Virtual Memory

Introduction

- Virtual memory != Virtual address

    Virtual address is address issued by CPU’s execution unit, later converted by MMU to physical address

    Virtual memory is a memory management technique employed by OS (with hardware support, of course)

Unused parts of program

```
int a[4096][4096]
int f(int m[][4096]) {
    int i, j;
    for(i = 0; i < 1024; i++)
        m[0][i] = 200;
}
int main() {
    int i, j;
    for(i = 0; i < 1024; i++)
        a[1][i] = 200;
    if(random() == 10)
        f(a);
}
```

All parts of array a[] not accessed

Function f() may not be called

Some problems with schemes discussed so far

- Code needs to be in memory to execute, But entire program rarely used

    Error code, unusual routines, large data structures are rarely used

- So, entire program code, data not needed at same time

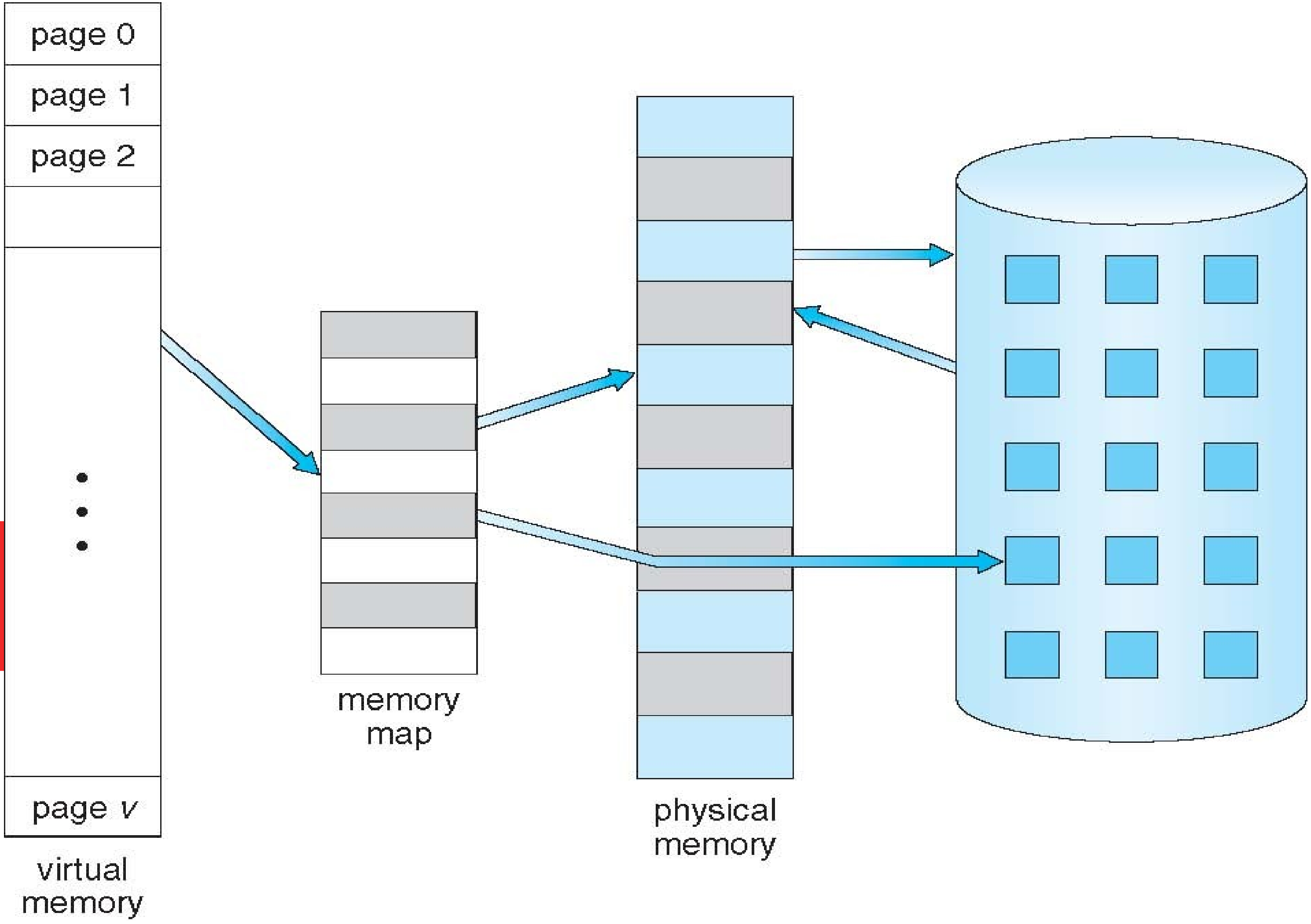
- So, consider ability to execute partially-loaded program

    One Program no longer constrained by limits of physical memory

    One Program and collection of programs could be larger than physical memory

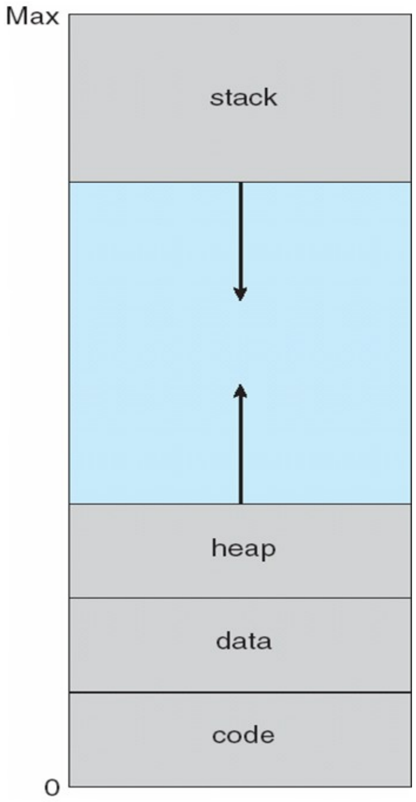
What is virtual memory?

- Virtual memory – separation of user 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 memory can be implemented via:
  - Demand paging
  - Demand segmentation

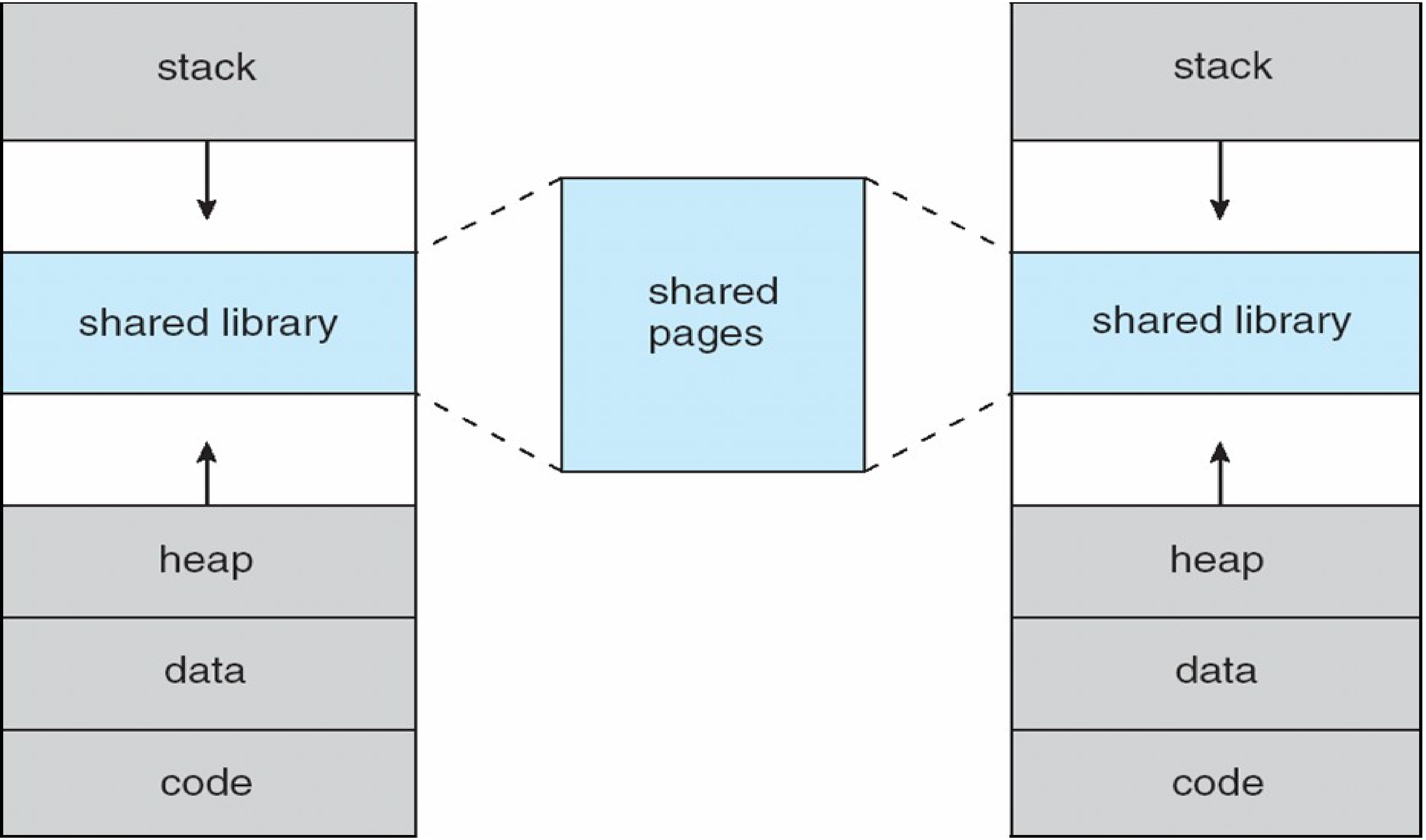


Virtual Memory Larger Than Physical Memory

Virtual Address space



Enables **sparse** address spaces with holes left for growth, dynamically linked libraries, etc



Shared pages Using Virtual Memory

System libraries shared via mapping into virtual address space

Shared memory by mapping same page-frames into page tables of involved processes

Pages can be shared during `fork()`, speeding process creation (more later)

Demand Paging

Demand Paging

- Load a “page” to memory when it’s neded (on demand)
    - Less I/O needed, no unnecessary I/O
    - Less memory needed
    - Faster response
    - More users
-

Demand Paging

- Options:
    - Load entire process at load time : achieves little
    - Load some pages at load time: good
    - Load no pages at load time: pure demand paging

New meaning for valid/invalid bits in page table

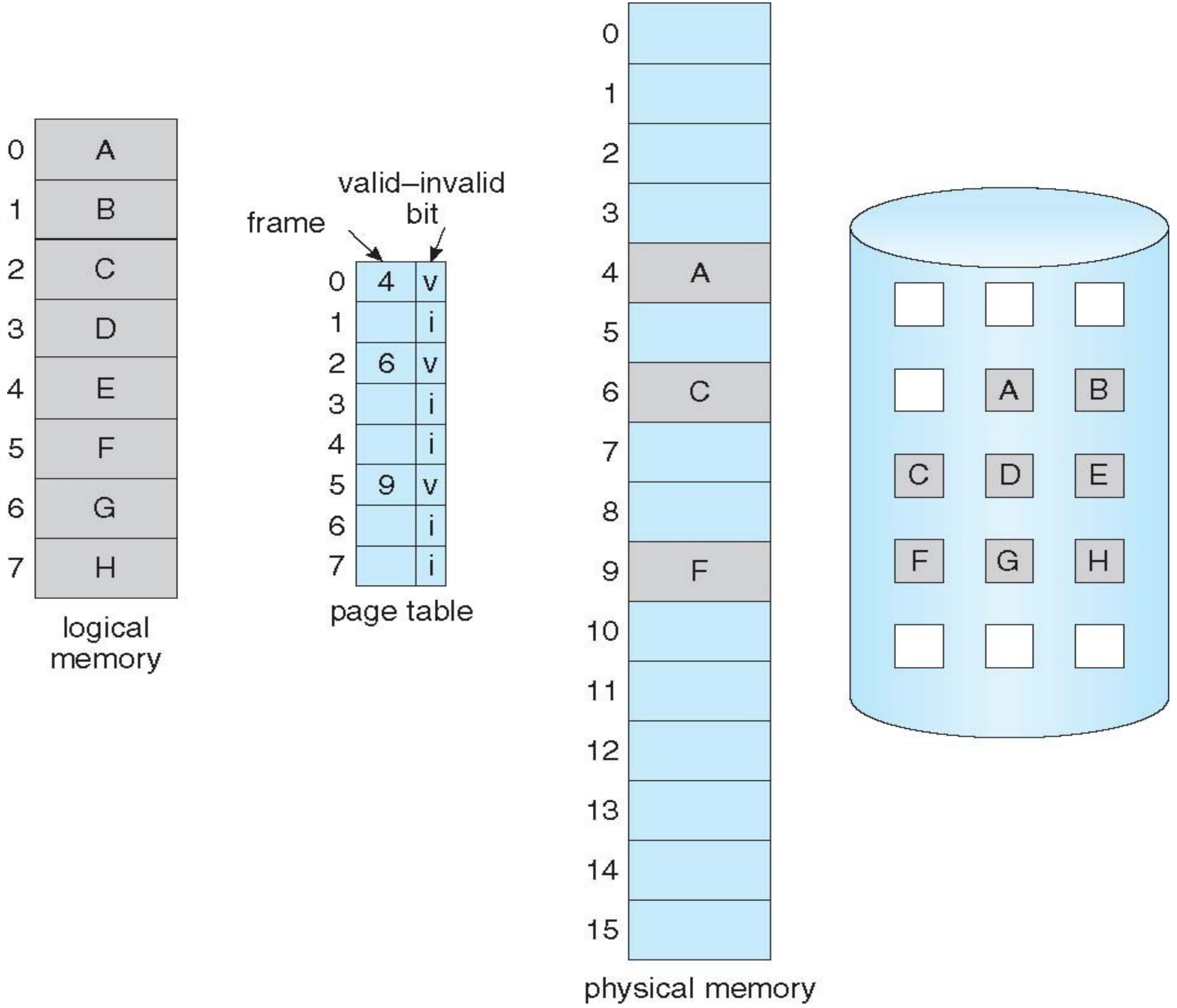
Frame #	valid-invalid bit
	v
	v
	v
	v
	i
….	
	i
	i

- With each page table entry a valid–invalid bit is associated

  v: in-memory – memory resident

  i : not-in-memory or illegal

- During address translation, if valid–invalid bit in page table entry is I : **raises trap called page fault**



Page

Table

With

Some pages

Not

In memory

Page fault

Page fault

- Page fault is a hardware interrupt
- It occurs when the page table entry corresponding to current memory access is “i”
- All actions that a kernel takes on a hardware interrupt are taken!

    Change of stack to kernel stack

    Saving the context of process

    Switching to kernel code

On a Page fault

1) Operating system looks at another data structure (table), most likely in PCB itself, to decide:

    If it’s Invalid reference -> abort the process (segfault)

    Just not in memory -> Need to get the page in memory

2) Get empty frame (this may be complicated!)
3) Swap page into frame via scheduled disk/IO operation
4) Reset tables to indicate page now in memory.
5) Set validation bit = v
6) Restart the instruction that caused the page fault

Additional problems

- Extreme case – start process with *no* pages in memory
  - OS sets instruction pointer to first instruction of process, non-memory-resident -> page fault
  - And for every other process pages on first access
  - **Pure demand paging**
- Actually, a given instruction could access multiple pages -> multiple page faults
  - 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

Instruction restart

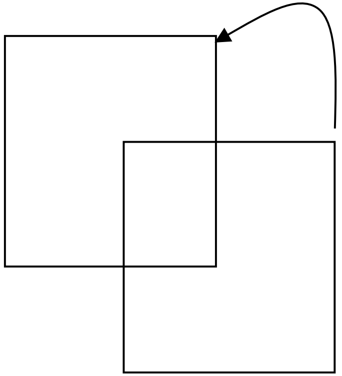
- A critical Problem
- Consider an instruction that could access several different locations

```
movarray 0x100, 0x200, 20
```

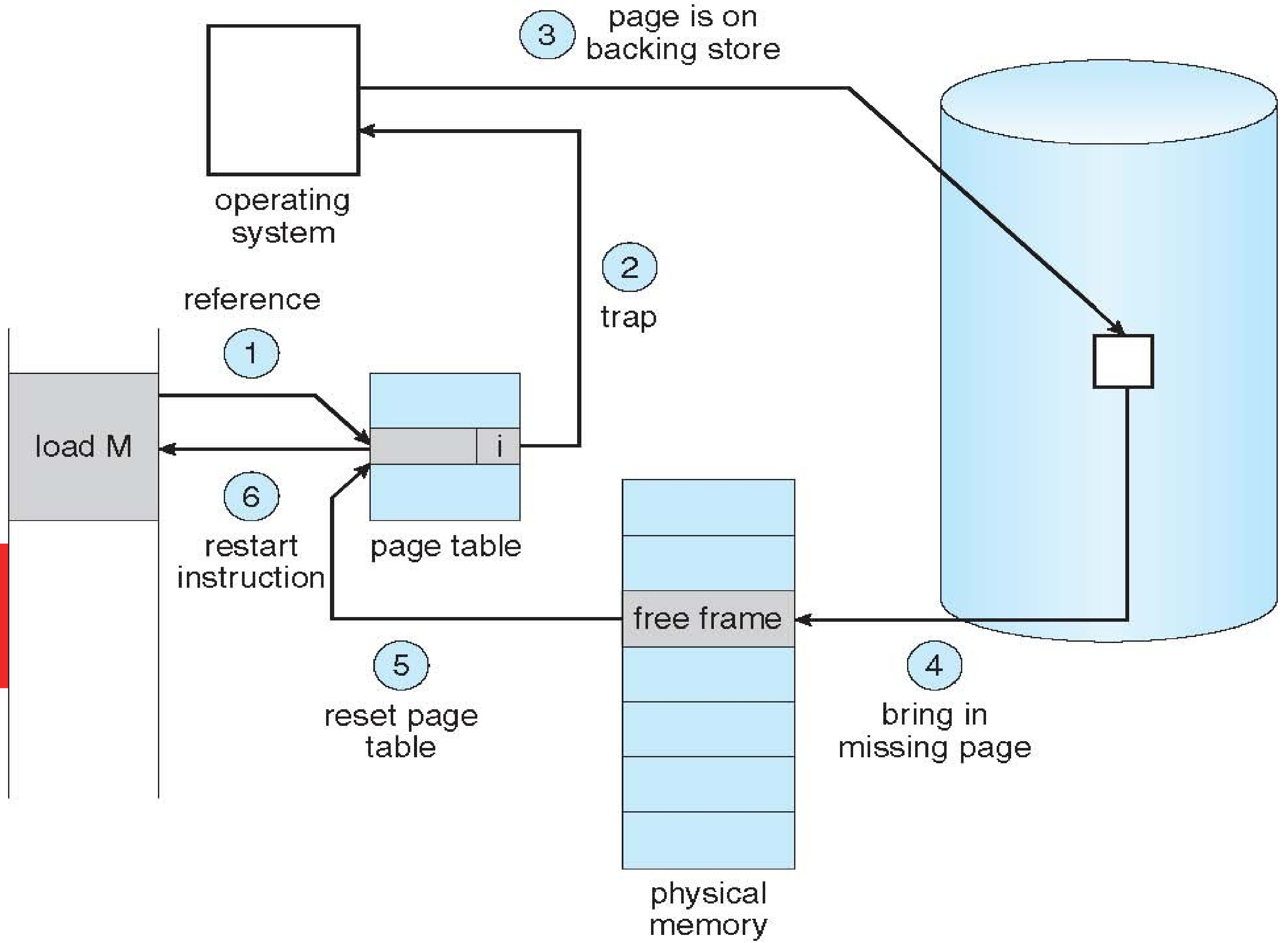
\# copy 20 bytes from address 0x100 to address 0x200

```
movarray 0x100, 0x110, 20
```

\# what to do in this case?



Handling A Page Fault



Page fault handling

1) Trap to the operating system
2) Default trap handling():

   Save the process registers and process state

   Determine that the interrupt was a page fault. Run page fault handler.

3) Page fault handler(): Check that the page reference was legal and determine the location of the page on the disk. If illegal, terminate process.
4) Find a free frame. Issue a read from the disk to a free frame:

   Process waits in a queue for disk read. Meanwhile many processes may get scheduled.

   Disk DMA hardware transfers data to the free frame and raises interrupt in end

Page fault handling

6) (as said on last slide) While waiting, allocate the CPU to some other process
7) (as said on last slide) Receive an interrupt from the disk I/O subsystem (I/O completed)
8) Default interrupt handling():
    - Save the registers and process state for the other user
    - Determine that the interrupt was from the disk
9) Disk interrupt handler():
    - Figure out that the interrupt was for our waiting process
    - Make the process runnable
10) Wait for the CPU to be allocated to this process again
    - Kernel restores the page table of the process, marks entry as “v”
    - Restore the user registers, process state, and new page table, and then resume the interrupted instruction

Performance of demand paging

Page Fault Rate 0 <= p <= 1

if p = 0 no page faults

if p = 1, every reference is a fault

Effective (memory) Access Time (EAT)

EAT = (1 – p) * memory access time +

p * (page fault overhead // Kernel code execution time

+ swap page out // time to write an occupied frame to disk

+ swap page in // time to read data from disk into free frame

+ restart overhead) // time to reset process context, restart it

Performance of demand paging

Memory access time = 200 nanoseconds

Average page-fault service time = 8 milliseconds
EAT = (1 – p) x 200 + p (8 milliseconds)

= (1 – p x 200 + p x 8,000,000

= 200 + p x 7,999,800

If one access out of 1,000 causes a page fault, then

EAT = 8.2 microseconds.

This is a slowdown by a factor of 40!!

If want performance degradation < 10 percent

220 > 200 + 7,999,800 x p
20 > 7,999,800 x p

p < .0000025

< one page fault in every 400,000 memory accesses

An optimization: Copy on write

The problem with fork() and exec(). Consider the case of a shell

```
scanf("%s", cmd);
if(strcmp(cmd, "exit") == 0)
    return 0;
pid = fork(); // A->B
if(pid == 0) {
    ret = execl(cmd, cmd, NULL);
    if(ret == -1) {
        perror("execution failed");
        exit(errno);
    }
} else {
    wait(0);
}
```

- During fork()
  - Pages of parent were duplicated
  - Equal amount of page frames were allocated
  - Page table for child differed from parent (as it has another set of frames)
- In exec()
  - The page frames of child were taken away and new frames were allocated
  - Child’s page table was rebuilt!
- Waste of time during fork() if the exec() was to be called immediately

An optimization: Copy on write

- Copy-on-Write (COW) allows both parent and child processes to initially share the same pages in memory

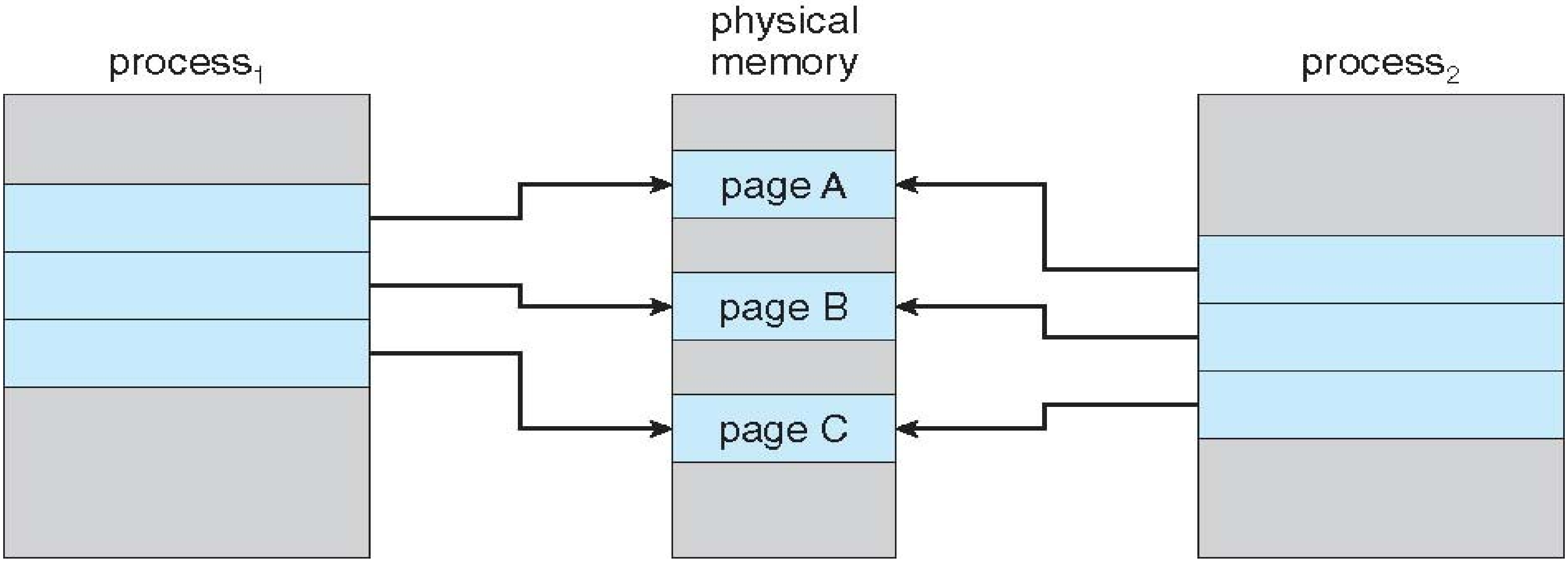
  If either process modifies a shared page, only then is the page copied

- COW allows more efficient process creation as only modified pages are copied

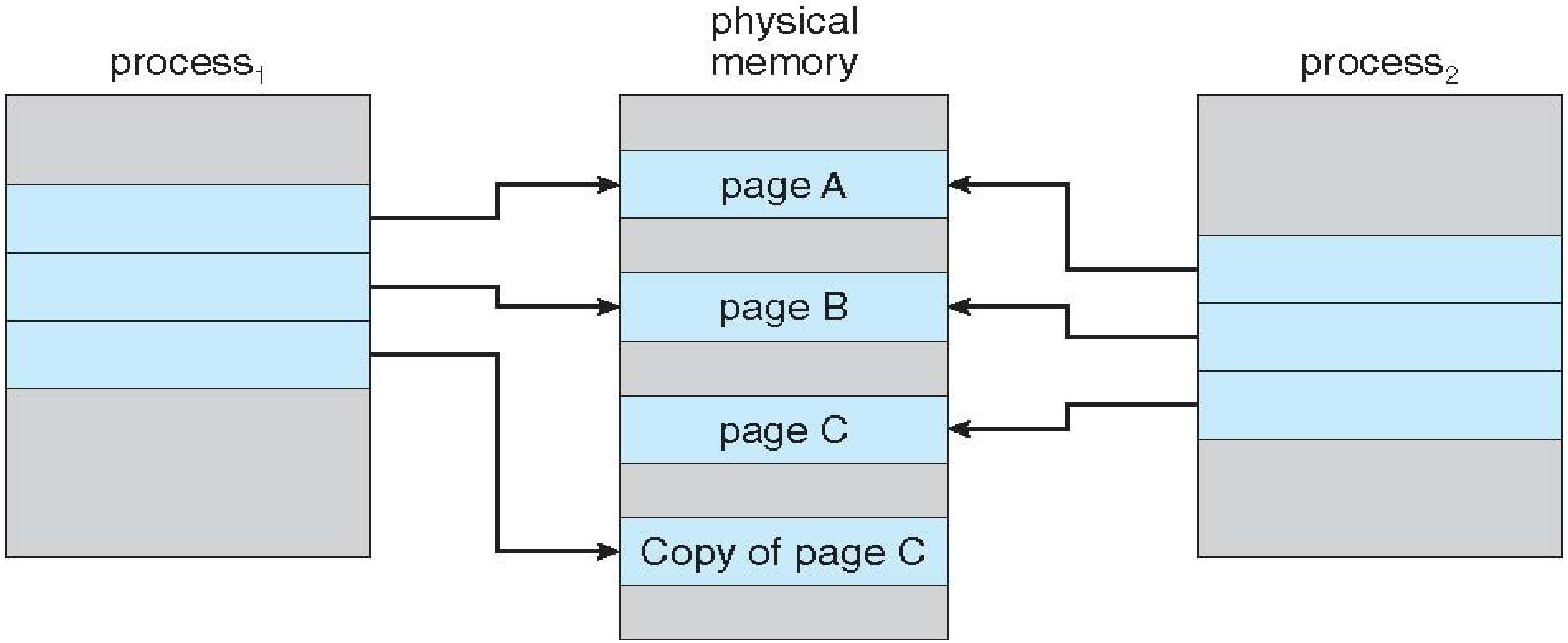
- vfork() variation on fork() system call has parent suspend and child using copy-on-write address space of parent

  Designed to have child call exec()

  Very efficient



Before Process 1 Modifies Page C



After Process 1 Modifies Page C

Challenges and improvements in implementation

- Choice of backing store

  For stack, heap pages: on swap partition

  For code, shared library? : swap partition or the actual executable file on the file-system?

  If the choice is file-system for code, then the page-fault handler needs to call functions related to file-system

- Is the page table itself pagable?

  If no, good

  If Yes, then there can be page faults in accessing the page tables themselves! More complicated!

- Is the kernel code pagable?

  If no, good

  If yes, life is very complicated ! Page fault in running kernel code, interrupt handlers, system calls, etc.