DCS216 Operating Systems

Lecture 20
Memory (3)
Demand Paging

May 15th, 2024

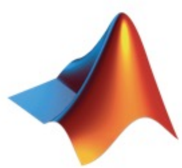
Instructor: Xiaoxi Zhang

Sun Yat-sen University

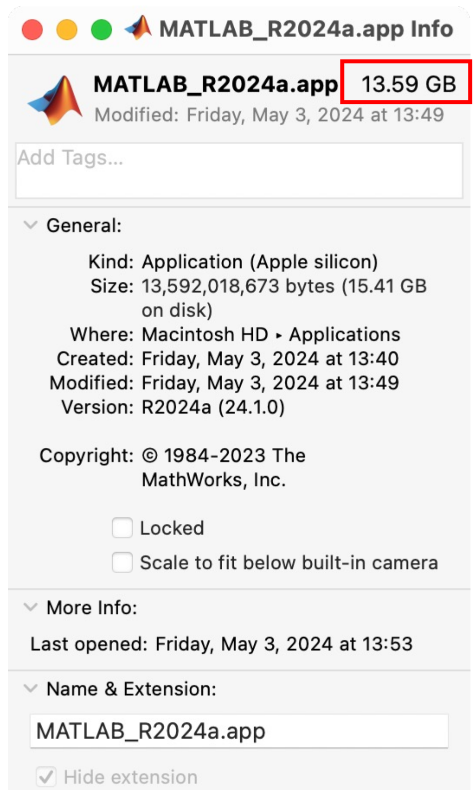
Content

Background

- So far, we require that the entire program be loaded into memory **before** it can execute.
- **Question:** How to execute a program that is **larger** (in size) **than** the physical memory available on the system?



MATLAB_R2024a.
app

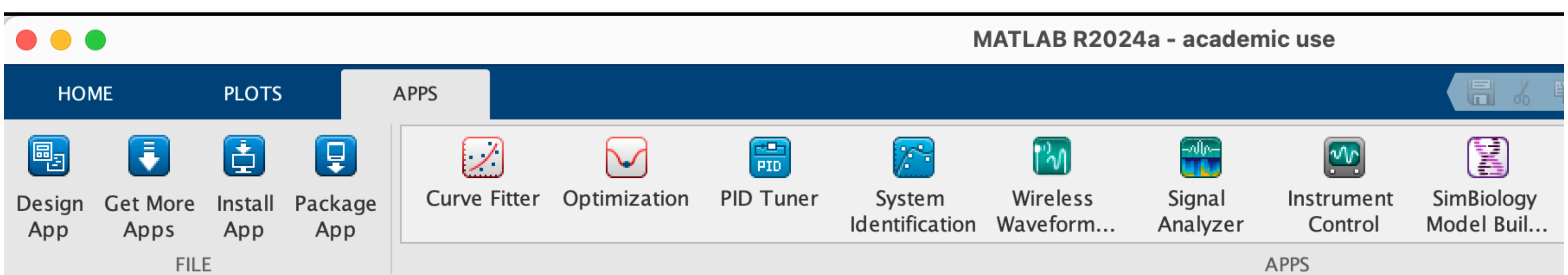


M2

8 核中央处理器
8 核图形处理器
8GB 统一内存
256GB 固态硬盘[1]

Background

- The requirement that instructions must be in physical memory to be executed *seems* both **necessary** and **reasonable**.
- However, it limits the size of a program to the size of available physical memory.
- In fact, a closer look at real programs (such as *Matlab*) shows us that, in many cases, the entire program is not needed:
  - Programs often have code to handle unusual error conditions.
    - These errors seldom occur in practice $\Rightarrow$ almost never executed.
  - Certain options and features of a program may be used very rarely.
    - For example, Matlab has many toolboxes installed by default, but rarely used unless explicitly invoked.

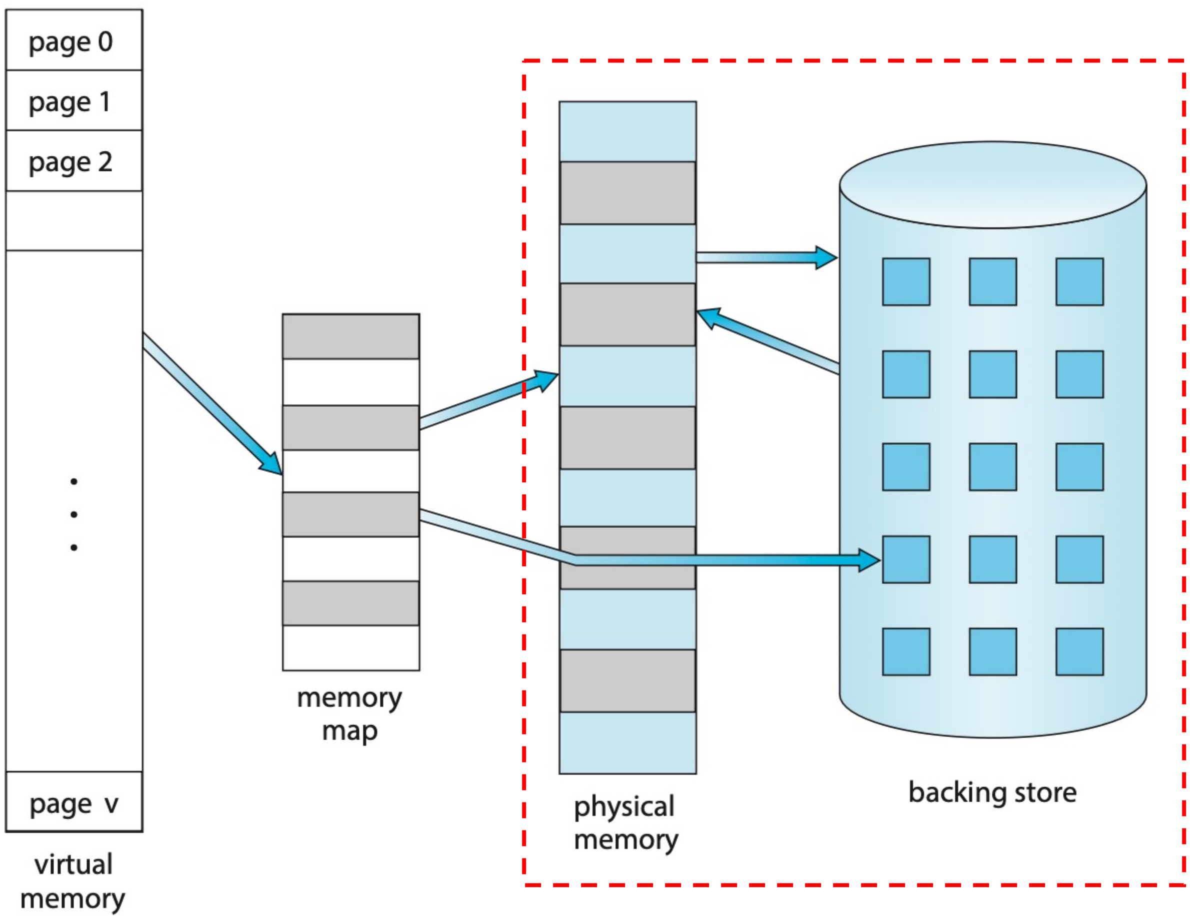


Background

- Even in those cases where the entire program is needed, it may not all be needed **at the same time**.
- Consider the *ability* to execute **partially-loaded** program
  - Program size no longer **constrained** by limits of physical memory
  - Each program takes **less** physical memory while running
    - $\Rightarrow$ more programs running at the same time
    - Increased CPU utilization and throughput with no increase in response time or turnaround time
  - Less I/O needed to load or swap programs into memory
    - $\Rightarrow$ each program runs faster

Virtual Memory

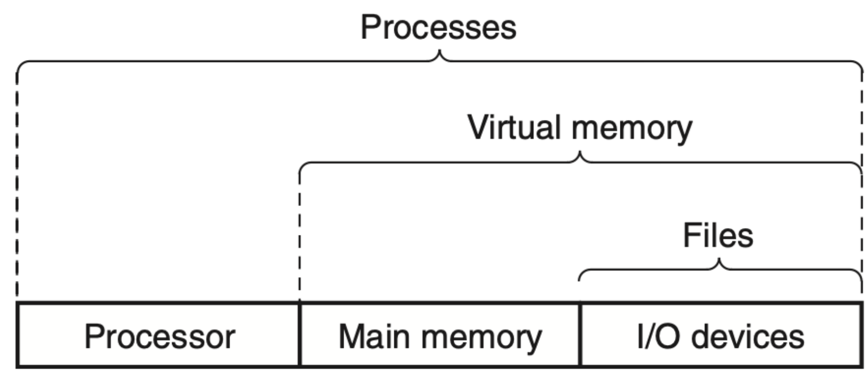
- **Virtual Memory:** separation of **logical** memory from **physical** memory
  - It allows an extremely large virtual memory to be provided for developers when only a smaller physical memory is available.

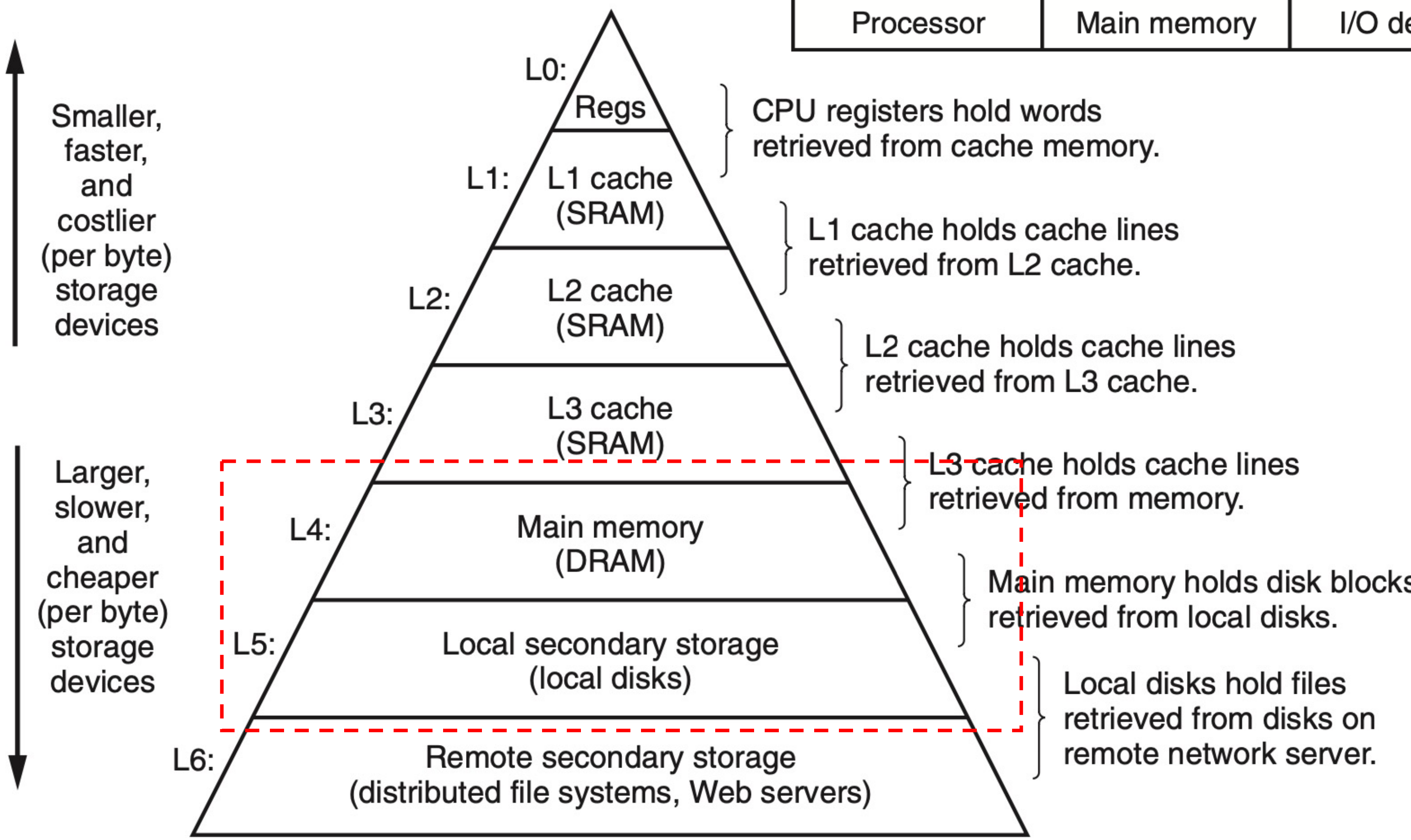


Pages in **Virtual Memory** may reside in either physical memory or backing store.

Virtual Memory

- The Memory Hierarchy





Virtual Memory

Virtual Memory

- **Virtual Memory:** separation of **logical** memory from **physical** memory
  - Only part of the program needs to be in memory for execution
  - Logical address space can therefore be much larger than physical address space
  - Allows address spaces to be shared by several processes
  - Allows for more efficient process creation
  - More programs running concurrently
  - Less I/O needed to load or swap processes
- **Virtual Address Space:** logical view of how processes is stored in memory
  - Usually start at address `0`, contiguous address until end of space
  - Meanwhile, physical memory organized in physical frames
  - MMU must map logical address into physical address.

Virtual Memory

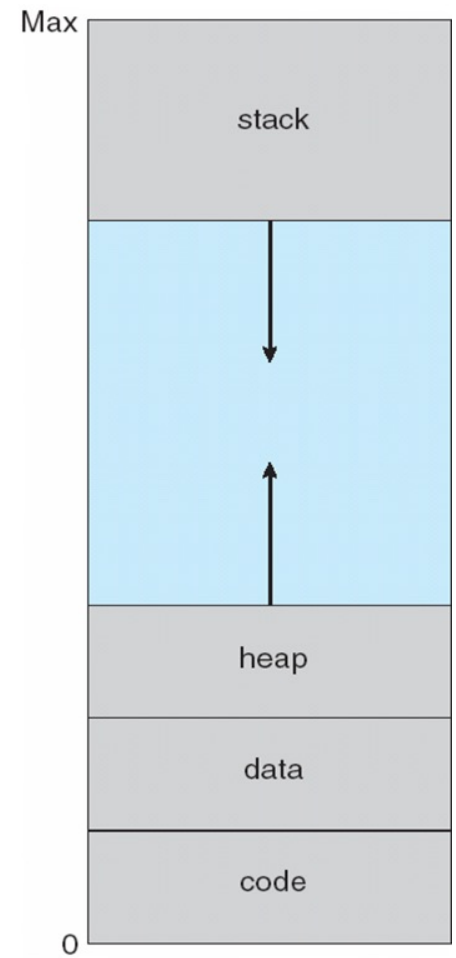
- A tiny example:
  - 4-page/**frame** Physical Memory
  - 8-page/**block** Backing Store (Swap Space) on disk
  - 4 Processes
    - Each process has its own **Page Table**
      - i.e., mappings from VPN to PFN
    - 3 active Processes (**Proc 0, Proc 1, Proc 2**)
      - some of their valid pages in memory
    - 1 inactive Process (**Proc 3**)
      - all of its pages **swapped out** to disk.

	PFN 0	PFN 1	PFN 2	PFN 3
Physical Memory	Proc 0 [VPN 0]	Proc 1 [VPN 2]	Proc 1 [VPN 3]	Proc 2 [VPN 0]

	Block 0	Block 1	Block 2	Block 3	Block 4	Block 5	Block 6	Block 7
Backing Store (Swap Space)	Proc 0 [VPN 1]	Proc 0 [VPN 2]	[Free]	Proc 1 [VPN 0]	Proc 1 [VPN 1]	Proc 3 [VPN 0]	Proc 2 [VPN 1]	Proc 3 [VPN 1]

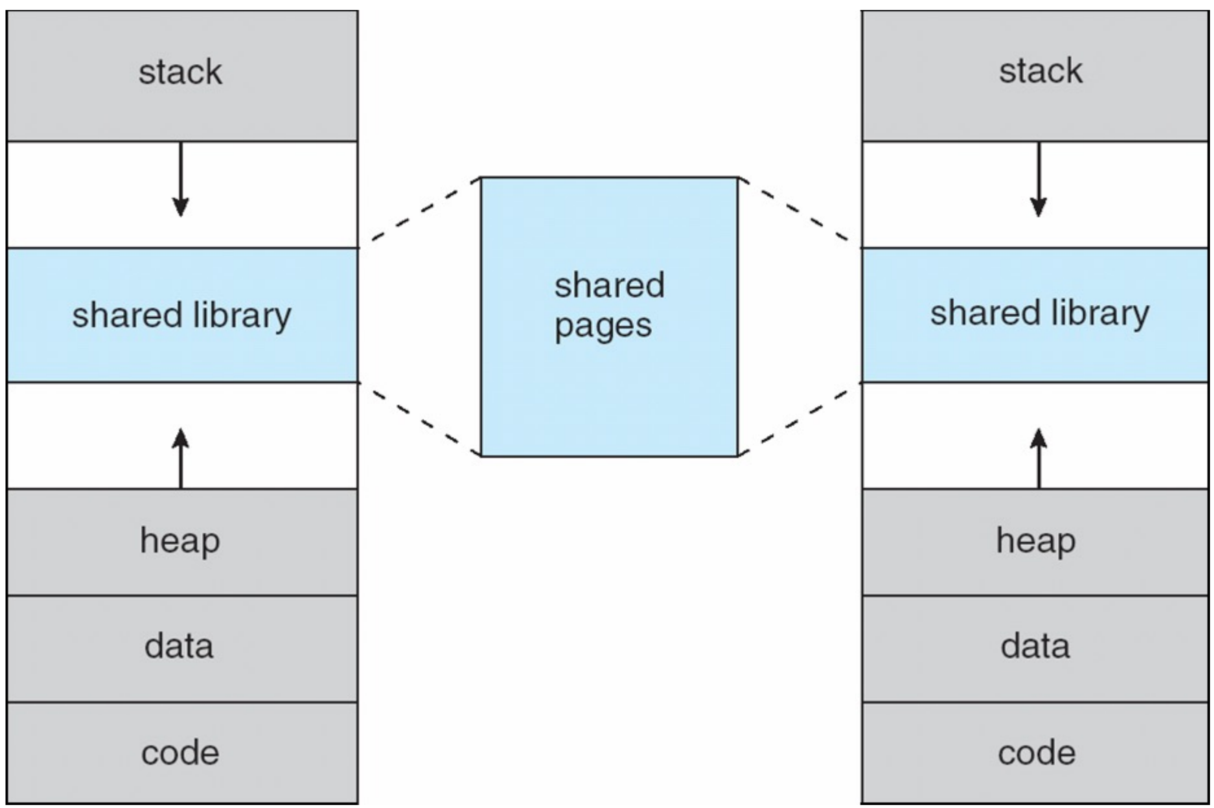
Virtual Address Space

- Usually design virtual address space for stack to start at Max addr and grow downward, while heap grows upward.
  - Maximize address space use
  - Unused address space between stack and heap
    - No physical memory needed until heap or stack grows to a new page.
- Enable **sparse** address spaces with holes left for growth, dynamically linked libraries, etc.



Virtual Address Space

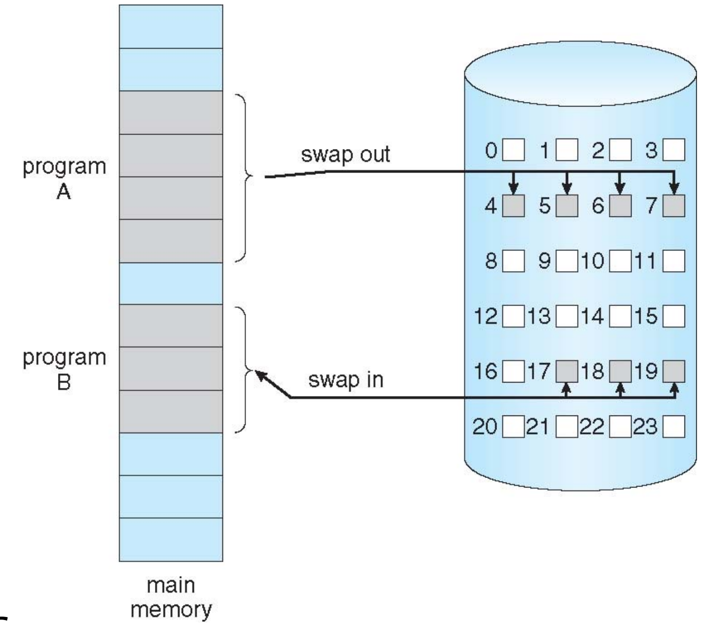
- **Shared Library** Using Virtual Memory
  - System libraries shared via mapping into virtual address space
- **Shared Memory** by mapping pages into virtual address space
- Pages can be shared during `fork()`, speeding process creation



Demand Paging

Basic Concepts

- Bring a page into memory only when it is needed (**on demand**).
  - Less I/O needed, no unnecessary I/O
  - Less memory needed
  - Faster response
  - More users.
- Page is needed $\Rightarrow$ reference to it
  - invalid and illegal reference $\Rightarrow$ abort
  - invalid not-in-memory $\Rightarrow$ bring to memory



- Similar to paging system with swapping
  - Pager is the swapper that deals with pages
  - With swapping, pager guesses which pages will be used before swapping out again.
- Lazy swapper – Never swaps a page into memory unless page will be needed.
  - Pager brings only those pages needed into memory.

Demand Paging

Basic Concepts

Valid/Invalid Bit

- Within each **Page Table Entry** (**PTE**), a **valid** bit is needed *(also called **present** bit on some systems)*
  - `v (or 1)`: page in memory (**valid**)
  - `i (or 0)`: page not in memory (**invalid**)
- Initially set valid bits to `i (0)` for all entries

Page Table

Frame #	v-i bit
	v
	v
	v
	v
	i
… …	
	i
	i

- During address translation, if valid bit in PTE is `i`, then **page fault**.
  - invalid bit **does not** indicate **illegal** virtual address

Illegal Address Reference Examples

- **Illegal** address reference generally triggers segmentation fault.

```c
/* illegal_addr1.c */
#include <stdio.h>

int main() {
    int *p = NULL;
    // p is a NULL pointer
    *p = 42;
    // dereferencing a NULL pointer
    //      --> segfault
    printf("*p: %d\n", *p);
    return 0;
}
```

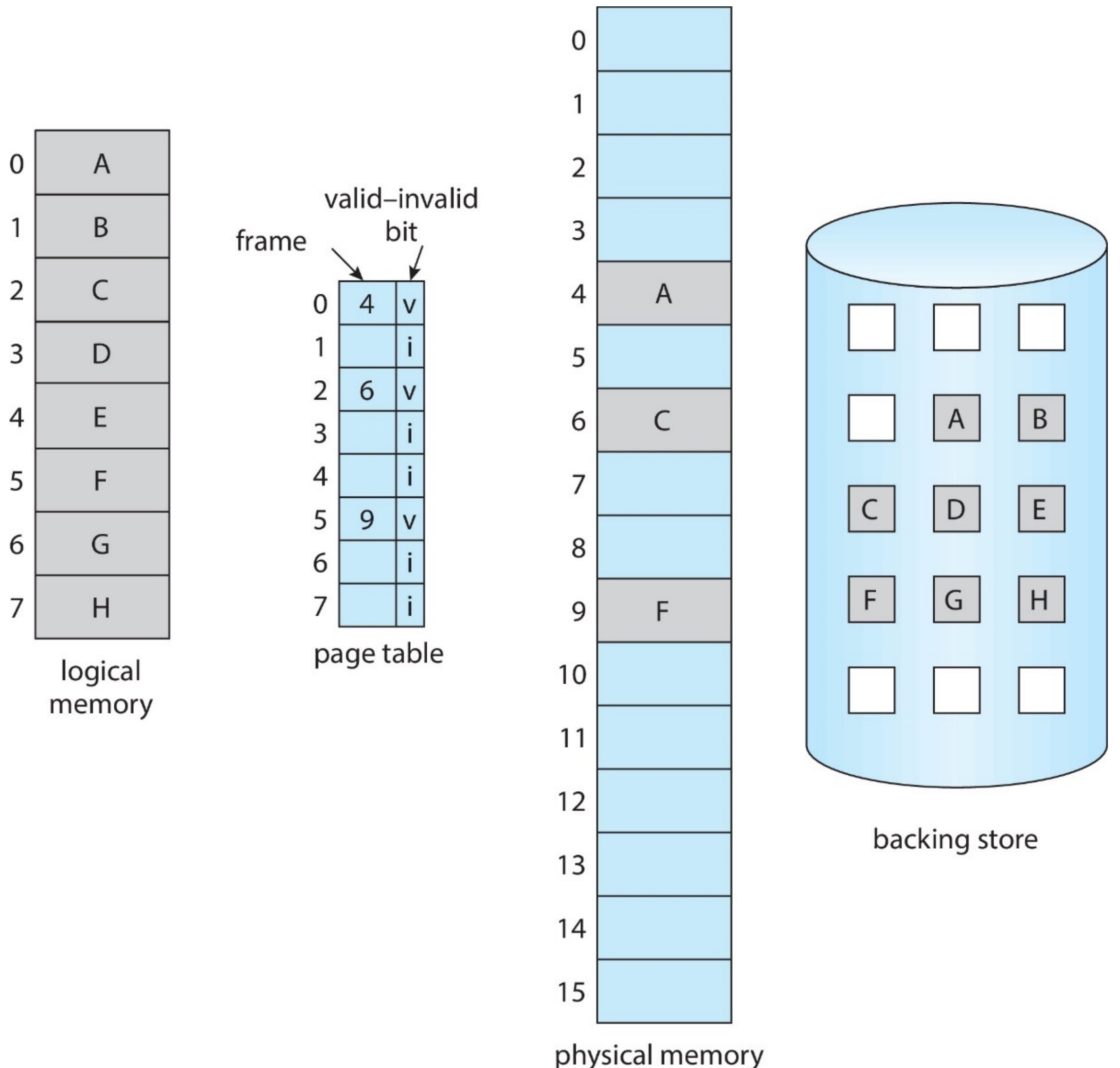
```c
/* illegal_addr2.c */
#include <stdio.h>

int main() {
    int *p = (void *)0xFFFF880000000000;
    // p points kernel address space
    *p = 42;
    // trying to access kernel addres space
    //      --> segfault
    printf("*p: %d\n", *p);
    return 0;
}
```

```c
/* illegal_addr3.c */
#include <stdio.h>

int main() {
    int *p = (void *)main;
    // p points to addr of main
    *p = 42;
    // trying to modify the first 4 bytes of main (read-only code segment)
    //      --> segfault
    printf("*p: %d\n", *p);
    return 0;
}
```

Demand Paging

Basic Concepts

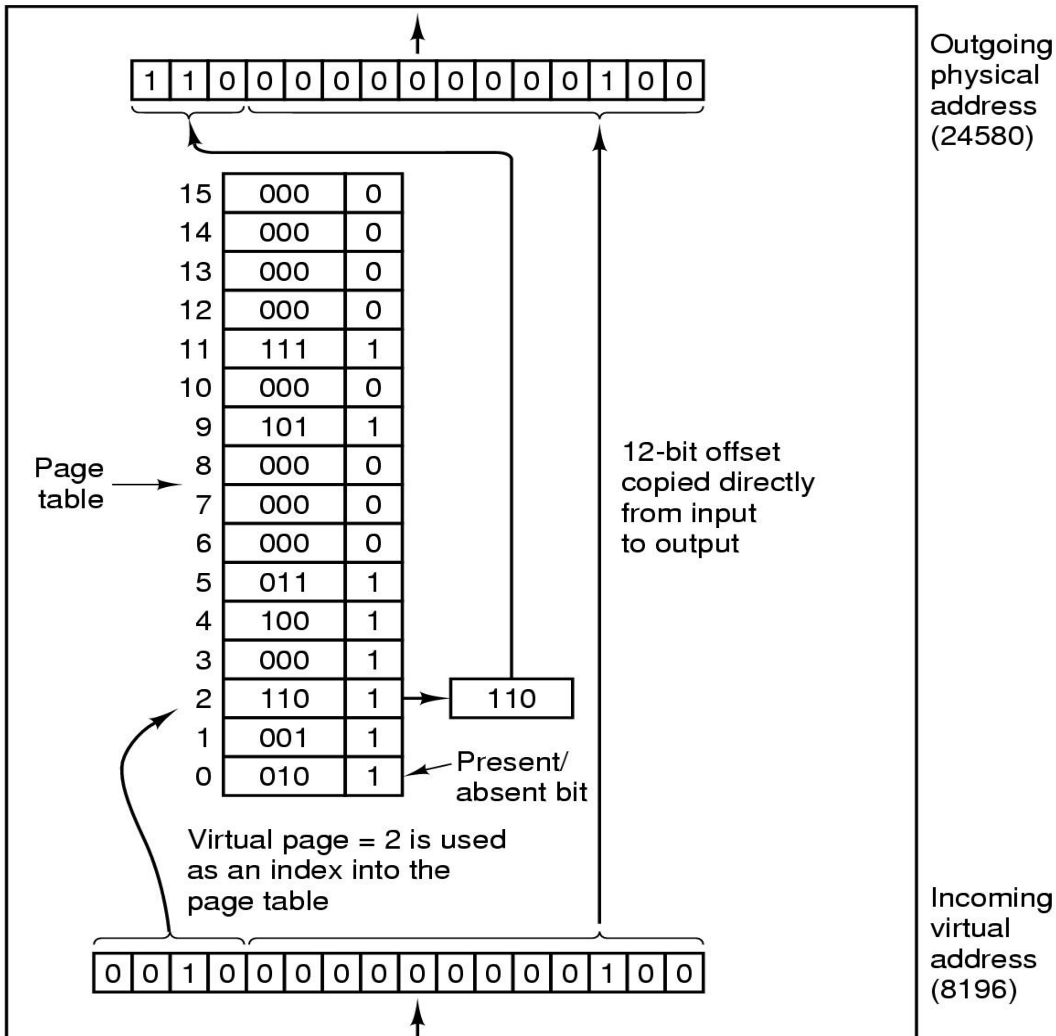
- **Page Table** when some pages are not in main memory



Demand Paging

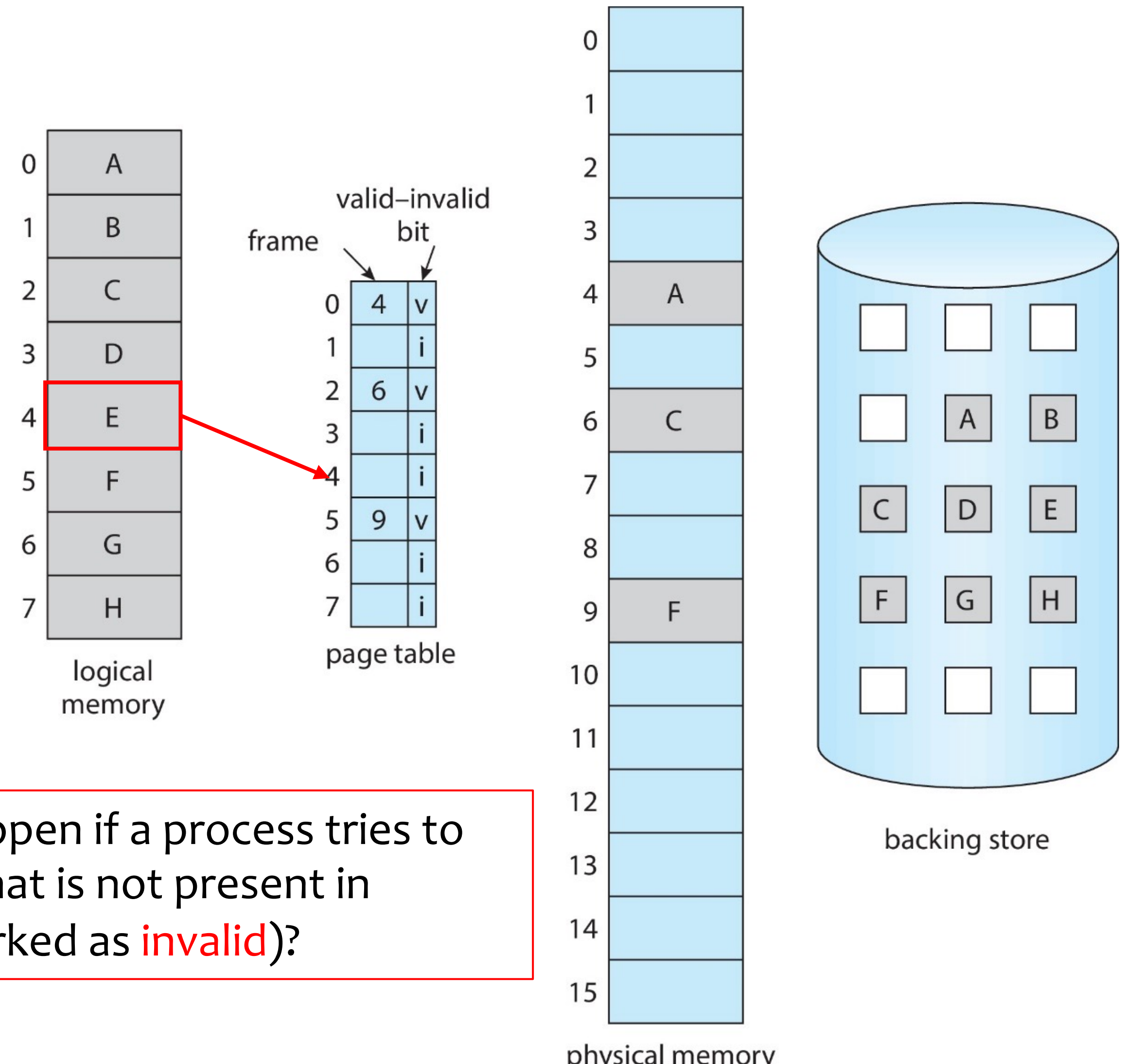
Basic Concepts

- Virtual Memory Mapping Example



Demand Paging

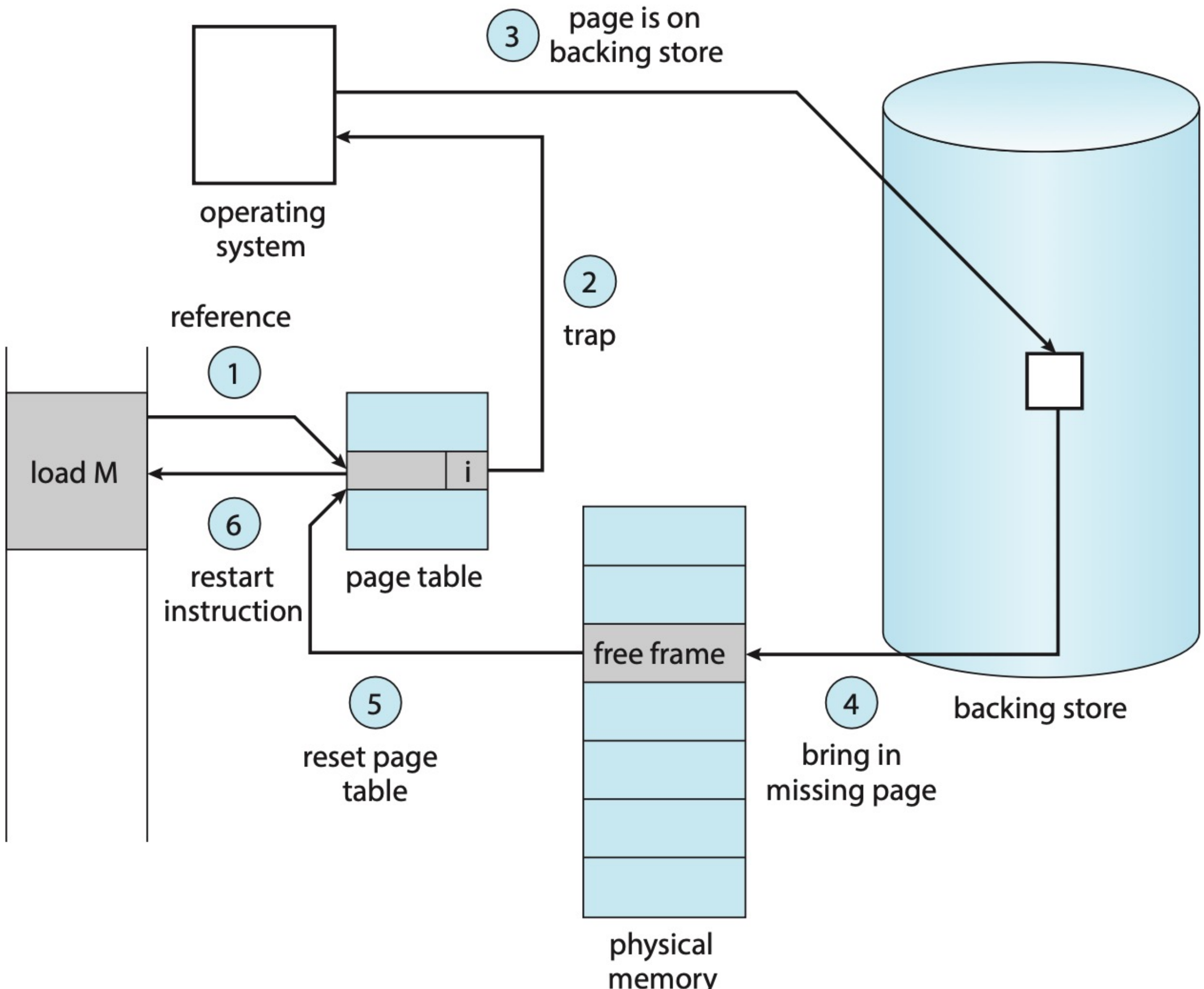
Page Fault



What would happen if a process tries to access a page that is not present in memory (or marked as invalid)?

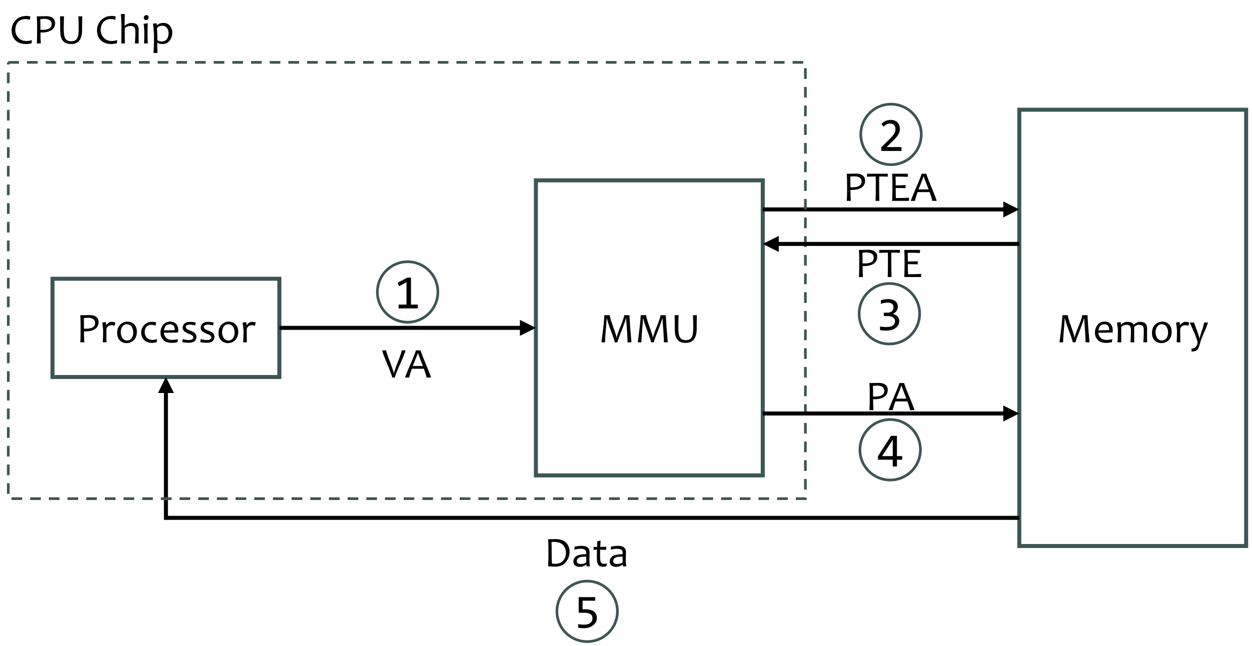
Page Fault

- Access to a page marked as invalid causes a Page Fault (缺页错误).

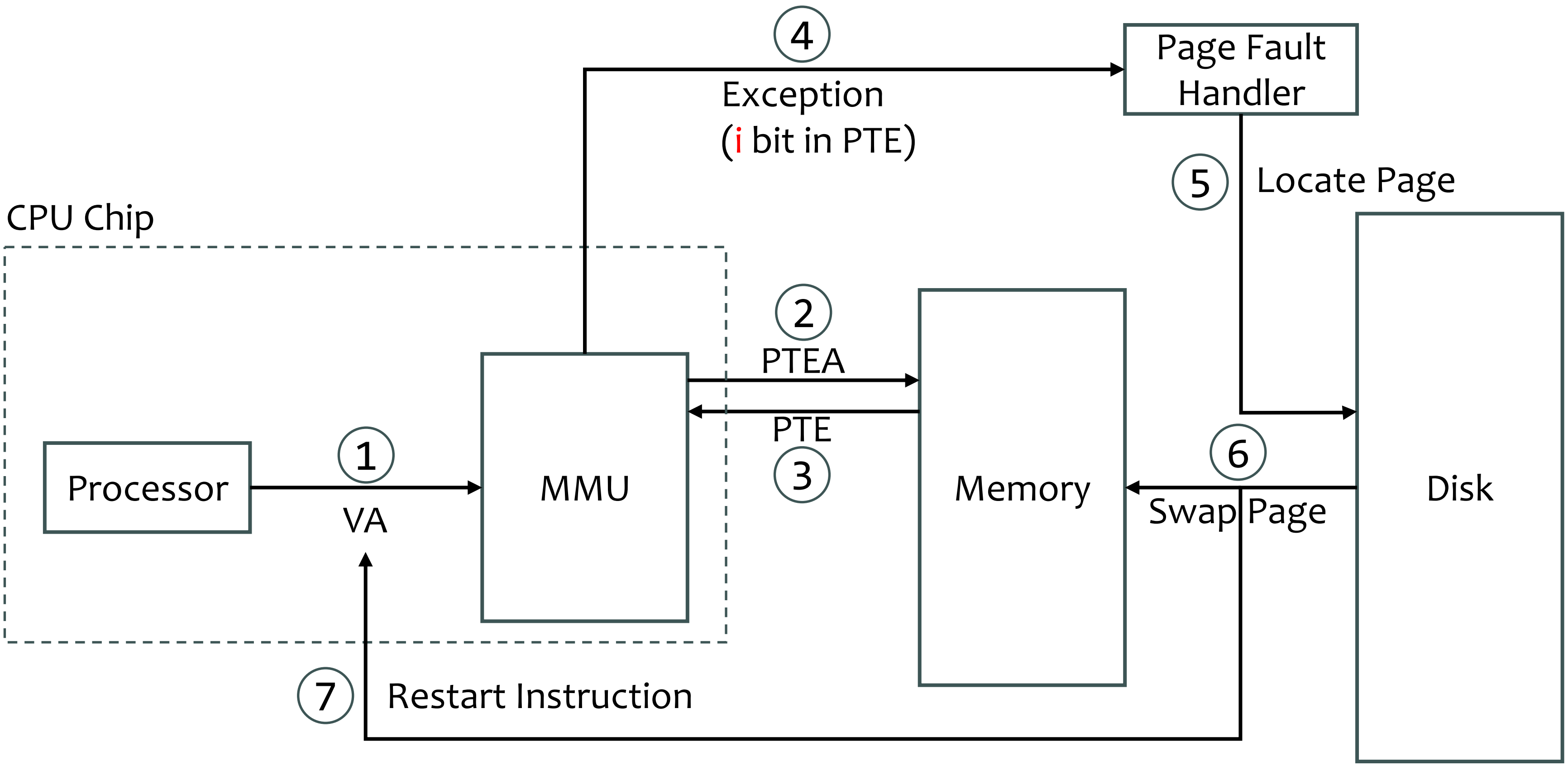


- **Page Fault**
  - Access to a page marked as invalid causes a **Page Fault**.
  - Steps in handling a **Page Fault**:
    1. **Check** an internal table (usually kept within the **PCB**) for this process to determine whether the memory **reference** was **legal** or **illegal**.
    2. If the reference is **legal** and the **valid bit** in **PTE** is invalid (page not in memory, but in backing store), **trap** into the **Page Fault Handler**.
    3. **Locate** the desired page/block on **disk** (swap space).
    4. Find a **free frame** (e.g., by taking one from the **free-frame list**) and swap the desired page/block from disk into the **free frame** via **scheduled disk read operation**.
    5. When the **storage read** is complete, we modify the internal table kept within the **PCB** and the **Page Table** to indicate that the page is now in memory (i.e., by setting the valid bit to `1 (valid)` in the **PTE**).
    6. **Restart** the instruction that was interrupted by the **trap**. *The process can now access the page as though it had always been in memory.*

Page Hit



Page Fault

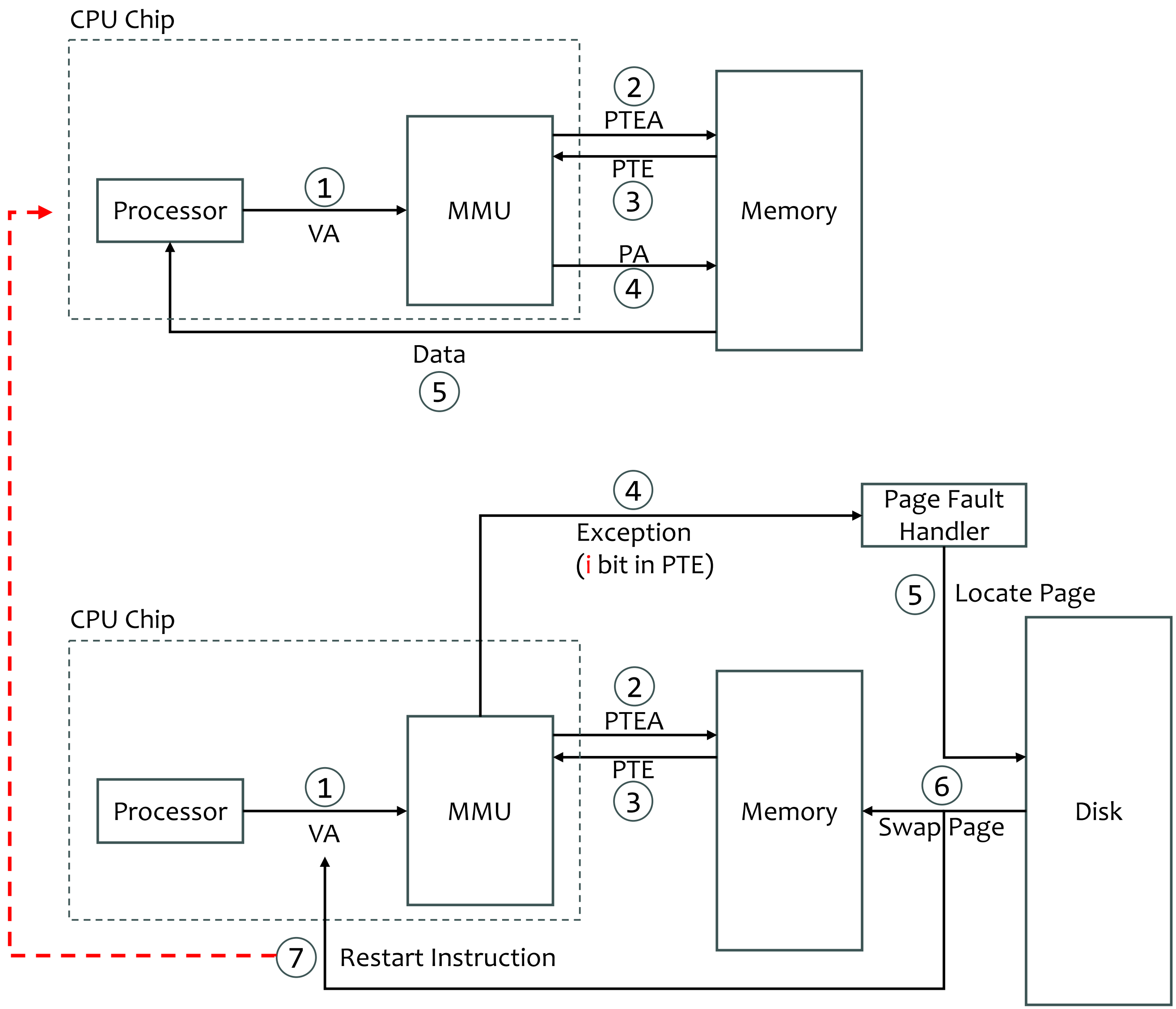


4
Exception
(i bit in PTE)
Page Fault
Handler
5
Locate Page
CPU Chip
2
PTEA
PTE
3
1
Processor
VA
MMU
Memory
6
Swap Page
Disk
7
Restart Instruction

Page Hit vs. Page Fault

Page Hit:

Page Fault:



Dynamics of Demand Paging

- Typically, each process has *its own* **page table.**
- Each **Page Table Entry** (**PTE**) contains a valid bit to indicate whether the page is in memory or not
  - If it is in main memory, the **PTE** contains the frame number (**PFN**) of the corresponding frame in main memory
  - Otherwise, the **PTE** may contain the address of the **page** block on **disk**.
- Example: Intel x86 architecture **PTE:**
  - 2-Level Page Table (10, 10, 12-bit offset)
  - Intermediate Page Tables called "Directories"

Present Bit
(valid bit)

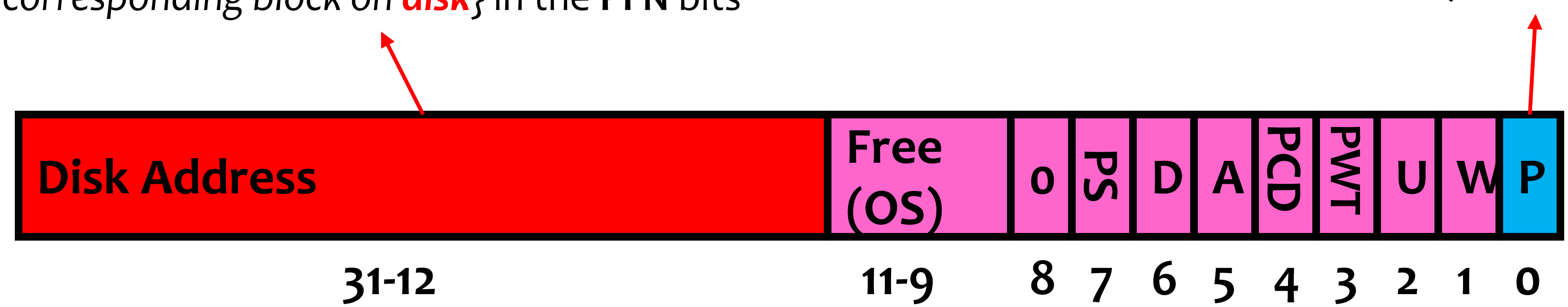
Physical Frame Number	Free (OS)	0	PS	D	A	PCD	PWT	U	W	P
31-12	11-9	8	7	6	5	4	3	2	1	0

Demand Paging

- **Dynamics of Demand Paging**
  - Typically, each process has *its own* **page table.**
  - Each **Page Table Entry** (**PTE**) contains a valid bit to indicate whether the page is in memory or not
    - If it is in main memory, the **PTE** contains the frame number (**PFN**) of the corresponding frame in main memory
    - Otherwise, the **PTE** may contain the address of the **page** block on **disk**.
  - Example: Intel x86 architecture **PTE**:
    - 2-Level Page Table (10, 10, 12-bit offset)
    - Intermediate Page Tables called "Directories"

Some OSes store the **address** of {*the corresponding block on **disk***} in the **PFN** bits

Present Bit (valid bit)



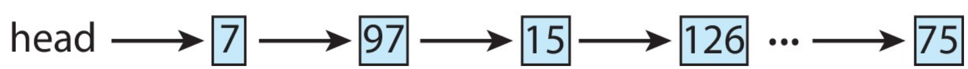
Demand Paging

Aspects of Demand Paging

- Extreme case – start process with **no** pages in main memory.
  - OS sets Instruction Pointer to the first instruction of process
    - which is on a non-memory-resident page ⇒ triggers **Page Fault**.
  - ..and for every other pages upon **first** access
  - This is called **Pure Demand Paging**.
    - **never bring a page into memory until it is required.**
- Actually, a given instruction could access multiple pages and cause multiple page faults.
  - Consider fetch and decode of instruction which adds two numbers fro memory and stores result back to memory.
  - Pain decreased because of locality of reference.
- Hardware support needed for demand paging.
  - Page Table with valid/invalid bit
  - Secondary Memory (swap device with swap space)
  - Instruction Restart

Free-Frame List

- When a page fault occurs, the OS must bring the desired page from secondary storage (swap space) into main memory.
- Most OSes maintain a **free-frame list** – a pool of free frames for satisfying such requests.



- The OS typically allocates free frames using a technique known as **zero-fill-on-demand**:
  - the content of the frames zeroed-out before being allocated.
- When a system starts up, all available memory is placed on the **free-frame list**.

Stages in Demand Paging (the Worst Case)

1. Trap into the Operating System
2. Save the user registers and process state
3. Determine that the interrupt was a page fault
4. Check that the page reference was legal and determine that the location of the block (page) on disk.
5. Issue a read request from the disk to a free frame:
   1. Wait in a queue for this device until the read request is serviced
   2. Wait for the device seek and/or latency time
   3. Begin the transfer of the page to a free frame
6. While waiting, allocate the CPU to some other user
7. Receive an interrupt from the disk I/O subsystem (I/O completed)
8. Save the registers and process state for the other user
9. Determine that the interrupt was from the disk
10. Correct the Page Table and other tables to show page is in memory
11. Wait for the CPU to be allocated to this process again
12. Restore the user registers, process state, and new page table, and then **resume** the interrupted instruction.

Performance of Demand Paging

- Three major activities:
  - **Service** the **page-fault** interrupt
    - careful coding means just ~100 instruction cycles (~100 ns) needed
  - **Read** in the page (from disk)
    - lots of time, e.g., ~8 milliseconds (~8,000,000 ns)

    Dominant Factor
  - **Restart** the process
    - again, just a small amount of time (~100 ns)
- **Page Fault Rate** p $(0 \leq p \leq 1)$
  - $p = 0 \Rightarrow$ no page faults
  - $p = 1 \Rightarrow$ every reference causes a page fault.
- **Effective Access Time (EAT):**
  - **EAT** = $\underbrace{(1-p) \times \texttt{memory\_access\_time}}_{\text{Time spent for normal memory access}} + \underbrace{p \times \texttt{page\_fault\_time}}_{\text{Time spent for handling page fault}}$

Performance of Demand Paging

- *Example*:
  - `memory_access_time` = 200 nanoseconds
  - Average `page_fault_time` = 8 milliseconds = 8,000,000 ns
  - **EAT** `= (1-p) x 200 (ns) + p x 8 (ms)`
    `= (1-p) x 200 + p x 8,000,000 (ns)`
    `= 200 + 7,999,800 x p (ns)`
  - **EAT** is directly proportional to the page-fault rate p.
- If one access out of 1,000 causes a page fault ($p = 0.001$), then
  - `EAT = 200 + 7,999,800 x 0.001 = 8199.8 (ns)`
    - $\Rightarrow$ a slowdown by a factor of
      `8199.8 / 200 = 41`
- If we want performance degradation < 10%
  - `EAT = 220 < 200 + 7,999,800 x p`

  $\Rightarrow$ `20 > 7,999,800 x p`
  - `p < 2.5 x 10`$^{-6}$.
    - i.e., less than one page fault in every 400,000 memory accesses.

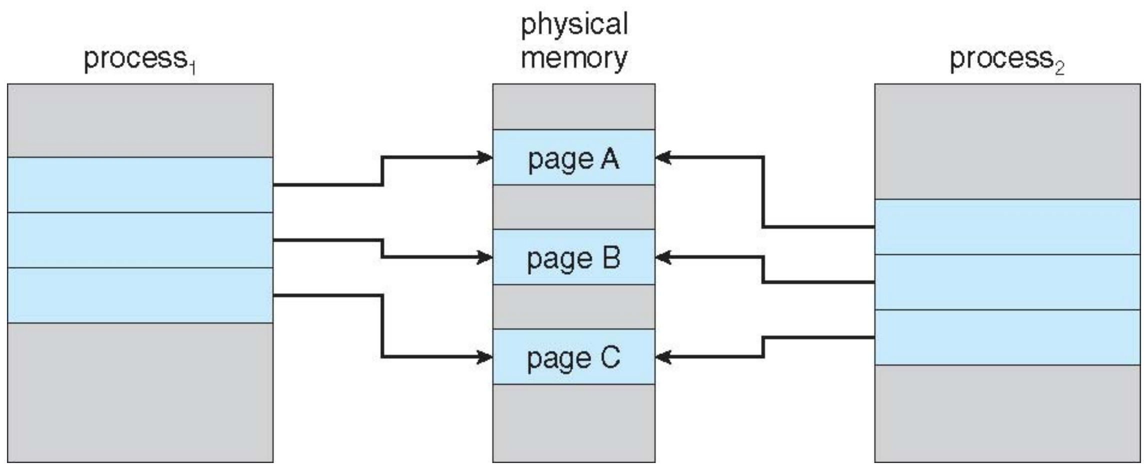
- **Demand Paging Optimizations**
  - Swap space I/O faster than file system I/O even if on the same device.
    - Swap allocated in large chunks, less management needed than FS
  - Copy entire process image to swap space at process load time
    - Then page in and out of swap space
    - Used in older BSD Unix
  - Demand page in from program binary on disk, but discard rather than paging out when freeing frame
    - Used in Solaris and current BSD
    - Still need to write to swap space
      - Pages not associated with a file (like stack and heap)
        - anonymous memory
      - Pages modified in memory but not yet written back to file system
  - Mobile systems
    - Typically don't support swapping
    - Instead, demand page from file system and reclaim read-only pages (such as code)

Copy-on-Write (CoW)

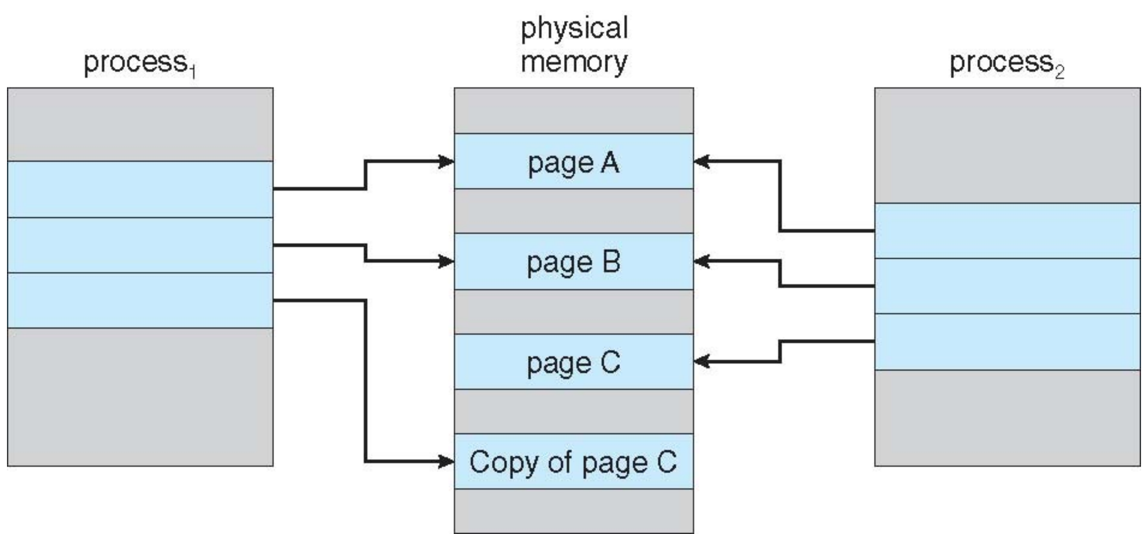
- Semantically, `fork()` syscall will create a copy of the **parent**'s address space for the **child** process
  - In early UNIX systems, it will simply **duplicate** all the active pages belonging to the parent for the child process.
  - Very expensive (huge amount of memory accesses)
- Copy-on-Write (CoW, **写入时复制**) allows both parent and child processes to initially ***share*** the same pages in memory
  - If either process modifies a shared page, only then is the page copied.
- **CoW** allows more efficient process creation
  - only modified pages are copied.

Copy-on-Write (CoW)

- **Before** Process 1 Modifies **Page C:**



- **After** Process 1 Modifies **Page C:**



Copy-on-Write (CoW)

```
/* CoW.c */
#define PAGE_SHIFT 12
#define PAGE_SIZE (1 << PAGE_SHIFT)
#define PAGEMAP_ENTRY 8

uint64_t get_physical_address(uint64_t virtual_address);

int main() {
    int *addr = malloc(PAGE_SIZE); // Allocate space (PAGE_SIZE) in the heap
    *addr = 10; // Initialize addr with some value

    pid_t rc = fork();
    if (rc == 0) {          /* Child process */
        printf(" Child: virtual  addr before child modification: %d (%p)\n", *addr, addr);
        printf(" Child: physical addr before child modification: %d (%p)\n", *addr, (void
*)get_physical_address((uint64_t)addr));
        *addr = 20;         /* Modify *addr */
        printf(" Child: virtual  addr  after child modification: %d (%p)\n", *addr, addr);
        printf(" Child: physical addr  after child modification: %d (%p)\n", *addr, (void
*)get_physical_address((uint64_t)addr));
    } else {                /* Parent process */
        printf("Parent: virtual  addr before child modification: %d (%p)\n", *addr, addr);
        printf("Parent: physical addr before child modification: %d (%p)\n", *addr, (void
*)get_physical_address((uint64_t)addr));
        sleep(1);           /* Wait for child process to modify *addr */
        printf("Parent: virtual  addr  after child modification: %d (%p)\n", *addr, addr);
        printf("Parent: physical addr  after child modification: %d (%p)\n", *addr, (void
*)get_physical_address((uint64_t)addr));
        return 0;
    }
}
```

为addr分配一个大小为PAGE_SIZE(4096字节)的空间，以确保addr单独占有一个page

Copy-on-Write (CoW)

```
/* CoW.c */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <fcntl.h>
#include <stdint.h>
#include <sys/mman.h>

#define PAGE_SHIFT 12
#define PAGE_SIZE (1 << PAGE_SHIFT)
#define PAGEMAP_ENTRY 8

uint64_t get_physical_address(uint64_t
virtual_address) {
    int pagemap_fd;
    uint64_t offset, physical_address;
    uint64_t pagemap_entry;

    // Open the pagemap file for the current process
    pagemap_fd = open("/proc/self/pagemap",
O_RDONLY);

    // Calculate the offset in the pagemap file
    offset = (virtual_address / PAGE_SIZE) *
PAGEMAP_ENTRY;
```

```
    // Seek to the appropriate offset
    if (lseek(pagemap_fd, offset, SEEK_SET) ==
(off_t)-1) {
        perror("lseek");
        close(pagemap_fd);
        return -1;
    }

    // Read the pagemap entry
    if (read(pagemap_fd, &pagemap_entry,
PAGEMAP_ENTRY) != PAGEMAP_ENTRY) {
        perror("read");
        close(pagemap_fd);
        return -1;
    }

    close(pagemap_fd);

    // Extract the physical frame number (PFN) from
the pagemap entry
    if (pagemap_entry & (1ULL << 63)) {
        physical_address = (pagemap_entry & ((1ULL
<< 55) -1)) *PAGE_SIZE;
        physical_address |= (virtual_address &
(PAGE_SIZE - 1));
    } else {
        return -1;
    }
    return physical_address;
}
```

Copy-on-Write (CoW)

```
/* CoW.c */
#define PAGE_SHIFT 12
#define PAGE_SIZE (1 << PA
#define PAGEMAP_ENTRY 8

uint64_t get_physical_addr

int main() {
    int *addr = malloc(PAGE_SIZE); // Allocate space (PAGE_SIZE) in the heap
    *addr = 10; // Initialize addr with some value

    pid_t rc = fork();
    if (rc == 0) {          /* Child process */
        printf(" Child: virtual  addr before child modification: %d (%p)\n", *addr, addr);
        printf(" Child: physical addr before child modification: %d (%p)\n", *addr, (void
*)get_physical_address((uint64_t)addr));
        *addr = 20;         /* Modify *addr */
        printf(" Child: virtual  addr  after child modification: %d (%p)\n", *addr, addr);
        printf(" Child: physical addr  after child modification: %d (%p)\n", *addr, (void
*)get_physical_address((uint64_t)addr));
    } else {                /* Parent process */
        printf("Parent: virtual  addr before child modification: %d (%p)\n", *addr, addr);
        printf("Parent: physical addr before child modification: %d (%p)\n", *addr, (void
*)get_physical_address((uint64_t)addr));
        sleep(1);           /* Wait for child process to modify *addr */
        printf("Parent: virtual  addr  after child modification: %d (%p)\n", *addr, addr);
        printf("Parent: physical addr  after child modification: %d (%p)\n", *addr, (void
*)get_physical_address((uint64_t)addr));
        return 0;
    }
}
```

```
$ gcc -g -Wall -o CoW CoW.c
$ sudo ./CoW
Parent: virtual  addr before child modification: 10 (0x5555555592a0)
Parent: physical addr before child modification: 10 (0x6952822a0)
 Child: virtual  addr before child modification: 10 (0x5555555592a0)
 Child: physical addr before child modification: 10 (0x6952822a0)
```

Copy-on-Write (CoW)

```
/* CoW.c */
#define PAGE_SHIFT 12
#define PAGE_SIZE (1 << PA
#define PAGEMAP_ENTRY 8

uint64_t get_physical_addr

int main() {
    int *addr = malloc(PAG
    *addr = 10; // Initialize addr with some value

    pid_t rc = fork();
    if (rc == 0) {          /* Child process */
        printf(" Child: virtual  addr before child modification: %d (%p)\n", *addr, addr);
        printf(" Child: physical addr before child modification: %d (%p)\n", *addr, (void
*)get_physical_address((uint64_t)addr));
        *addr = 20;         /* Modify *addr */
        printf(" Child: virtual  addr  after child modification: %d (%p)\n", *addr, addr);
        printf(" Child: physical addr  after child modification: %d (%p)\n", *addr, (void
*)get_physical_address((uint64_t)addr));
    } else {                /* Parent process */
        printf("Parent: virtual  addr before child modification: %d (%p)\n", *addr, addr);
        printf("Parent: physical addr before child modification: %d (%p)\n", *addr, (void
*)get_physical_address((uint64_t)addr));
        sleep(1);           /* Wait for child process to modify *addr */
        printf("Parent: virtual  addr  after child modification: %d (%p)\n", *addr, addr);
        printf("Parent: physical addr  after child modification: %d (%p)\n", *addr, (void
*)get_physical_address((uint64_t)addr));
        return 0;
    }
}
```

```
$ gcc -g -Wall -o CoW CoW.c
$ sudo ./CoW
Parent: virtual  addr before child modification: 10 (0x5555555592a0)
Parent: physical addr before child modification: 10 (0x6952822a0)
 Child: virtual  addr before child modification: 10 (0x5555555592a0)
 Child: physical addr before child modification: 10 (0x6952822a0)
 Child: virtual  addr  after child modification: 20 (0x5555555592a0)
 Child: physical addr  after child modification: 20 (0xb1b4a02a0)
```

Copy-on-Write (CoW)

```
/* CoW.c */
#define PAGE_SHIFT 12
#define PAGE_SIZE (1 << PA
#define PAGEMAP_ENTRY 8

uint64_t get_physical_addr

int main() {
    int *addr = malloc(PAG
    *addr = 10; // Initial

    pid_t rc = fork();
    if (rc == 0) {          /* Child process */
        printf(" Child: virtual  addr before child modification: %d (%p)\n", *addr, addr);
        printf(" Child: physical addr before child modification: %d (%p)\n", *addr, (void *)get_physical_address((uint64_t)addr));
        *addr = 20;         /* Modify *addr */
        printf(" Child: virtual  addr  after child modification: %d (%p)\n", *addr, addr);
        printf(" Child: physical addr  after child modification: %d (%p)\n", *addr, (void *)get_physical_address((uint64_t)addr));
    } else {                /* Parent process */
        printf("Parent: virtual  addr before child modification: %d (%p)\n", *addr, addr);
        printf("Parent: physical addr before child modification: %d (%p)\n", *addr, (void *)get_physical_address((uint64_t)addr));
        sleep(1);           /* Wait for child process to modify *addr */
        printf("Parent: virtual  addr  after child modification: %d (%p)\n", *addr, addr);
        printf("Parent: physical addr  after child modification: %d (%p)\n", *addr, (void *)get_physical_address((uint64_t)addr));
        return 0;
    }
}
```

```
$ gcc -g -Wall -o CoW CoW.c
$ sudo ./CoW
Parent: virtual  addr before child modification: 10 (0x5555555592a0)
Parent: physical addr before child modification: 10 (0x6952822a0)
 Child: virtual  addr before child modification: 10 (0x5555555592a0)
 Child: physical addr before child modification: 10 (0x6952822a0)
 Child: virtual  addr  after child modification: 20 (0x5555555592a0)
 Child: physical addr  after child modification: 20 (0xb1b4a02a0)
Parent: virtual  addr  after child modification: 10 (0x5555555592a0)
Parent: physical addr  after child modification: 10 (0x6952822a0)
```

Copy-on-Write

- **fork() vs. vfork()**
  - `vfork()` is a variant of `fork()` in several versions of UNIX (including Linux, macOS, and BSD UNIX).
    - `vfork()` is deprecated and not part of the POSIX standard
      - not recommended if you wish to write portable code.
  - Unlike `fork()`, `vfork()` does not use **Copy-on-Write.**
  - With `vfork()`, the **parent** process is suspended, and the **child** process uses the **same** address space of the **parent.**
    - The child process is intended to call `exec()` **immediately** after creation.
      - Recall that `exec()` will **replace** the current child process image with a new process image specified by the arguments of `exec()`
        - ⇒ **new address space** that is different from the parent.
    - If the child process modifies the address space of the parent, it will lead to undefined behavior.
    - The parent process **resumes** after the child process either calls `exec()` or exits.

Copy-on-Write

fork() vs. vfork()

```c
/* ex_fork.c */

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>

int main() {
    int data = 10;
    pid_t rc = fork();
    if (rc == 0) {    /* Child */
        data = 20;
        printf(" (Child) data: %d\n", data);
        _exit(0);
    } else {          /* Parent */
        wait(NULL);
        printf("(Parent) data: %d\n", data);
    }
}
```

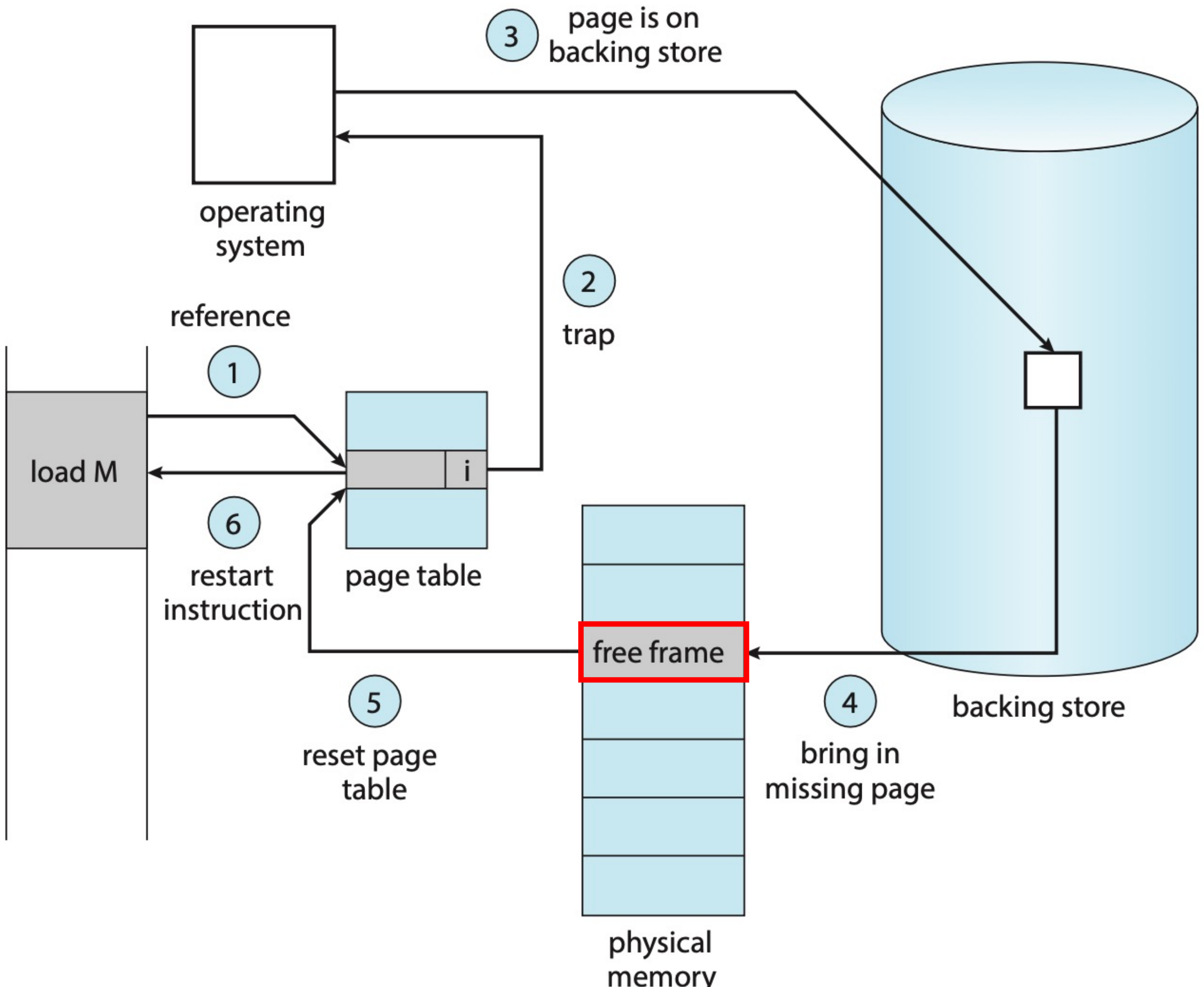
```c
/* ex_vfork.c */
#define _XOPEN_SOURCE 500
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>

int main() {
    int data = 10;
    pid_t rc = vfork();
    if (rc == 0) {    /* Child */
        data = 20;
        printf(" (Child) data: %d\n", data);
        _exit(0);
    } else {          /* Parent */
        wait(NULL);
        printf("(Parent) data: %d\n", data);
    }
}
```

```
$ ./ex_fork
 (Child) data: 20
(Parent) data: 10
```

```
$ ./ex_vfork
 (Child) data: 20
(Parent) data: 20
```

Demand Paging

- What happens if there is **no free frame**?

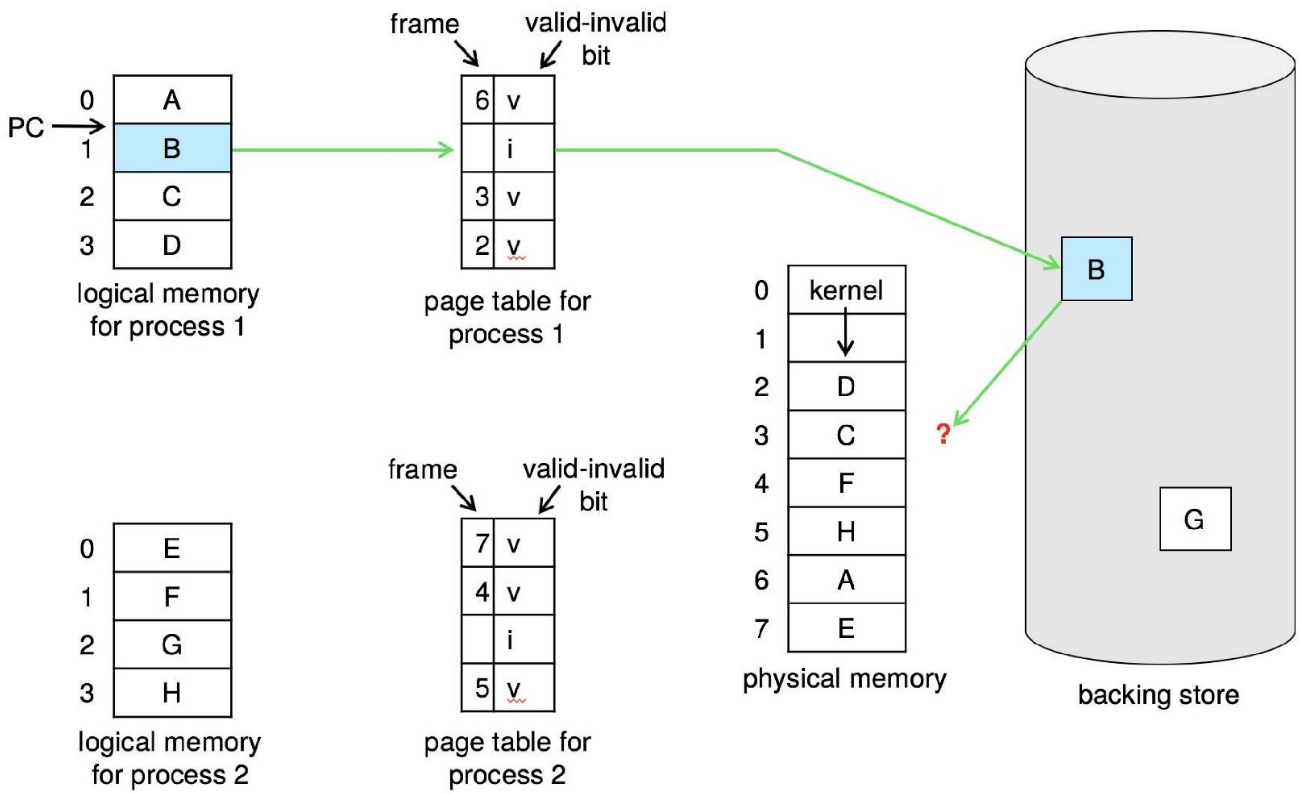


Demand Paging

- **What can the OS do when there is no free frame?**
  - Option **#1**: Terminate the user process
    - ⇒ Not the best choice, as it destroys the purpose of Demand Paging
  - Option **#2**: The OS could swap out a process (the entire process), **freeing all its frames** and reducing the level of multiprogramming
    - ⇒ A good option under certain circumstances
  - Option **#3**: Make use of **Page Replacement** technique.
    - ⇒ The most **common** solution.

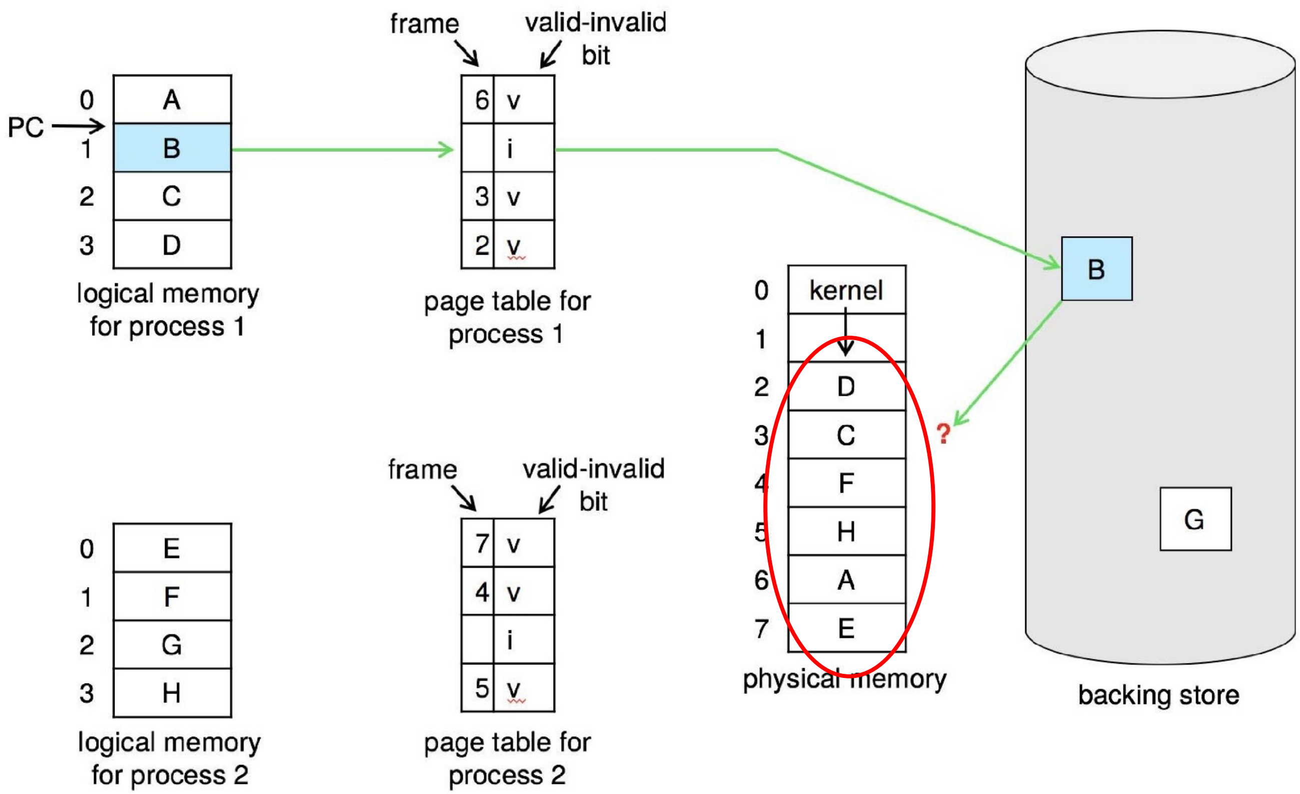
Page Replacement

- What happens if there is **no free frame**?
  - ⇒ Page Replacement (页面置换)
  - Find some frame (victim frame) in memory that are *not really in use*
    - and then **swap** it out.



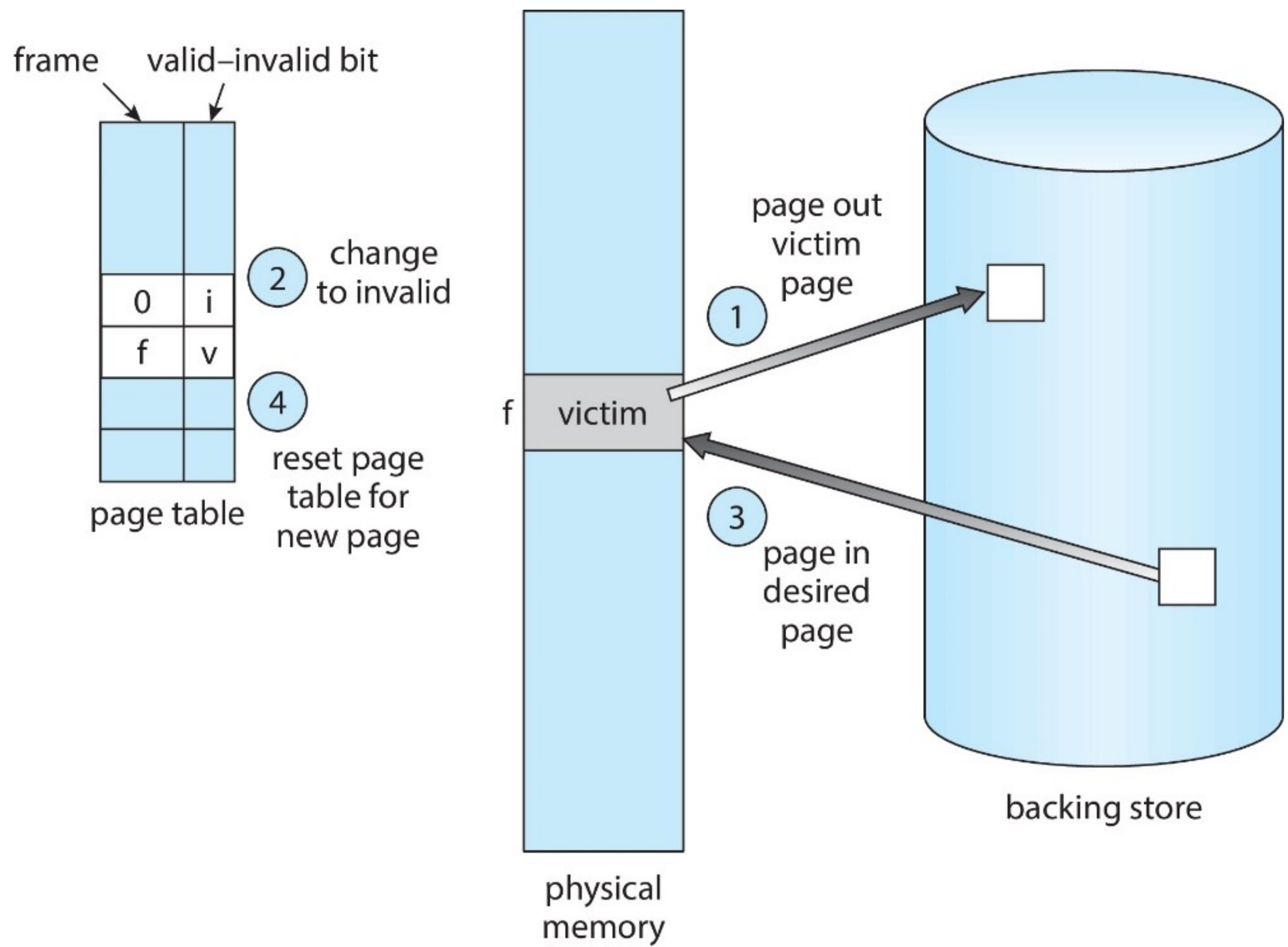
Page Replacement

- What happens if there is **no free frame**?
  - ⇒ Page Replacement (页面置换)
  - The **Question** is: How to choose the victim frame?
    - ⇒ **Page Replacement Algorithms** (*more on this in the next lecture*)



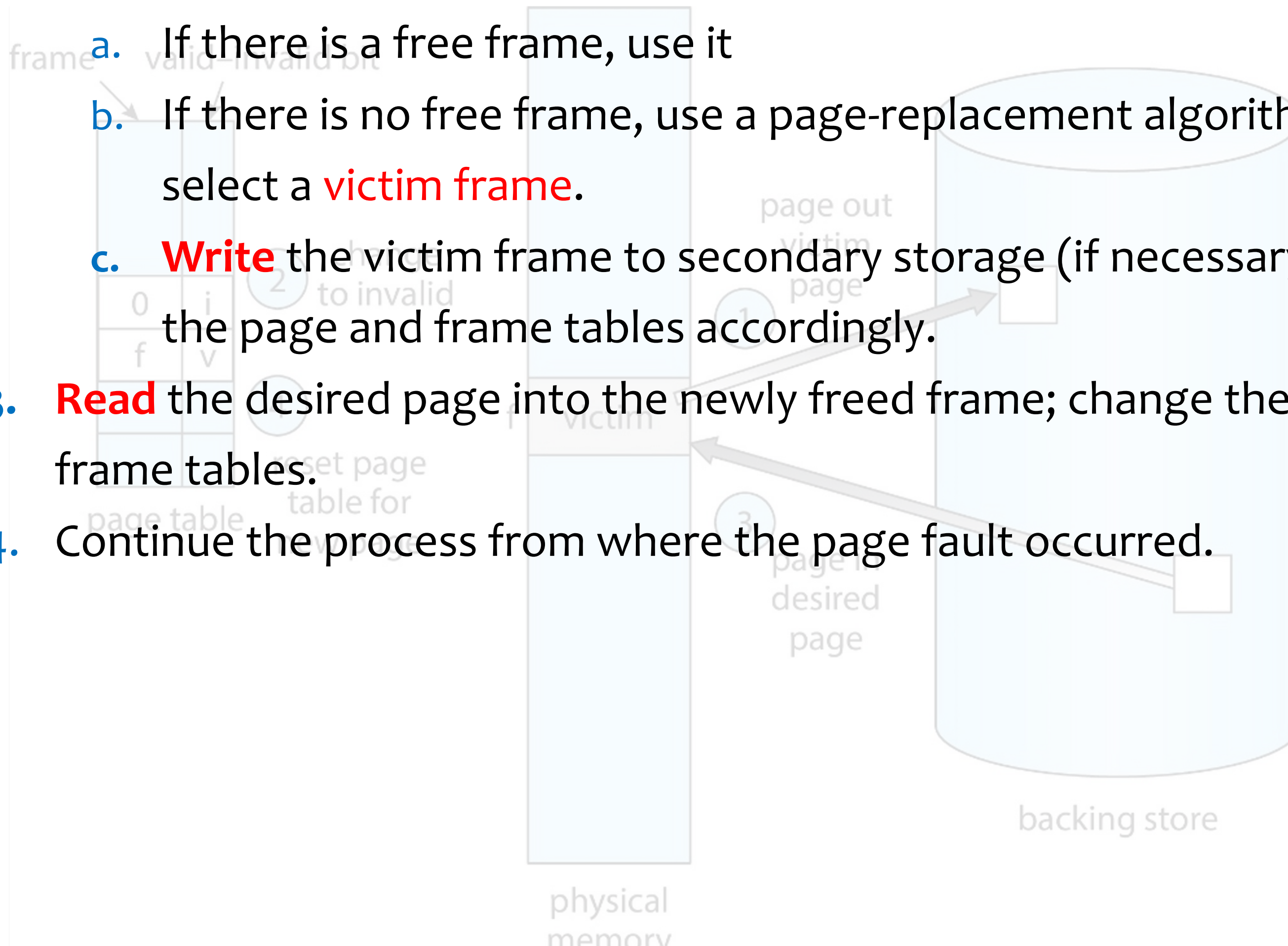
Page Replacement

- Steps in handling a **Page Replacement**.



Page Replacement

- Steps in handling a **Page Replacement**.
  1. Find the location of the desired page/block on disk.
  2. Find a free frame:
     a. If there is a free frame, use it
     b. If there is no free frame, use a page-replacement algorithm to select a victim frame.
     c. **Write** the victim frame to secondary storage (if necessary); change the page and frame tables accordingly.
  3. **Read** the desired page into the newly freed frame; change the page and frame tables.
  4. Continue the process from where the page fault occurred.



Page Replacement

- Steps in handling a **Page Replacement**.
  1. Find the location of the desired page/block on disk.
  2. Find a free frame:
     a. If there is a free frame, use it
     b. If there is no free frame, use a page-replacement algorithm to select a victim frame.
     c. **Write** the victim frame to secondary storage (if necessary); change the page and frame tables accordingly.
  3. **Read** the desired page into the newly freed frame; change the page and frame tables.
  4. Continue the process from where the page fault occurred.

Notice that, if no frames are free, **two** page transfers (one for the page-out and one for the page-in) are required. This situation effectively **doubles** the page-fault service time and **increases** the effective access time accordingly.

Page Replacement

The "Dirty" Bit

- We can **reduce** this overhead (of page-out) by using a dirty bit (or modify bit) in the **PTE**.

Physical Frame Number	Free (OS)	0	PS	D	A	PCD	PWT	U	W	P
31-12	11-9	8	7	6	5	4	3	2	1	0

- The dirty bit for a page is set by the hardware whenever any byte in the page is written into, indicating that the page has been modified.
- Thus, when we select a victim page for replacement, we examine its dirty bit:
  - If the dirty bit is not set (`0`), it means this page has not been modified, thus we do not need to write the memory page (frame) into disk
  - If the dirty bit is set (1), which means that this page has been modified and we must write that page (frame) into disk.

Thank you!