

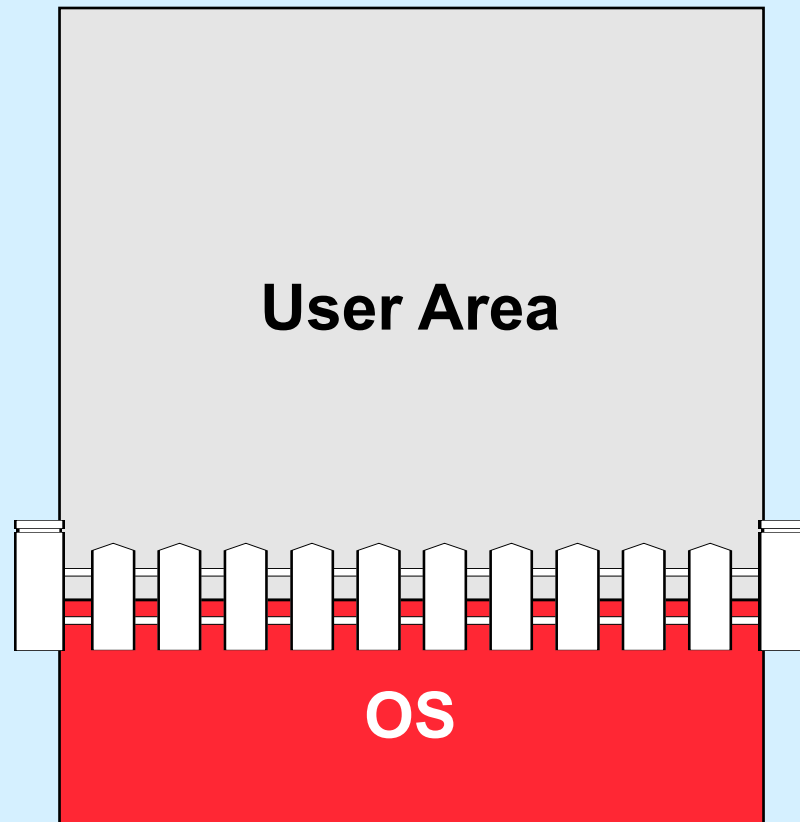
# CS 33

## Virtual Memory

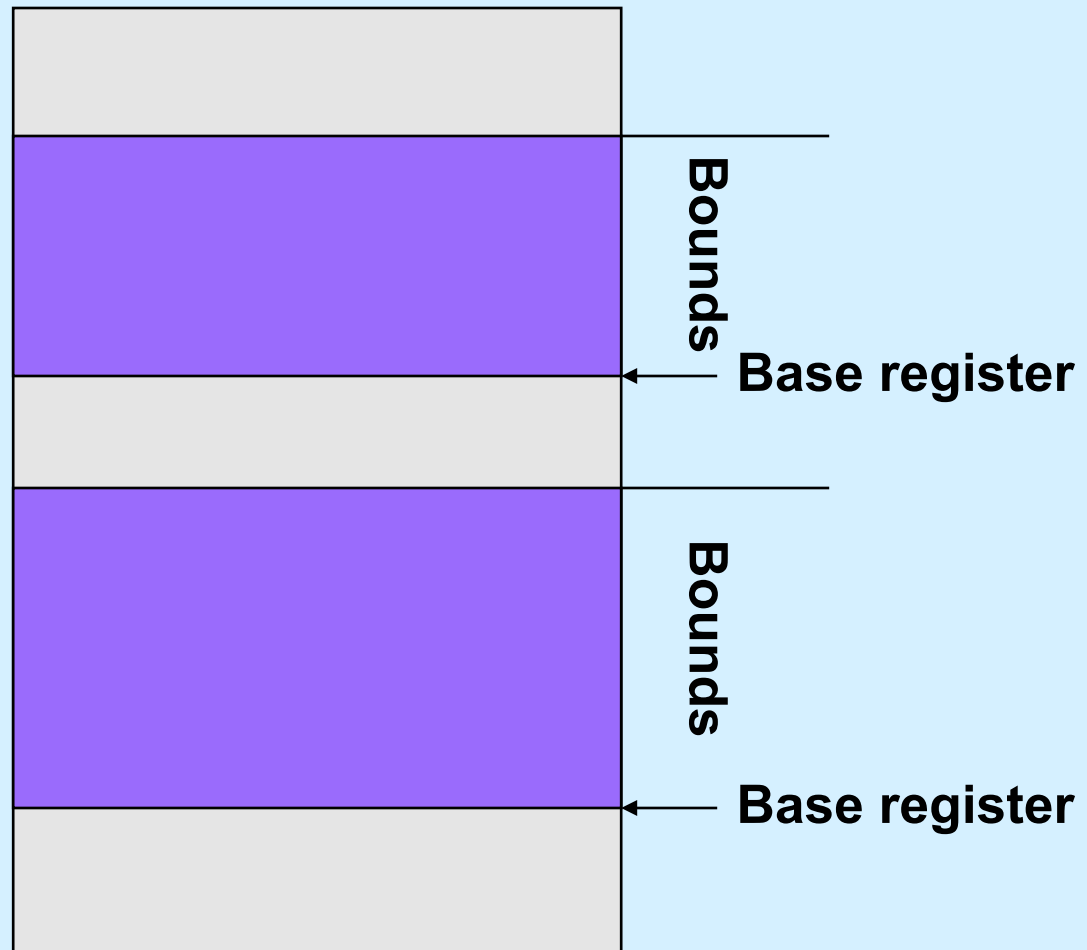
# The Address-Space Concept

- **Protect processes from one another**
- **Protect the OS from user processes**
- **Provide efficient management of available storage**

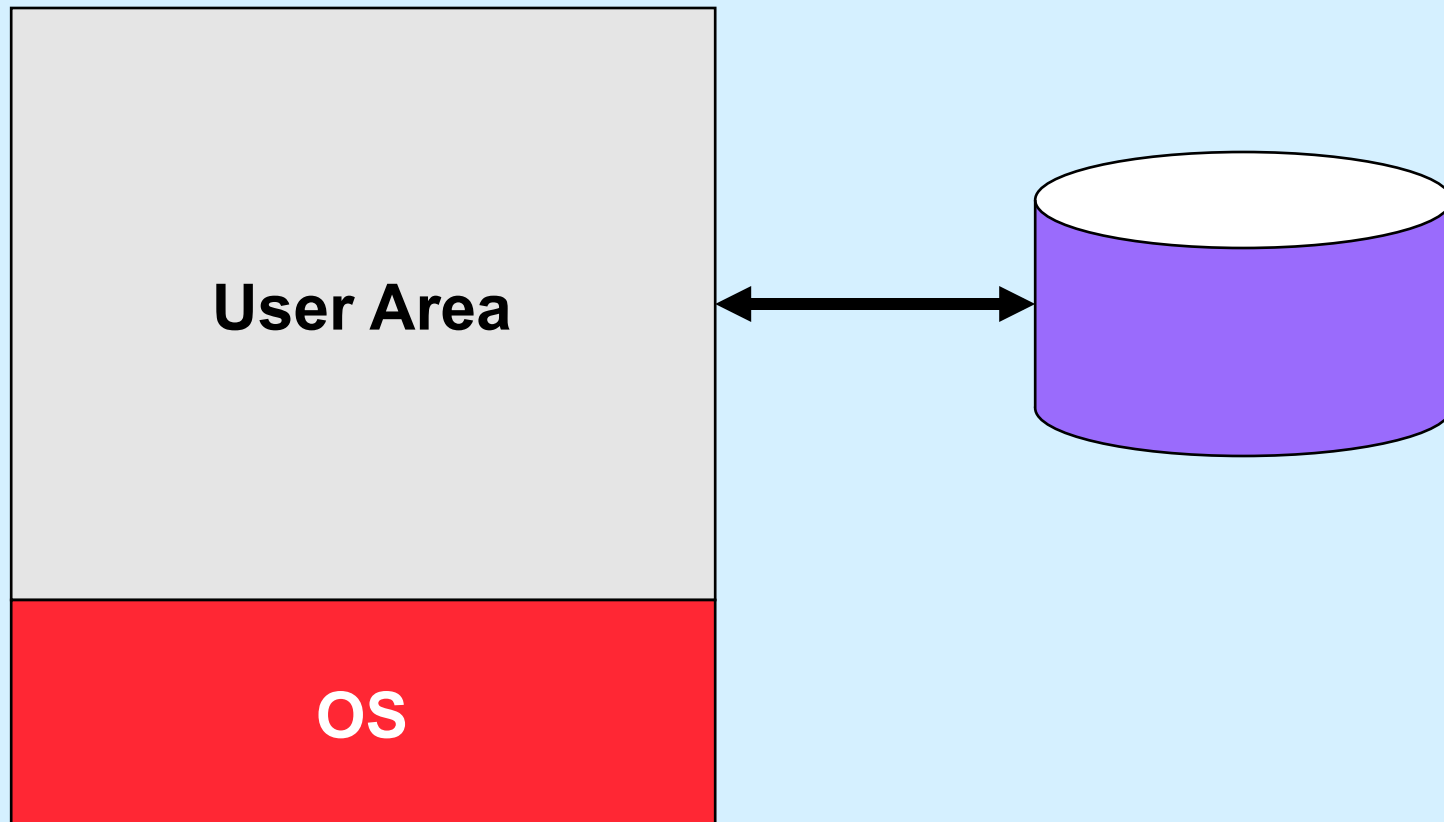
# Memory Fence



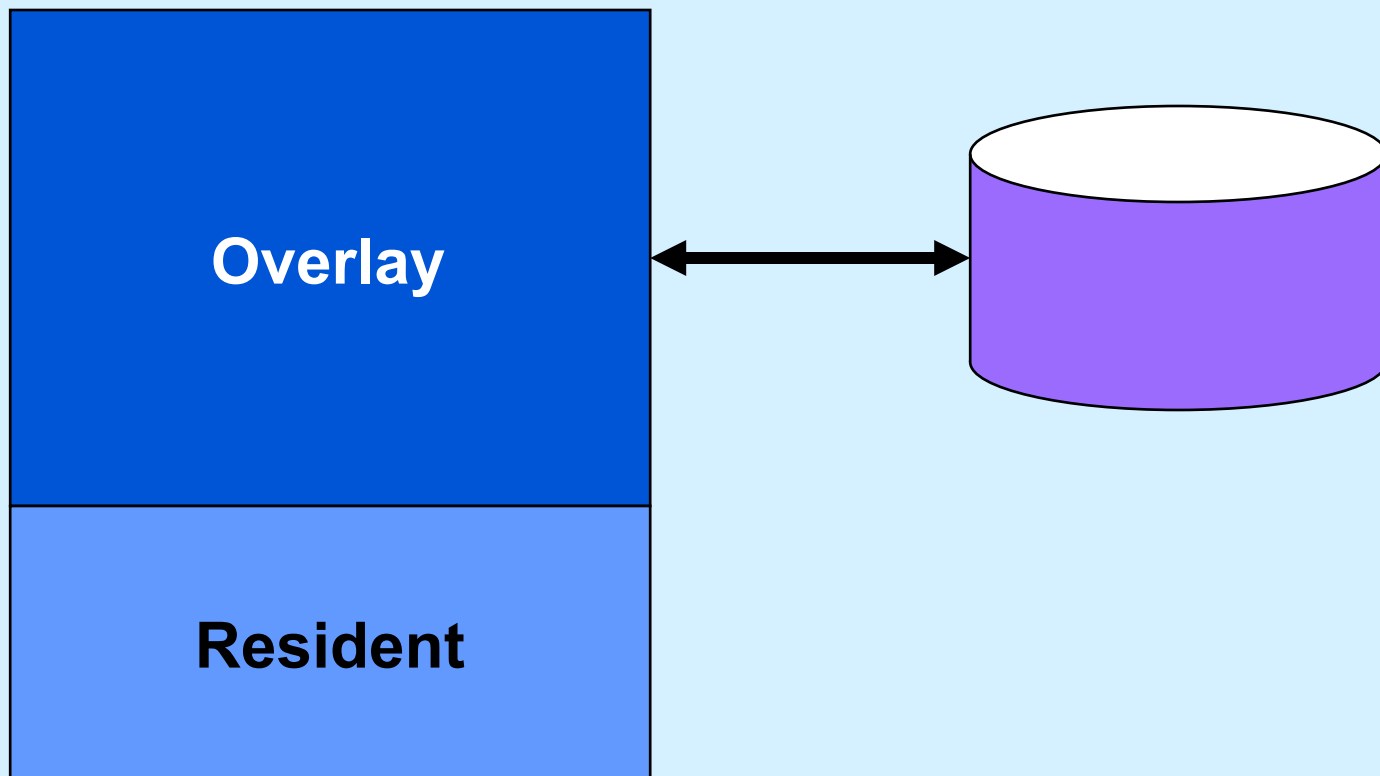
# Base and Bounds Registers



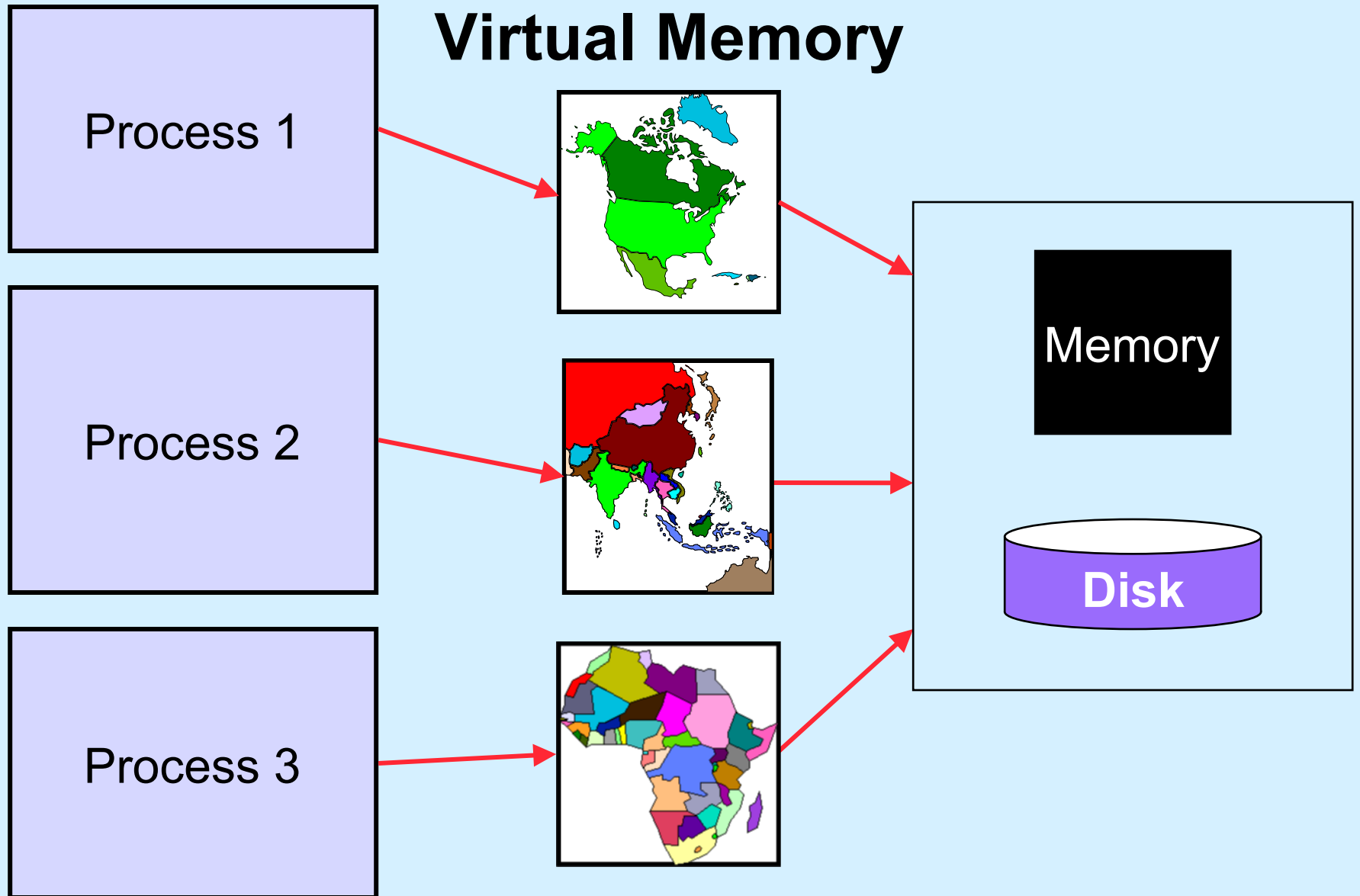
# Swapping



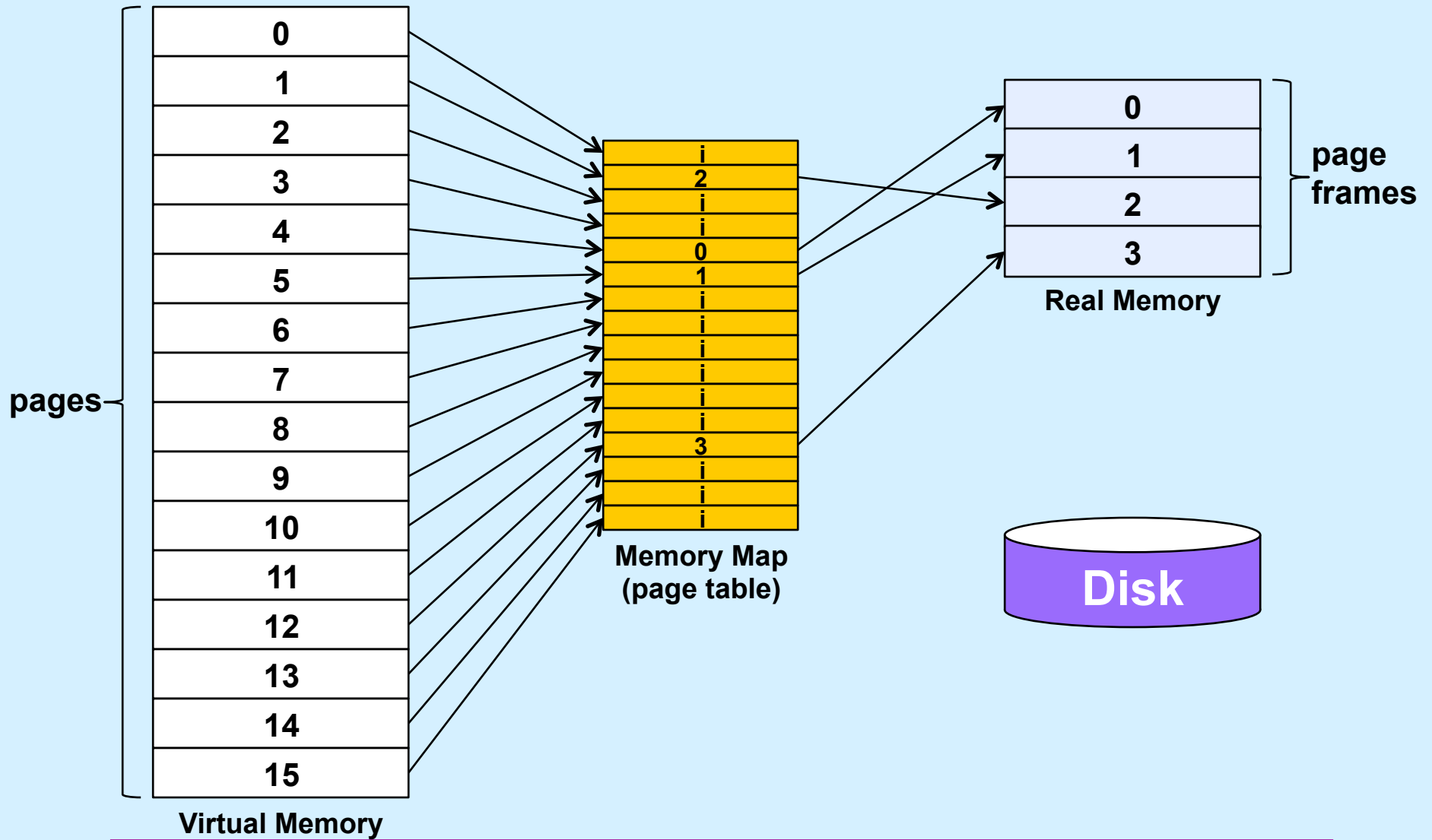
# Overlays



# Virtual Memory

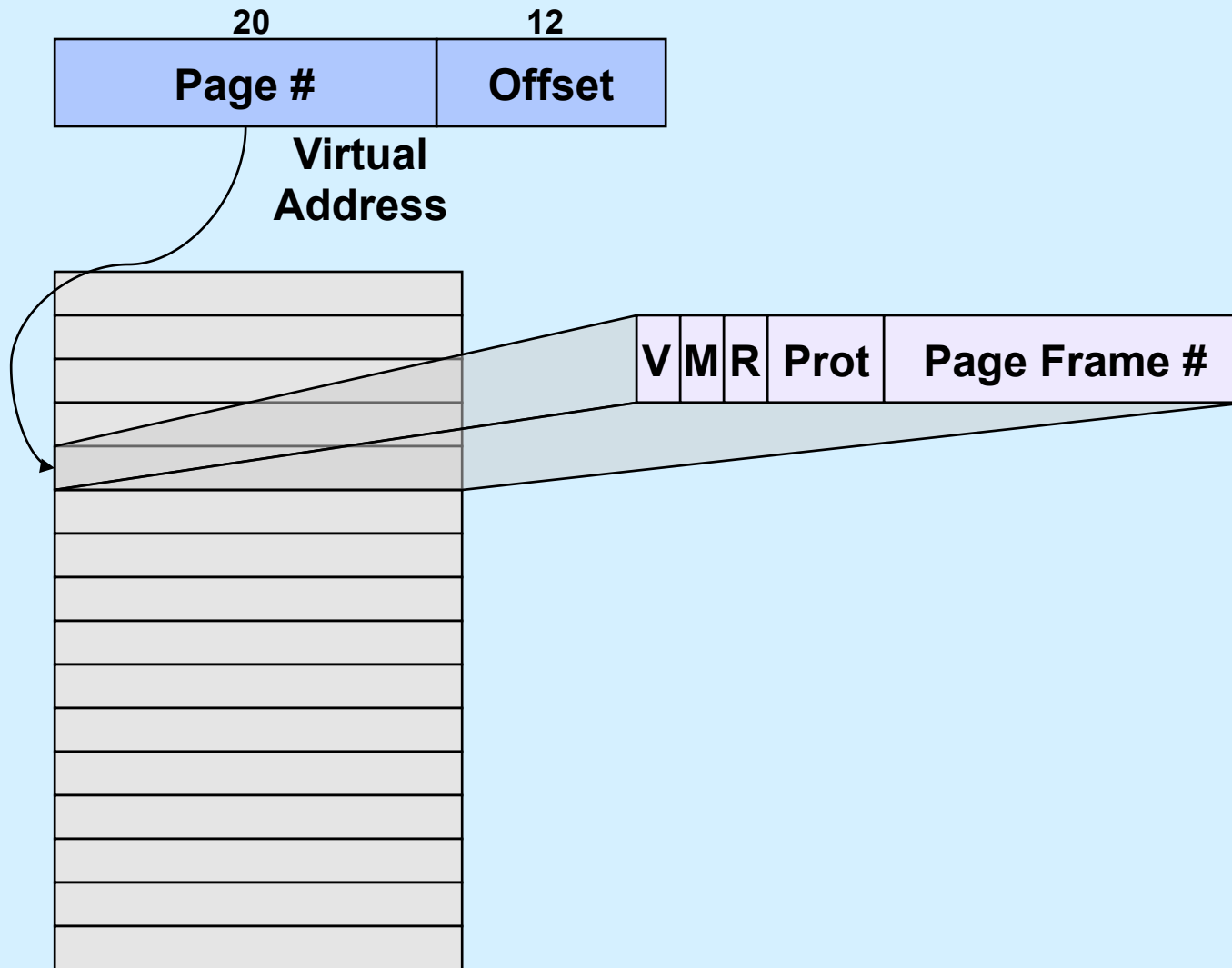


# Memory Maps





# Page Tables



# Quiz 1

How many  $2^{12}$ -byte pages fit in a 32-bit address space?

- a) a bit over a 1000
- b) a bit over a million
- c) a bit over a billion
- d) none of the above

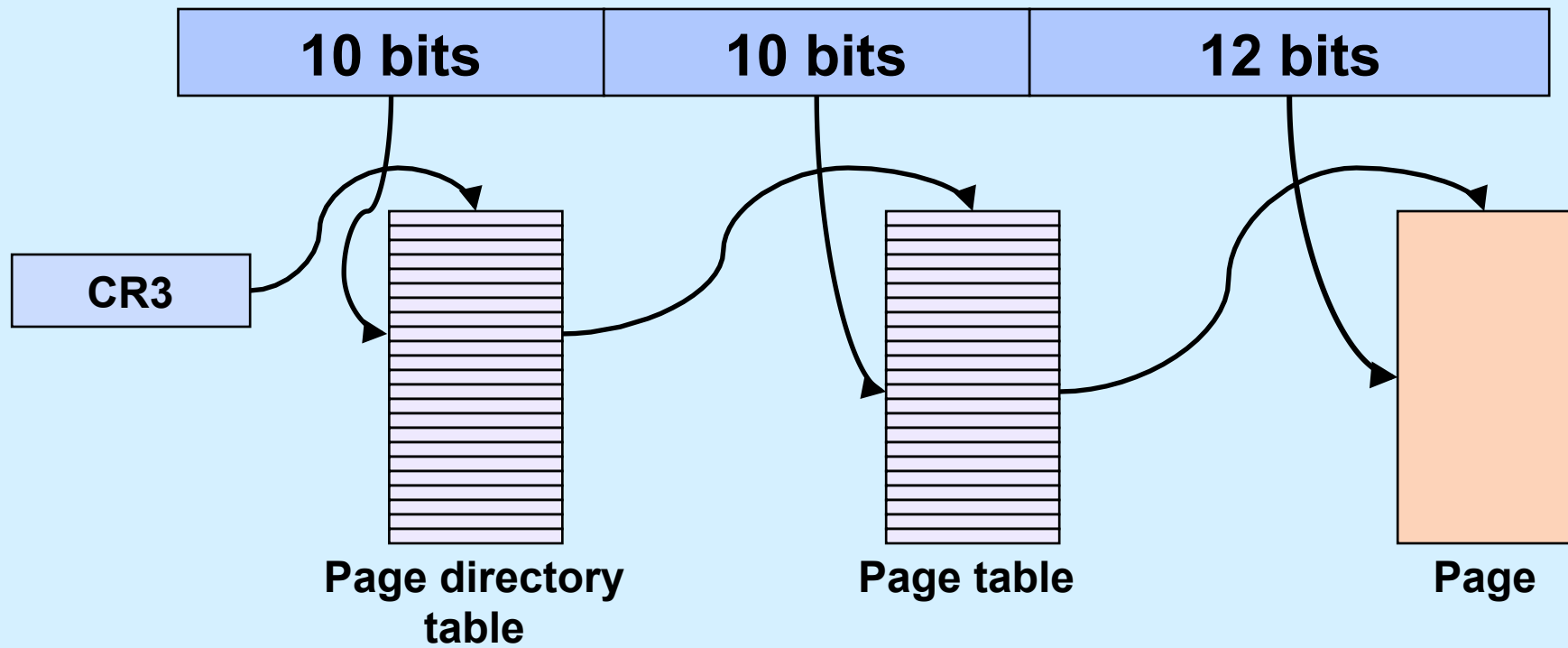
# VM is Your Friend ...

- **Not everything has to be in memory at once**
  - pages brought in (and pushed out) when needed
  - unallocated parts of the address space consume no memory
    - » e.g., hole between stack and dynamic areas
- **What's mine is not yours** (and vice versa)
  - address spaces are disjoint
- **Sharing is ok though ...**
  - address spaces don't have to be disjoint
    - » a single page frame may be mapped into multiple processes
- **I don't trust you (or me)**
  - access to individual pages can be restricted
    - » read, write, execute, or any combination

# Page-Table Size

- **Consider a full  $2^{32}$ -byte address space**
  - assume 4096-byte ( $2^{12}$ -byte) pages
  - 4 bytes per page-table entry
  - the page table would consist of  $2^{32}/2^{12}$  ( $= 2^{20}$ ) entries
  - its size would be  $2^{22}$  bytes (or 4 megabytes)
    - » at \$100/gigabyte
      - around \$0.40
- **For a  $2^{64}$ -byte address space**
  - assume 4096-byte ( $2^{12}$ -byte) pages
  - 8 bytes per page-table entry
  - the page table would consist of  $2^{64}/2^{12}$  ( $= 2^{52}$ ) entries
  - its size would be  $2^{55}$  bytes (or 32 petabytes)
    - » at \$1/gigabyte
      - over \$33 million

# IA32 Paging

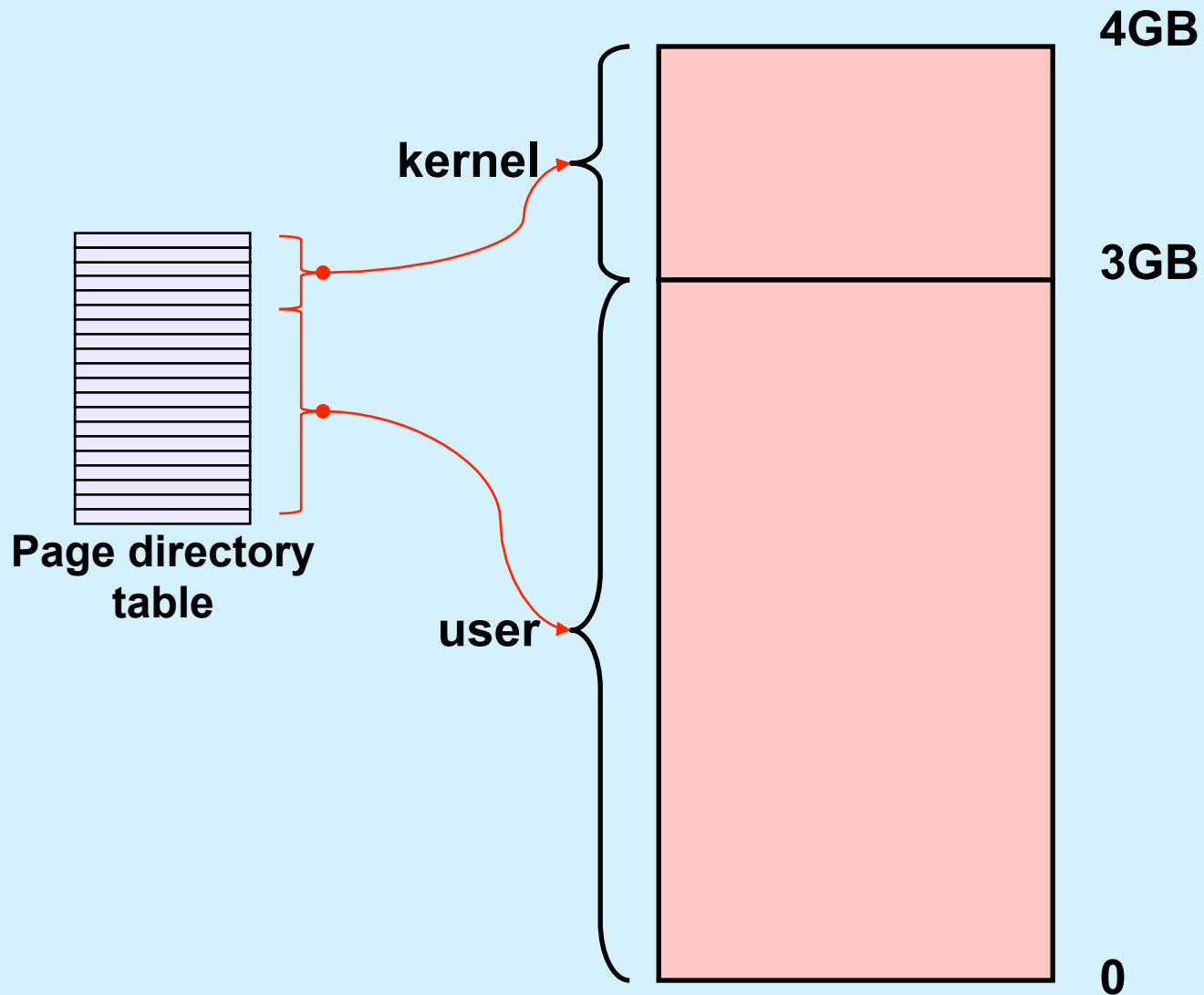


# Quiz 2

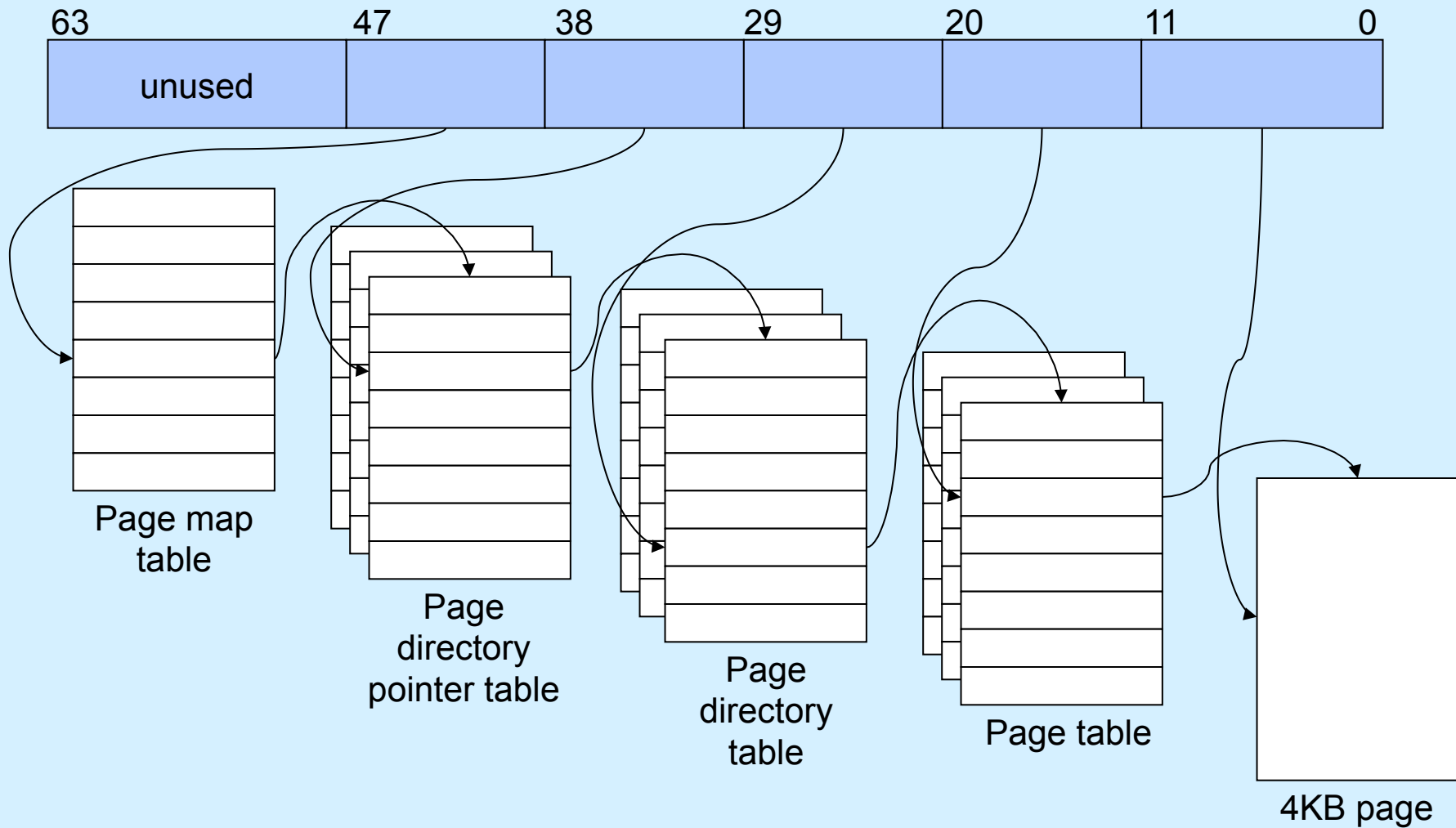
**Can a page start at a virtual address that's not divisible by the page size?**

- a) yes**
- b) no**

# Linux Intel IA32 VM Layout

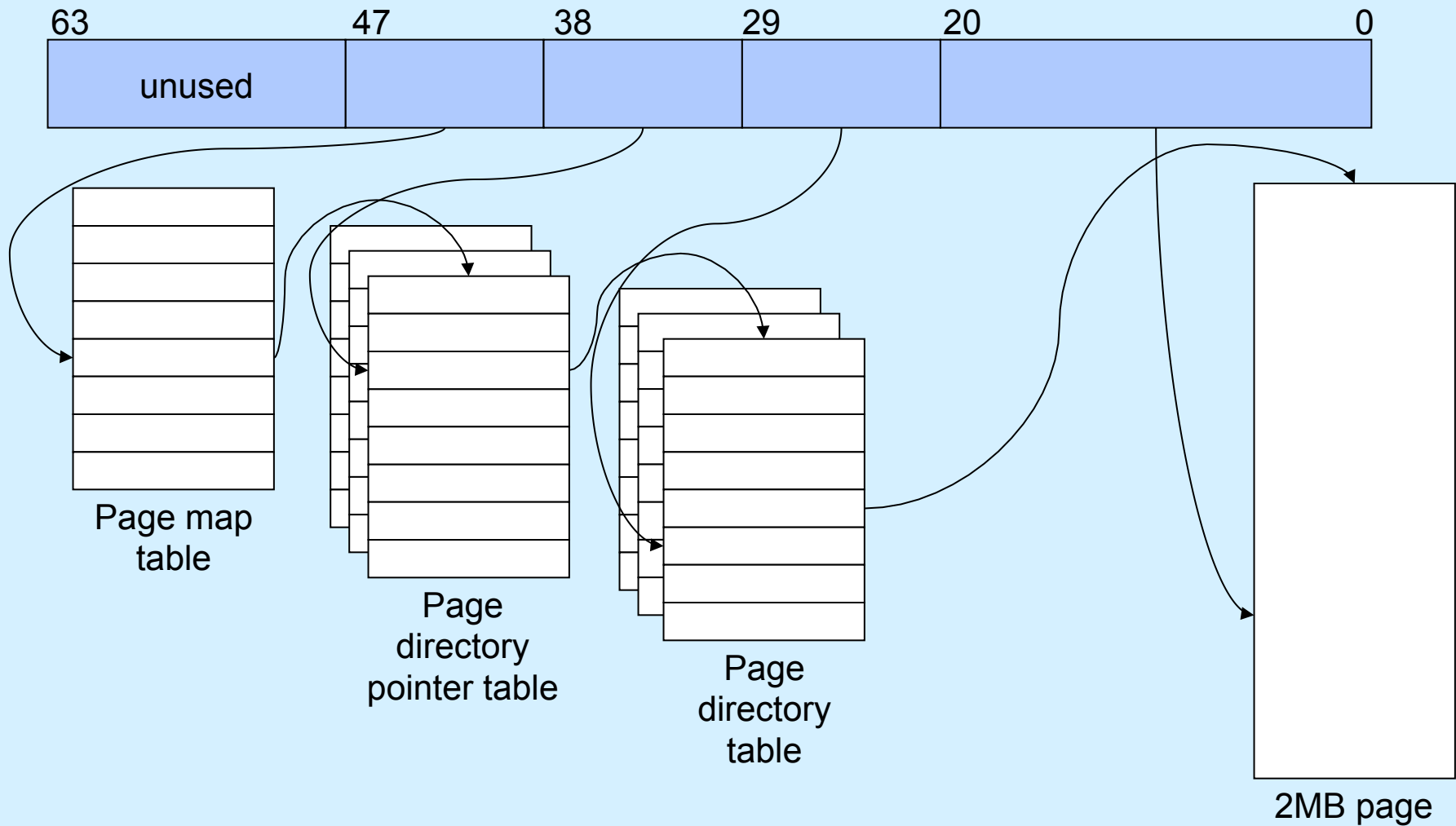


# x86-64 Virtual Address Format 1

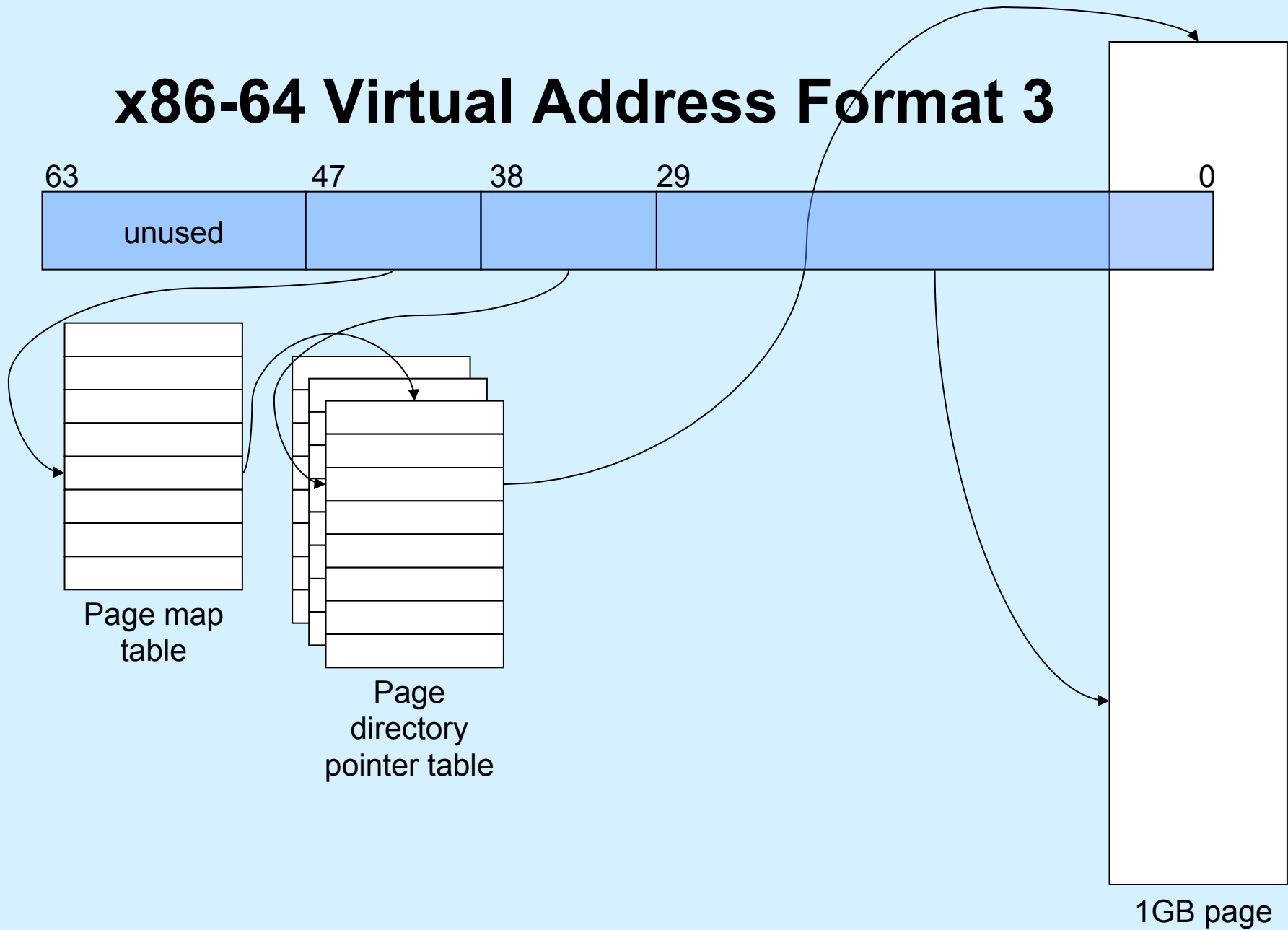




# x86-64 Virtual Address Format 2



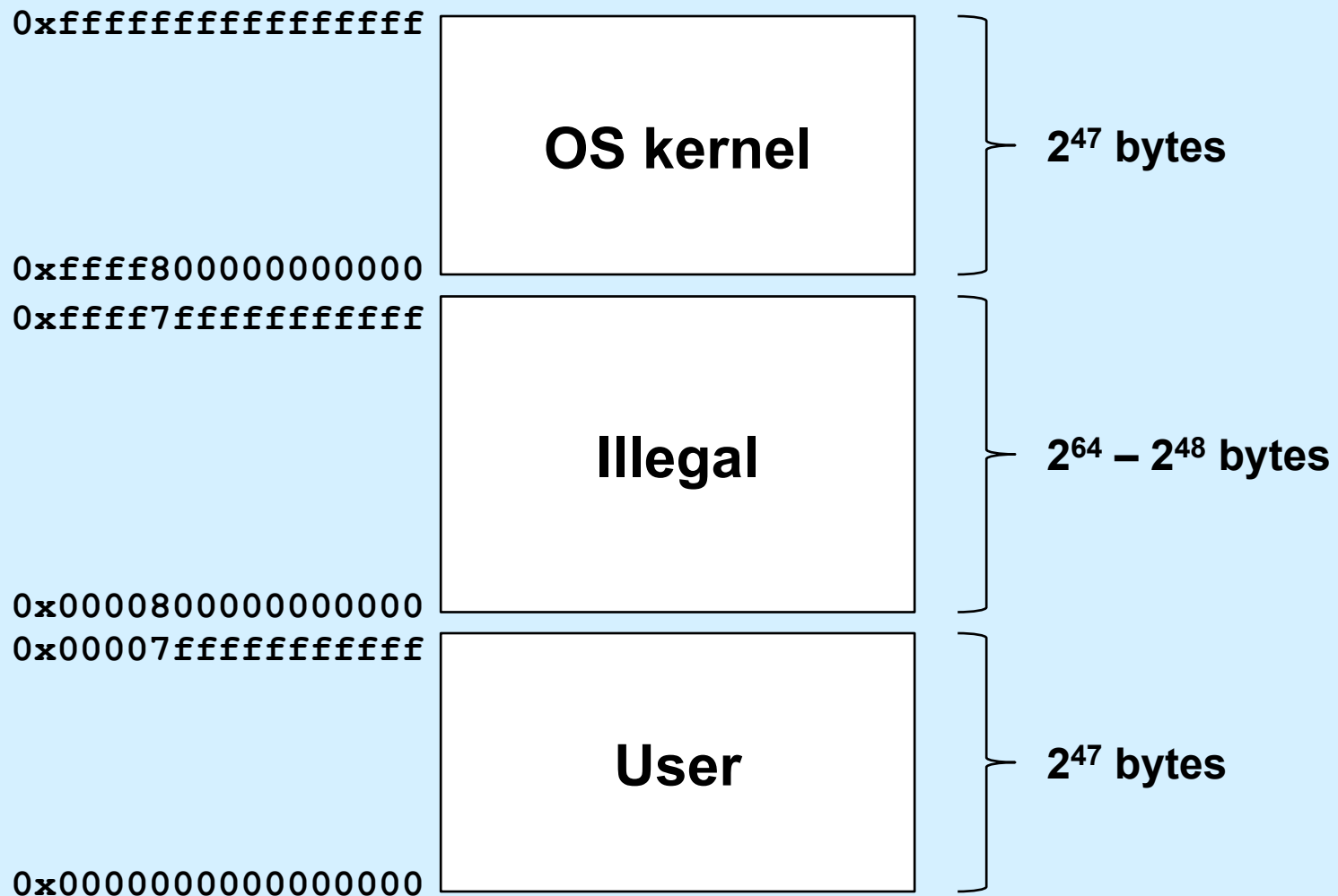
# x86-64 Virtual Address Format 3



# Why Multiple Page Sizes?

- **External fragmentation**
  - for region composed of 4KB pages, average external fragmentation is 2KB
  - for region composed of 1GB pages, average external fragmentation is 512MB
- **Page-table overhead**
  - larger page sizes have fewer page tables
    - » less overhead in representing mappings

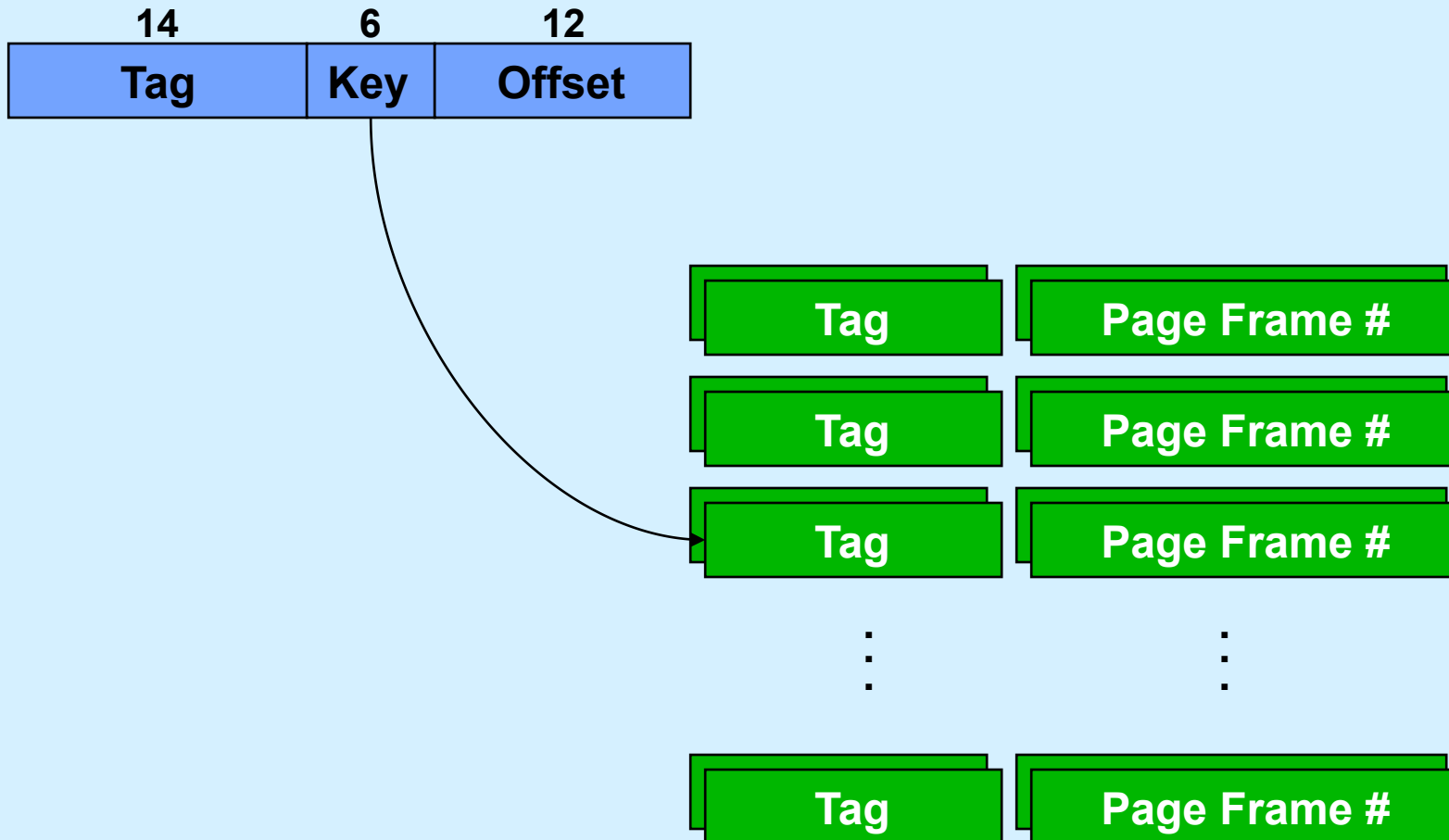
# Address Space



# Performance

- **Page table resides in real memory (DRAM)**
- **A 32-bit virtual-to-real translation requires two accesses to page tables, plus the access to the ultimate real address**
  - three real accesses for each virtual access
  - 3X slowdown!
- **A 64-bit virtual-to-real translation requires four accesses to page tables, plus the access to the ultimate real address**
  - 5X slowdown!

# Translation Lookaside Buffers



# Quiz 3

Recall that there is a 5x slowdown on memory references via virtual memory on the x86-64. If all references are translated via the TLB, the slowdown will be

- a) 1x
- b) 2x
- c) 3x
- d) 4x

# OS Role in Virtual Memory

- **Memory is like a cache**
  - quick access if what's wanted is mapped via page table
  - slow if not — OS assistance required
- **OS**
  - make sure what's needed is mapped in
  - make sure what's no longer needed is not mapped in



# Mechanism

- **Program references memory**
  - if reference is mapped, access is quick
    - » even quicker if translation in TLB and referent in on-chip cache
  - if not, page-translation fault occurs and OS is invoked
    - » determines desired page
    - » maps it in, if legal reference

# Issues

- **Fetch policy**
  - when are items put in the cache?
- **Placement policy**
  - where do they go in the cache?
- **Replacement policy**
  - what's removed to make room?

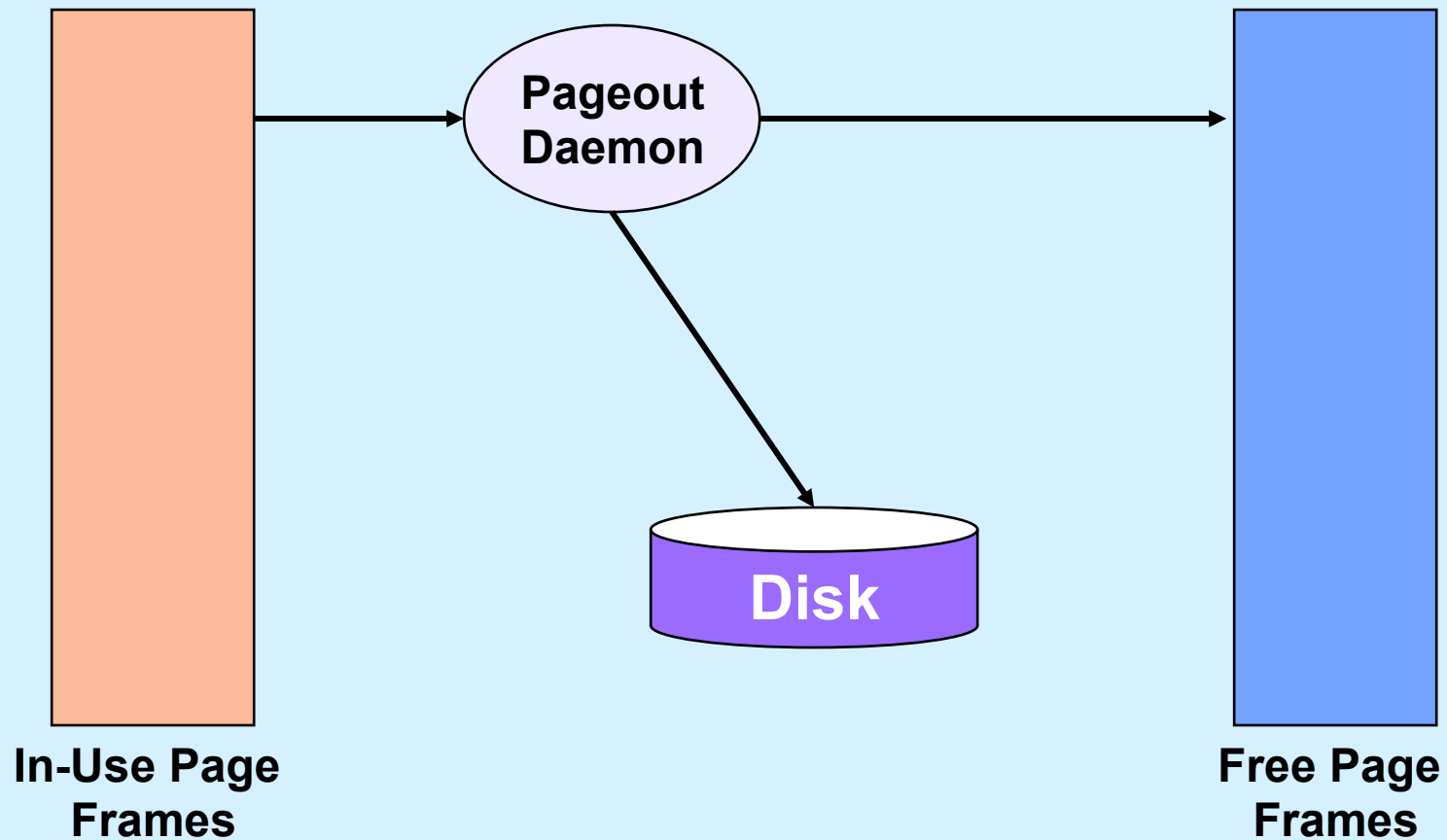
# Hardware Caches

- **Fetch policy**
  - when are items put in the cache?
    - » when they're referenced
    - » prefetch might be possible (e.g., for sequential access)
- **Placement policy**
  - where do they go in the cache?
    - » usually determined by cache architecture
    - » if there's a choice, it's typically a random choice
- **Replacement policy**
  - what's removed to make room?
    - » usually determined by cache architecture
    - » if there's a choice, it's typically a random choice

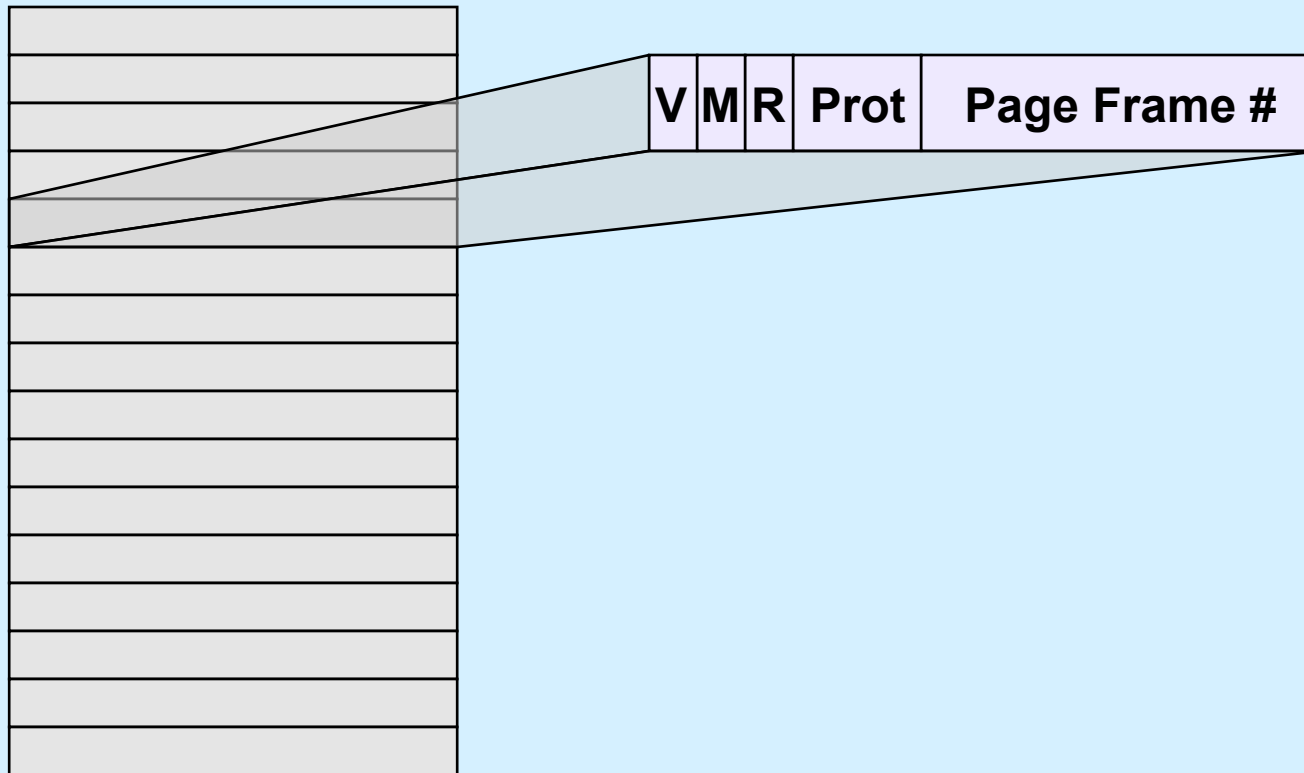
# Software Caches

- **Fetch policy**
  - when are items put in the cache?
    - » when they're referenced
    - » prefetch might be easier than for hardware caches
- **Placement policy**
  - where do they go in the cache?
    - » usually doesn't matter (no memory is more equal than others)
- **Replacement policy**
  - what's removed to make room?
    - » would like to remove that whose next use is farthest in future
    - » instead, remove that whose last reference was farthest in the past

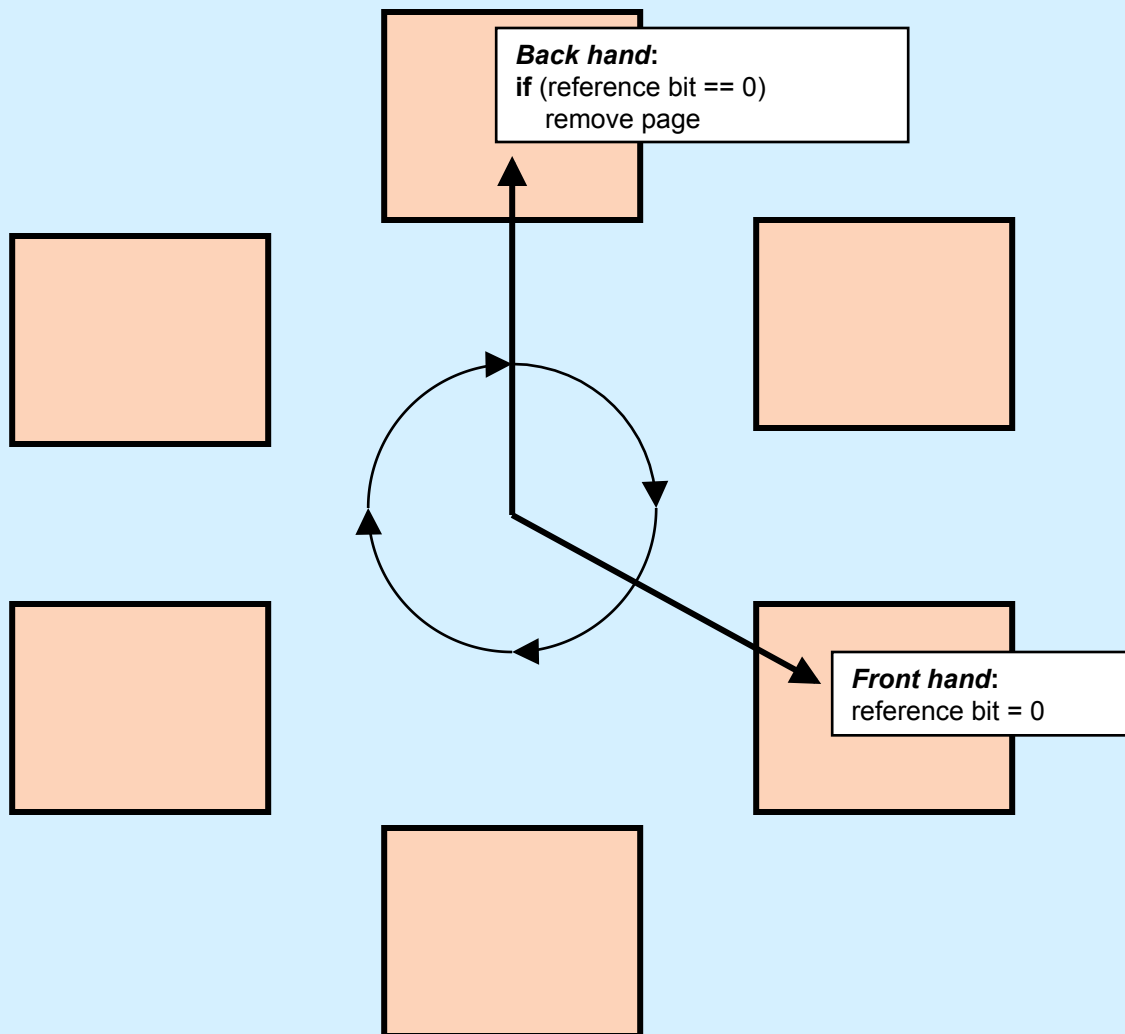
# The “Pageout Daemon”



# Managing Page Frames



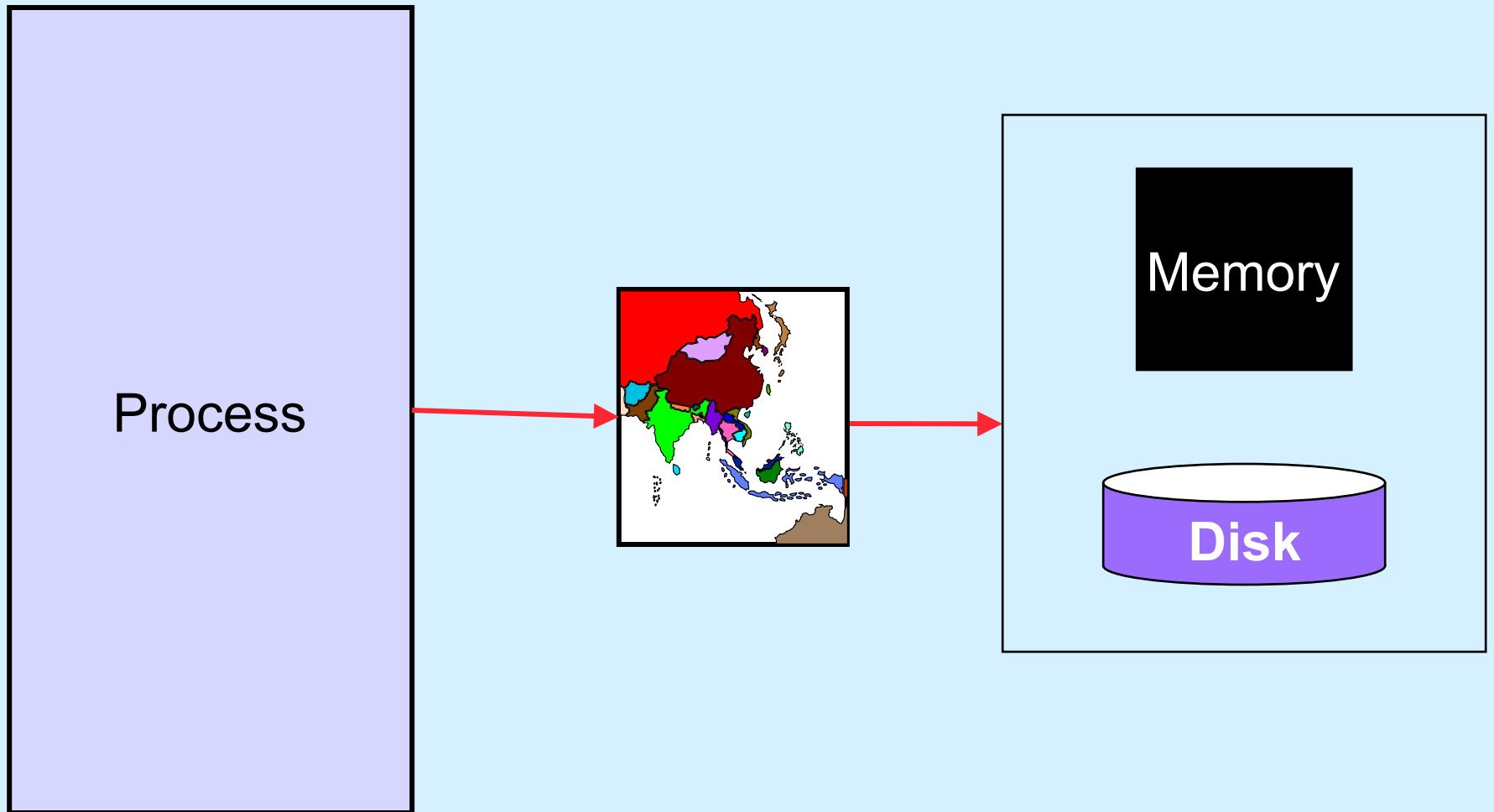
# Clock Algorithm



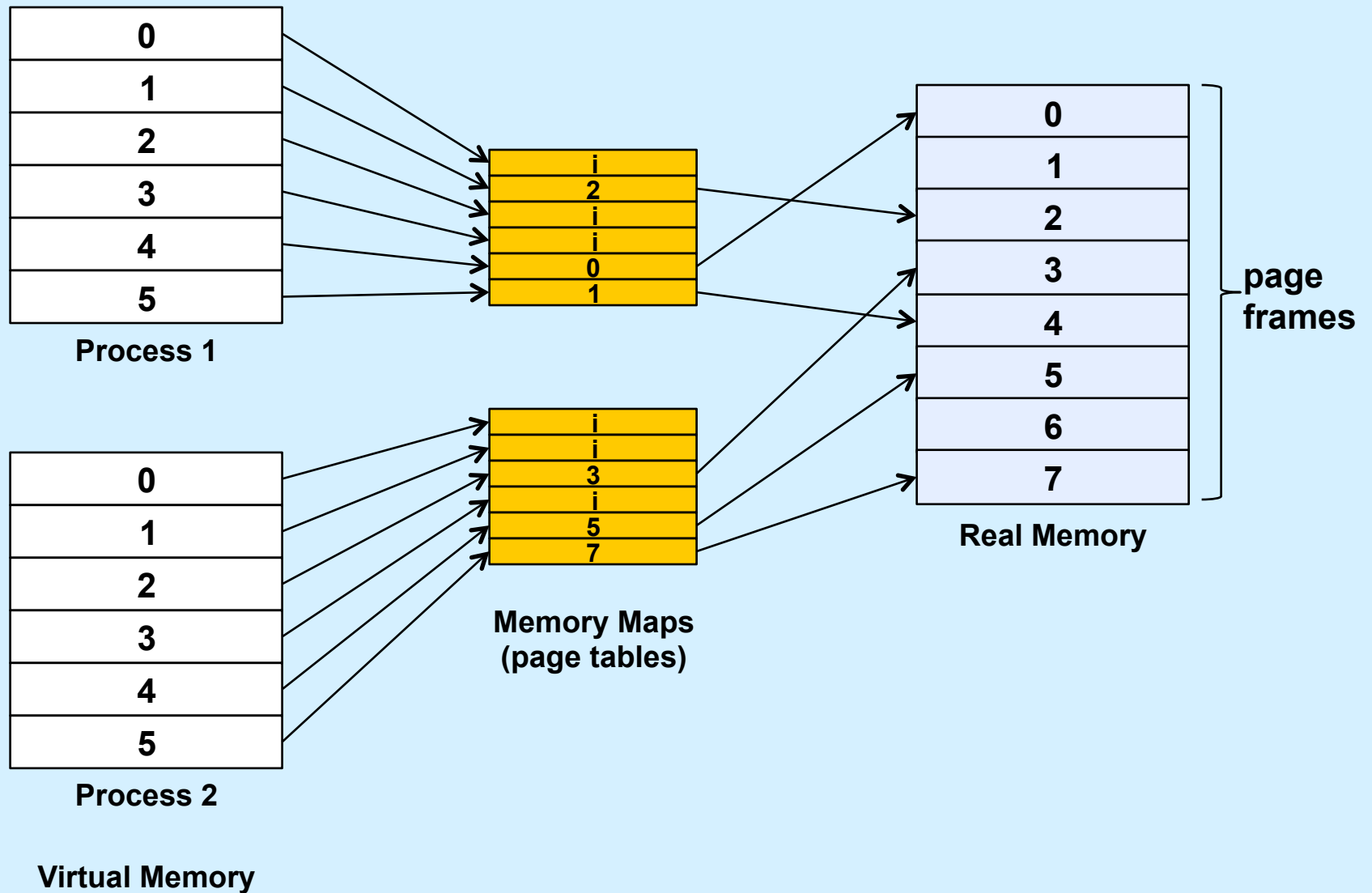
# Why is virtual memory used?



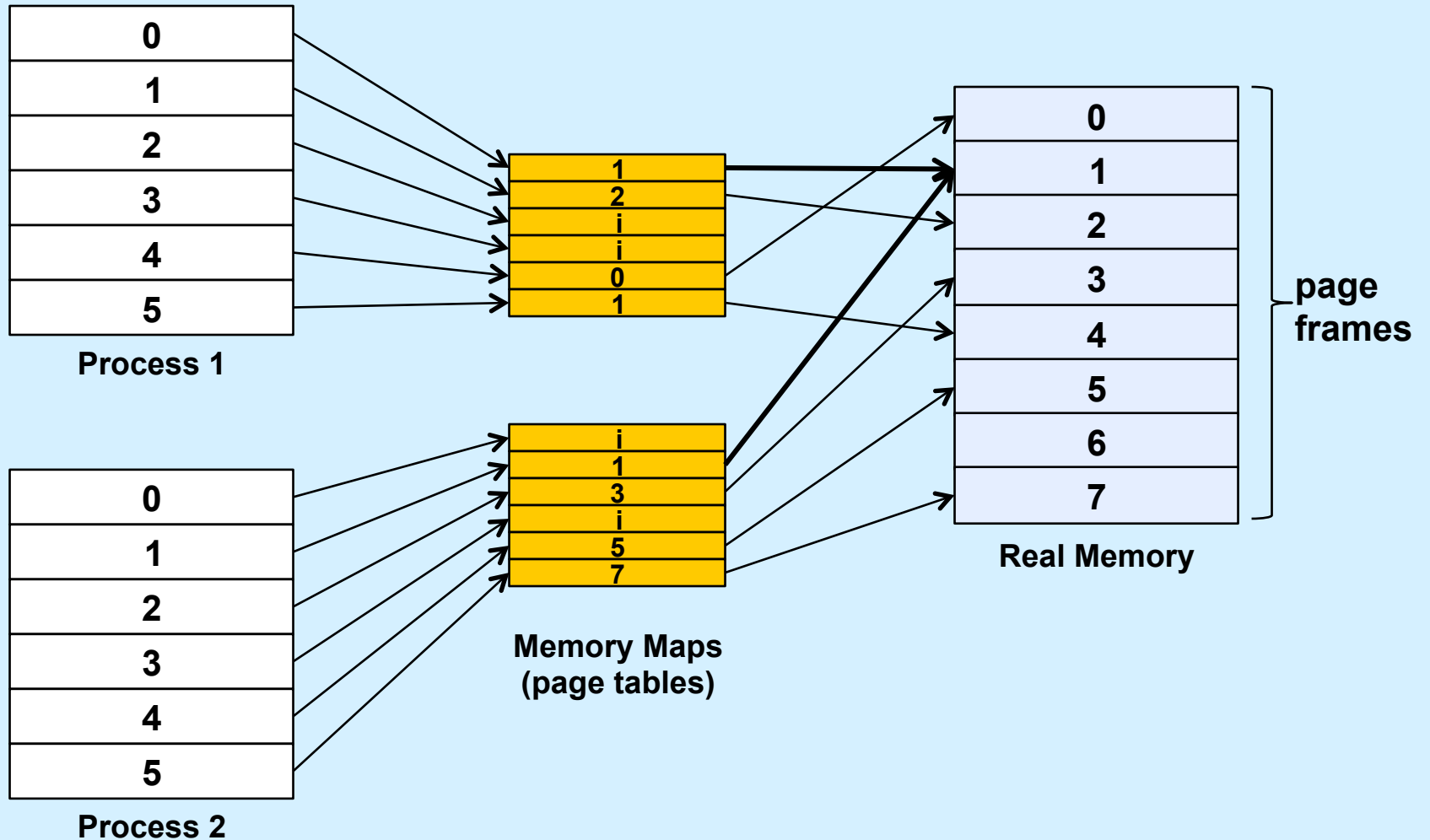
# More VM than RM



# Isolation



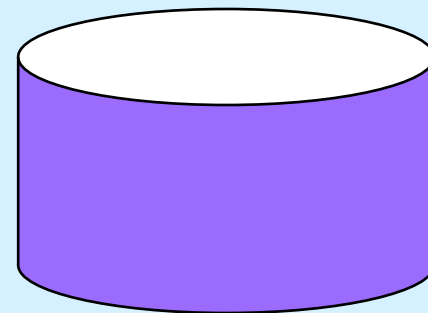
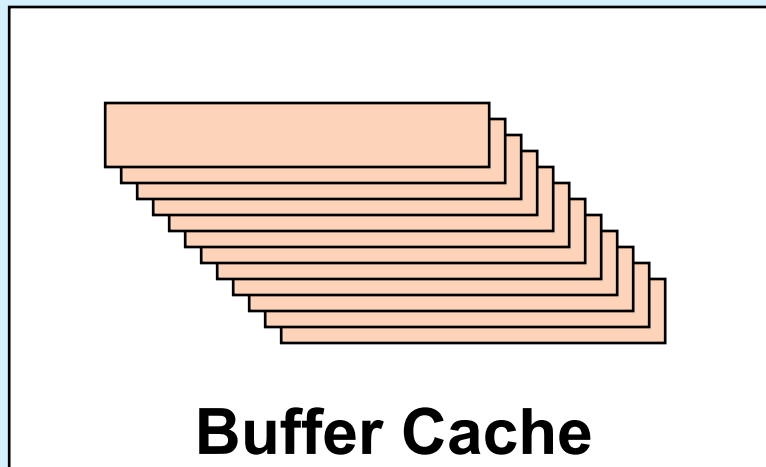
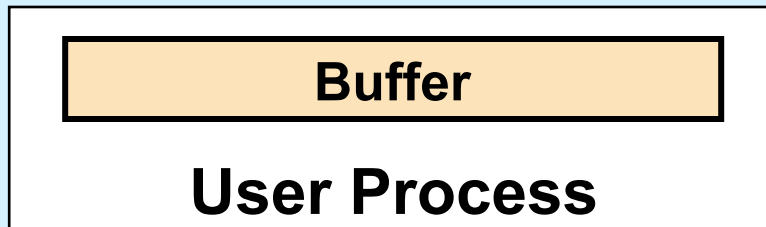
# Sharing



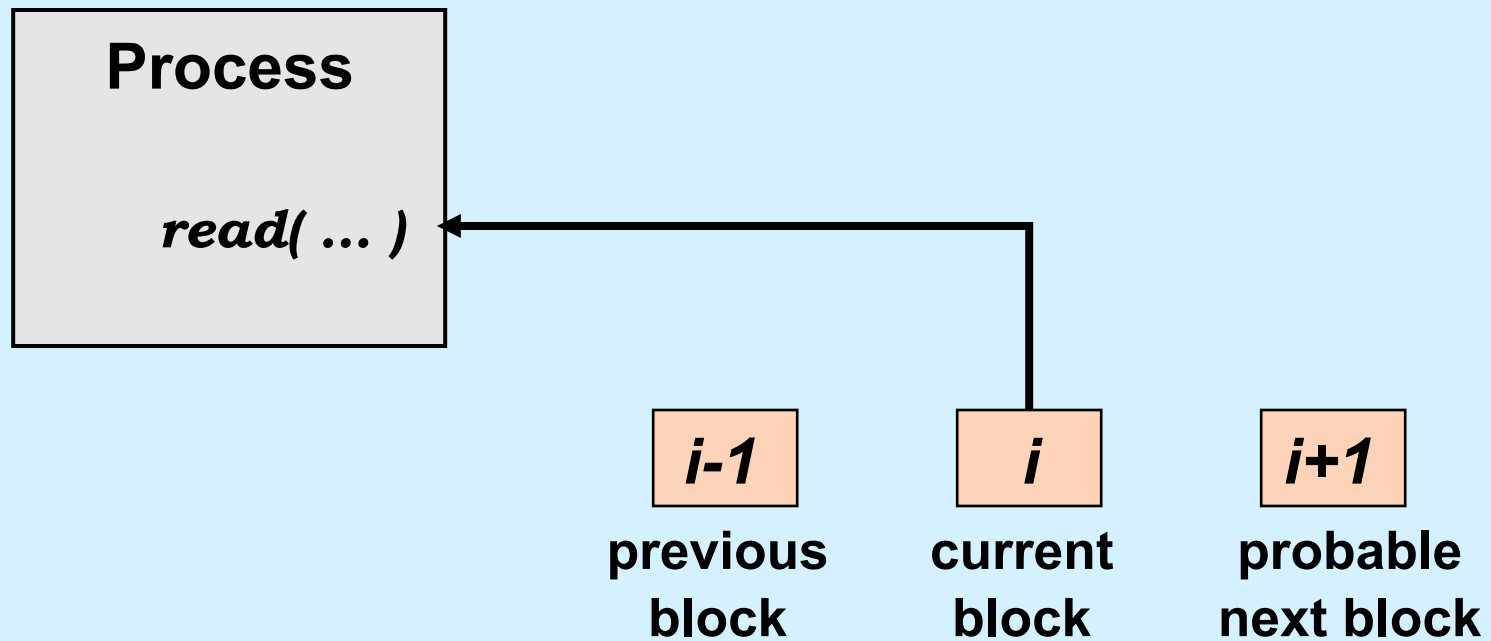
**Virtual Memory**

---

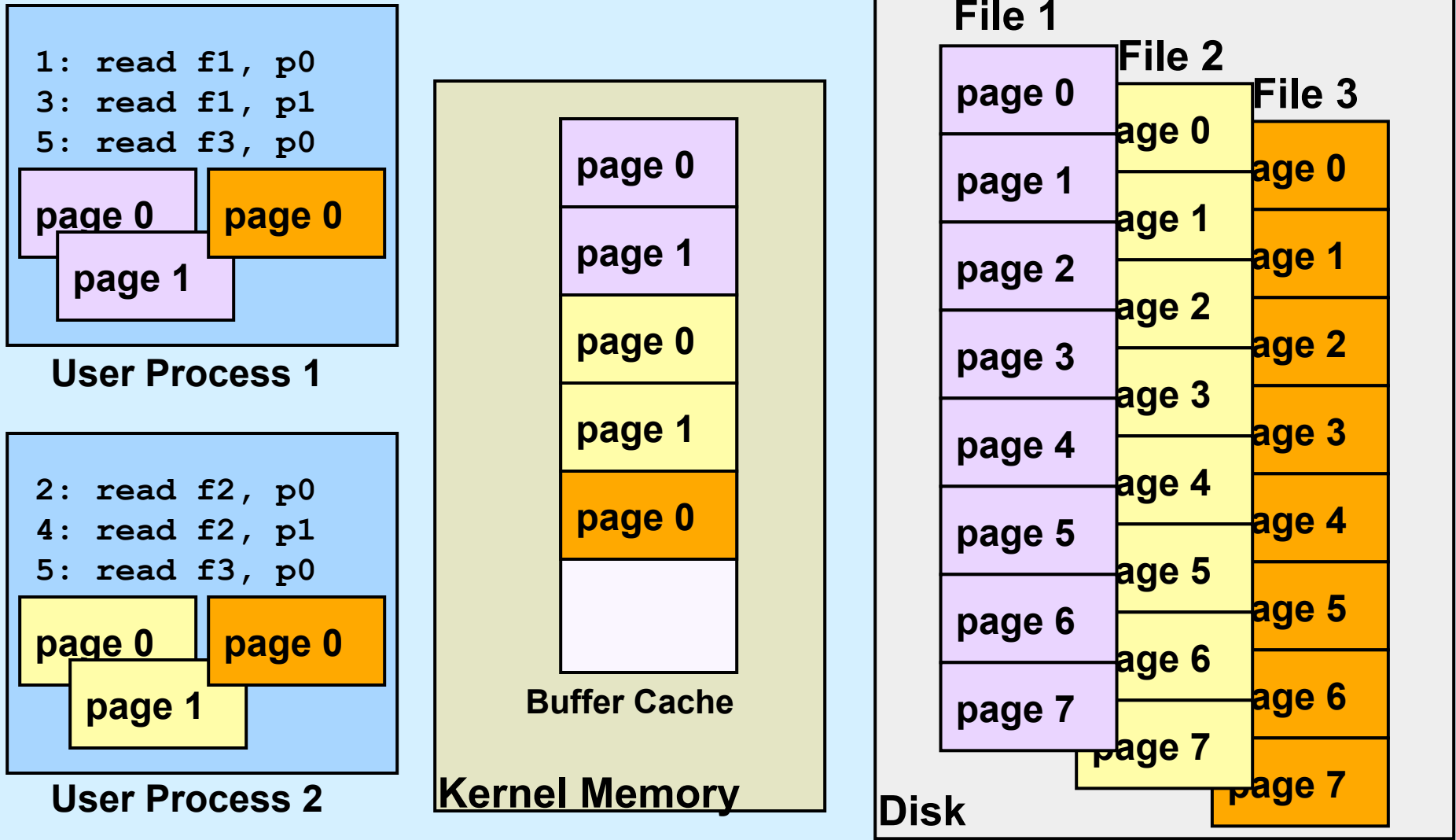
# File I/O



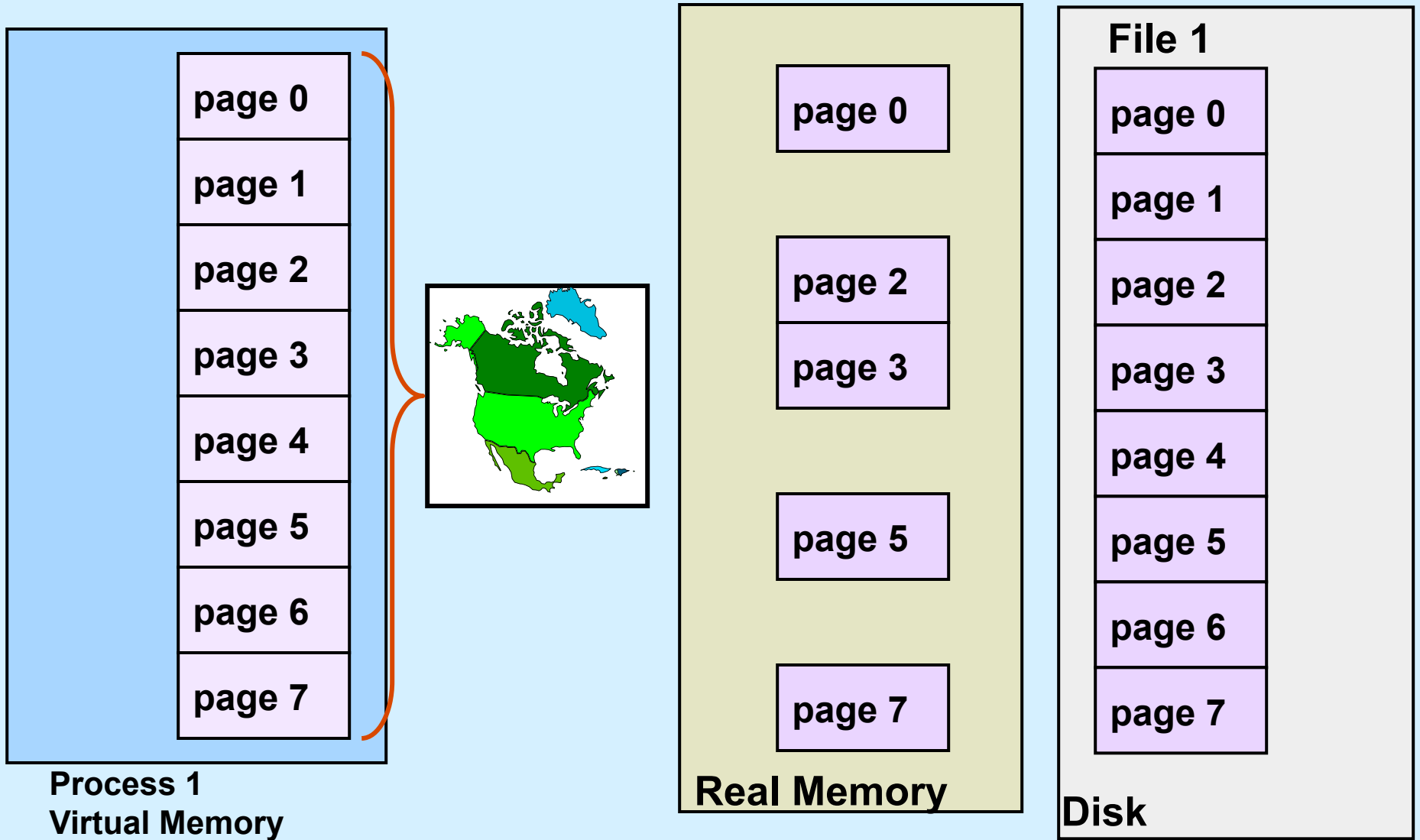
# Multi-Buffered I/O



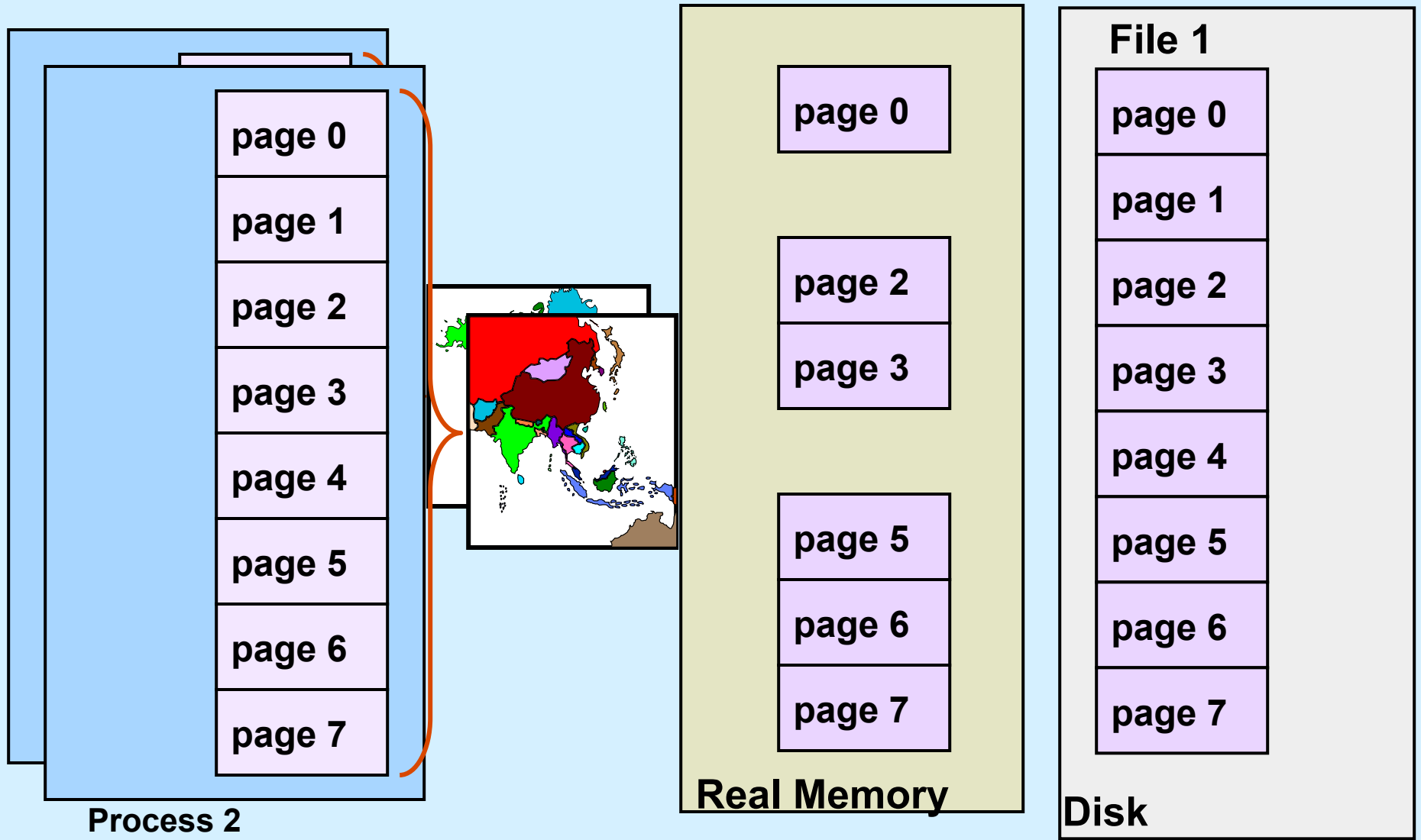
# Traditional I/O



# Mapped File I/O



# Multi-Process Mapped File I/O





# Mapped Files

- **Traditional File I/O**

```
char buf[BigEnough];
fd = open(file, O_RDWR);
for (i=0; i<n_recs; i++) {
    read(fd, buf, sizeof(buf));
    use(buf);
}
```

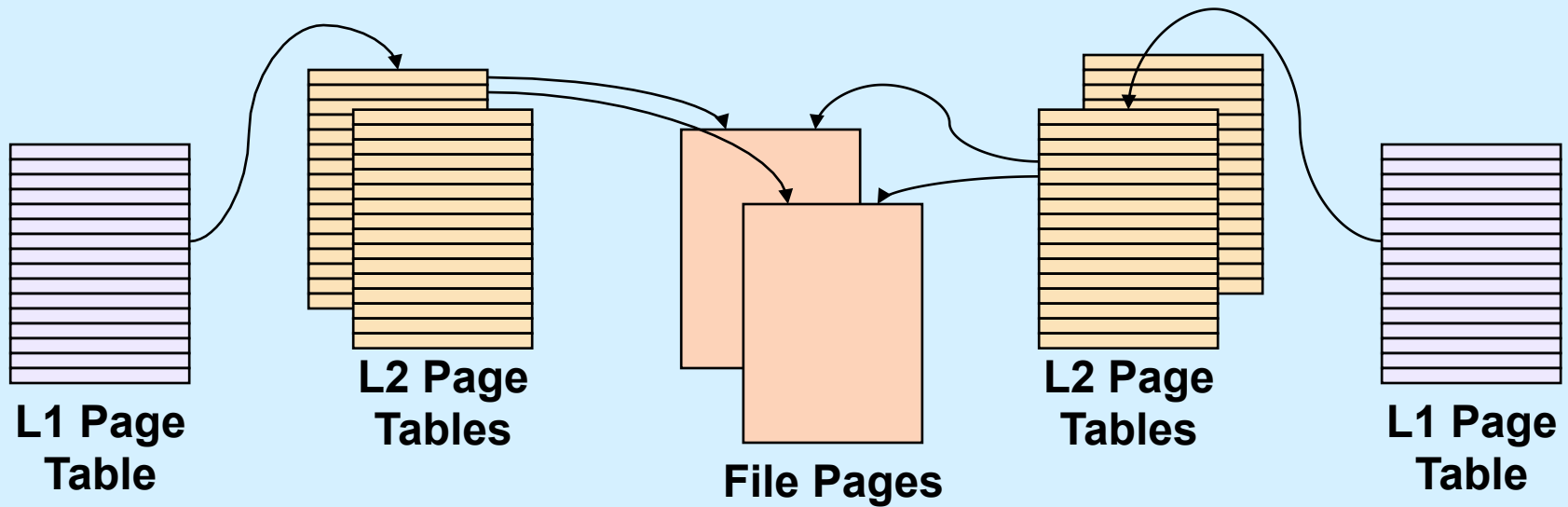
- **Mapped File I/O**

```
record_t *MappedFile;
fd = open(file, O_RDWR);
MappedFile = mmap(... , fd, ...);
for (i=0; i<n_recs; i++)
    use(MappedFile[i]);
```

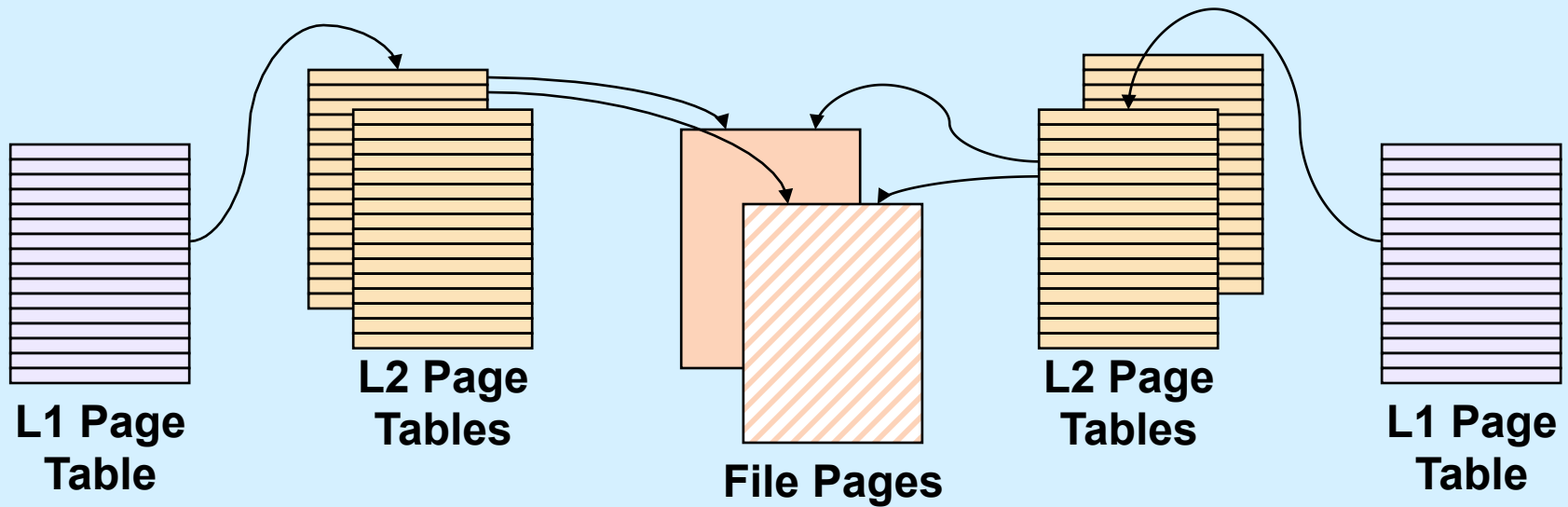
# Mmap System Call

```
void *mmap(  
    void *addr,  
    // where to map file (0 if don't care)  
    size_t len,  
    // how much to map  
    int prot,  
    // memory protection (read, write, exec.)  
    int flags,  
    // shared vs. private, plus more  
    int fd,  
    // which file  
    off_t off  
    // starting from where  
);
```

# The *mmap* System Call

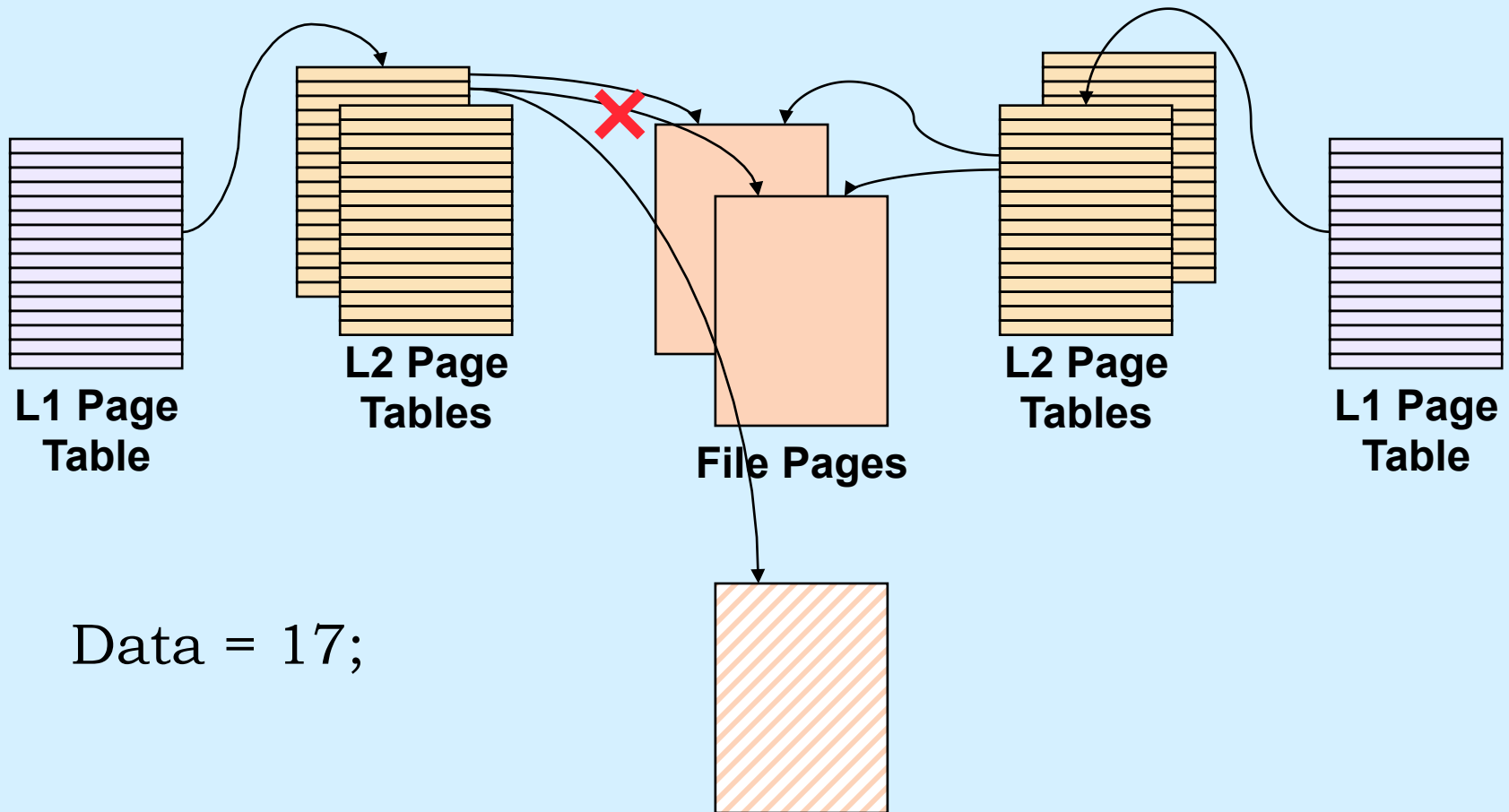


# Share-Mapped Files



Data = 17;

# Private-Mapped Files



Data = 17;

# Example

```
int main( ) {
    int fd;
    dataObject_t *dataObjectp;

    fd = open("file", O_RDWR);
    if ((int)(dataObjectp = (dataObject_t *)mmap(0,
        sizeof(dataObject_t),
        PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0)) == -1) {
        perror("mmap");
        exit(1);
    }

    // dataObjectp points to region of (virtual) memory
    // containing the contents of the file

    ...
}
```

# fork and mmap

```
int main() {
    int x=1;

    if (fork() == 0) {
        // in child
        x = 2;
        exit(0);
    }
    // in parent
    while (x==1) {
        // will loop forever
    }
    return 0;
}
```

```
int main() {
    int fd = open( ... );
    int *xp = (int *)mmap(...,
        MAP_SHARED, fd, ...);
    xp[0] = 1;
    if (fork() == 0) {
        // in child
        xp[0] = 2;
        exit(0);
    }
    // in parent
    while (xp[0]==1) {
        // will terminate
    }
    return 0;
}
```