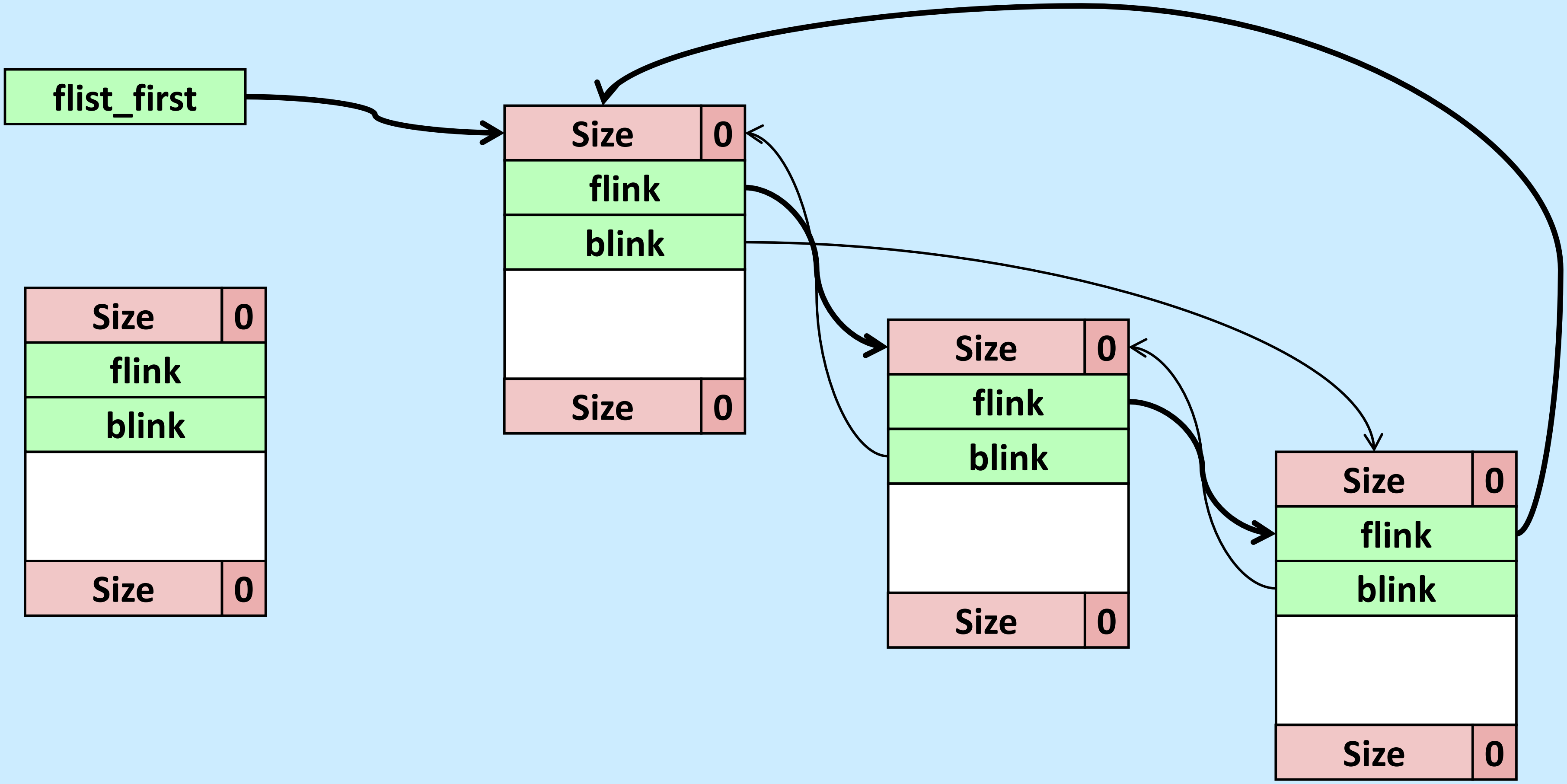
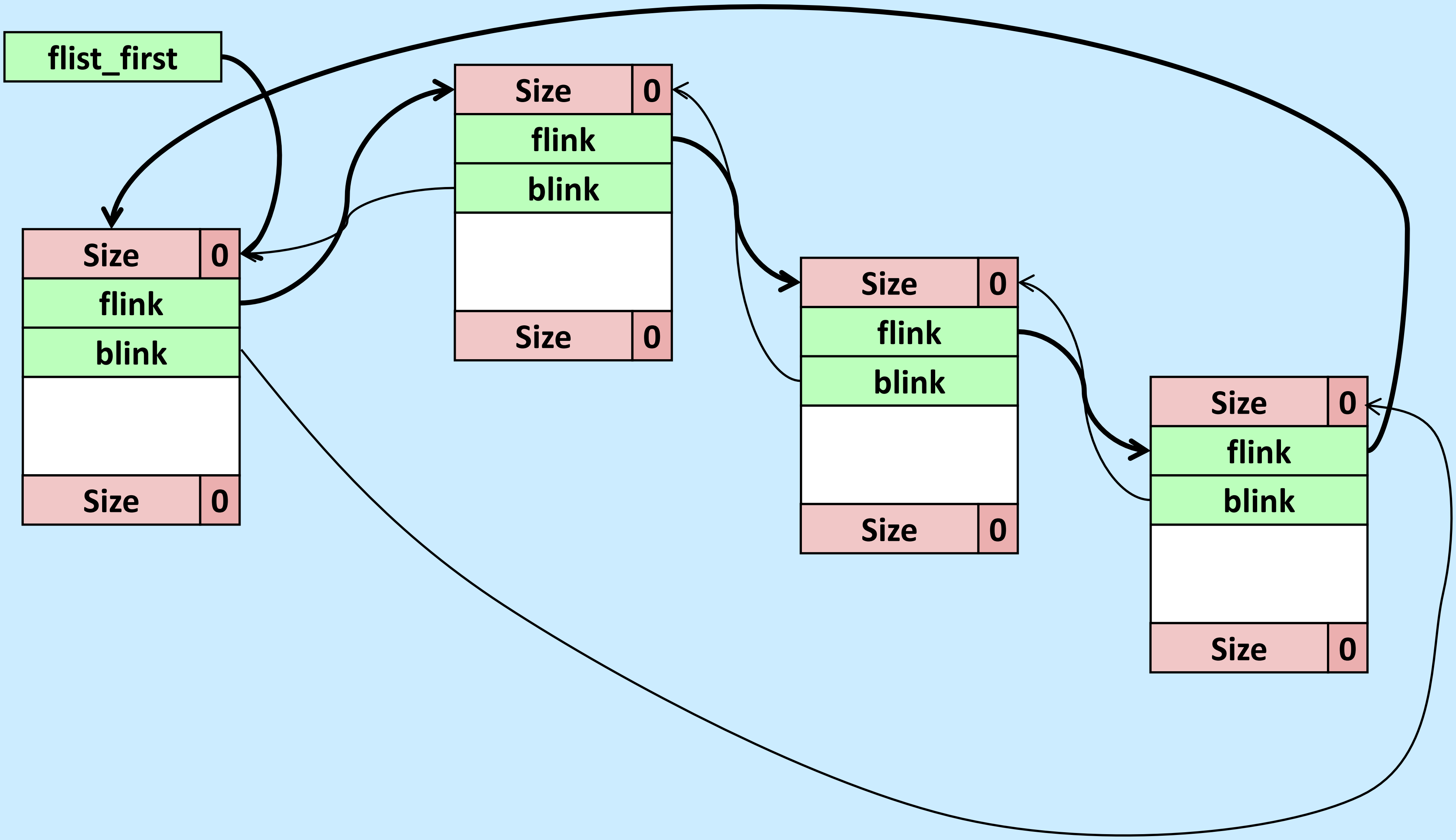
CS 33

Storage Allocation (2)

Adding a Block to the Free List (1)



Adding a Block to the Free List (2)



Accessing the Object

```
block_t *block_flink(block_t *b) {
  return (block_t *)b->payload[0];
}

void block_set_flink(block_t *b, block_t *next) {
  b->payload[0] = (size_t)next;
}

block_t *block_blink(block_t *b) {
  return (block_t *)b->payload[1];
}

void block_set_blink(block_t *b, block_t *next) {
  b->payload[1] = (size_t)next;
}
```

Insertion Code

```
void insert_free_block(block_t *fb) {
  assert(!block_allocated(fb));
  if (flist_first != NULL) {
    block_t *last =
      block_blink(flist_first);
    block_set_flink(fb, flist_first);
    block_set_blink(fb, last);
    block_set_flink(last, fb);
    block_set_blink(flist_first, fb);
  } else {
    block_set_flink(fb, fb);
    block_set_blink(fb, fb);
  }
  flist_first = fb;
}
```

Performance

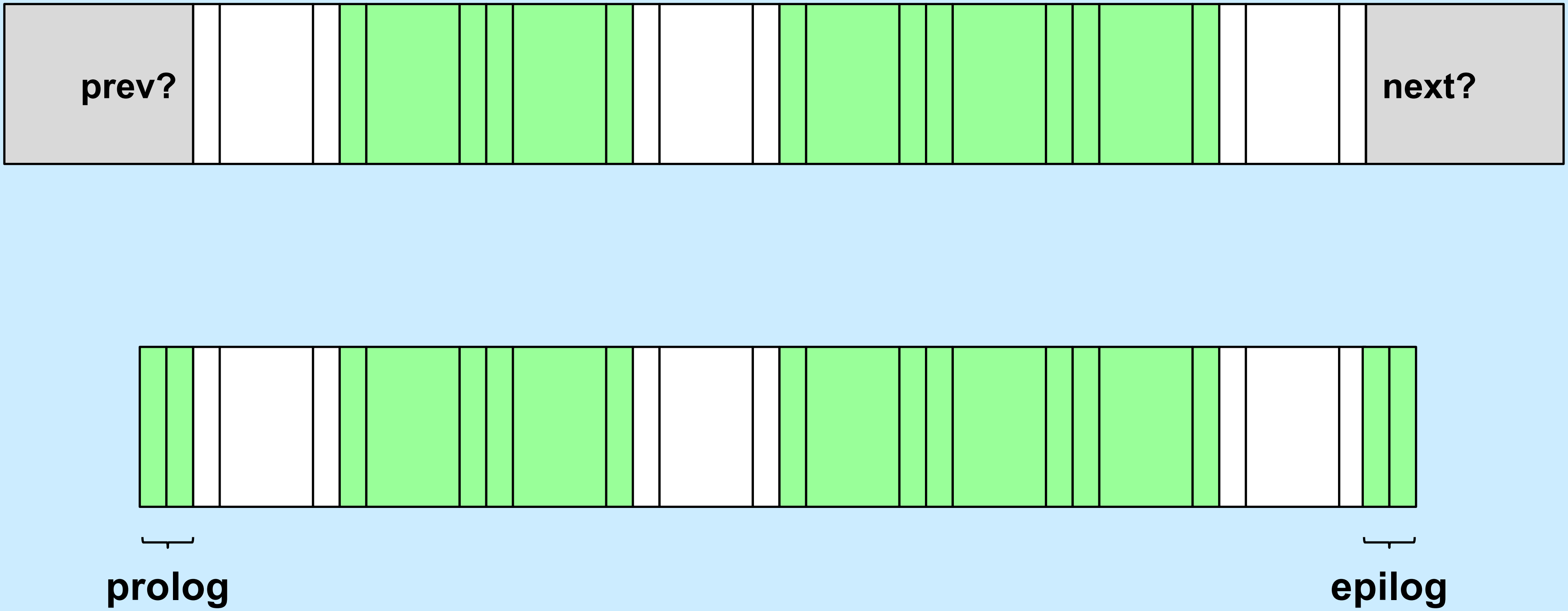
- Won’t all the calls to the accessor functions slow things down a lot?
  - yes — not just a lot, but tons
- Why not use macros (#define) instead?
  - the textbook does this
  - it makes the code impossible to debug
    - gdb shows only the name of the macro, not its body
- What to do????

Inline Functions

```
static inline size_t block_size(
    block_t *b) {
  return b->size & -2;
}
```

- when debugging (–O0), the code is implemented as a normal function
  - » easy to debug with gdb
- when optimized (–O1, –O2), calls to the function are replaced with the body of the function
  - » no function-call overhead

Prolog and Epilog



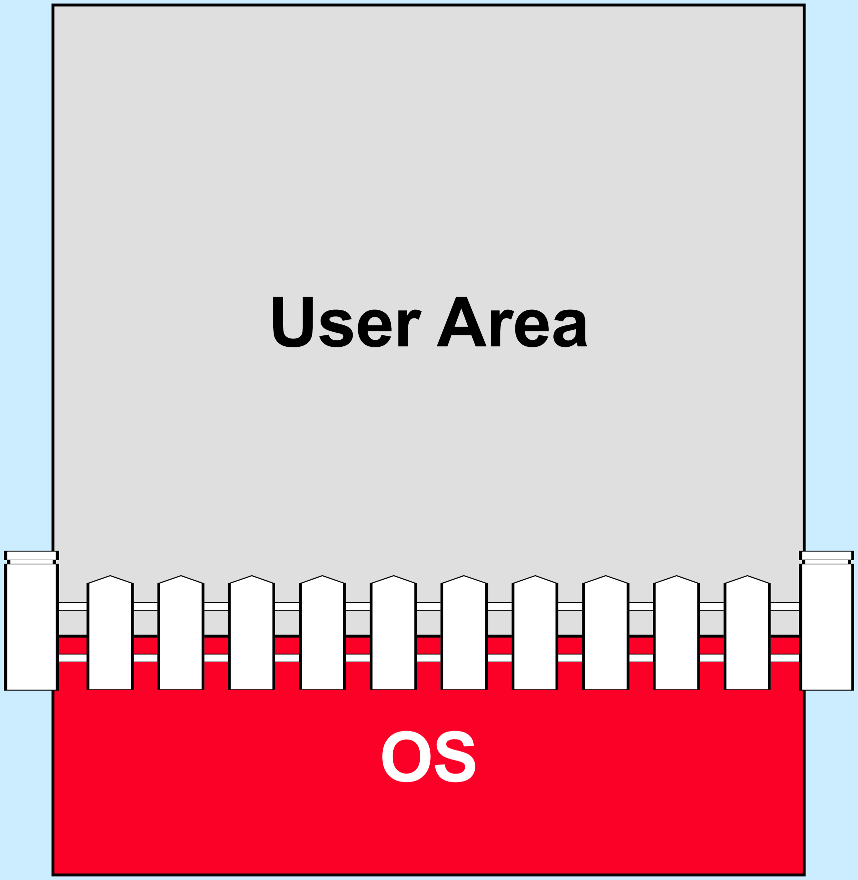
CS 33

Virtual Memory

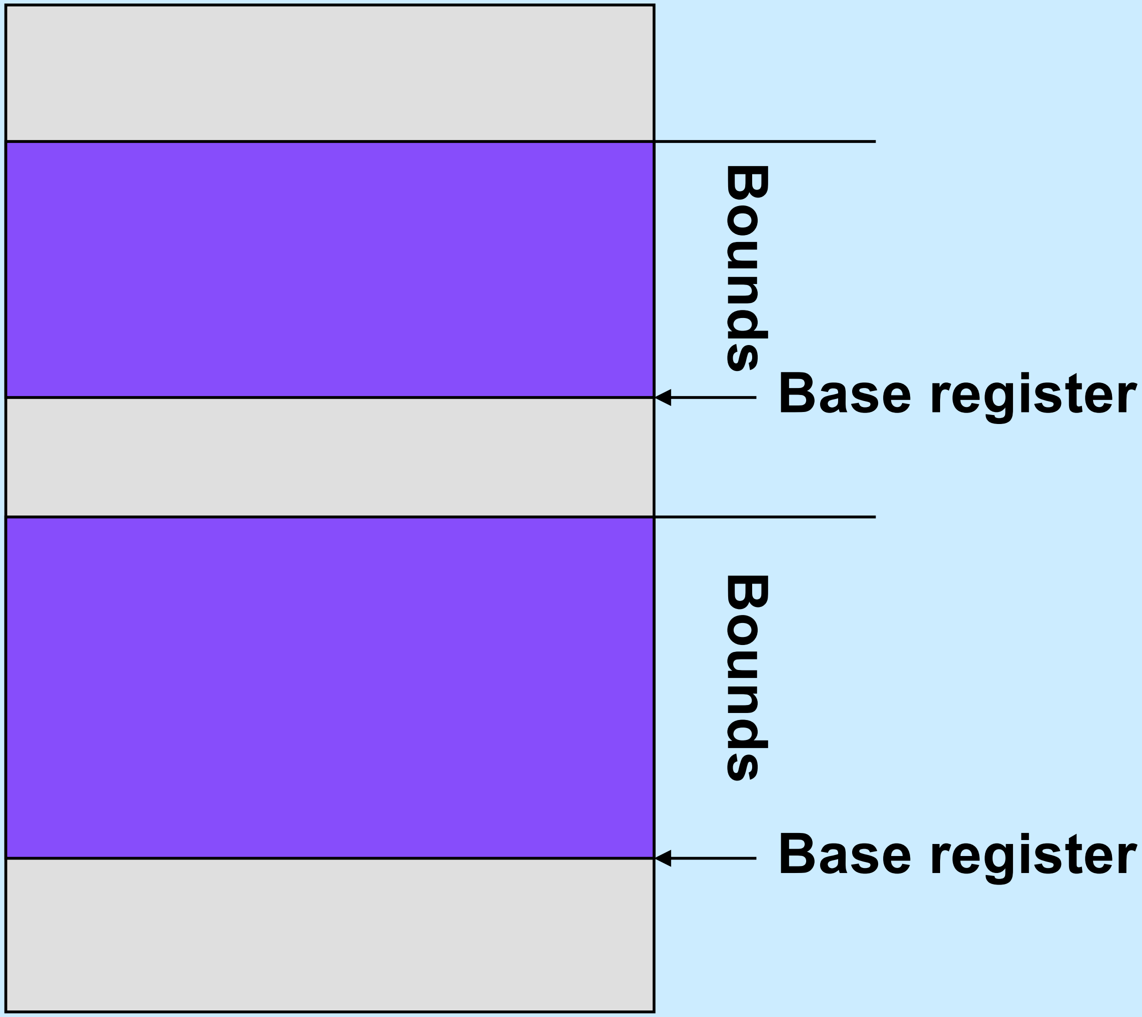
The Address-Space Concept

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

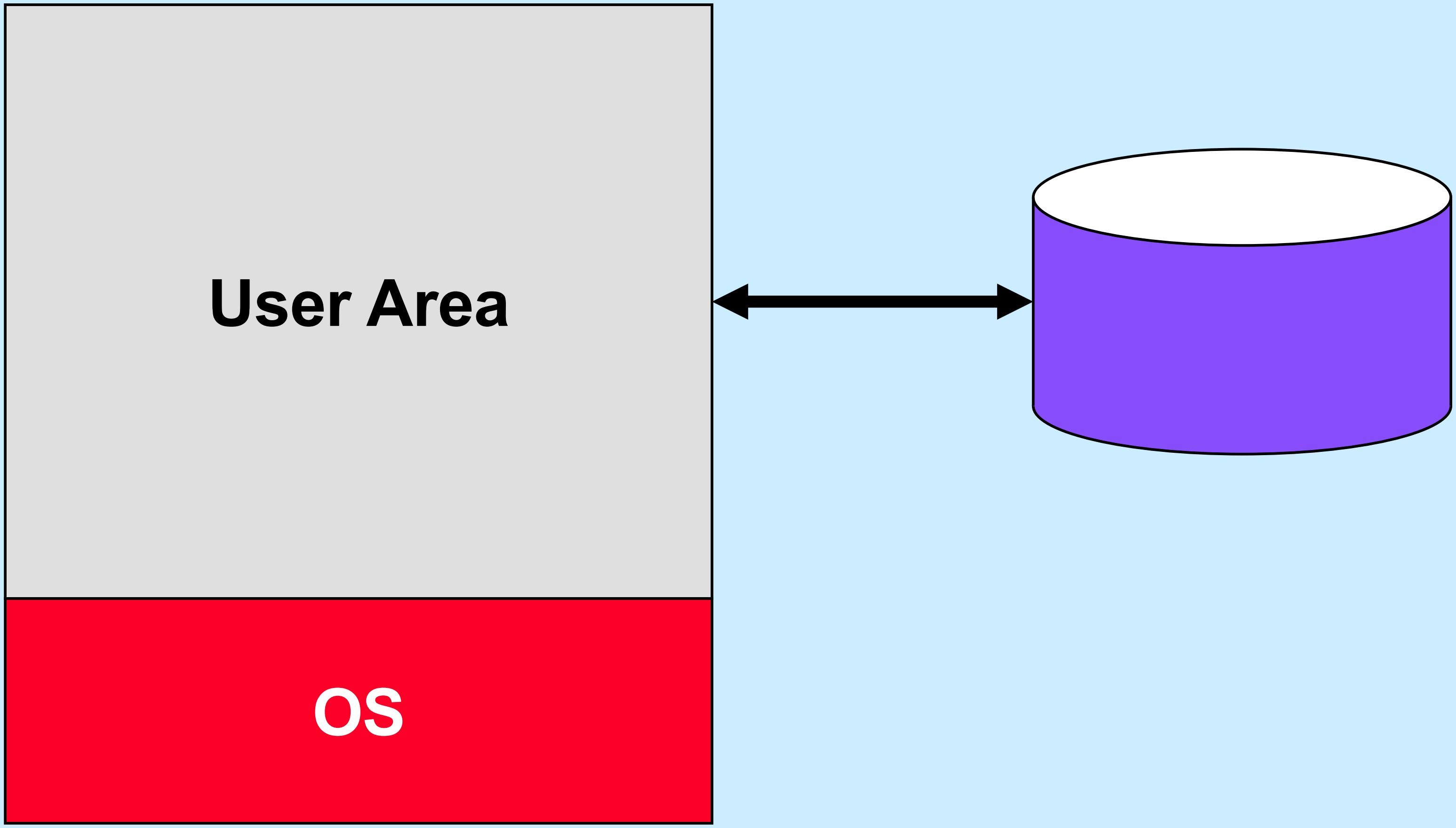
Memory Fence



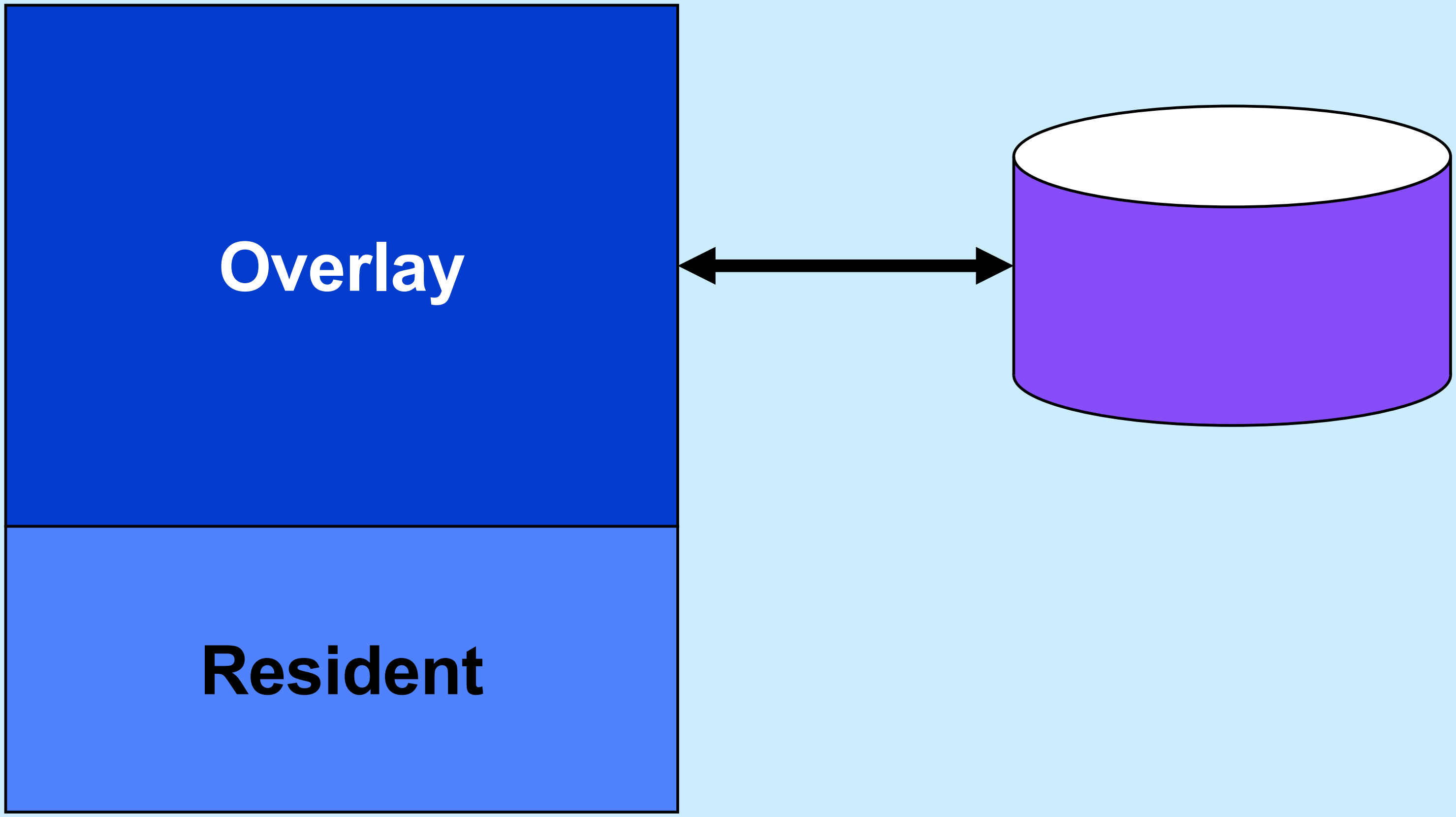
Base and Bounds Registers



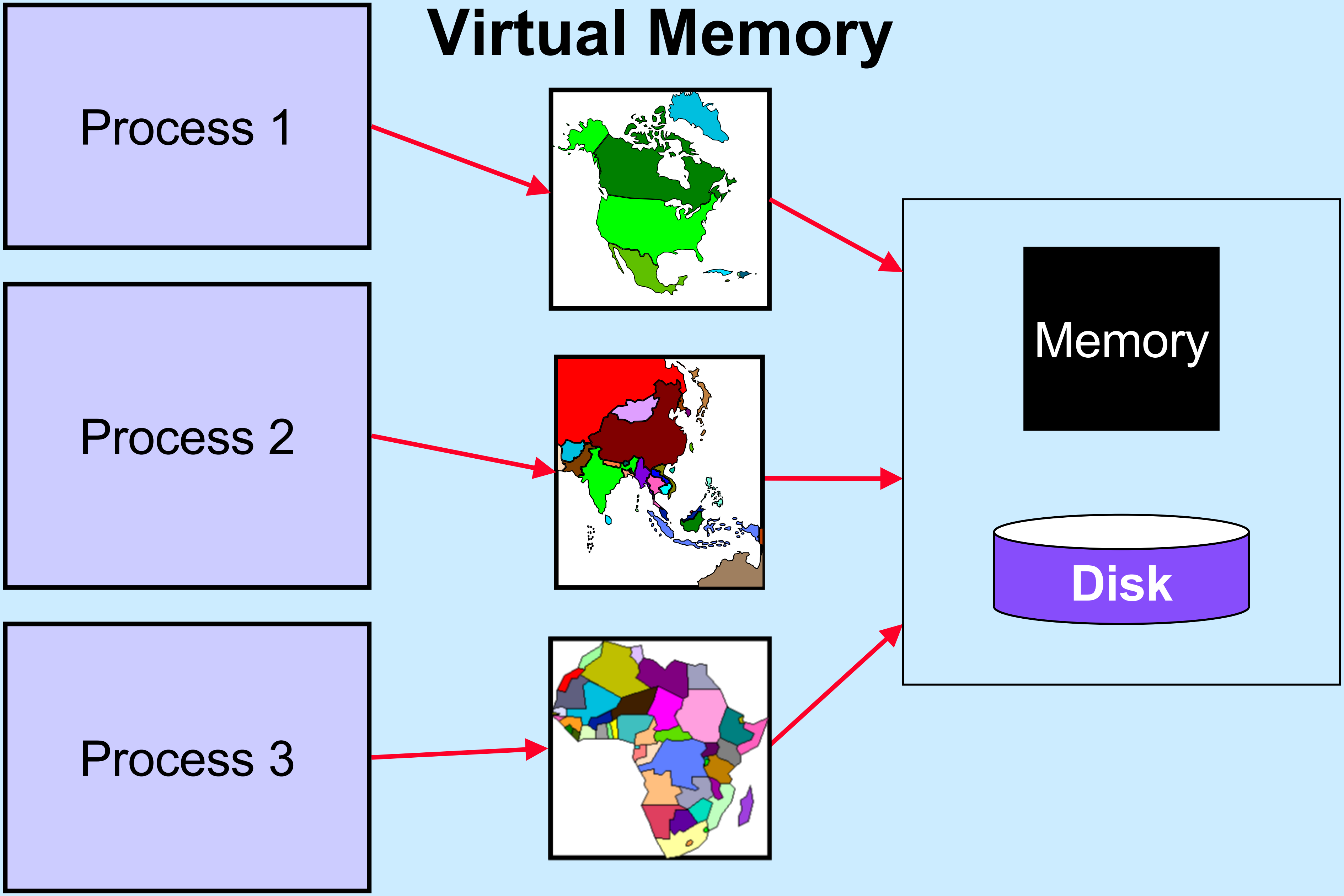
Swapping



Overlays



Virtual Memory



Process 1

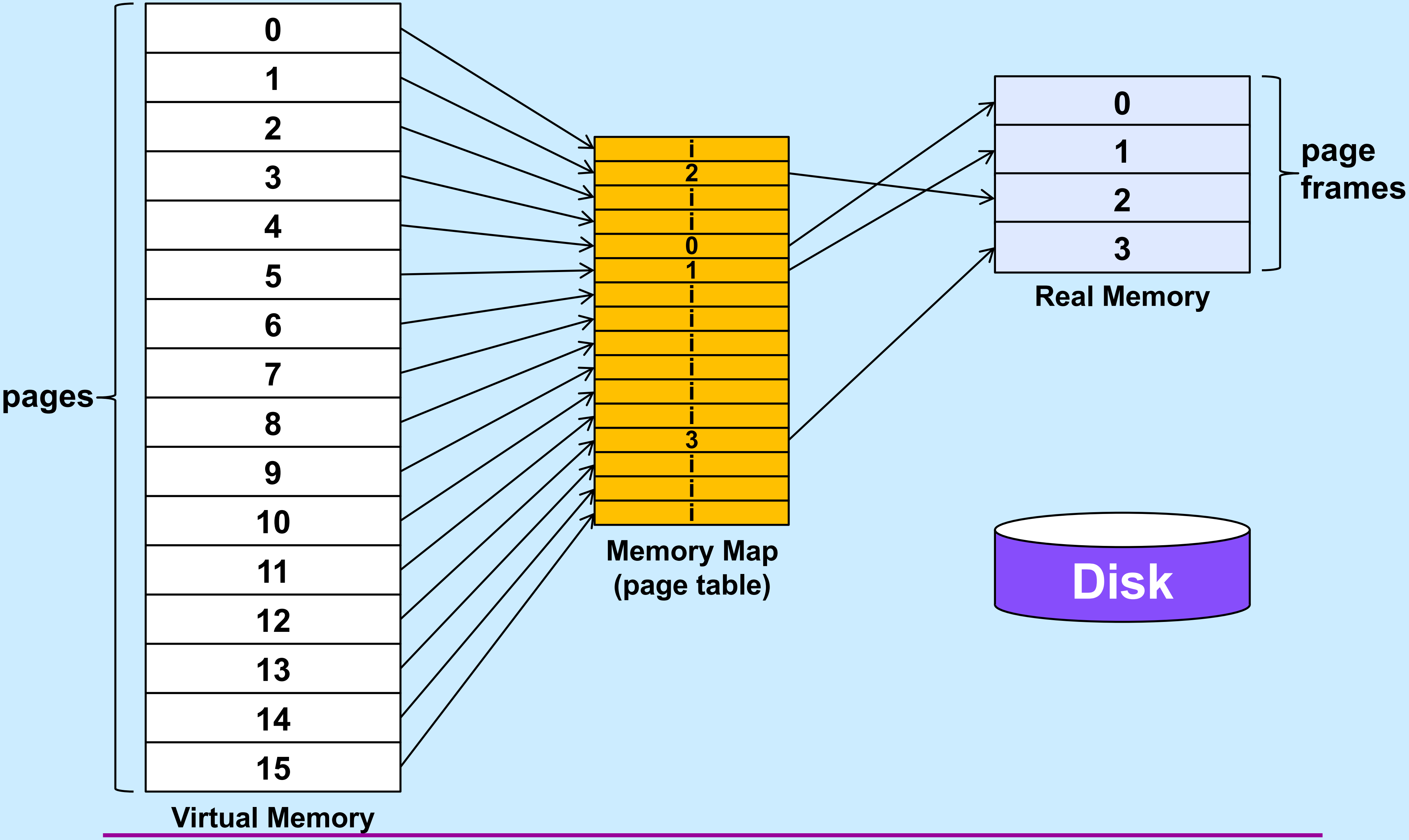
Process 2

Process 3

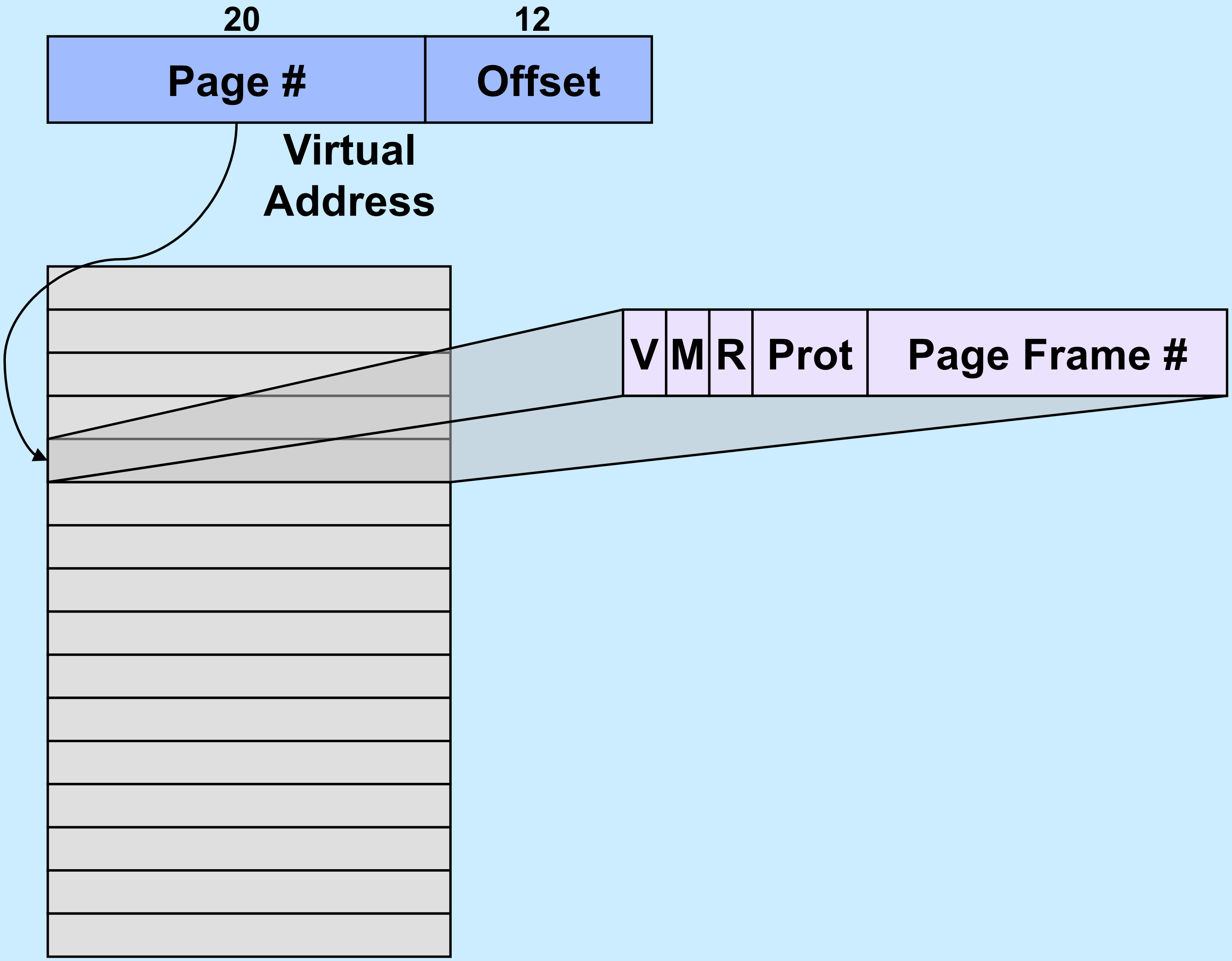
Memory

Disk

Memory Maps


pages
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
Virtual Memory
i
2
i
i
0
1
i
i
i
i
i
i
3
i
i
i
Memory Map
(page table)
0
1
2
3
Real Memory
page
frames
Disk

Page Tables



Quiz 1

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

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

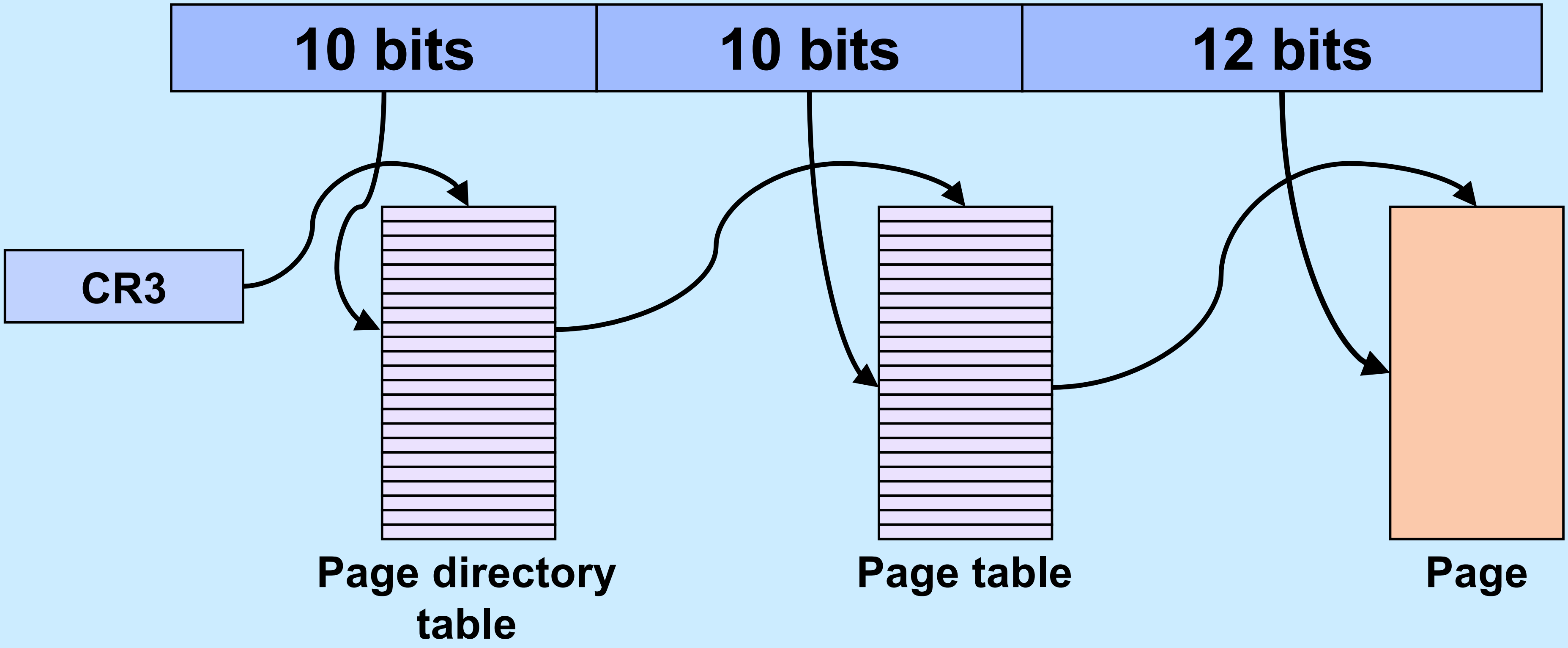
VM is Your Friend ...

- **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


10 bits
10 bits
12 bits
CR3
Page directory table
Page table
Page

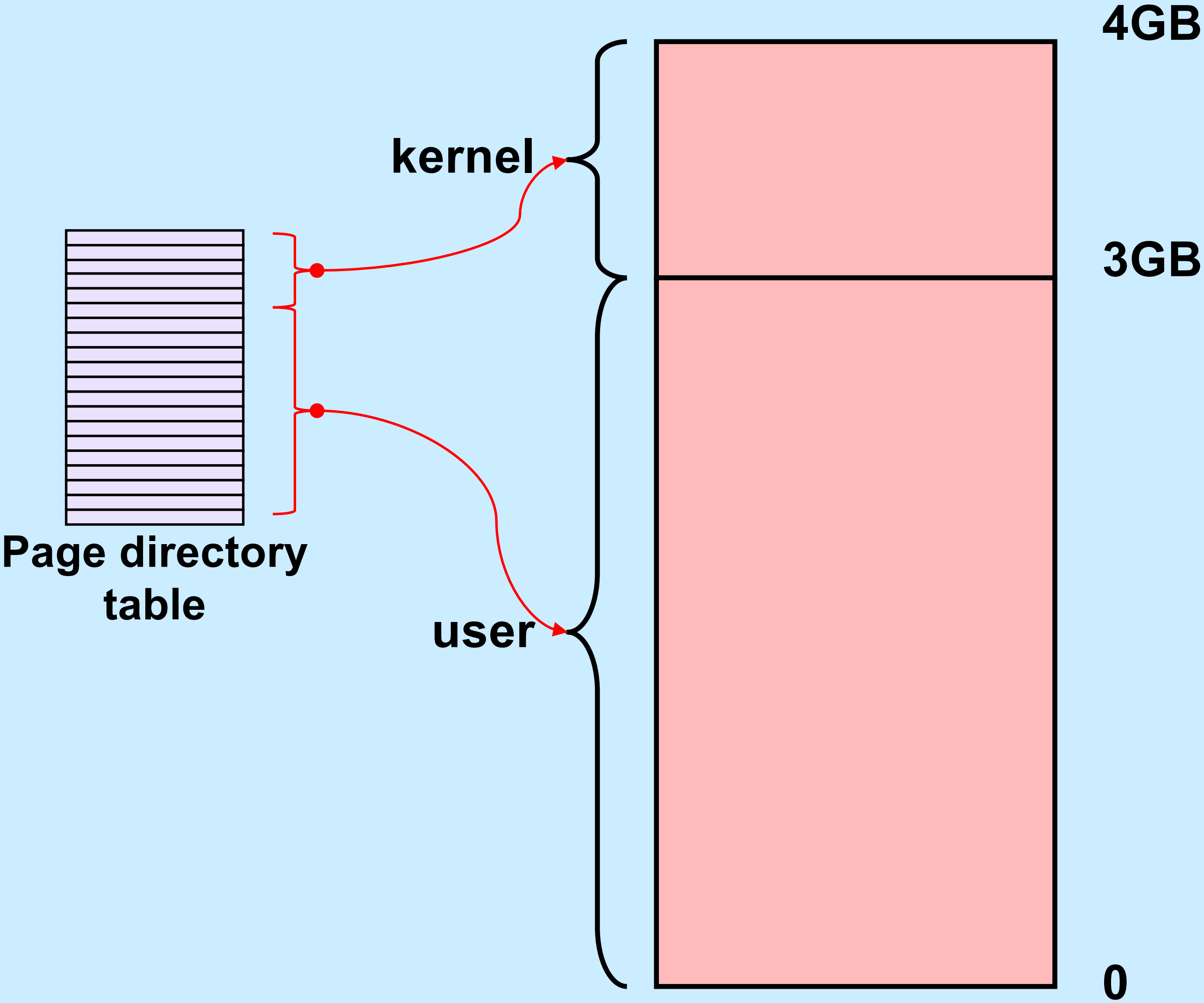
Quiz 2

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

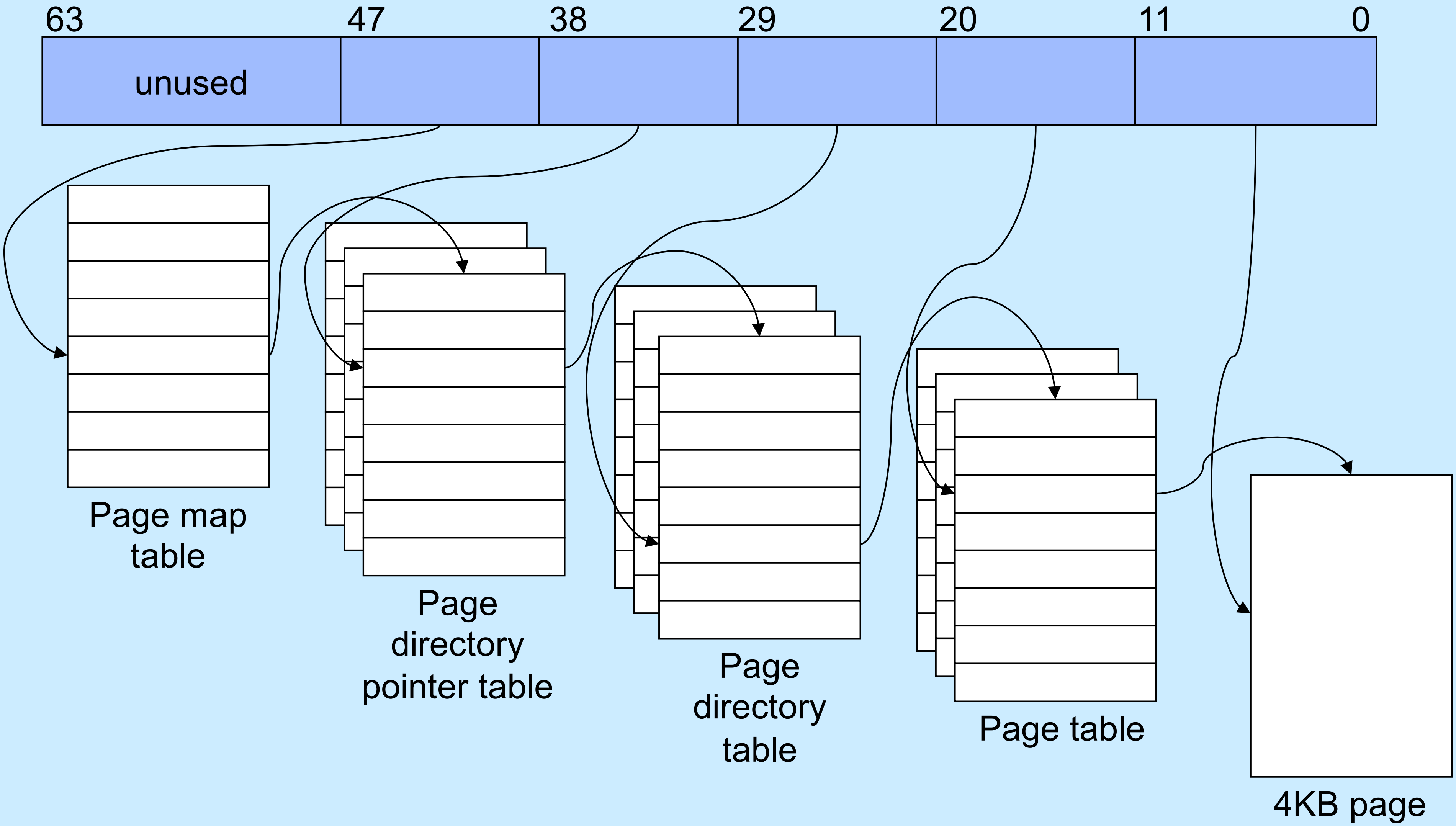
a) yes

b) no

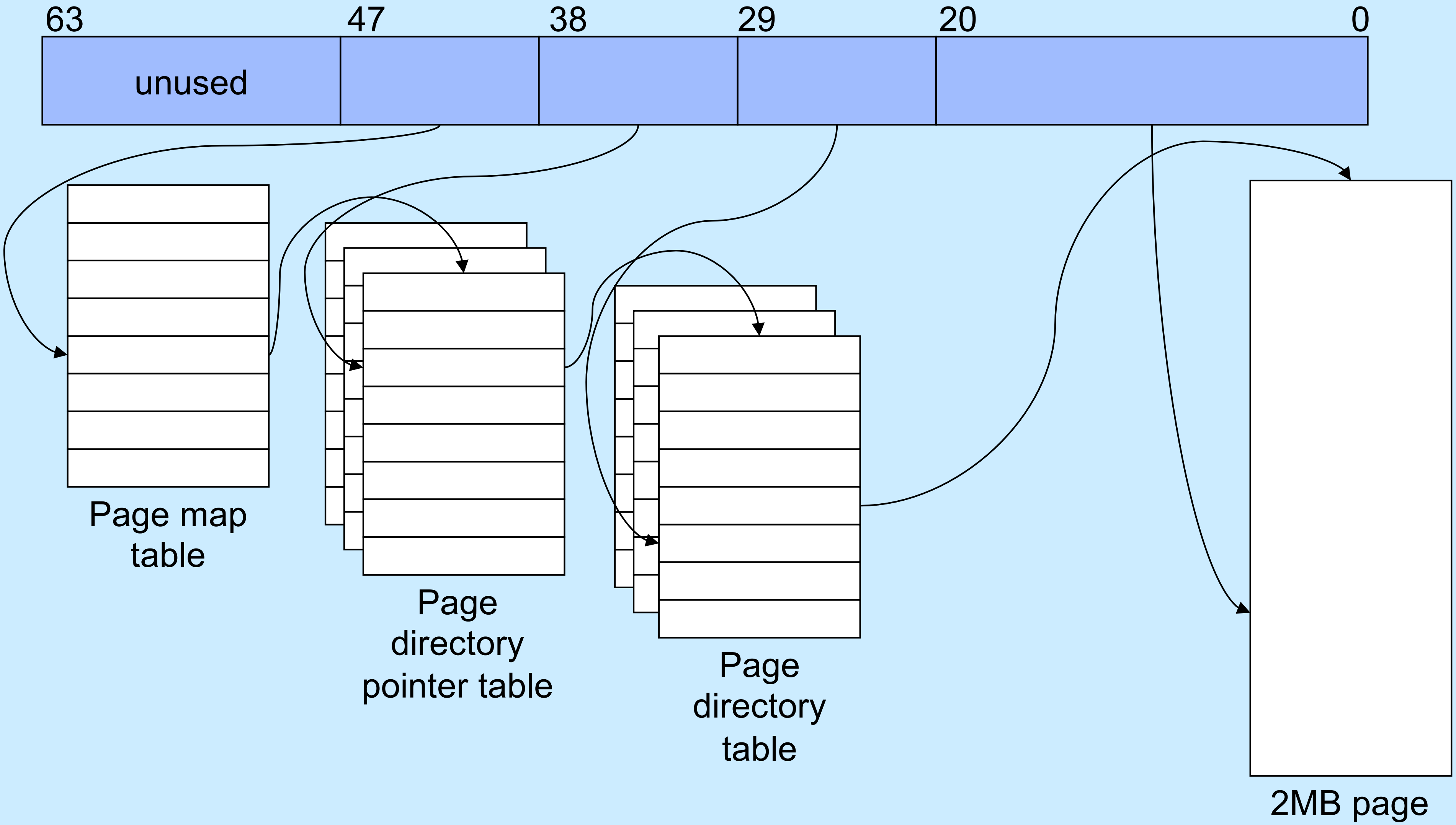
Linux Intel IA32 VM Layout



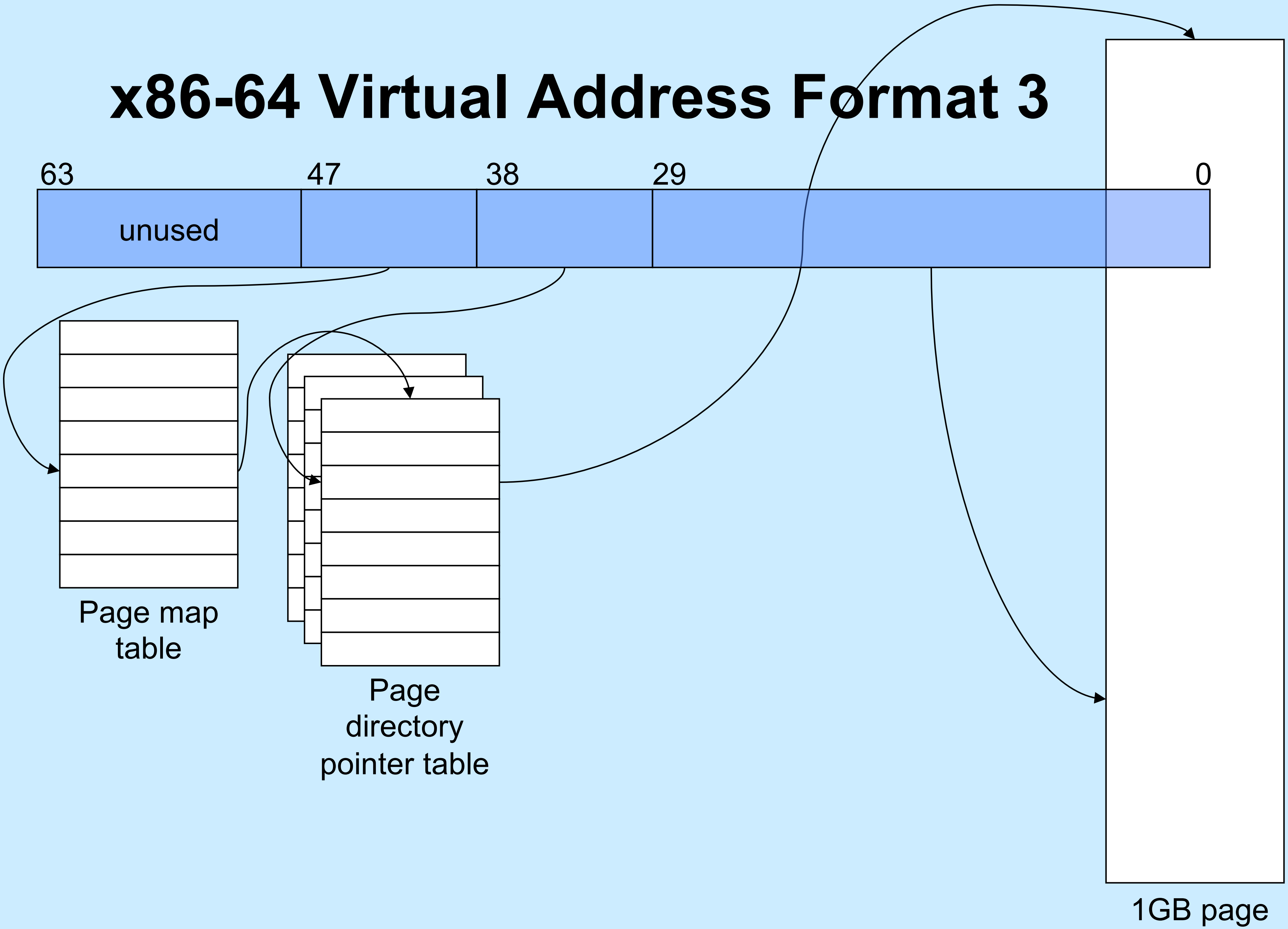
x86-64 Virtual Address Format 1



x86-64 Virtual Address Format 2



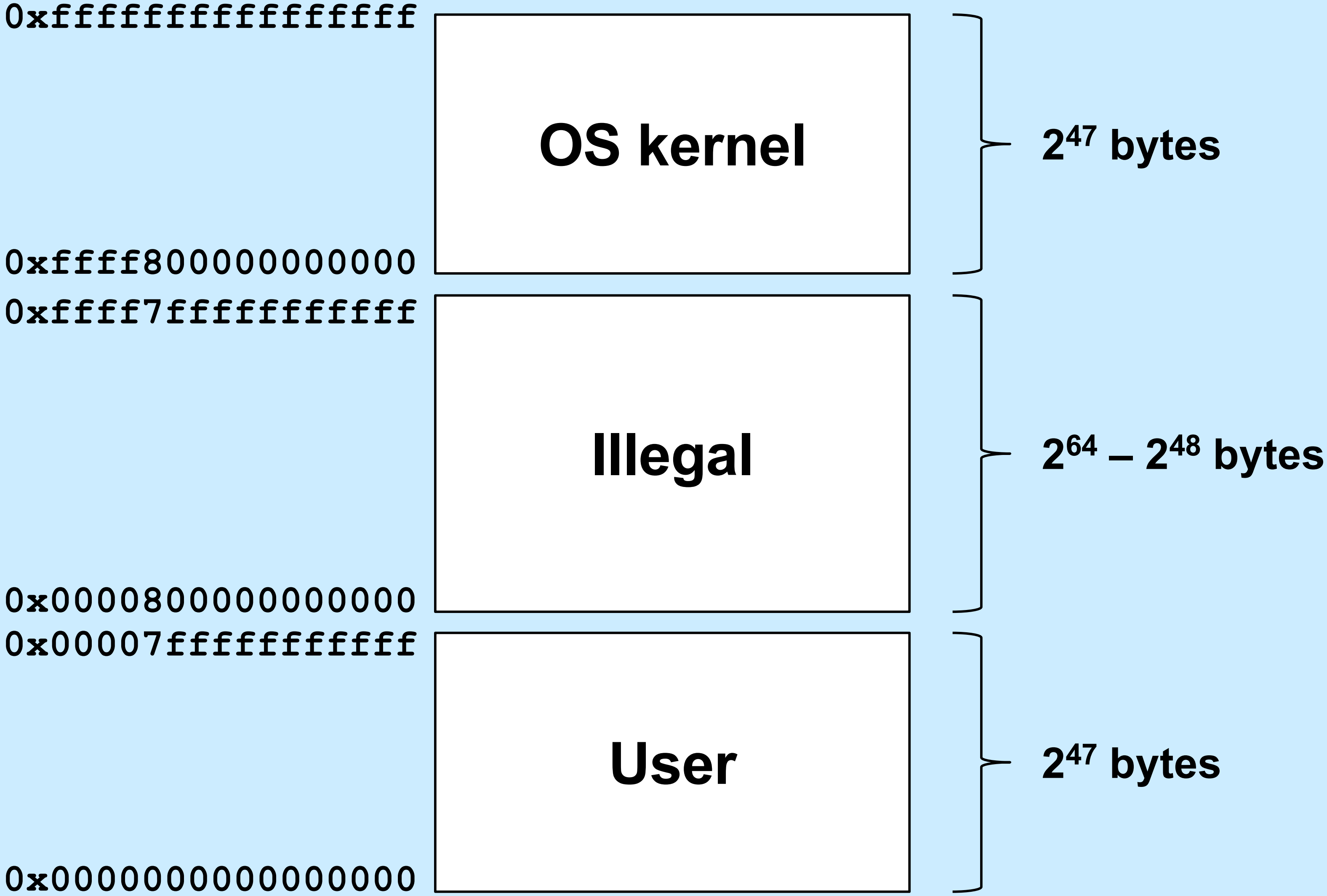
x86-64 Virtual Address Format 3



Why Multiple Page Sizes?

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

x86-64 Address Space



`0xffffffffffffffff`

OS kernel — 2^{47} bytes

`0xffff800000000000`

`0xffff7fffffffffff`

Illegal — $2^{64} – 2^{48}$ bytes

`0x0000800000000000`

`0x00007fffffffffff`

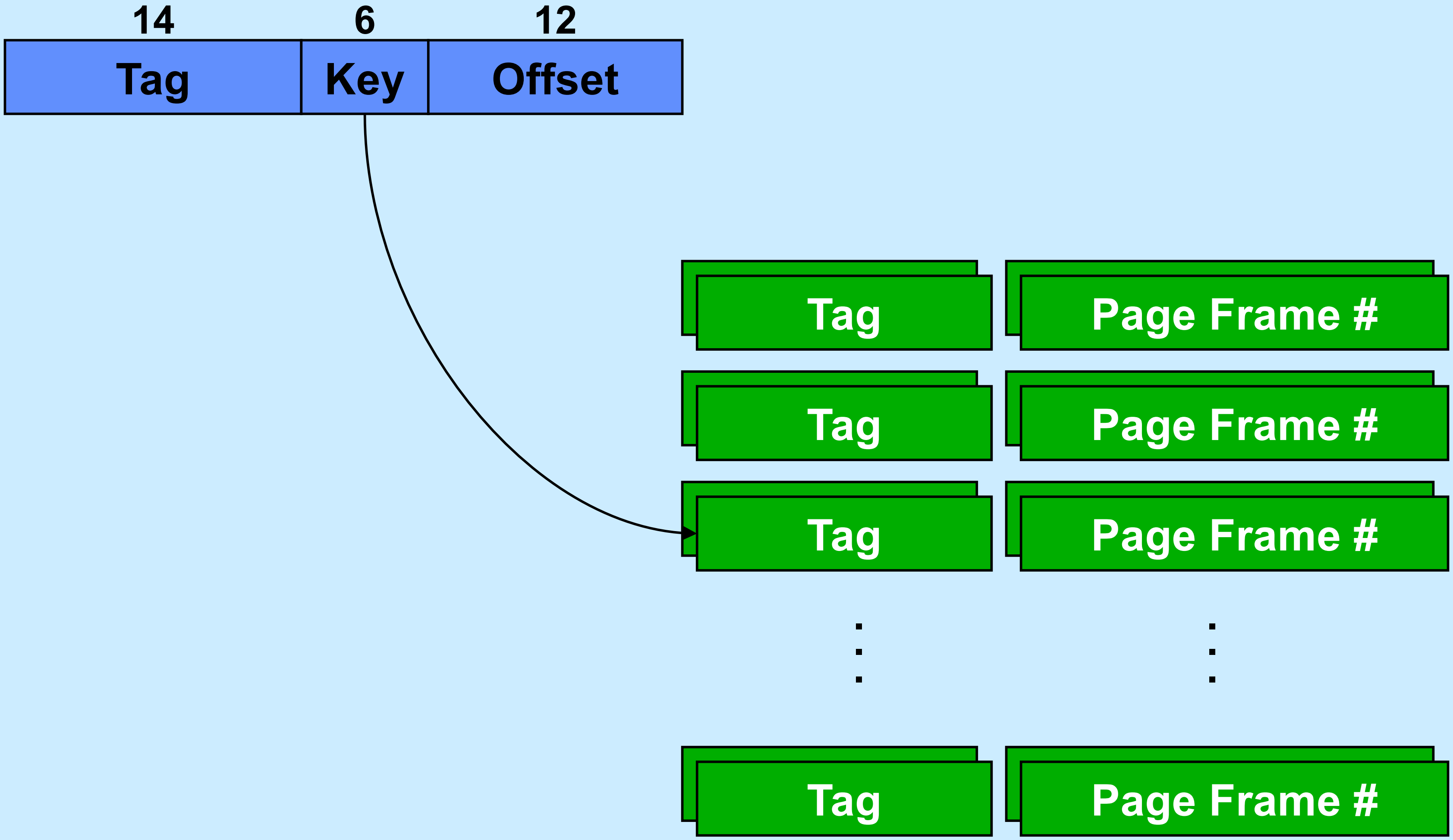
User — 2^{47} bytes

`0x0000000000000000`

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) .5x (i.e. it will be faster, not slower)
b) 1x
c) 2x
d) 3x
e) 4x

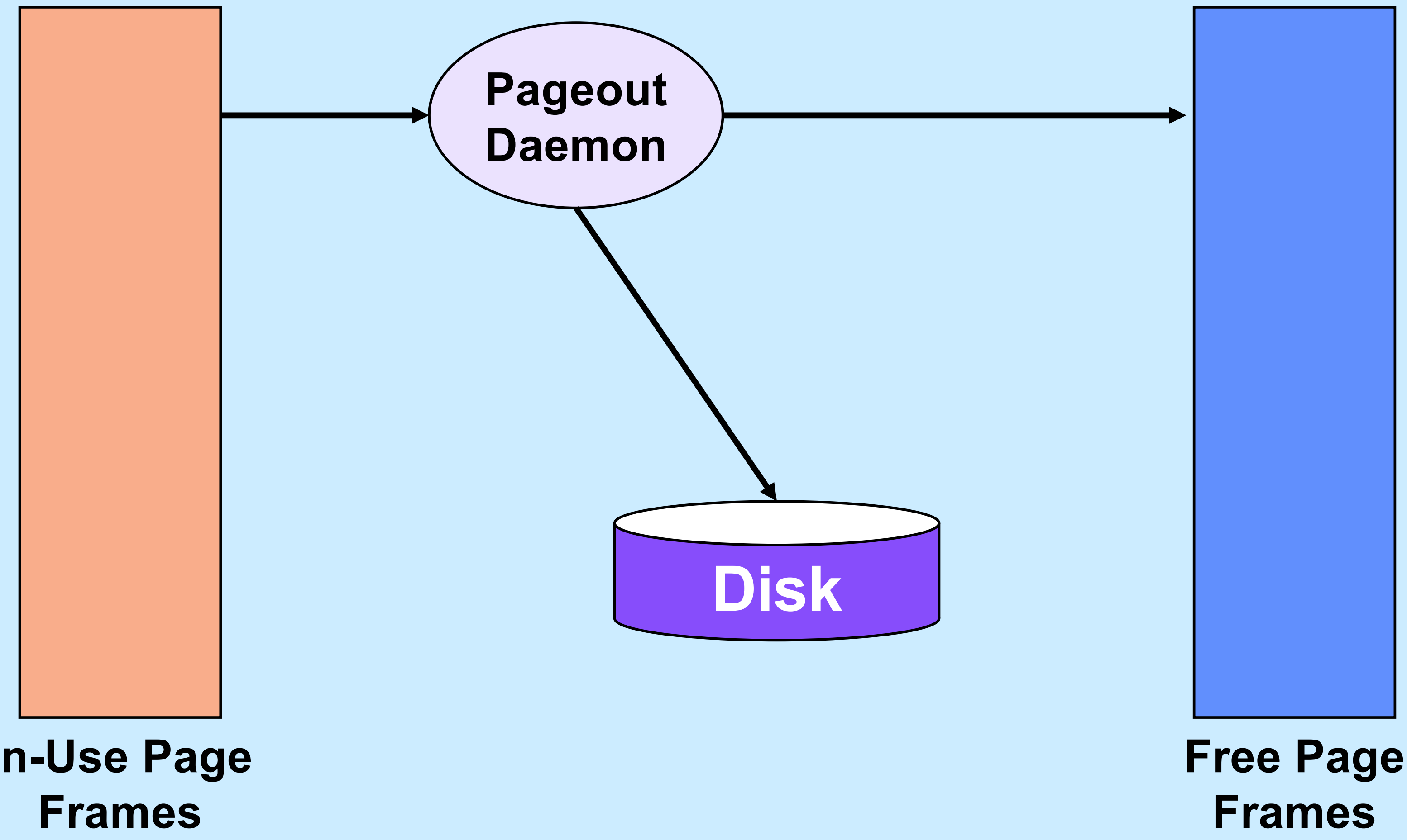
OS Role in Virtual Memory

- 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

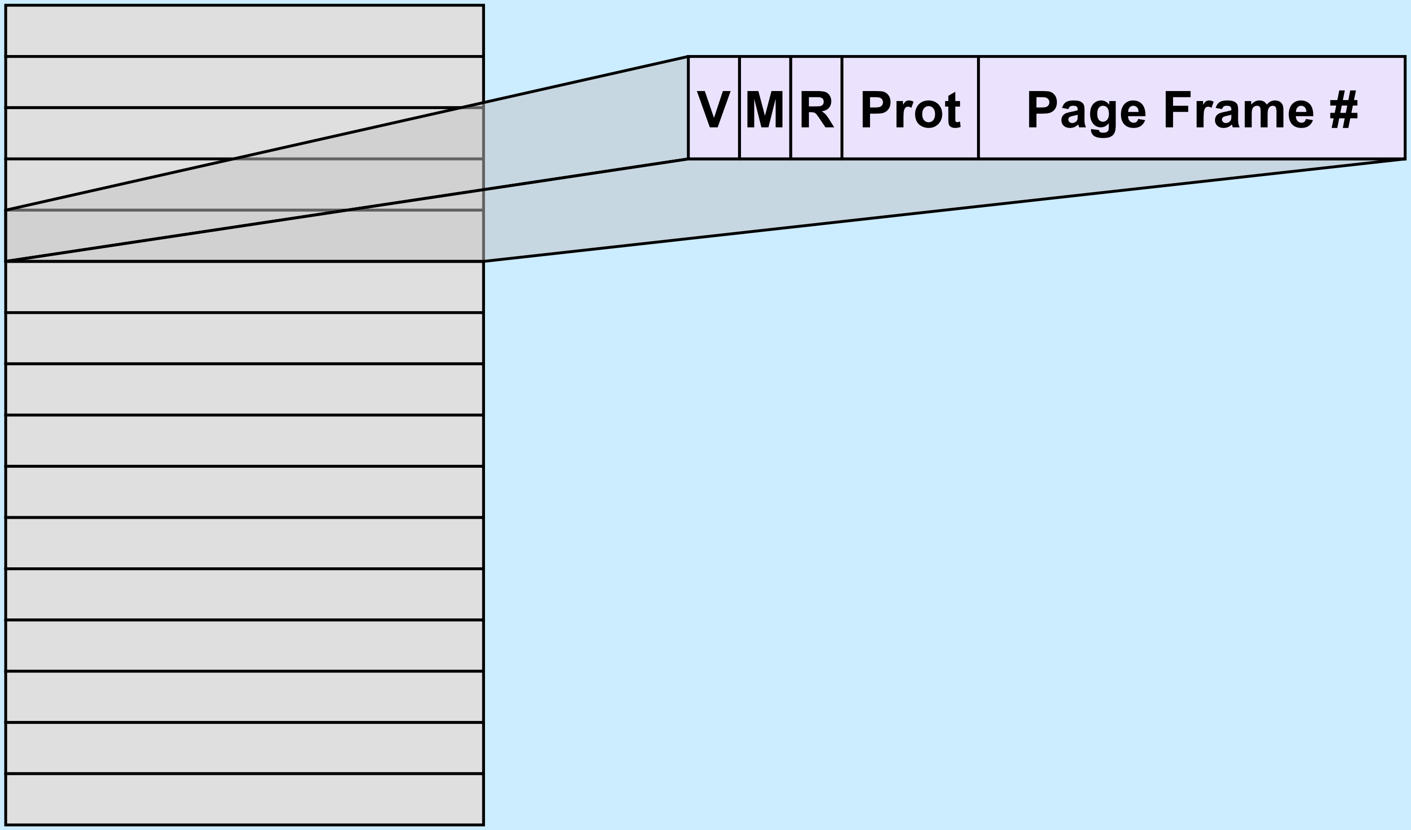
Mechanism

- 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

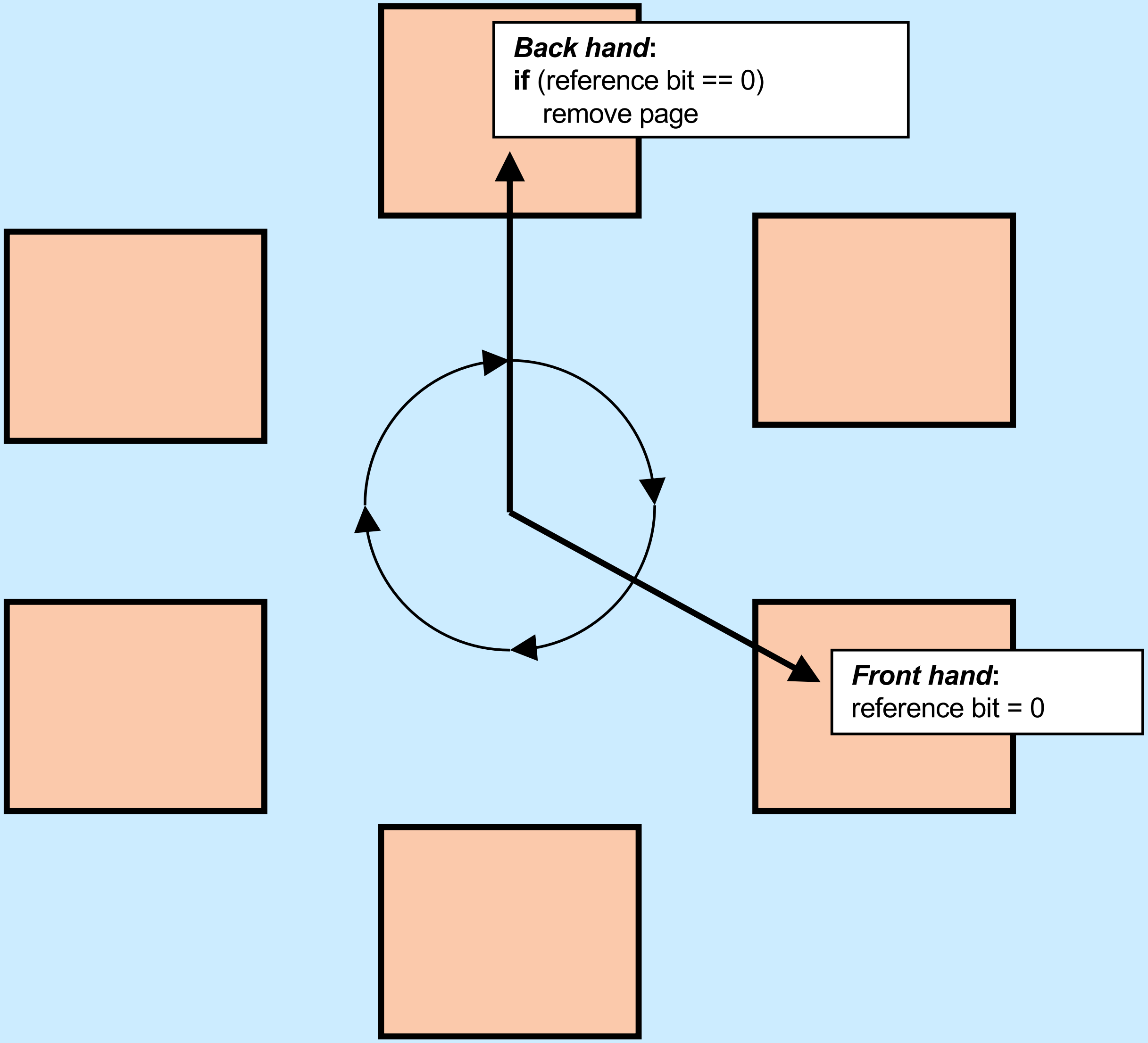
The “Pageout Daemon”



Managing Page Frames



Clock Algorithm

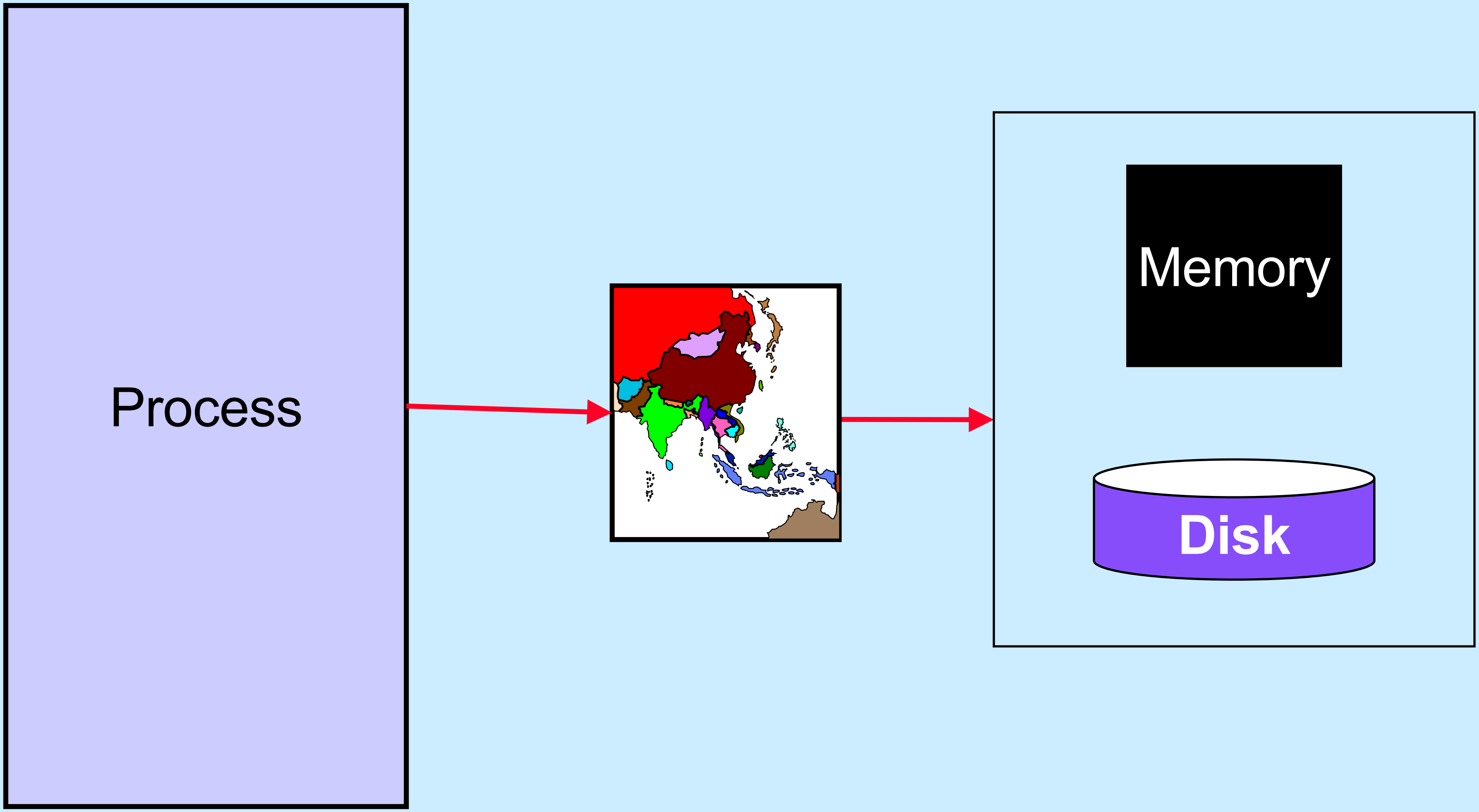


***Back hand*:**
if (reference bit == 0)
remove page

***Front hand*:**
reference bit = 0

Why is virtual memory used?

More VM than RM

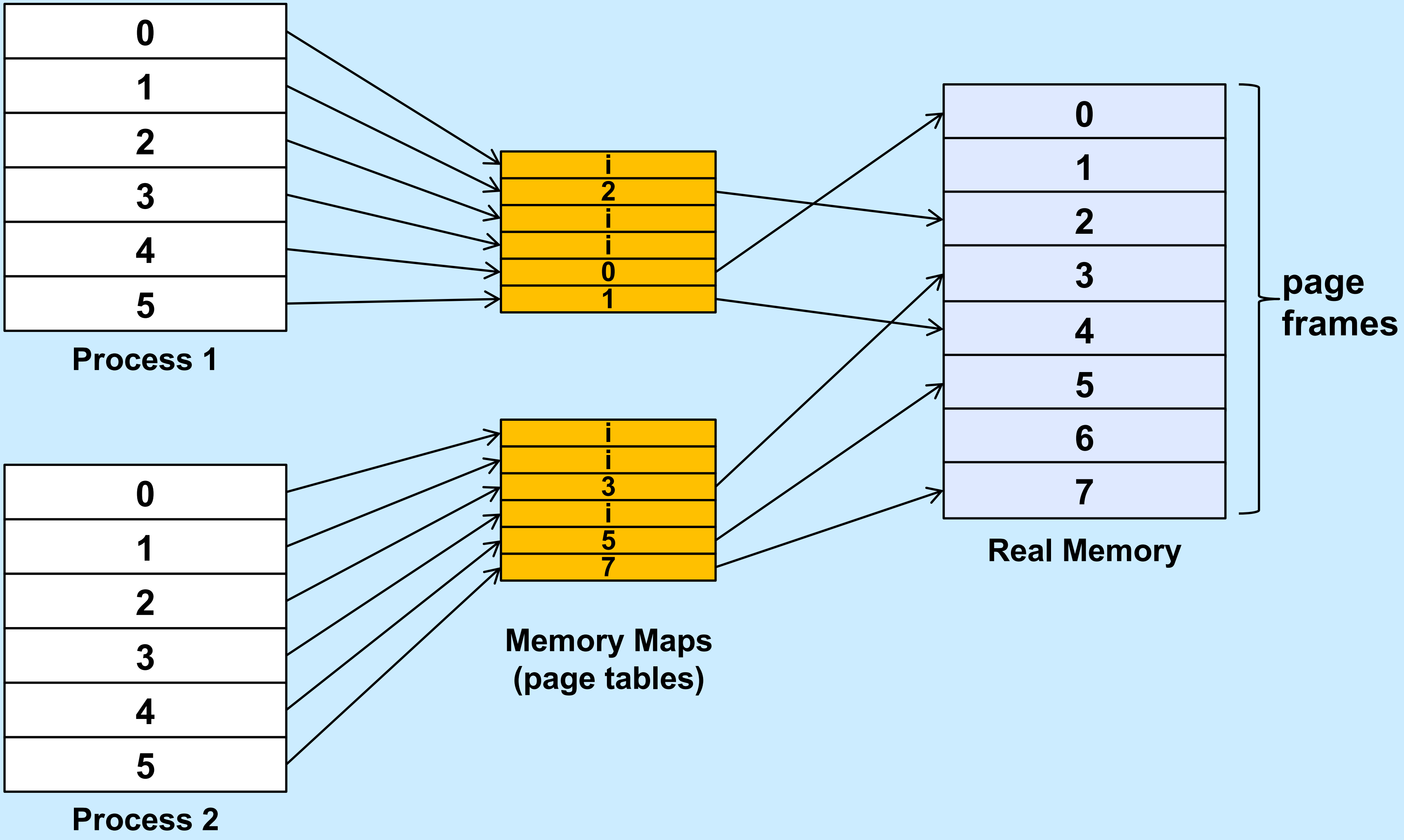


Process

Memory

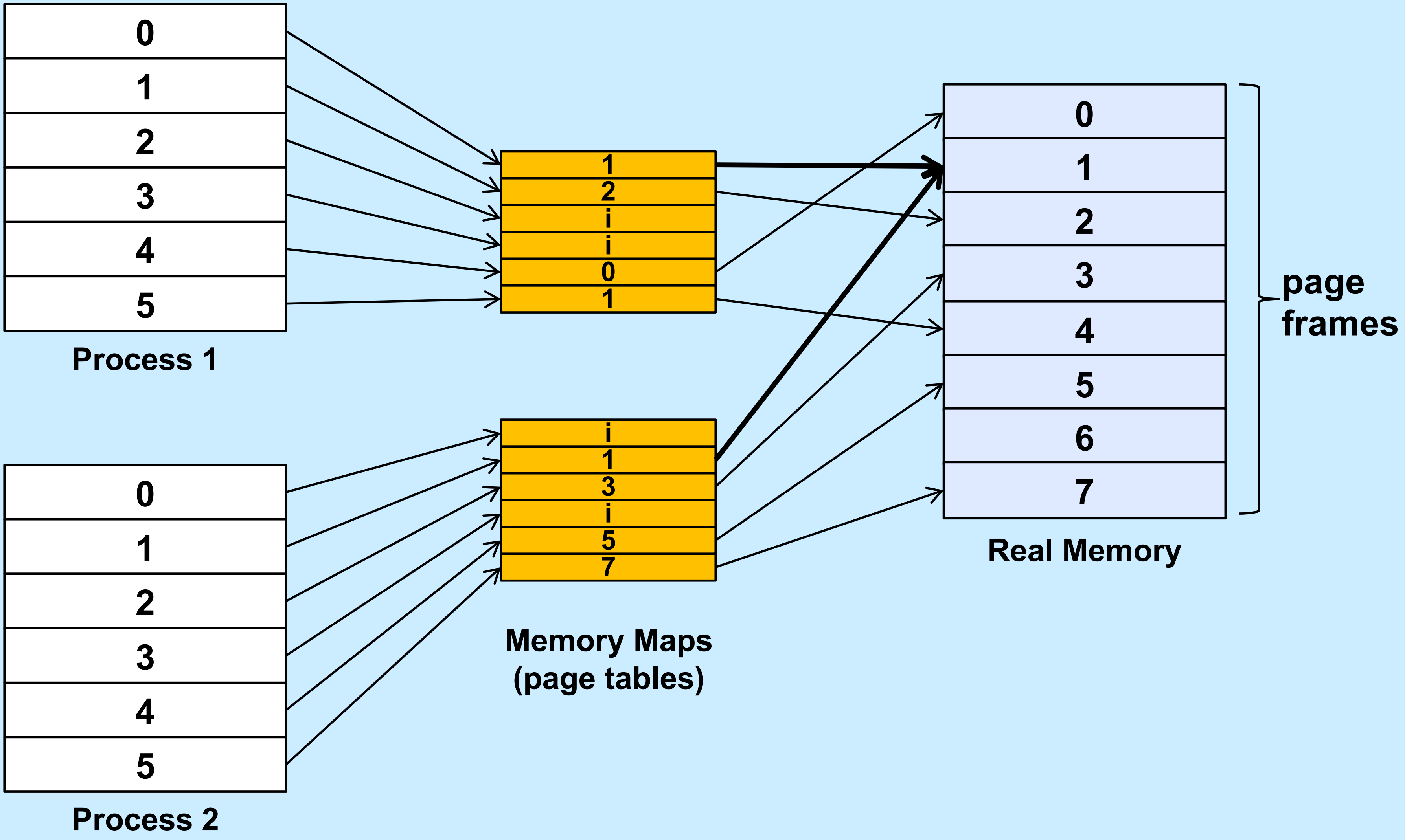
Disk

Isolation



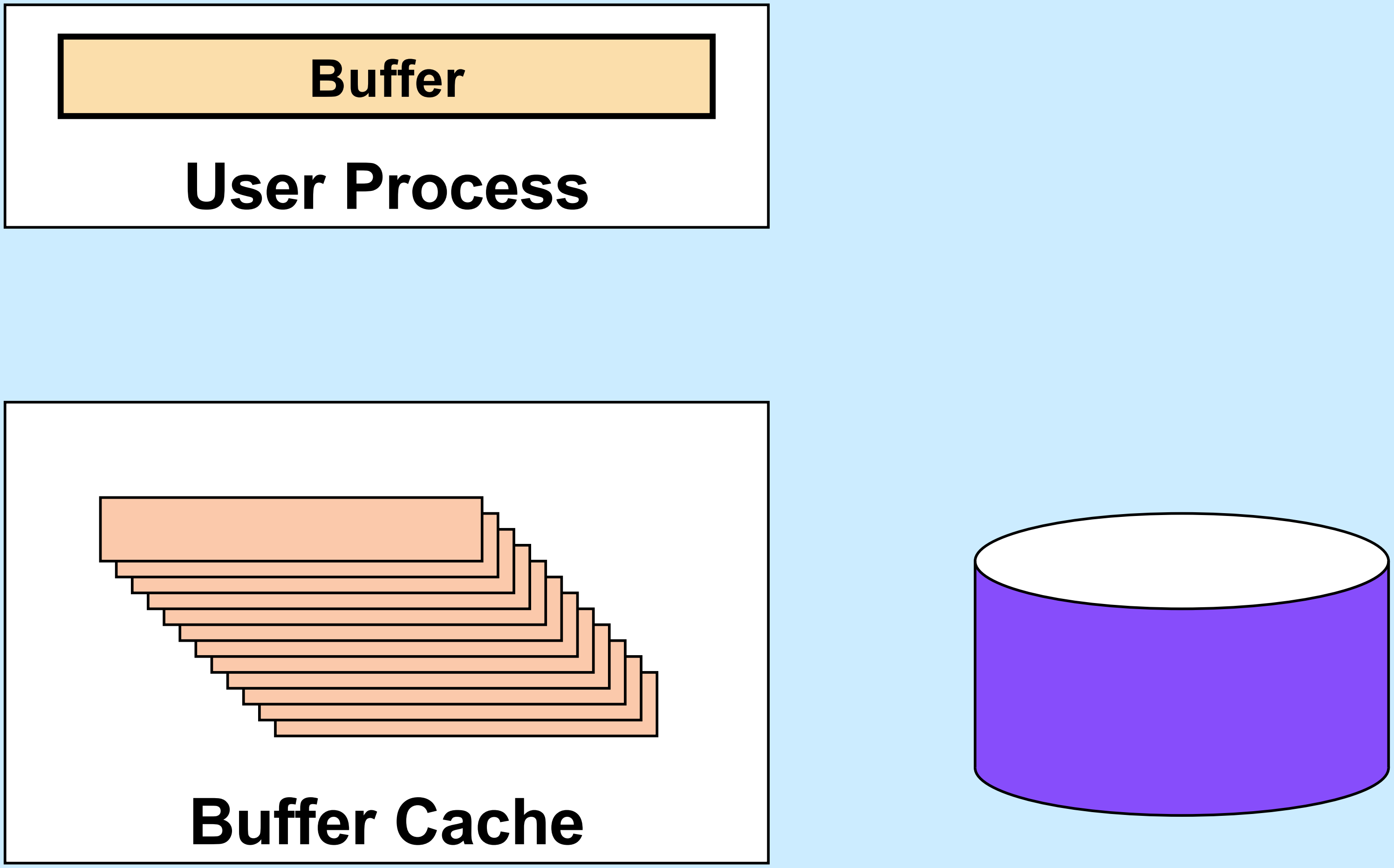
Virtual Memory

Sharing



Virtual Memory

File I/O

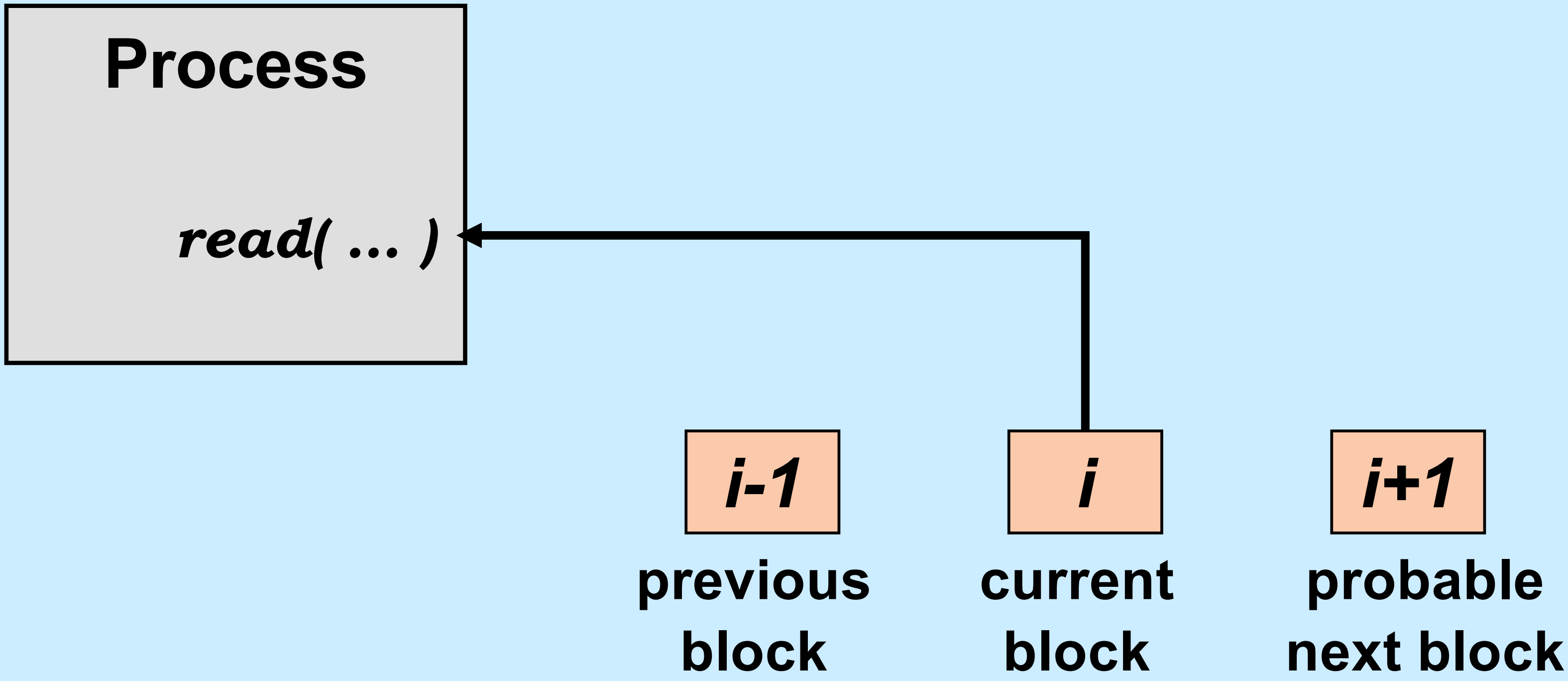


Buffer

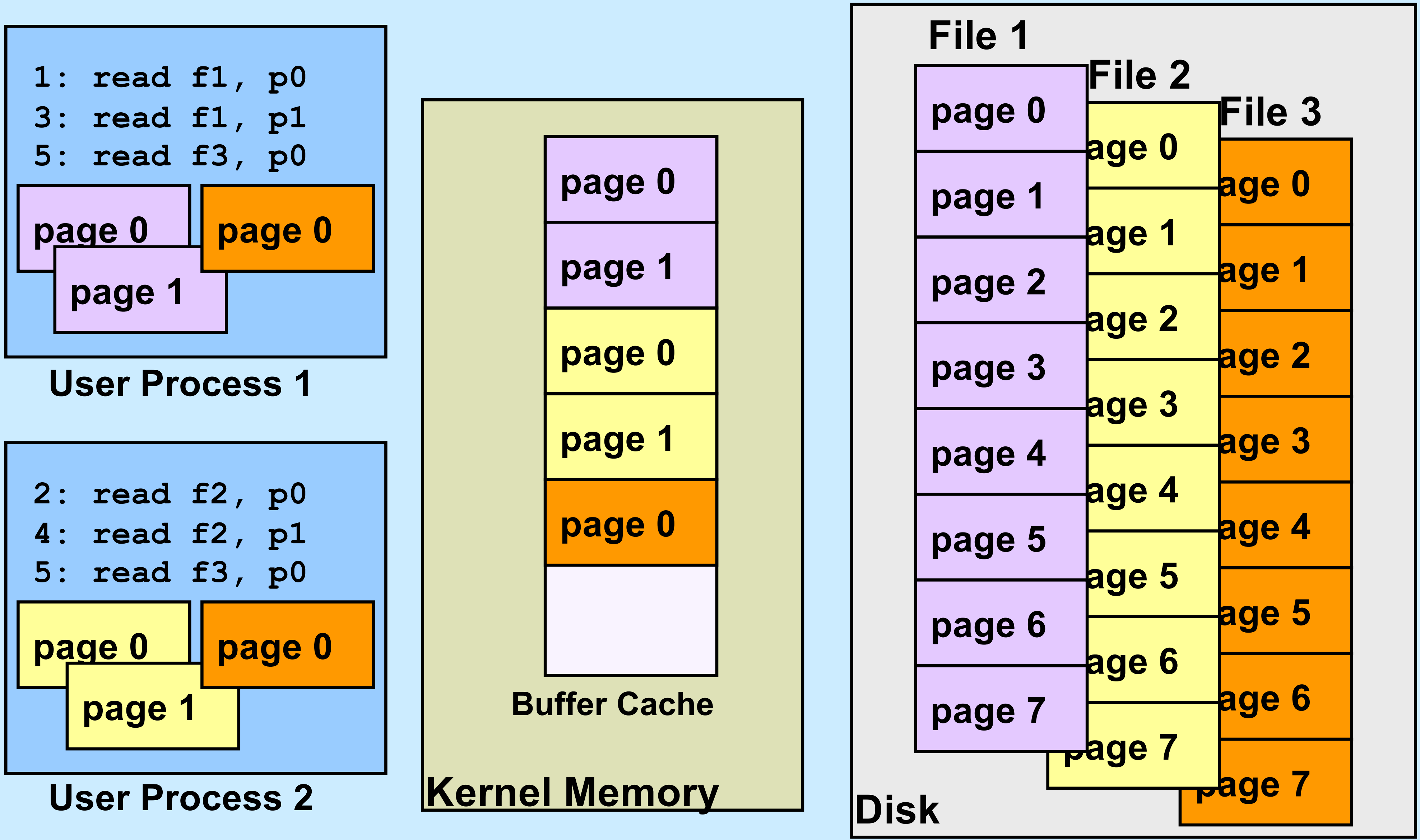
User Process

Buffer Cache

Multi-Buffered I/O



Traditional I/O



```
1: read f1, p0
3: read f1, p1
5: read f3, p0
```

page 0
page 1
page 0

User Process 1

```
2: read f2, p0
4: read f2, p1
5: read f3, p0
```

page 0
page 1
page 0

User Process 2

page 0
page 1
page 0
page 1
page 0

Buffer Cache

Kernel Memory

File 1

page 0
page 1
page 2
page 3
page 4
page 5
page 6
page 7

File 2

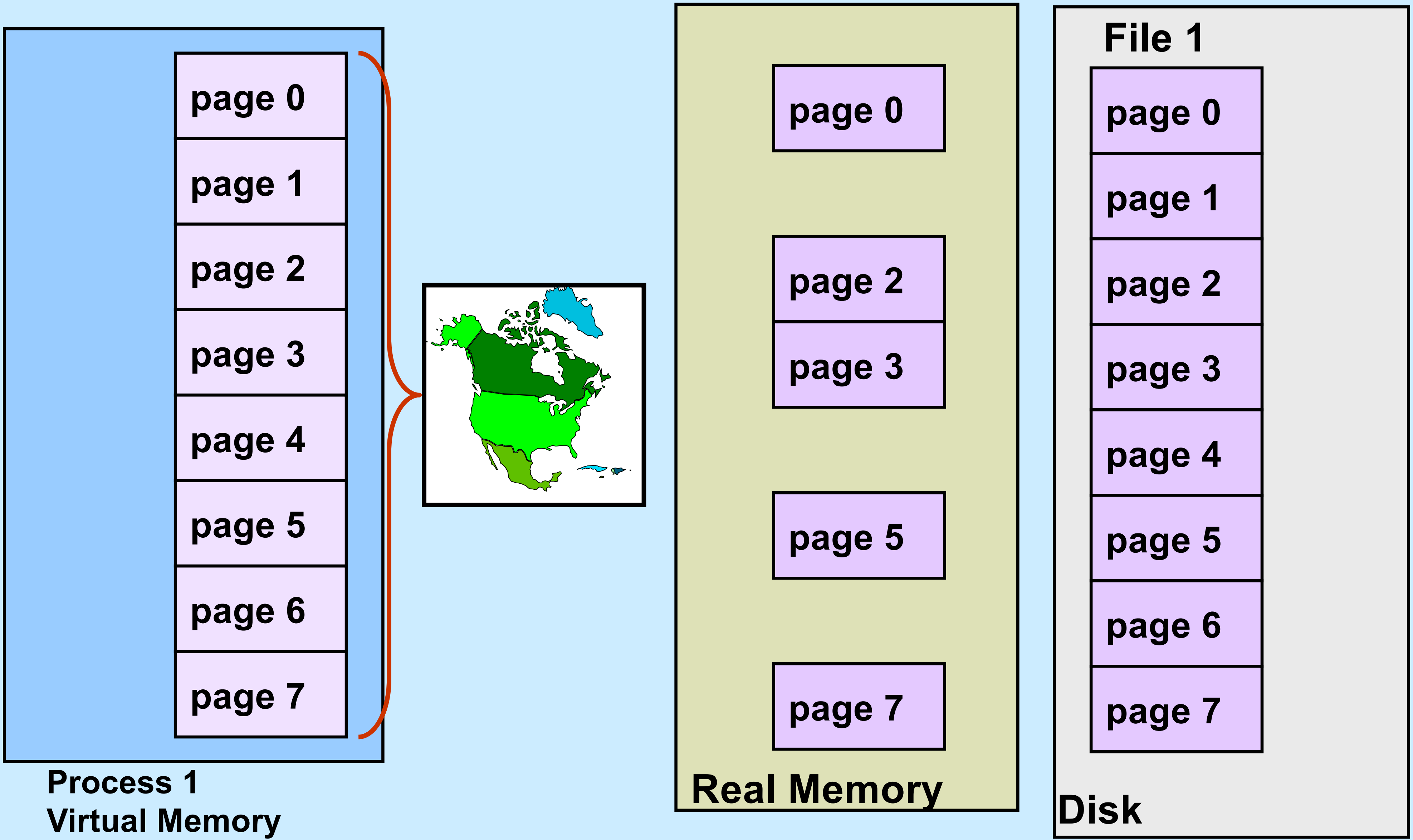
age 0
age 1
age 2
age 3
age 4
age 5
age 6
page 7

File 3

age 0
age 1
age 2
age 3
age 4
age 5
age 6
page 7

Disk

Mapped File I/O



page 0
page 1
page 2
page 3
page 4
page 5
page 6
page 7

Process 1
Virtual Memory

page 0
page 2
page 3
page 5
page 7

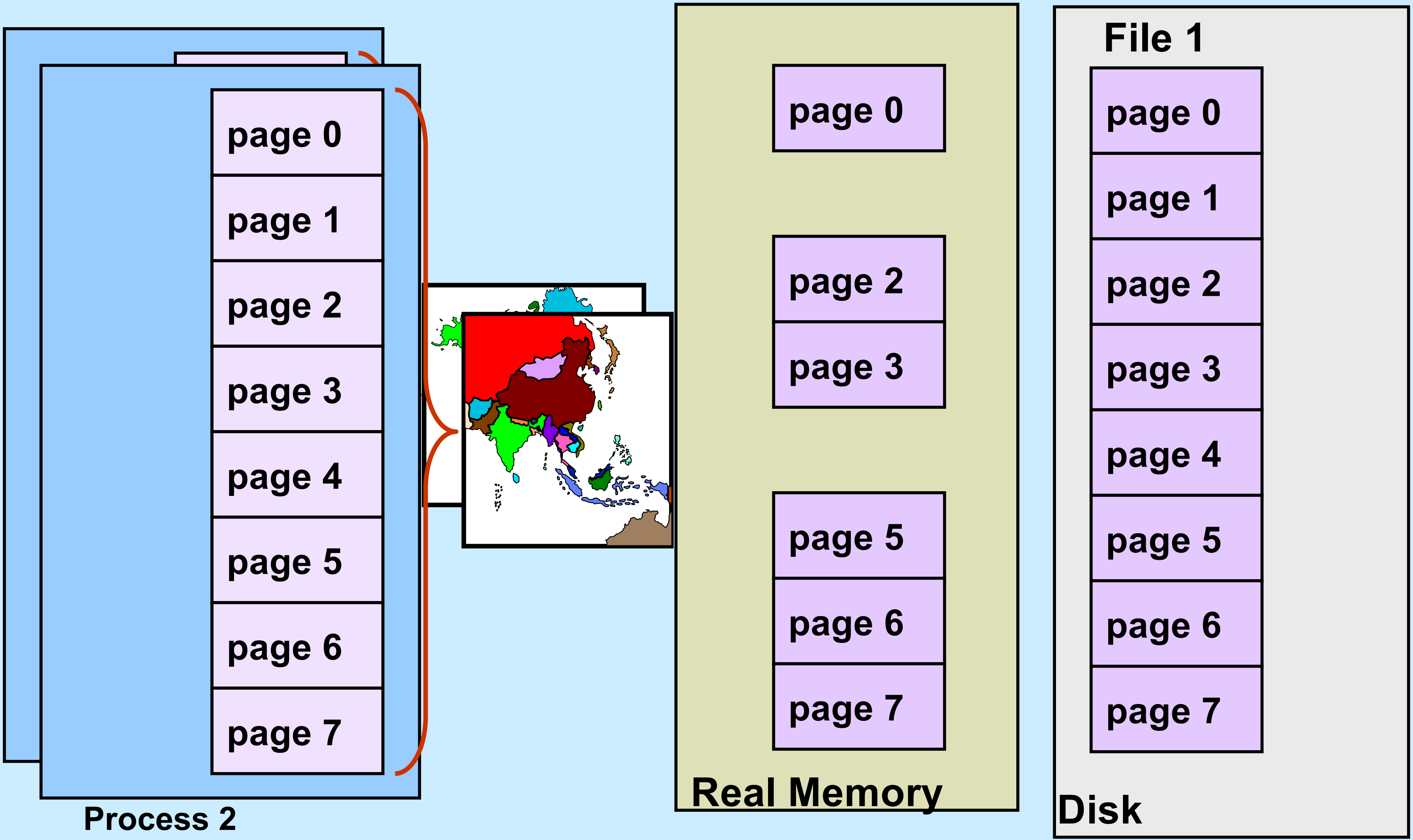
Real Memory

File 1

page 0
page 1
page 2
page 3
page 4
page 5
page 6
page 7

Disk

Multi-Process Mapped File I/O



page 0
page 1
page 2
page 3
page 4
page 5
page 6
page 7

Process 2
Virtual Memory

page 0
page 2
page 3
page 5
page 6
page 7

Real Memory

File 1

page 0
page 1
page 2
page 3
page 4
page 5
page 6
page 7

Disk

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);
}
```

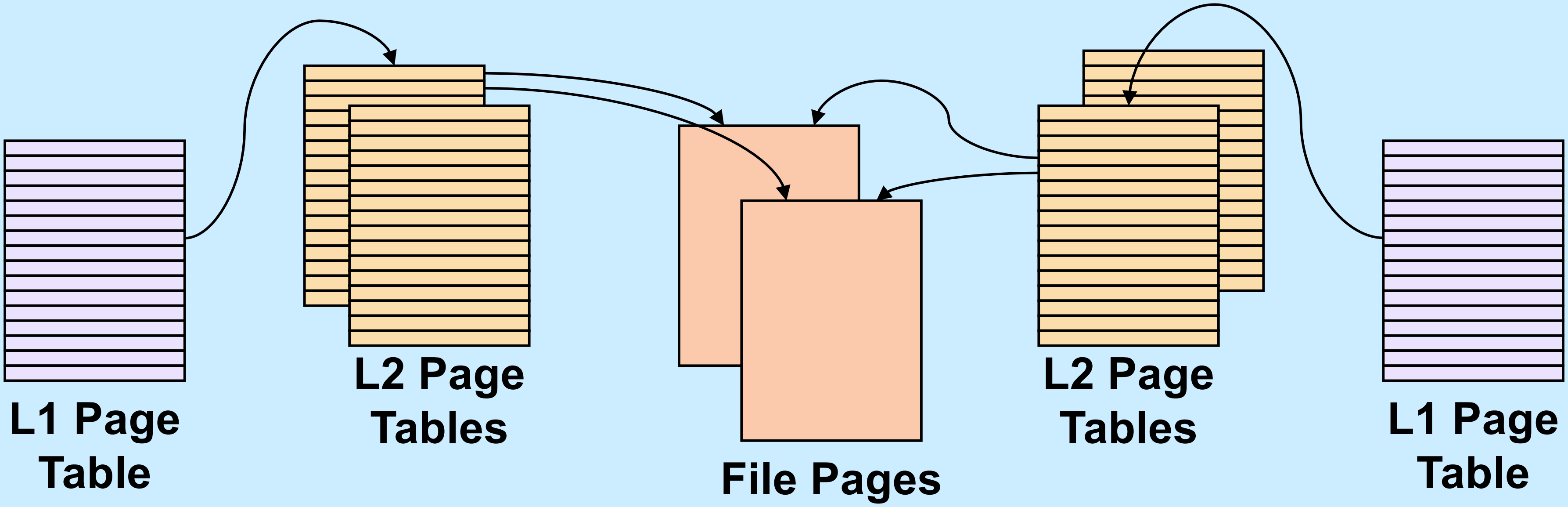
- **Mapped File I/O**

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

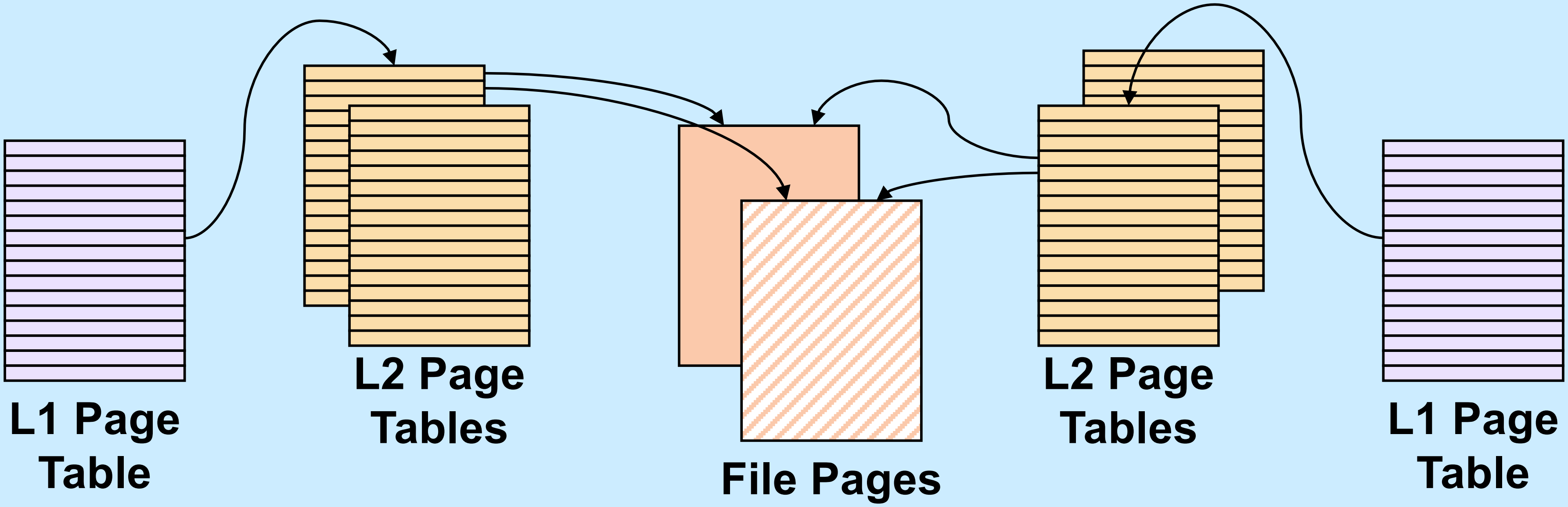
Mmap System Call

```
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

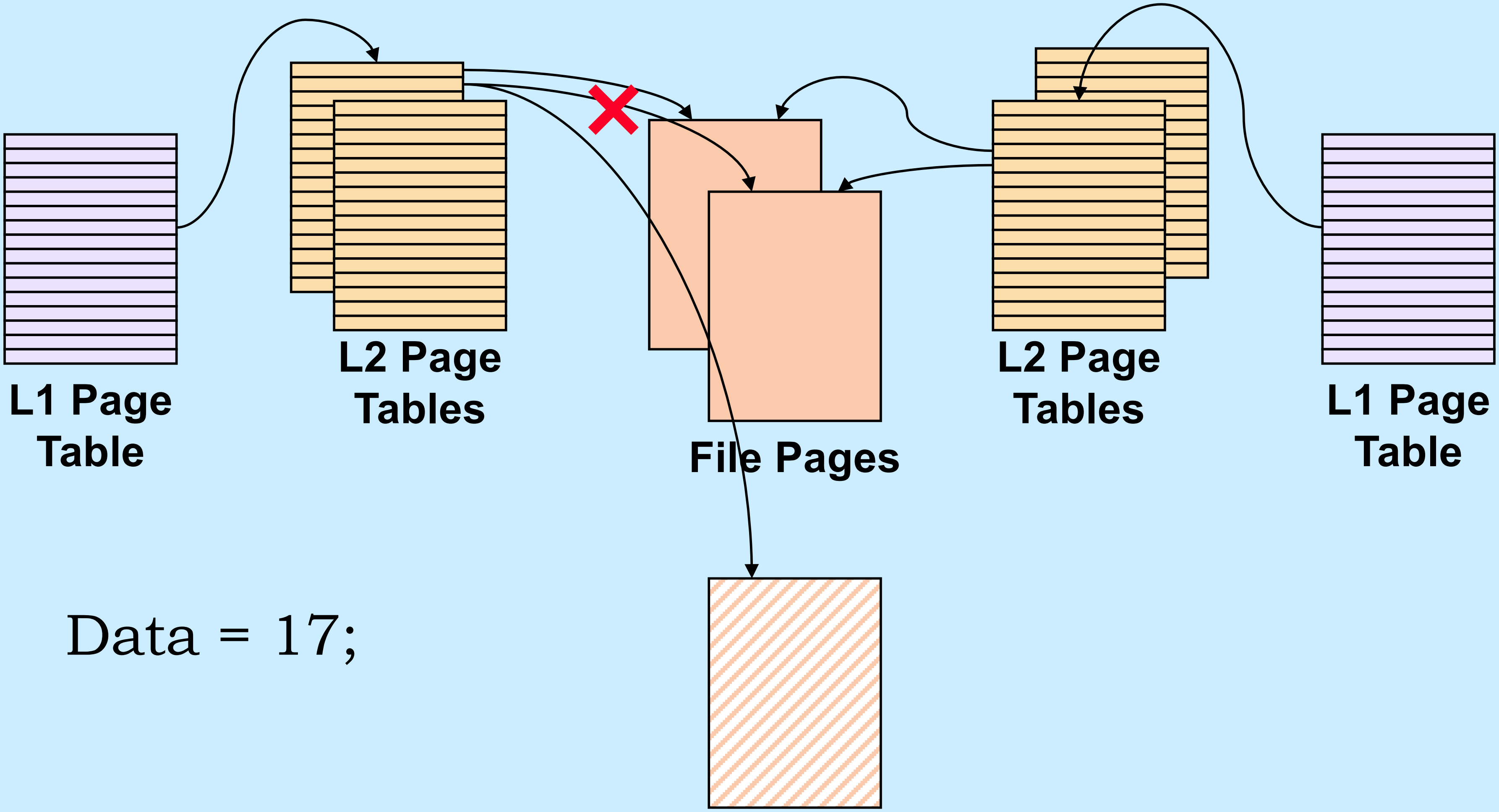


Share-Mapped Files



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;
}
```