Operating System Review 2

Chapter 6 to 9

Chapter 6: Process Synchronization

Producer/Consumer Problem

- Producer

```
item nextProduced;
while( true ) {
/* Produce an item and store it in nextProduced */
nextProduced = makeNewItem( . . . );

/* Wait for space to become available */
while( ( ( in + 1 ) % BUFFER_SIZE ) == out )
      ; /* Do nothing */

/* And then store the item and repeat the loop. */
buffer[ in ] = nextProduced;
in = ( in + 1 ) % BUFFER_SIZE;
}
```

Producer/Consumer Problem

- Consumer

```
item nextConsumed;
while( true ) {
/* Wait for an item to become available */
while( in == out )
      ; /* Do nothing */
/* Get the next available item */
nextConsumed = buffer[ out ];
out = ( out + 1 ) % BUFFER_SIZE;

/* Consume the item in nextConsumed
     ( Do something with it ) */

}
```

Producer/Consumer Problem (why not use counter?)

Producer Process:

```
while (true)
{
     /* produce an item in nextProduced */
     while (counter == BUFFER_SIZE)
        ; /* do nothing */
     buffer[in] = nextProduced;
     in = (in + 1) % BUFFER_SIZE;
     counter++;
}
```

Race Starts!

Consumer Process:

```
while (true)
{
     while (counter == 0)
        ; /* do nothing */
     nextConsumed = buffer[out];
     out = (out + 1) % BUFFER_SIZE;
     counter--;
     /* consume the item in nextConsumed */
}
```

Arbitrary Result

Producer:

$$\begin{aligned} register_1 &= \texttt{counter} \\ register_1 &= register_1 + 1 \\ \texttt{counter} &= register_1 \end{aligned}$$

Consumer:

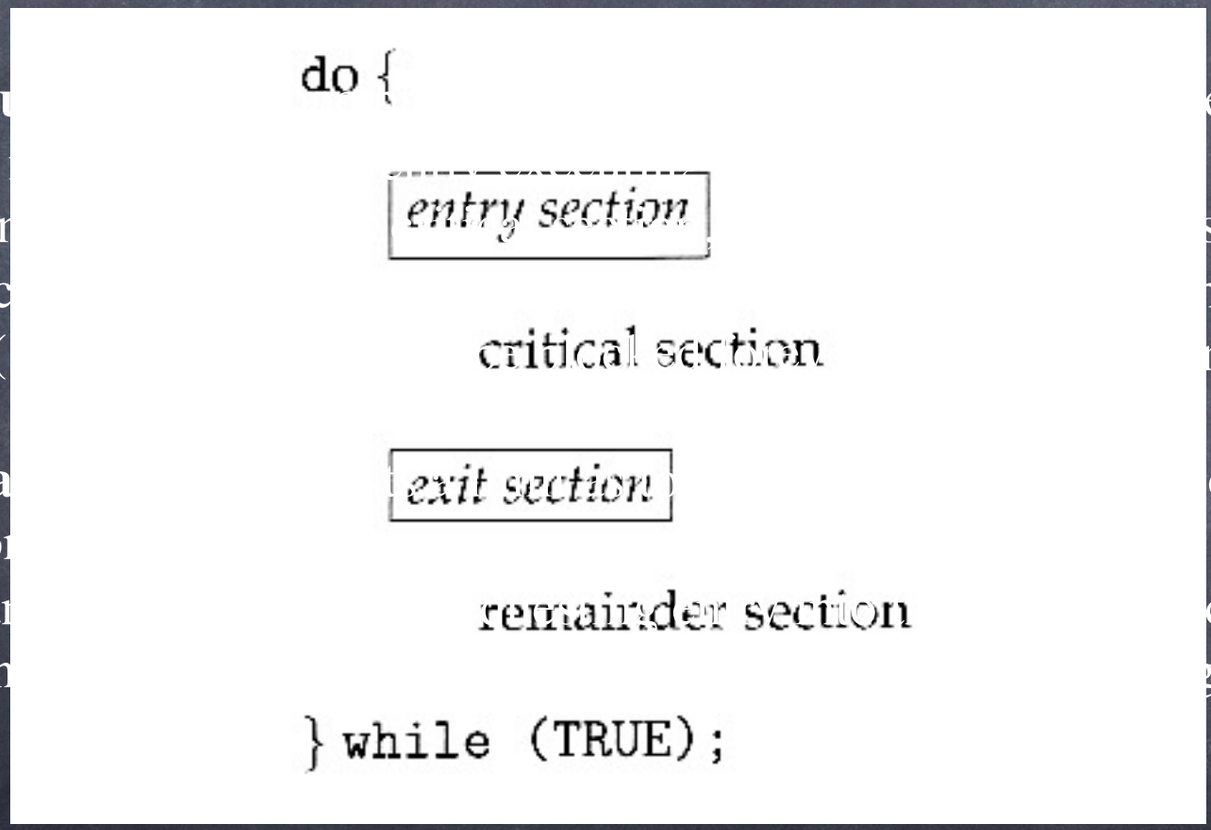
$$\begin{aligned} register_2 &= \texttt{counter} \\ register_2 &= register_2 - 1 \\ \texttt{counter} &= register_2 \end{aligned}$$

Interleaving:

T_0:	*producer*	execute	$register_1 = \texttt{counter}$	$\{register_1 = 5\}$
T_1:	*producer*	execute	$register_1 = register_1 + 1$	$\{register_1 = 6\}$
T_2:	*consumer*	execute	$register_2 = \texttt{counter}$	$\{register_2 = 5\}$
T_3:	*consumer*	execute	$register_2 = register_2 - 1$	$\{register_2 = 4\}$
T_4:	*producer*	execute	$\texttt{counter} = register_1$	$\{counter = 6\}$
T_5:	*consumer*	execute	$\texttt{counter} = register_2$	$\{counter = 4\}$

Critical Section

1 **Mutual Exclu** eir critical section.
2 **Progress** - If and one or more processes war s not in their remainder sec nnot be postponed indefinitely. (nto their critical sections.)
3 **Bounded Wa** es can get into their critical sectio and before that request is gra ction will get a turn eventually, an go first.)



Synchronization Hardware

```
boolean TestAndSet(boolean *target) {
   boolean rv = *target;
   *target = TRUE;
   return rv;
}
```

Figure 5.3 The definition of the `TestAndSet()` instruction.

```
do {
   while (TestAndSetLock(&lock))
      ; // do nothing

      // critical section

   lock = FALSE;

      // remainder section
}while (TRUE);
```

Figure 5.4 Mutual-exclusion implementation with `TestAndSet()`.

Synchronization Hardware

```
int compare_and_swap(int *value, int expected, int new_value) {
   int temp = *value;

   if (*value == expected)
      *value = new_value;

   return temp;
}
```

Figure 5.5 The definition of the `compare_and_swap()` instruction.

```
do {
   while (compare_and_swap(&lock, 0, 1) != 0)
      ; /* do nothing */

      /* critical section */

   lock = 0;

      /* remainder section */
} while (true);
```

Figure 5.6 Mutual-exclusion implementation with the `compare_and_swap()` instruction.

Semaphores

Wait:

```
wait(S) {
    while S <= 0
       ; // no-op
    S--;
}
```

Signal:

```
signal(S) {
    S++;
}
```

```
do {
   waiting(mutex);

      // critical section

   signal(mutex);

      // remainder section
}while (TRUE);
```

Semaphores Implementation

Semaphore Structure:

```
typedef struct {
        int value;
        struct process *list;
} semaphore;
```

Wait Operation:

```
wait(semaphore *S) {
            S->value--;
            if (S->value < 0) {
                    add this process to S->list;
                    block();
            }
}
```

Signal Operation:

```
signal(semaphore *S) {
        S->value++;
        if (S->value <= 0) {
                remove a process P from S->list;
                wakeup(P);
        }
}
```

Bounded Buffer Problem

```
do {
    . . .
   // produce an item in nextp
    . . .
   wait(empty);
   wait(mutex);
    . . .
   // add nextp to buffer
    . . .
   signal(mutex);
   signal(full);
  }while (TRUE);
```

Figure 5.9 The structure of the producer process.

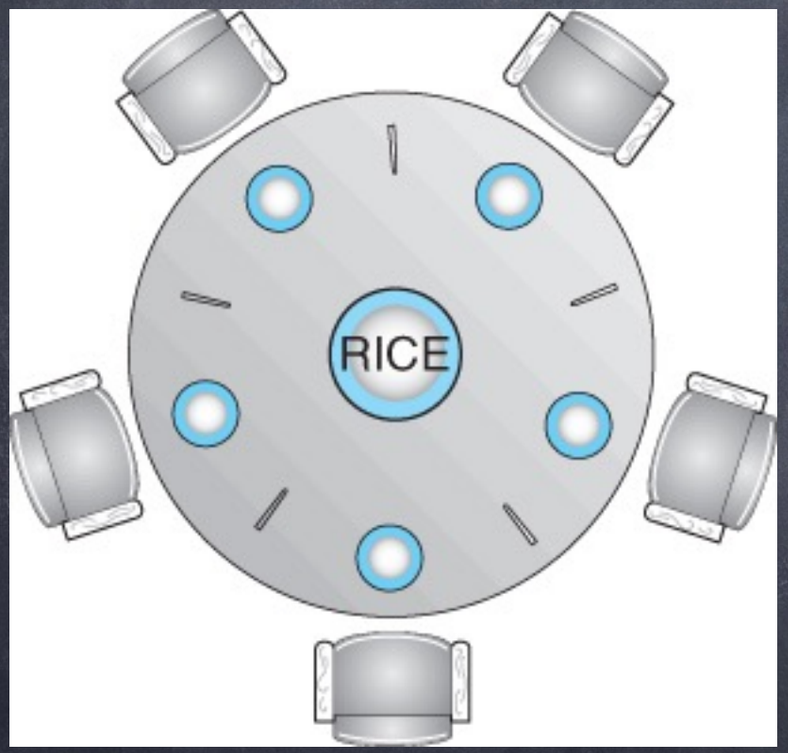
```
do {
   wait(full);
   wait(mutex);
    . . .
   // remove an item from buffer to nextc
    . . .
   signal(mutex);
   signal(empty);
    . . .
   // consume the item in nextc
    . . .
}while (TRUE);
```

Readers-Writer Problem

```
do {
   wait(rw_mutex);
      . . .
   /* writing is performed */
      . . .
   signal(rw_mutex);
} while (true);
```

```
do {
   wait(mutex);
   read_count++;
   if (read_count == 1)
      wait(rw_mutex);
   signal(mutex);
      . . .
   /* reading is performed */
      . . .
   wait(mutex);
   read_count--;
   if (read_count == 0)
      signal(rw_mutex);
   signal(mutex);
} while (true);
```

The Dining-Philosophers Problem




```
do {
   wait(chopstick[i]);
   wait(chopstick[(i+1) % 5]);
      . . .
   // eat
      . . .
   signal(chopstick[i]);
   signal(chopstick[(i+1) % 5]);
      . . .
   // think
      . . .
}while (TRUE);
```

Monitor

```
monitor monitor name
{
   // shared variable declarations

   procedure P1 ( . . . ) {
      . . .
   }

   procedure P2 ( . . . ) {
      . . .
   }

         .
         .
         .
   procedure Pn ( . . . ) {
      . . .
   }

   initialization code ( . . . ) {
      . . .
   }
}
```

The Dining-Philosophers Problem (using Monitor)

```
monitor DiningPhilosophers
{
   enum {THINKING, HUNGRY, EATING} state[5];
   condition self[5];

void pickup(int i) {
   state[i] = HUNGRY;
   test(i);
   if (state[i] != EATING)
      self[i].wait();
}

void putdown(int i) {
   state[i] = THINKING;
   test((i + 4) % 5);
   test((i + 1) % 5);
}
```

```
void test(int i) {
   if ((state[(i + 4) % 5] != EATING) &&
    (state[i] == HUNGRY) &&
    (state[(i + 1) % 5] != EATING)) {
      state[i] = EATING;
      self[i].signal();
   }
}

   initialization_code() {
      for (int i = 0; i < 5; i++)
         state[i] = THINKING;
    }
}
```

Transactions

T_0	T_1
read(A)	
write(A)	
read(B)	
write(B)	
	read(A)
	write(A)
	read(B)
	write(B)

T_0	T_1
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)

Timestamp Scheduling

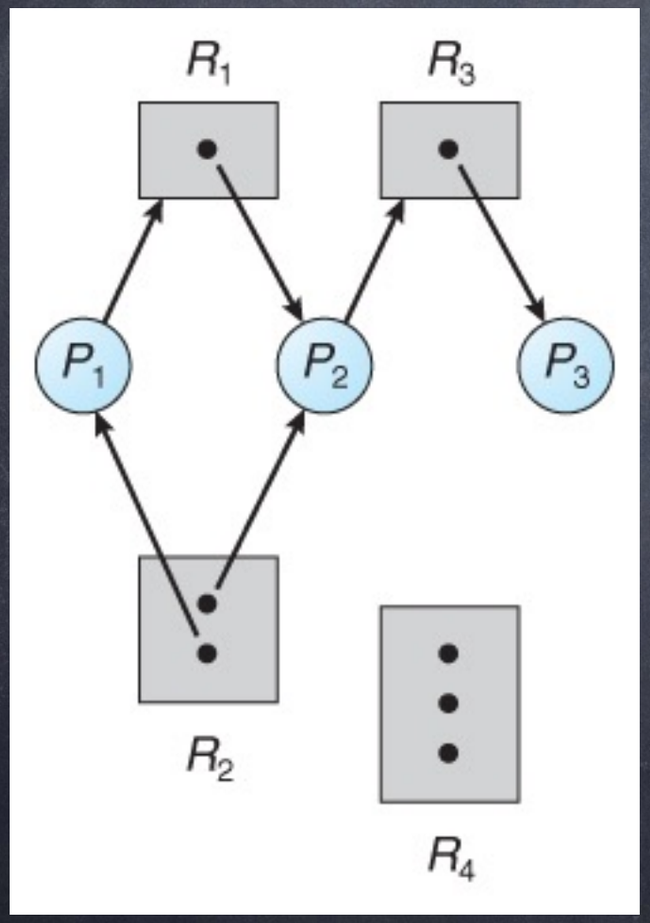
- Suppose transaction Ti issues a read on data item Q:
  - If TS(Ti) < the W-timestamp for Q, then it is attempting to read data that has been changed. Transaction Ti is rolled back, and restarted with a new timestamp.
  - If TS(Ti) > the W-timestamp for Q, then the R-timestamp for Q is updated to the later of its current value and TS(Ti).
- Suppose Ti issues a write on Q:
  - If TS(Ti) < the R-timestamp for Q, then it is attempting to change data that has already been read in its unaltered state. Ti is rolled back and restarted with a new timestamp.
  - If TS(Ti) < the W-timestamp for Q it is also rolled back and restarted, for similar reasons.
  - Otherwise, the operation proceeds, and the W-timestamp for Q is updated to TS(Ti).

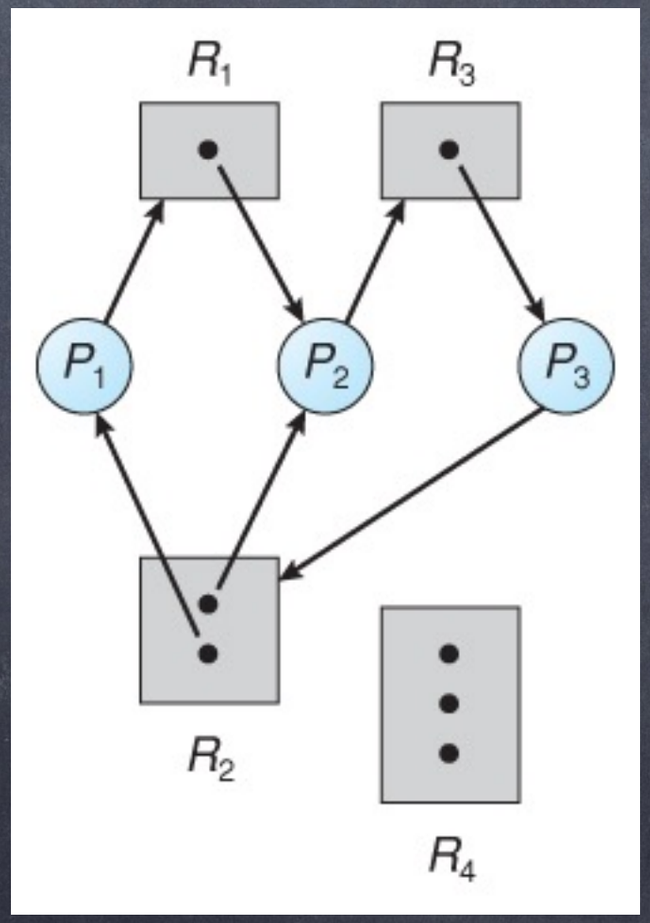
Chapter 7 Deadlock

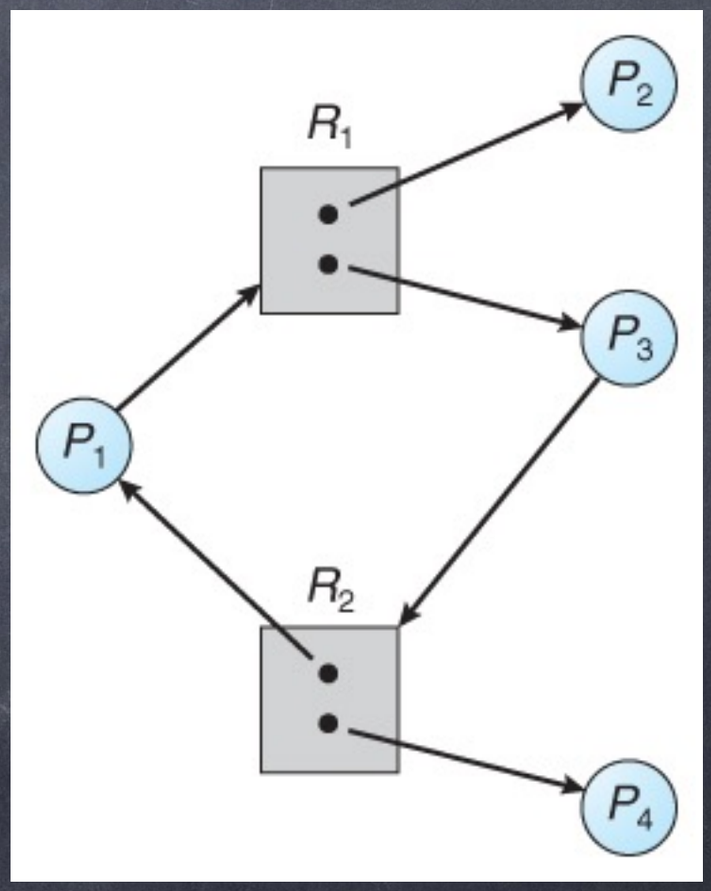
Four Conditions

1. **Mutual Exclusion** - At least one resource must be held in a non-sharable mode; If any other process requests this resource, then that process must wait for the resource to be released.
2. **Hold and Wait** - A process must be simultaneously holding at least one resource and waiting for at least one resource that is currently being held by some other process.
3. **No preemption** - Once a process is holding a resource (i.e. once its request has been granted), then that resource cannot be taken away from that process until the process voluntarily releases it.
4. **Circular Wait** - A set of processes { P0, P1, P2, . . ., PN } must exist such that every P[i] is waiting for P[(i + 1) % (N + 1)]. (Note that this condition implies the hold-and-wait condition, but it is easier to deal with the conditions if the four are considered separately.)

Resource-Allocation Graph







Deadlock Prevention

Mutual Exclusion

- Shared resources such as read-only files do not lead to deadlocks.
- Unfortunately some resources, such as printers and tape drives, require exclusive access by a single process.

Hold and Wait

- To prevent this condition processes must be prevented from holding one or more resources while simultaneously waiting for one or more others. There are several possibilities for this:
  - Require that all processes request all resources at one time. This can be wasteful of system resources if a process needs one resource early in its execution and doesn't need some other resource until much later.
  - Require that processes holding resources must release them before requesting new resources, and then re-acquire the released resources along with the new ones in a single new request. This can be a problem if a process has partially completed an operation using a resource and then fails to get it re-allocated after releasing it.

No Preemption

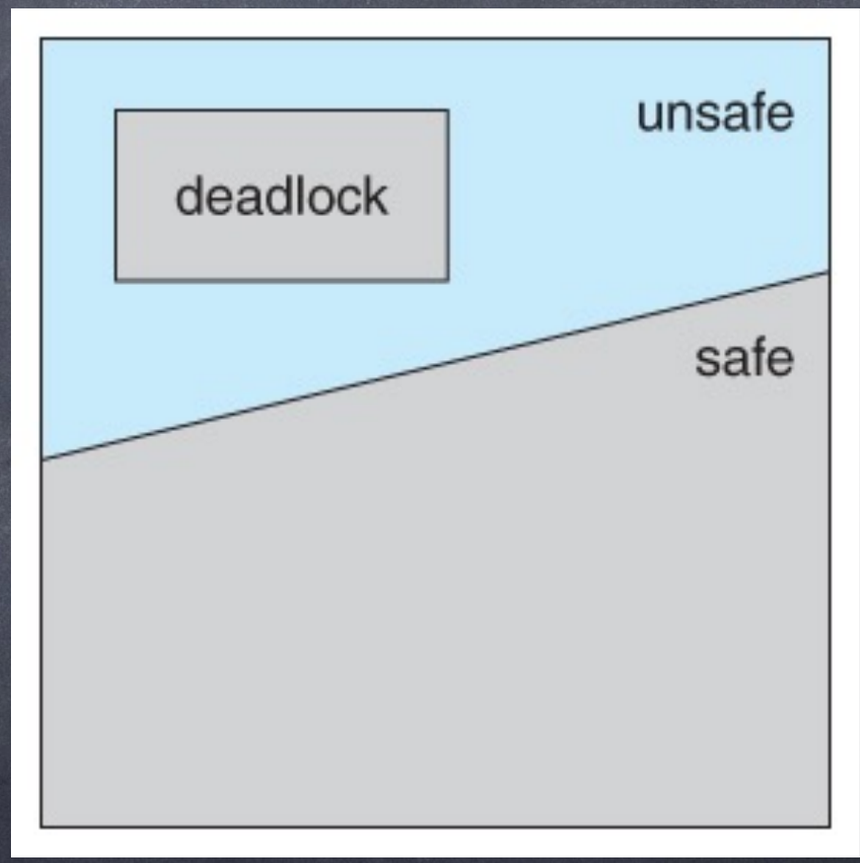
- Preemption of process resource allocations can prevent this condition of deadlocks, when it is possible.

Circular Wait

- One way to avoid circular wait is to number all resources, and to require that processes request resources only in strict order

Deadlock Avoidance

- A state is ***safe*** if the system can allocate all resources requested by all processes (up to their stated maximums) without entering a deadlock state.
- More formally, a state is safe if there exists a ***safe sequence*** of processes { P0, P1, P2, ..., PN } such that all of the resource requests for Pi can be granted using the resources currently allocated to Pi and all processes Pj where j < i. (I.e. if all the processes prior to Pi finish and free up their resources, then Pi will be able to finish also, using the resources that they have freed up.)



Is it safe?

Total = 12

	Maximum Needs	Current Allocation
P0	10	5
P1	4	2
P2	9	2

Safety Algorithm

1. Let ***Work*** and ***Finish*** be vectors of length *m* and *n*, respectively.
   Initialize:

   Work = *Available*

   Finish [*i*] = *false* for *i* = 0, 1, …, *n*- 1.

2. Find an *i* such that both:

   (a) *Finish* [*i*] = *false*

   (b) $Need_i \leq Work$

   If no such *i* exists, go to step 4.

3. $Work = Work + Allocation_i$
   Finish[*i*] = *true*
   go to step 2.

4. If *Finish* [*i*] == true for all *i*, then the system is in a safe state.

Resource-request Algorithm (bankers algorithm)

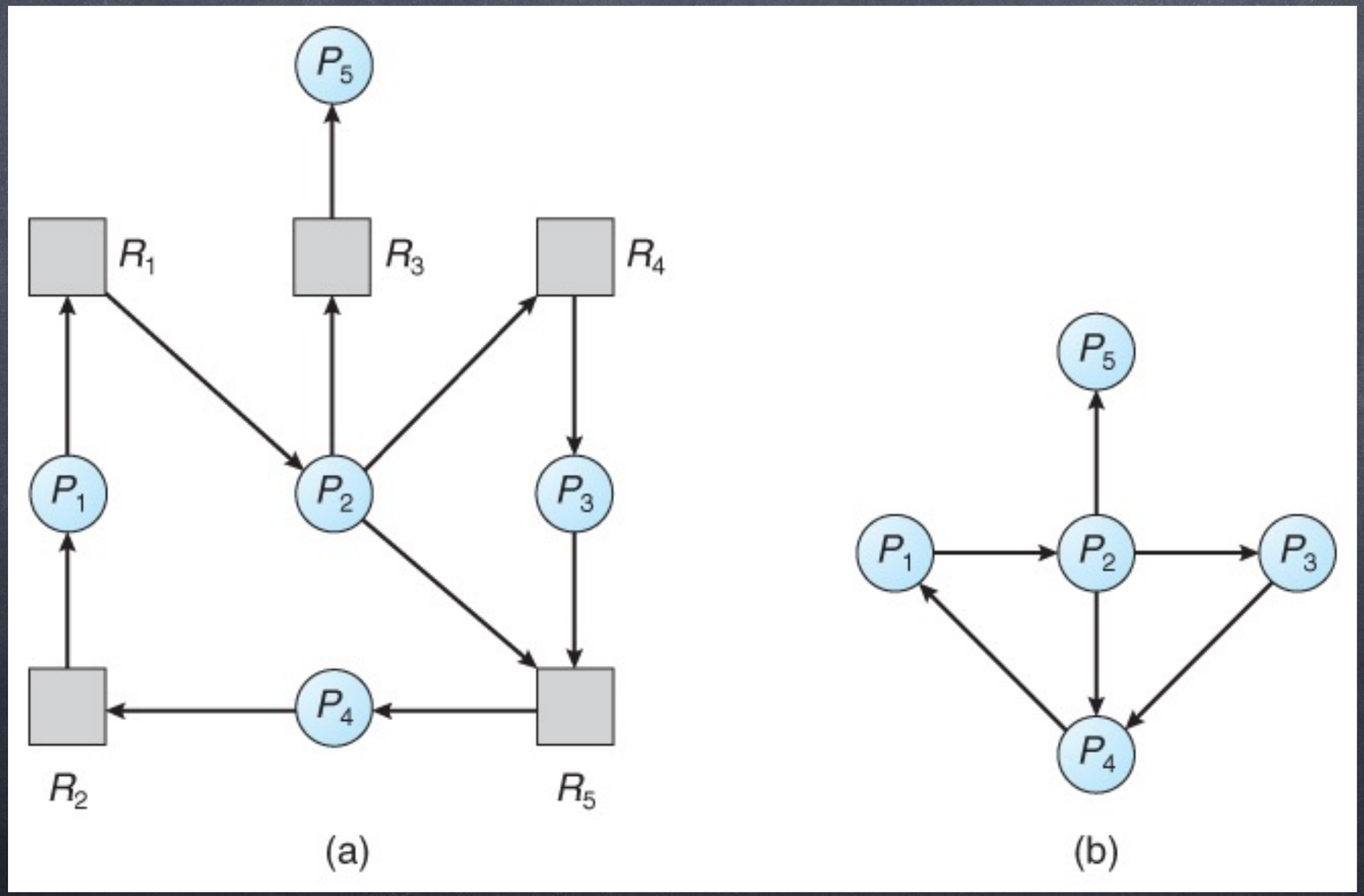
Request = request vector for process P_i. If $Request_i[j] = k$ then process P_i wants k instances of resource type R_j.

1. If $Request_i \leq Need_i$ go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim.
2. If $Request_i \leq Available$, go to step 3. Otherwise P_i must wait, since resources are not available.
3. Pretend to allocate requested resources to P_i by modifying the state as follows:

$$Available = Available - Request;$$
$$Allocation_i = Allocation_i + Request_i;$$
$$Need_i = Need_i - Request_i;$$

- *If safe ⇒ the resources are allocated to Pi.*
- *If unsafe ⇒ Pi must wait, and the old resource-allocation state is restored*

Deadlock Detection



Recovery from Deadlock

Process Termination

- Two basic approaches, both of which recover resources allocated to terminated processes:
  - Terminate all processes involved in the deadlock. This definitely solves the deadlock, but at the expense of terminating more processes than would be absolutely necessary.
  - Terminate processes one by one until the deadlock is broken. This is more conservative, but requires doing deadlock detection after each step.

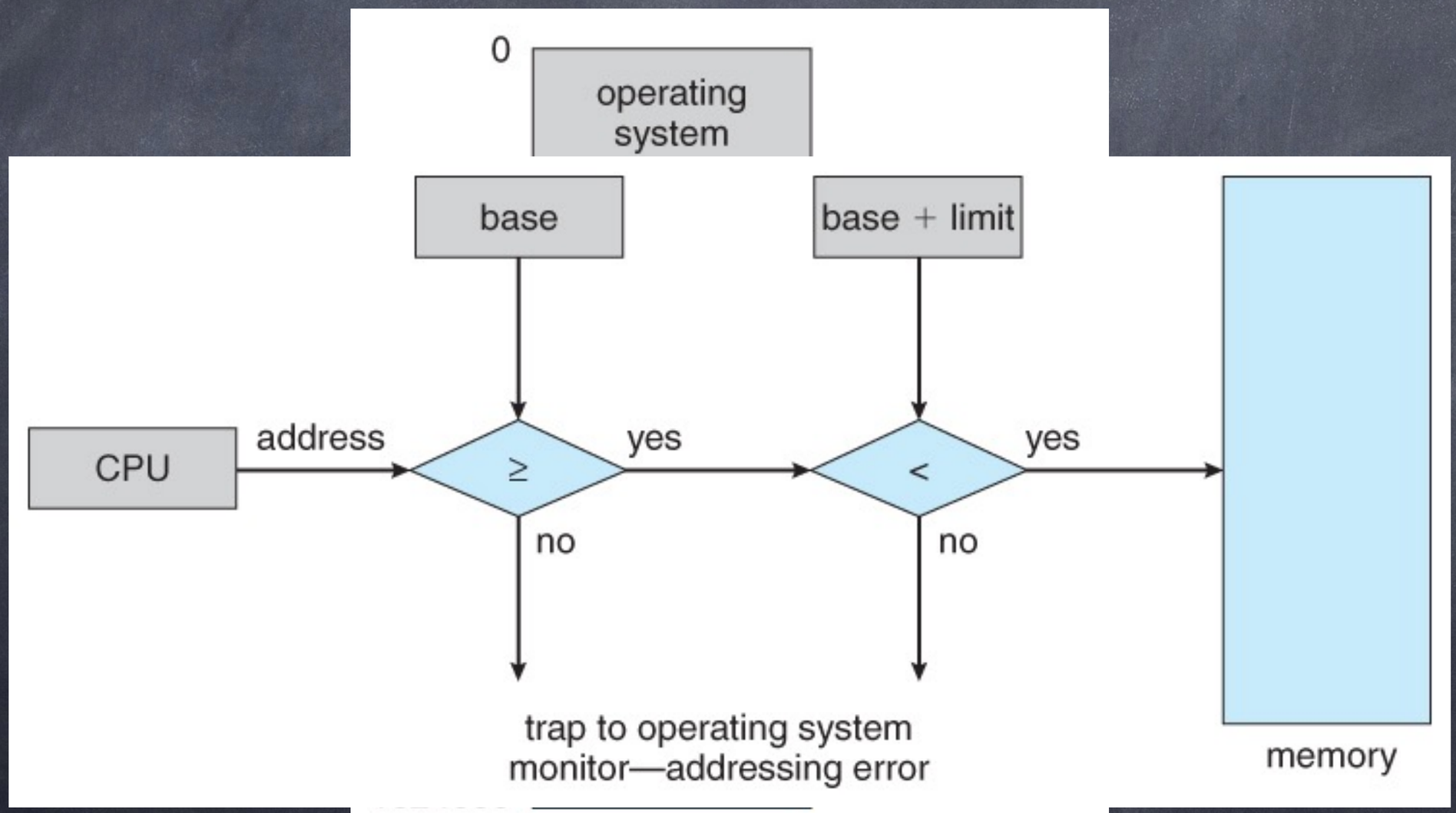
Resource Preemption

- When preempting resources to relieve deadlock, there are three important issues to be addressed:
  1. **Selecting a victim** - Deciding which resources to preempt from which processes involves many of the same decision criteria outlined above.
  2. **Rollback** - Ideally one would like to roll back a preempted process to a safe state prior to the point at which that resource was originally allocated to the process. Unfortunately it can be difficult or impossible to determine what such a safe state is, and so the only safe rollback is to roll back all the way back to the beginning. (I.e. abort the process and make it start over.)

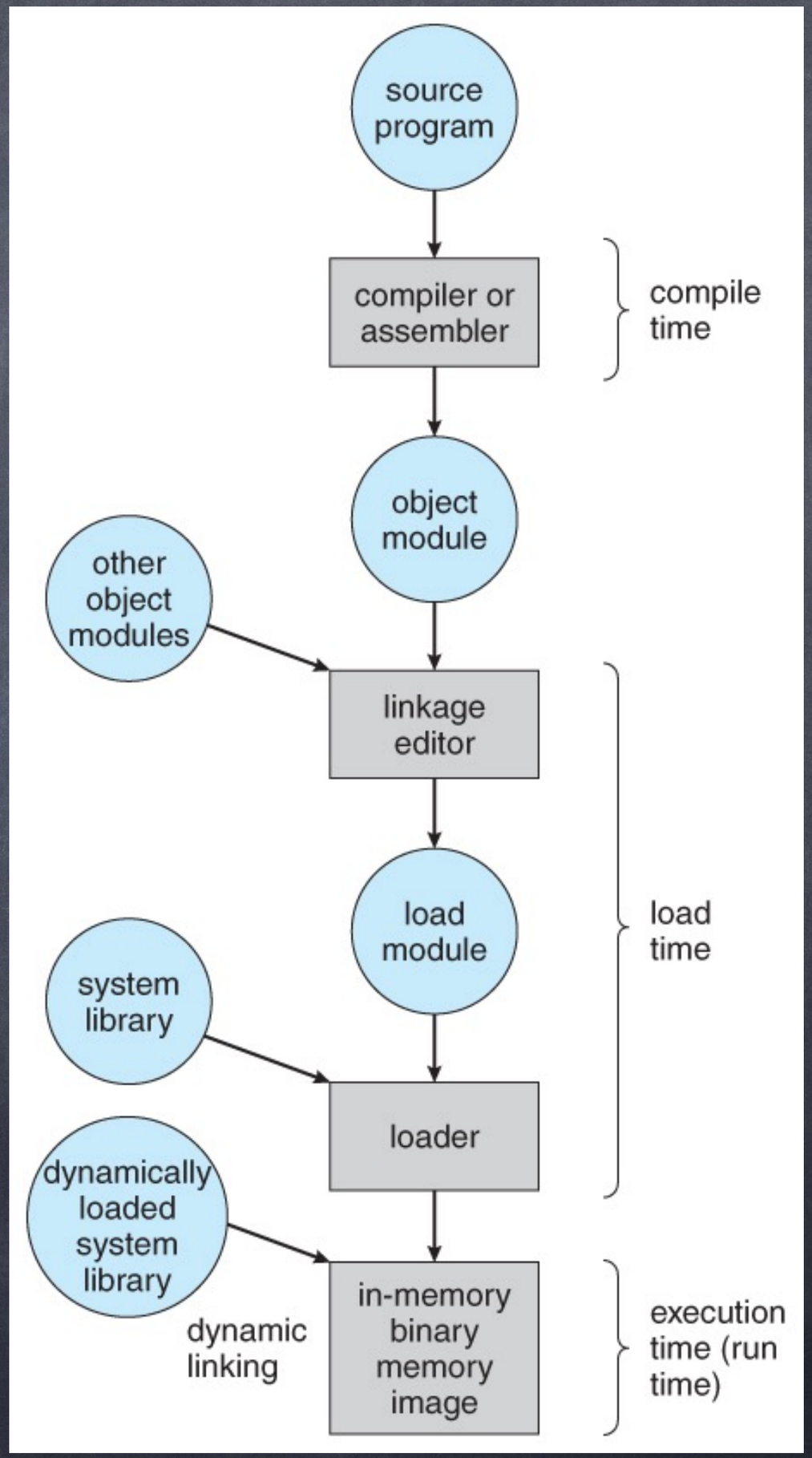
Chapter 8-9

Memory

Basic Idea


0
operating system
base
base + limit
CPU
address
≥
yes
<
yes
no
no
trap to operating system
monitor—addressing error
memory

Address Binding



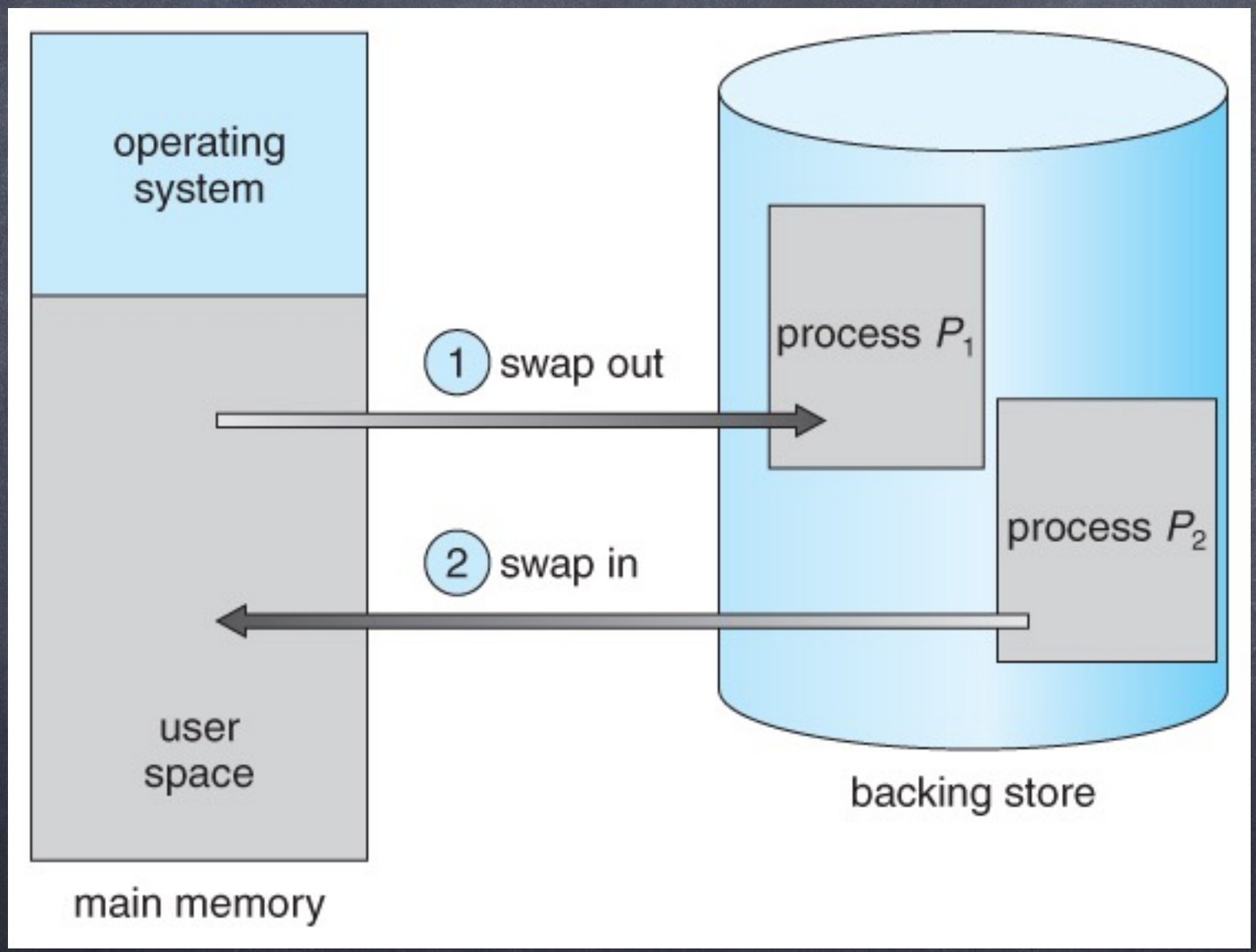
Dynamic Loading and Linking

Rather than loading an entire program into memory at once, dynamic loading loads up each routine as it is called. The advantage is that unused routines need never be loaded, reducing total memory usage and generating faster program startup times. The downside is the added complexity and overhead of checking to see if a routine is loaded every time it is called and then then loading it up if it is not already loaded.

dynamic linking, however, only a stub is linked into the executable module, containing references to the actual library module linked in at run time.

- This method saves disk space, because the library routines do not need to be fully included in the executable modules, only the stubs.

Swapping

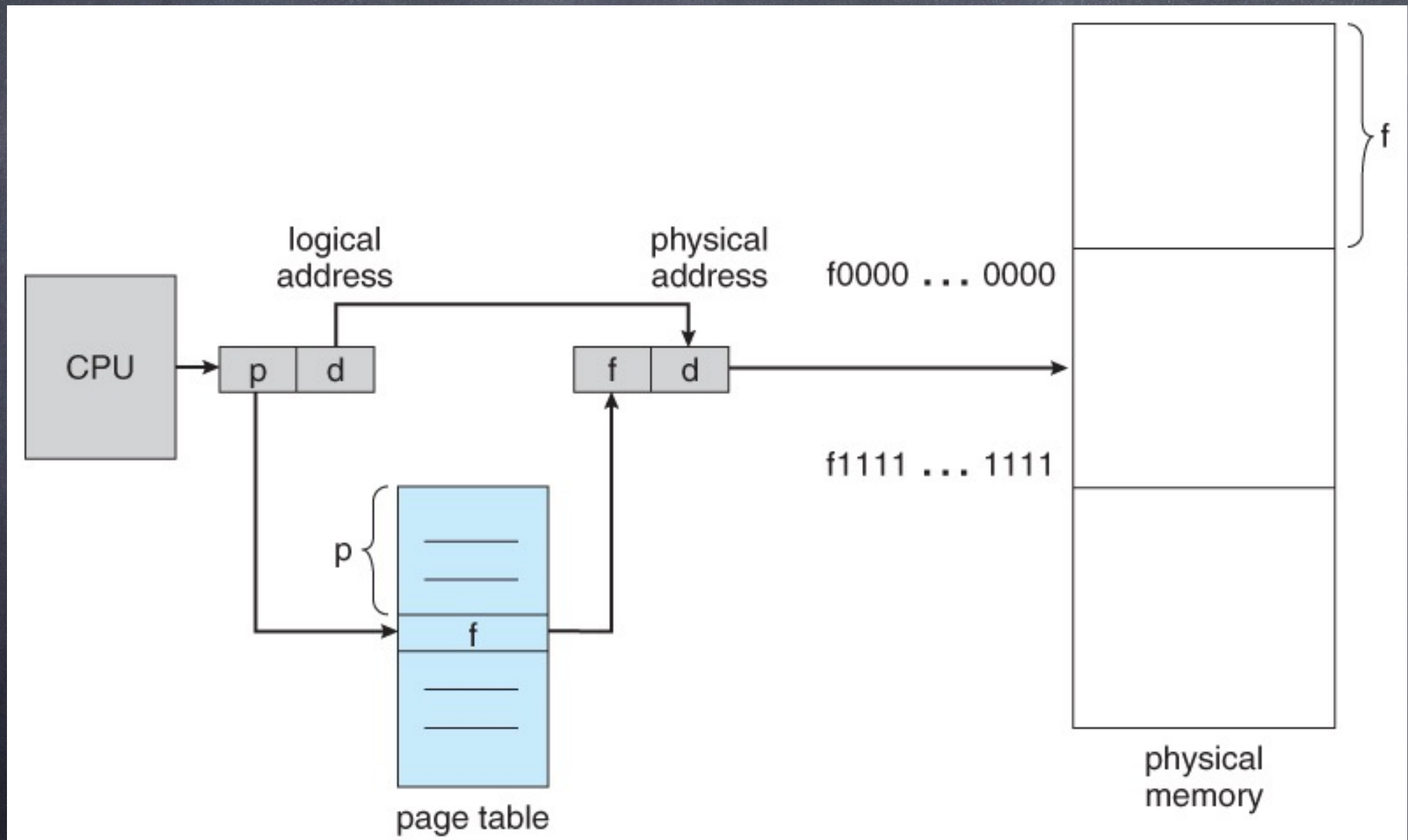

operating system
user space
main memory
1 swap out
2 swap in
process P_1
process P_2
backing store

Memory Allocation

An alternate approach is to keep a list of unused (free) memory blocks (holes), and to find a hole of a suitable size whenever a process needs to be loaded into memory. There are many different strategies for finding the "best" allocation of memory to processes, including the three most commonly discussed:

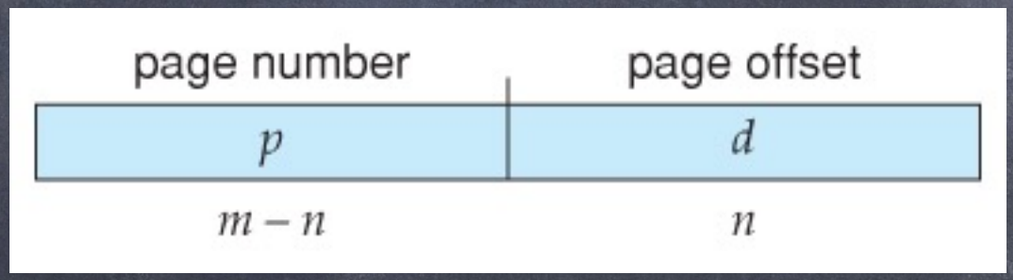
1. **First fit** - Search the list of holes until one is found that is big enough to satisfy the request, and assign a portion of that hole to that process. Whatever fraction of the hole not needed by the request is left on the free list as a smaller hole. Subsequent requests may start looking either from the beginning of the list or from the point at which this search ended.
2. **Best fit** - Allocate the *smallest* hole that is big enough to satisfy the request. This saves large holes for other process requests that may need them later, but the resulting unused portions of holes may be too small to be of any use, and will therefore be wasted. Keeping the free list sorted can speed up the process of finding the right hole.
3. **Worst fit** - Allocate the largest hole available, thereby increasing the likelihood that the remaining portion will be usable for satisfying future requests.

Paging


logical address
physical address
CPU
p
d
f
d
f0000 ... 0000
f1111 ... 1111
p
f
page table
physical memory

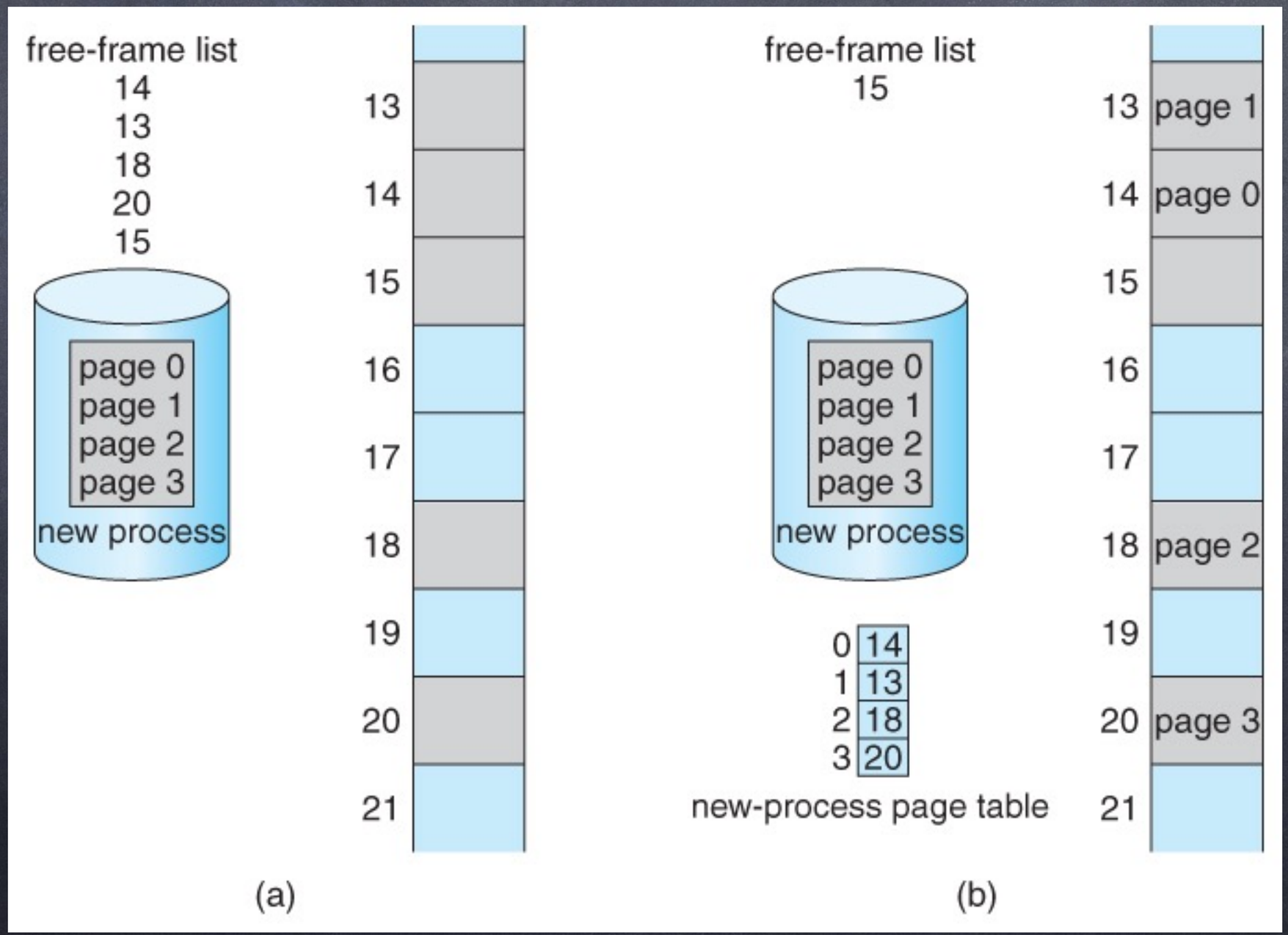
Page Size

Page numbers, frame numbers, and frame sizes are determined by the architecture, but are typically powers of two, allowing addresses to be split at a certain number of bits. For example, if the logical address size is 2^m and the page size is 2^n, then the high-order m-n bits of a logical address designate the page number and the remaining n bits represent the offset.

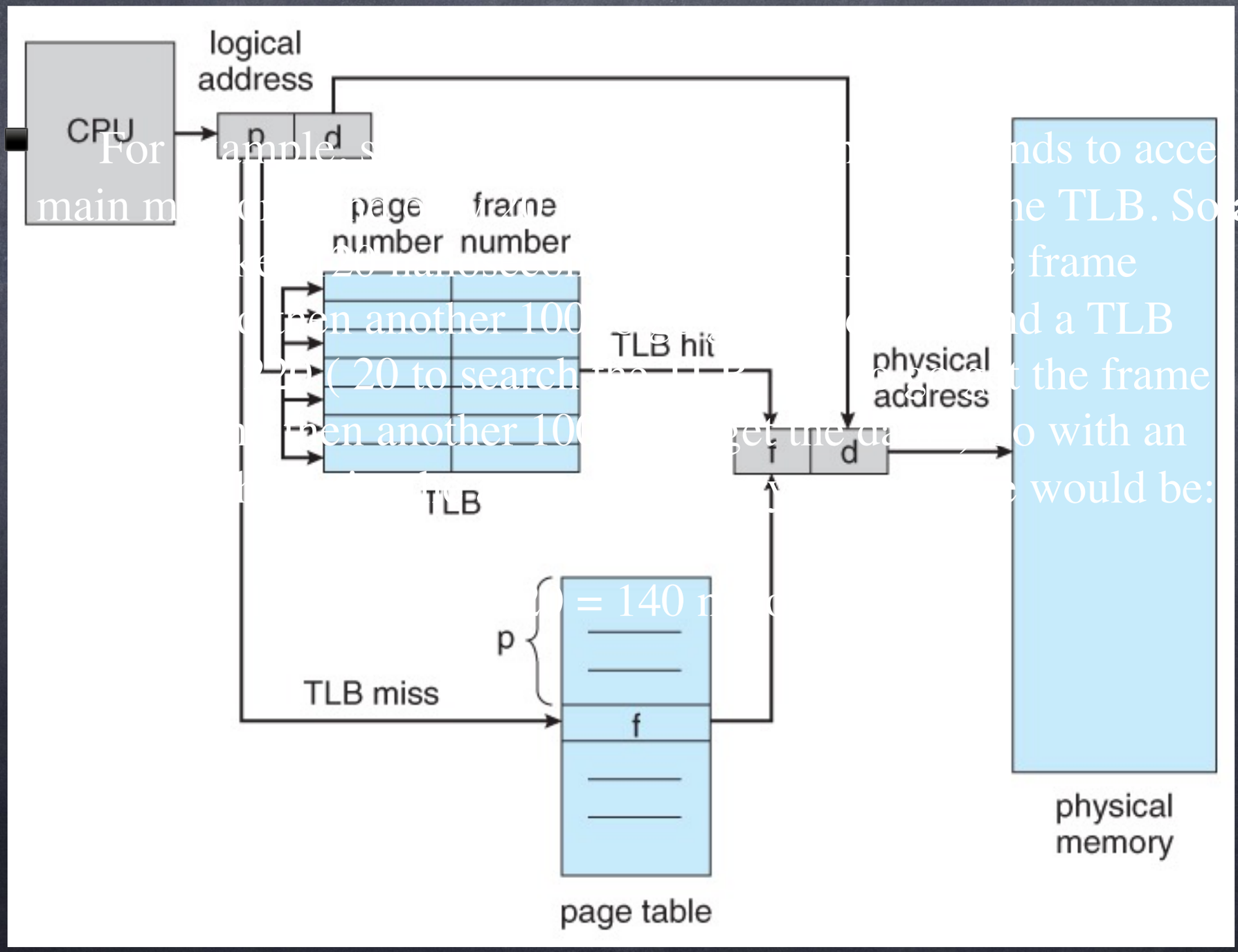


Page table entries (frame numbers) are typically 32 bit numbers, allowing access to 2^32 physical page frames. If those frames are 4 KB in size each, that translates to 16 TB of addressable physical memory. (32 + 12 = 44 bits of physical address space.)

Page Allocation

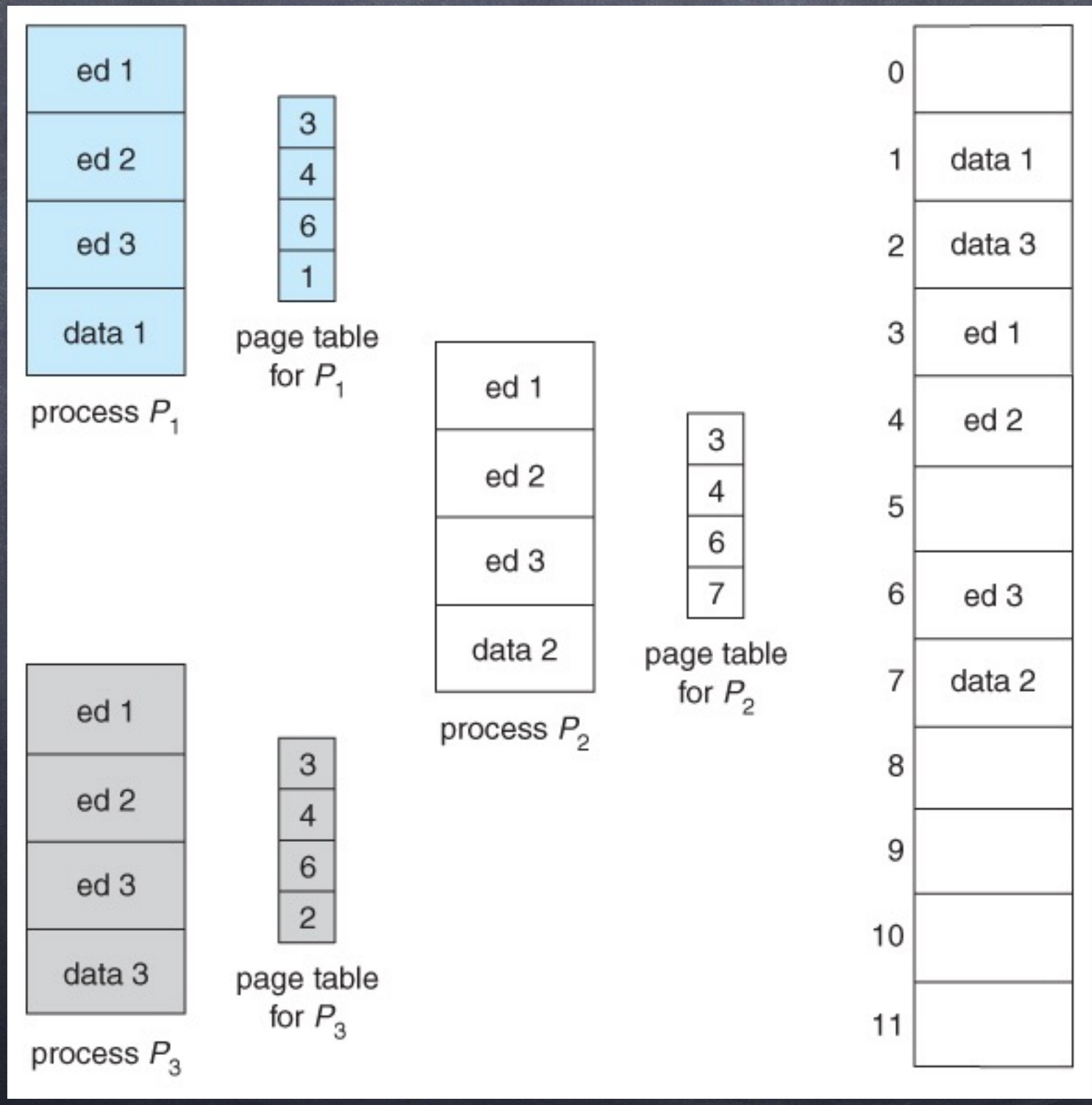


TLB = translation look-aside buffer



For [illegible] nds to acce s main m[illegible] he TLB. So a [illegible] frame [illegible] en another 100 [illegible] d a TLB [illegible] 20 to search [illegible] the frame [illegible] en another 100 [illegible] get the da[illegible] o with an [illegible] would be: [illegible] = 140 n[illegible]

Shared Page

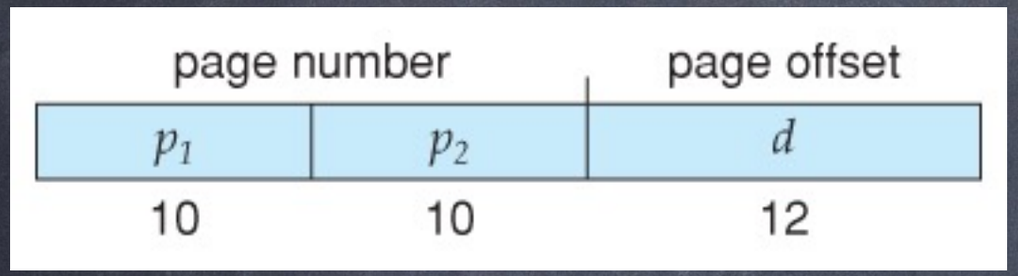

ed 1
ed 2
ed 3
data 1
process P_1
3
4
6
1
page table for P_1
ed 1
ed 2
ed 3
data 2
process P_2
3
4
6
7
page table for P_2
ed 1
ed 2
ed 3
data 3
process P_3
3
4
6
2
page table for P_3
0
1 data 1
2 data 3
3 ed 1
4 ed 2
5
6 ed 3
7 data 2
8
9
10
11

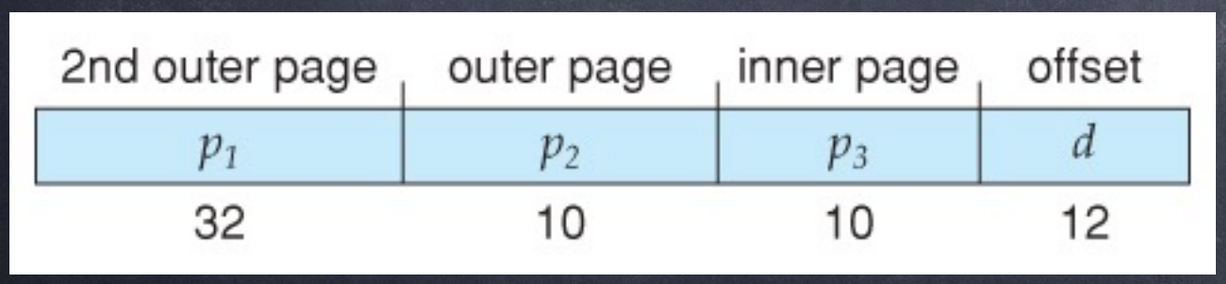
Structure of Paging

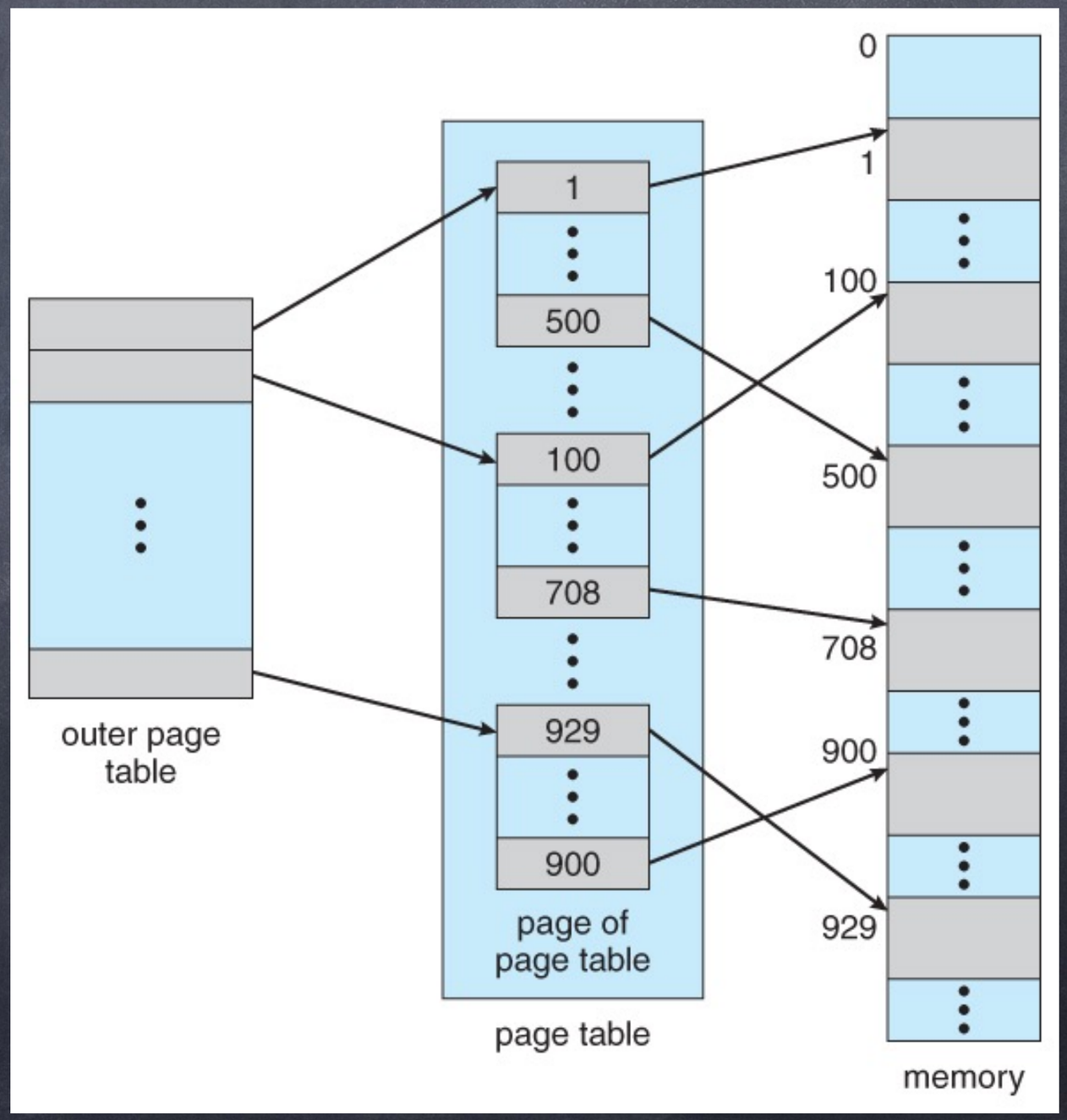
With a 2^32 address space and 4K (2^12) page sizes, this leave 2^20 entries in the page table. At 4 bytes per entry, this amounts to a 4 MB page table, which is too large to reasonably keep in contiguous memory.

- One option is to use a two-tier paging system, i.e. to page the page table.
- For example, the 20 bits described above could be broken down into two 10-bit page numbers. The first identifies an entry in the outer page table, which identifies where in memory to find one page of an inner page table. The second 10 bits finds a specific entry in that inner page table, which in turn identifies a particular frame in physical memory. (The remaining 12 bits of the 32 bit logical address are the offset within the 4K frame.)

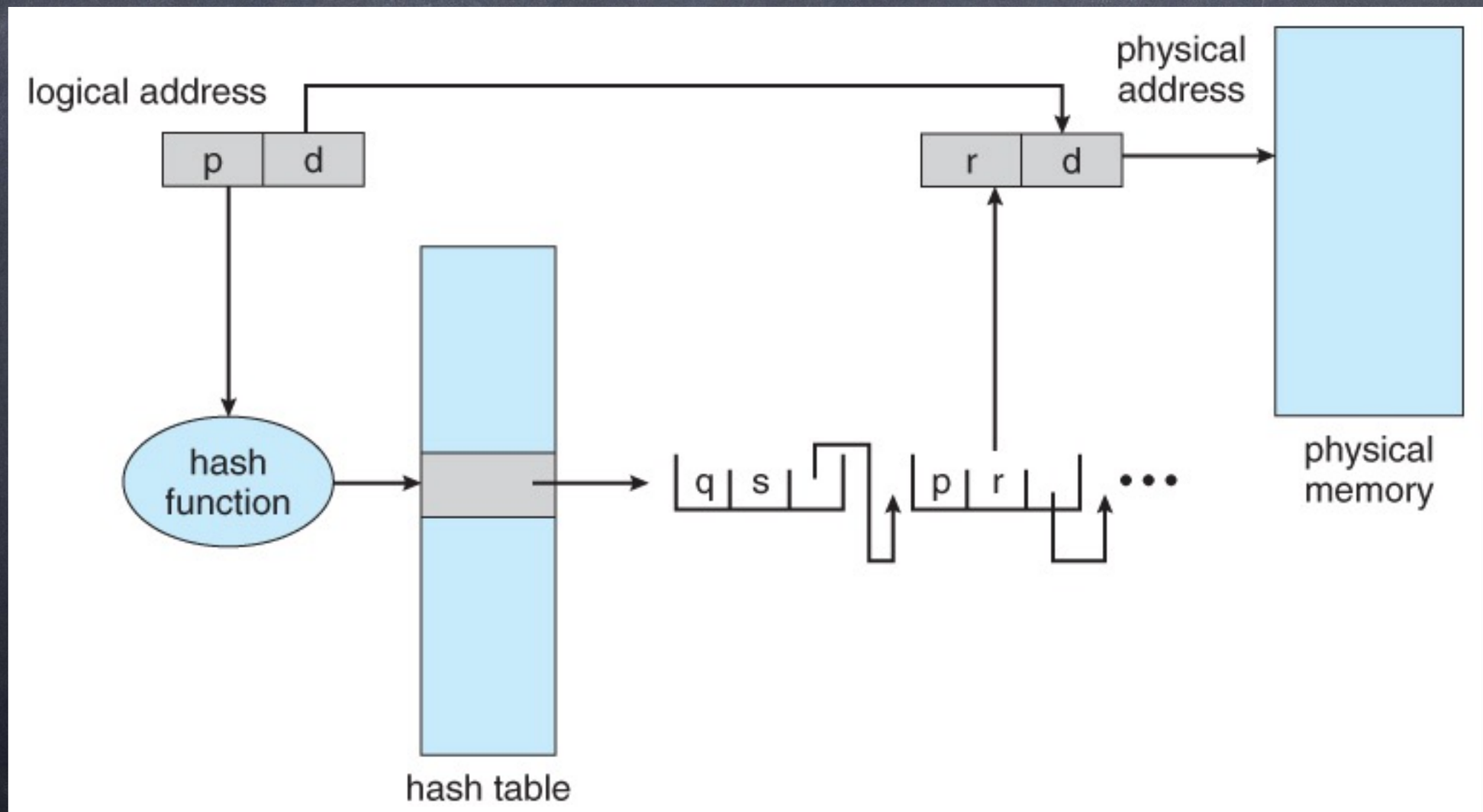
Hierarchical Paging



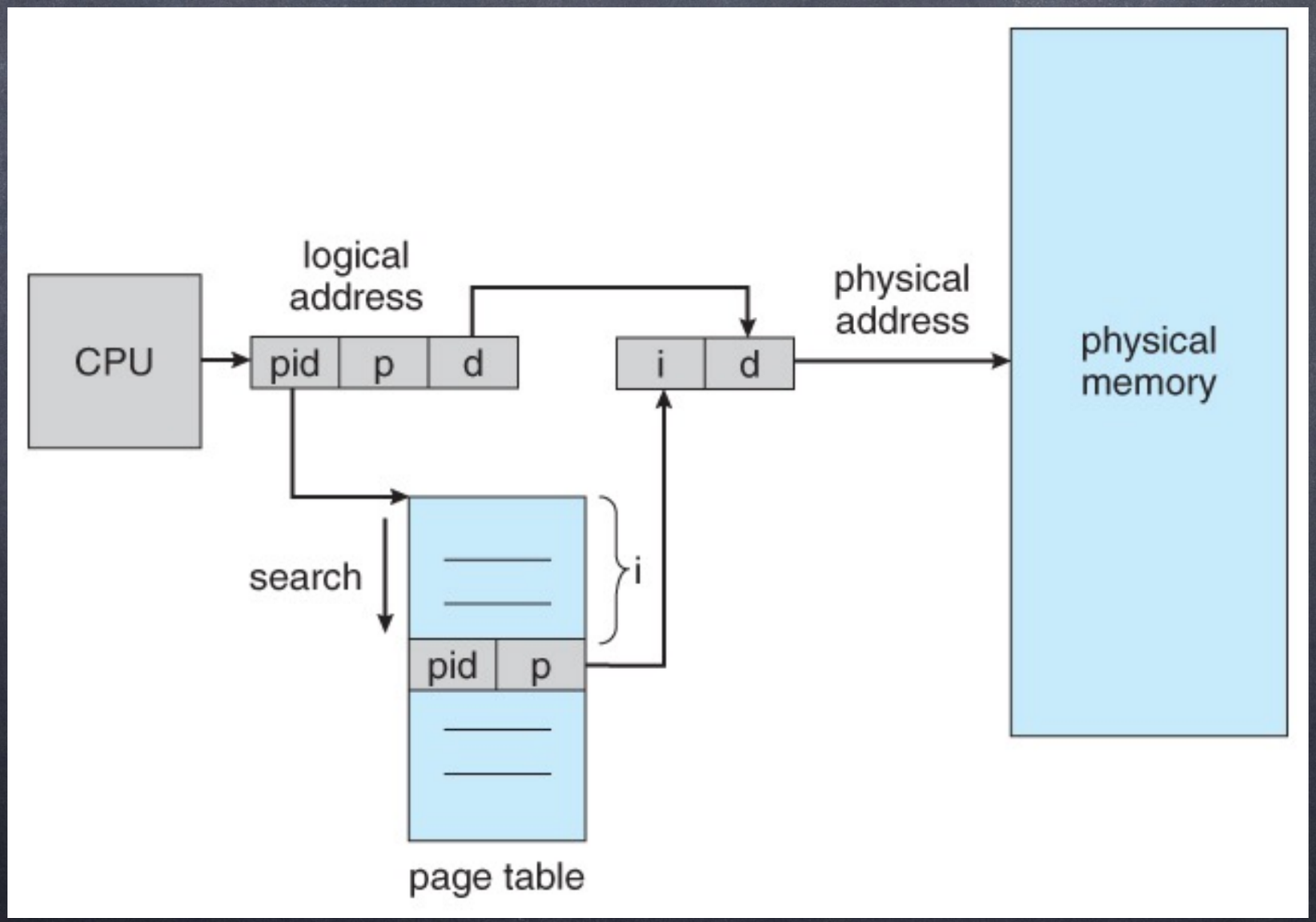




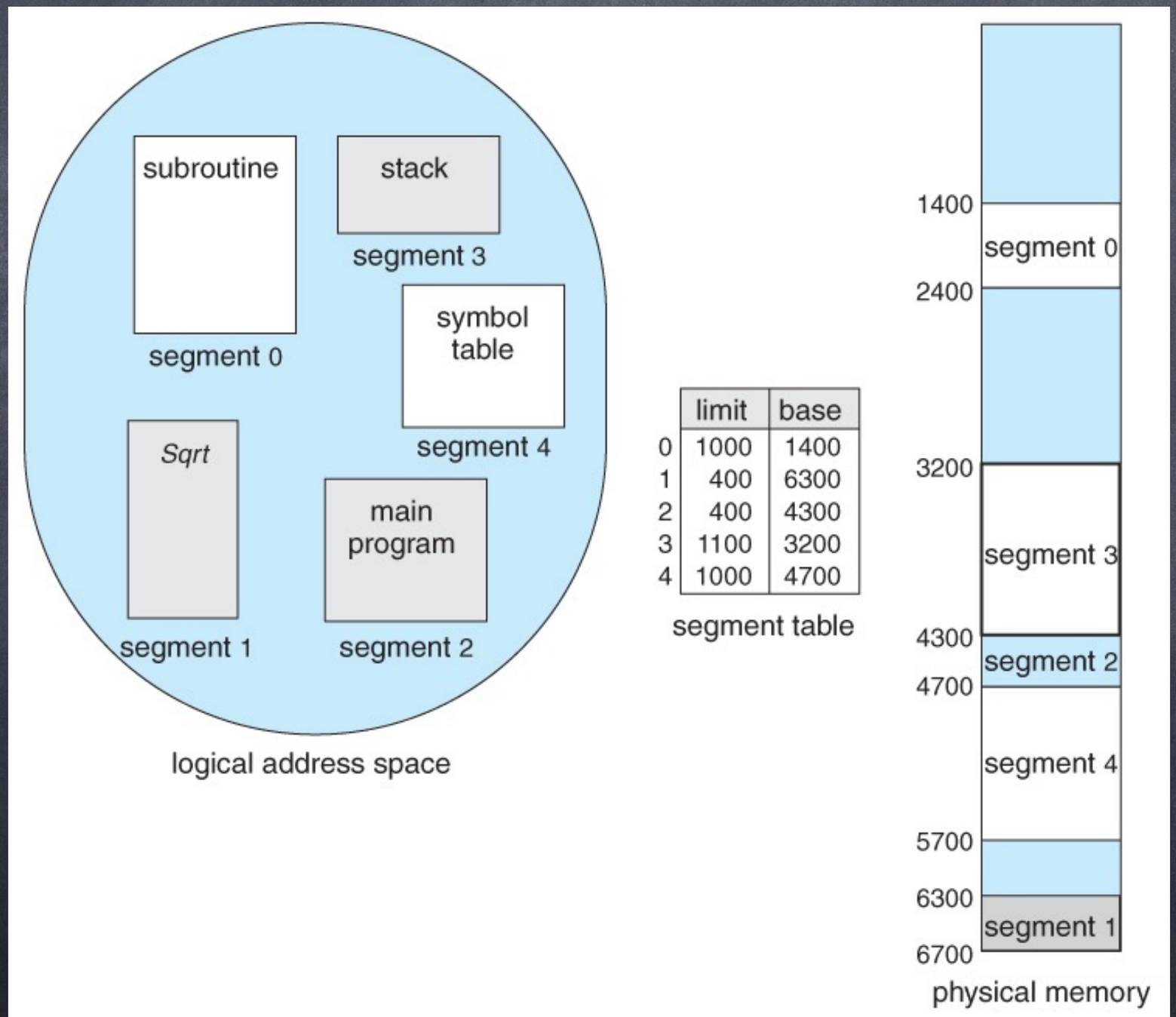
Hashed Paging



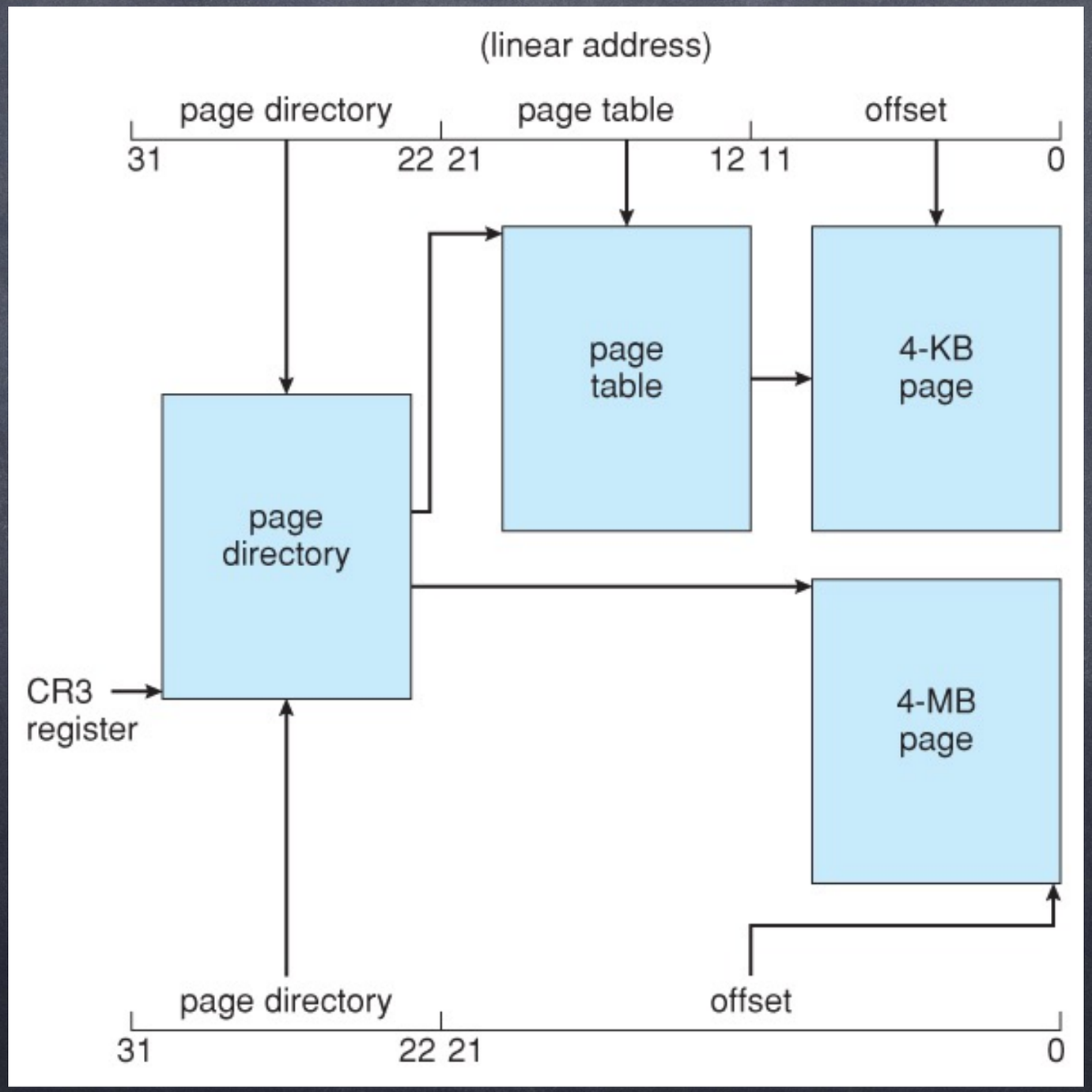
Inverted Paging



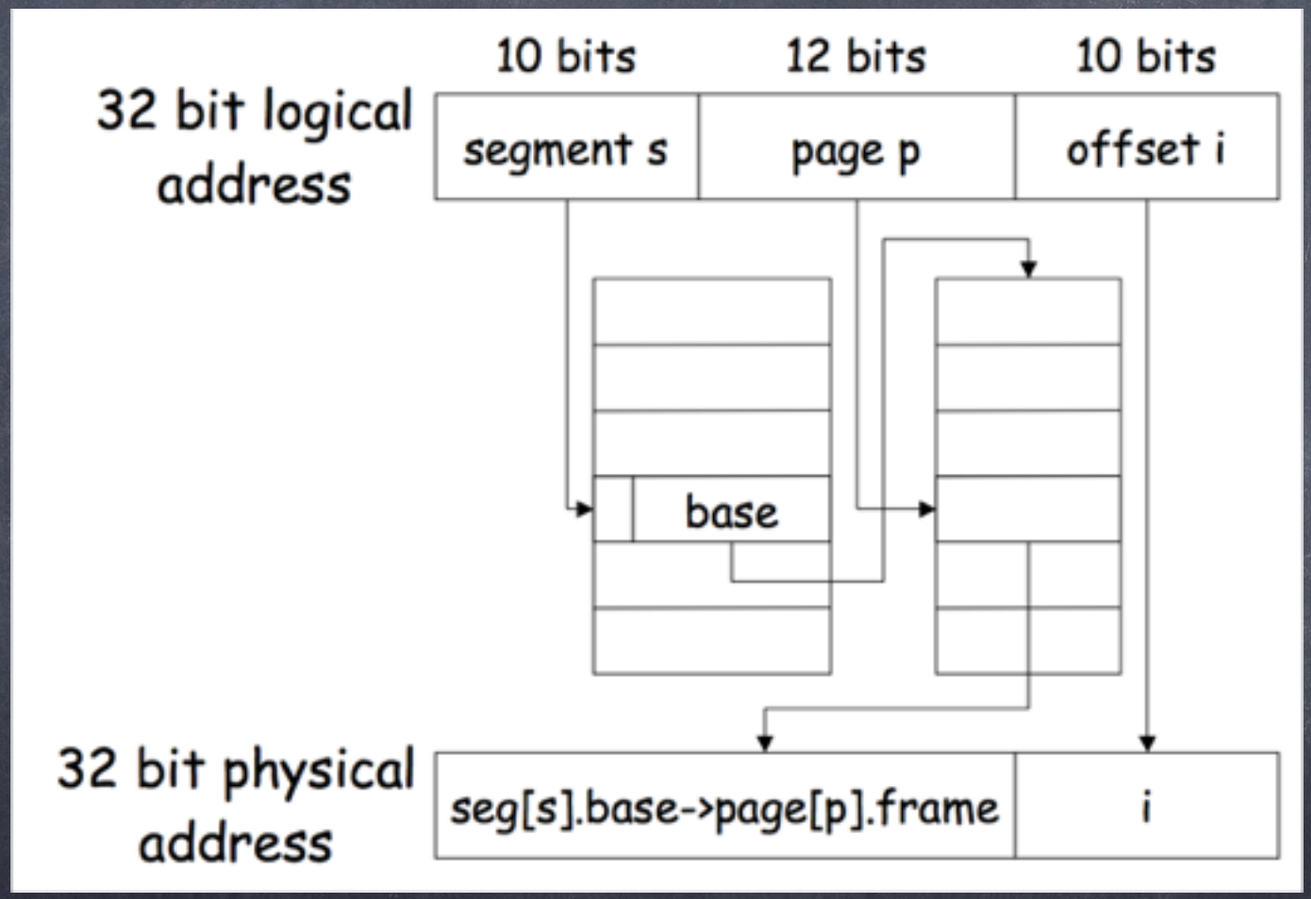
Segmentation



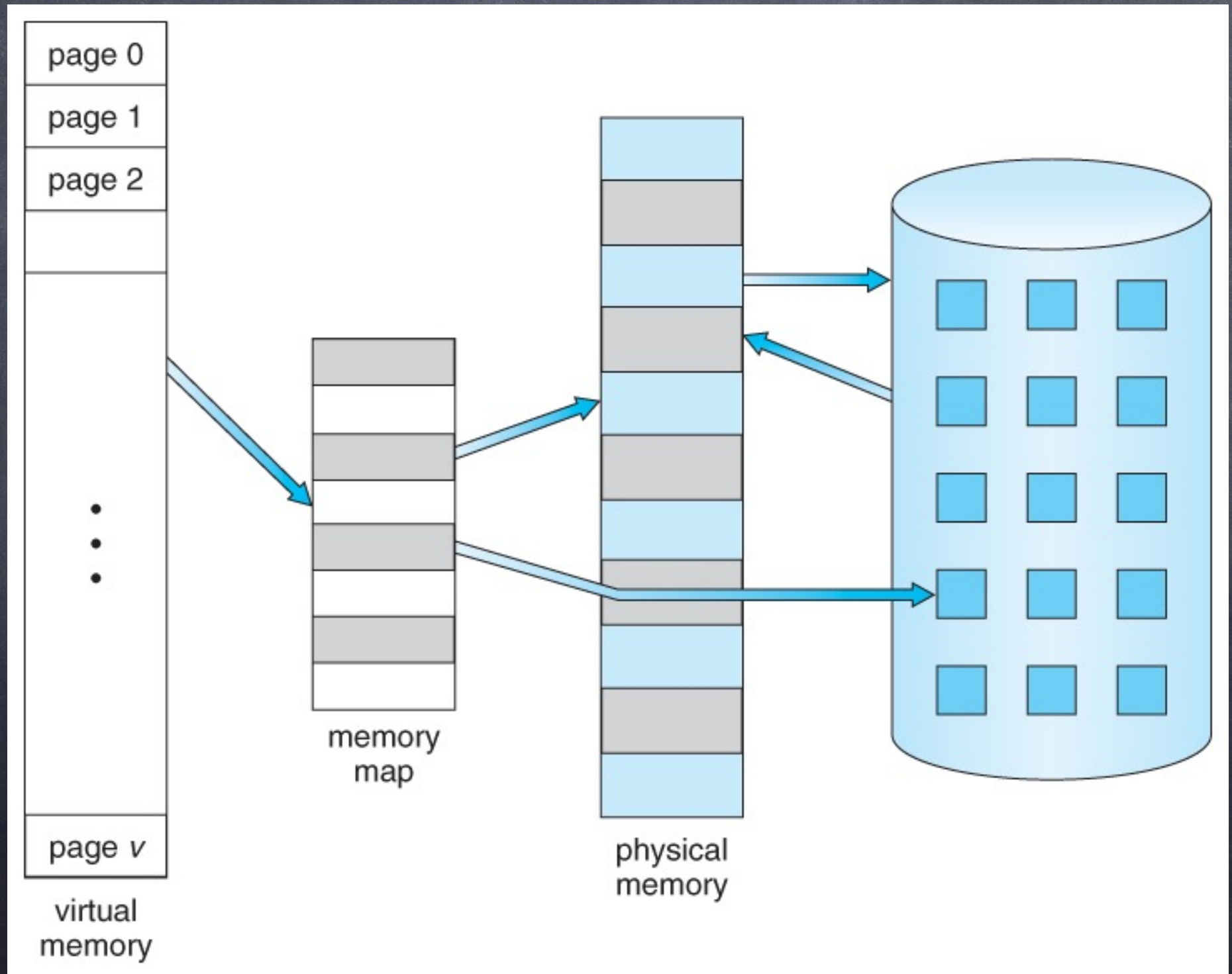
Intel CPU


(linear address)
page directory
page table
offset
31
22 21
12 11
0
page directory
page table
4-KB page
4-MB page
CR3 register
page directory
offset
31
22 21
0

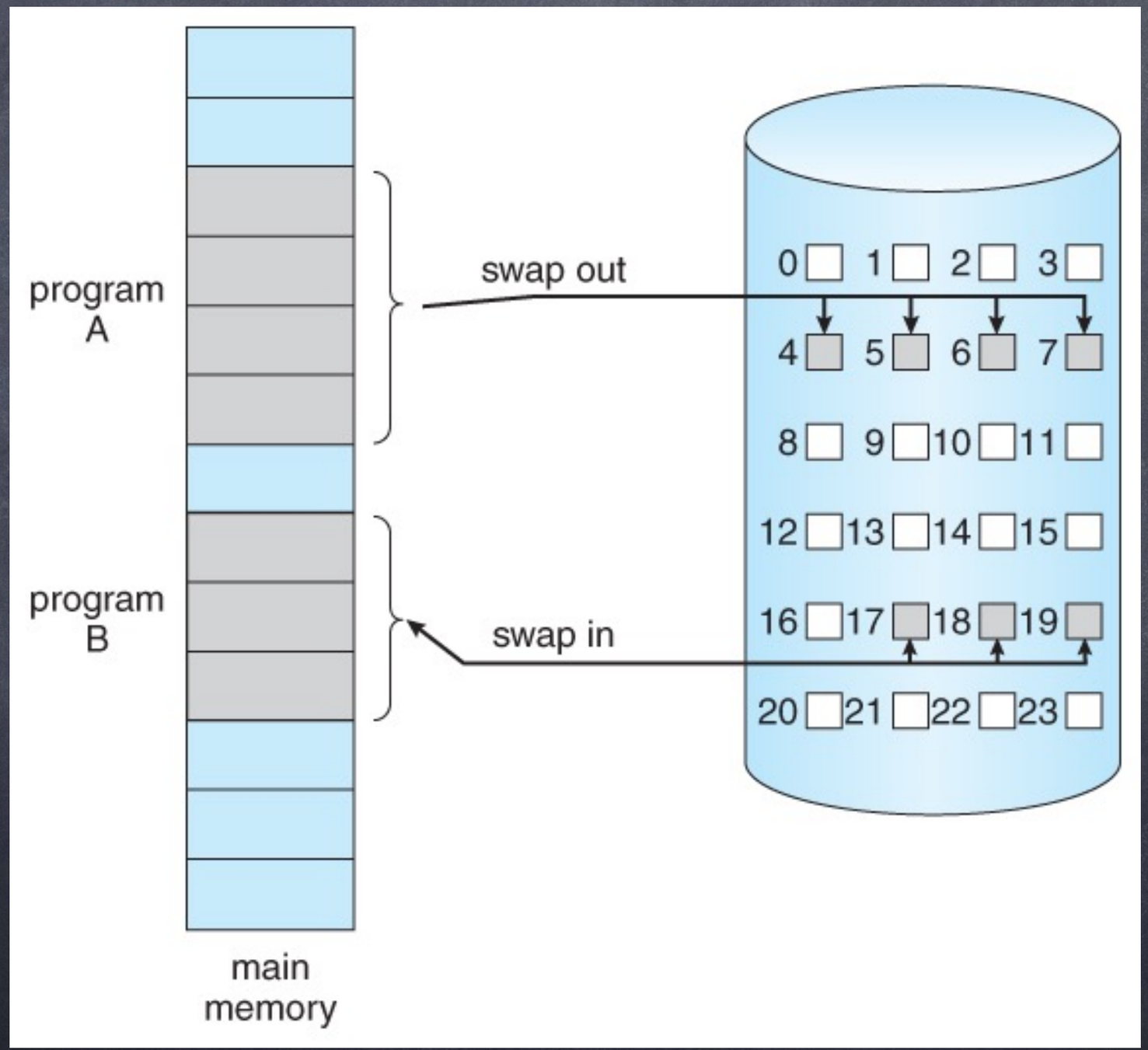
Segmentation + Paging



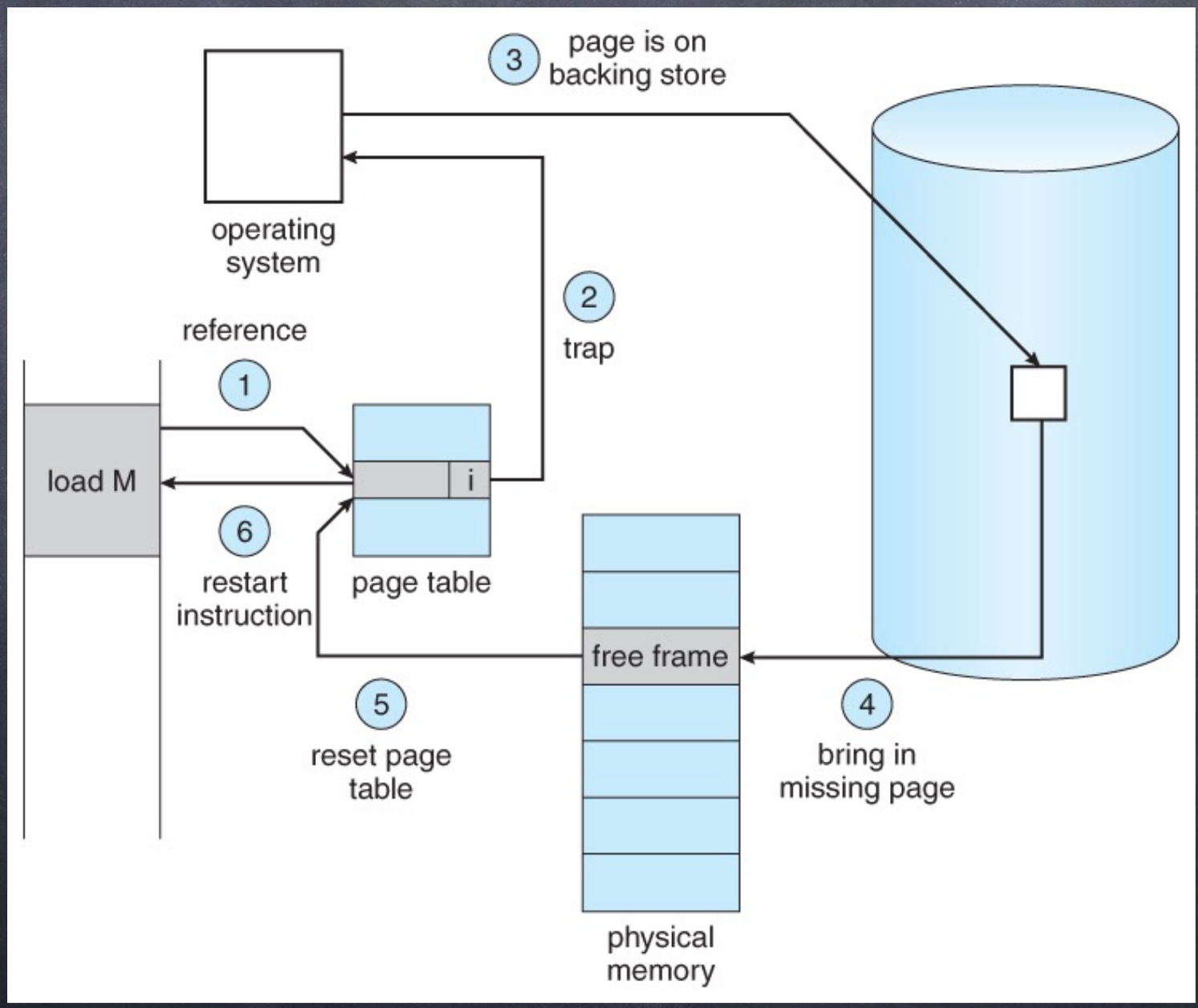
Virtual Memory



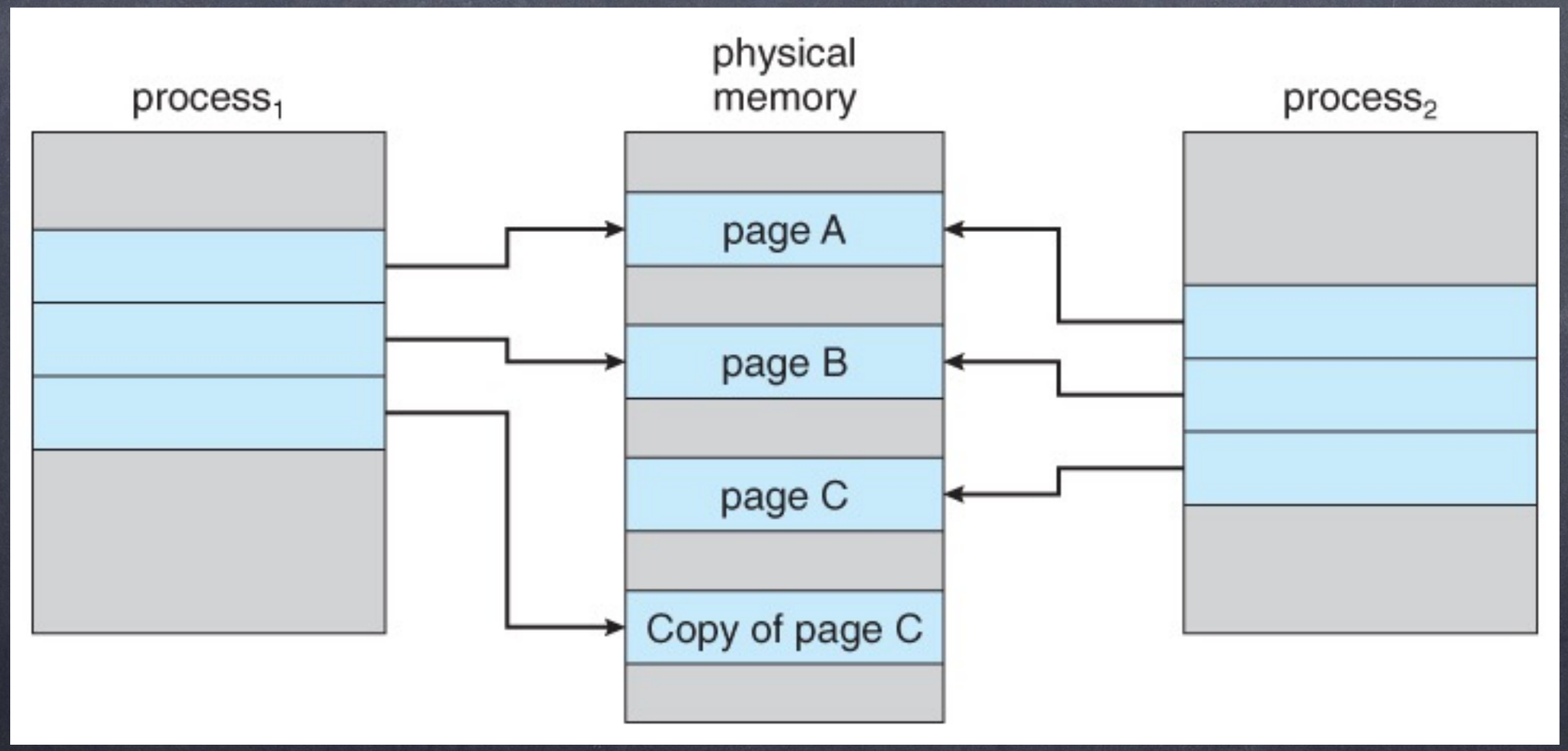
Demand Paging



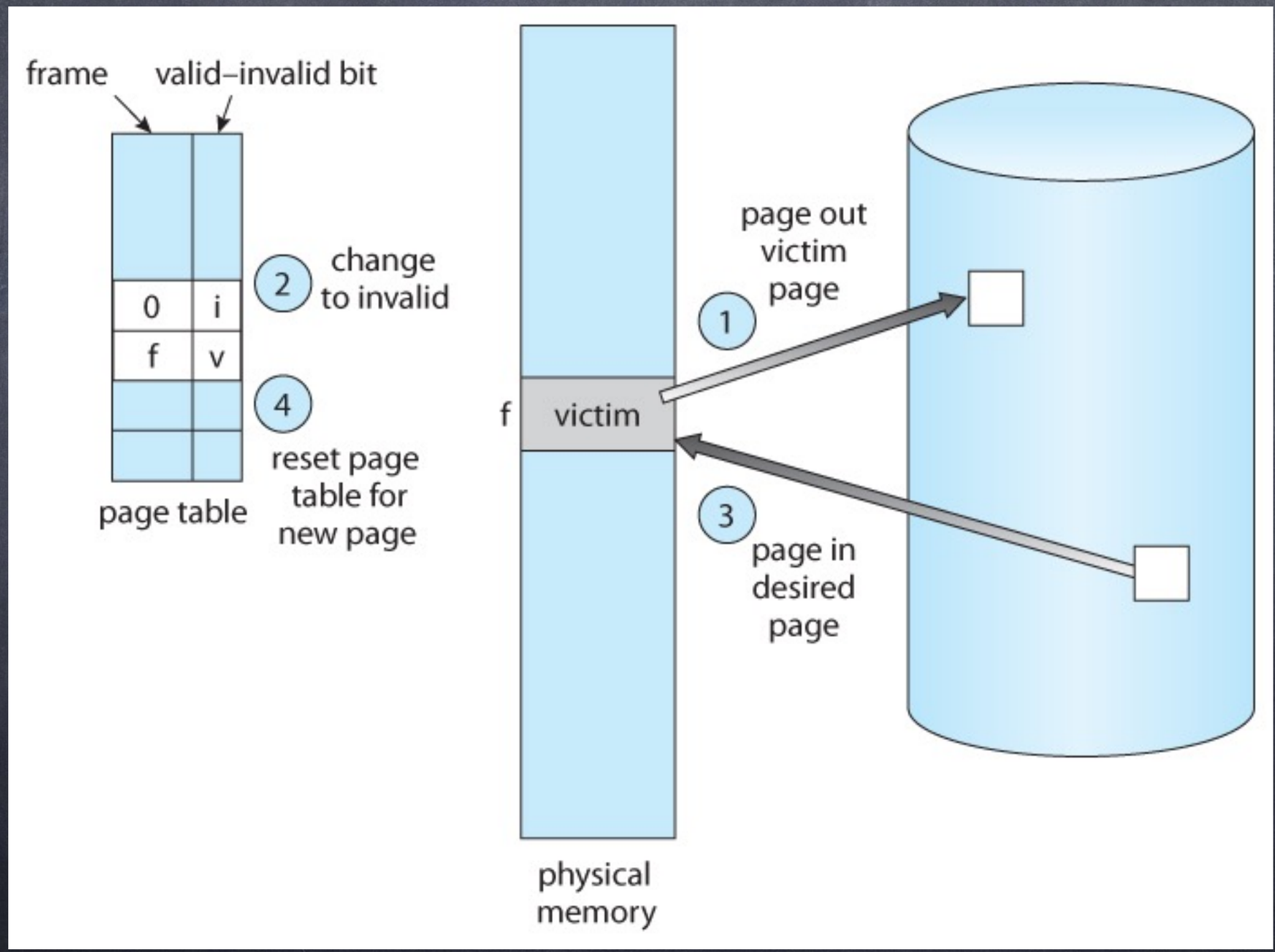
Page Fault


3
page is on
backing store
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

Copy on Write



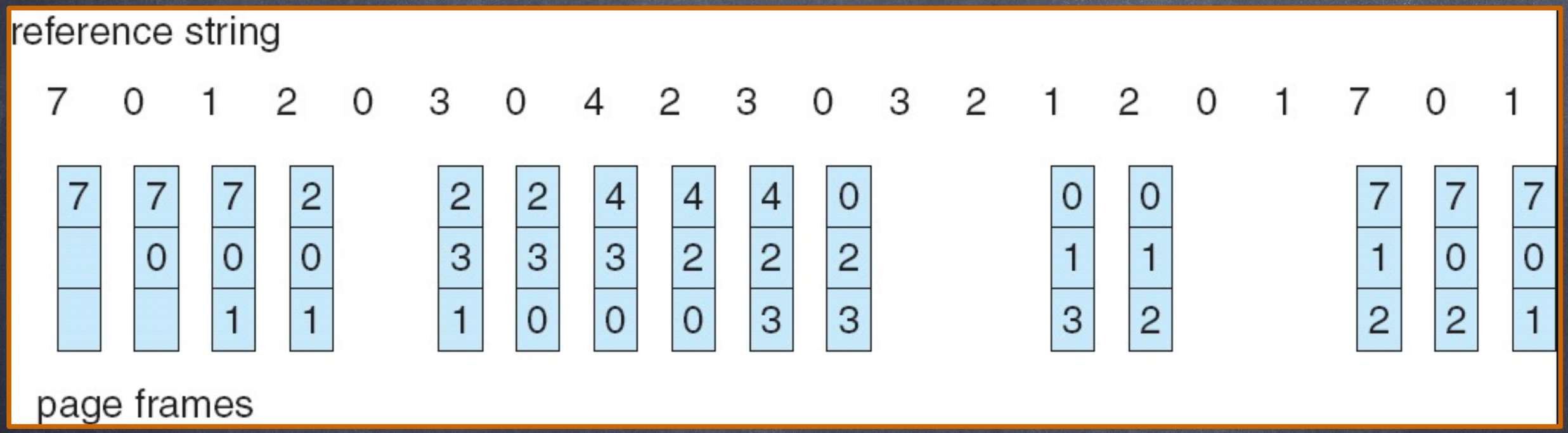
Page Replacement



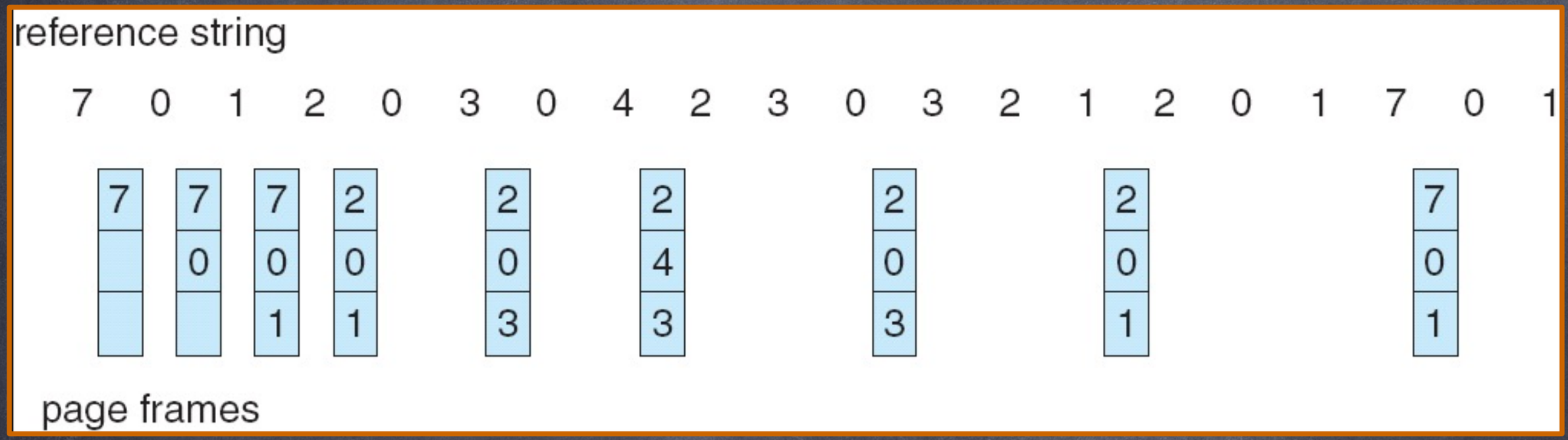
Replacement Algorithm

- FIFO
- Optimal
- LRU
- Reference Bit
- Second Chance

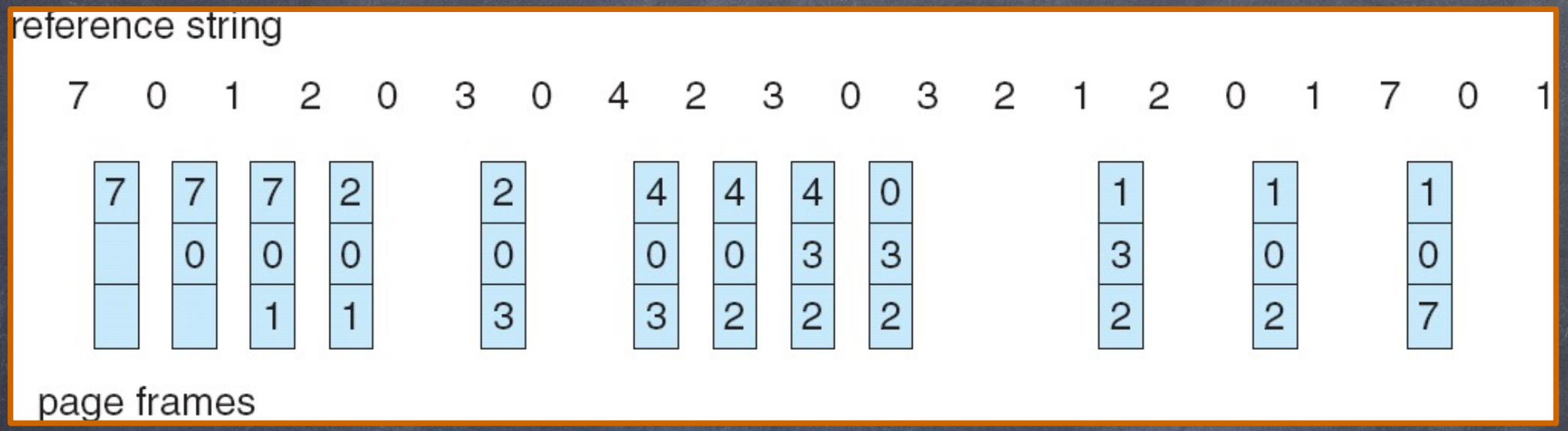
FIFO Page Replacement



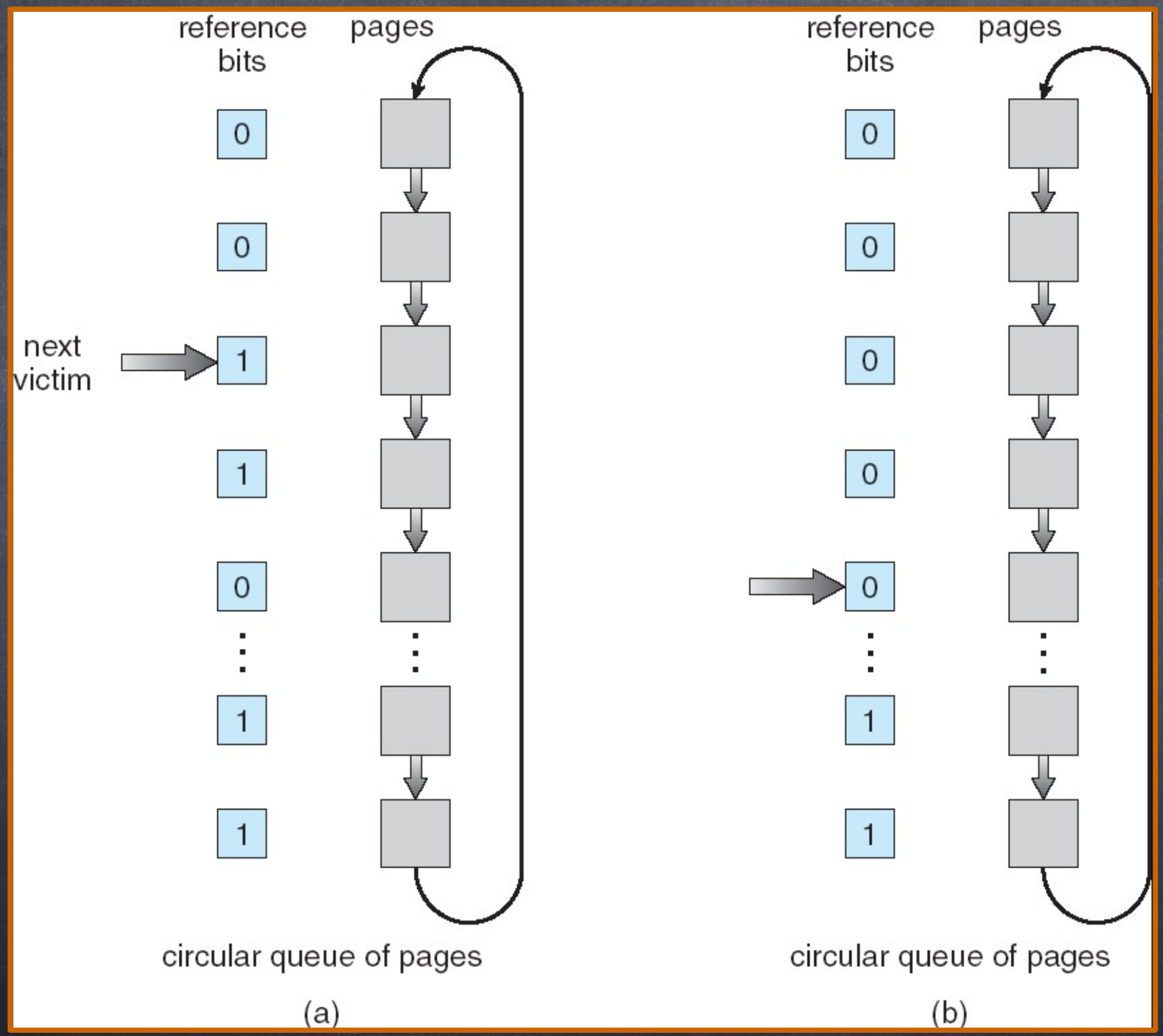
Optimal Page Replacement



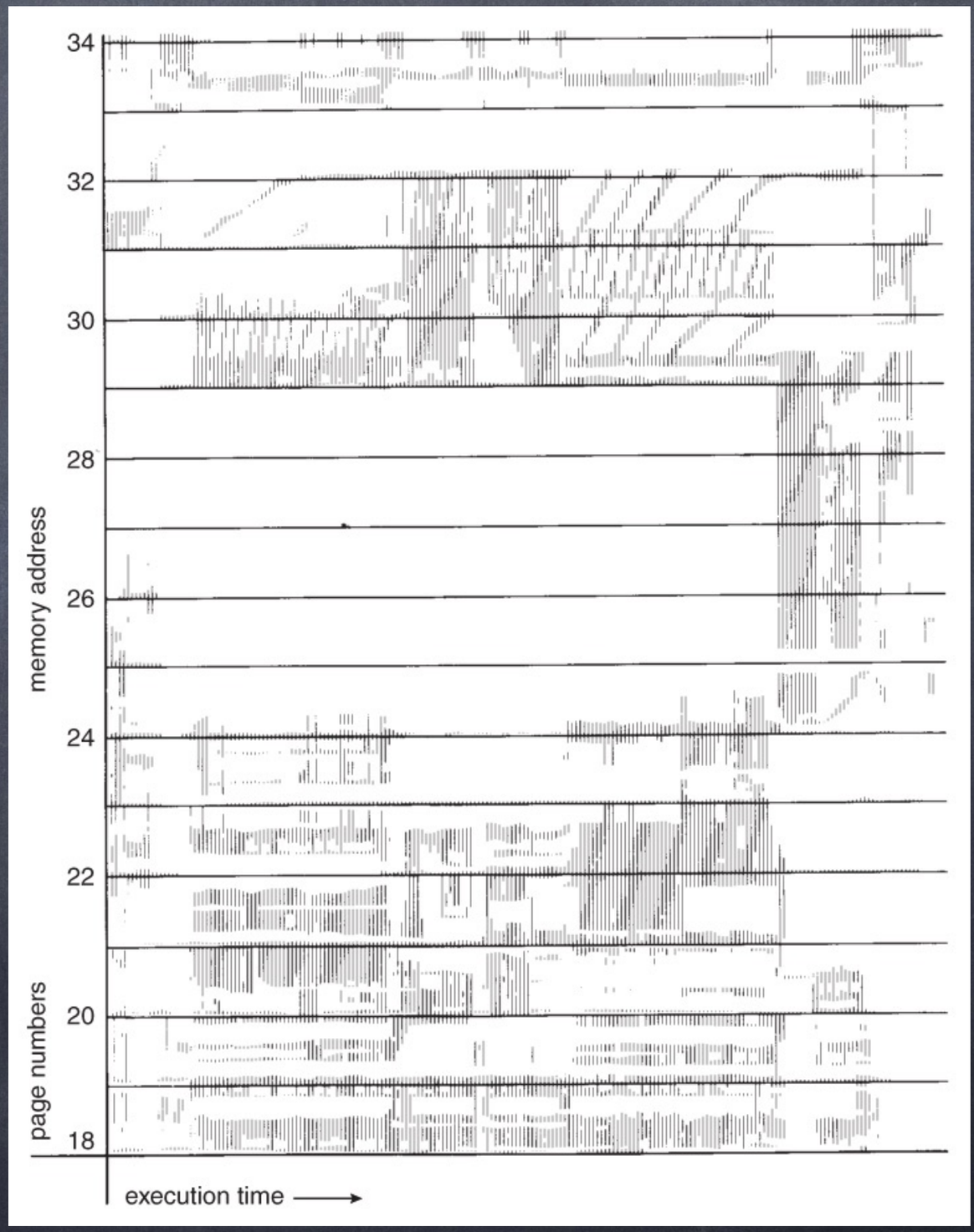
LRU Page Replacement



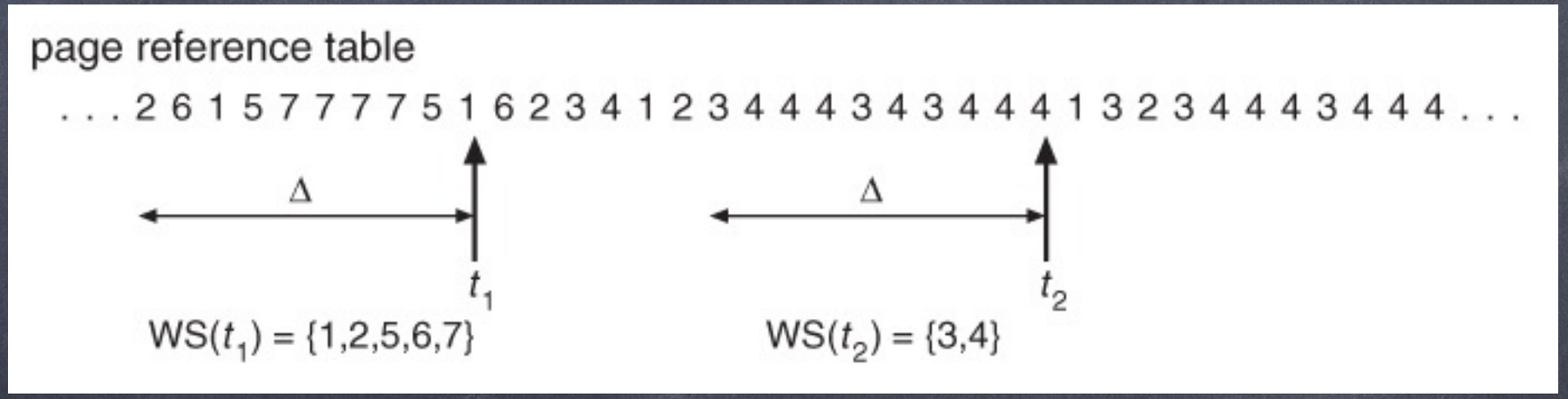
Second-Chance (clock) Page-Replacement Algorithm



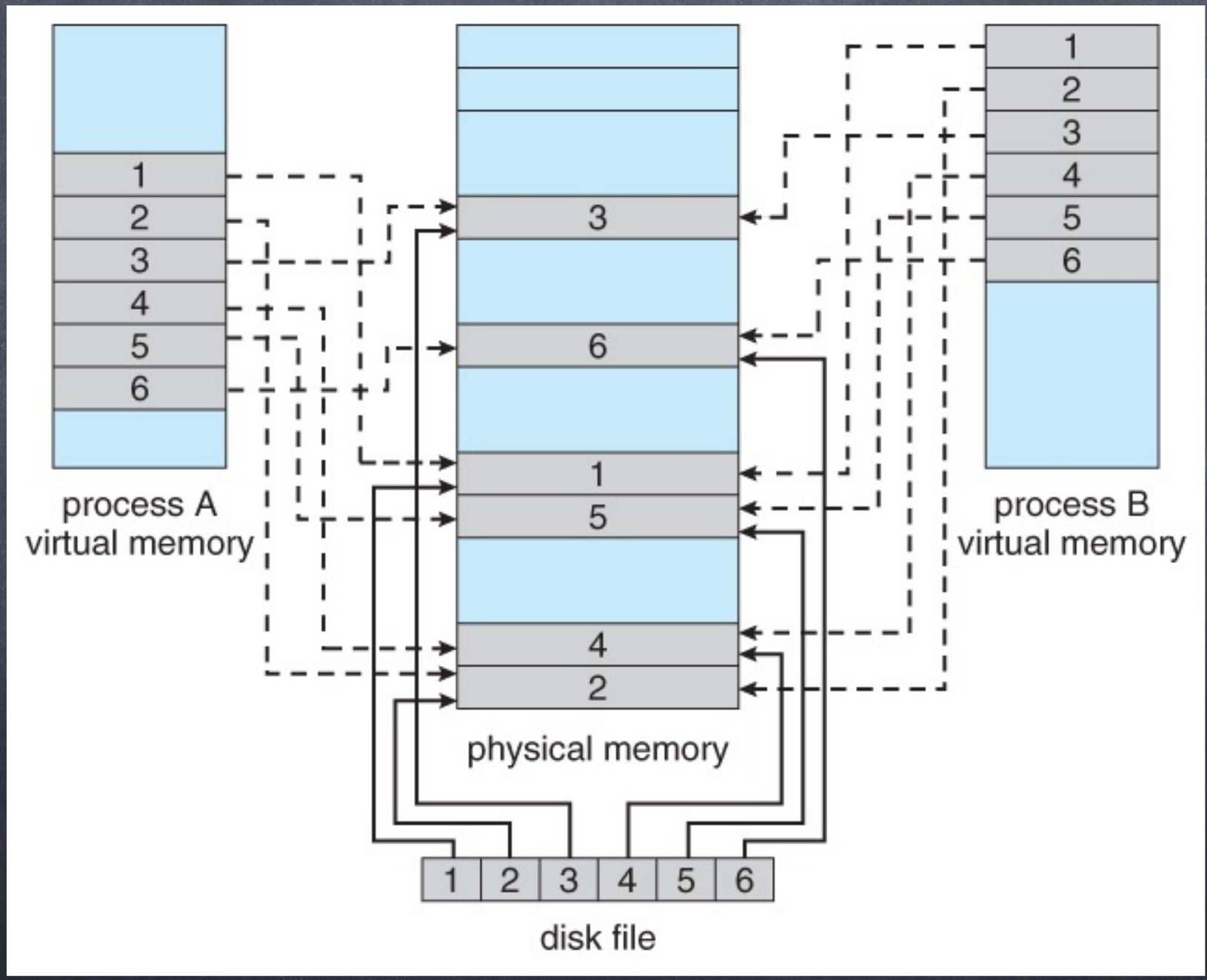
Thrashing


34
32
30
28
26
24
22
20
18
memory address
page numbers
execution time

Working Set


page reference table
. . . 2 6 1 5 7 7 7 7 5 1 6 2 3 4 1 2 3 4 4 4 3 4 3 4 4 4 1 3 2 3 4 4 4 3 4 4 4 . . .
Δ
t_1
WS(t_1) = {1,2,5,6,7}
Δ
t_2
WS(t_2) = {3,4}

Memory Mapped File



Kernel Memory Allocation

- Buddy Allocation

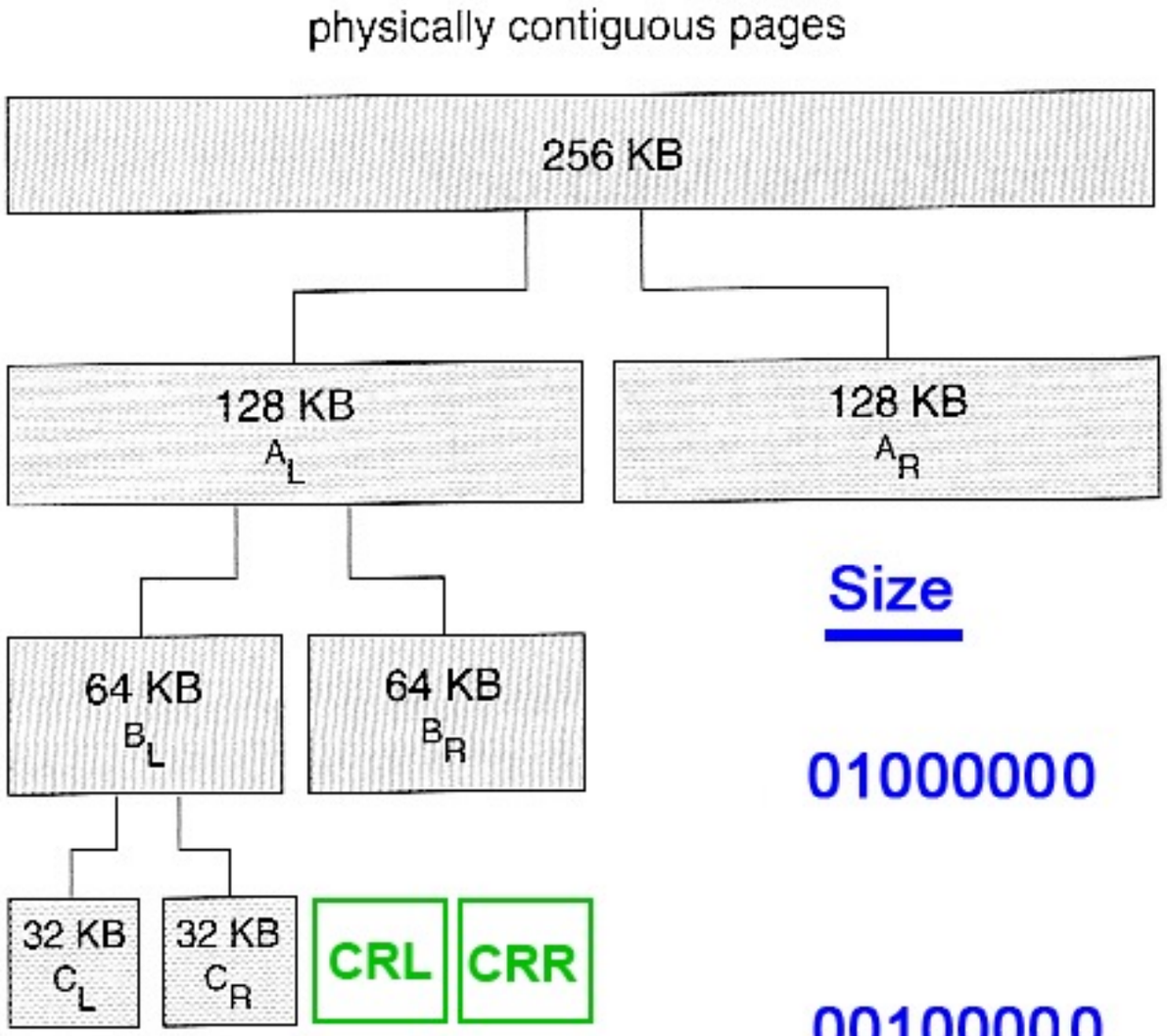


Figure 9.27 Buddy system allocation.

Buddy Addresses

00000000

00000000 10000000

Size

01000000 00000000 01000000

00100000 00000000 00100000

0100000 0110000

Slab Allocation