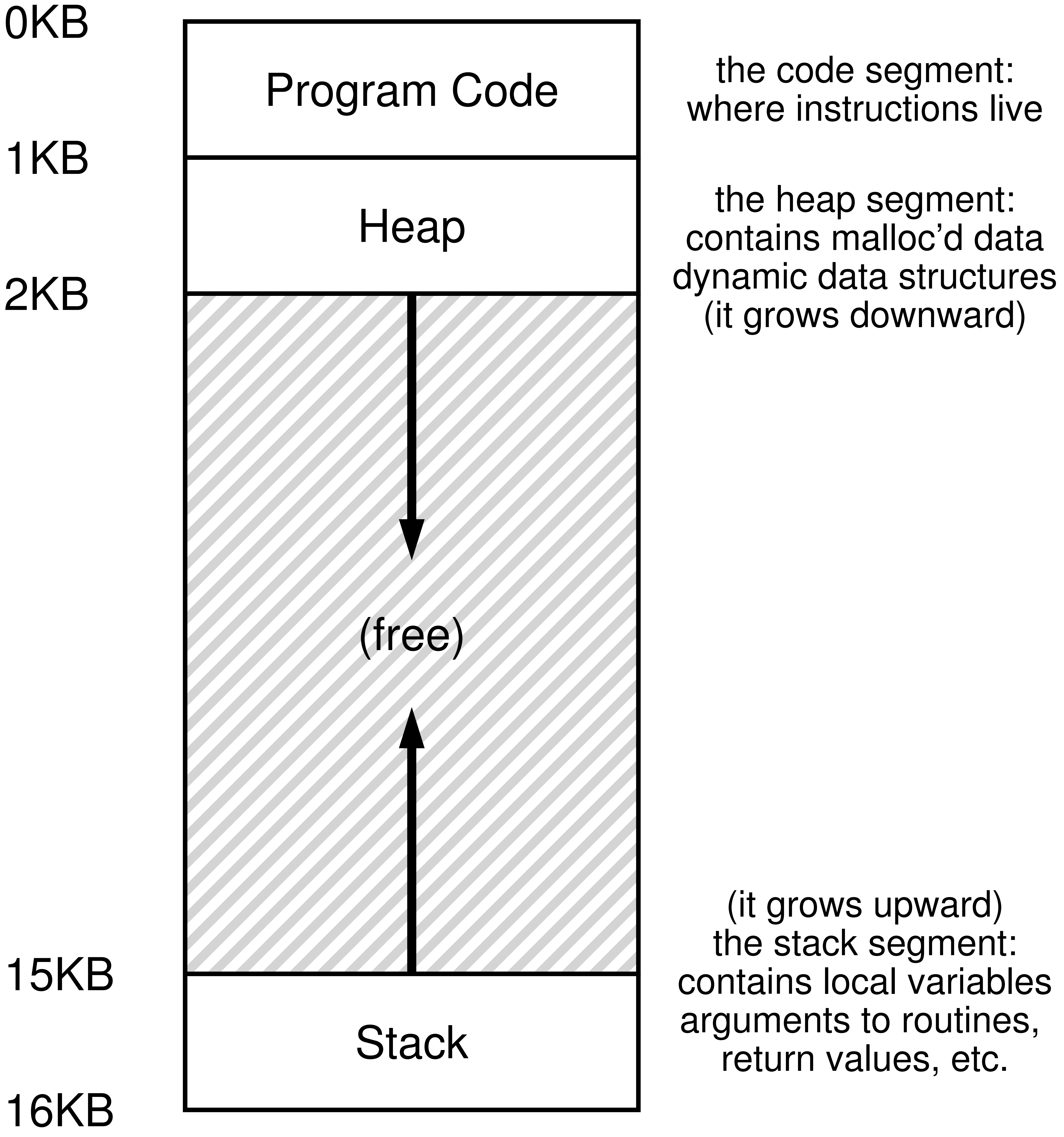
Paging

CSE 4001

Content

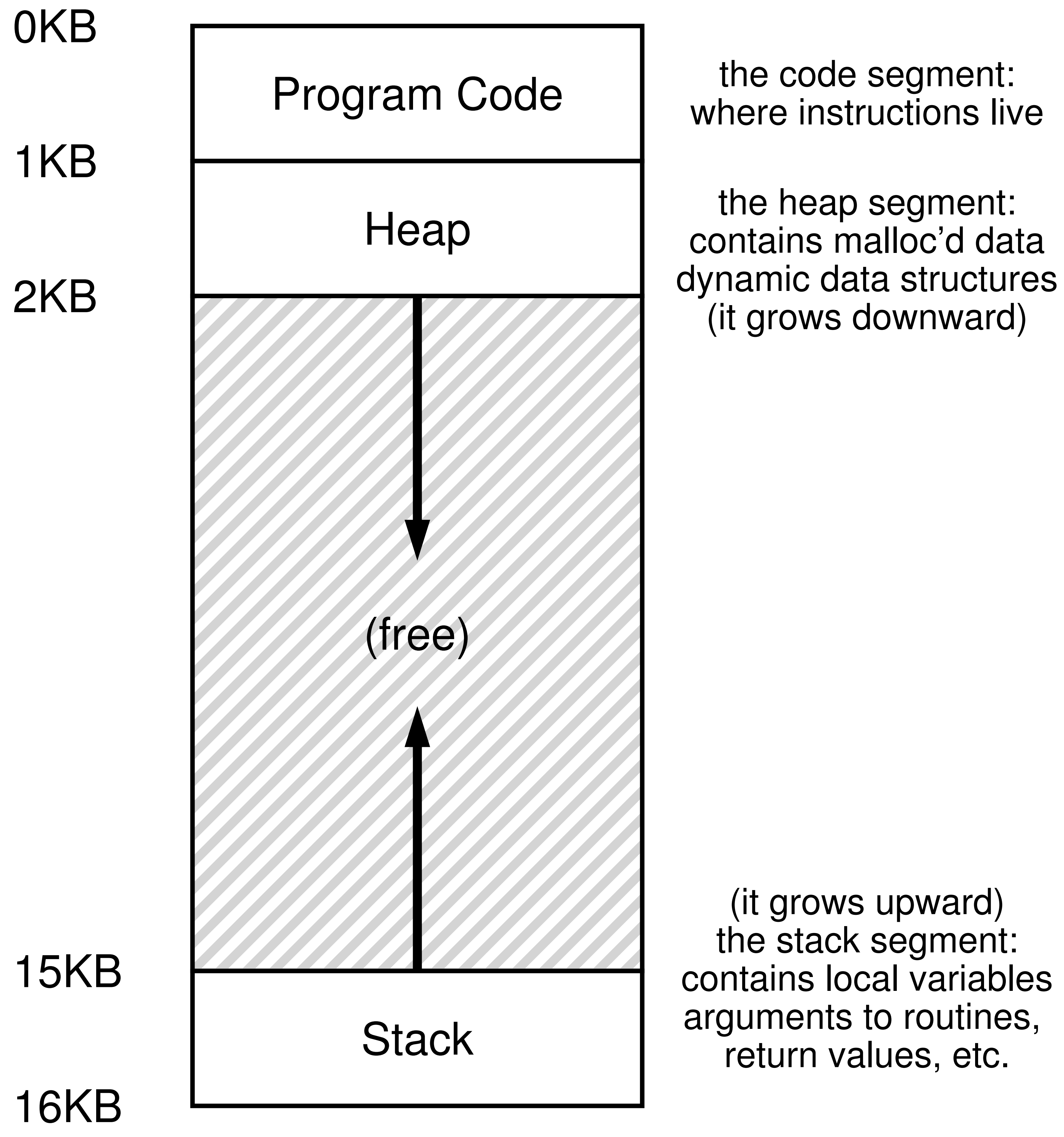
- Virtual address space
- Basic paging mechanism
- Limitations
- Protection
- Shared pages

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

Paging

Basic problem with allocating contiguous blocks of memory for processes

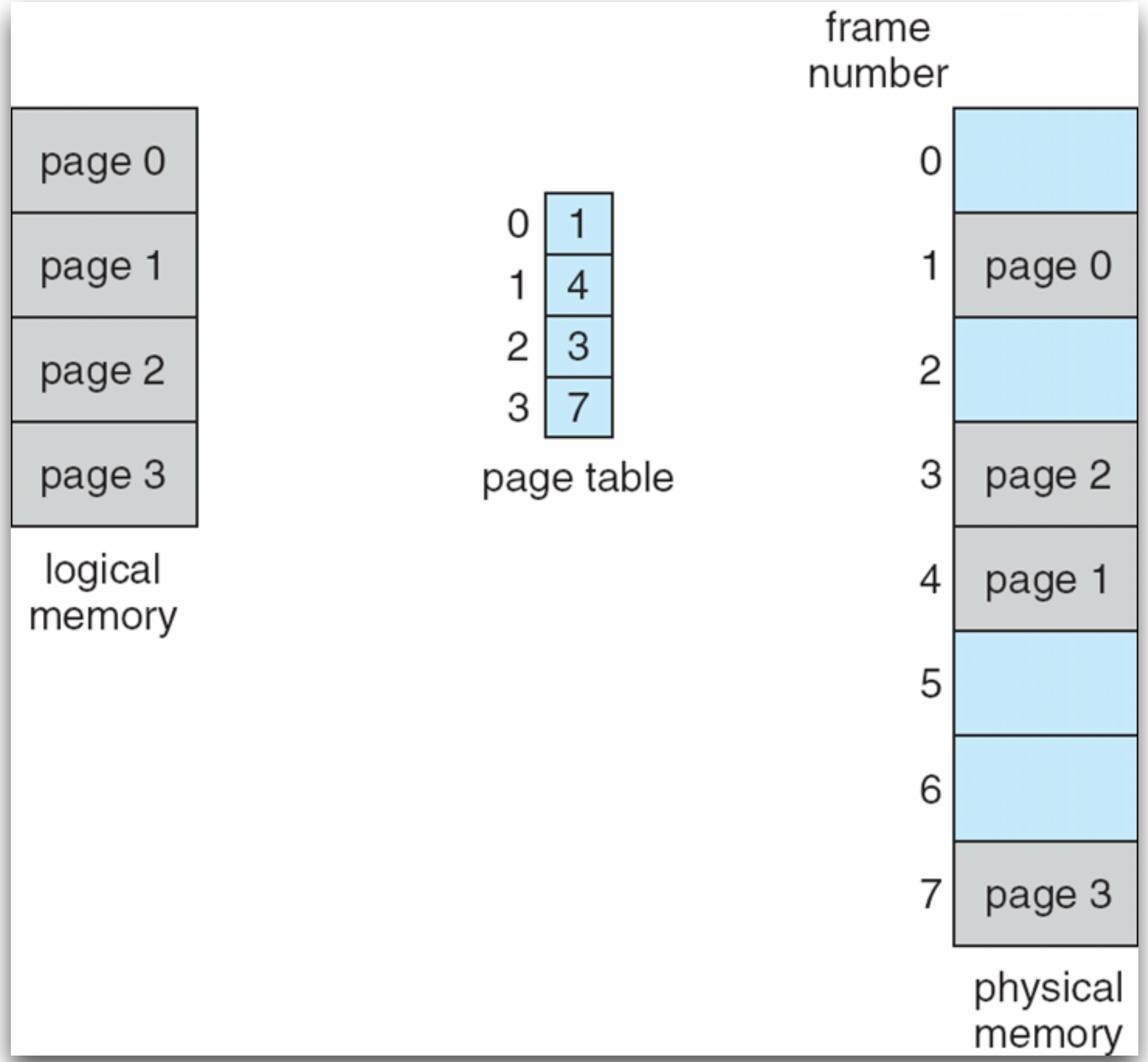
- Determining the size of memory blocks is difficult because different processes have different memory requirements.

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

Paging

Basic paging method

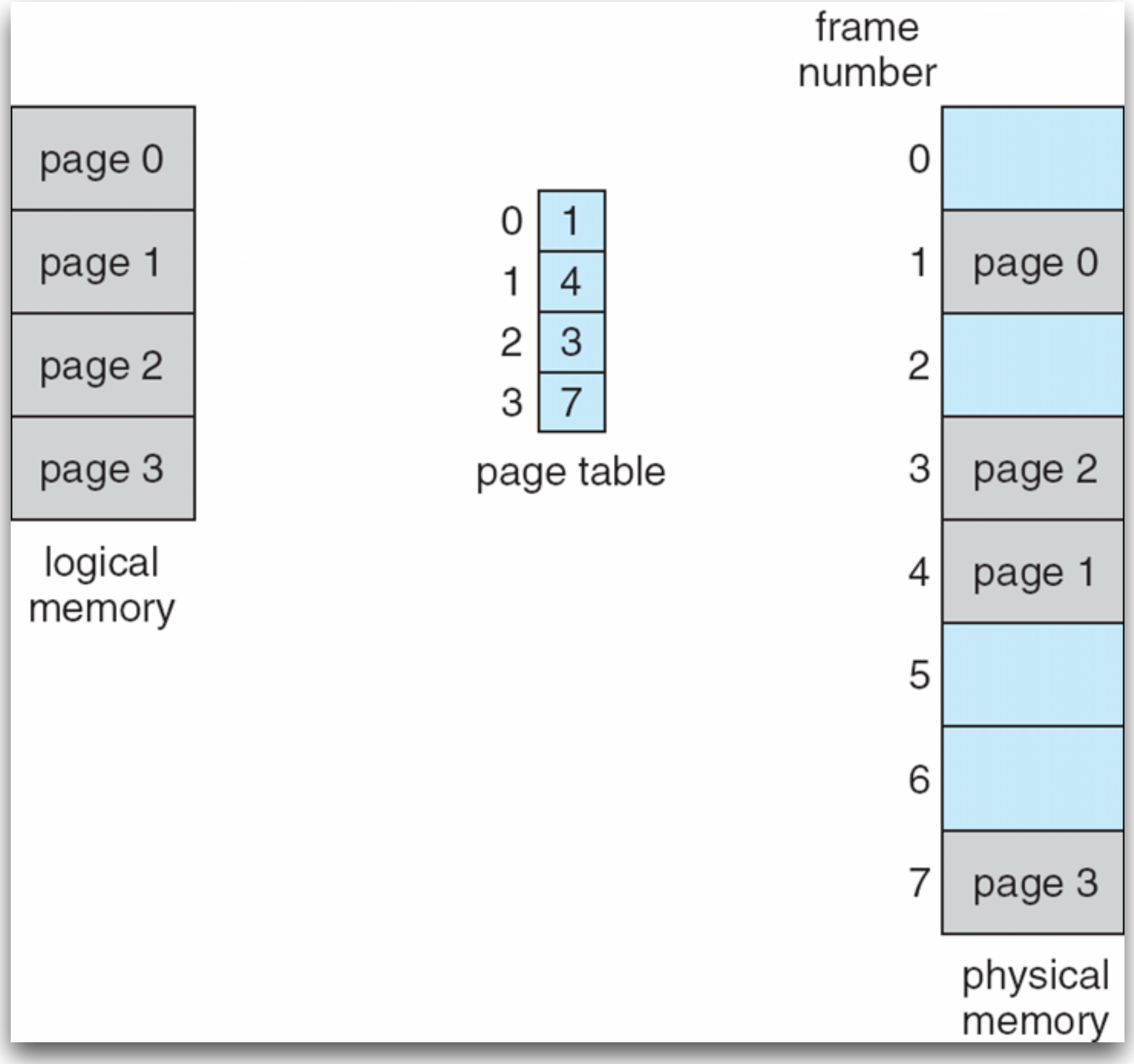
- Divide physical memory into fixed-sized blocks called **frames** (size is power of 2, between 512 bytes and 16 MB).
- Divide logical memory into blocks of same size called **pages.**



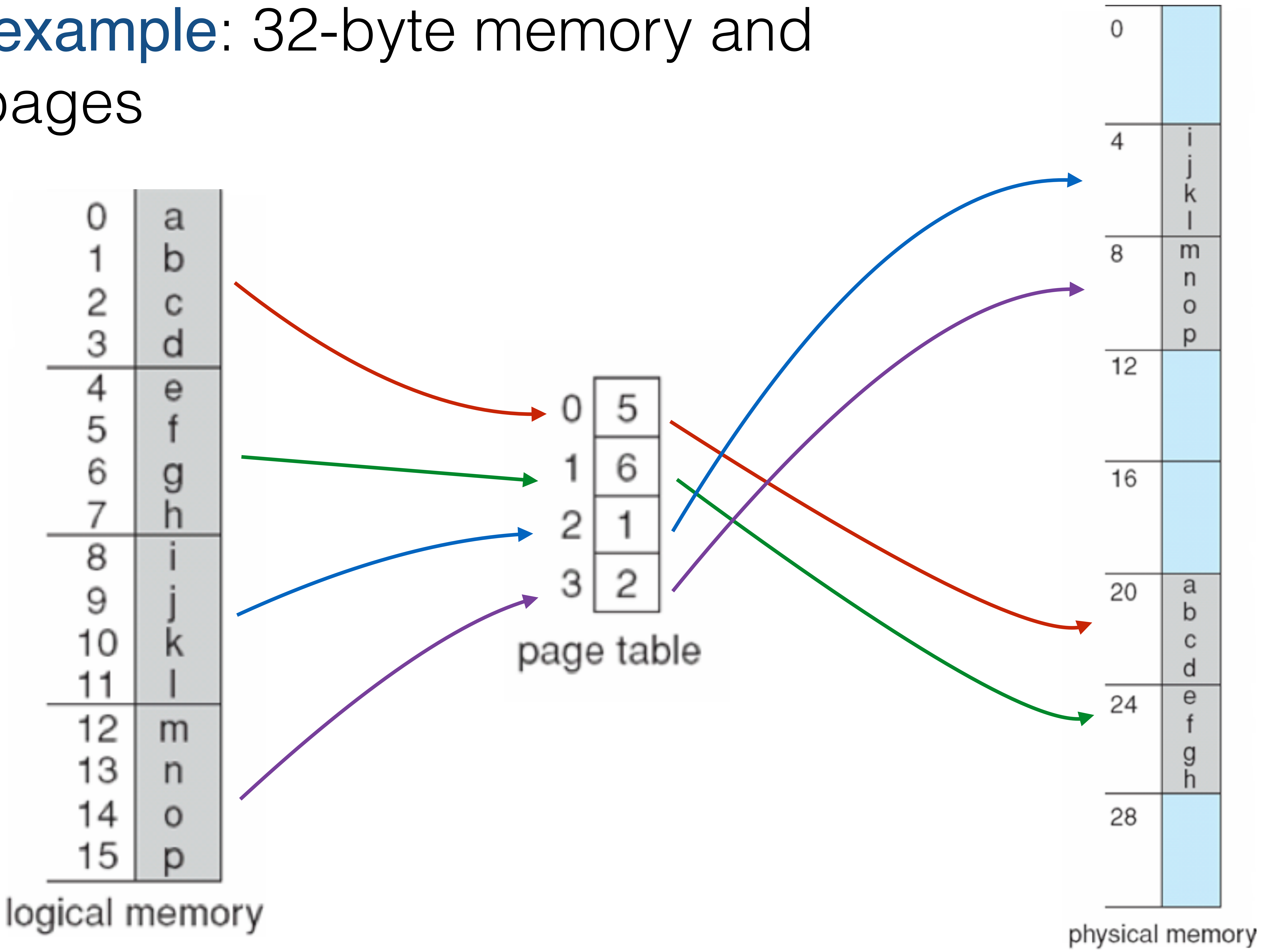
Paging

Basic paging method

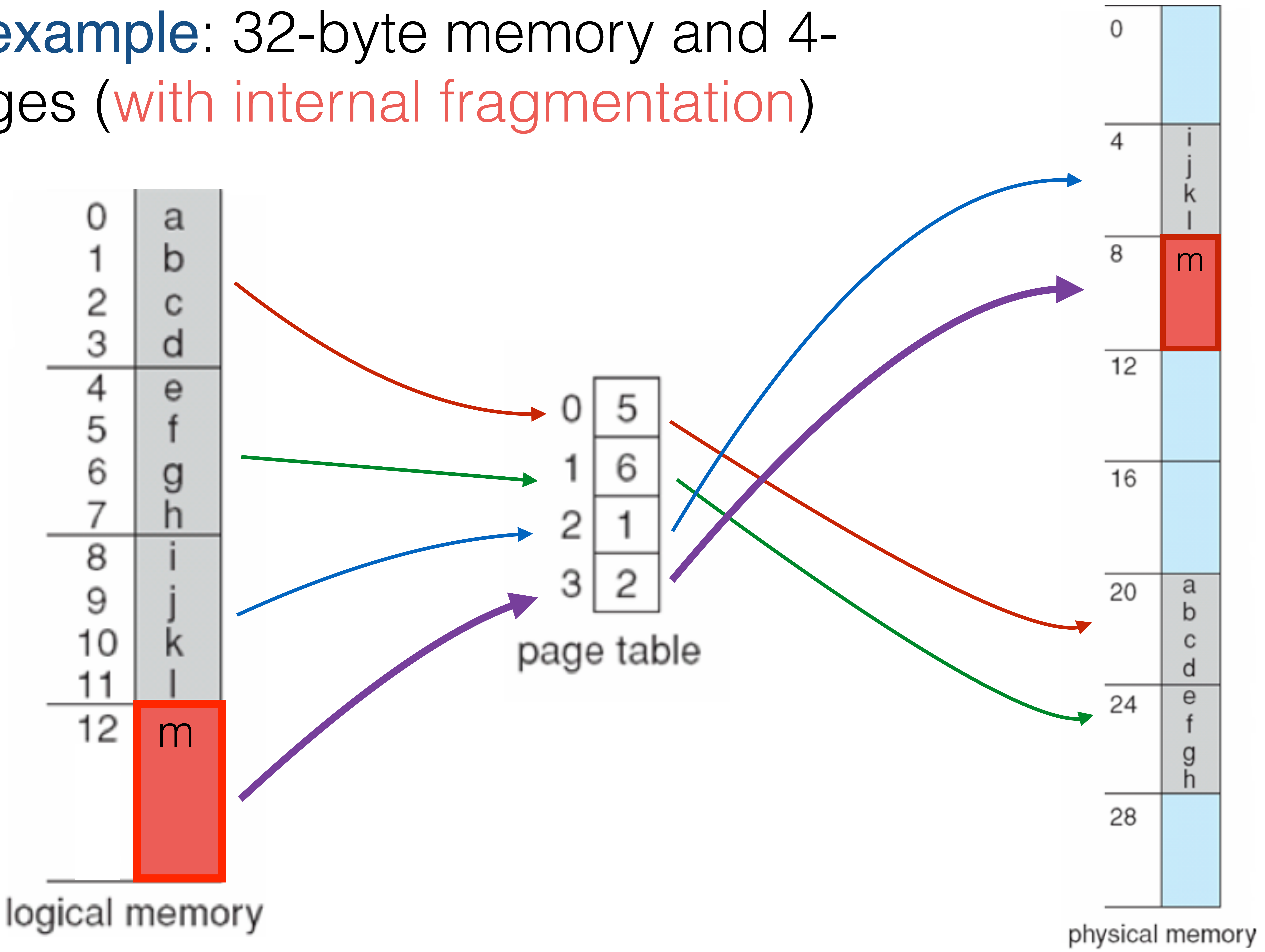
- Any page can be assigned to any free page frame
- External fragmentation is eliminated
- Internal fragmentation is at most a part of one page per process



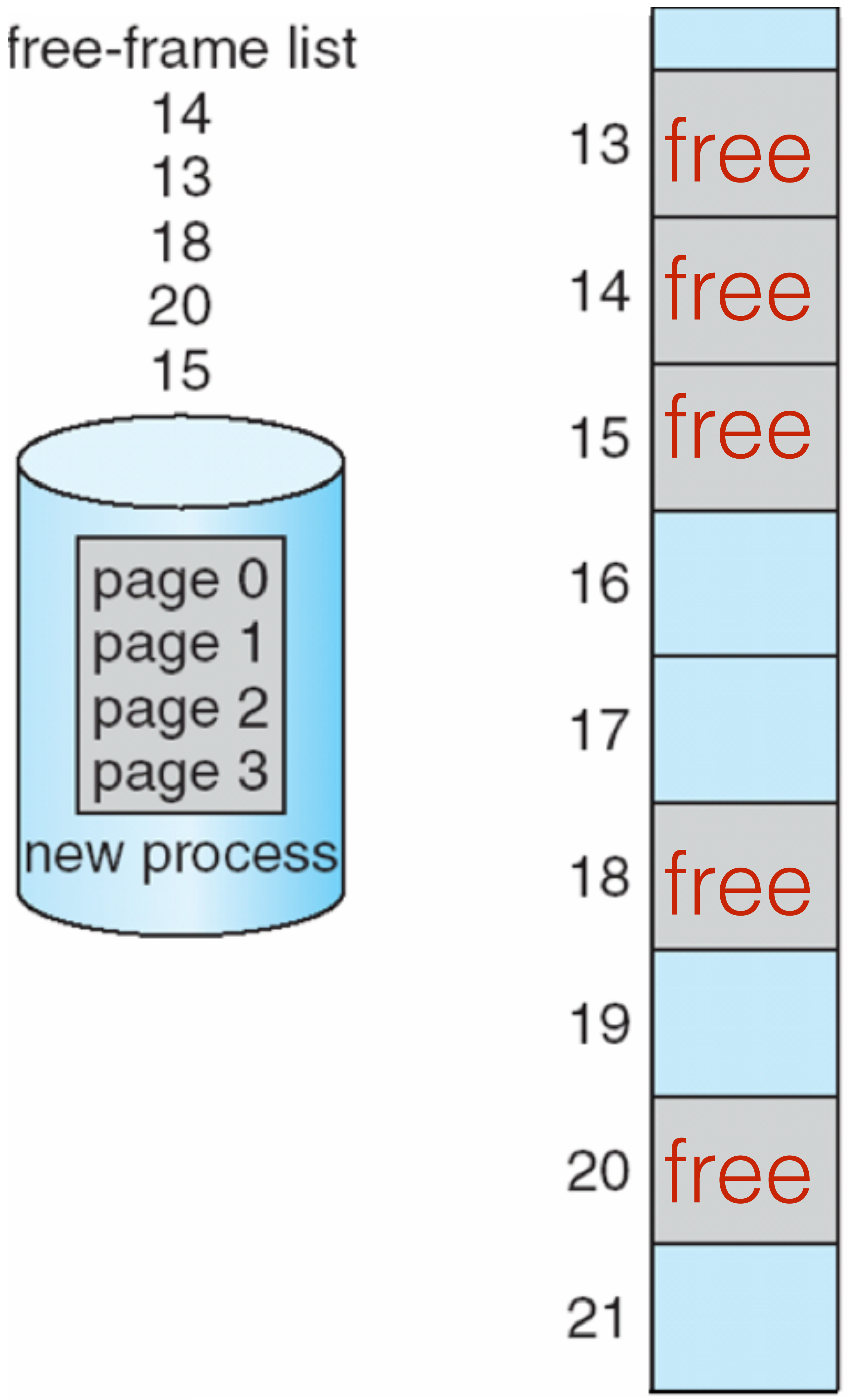
Paging example: 32-byte memory and 4-byte pages



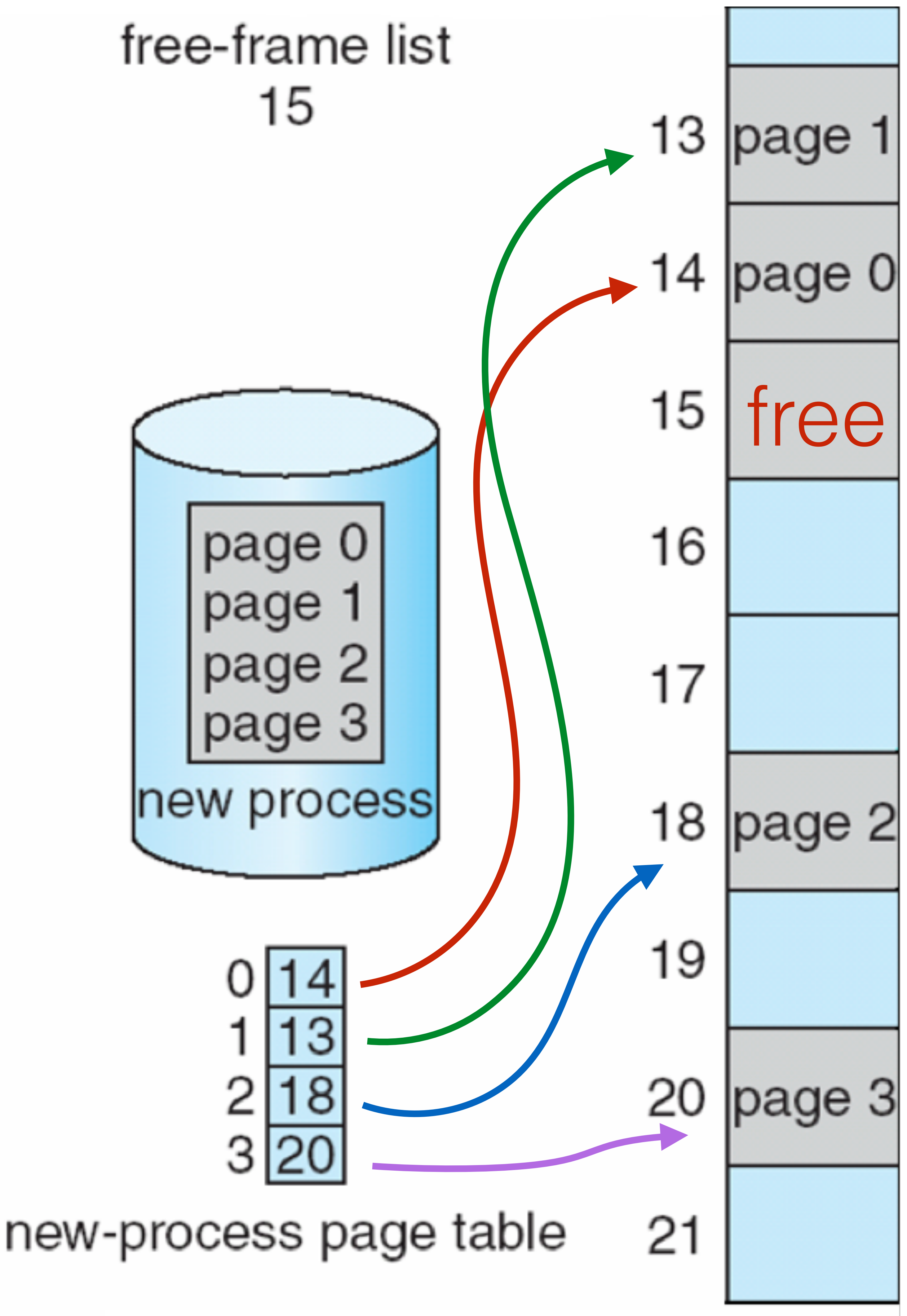
Paging example: 32-byte memory and 4-byte pages (with internal fragmentation)



New process is executed: free frames before and after allocation



Before allocation



After allocation

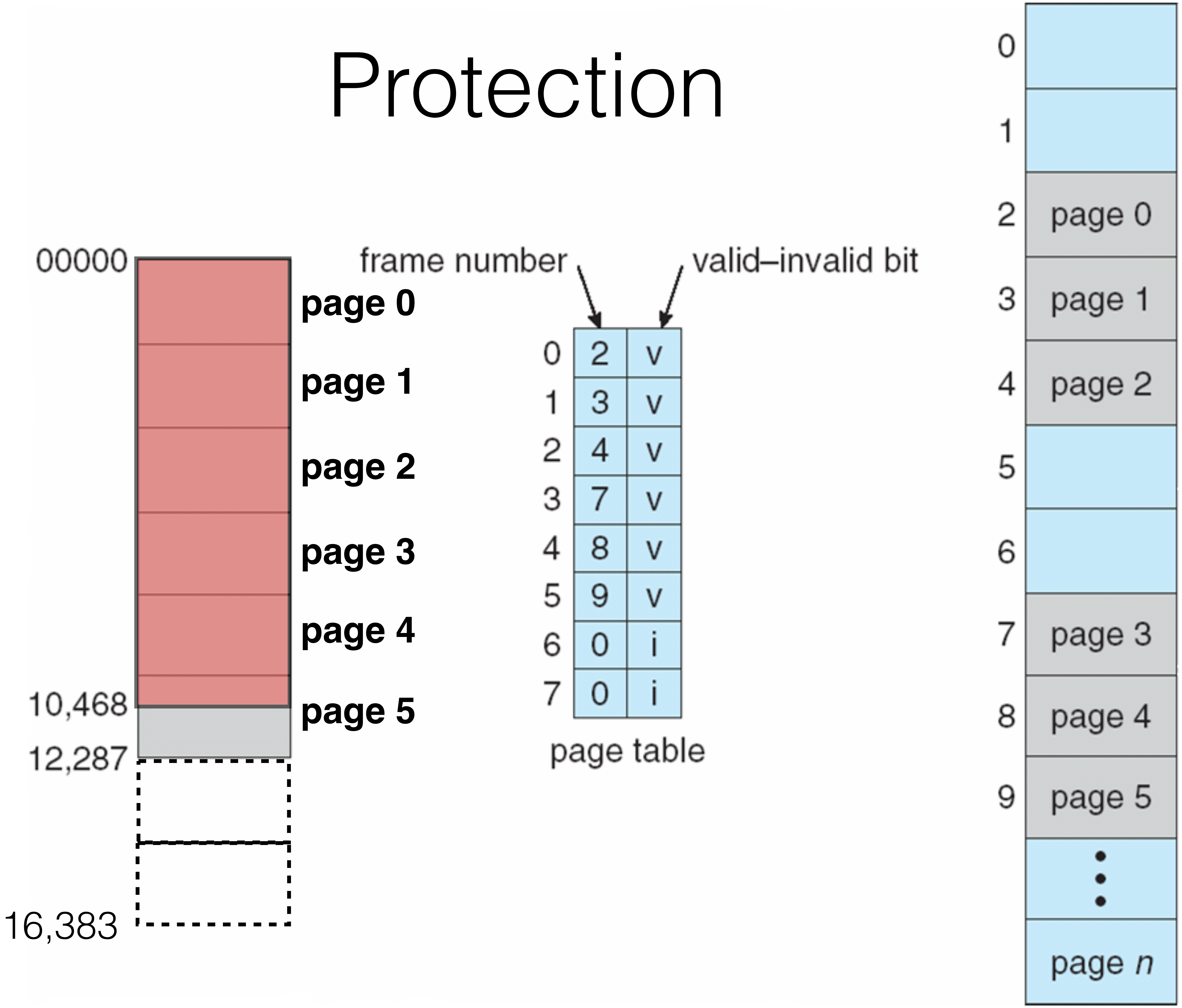
Paging Limitations - Space

- 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
10,468
page 5
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

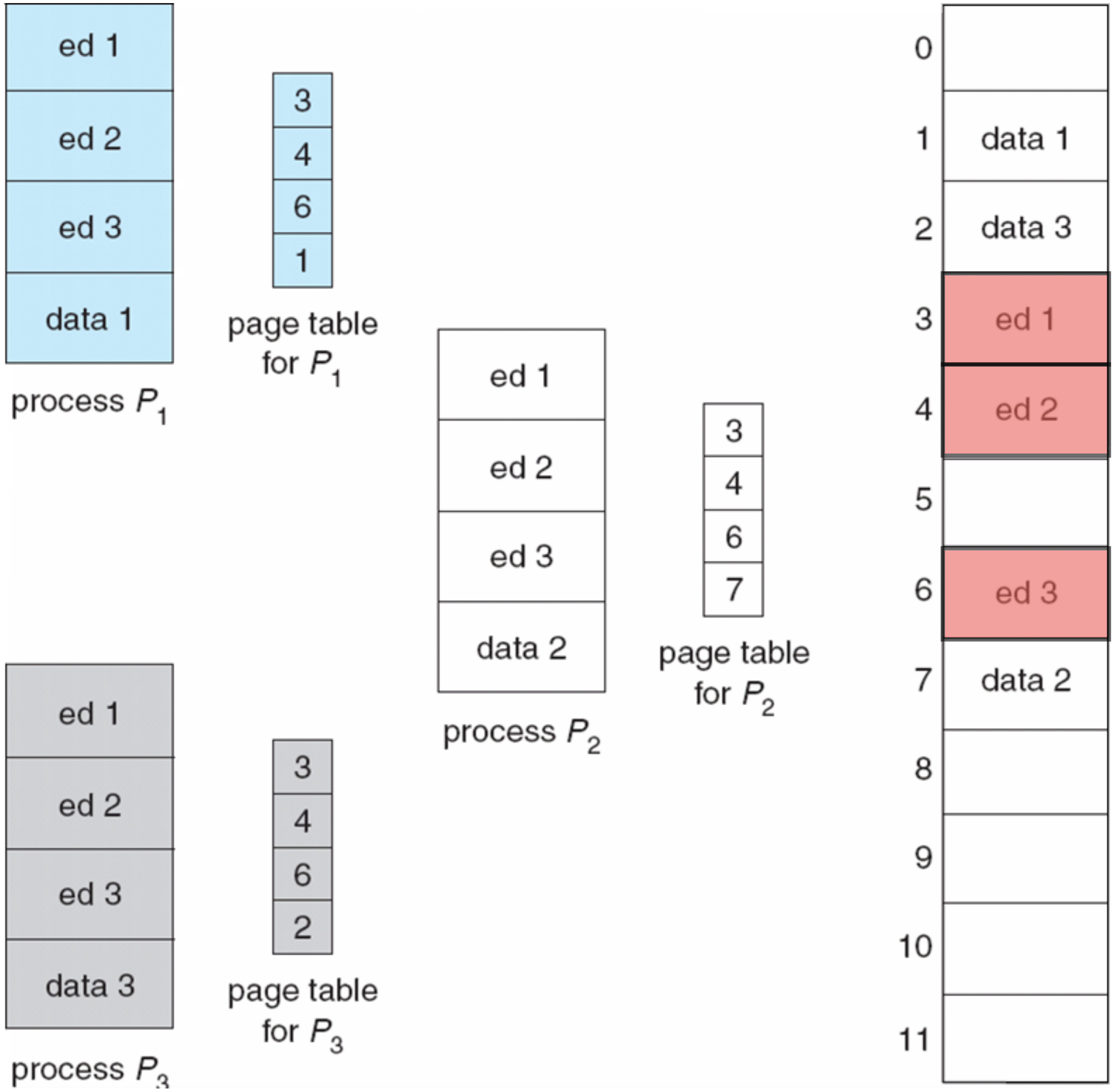
A few more useful aspects of paging

- 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( )`

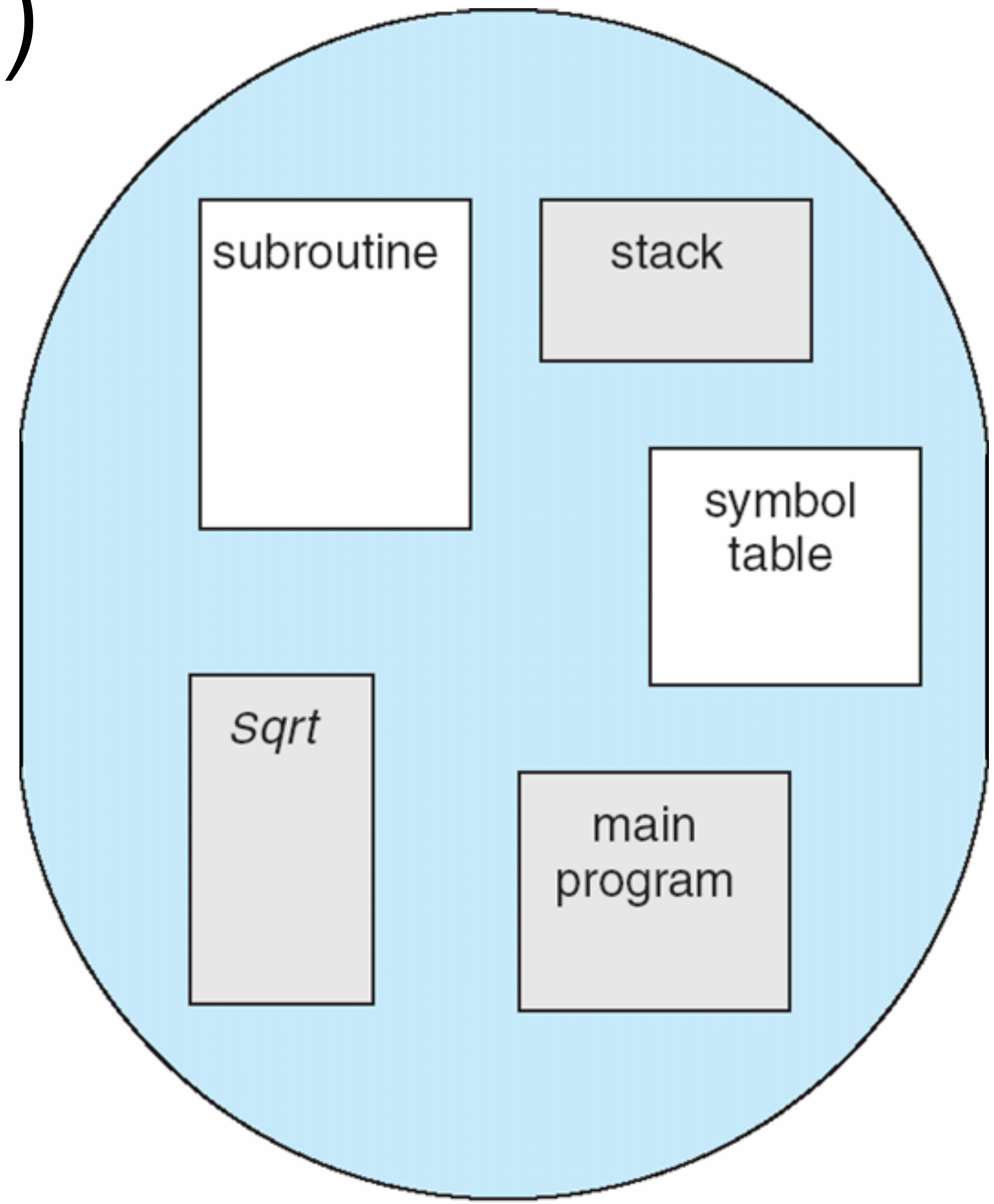
- copy-on-write (COW), e.g., on `fork( )`
  - Instead of copying all pages, create shared mappings of parent pages in child address space
    A. Make shared mappings read-only in child space
    B. When child does a write, a protection fault occurs, OS takes over and can then copy the page and resume child.

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.

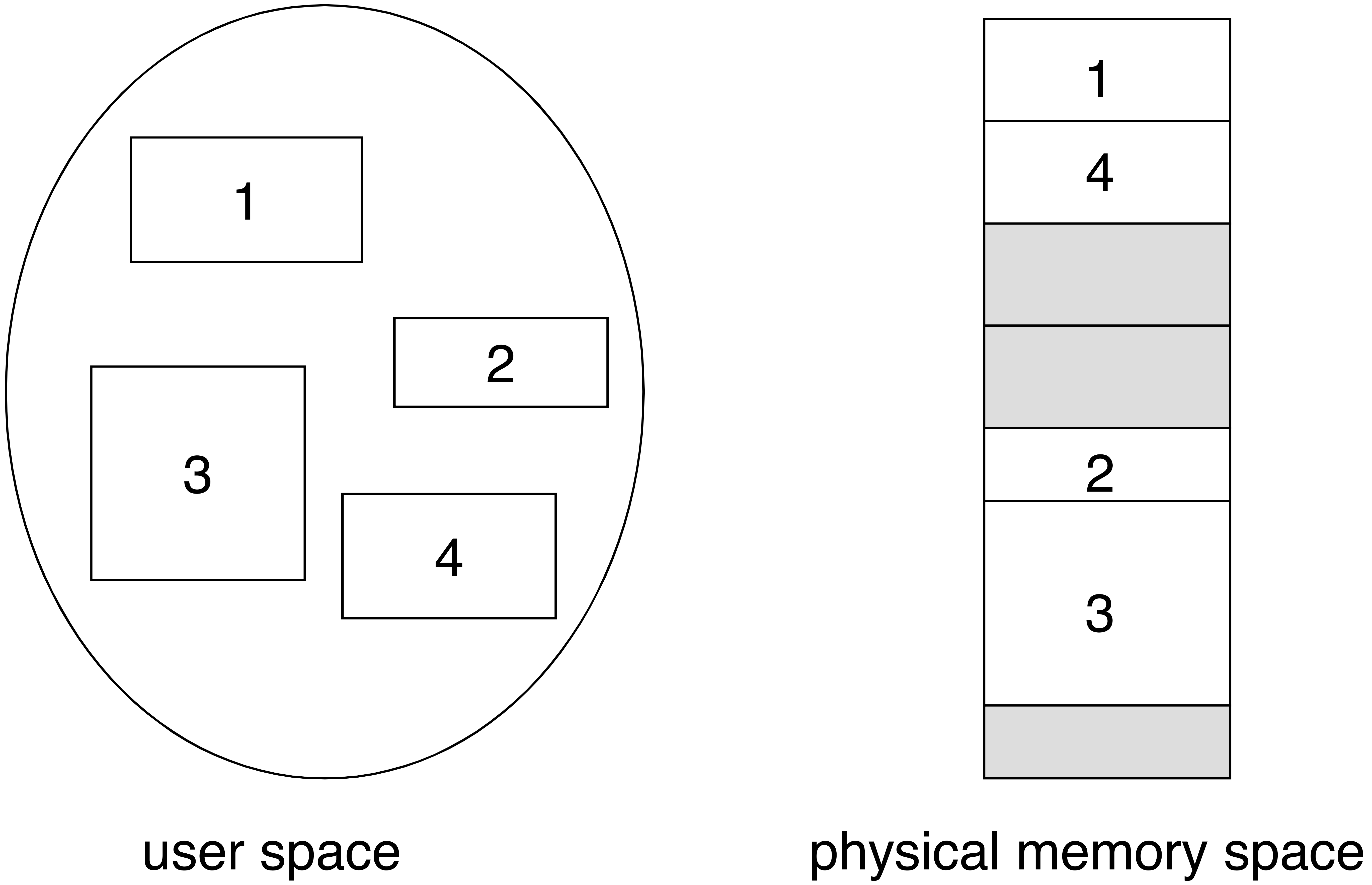


logical address

Logical view of segmentation

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

`<segment-number, offset>`



Segmentation Hardware

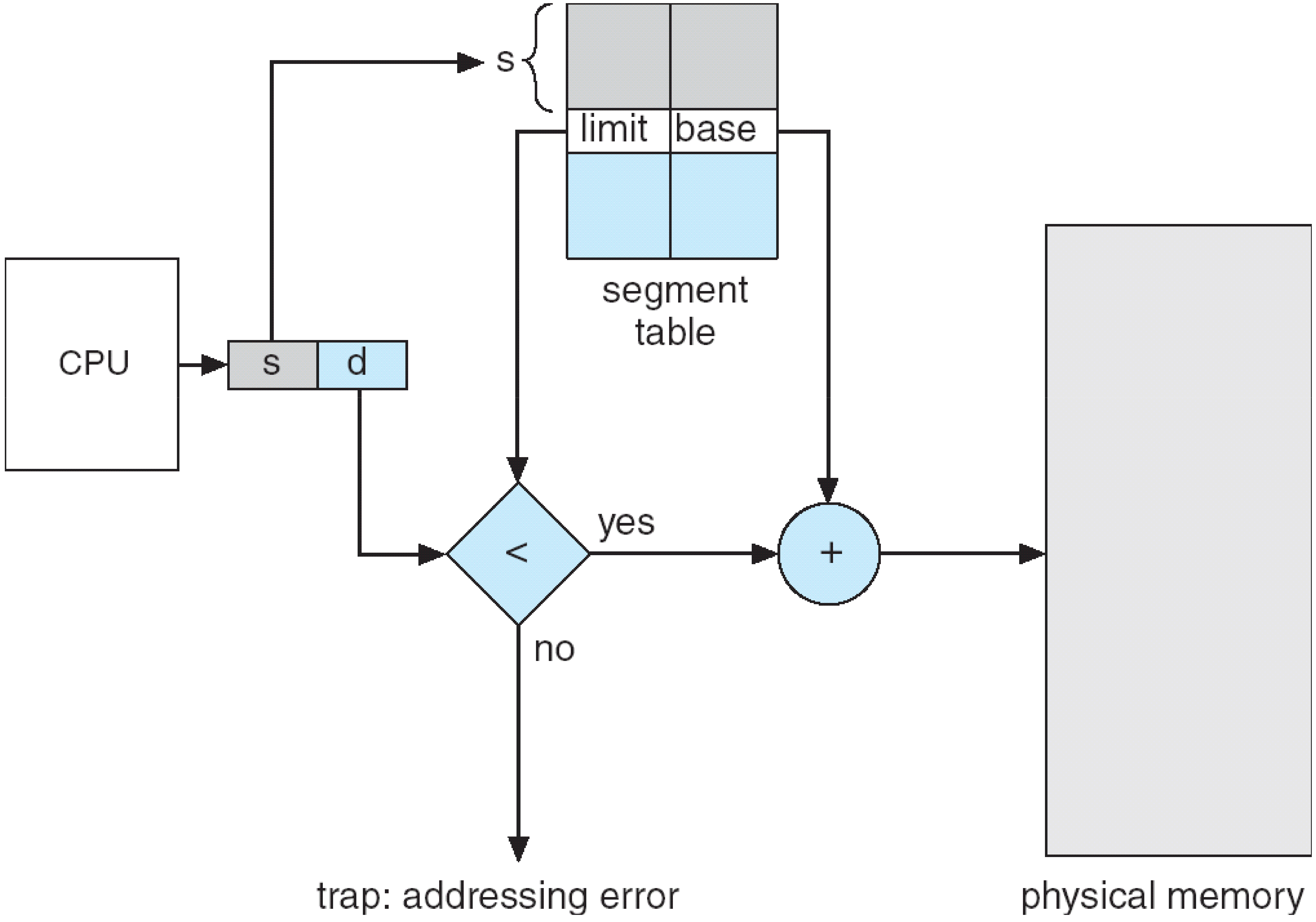
Segment tables:

Base: starting address of the segment in physical memory.

Limit: length of the segment.

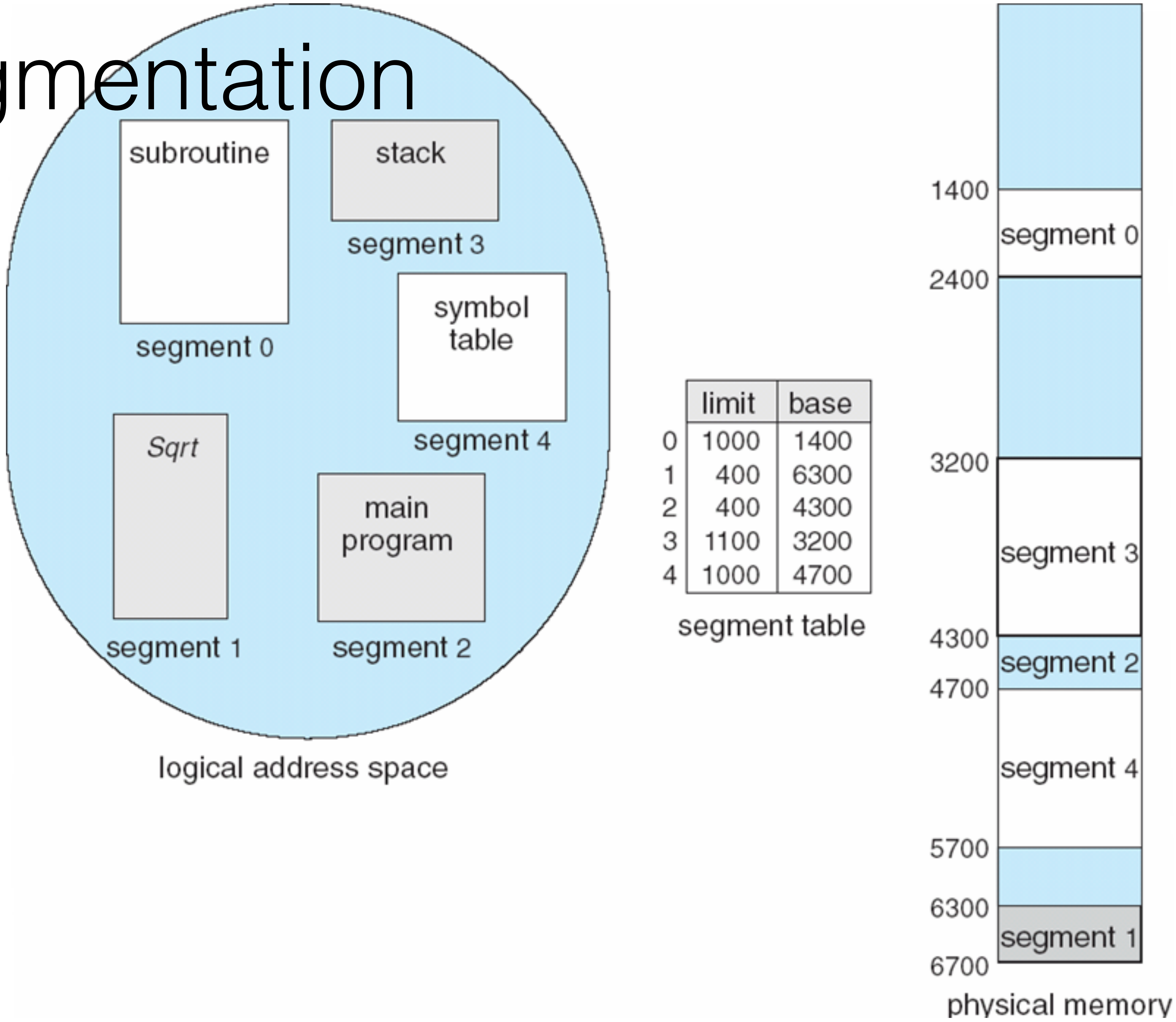
Additional metadata includes protection bits.

$\langle$segment-number, offset$\rangle$



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

Virtual Memory

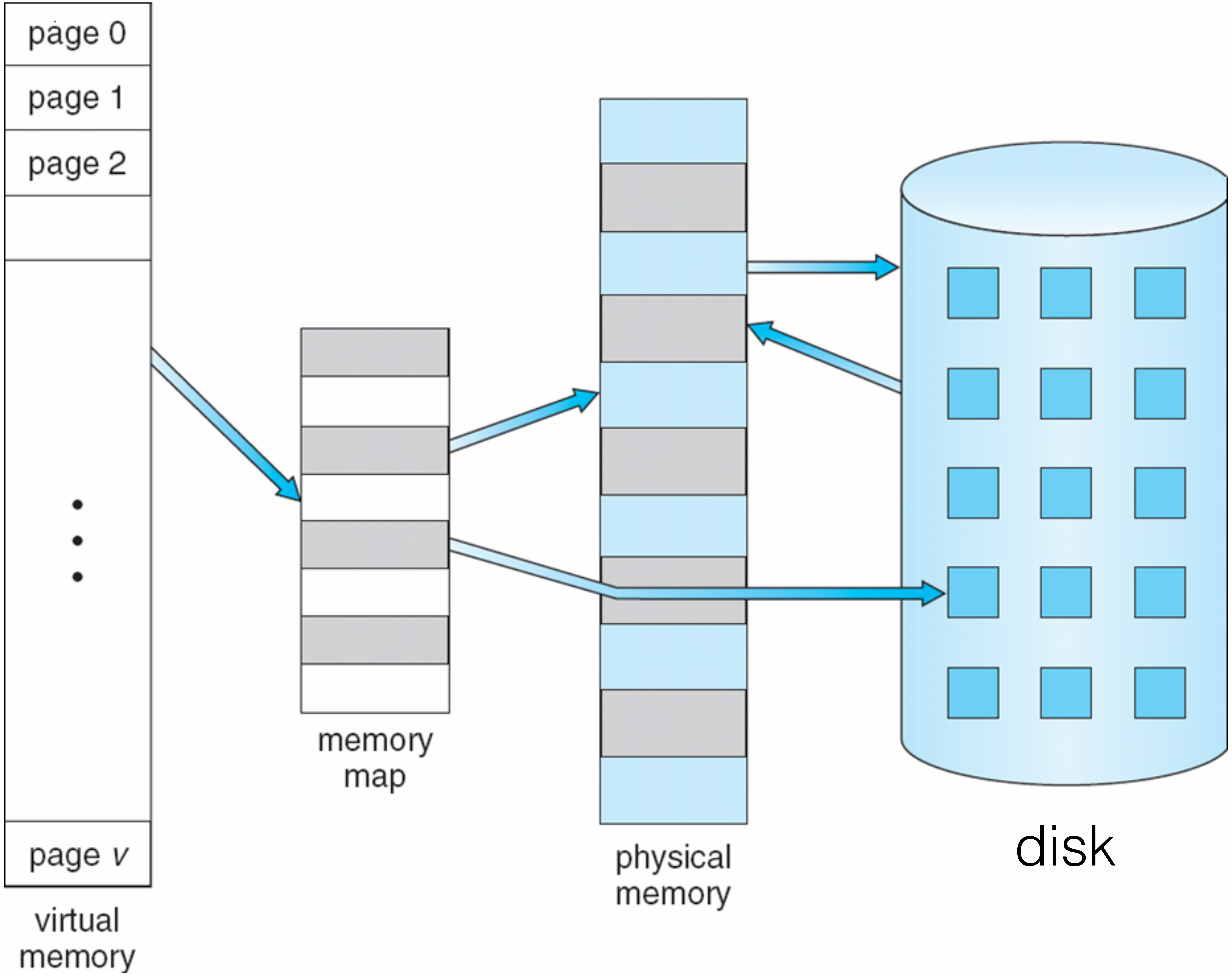
CSE 4001

Content

- Demand paging

Virtual Memory

- Separation of user logical memory from physical memory.
- Programs can be partially in memory for execution
- Logical address space can be much larger than physical address space



Implementation

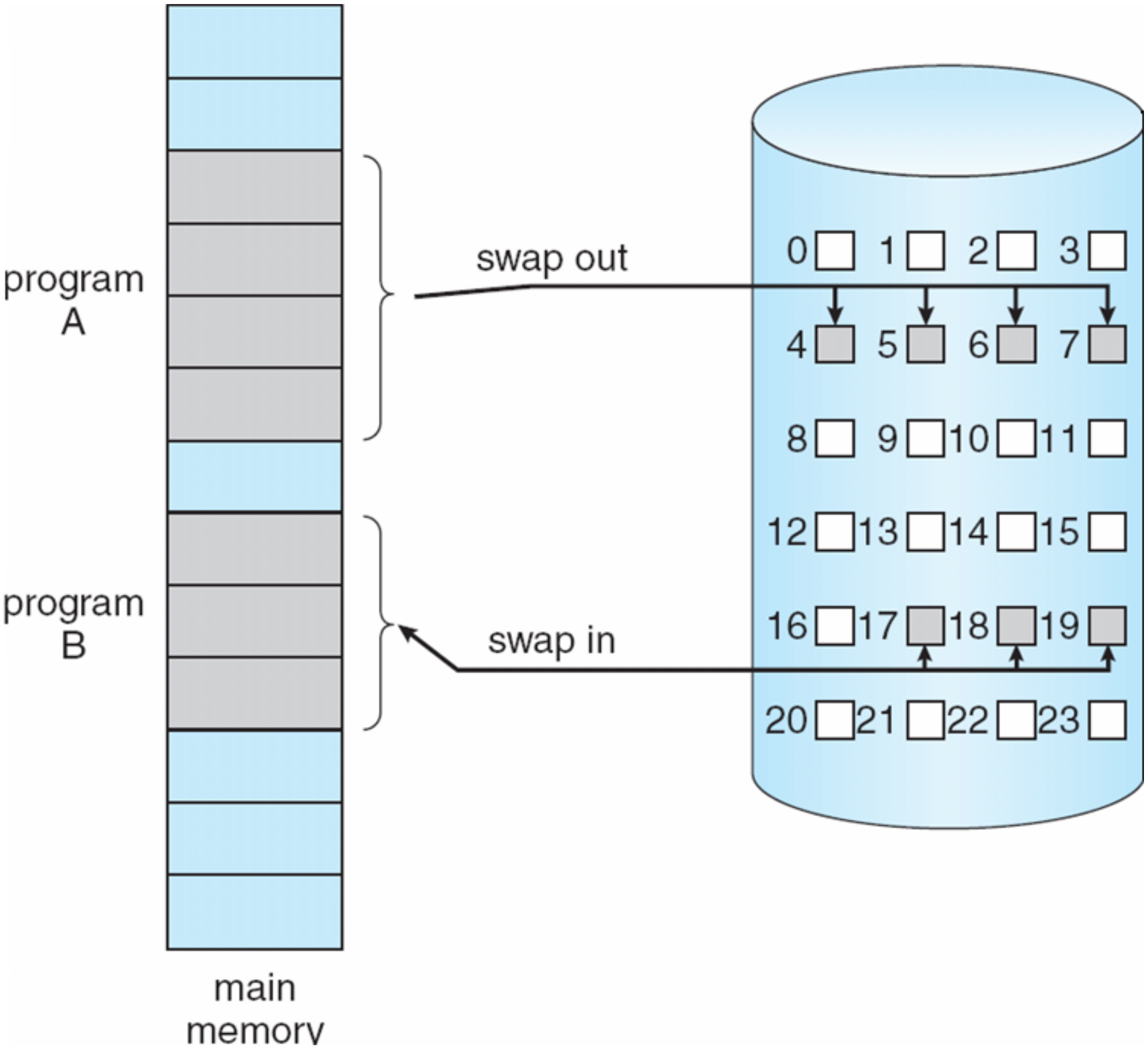
Virtual memory can be implemented via:

- Demand paging
- Demand segmentation

Demand paging

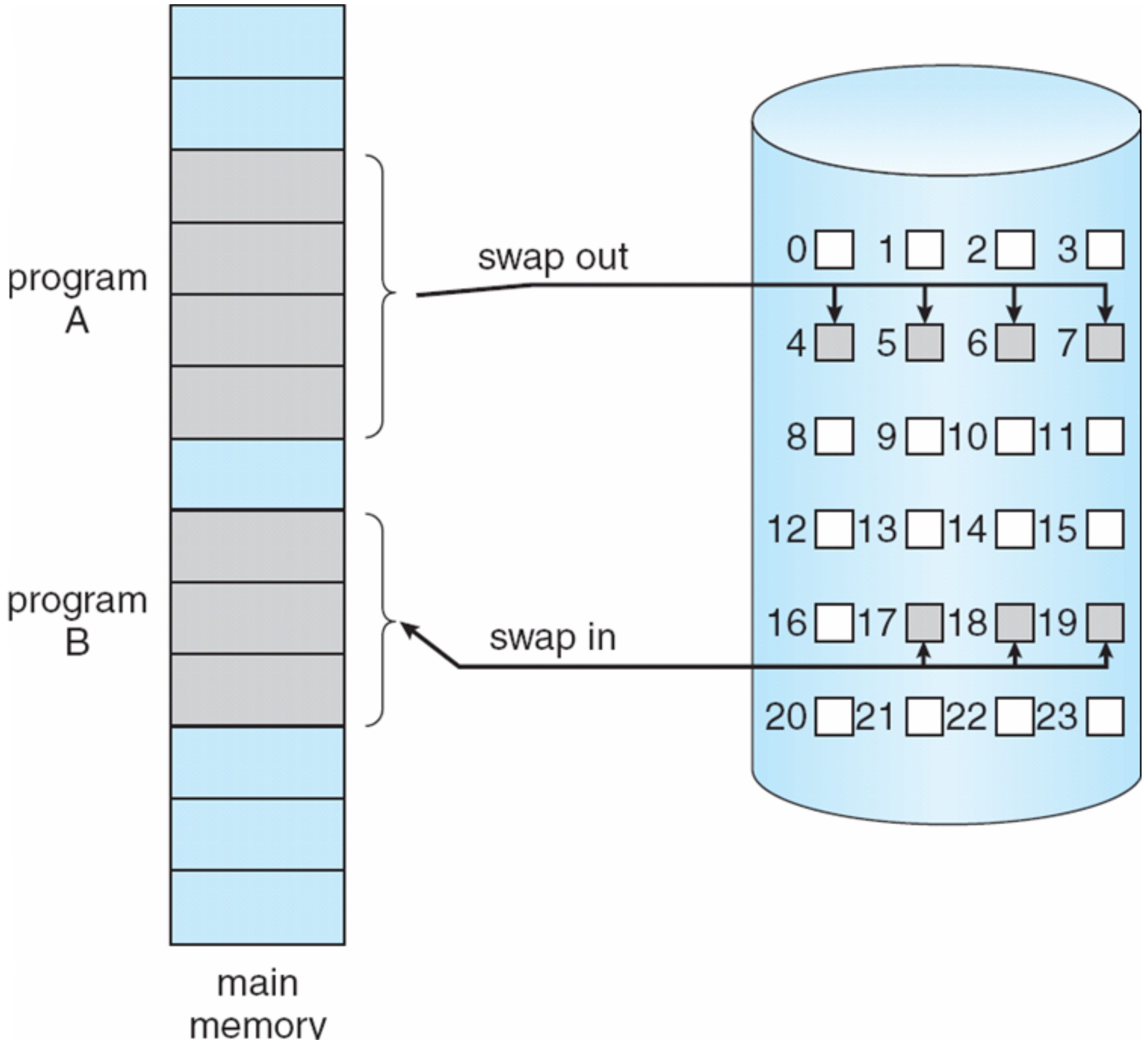
Bring a page into memory only when it is needed:

- Less I/O needed
- Less memory needs
- Faster response
- More users



Demand paging

- Demand paging is similar to a paging system with swapping, where processes reside in secondary memory (e.g., disk).
- Lazy swapper: only bring pages when they are needed.
- In the context of demand paging, we use the term pager instead of swapper.

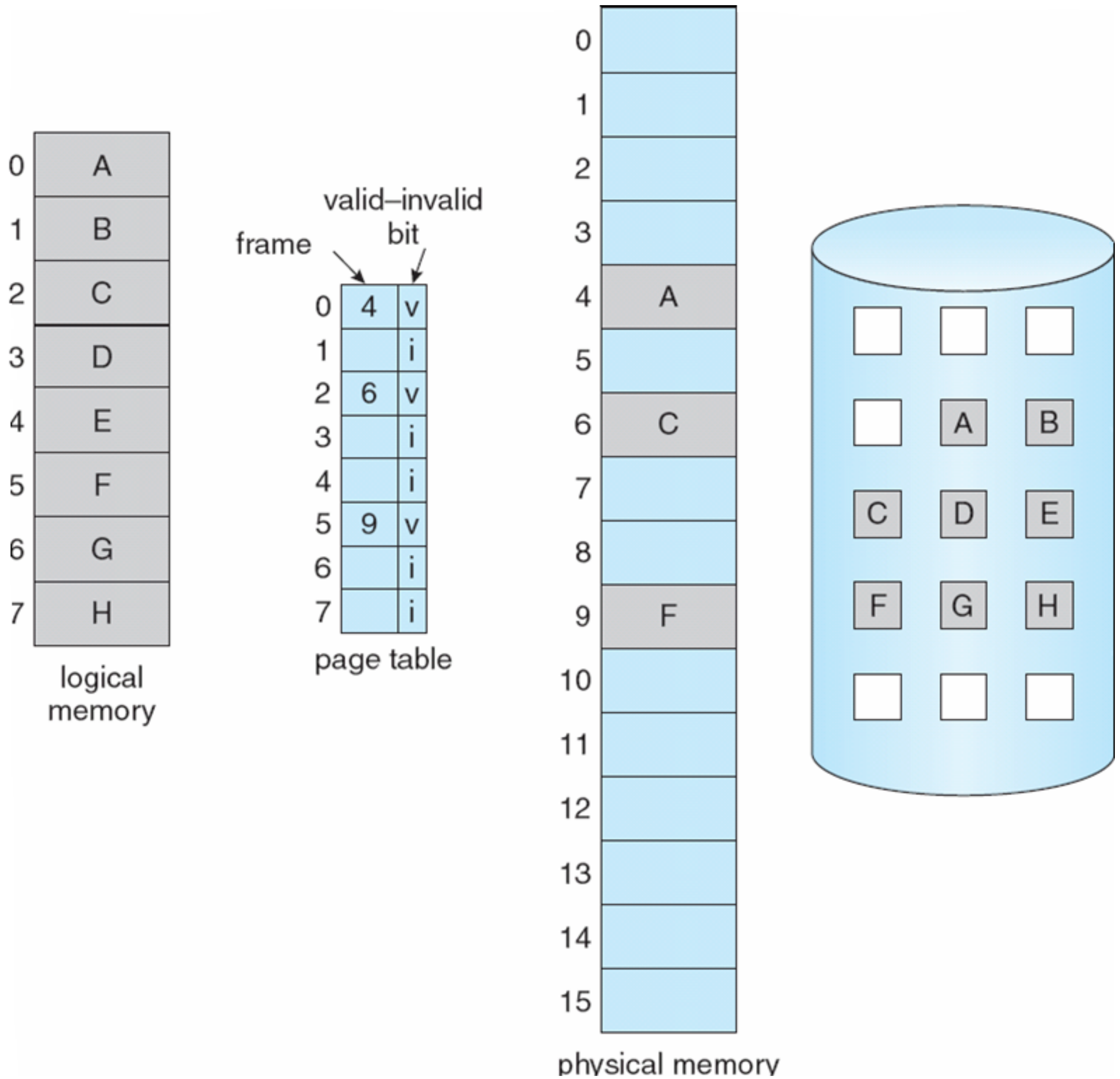


Valid-Invalid Bit

- Hardware support is needed to distinguish between the pages that are in memory and the ones that are on the disk.
- We can re-use the support provided by the valid-invalid bit in the page table.
  - Bit == valid then page is in memory (and is valid).
  - Bit == invalid then page is either not a valid one for that process or is valid but is currently in disk (pager needs to bring it to main memory).

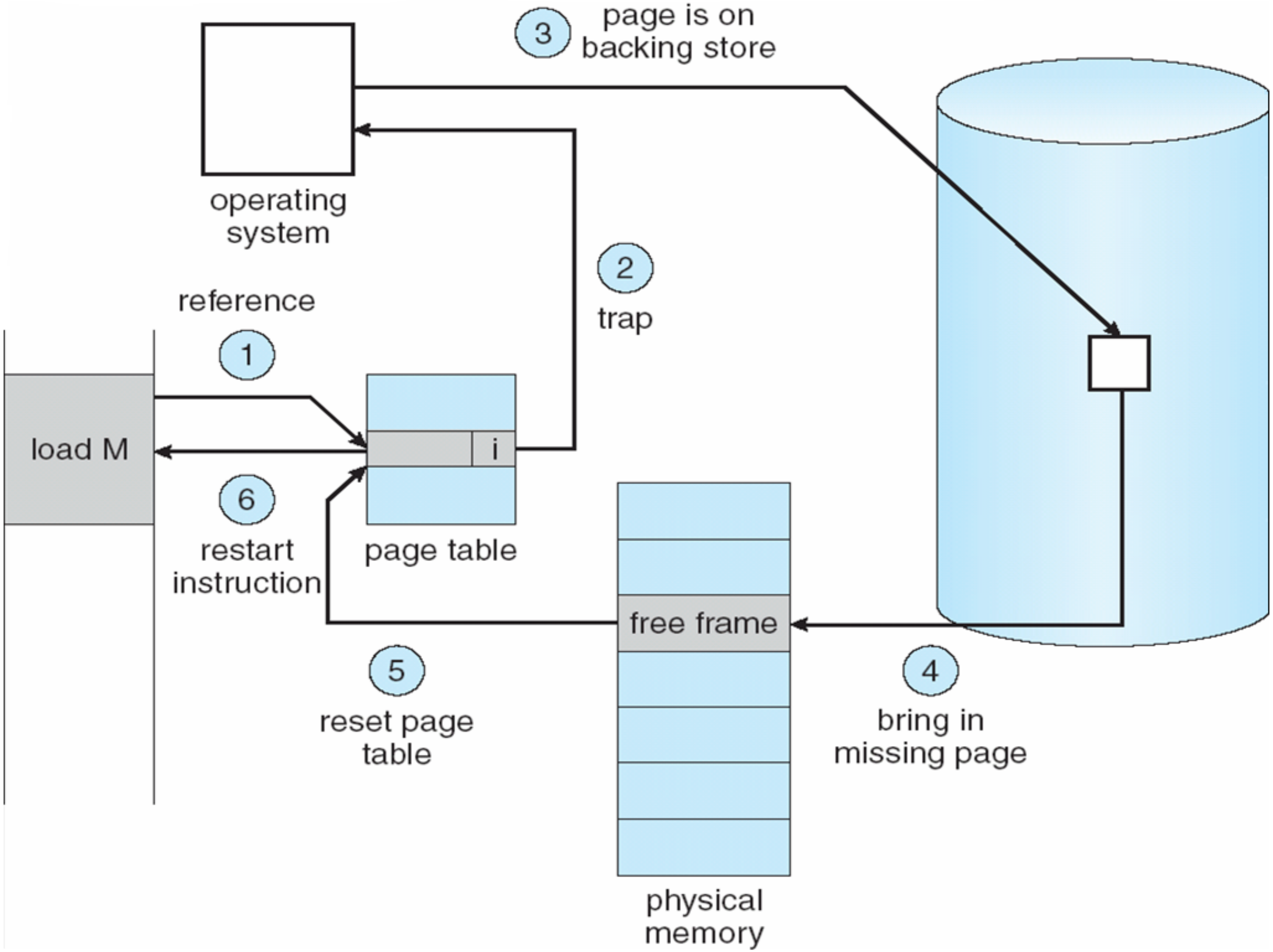
Valid-Invalid Bit

- Marking a page invalid has no effect if the process never attempts to access that page.
- Pages that are in memory are called memory resident.

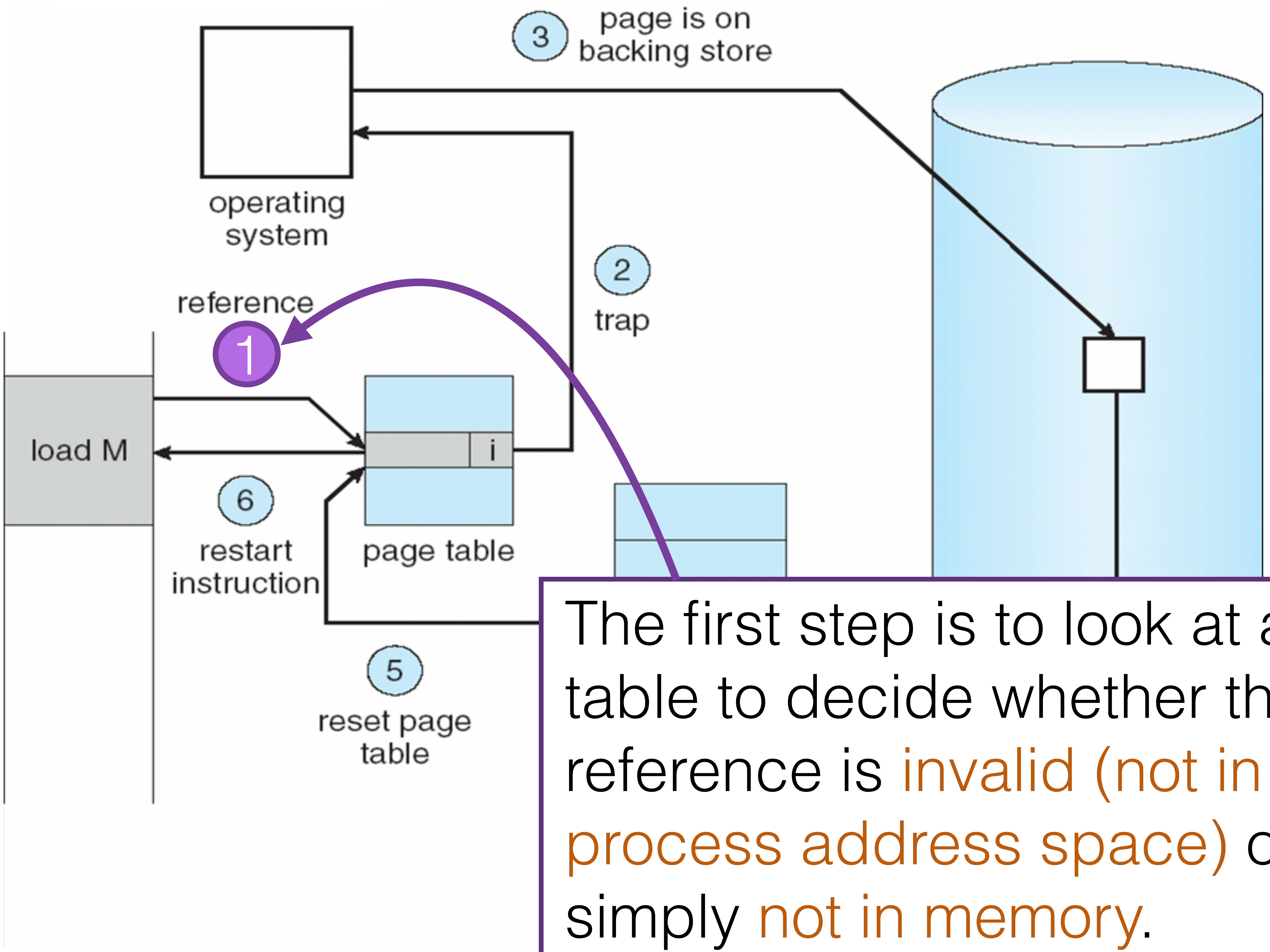


Page Faults

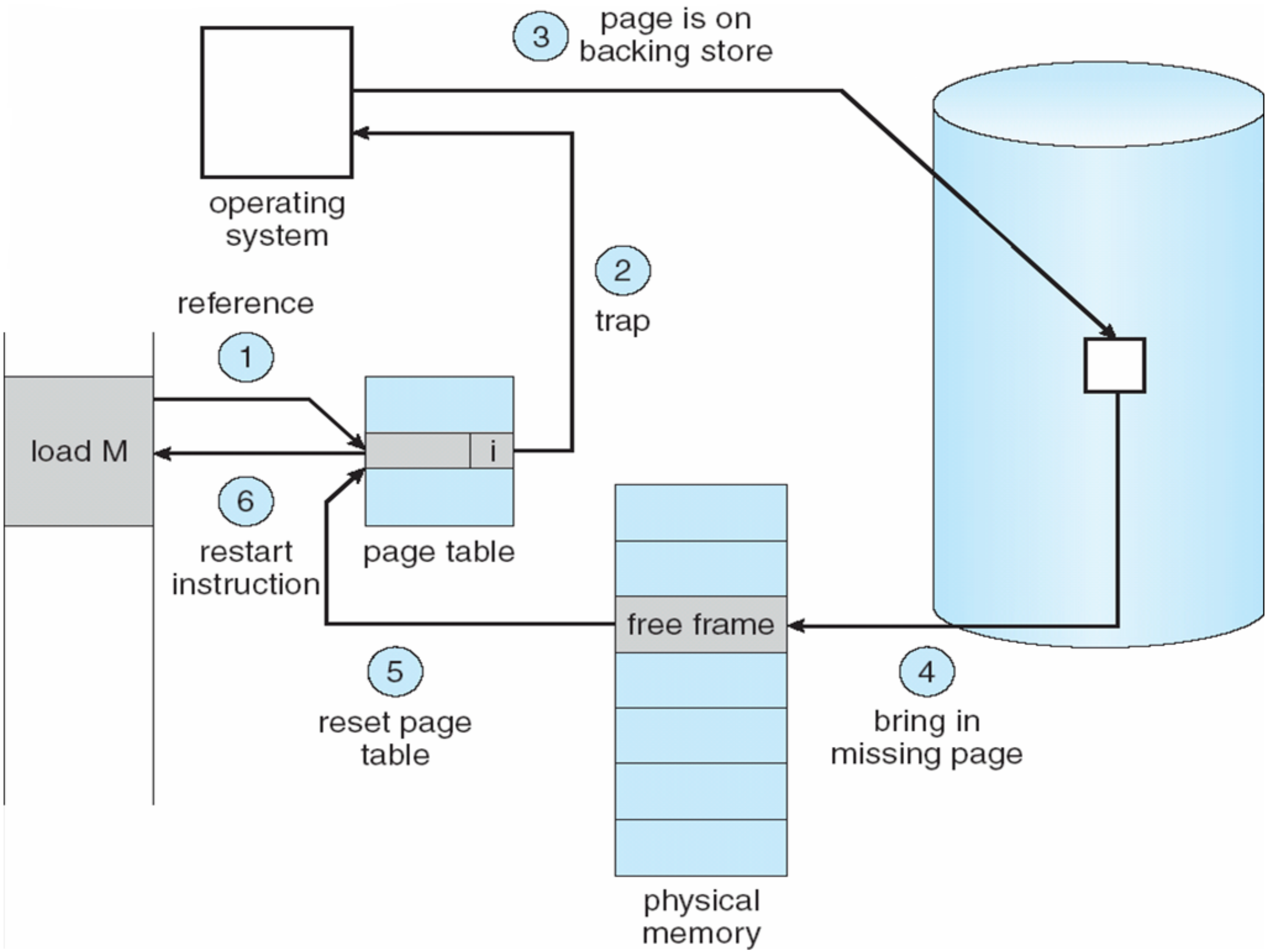
- What happens when a process tries to access non-resident pages?
  - Page Fault: A trap that results because the OS's failed to bring the desired page into memory.



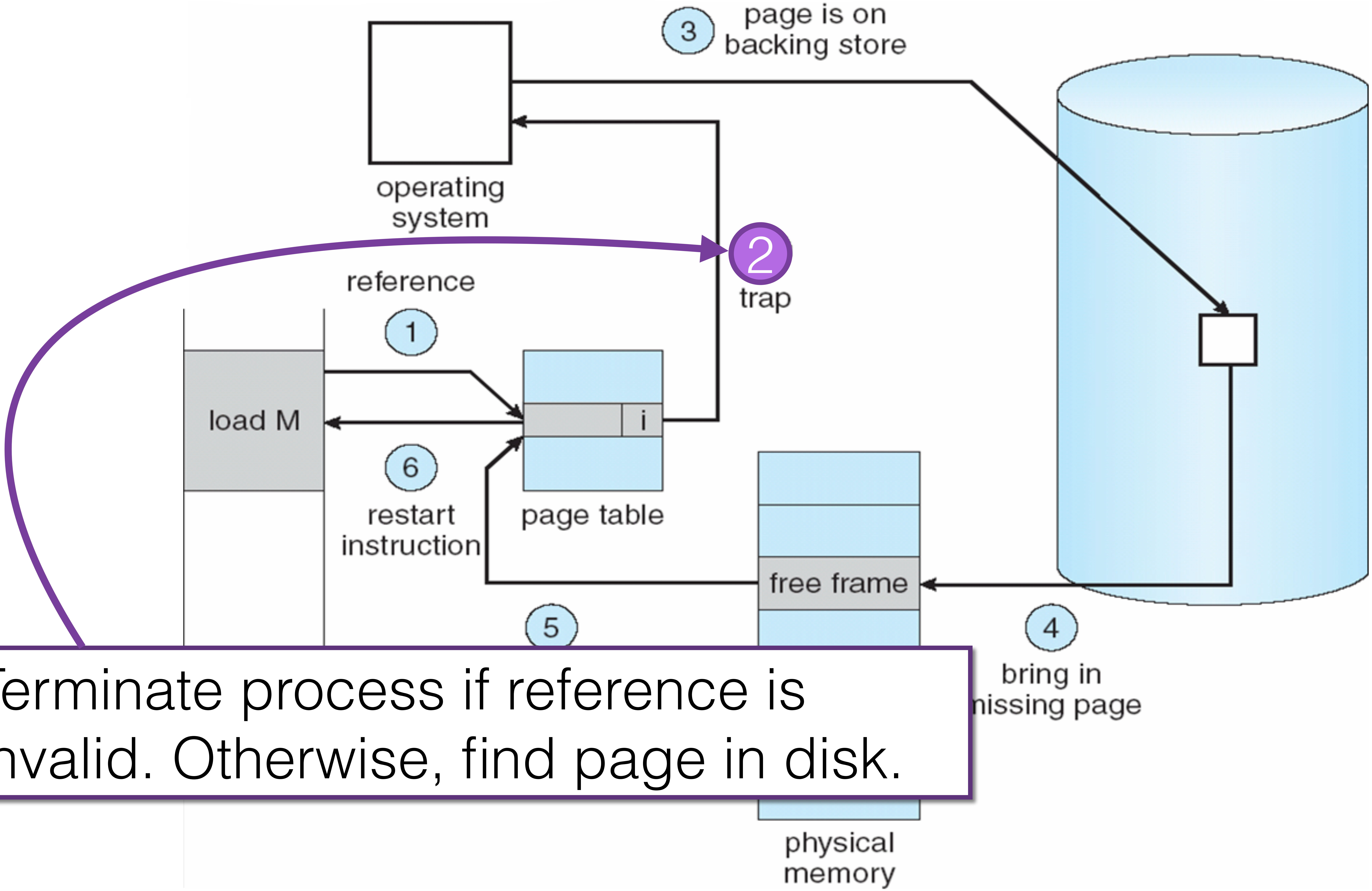
page is on
backing store
operating
system
trap
reference
load M
i
restart
instruction
page table
free frame
reset page
table
bring in
missing page
physical
memory
1
2
3
4
5
6



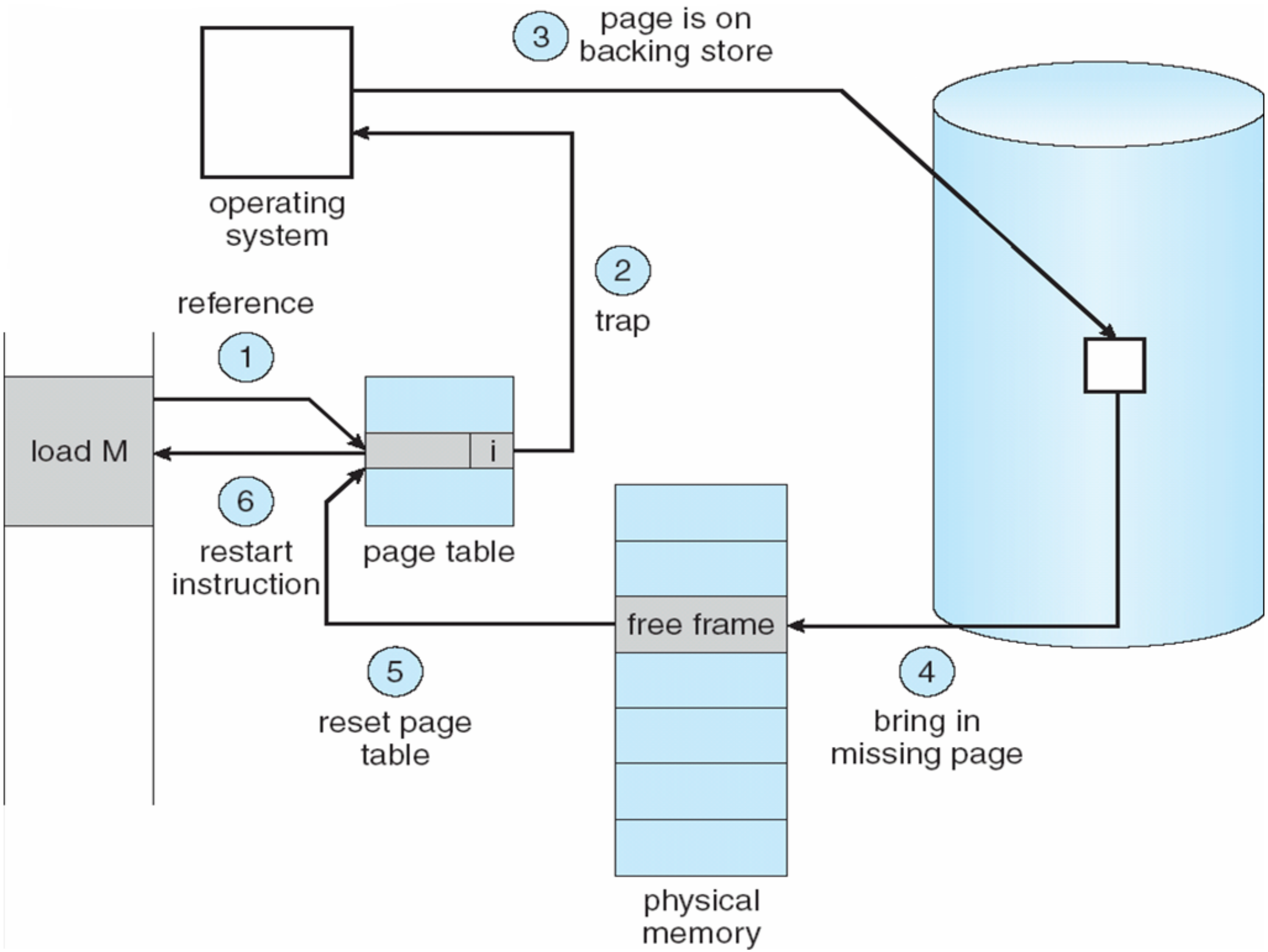
The first step is to look at another table to decide whether the actual reference is invalid (not in the process address space) or is simply not in memory.



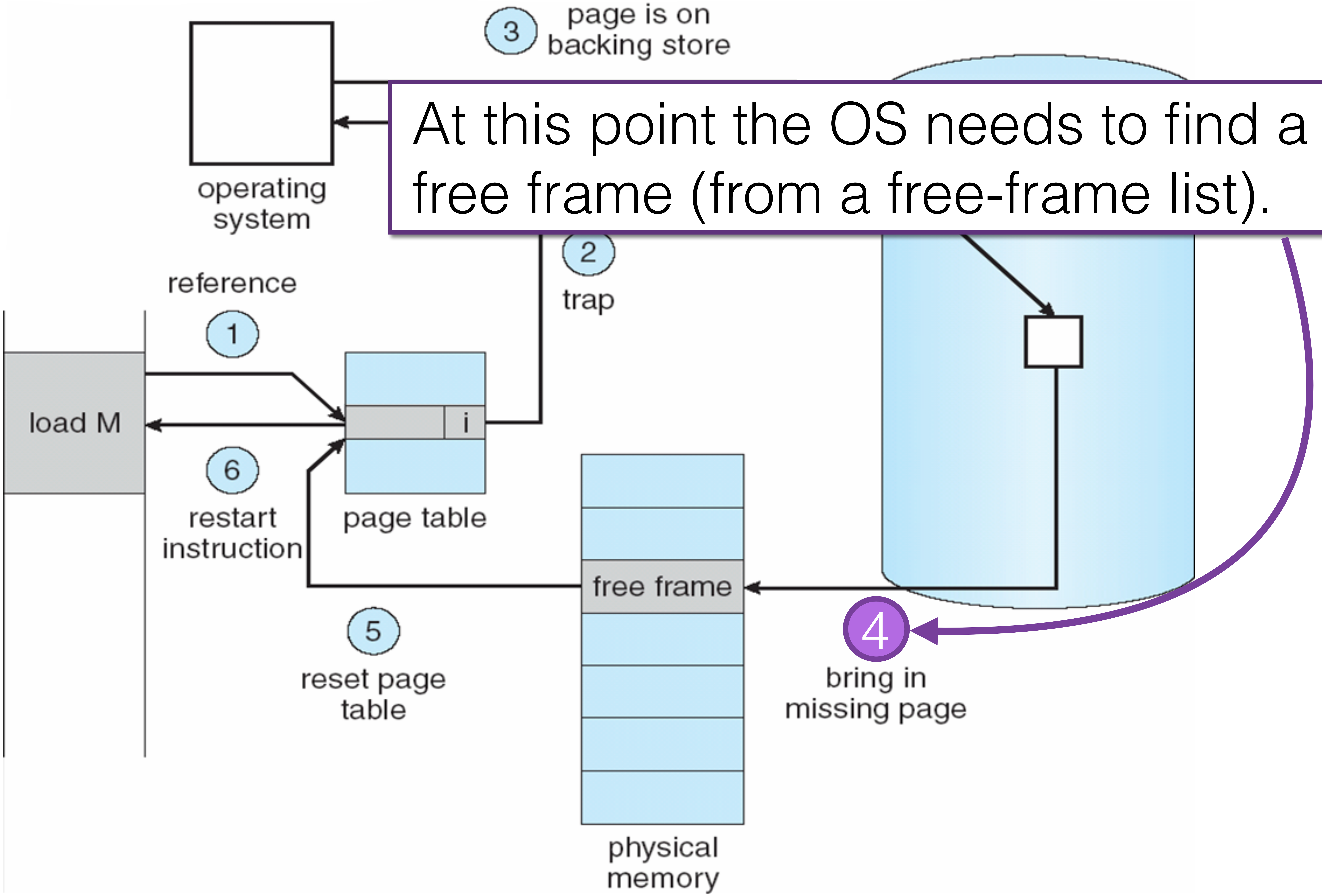
page is on backing store
operating system
reference
trap
load M
i
page table
restart instruction
free frame
reset page table
bring in missing page
physical memory
1
2
3
4
5
6

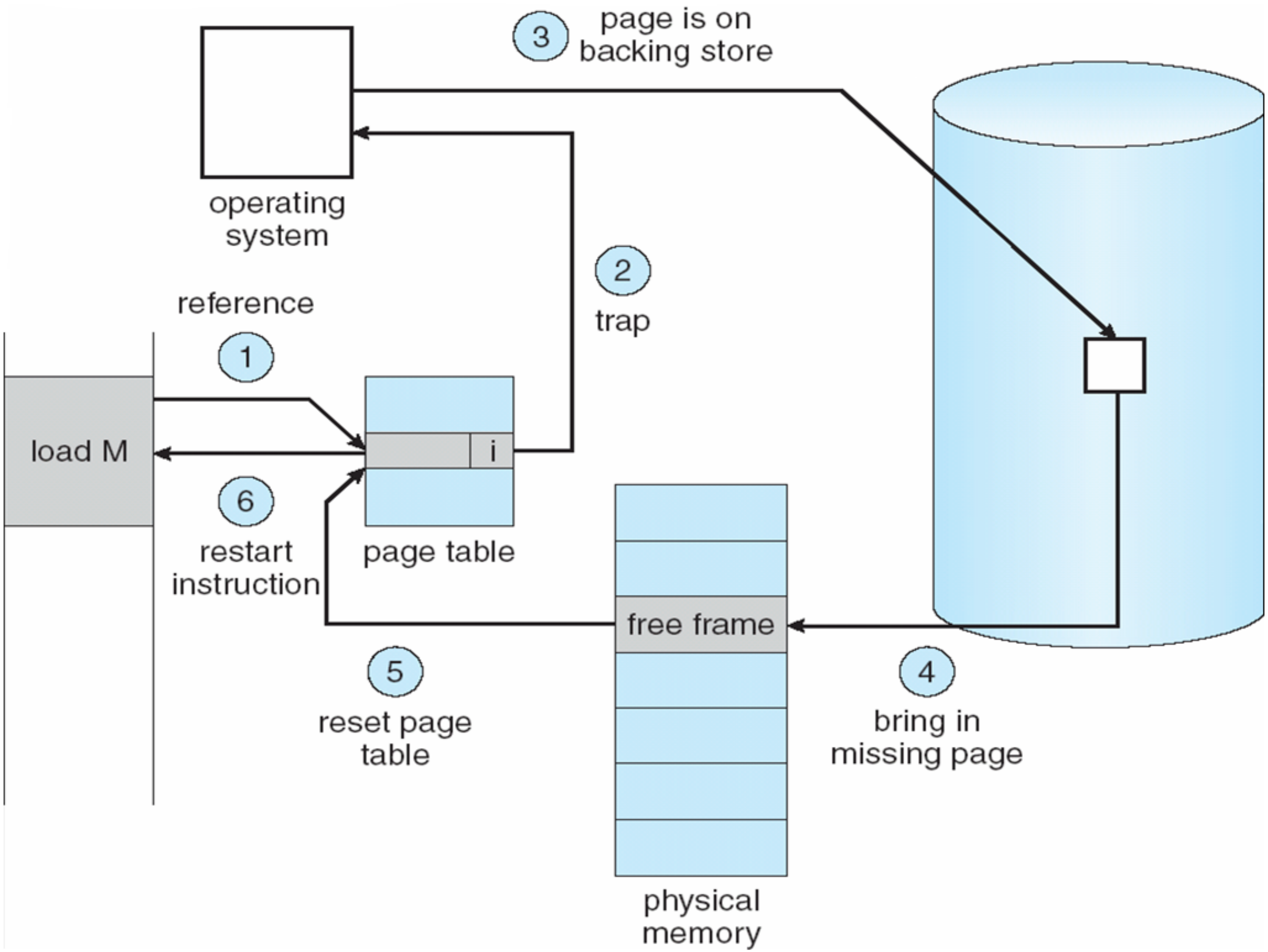


Terminate process if reference is invalid. Otherwise, find page in disk.


page is on
backing store
3
operating
system
2
trap
reference
1
load M
i
6
restart
instruction
page table
free frame
5
reset page
table
4
bring in
missing page
physical
memory

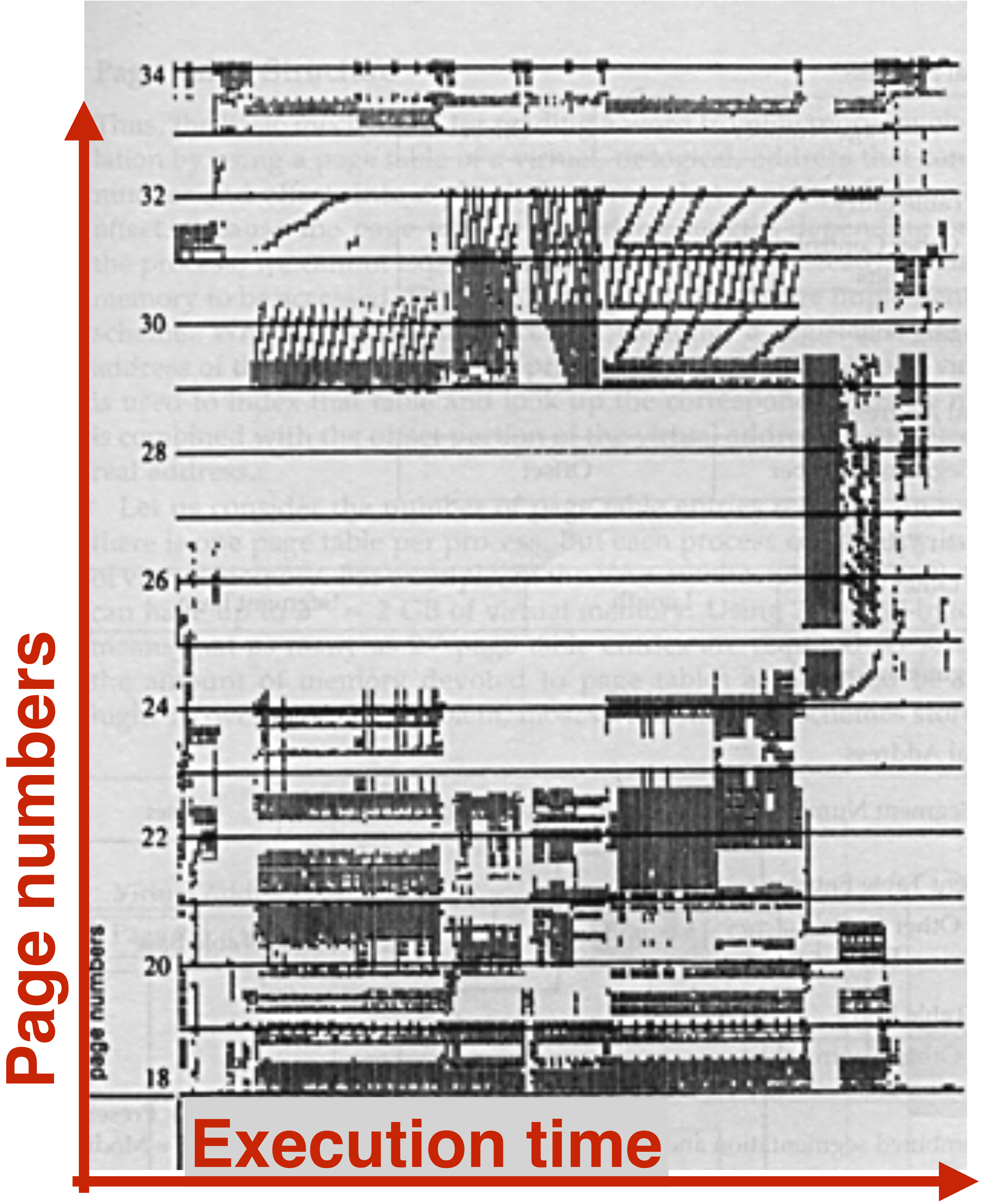
At this point the OS needs to find a free frame (from a free-frame list).




page is on
backing store
3
operating
system
2
trap
reference
1
load M
i
6
restart
instruction
page table
free frame
5
reset page
table
4
bring in
missing page
physical
memory

Locality in memory-reference pattern

- Theoretically, some programs could access several new pages with a single instruction.
- In this case, system performance could be seriously degraded.
- Luckily, this behavior is unlikely.



Writing code with demand-paging in mind…

- Program structure
  - `Int[128,128] data;`
  - Each row is stored in one page
  - Program 1

```
for (j = 0; j <128; j++)
    for (i = 0; i < 128; i++)
        data[i,j] = 0;
```

128 x 128 = 16,384 page faults

- Program 2

```
for (i = 0; i < 128; i++)
    for (j = 0; j < 128; j++)
        data[i,j] = 0;
```

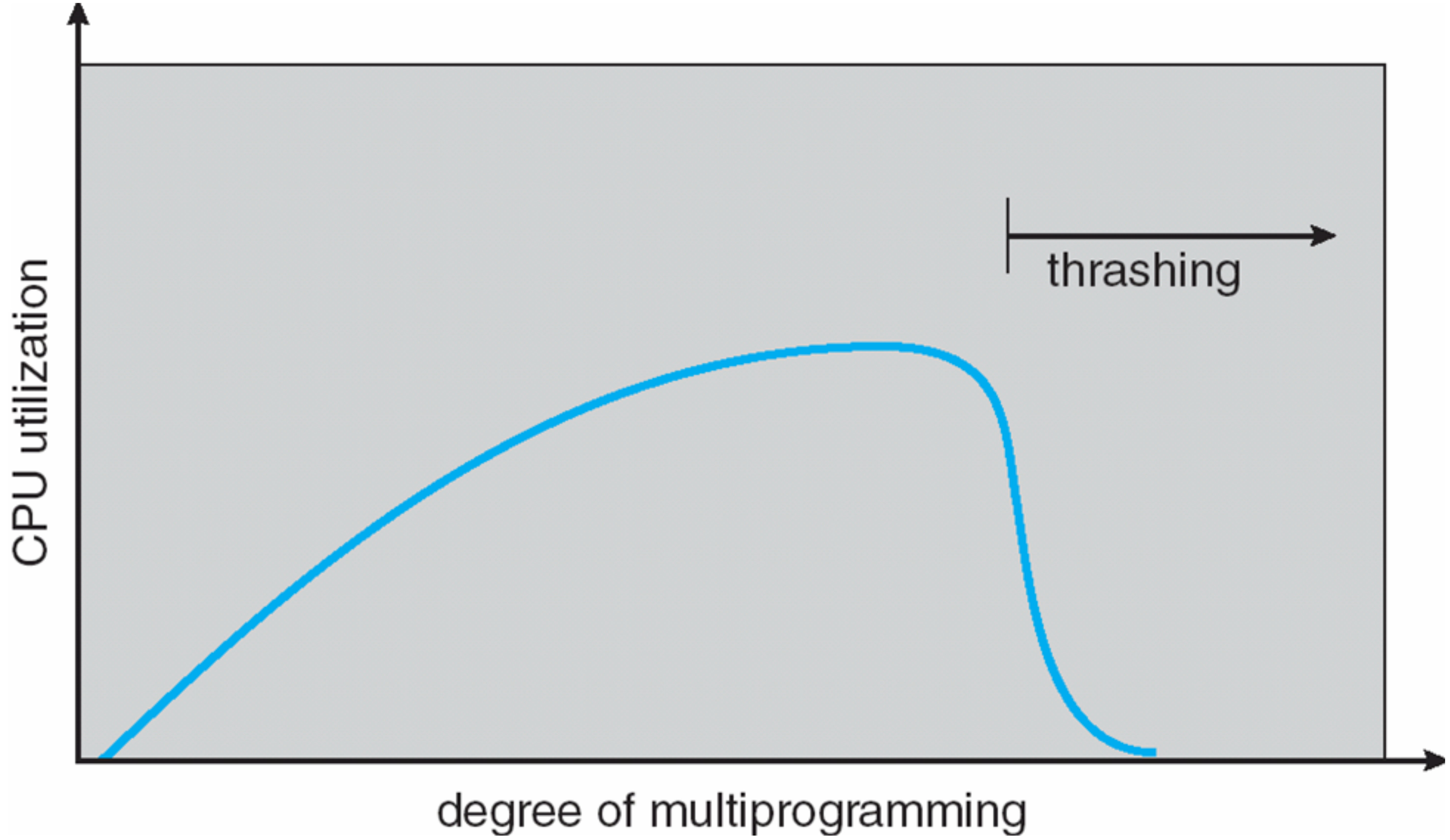
128 page faults

Thrashing

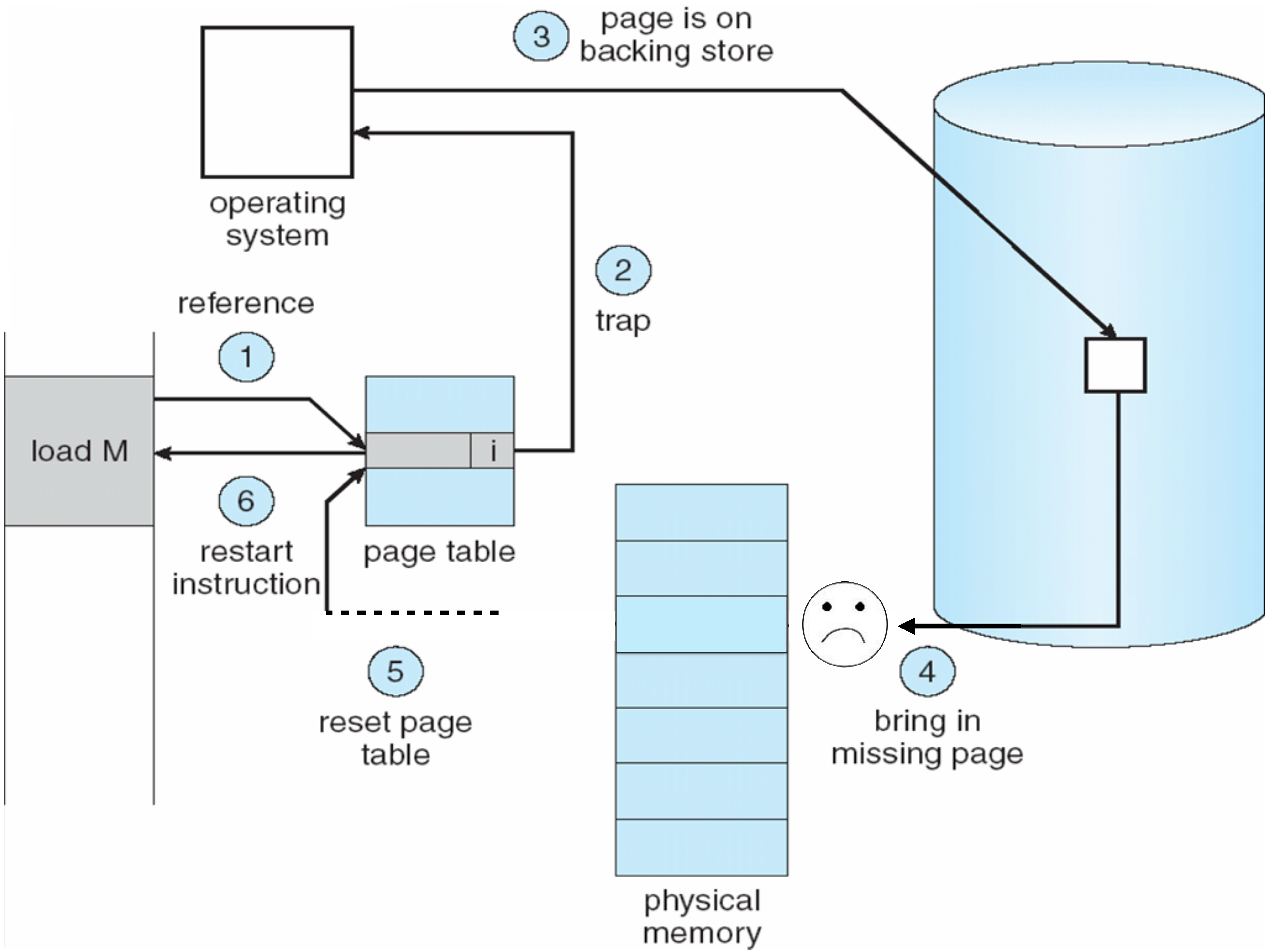
- The process does not have “enough” pages, the page-fault rate is very high and CPU becomes sub-utilized.
- The OS wants to maximize CPU utilization. As a result, it decides that it is a good idea to increase the degree of multiprogramming by adding new processes to the system.
- Thrashing: A process is spending more time paging that executing.

Thrashing

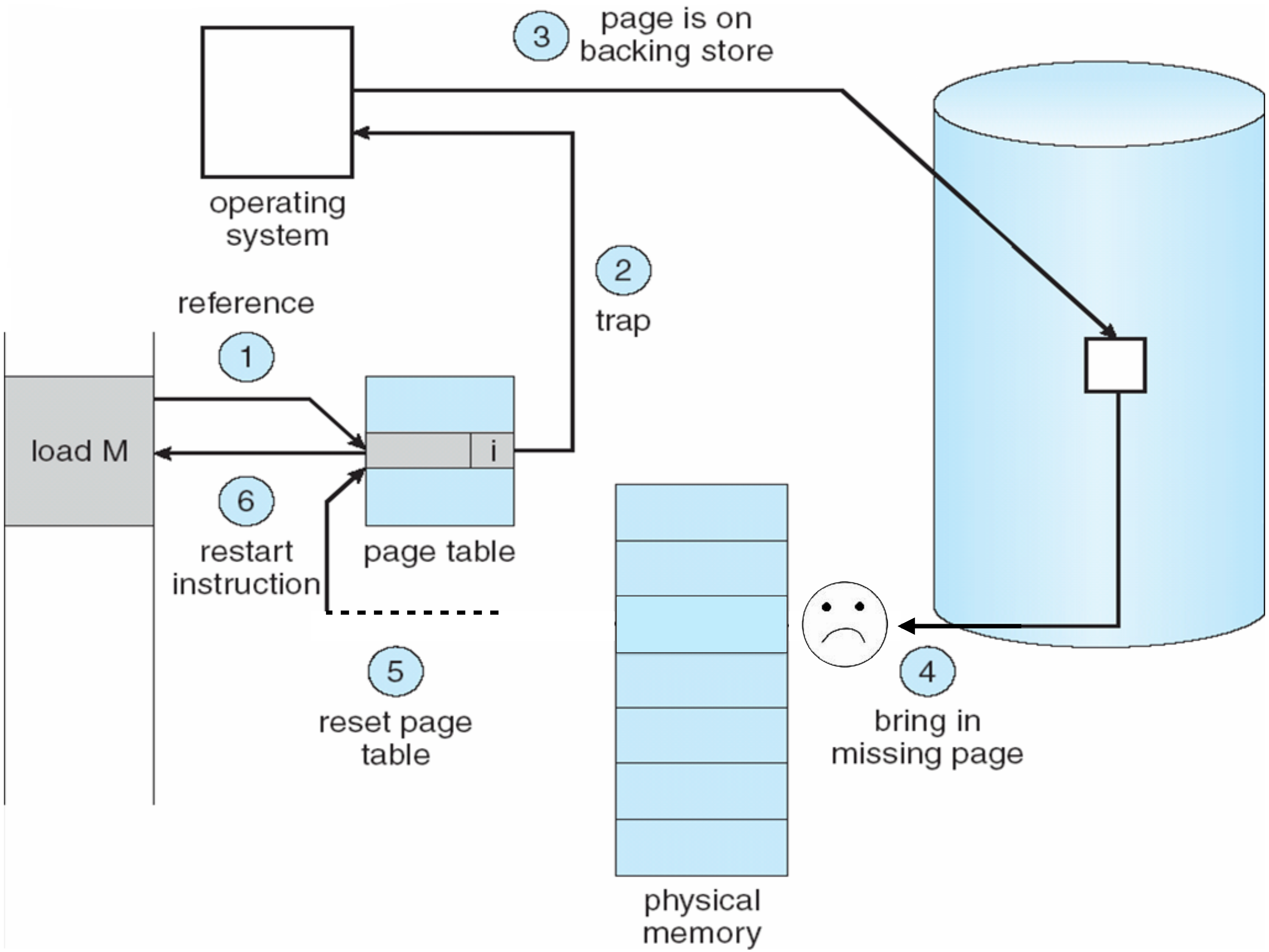
- The OS wants to maximize CPU utilization. As a result, it decides that it is a good idea to increase the degree of multiprogramming by adding new processes to the system.

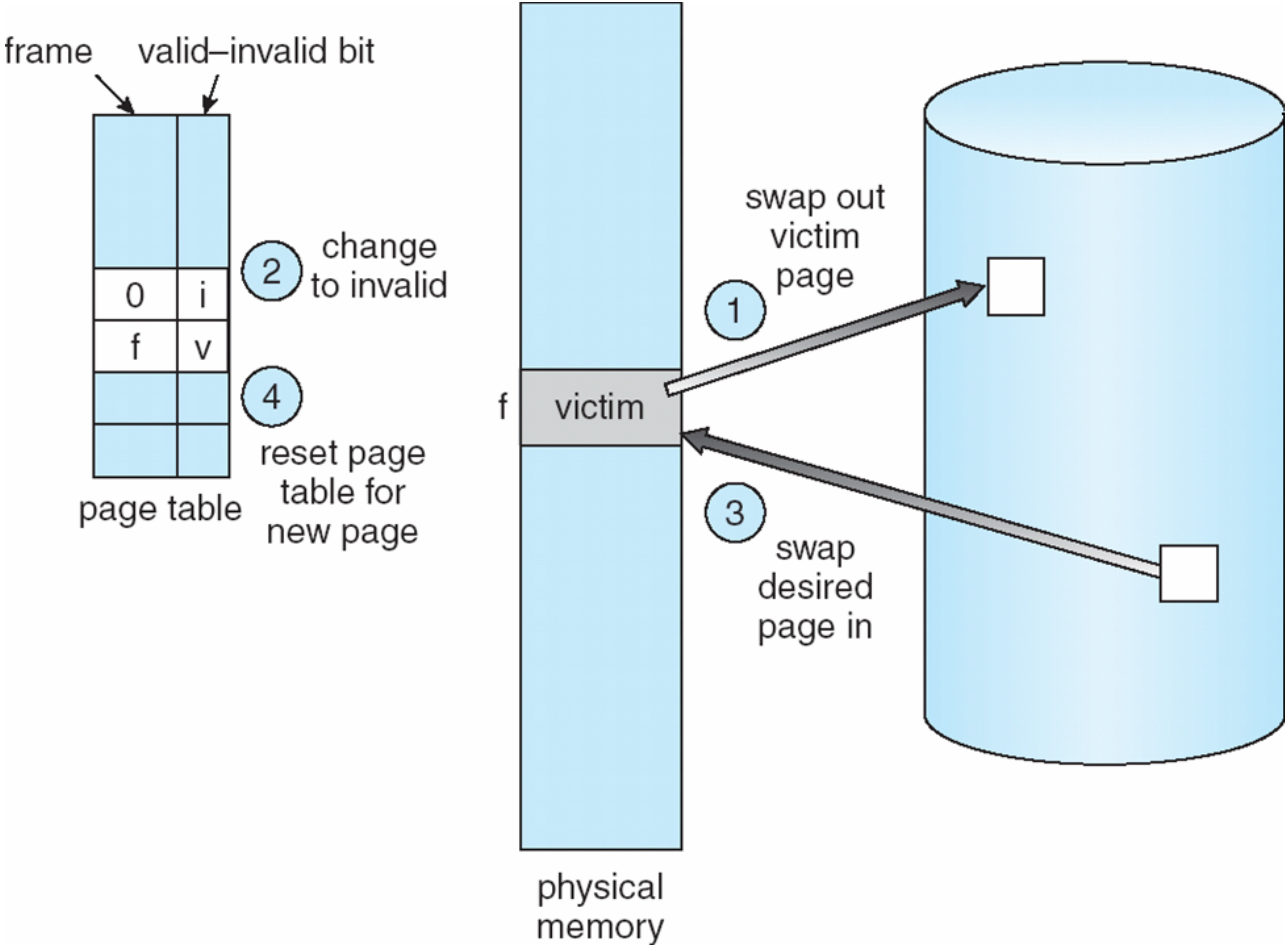


What happens is there is no free frame?



What happens is there is no free frame?

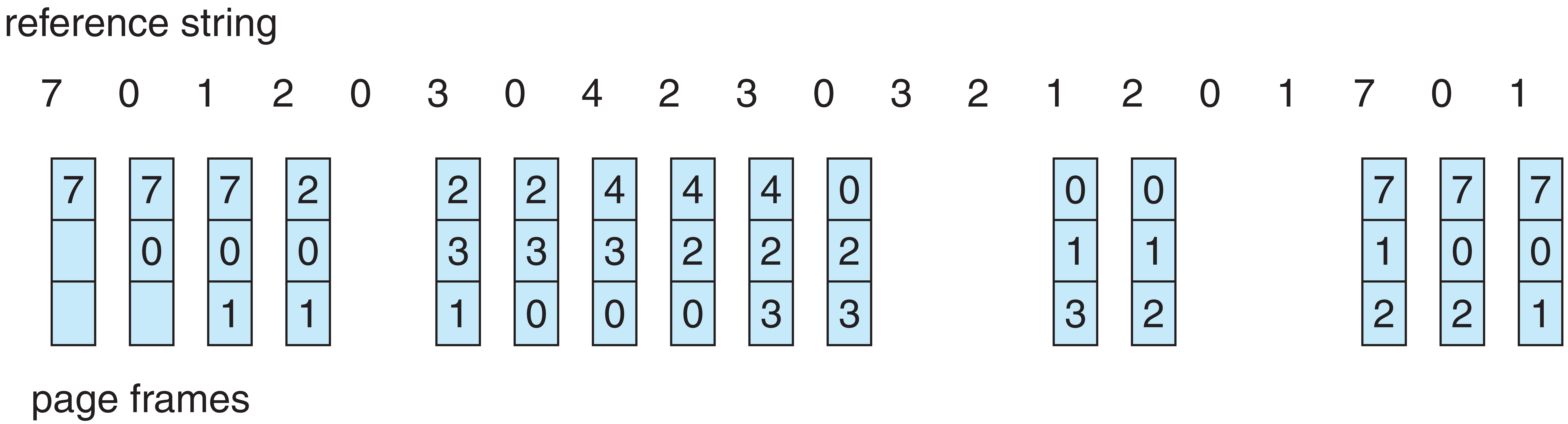



frame
valid–invalid bit
0
i
f
v
2
change to invalid
4
reset page table for new page
page table
f
victim
1
swap out victim page
3
swap desired page in
physical memory

Page-Replacement Algorithms

- FIFO algorithm
- Optimal page-replacement algorithm
- Least-recently used (LRU) algorithm
- Second-chance algorithm (clock)

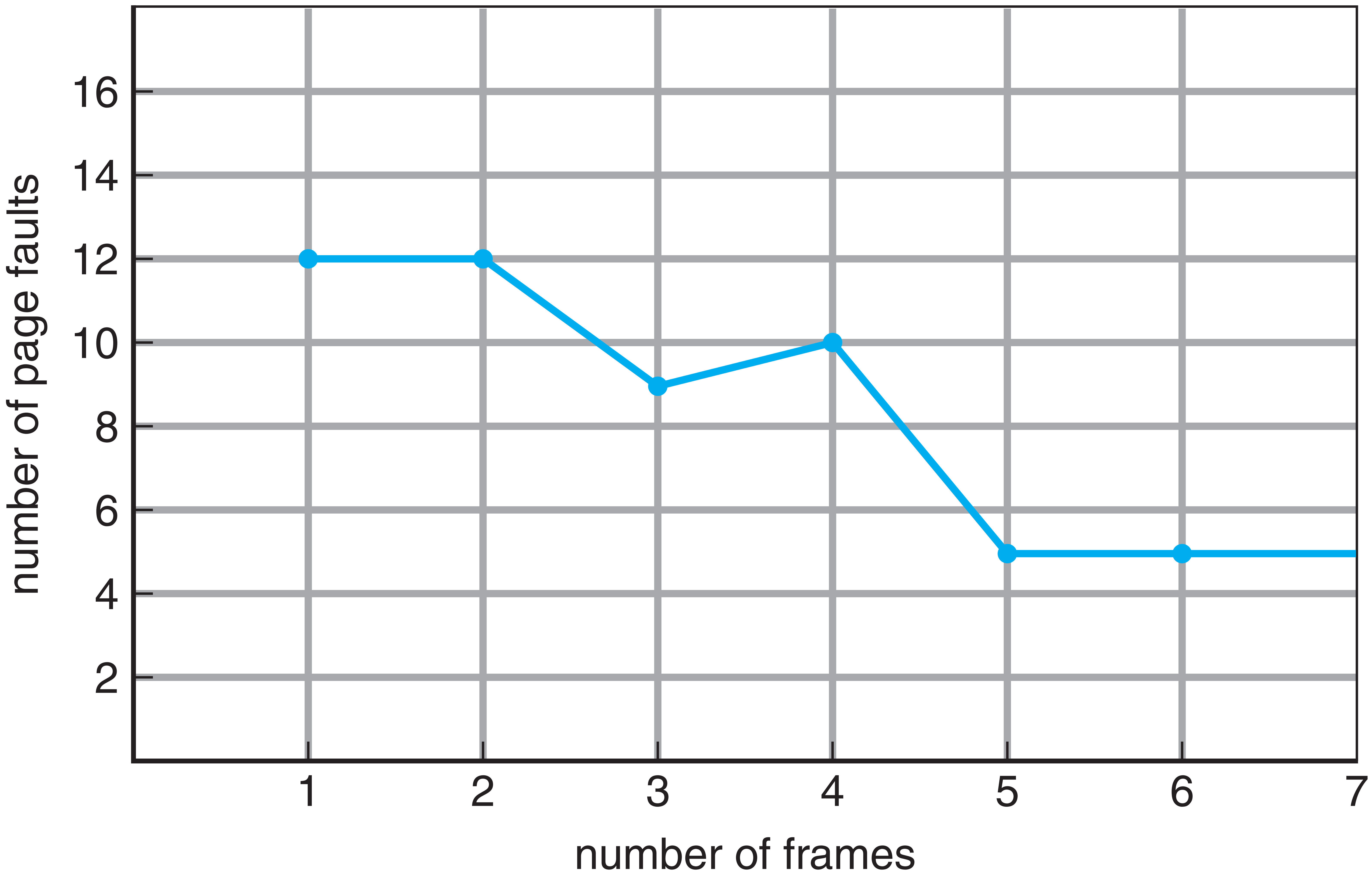
FIFO Algorithms



FIFO Algorithms

Access	Hit/Miss?	Evict	Resulting Cache State	
0	Miss		First-in→	0
1	Miss		First-in→	0, 1
2	Miss		First-in→	0, 1, 2
0	Hit		First-in→	0, 1, 2
1	Hit		First-in→	0, 1, 2
3	Miss	0	First-in→	1, 2, 3
0	Miss	1	First-in→	2, 3, 0
3	Hit		First-in→	2, 3, 0
1	Miss	2	First-in→	3, 0, 1
2	Miss	3	First-in→	0, 1, 2
1	Hit		First-in→	0, 1, 2

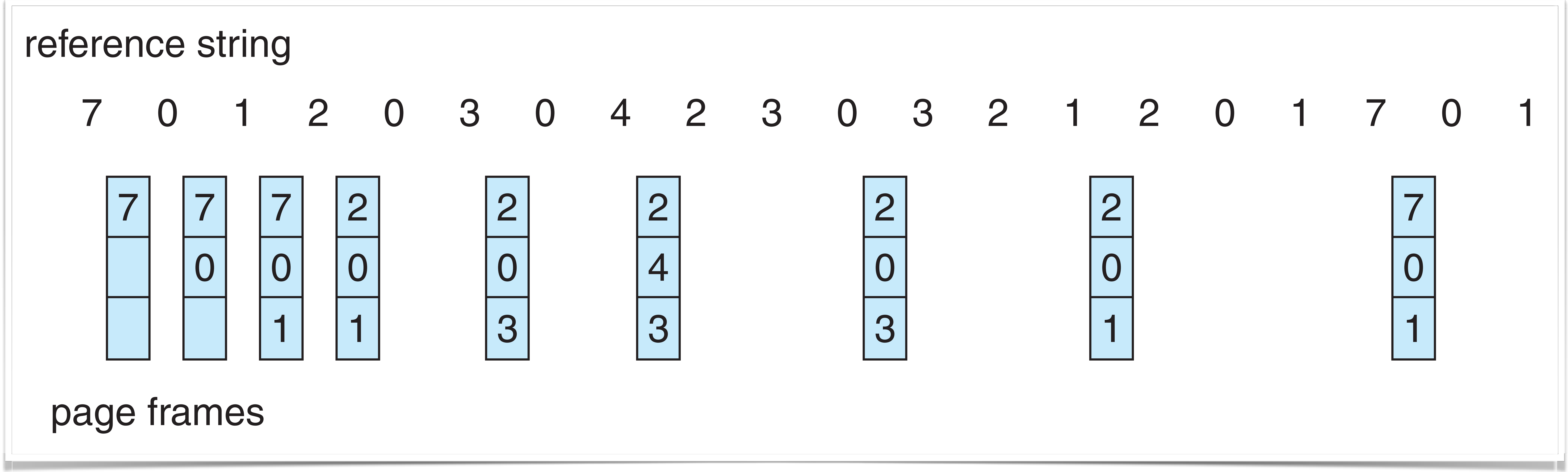
Belady’s anomaly



Paper: L. A. Belady, R. A. Nelson, G. S. Shedler, An anomaly in space-time char- acteristics of certain programs running in paging machine, Comm. ACM , 12, 1 (1969) 349–353.

Optimal Algorithm

Policy: Replace the page that will not be used for the longest period of time.



Optimal Algorithm

Access	Hit/Miss?	Evict	Resulting Cache State
0	Miss		0
1	Miss		0, 1
2	Miss		0, 1, 2
0	Hit		0, 1, 2
1	Hit		0, 1, 2
3	Miss	2	0, 1, 3
0	Hit		0, 1, 3
3	Hit		0, 1, 3
1	Hit		0, 1, 3
2	Miss	3	0, 1, 2
1	Hit		0, 1, 2

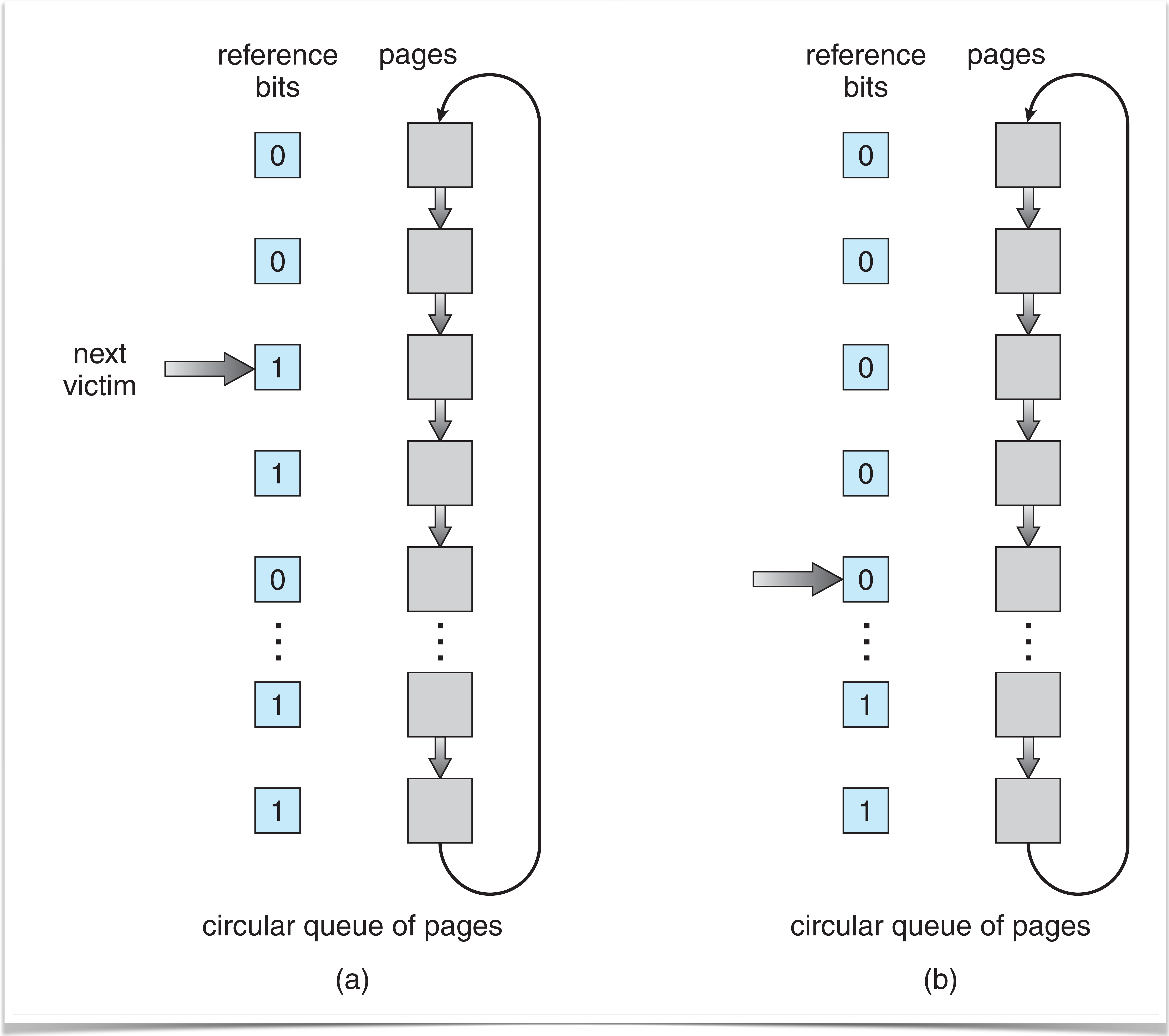
LRU Algorithm

Access	Hit/Miss?	Evict	Resulting Cache State	
0	Miss		LRU→	0
1	Miss		LRU→	0, 1
2	Miss		LRU→	0, 1, 2
0	Hit		LRU→	1, 2, 0
1	Hit		LRU→	2, 0, 1
3	Miss	2	LRU→	0, 1, 3
0	Hit		LRU→	1, 3, 0
3	Hit		LRU→	1, 0, 3
1	Hit		LRU→	0, 3, 1
2	Miss	0	LRU→	3, 1, 2
1	Hit		LRU→	3, 2, 1

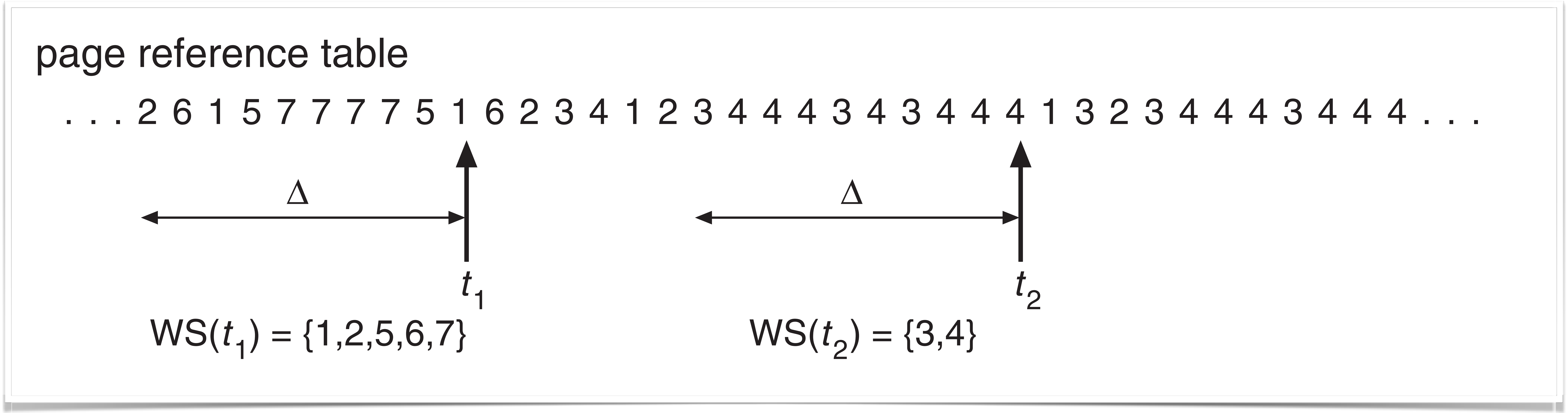
Second-chance Algorithm

- Whenever a page is referenced, the hardware sets the reference bit to 1.
- The O.S. sets the reference bit to 0 according to some policy.
- Evicting is free if page is not *dirty*.

    - Clock prioritize scan for pages that are both unused and clean.



Working-set Model



Working Sets and Page-fault frequency