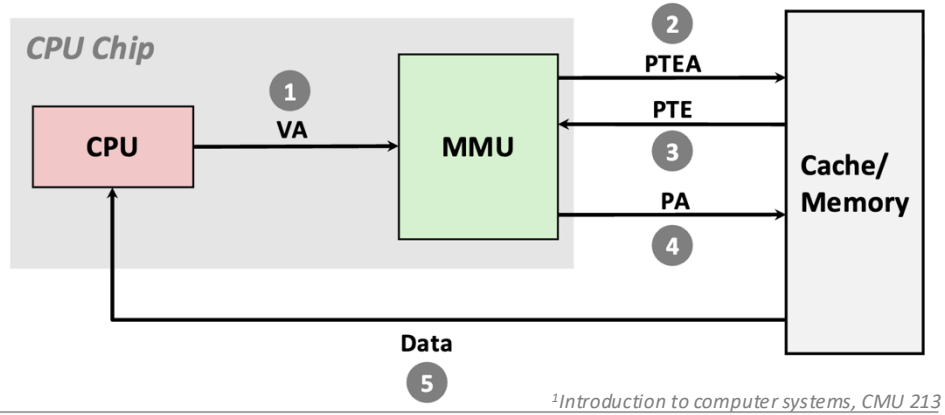
- Page table in action
  - CR3 register holds the physical page table address of the *current* process
  - VA consists of VPN and VPO
  - PA consists of PPN and PPO
- Page table operation
  - Converts VPN to PPN
  - VPN works as an offset
  - OS refers to the offset in PT
  - OS returns PPN



[1]Introduction to computer systems, CMU 213

How to implement address translation?

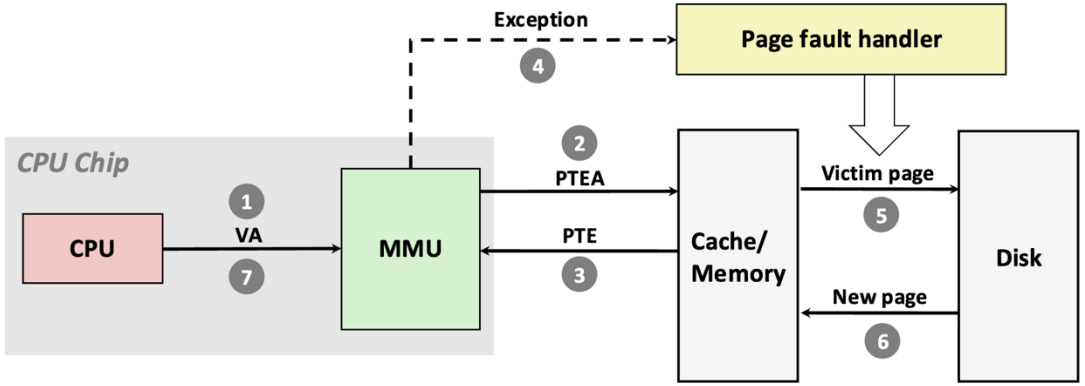
- Page hit
  - CPU sends VA to MMU
  - MMU fetches PTE from page table in memory
  - MMU and OS translate VA to PA
  - MMU sends PA to memory (or cache)
  - Memory (or cache) sends data to processor



[1] *Introduction to computer systems, CMU 213*

How to implement address translation?

- Page fault
  - CPU sends VA to MMU
  - MMU fetches PTE from page table in memory
  - MMU and OS translate VA to PA
  - MMU triggers page fault exception (PTE valid bit is zero)
  - Handler identifies victim (the page will go out to disk – evicted)
  - Handler pulls a new page and updates PTE in memory
  - Handler re-runs the VA to PA translation



[1]Introduction to computer systems, CMU 213

How to implement address translation?

- Page fault
  - CPU sends VA to MMU
  - MMU fetches PTE from page table in memory
  - MMU triggers page fault exception (PTE valid bit is zero)
  - Handler identifies victim (the page will go out to disk – evicted)
  - Handler pulls a new page and updates PTE in memory
  - Handler re-runs the VA to PA translation

- Demand paging
  - Pages won't be pulled from memory before accessed
  - Run malloc() for 1GB heap memory on a machine with 256MB DRAM → Successful!

How to implement address translation?

- Demand paging
  - Pages won't be pulled before accessed

```
int global_var = 42;

void print_maps(const char *label) {
   char path[64];
   char line[256];
   printf("\n=== %s (PID %d) /proc/self/maps ===\n", label, getpid());
   snprintf(path, sizeof(path), "/proc/%d/maps", getpid());
   FILE *f = fopen(path, "r");
   if (!f) { printf("  (could not open maps)\n"); return; }
   while (fgets(line, sizeof(line), f)) {
      if (strstr(line, "va_example1") ||
         strstr(line, "heap")  ||
         strstr(line, "stack"))
         printf("  %s", line);
   }
   fclose(f);
   fflush(stdout);
}
```

```
int main() {
   pid_t pid = fork();

   if (pid == 0) {
      global_var = 999;
      print_maps("Child");
      printf("[Child]  &global_var = %p, value = %d\n",
           (void*)&global_var, global_var);
      fflush(stdout);
      _exit(0);
   } else {
      wait(NULL);
      print_maps("Parent");
      printf("[Parent] &global_var = %p, value = %d\n",
           (void*)&global_var, global_var);
   }
   return 0;
}
```

How to implement address translation?

- Demand paging
  - OS does not allocate physical frames upon creation
  - OS allocate physical frames when accessed

- Notable examples
  - Copy on write: fork()
    - fork() does not create physical frames
    - OS sets all the pages to “read-only” and copies physical frames when accessed
  - calloc() vs. malloc()
    - calloc() fills up the new allocated memory with zeros
    - calloc() is much slower than malloc()

How to implement address translation?

- Page fault
  - CPU sends VA to MMU
  - MMU fetches PTE from page table in memory
  - MMU triggers page fault exception (PTE valid bit is zero)
  - Handler identifies victim (the page will go out to disk – evicted)
  - Handler pulls a new page, updates PTE in memory, and return the translation

- Demand paging
  - Pages won't be pulled from memory before accessed
  - Run malloc() for 1GB heap memory on a machine with 256MB DRAM → Successful!

- Thrashing
  - Many processes compete for too little DRAM
  - A system will spend most time memory swapping than executing (syllabus reading)

TOPICS FOR TODAY

- Part I: Virtual memory
  - Provide abstraction
    - Why do we need virtual memory?
    - What is a virtual address space?
    - How does OS provide this abstraction?
  - Offer standard libraries
    - How do we observe virtual memory?
    - How do we interact with virtual memory?
    - How does malloc() work behind?
  - Manage resources
    - (OS II) Page replacement algorithms
    - (OS II) Shared memory between two processes

How do we observe virtual memory?

- Process exposes its VA layout in
  - /proc/[pid]/maps
  - pmap -p [pid]
  - Example: $ cat /proc/$$/maps or $ pmap -p $$

- Output format
  - [VA start]-[VA end] [perms] [file offset] [device] [inode] [name]
  - 00401000-00402000 r-xp 00001000 00:43 4248... /path/to/binary

How do we observe virtual memory?

- Example output
  - Binary (header, .text, .rodata, …)

```
00400000-00401000 r--p 00000000 00:43 4248355350                /nfs/stak/users/hongsa/temp/va/va_example2
00401000-00402000 r-xp 00001000 00:43 4248355350                /nfs/stak/users/hongsa/temp/va/va_example2
00402000-00403000 r--p 00002000 00:43 4248355350                /nfs/stak/users/hongsa/temp/va/va_example2
…
```

  - Heap

```
01f36000-01f57000 rw-p 00000000 00:00 0                         [heap]
```

  - Shared libraries

```
7f7fede00000-7f7fede29000 r--p 00000000 fd:00 36142095          /usr/lib64/libc.so.6
7f7fede29000-7f7fedf9e000 r-xp 00029000 fd:00 36142095          /usr/lib64/libc.so.6
…
```

  - Dynamic linker

```
7f7fee103000-7f7fee105000 r--p 00000000 fd:00 36142091          /usr/lib64/ld-linux-x86-64.so.2
7f7fee105000-7f7fee12d000 r-xp 00002000 fd:00 36142091          /usr/lib64/ld-linux-x86-64.so.2
…
```

  - Stack

```
7ffff386d000-7ffff388f000 rw-p 00000000 00:00 0                 [stack]
```

  - Kernel data

```
7ffff39a6000-7ffff39aa000 r--p 00000000 00:00 0                 [vvar]
```

How do we interact with virtual memory?

- mmap()
  - Create a new mapping in the VA space
  - void *mmap(void *addr, size_t len, int prot, int flags, int fd, off_t offset)
    - *addr: the starting address for the new mapping (NULL – the kernel chooses it)
    - len: the length of the mapping
    - prot: the memory protection (PROT_EXEC, PROT_READ, PROT_WRITE, PROT_NONE)
    - flags: visibility to other processes (MAP_SHARED, MAP_SHARED_VALIDATE, MAP_PRIVATE)
    - fd: the contents of a file mapping

- munmap()
  - Delete the mapping for the specified address range
  - void *mmap(void *addr, size_t length)

How do we interact with virtual memory?

- mmap() examples
  - Mapping a file

```
int main(int argc, char *argv[]) {
    const char *path = argc > 1 ? argv[1] : "/etc/hosts";
    printf("=== File: %s ===\n\n", path);

    read_with_syscall(path);
    read_with_mmap(path);

    int fd = open(path, O_RDONLY);
    struct stat st; fstat(fd, &st);
    char *data = mmap(NULL, st.st_size,
                      PROT_READ, MAP_PRIVATE, fd, 0);
    close(fd);

    printf("\n/proc/self/maps (relevant lines):\n");
    system("grep -E 'hostname|heap|stack' /proc/self/maps");

    munmap(data, st.st_size);
    return 0;
}
```

```
// Method A: classic read()
void read_with_syscall(const char *path) {
    int fd = open(path, O_RDONLY);
    struct stat st; fstat(fd, &st);
    char *buf = malloc(st.st_size + 1);

    read(fd, buf, st.st_size);
    buf[st.st_size] = '\0';

    printf("[read()]  first 40 chars: %.40s\n", buf);
    free(buf);
    close(fd);
}

// Method B: mmap()
void read_with_mmap(const char *path) {
    int fd = open(path, O_RDONLY);
    struct stat st; fstat(fd, &st);

    char *data = mmap(NULL, st.st_size,
                      PROT_READ, MAP_PRIVATE, fd, 0);
    close(fd);
    printf("[mmap()]  first 40 chars: %.40s\n", data);

    printf("[mmap()]  mapped at VA:   %p\n", (void*)data);
    munmap(data, st.st_size);
}
```

How do we interact with virtual memory?

- mmap() examples
  - Mapping a file
  - read():
    - Disk
    - Kernel buffer
    - User buffer
    - Process uses data
  - mmap():
    - Disk
    - Kernel buffer
    - VA space (no copy)
    - Process uses data

```
// Method A: classic read()
void read_with_syscall(const char *path) {
   int fd = open(path, O_RDONLY);
   struct stat st; fstat(fd, &st);
   char *buf = malloc(st.st_size + 1);

   read(fd, buf, st.st_size);
   buf[st.st_size] = '\0';

   printf("[read()]  first 40 chars: %.40s\n", buf);
   free(buf);
   close(fd);
}

// Method B: mmap()
void read_with_mmap(const char *path) {
   int fd = open(path, O_RDONLY);
   struct stat st; fstat(fd, &st);

   char *data = mmap(NULL, st.st_size,
                     PROT_READ, MAP_PRIVATE, fd, 0);
   close(fd);
   printf("[mmap()]  first 40 chars: %.40s\n", data);

   printf("[mmap()]  mapped at VA:   %p\n", (void*)data);
   munmap(data, st.st_size);
}
```

How do we interact with virtual me

- mmap() examples
  - Anonymous mapping
    - malloc()'s internals
    - Kernel manages the mapping
  - mprotect ():
    - Changes the access permission

```
int main() {
   size_t size = 4096 * 3;
   char *buf = mmap(NULL, size,
                    PROT_READ | PROT_WRITE,
                    MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);
   printf("Anonymous mapping at: %p\n", (void*)buf);

   int all_zero = 1;
   for (size_t i = 0; i < size; i++)
      if (buf[i] != 0) { all_zero = 0; break; }
   printf("Zero-filled: %s\n", all_zero ? "yes" : "no");

   strcpy(buf, "hello from mmap");
   printf("Wrote: %s\n", buf);

   printf("\n/proc/self/maps (anon region):\n");
   char cmd[128];
   snprintf(cmd, sizeof(cmd),
            "grep -A2 -B2 '%lx' /proc/self/maps", (unsigned long)buf);
   system(cmd);

   mprotect(buf, size, PROT_READ);
   printf("\nAfter mprotect(PROT_READ):\n");
   system(cmd);  // permissions column should now show r--p

   munmap(buf, size);
   return 0;
}
```

TOPICS FOR TODAY

- Part I: Virtual memory
  - Provide abstraction
    - Why do we need virtual memory?
    - What is a virtual address space?
    - How does OS provide this abstraction?
  - Offer standard libraries
    - How do we observe virtual memory?
    - How do we interact with virtual memory?
    - How does malloc() work behind?
  - Manage resources
    - (OS II) Page replacement algorithms
    - (OS II) Shared memory between two processes

Thank You!

M/W 12:00 – 1:50 PM (LINC #200)

Sanghyun Hong

sanghyun.hong@oregonstate.edu