Paging

CSE 4001

Content

- Basic paging mechanism
- Limitations
- Protection
- Shared pages

Paging

Basic problem with allocating contiguous blocks of memory for processes

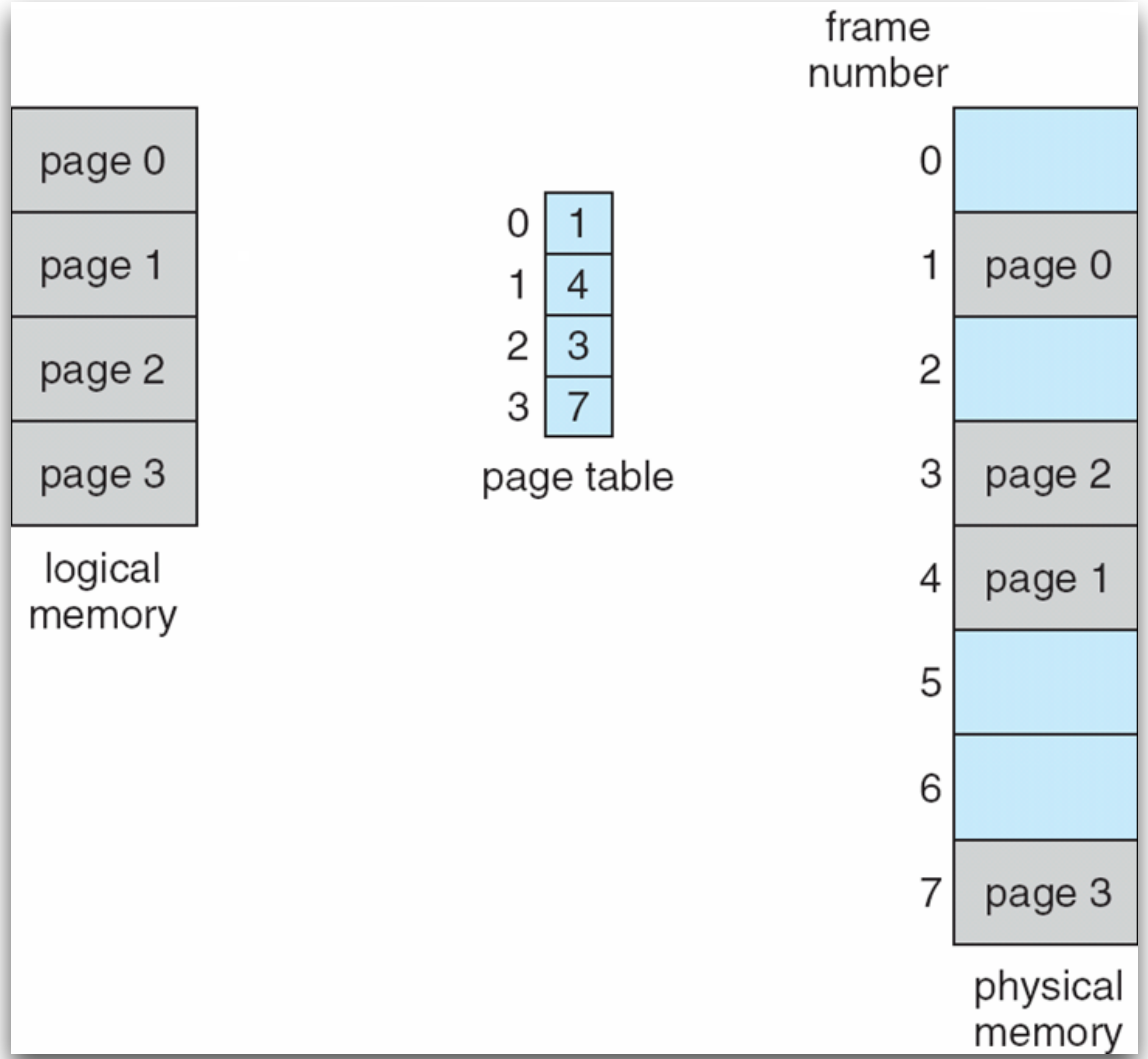
- Determining the size of memory blocks is difficult because different processes have different memory requirements.

Paging: physical address space is allowed to be non-contiguous

Paging

Basic paging method

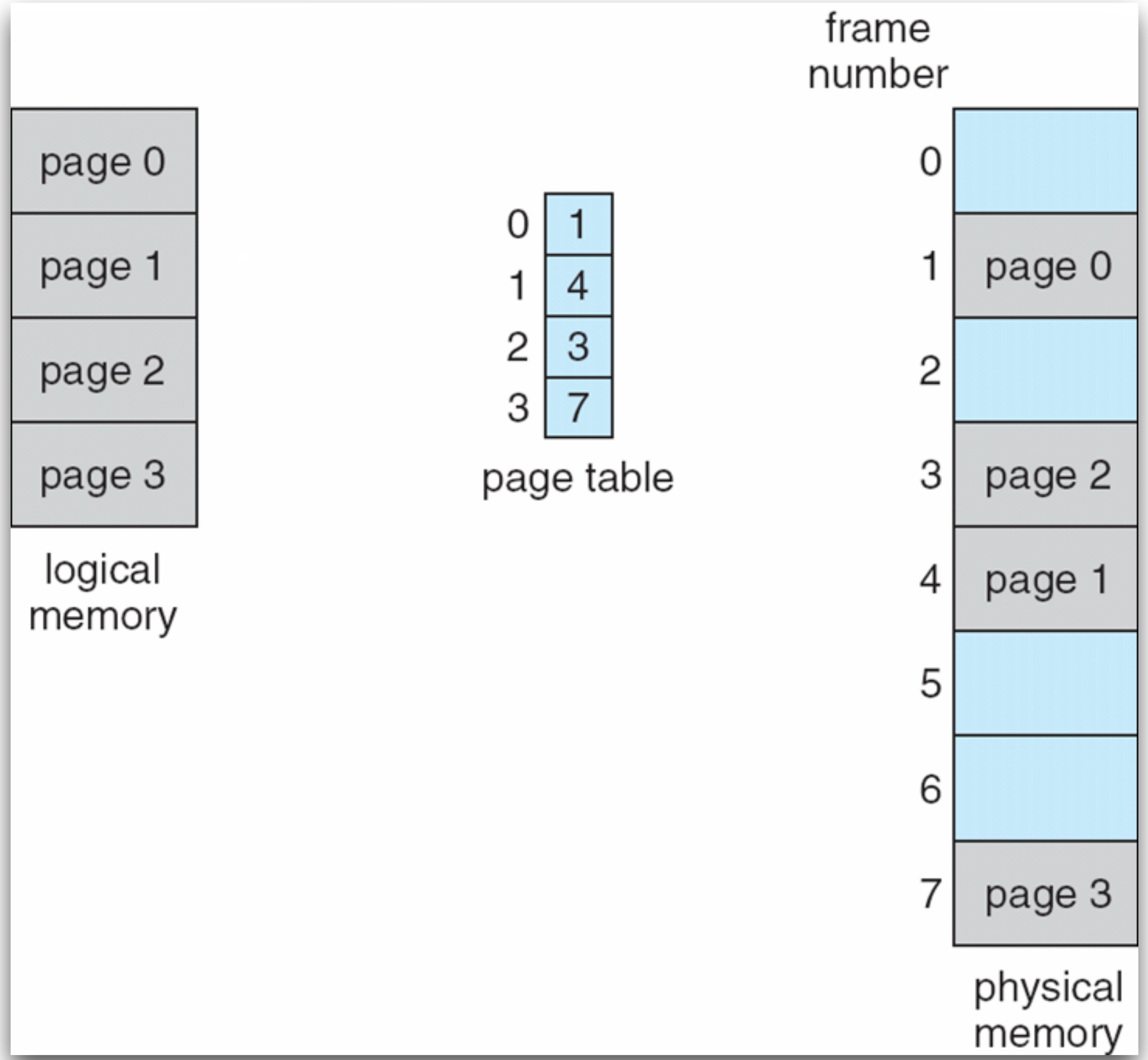
- Divide physical memory into fixed-sized blocks called **frames** (size is power of 2, between 512 bytes and 16 MB).
- Divide logical memory into blocks of same size called **pages.**



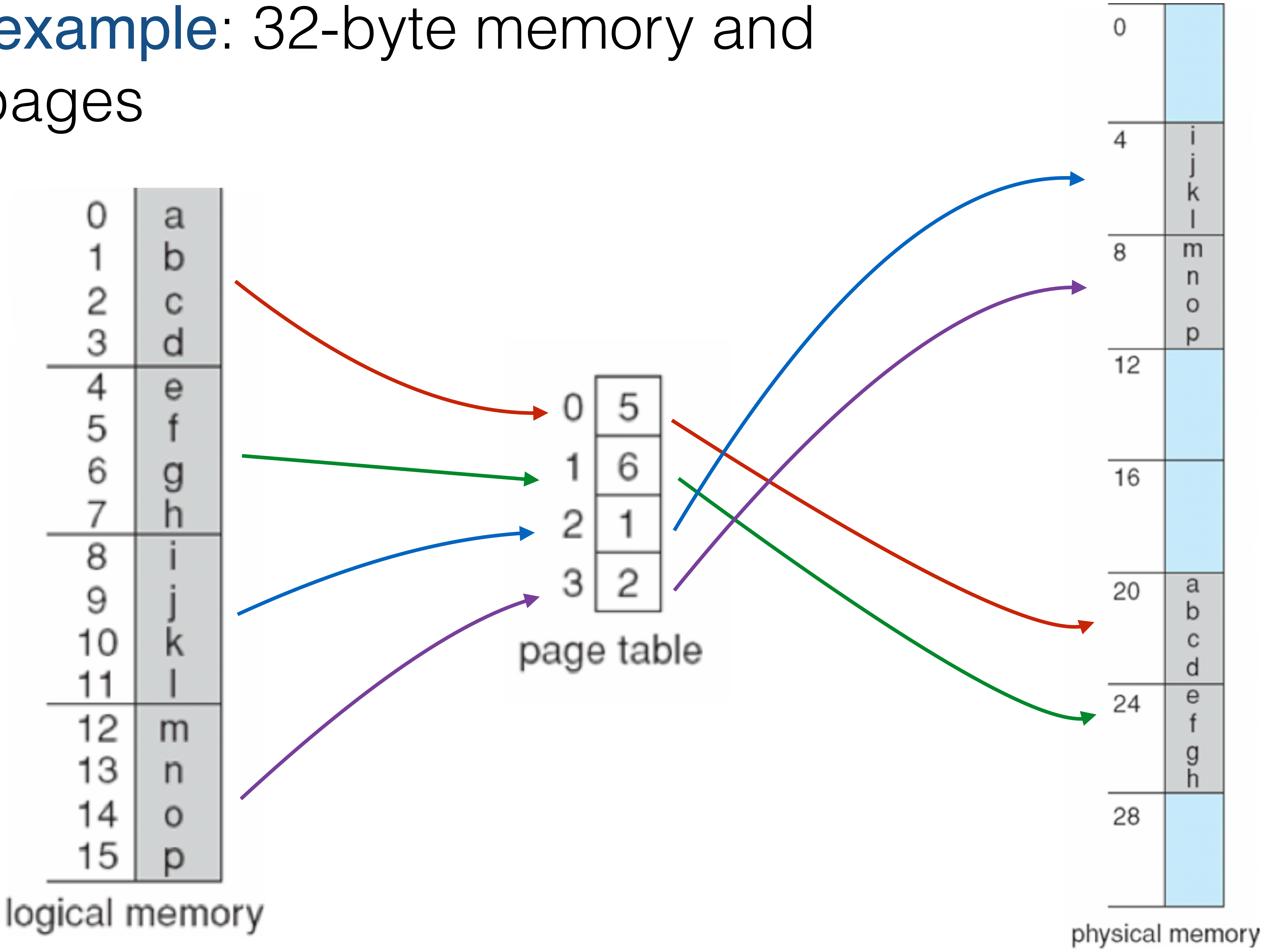
Paging

Basic paging method

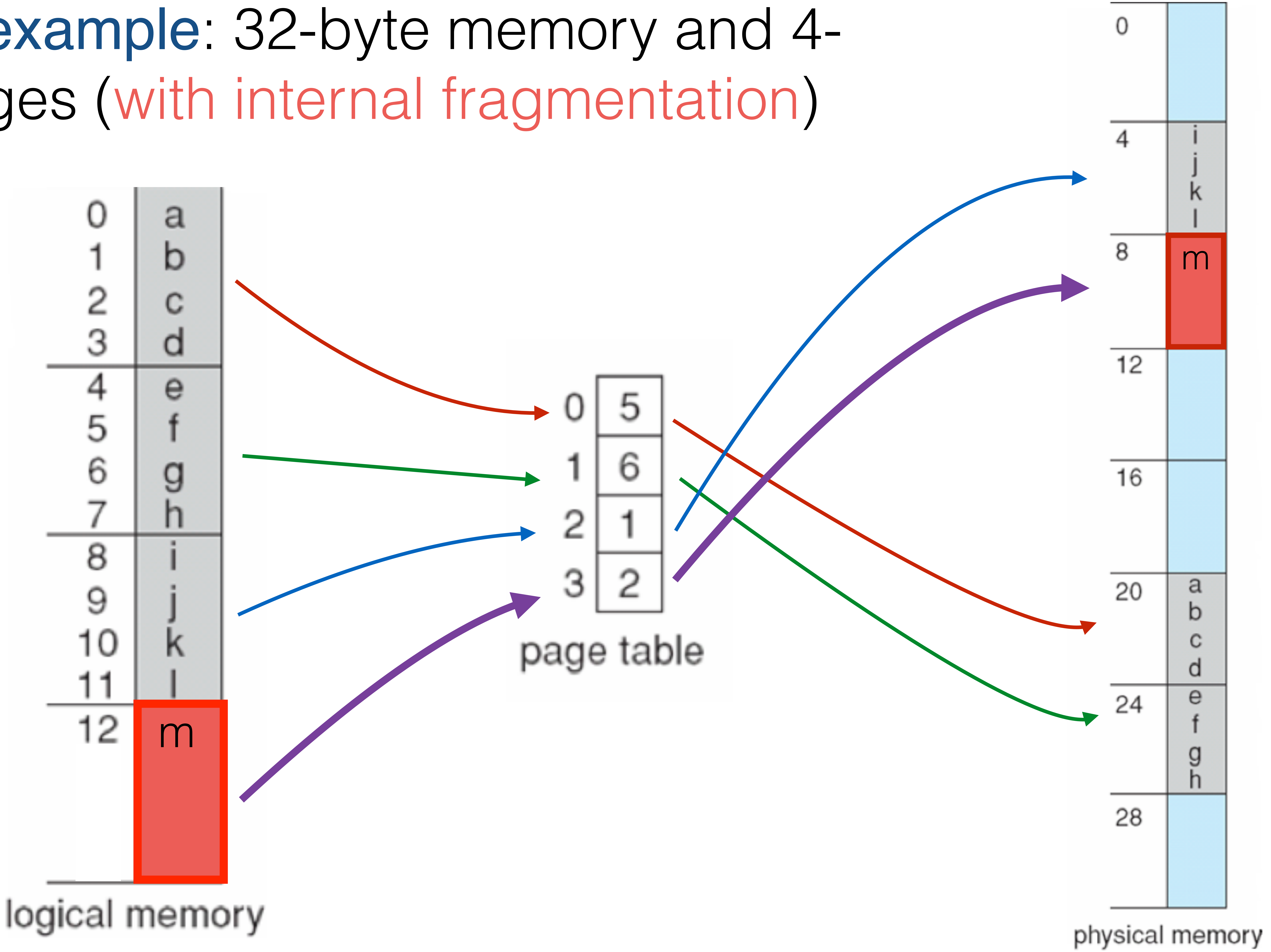
- Any page can be assigned to any free page frame
- External fragmentation is eliminated
- Internal fragmentation is at most a part of one page per process



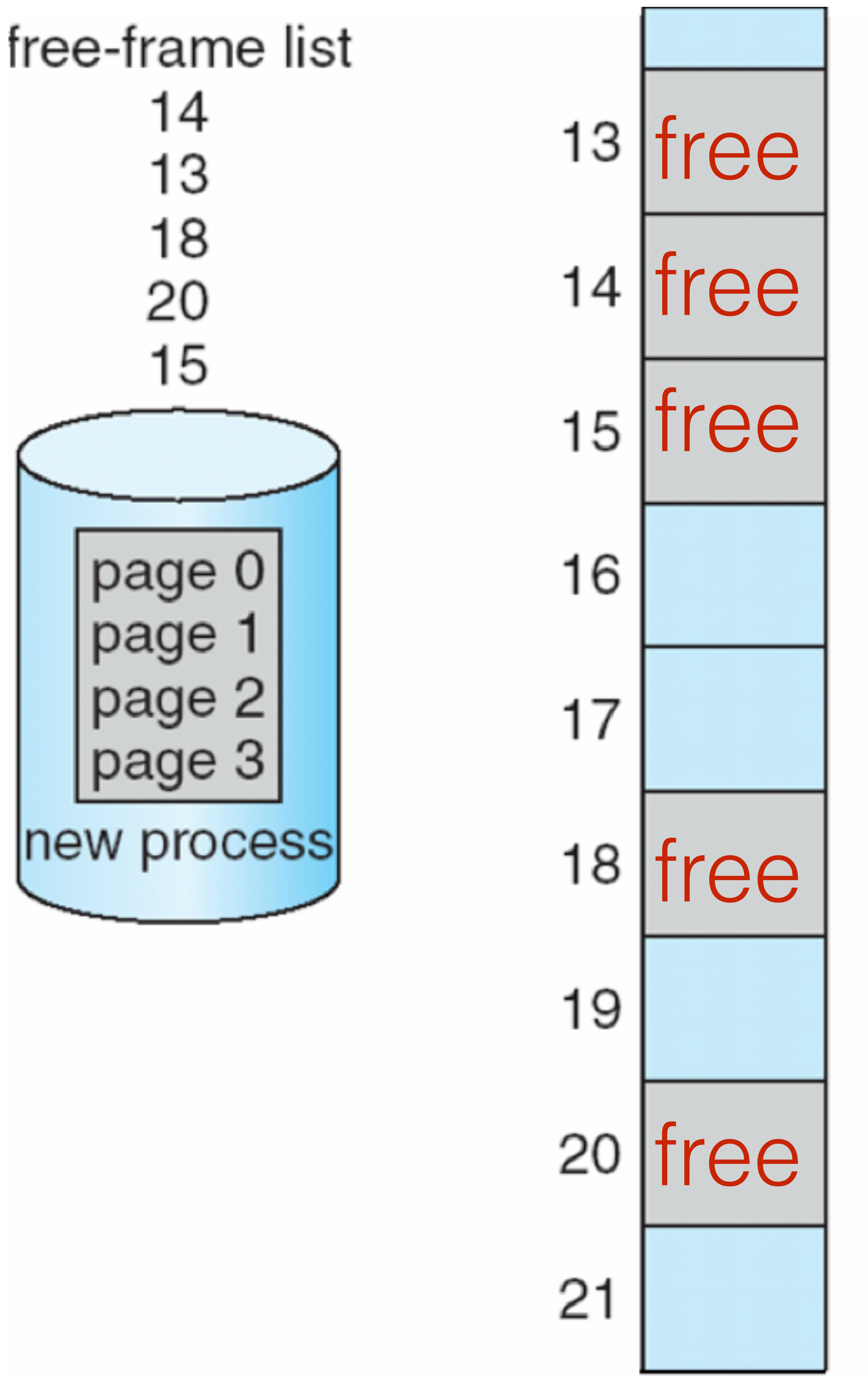
Paging example: 32-byte memory and 4-byte pages



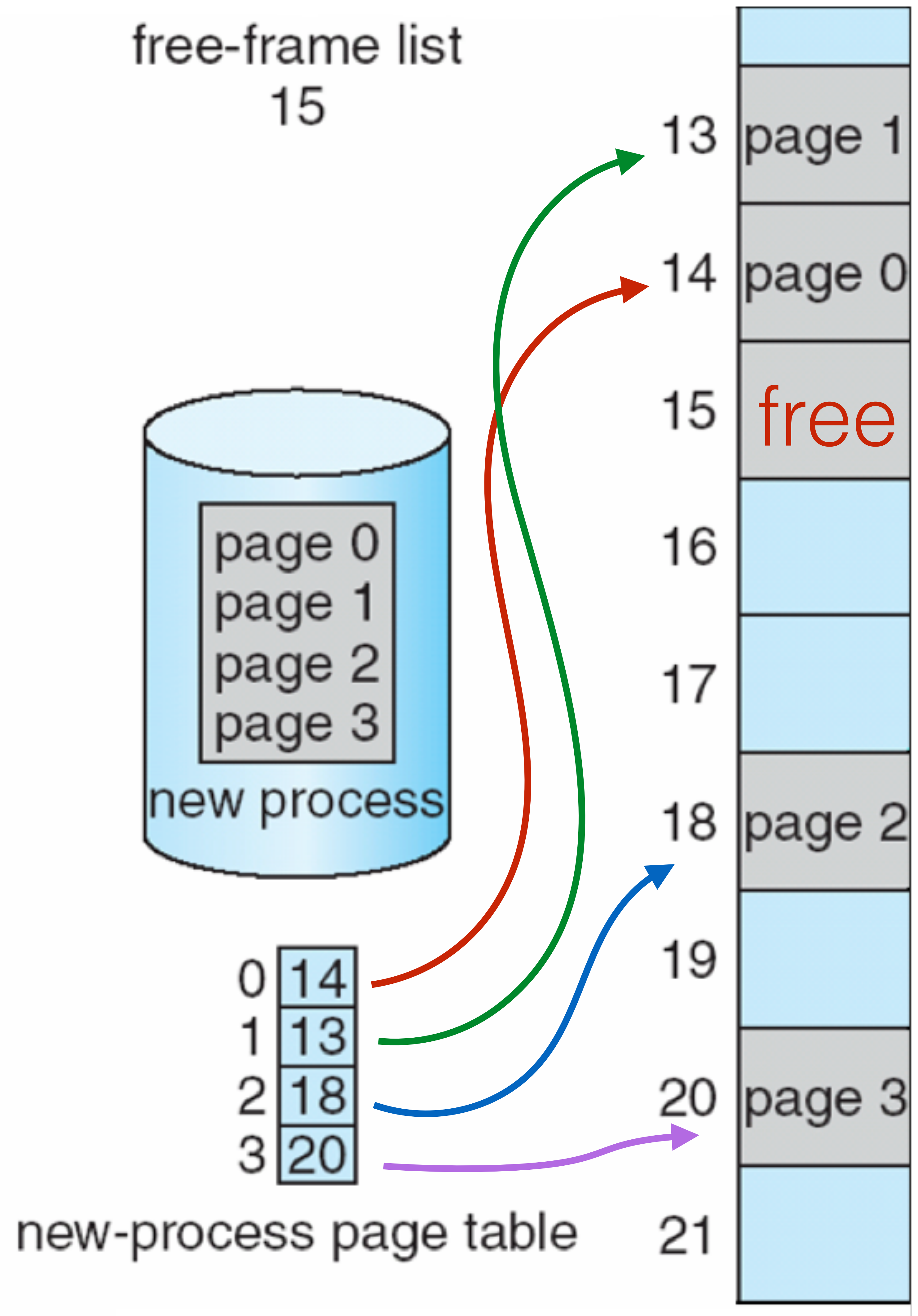
Paging example: 32-byte memory and 4-byte pages (with internal fragmentation)



New process is executed: free frames before and after allocation



Before allocation



After allocation

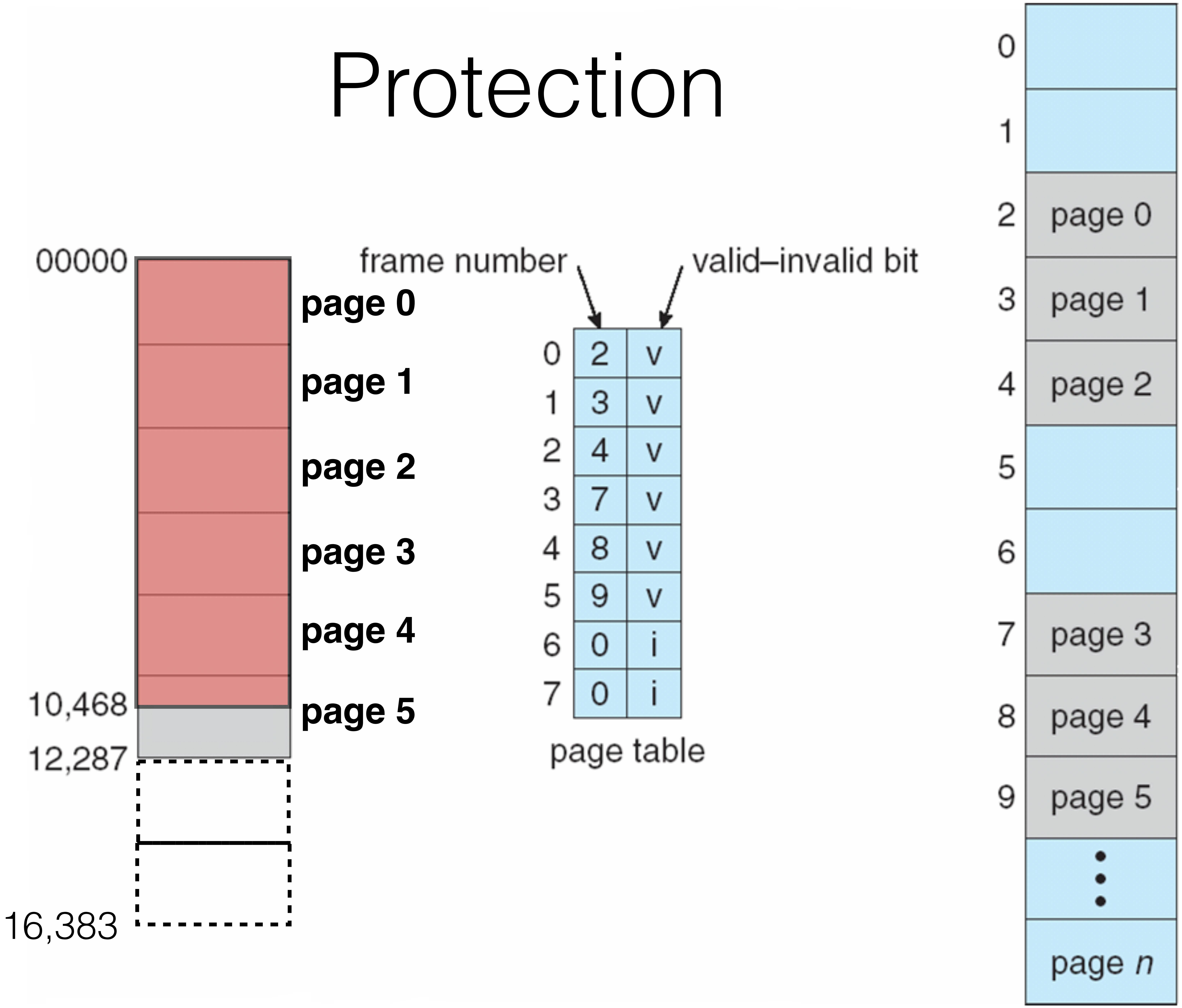
Paging Limitations - Space

- Page table might need a lot of space
- Registers can be used to store page tables but they are only feasible for small tables (e.g., 256 entries).
- Modern computers have page tables of 1 million entries.
- Such large page tables are kept in main memory and a page-table base register (PTBR) points to the table.

Protection

- Memory protection: each frame has a protection bit.
- **Valid-invalid** bit for each entry in the page table:
- “valid” indicates that the associated page is in the process’ logical address space, and is thus a legal page.
- “invalid” indicates that the page is not in the process’ logical address space.

Protection


00000
page 0
page 1
page 2
page 3
page 4
page 5
10,468
12,287
16,383
frame number
valid–invalid bit
0 2 v
1 3 v
2 4 v
3 7 v
4 8 v
5 9 v
6 0 i
7 0 i
page table
0
1
2 page 0
3 page 1
4 page 2
5
6
7 page 3
8 page 4
9 page 5
page n

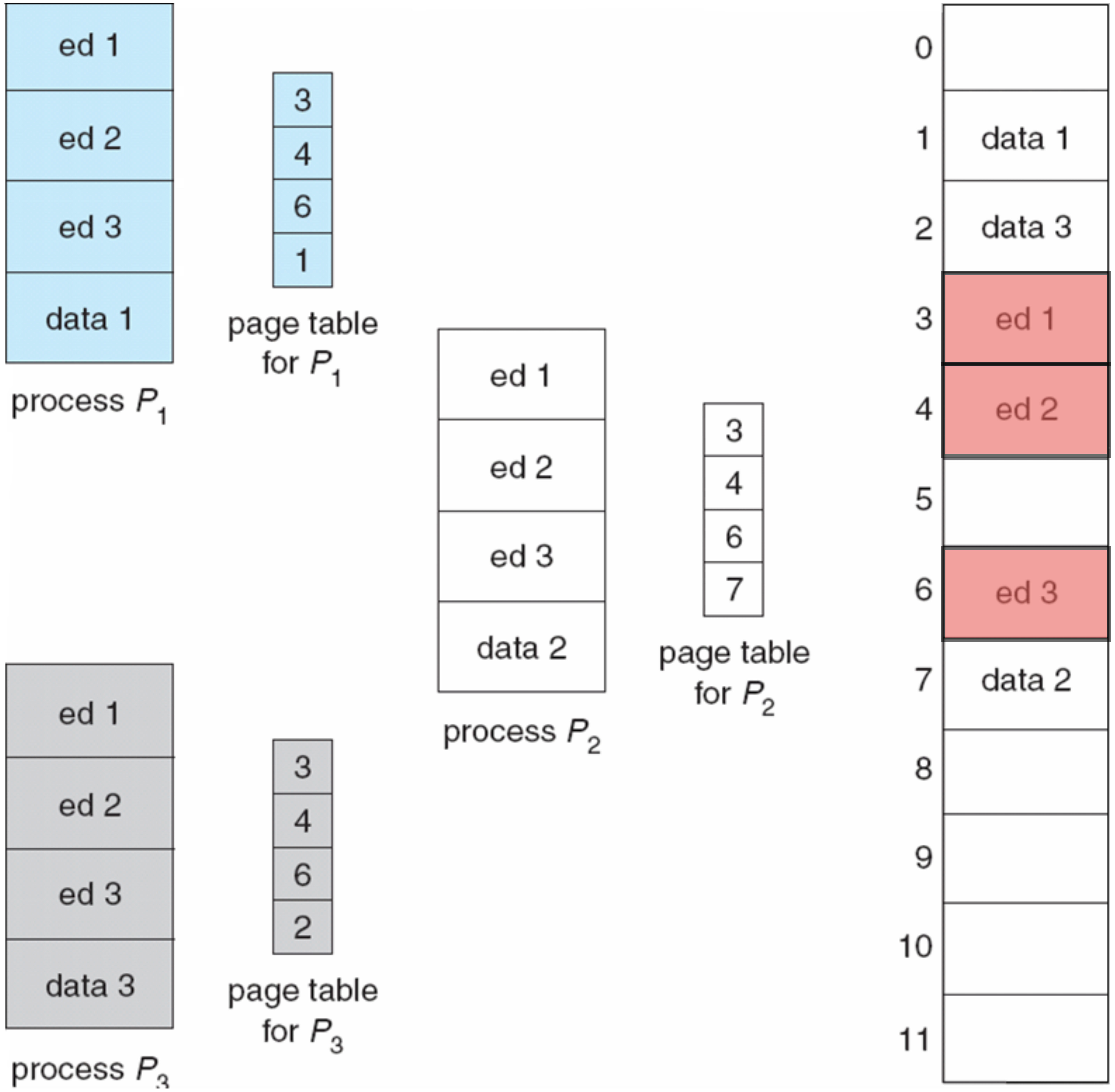
A few more useful aspects of paging

- Shared pages
- Copy-on-write
- Memory-mapped files

Shared Pages

- Paging allows for the possibility of sharing common code.
- Sharing pages is useful in time-sharing environments (e.g., 40 users, each executing a text editor).
- OS can implement shared-memory (IPC) using shared pages.

Example of shared Pages



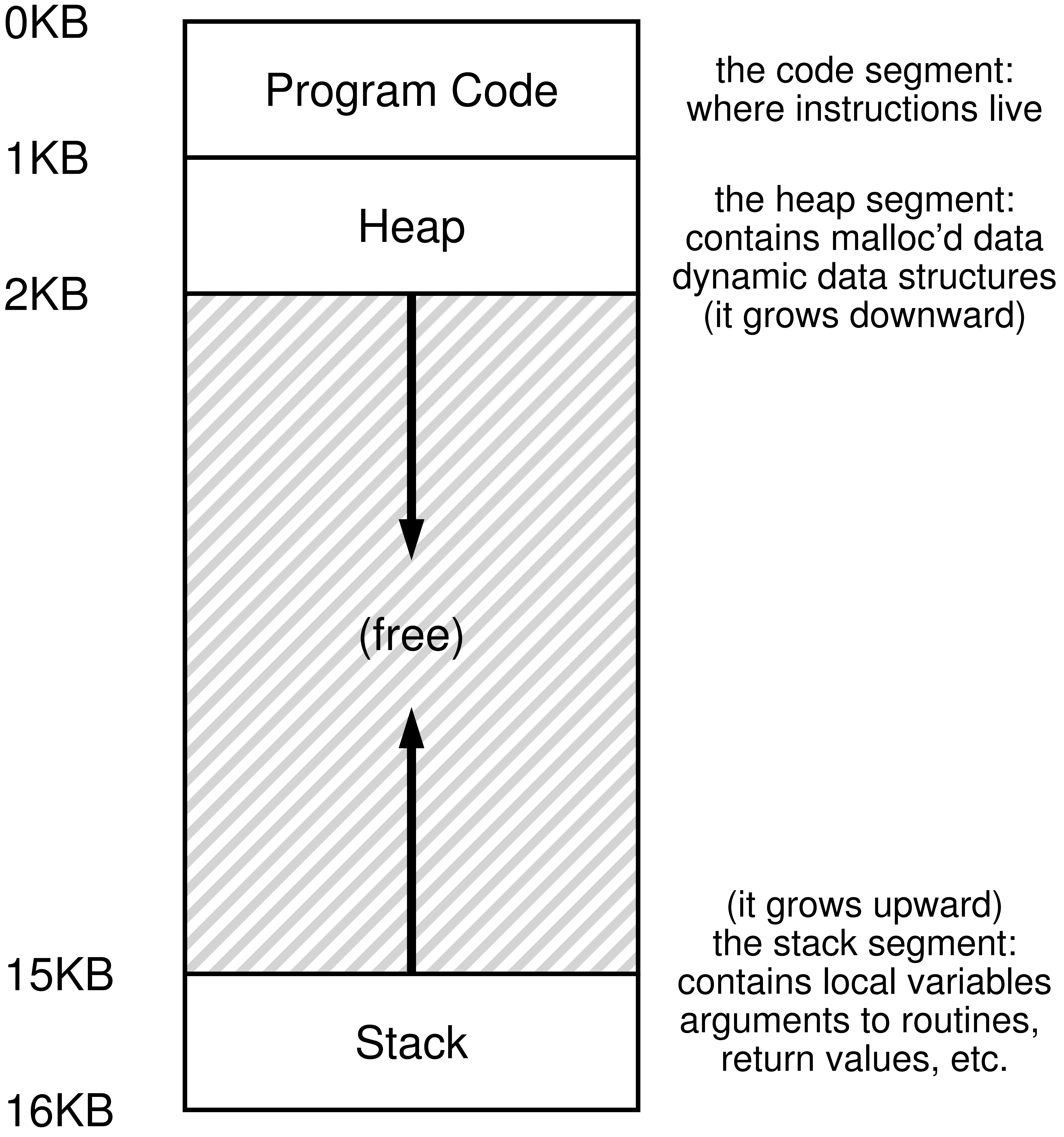
Copy-on-Write (COW), e.g. on `fork( )`

- copy-on-write (COW), e.g., on `fork( )`
  - Instead of copying all pages, create shared mappings of parent pages in child address space

    A. Make shared mappings read-only in child space

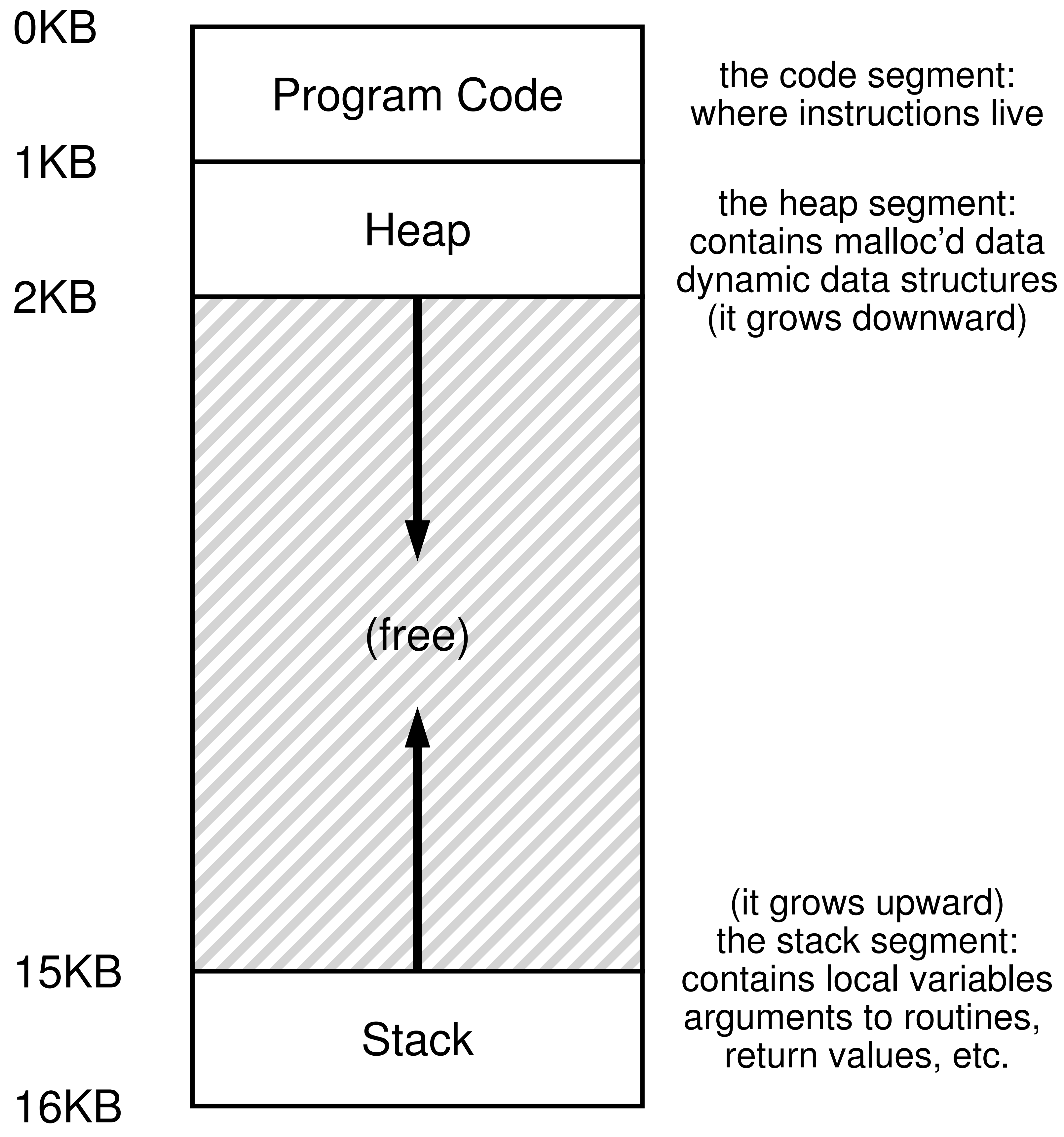
    B. When child does a write, a protection fault occurs, OS takes over and can then copy the page and resume child.

Example address space



http://pages.cs.wisc.edu/~remzi/OSTEP/vm-intro.pdf

Types of memory



Stack: Short-lived memory. Allocations and deallocations are managed *implicitly* (e.g., by the compiler), not by the programmer.

Heap: Long-lived memory. Allocations and deallocations are *explicitly* handled by the programmer.

Examples

```
void func() {
    int x;
    ...
}
```

Examples

```
void func() {
    int *x = (int *) malloc(sizeof(int));
    ...
}
```

Every address you see is virtual

Here's a little program that prints out the locations of the `main()` routine (where code lives), the value of a heap-allocated value returned from `malloc()`, and the location of an integer on the stack:

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[]) {
    printf("location of code  : %p\n", (void *) main);
    printf("location of heap  : %p\n", (void *) malloc(1));
    int x = 3;
    printf("location of stack : %p\n", (void *) &x);
    return x;
}
```

When run on a 64-bit Mac OS X machine, we get the following output:

```
location of code  : 0x1095afe50
location of heap  : 0x1096008c0
location of stack : 0x7fff691aea64
```

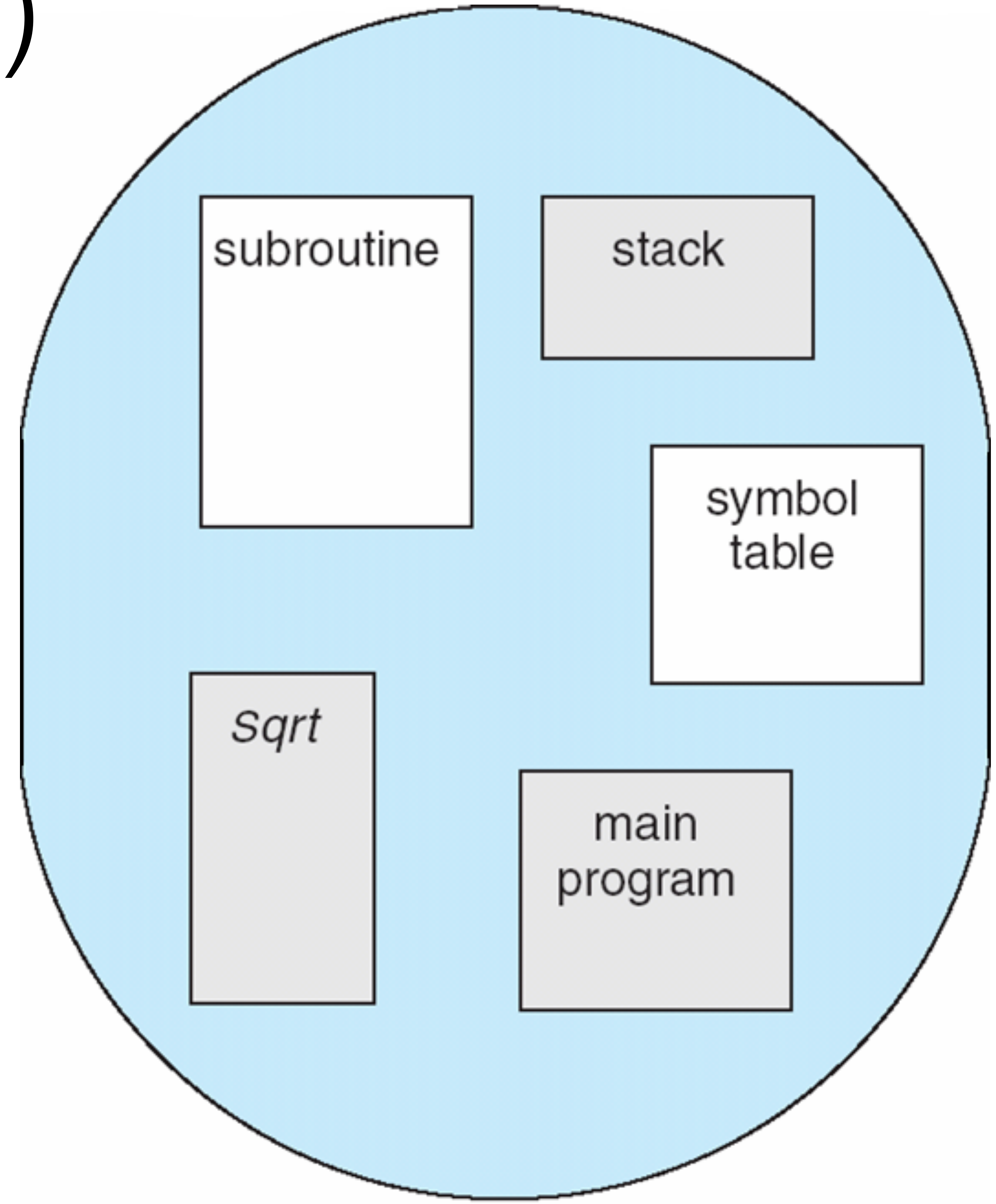
http://pages.cs.wisc.edu/~remzi/OSTEP/vm-intro.pdf

Segmentation

- Memory-management scheme that supports the user’s view of memory.
- View memory as a collection of variable-sized segments, with no necessary ordering among segments.

Segmentation (a program)

- We think of a program as a main program, a stack, a math library, etc.
- Each module is referred to by name
- In this view of a program, we might not care whether the stack is stored before or after the **`sqrt()`** function.

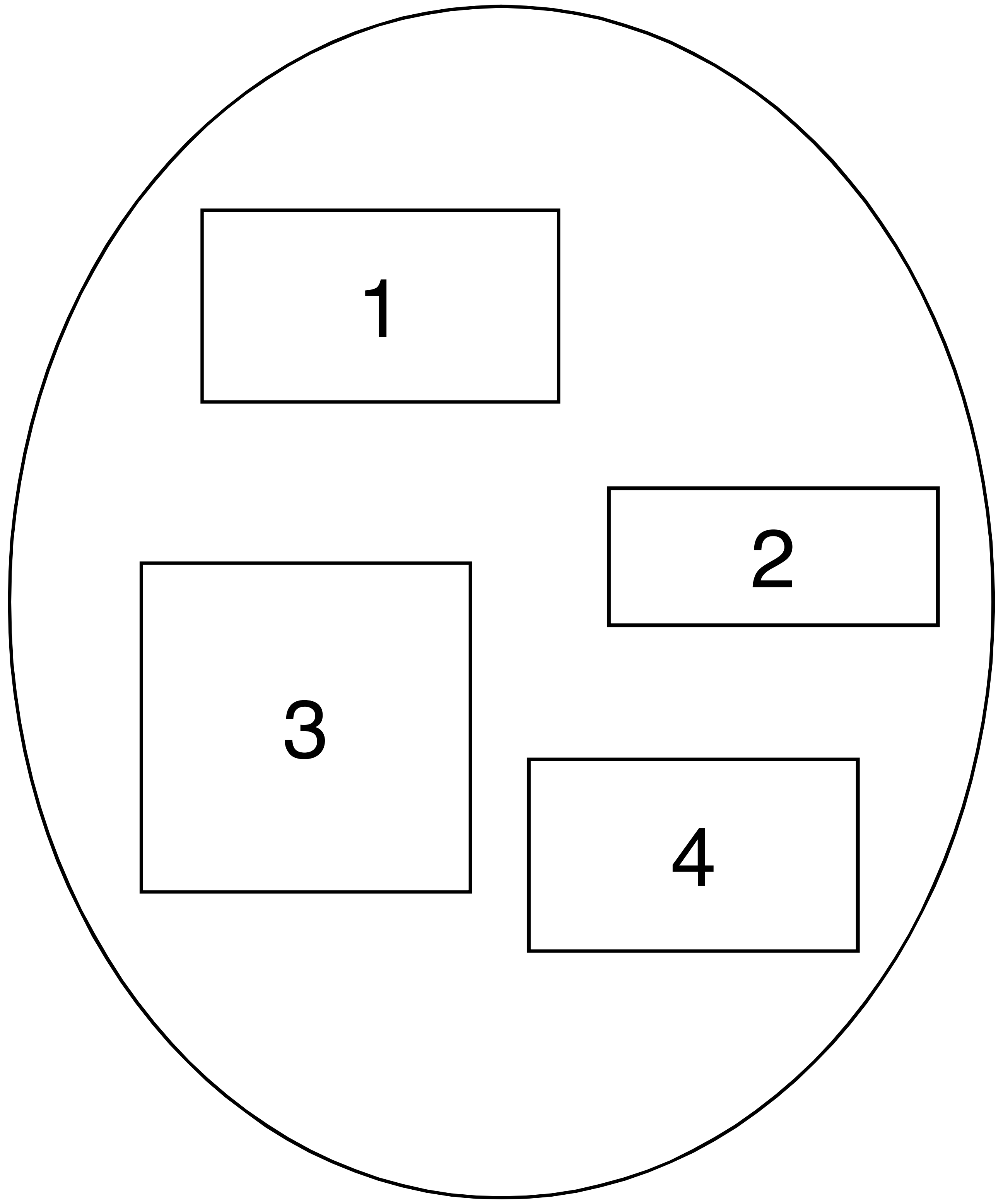


logical address

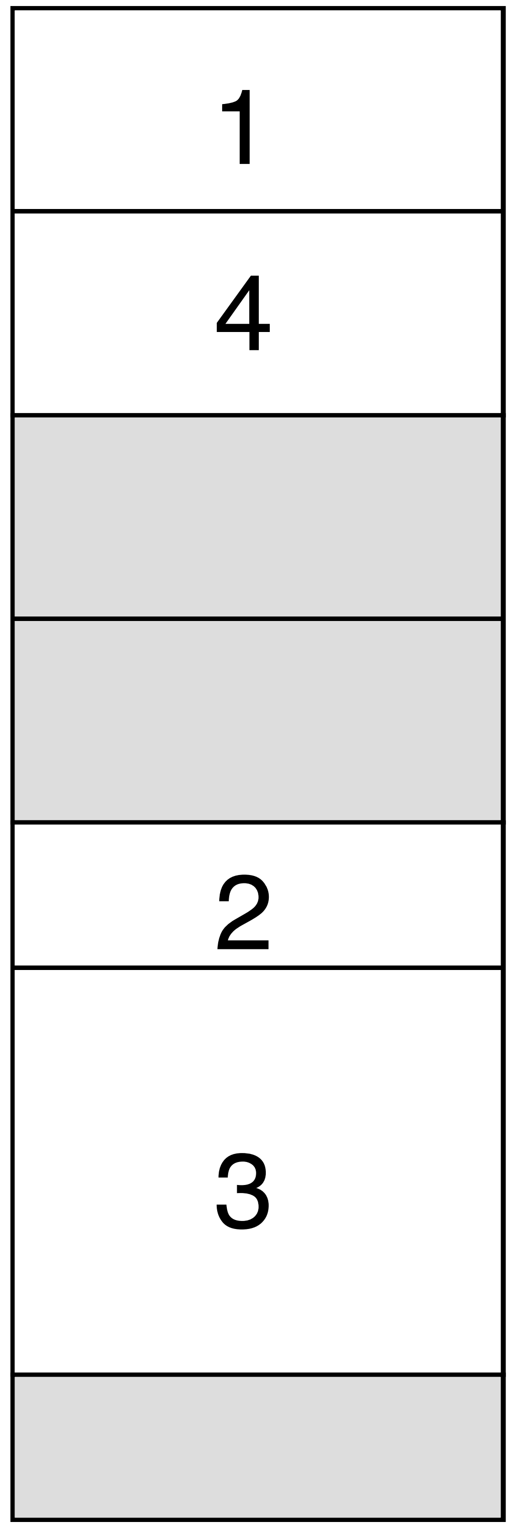
Logical view of segmentation

- For simplicity of implementation, each segment is addressed by a **segment number** and an **offset**:

`<segment-number, offset>`



user space



physical memory space

Segmentation Hardware

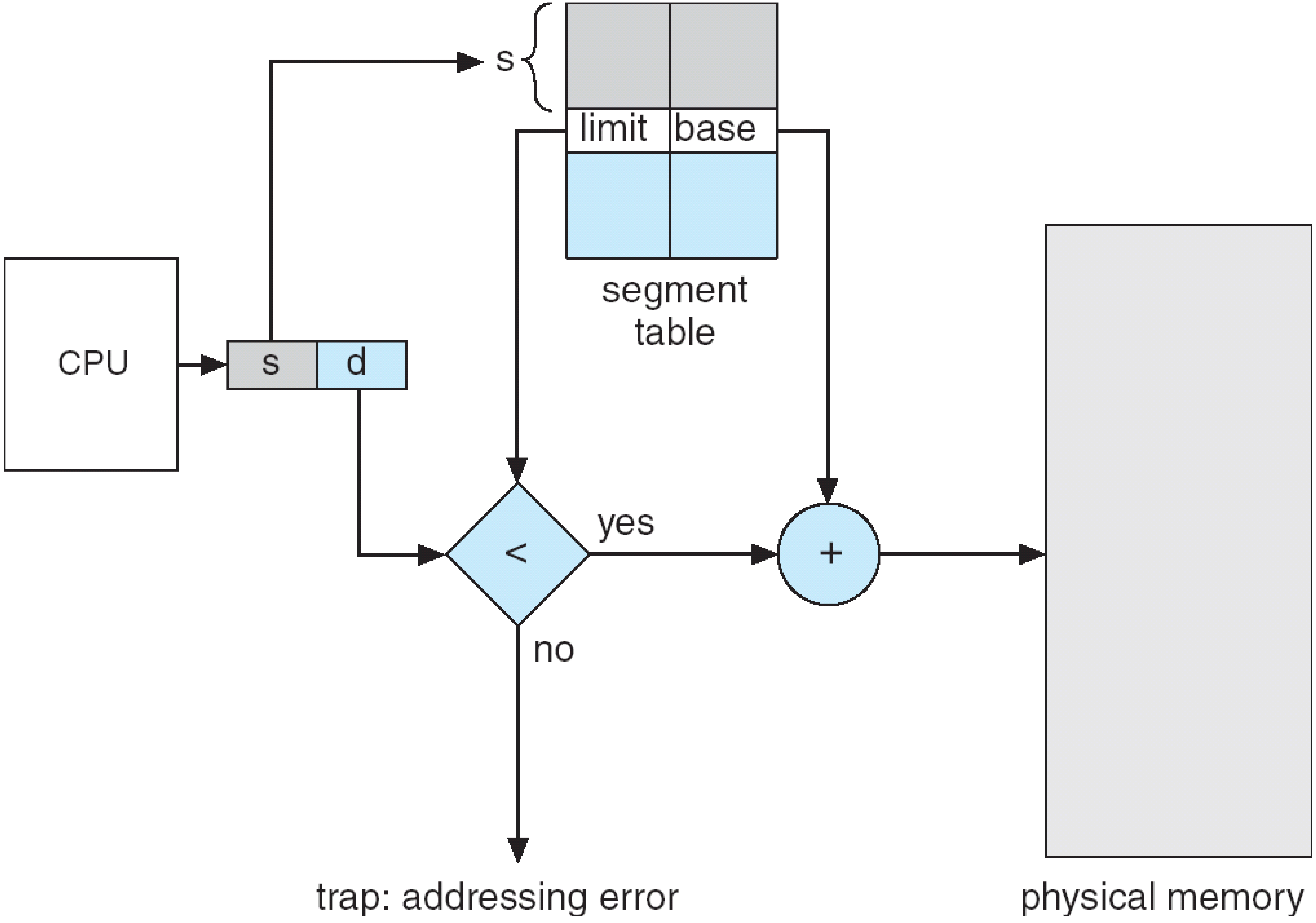
Segment tables:

Base: starting address of the segment in physical memory.

Limit: length of the segment.

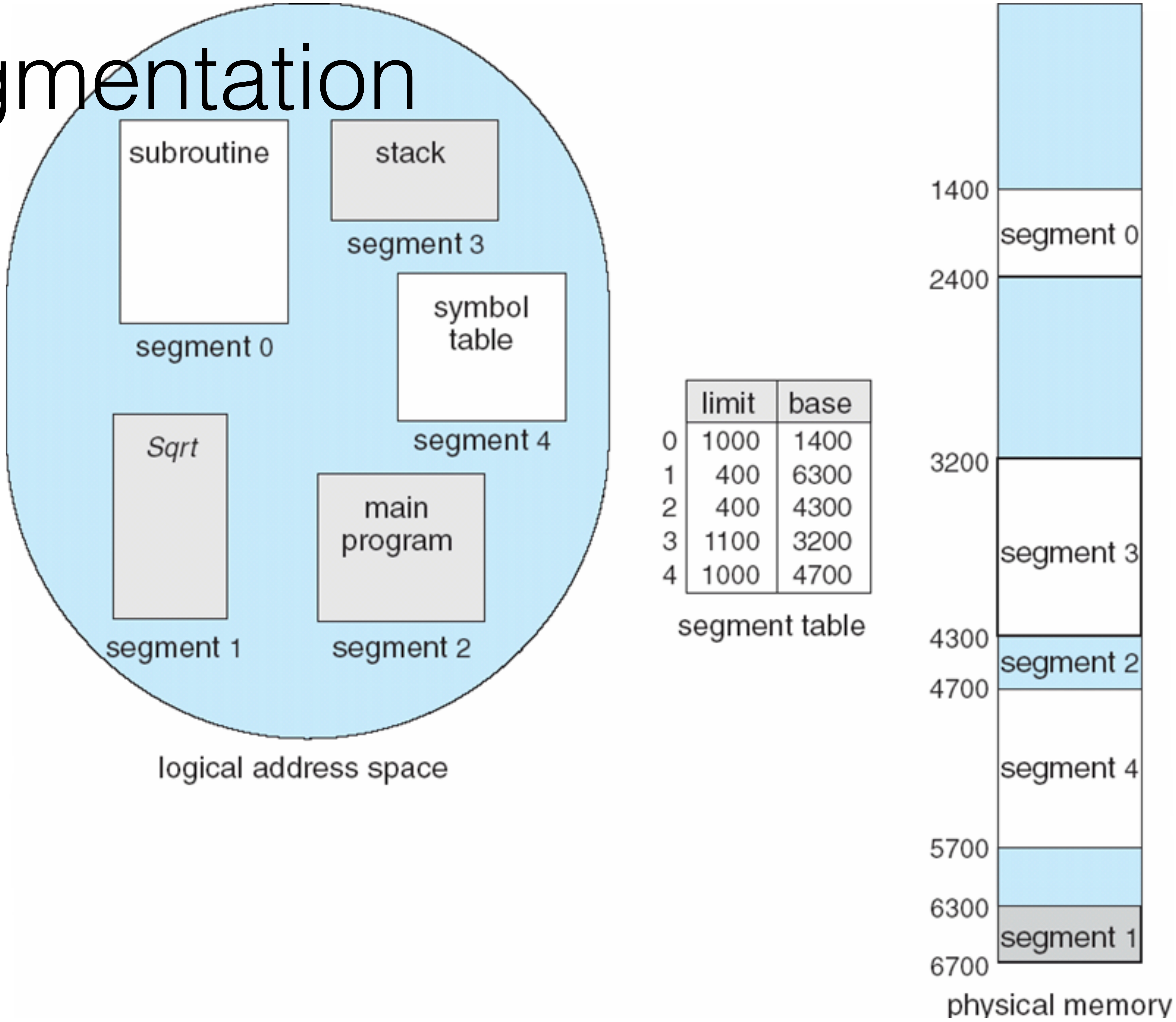
Additional metadata includes protection bits.

$<$segment-number, offset$>$



Example of segmentation

- Logical memory divided into 5 segments.
- Segment 2 is 400 bytes long and begins at location 4,300.
- Question: What happens if there is a reference to byte 1,222 of segment 0?



Some questions

How do paging and segmentation compare with respect to the following issues?

- External fragmentation
- Internal fragmentation
- Ability to share code across processes

Some questions

Assuming a 1-KB page size, what are the page numbers and offset for the following address:

A. 2375

B. 256

Some questions

Segment	Base	Length
0	219	600
1	2300	14
2	90	100
3	1327	580
4	1952	96

What are the physical addresses for the following logical addresses?

a. 0,430
b. 2,500