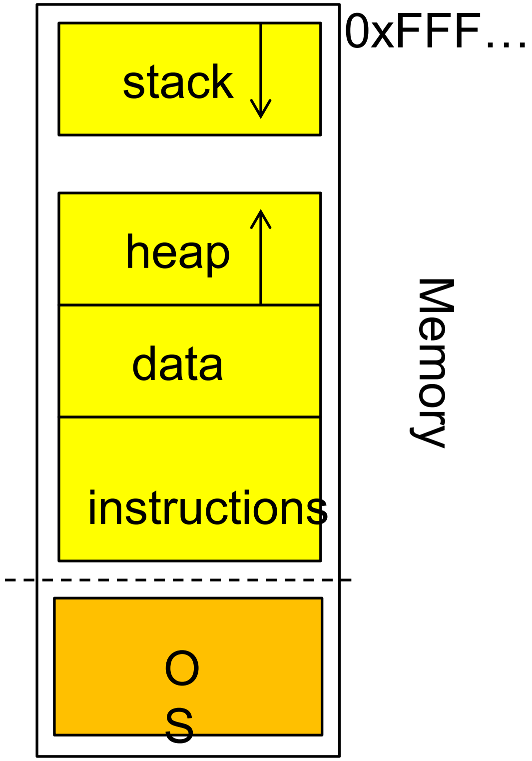
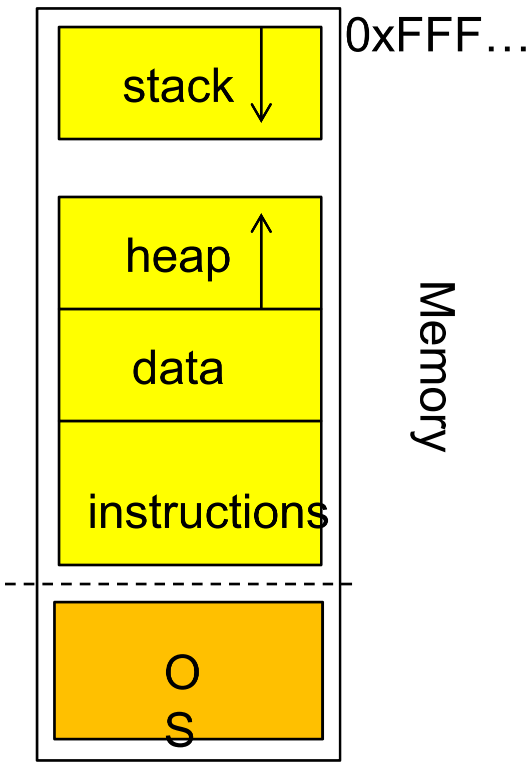
Operating Systems
Paging and Virtual Memory

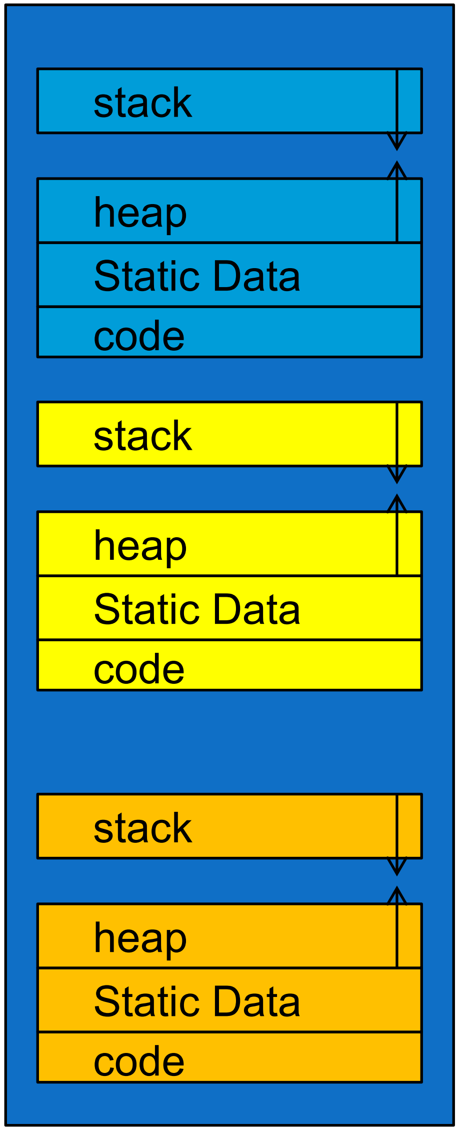
David Hay
Dror Feitelson

Process View Vs. Reality

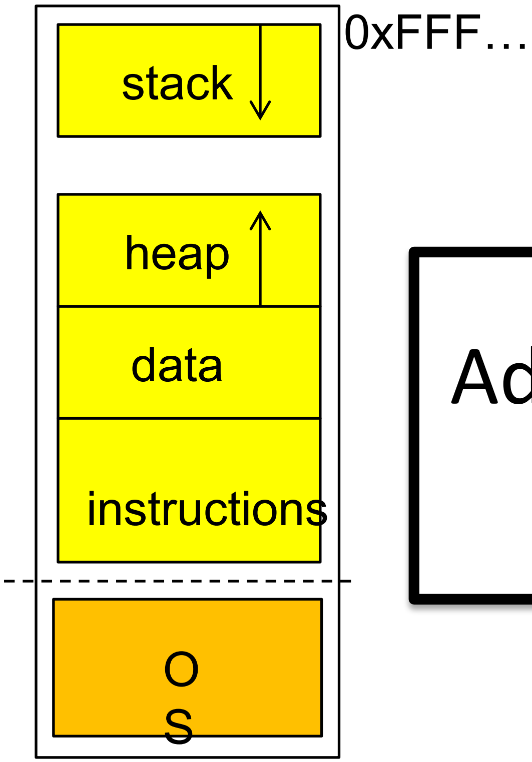


Process View Vs. Reality

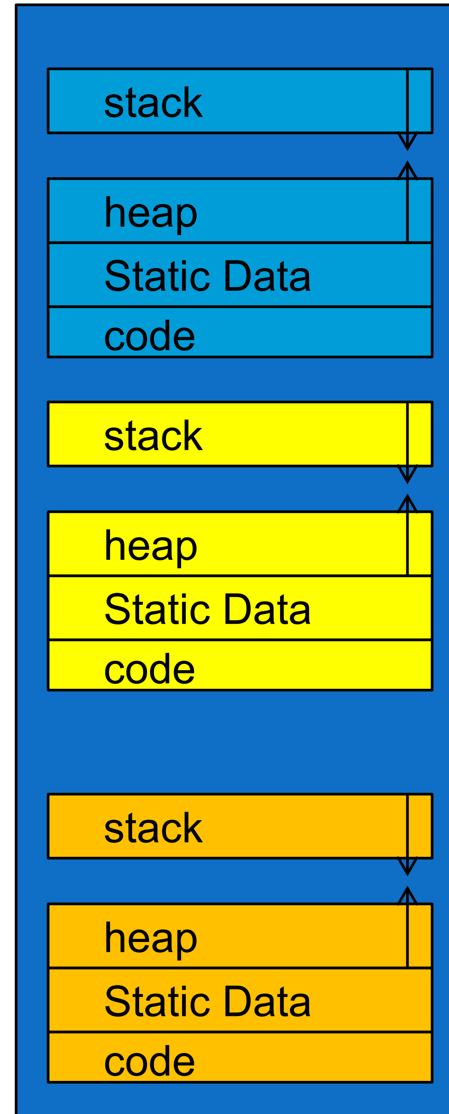




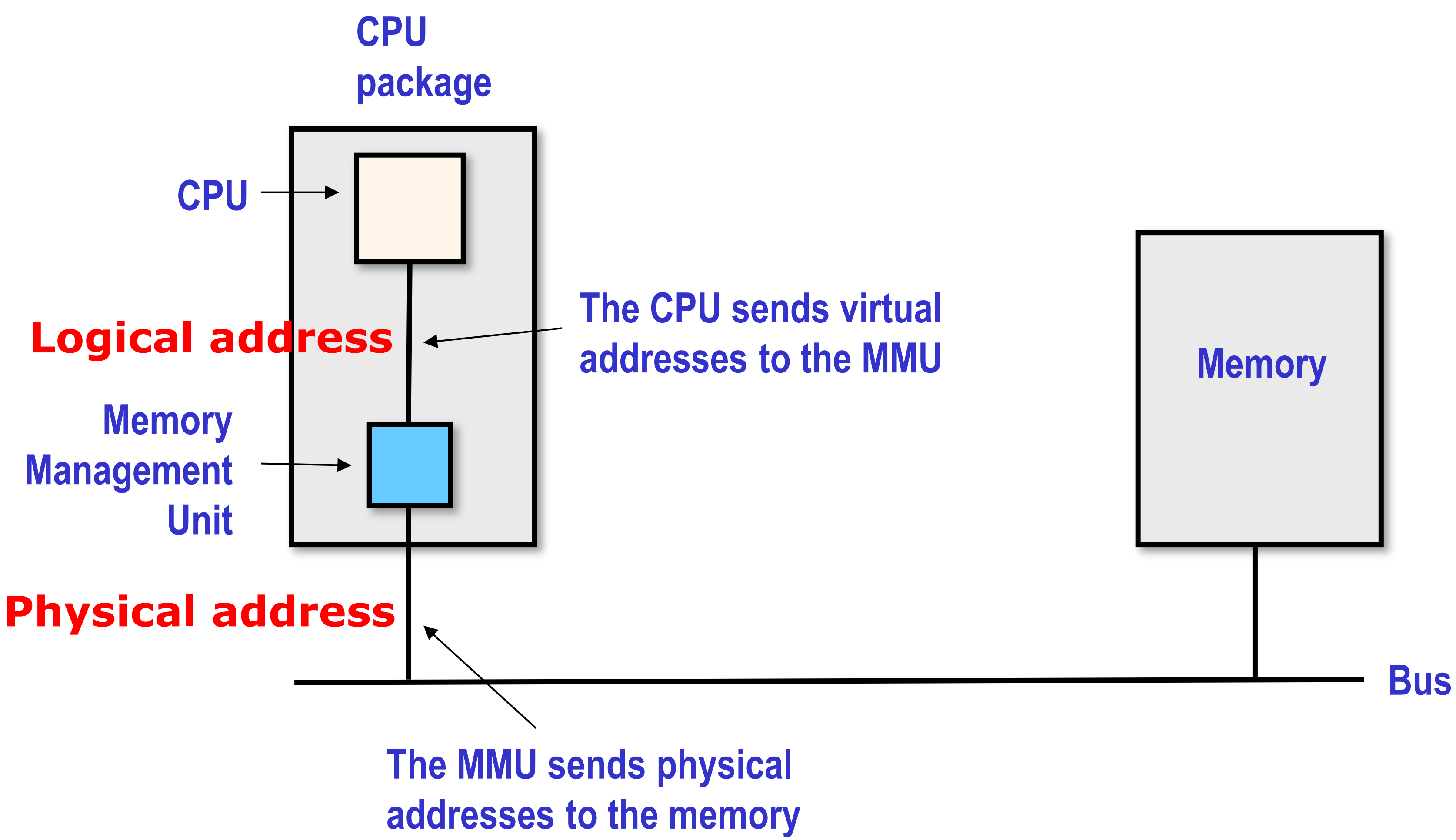
Process View Vs. Reality



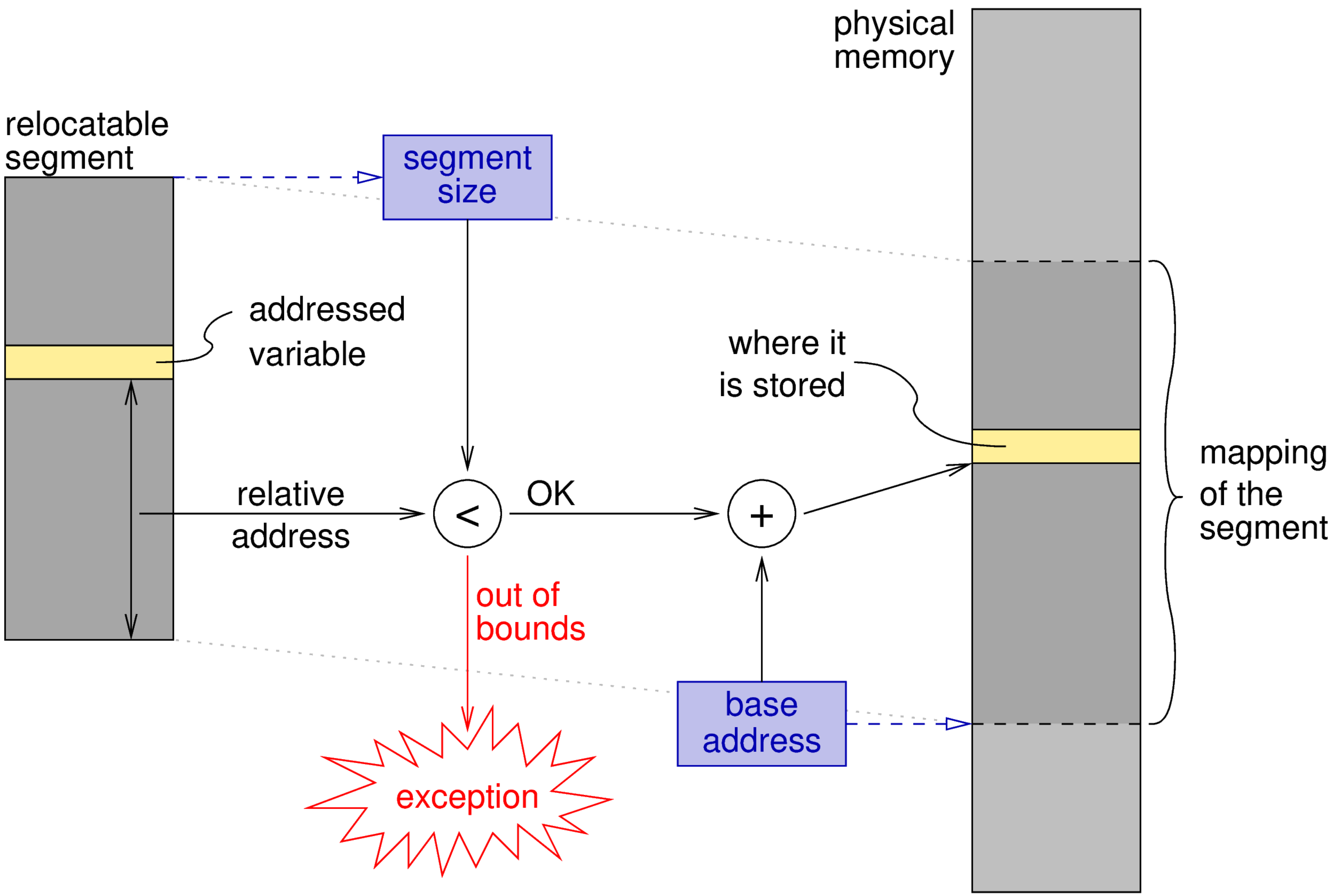
Address translation is required!



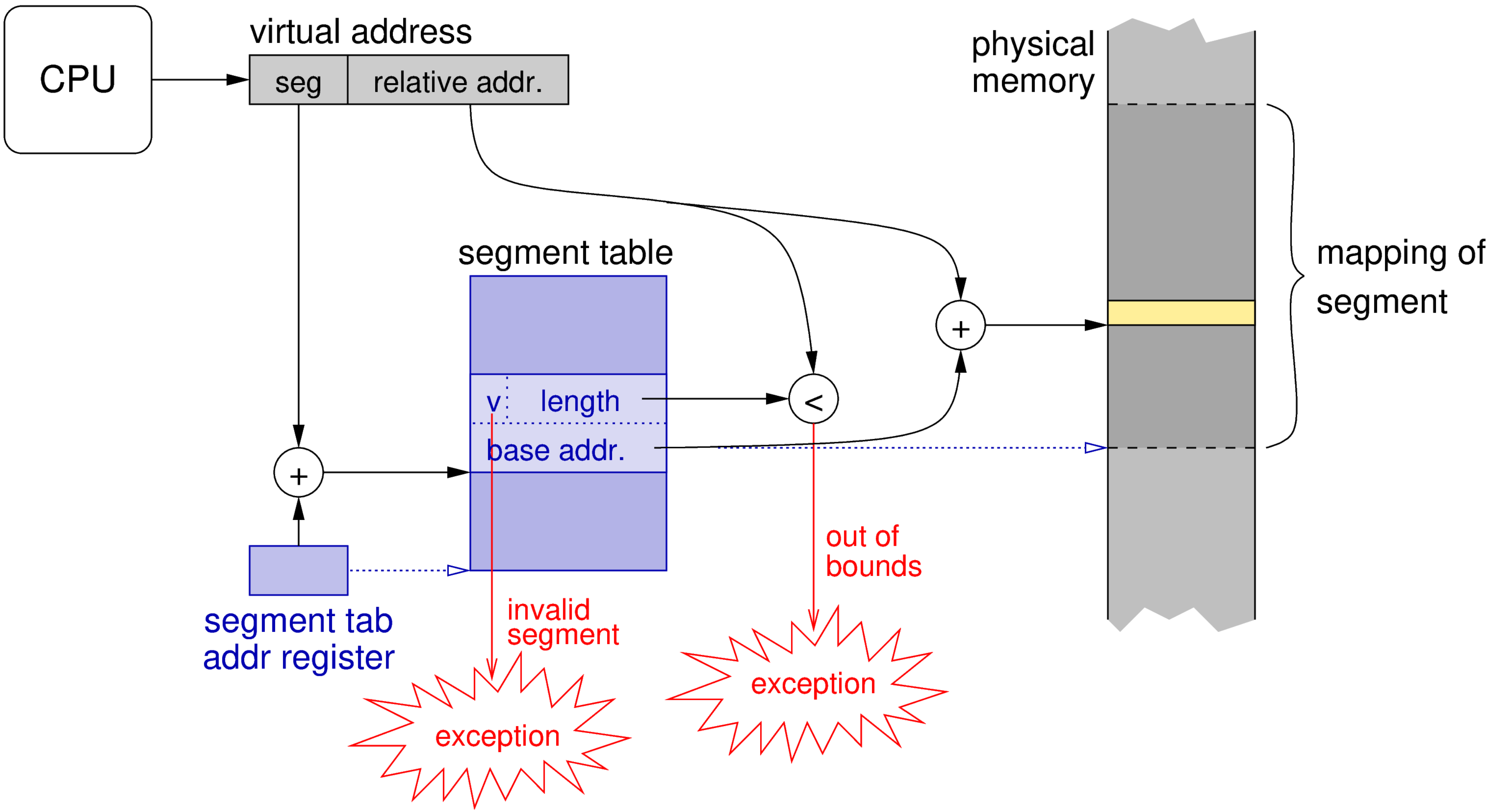
Memory Addressing Architecture



Segment Address Translation



Using a Segment Table



Fragmentation

- Situation in which we have enough memory to allocate to a process, but not contiguously, as the holes are scattered all over memory
  - Internal Fragmentation: free memory inside process' allocation
  - External Fragmentation: free memory between processes' allocations
- Contiguous allocation suffers from external fragmentation

The Solution: Paging

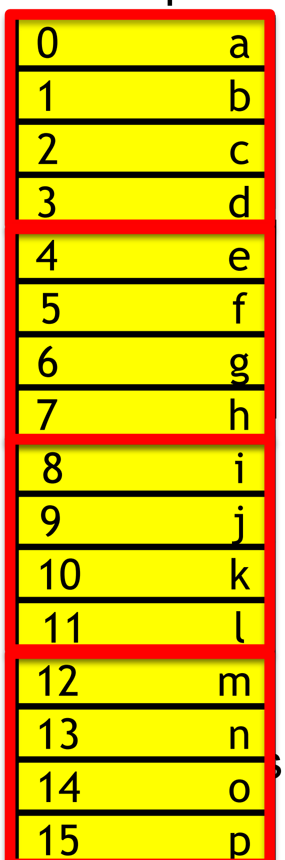
- Divide process address space into fixed-size **pages** (usually 4KB)
- Divide physical memory into **frames** of the same size
- Any page can be mapped to any frame
  - No external fragmentation!
- Mapping stored in page table
- Do not need to map a whole segment
  - Only the parts we are using

The Page Table

- Maps pages to frames
- Separate page table for each process
  - Or each segment
  - Switch tables as part of context switch
- Populated by the operating system
  - Reflects decisions what to map where
- Used by the MMU
  - To perform memory access at hardware speed

Address Translation with the Page Table

Logical addr. space



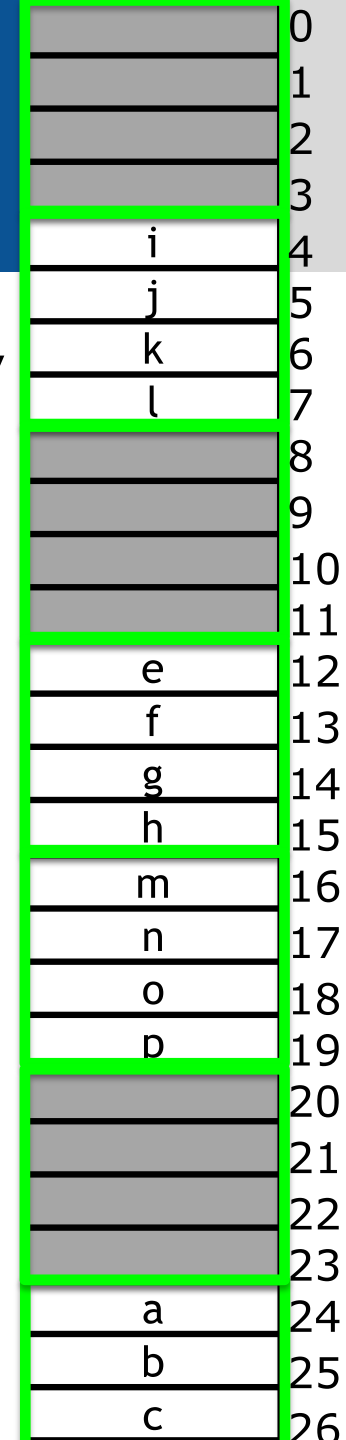
Page Table

Page	Frame
0	6
1	3
2	1
3	4

(per process)

Used by MMU

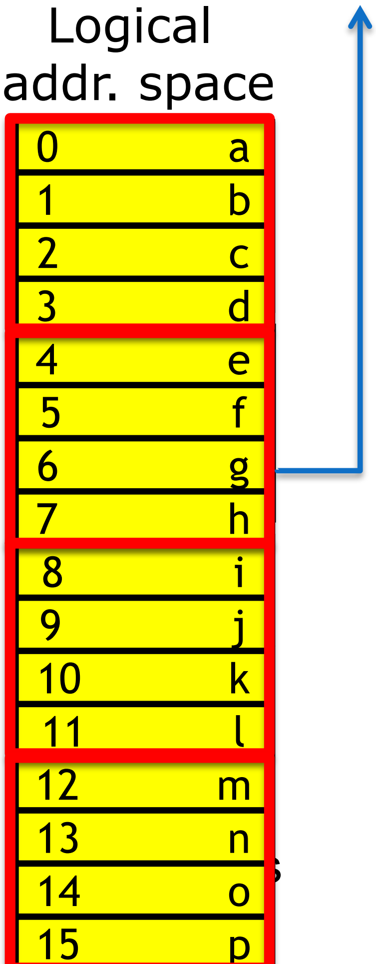
Physical memory



Address Translation with the Page Table

Address
6=00110

Logical
addr. space



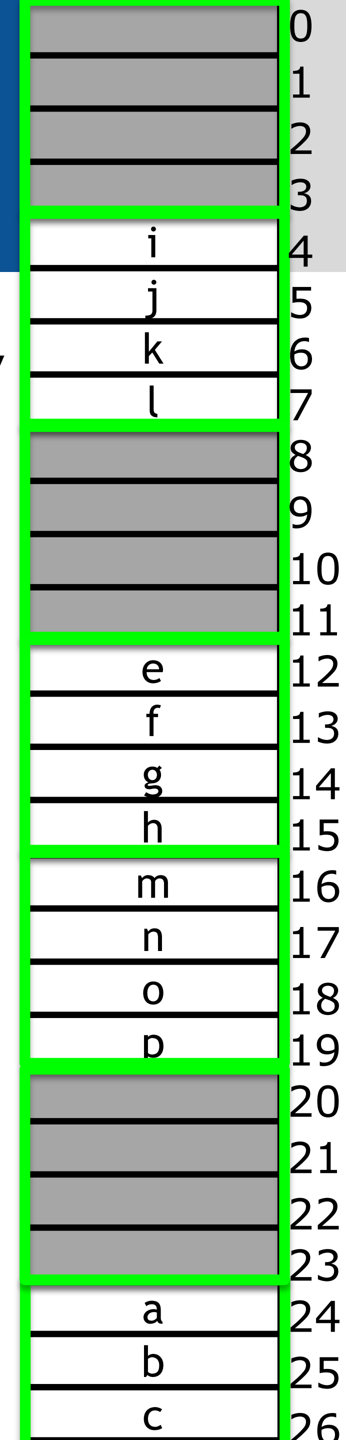
Page Table

Page	Frame
0	6
1	3
2	1
3	4

(per process)

Used by MMU

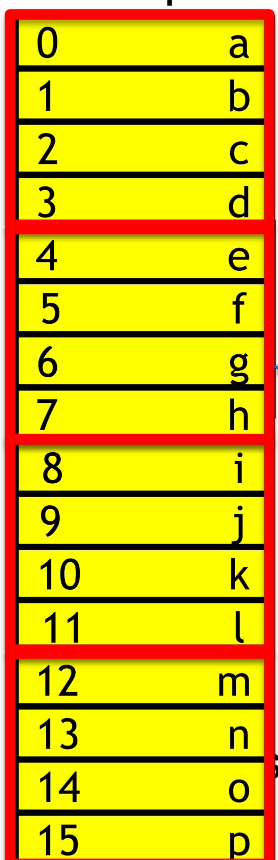
Physical
memory



Address Translation with the Page Table

Address
6=00110 → Page Offset
00110

Logical
addr. space



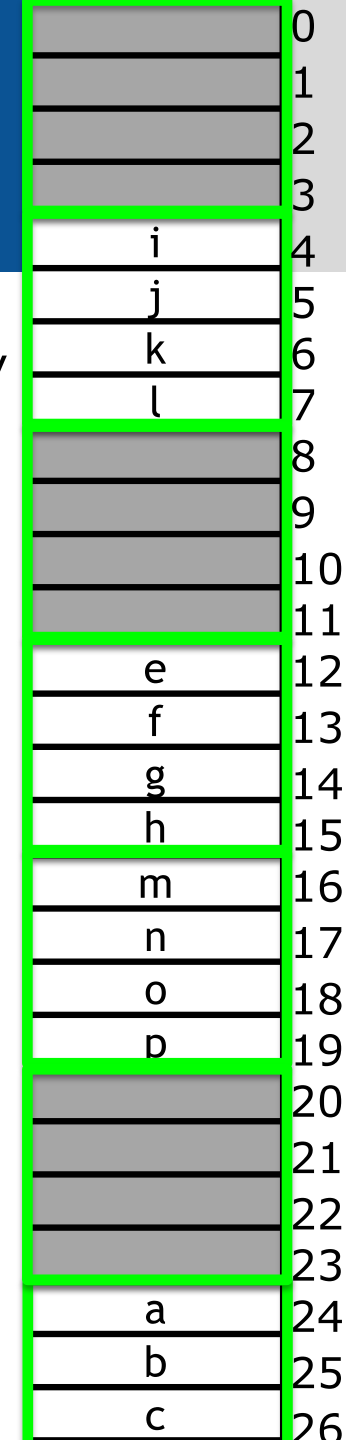
Page Table

Page	Frame
0	6
1	3
2	1
3	4

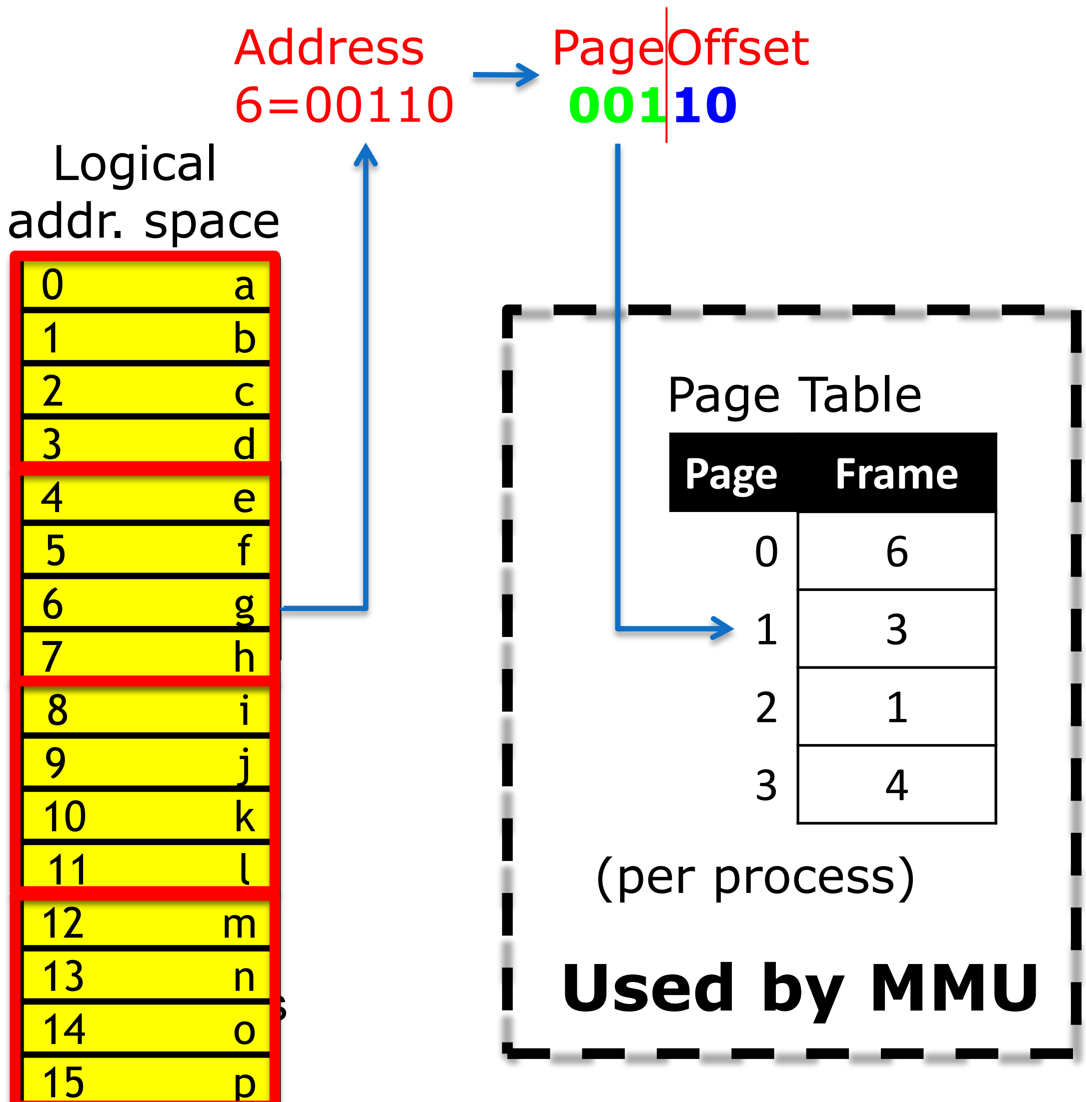
(per process)

Used by MMU

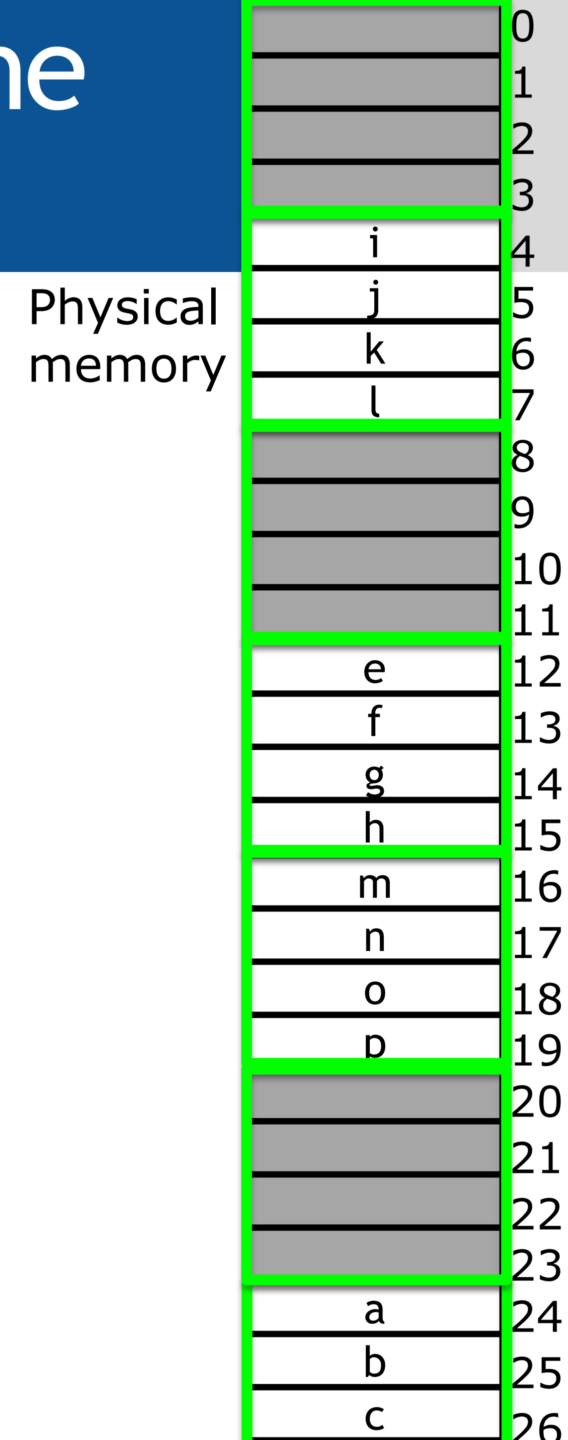
Physical
memory



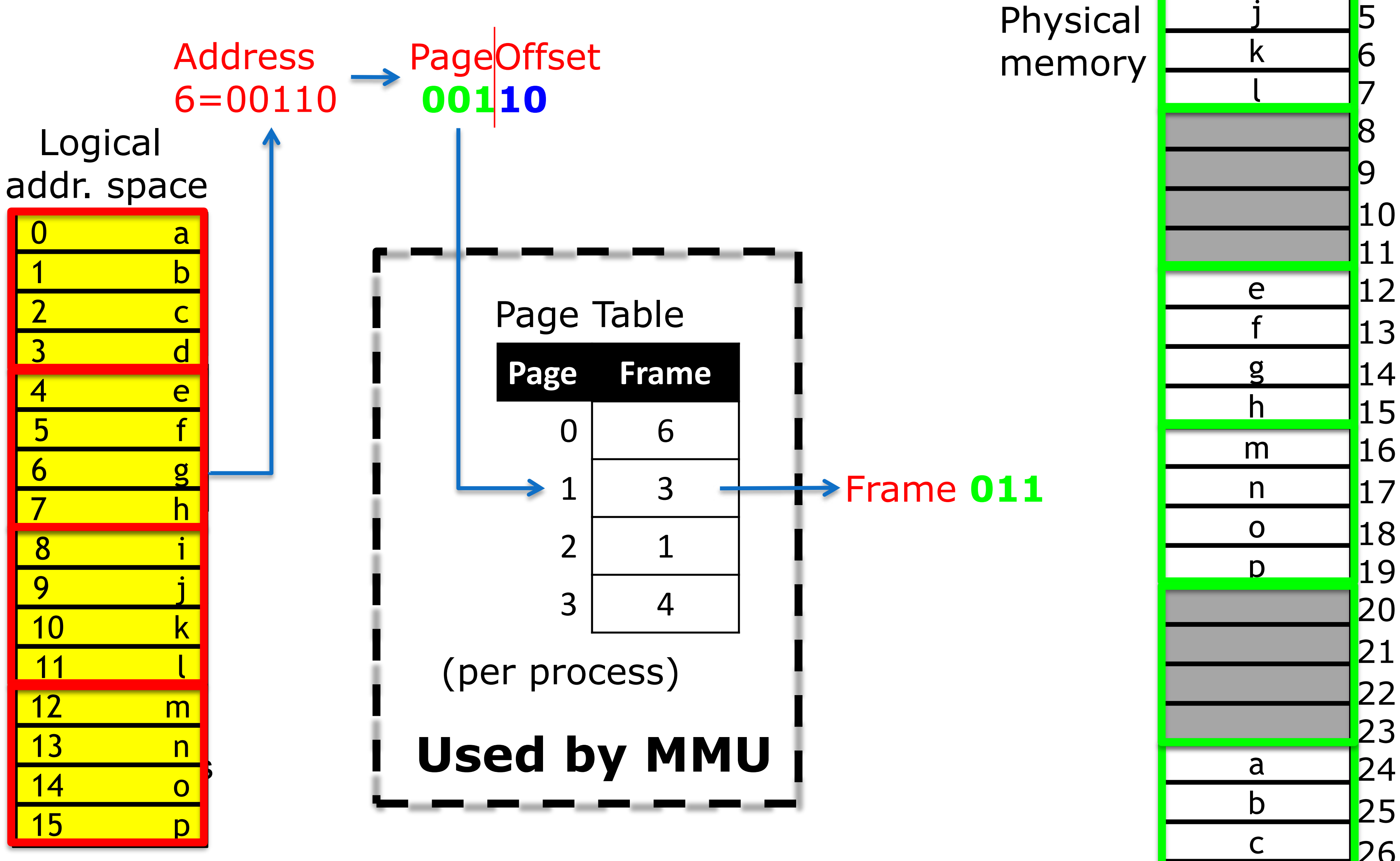
Address Translation with the Page Table



Page	Frame
0	6
1	3
2	1
3	4



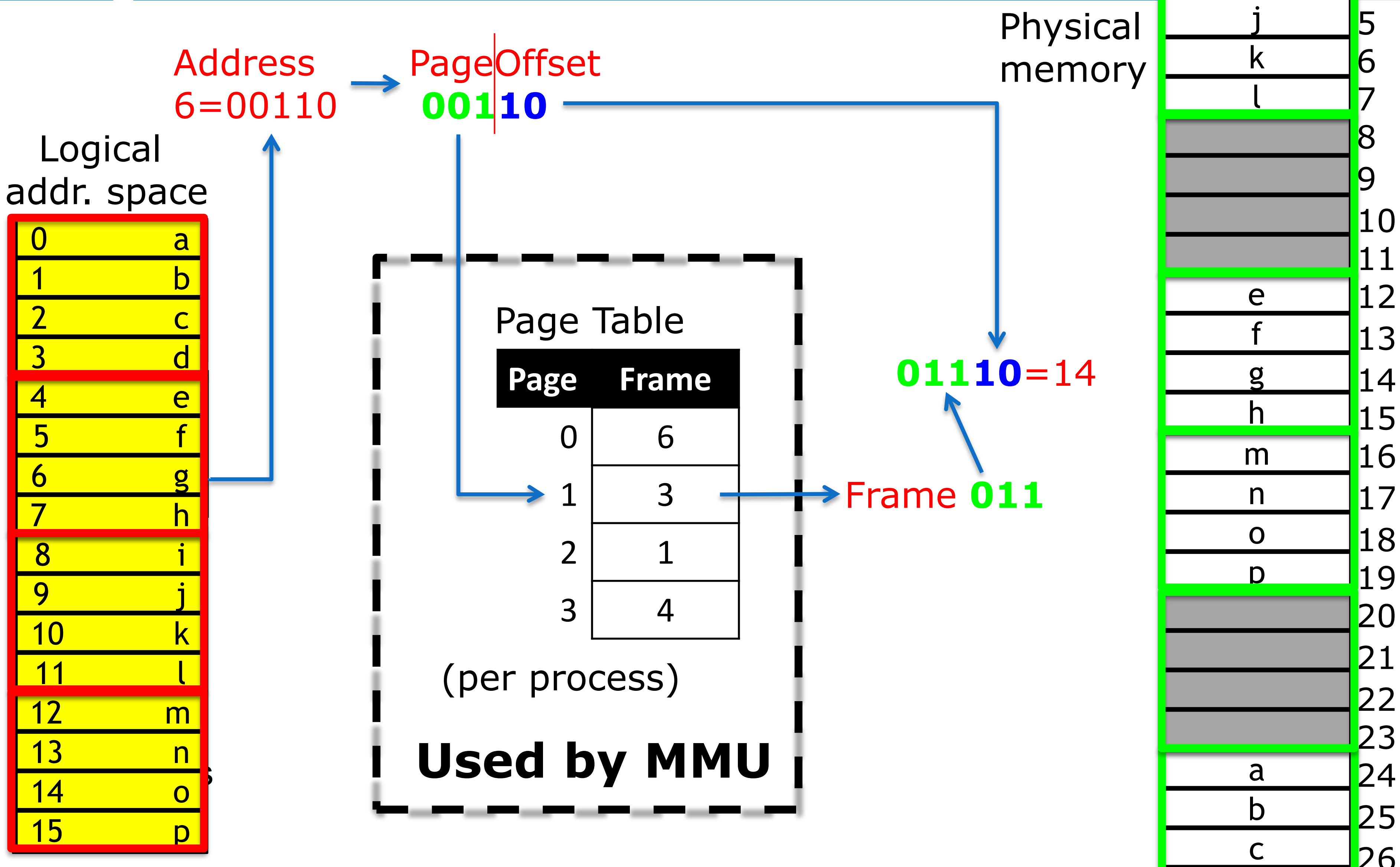
Address Translation with the Page Table



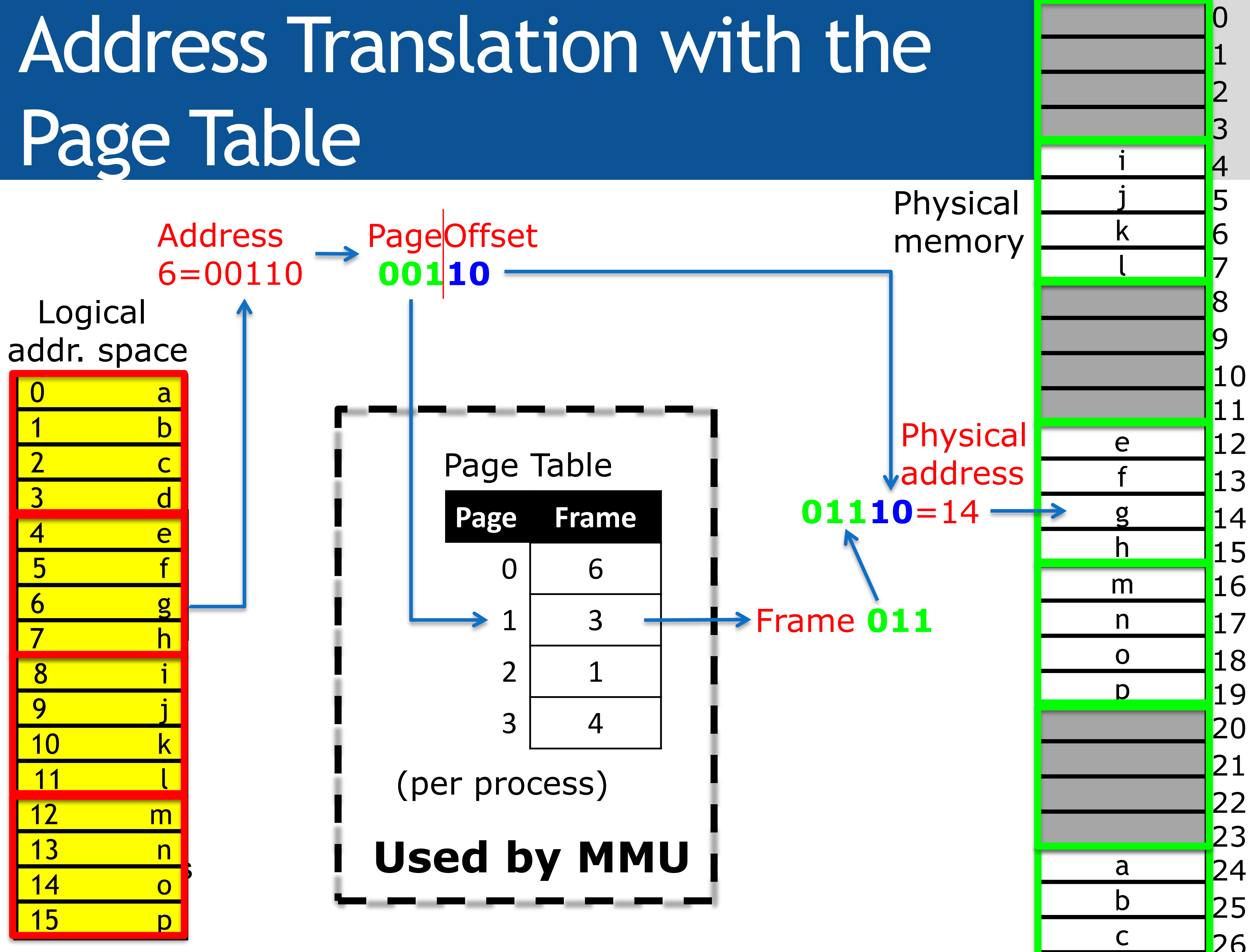
Page Table

Page	Frame
0	6
1	3
2	1
3	4

Address Translation with the Page Table



Address Translation with the Page Table



Overheads

- Page tables take up a lot of space
  – Need to be stored in memory
  – Improvement: optimize page size
  – Use sophisticated page table structures [later]
- Accessing the page table takes an additional memory access!
  – This is slow
  – Solution: cache recently used translations in the TLB

Optimal page size

- The page size tradeoff:
  - Small pages have less internal fragmentation
  - Small pages require larger page tables

Optimal page size

- The page size tradeoff:
  - Small pages have less internal fragmentation
  - Small pages require larger page tables
- p = page size
  s = size of the process
  e = size of the entry in the page table
  → **Overhead = (se/p)+p/2**

Optimal page size

- The page size tradeoff:
  - Small pages have less internal fragmentation
  - Small pages require larger page tables
- p = page size
  s = size of the process
  e = size of the entry in the page table
  → **Overhead = (se/p)+p/2**

Page table size: s/p pages → s/p entries, each of size e

Optimal page size

- The page size tradeoff:
  - Small pages have less internal fragmentation
  - Small pages require larger page tables
- p = page size
  s = size of the process
  e = size of the entry in the page table
  → **Overhead = (se/p)+p/2**

Page table size: s/p pages → s/p entries, each of size e

Internal fragmentation

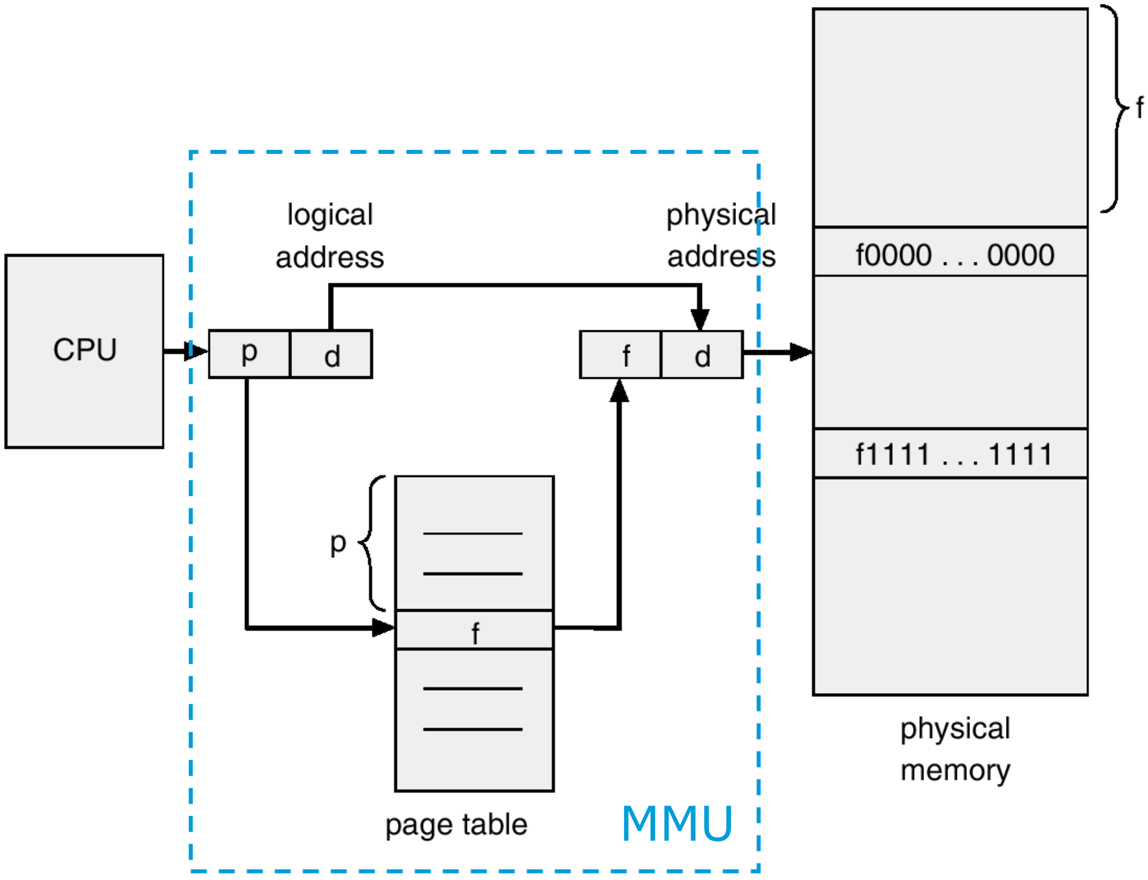
Optimal page size

- The page size tradeoff:
  - Small pages have less internal fragmentation
  - Small pages require larger page tables
- p = page size
  s = size of the process
  e = size of the entry in the page table
  → **Overhead = (se/p)+p/2**
  → **Optimal page size is √(2se)**

Optimal page size

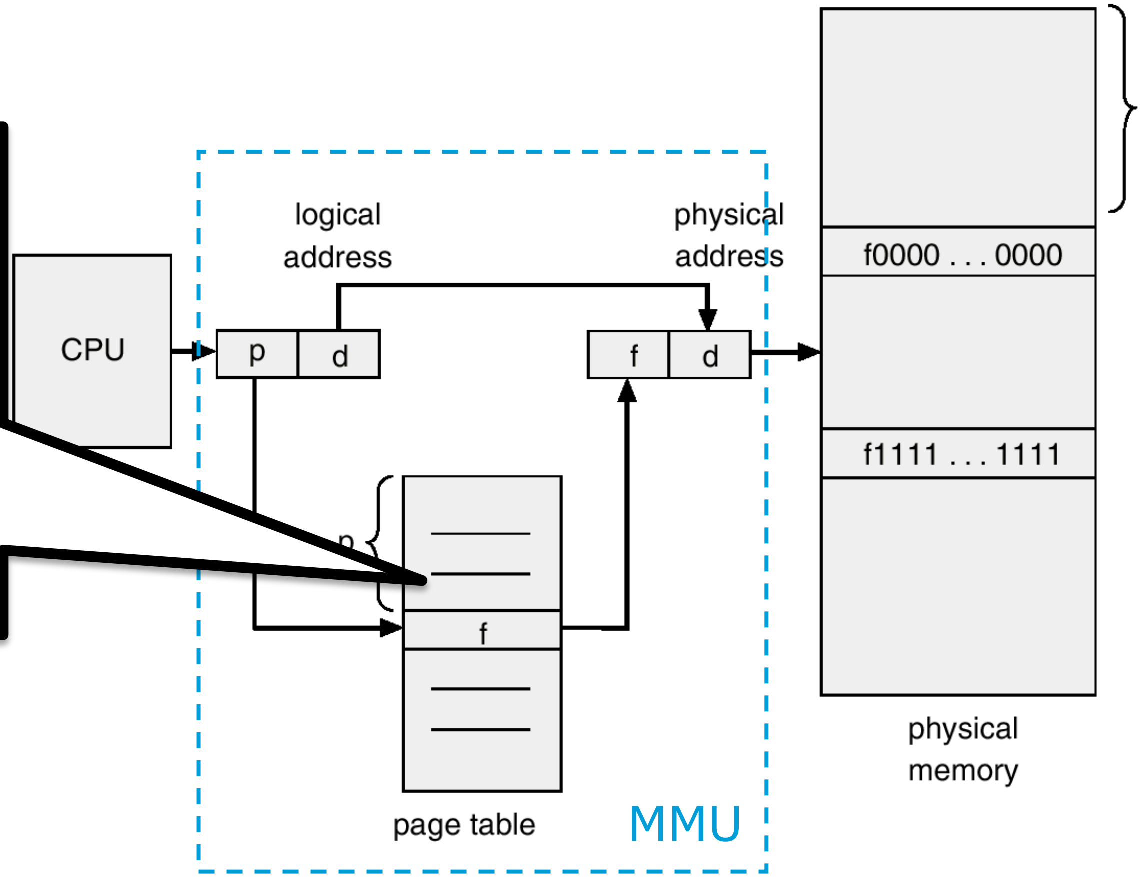
- The page size tradeoff:
  - Small pages have less internal fragmentation
  - Small pages require larger page tables
- p = page size
  s = size of the process
  e = size of the entry in the page table
  → Overhead = (se/p)+p/2
  → Optimal page size is √(2se)
- For s=1MB, e=64 bit → p=4KB

Address Translation Architecture



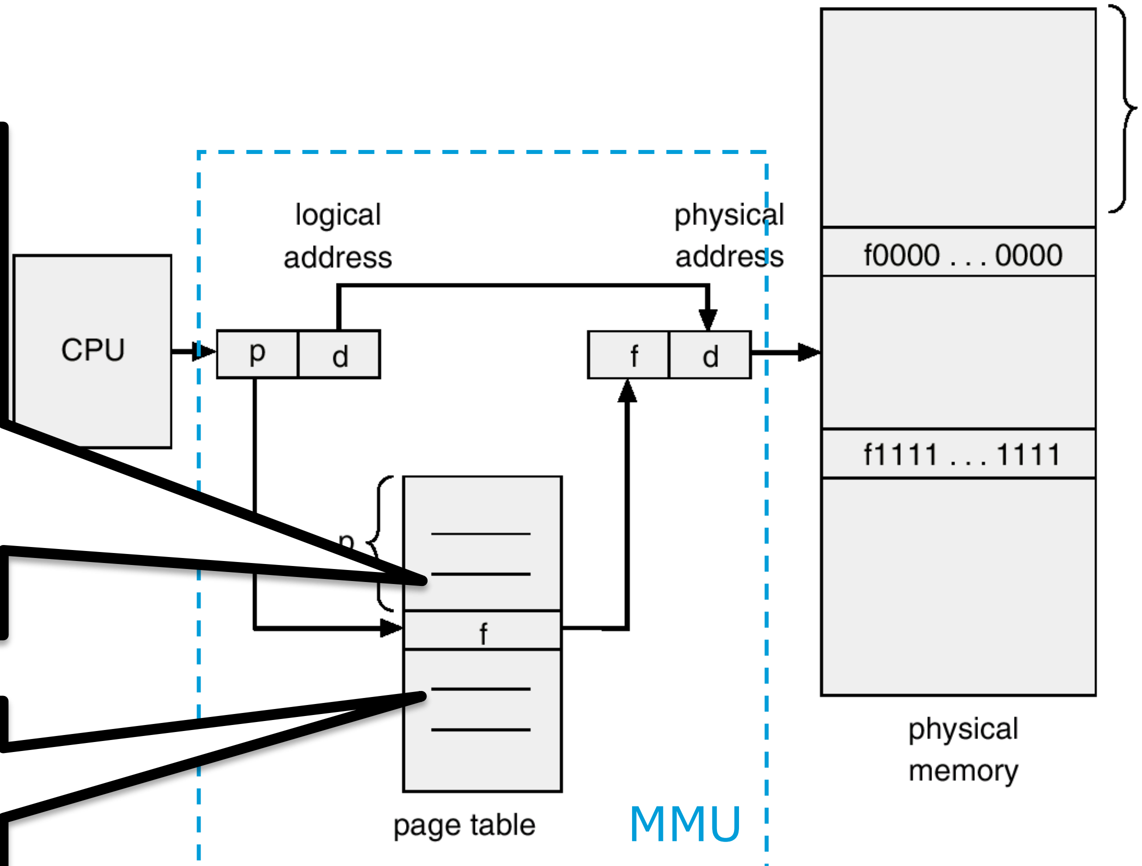
Address Translation Architecture

Page table size:
se/p
1MB process →
2KB *per process*
Need to stored it
in memory!



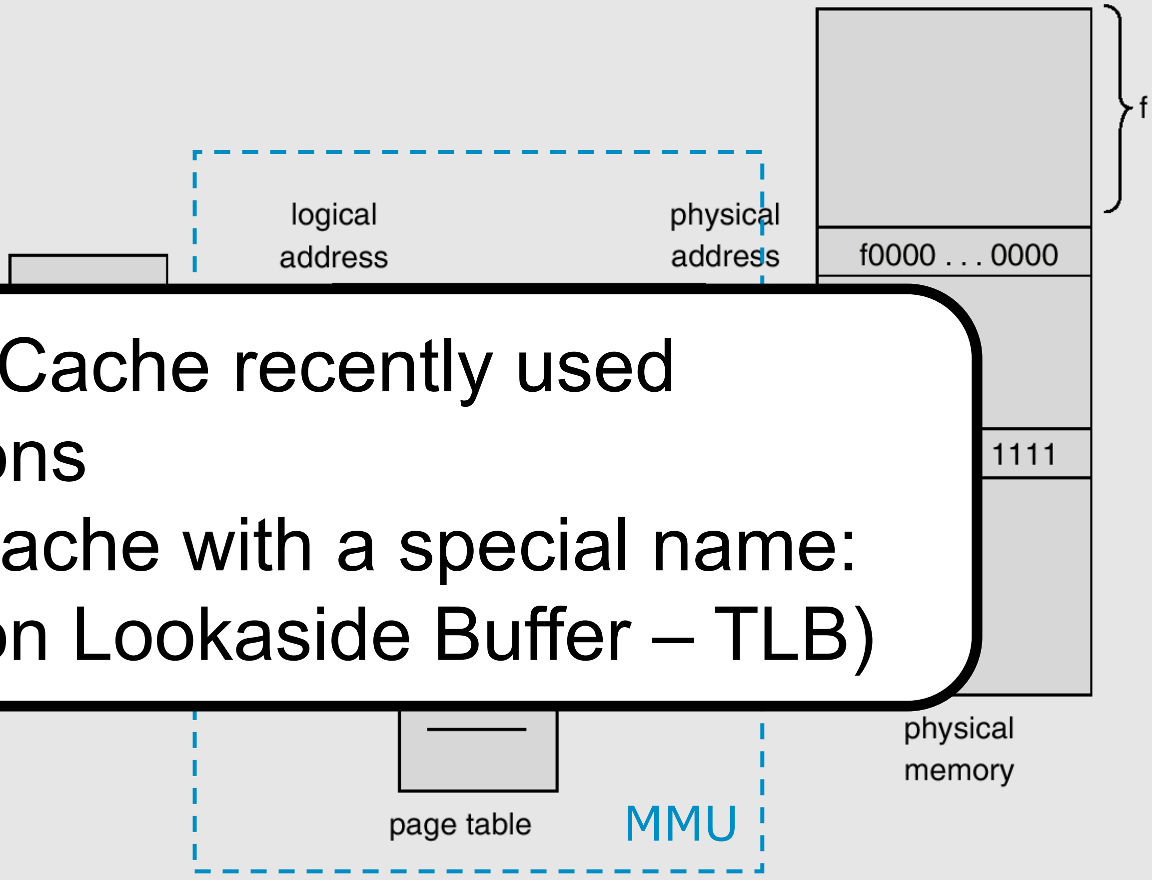
Address Translation Architecture

Page table size:
se/p
1MB process →
2KB *per process*
Need to stored it
in memory!



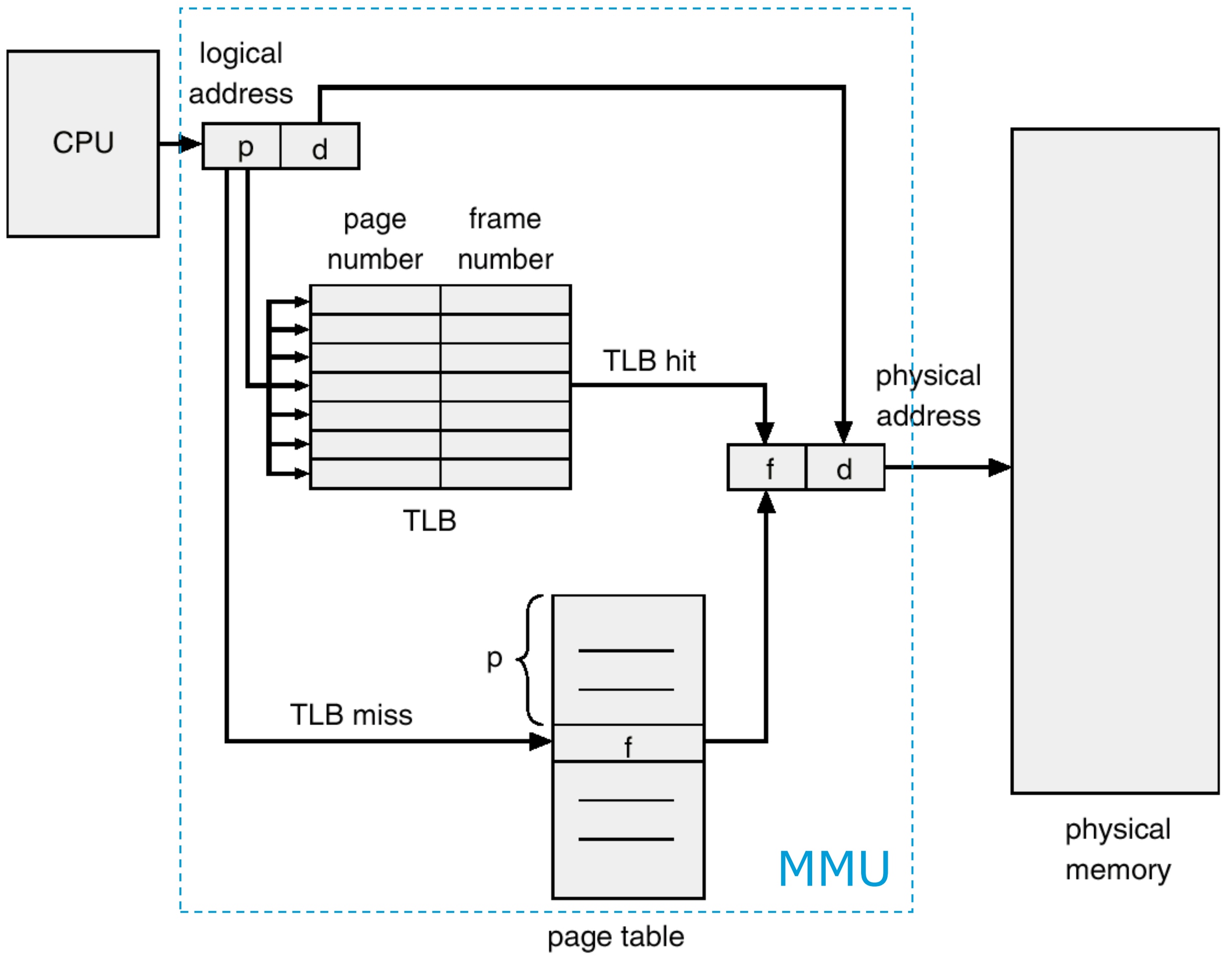
Extra memory access for each memory access!!

Address Translation Architecture

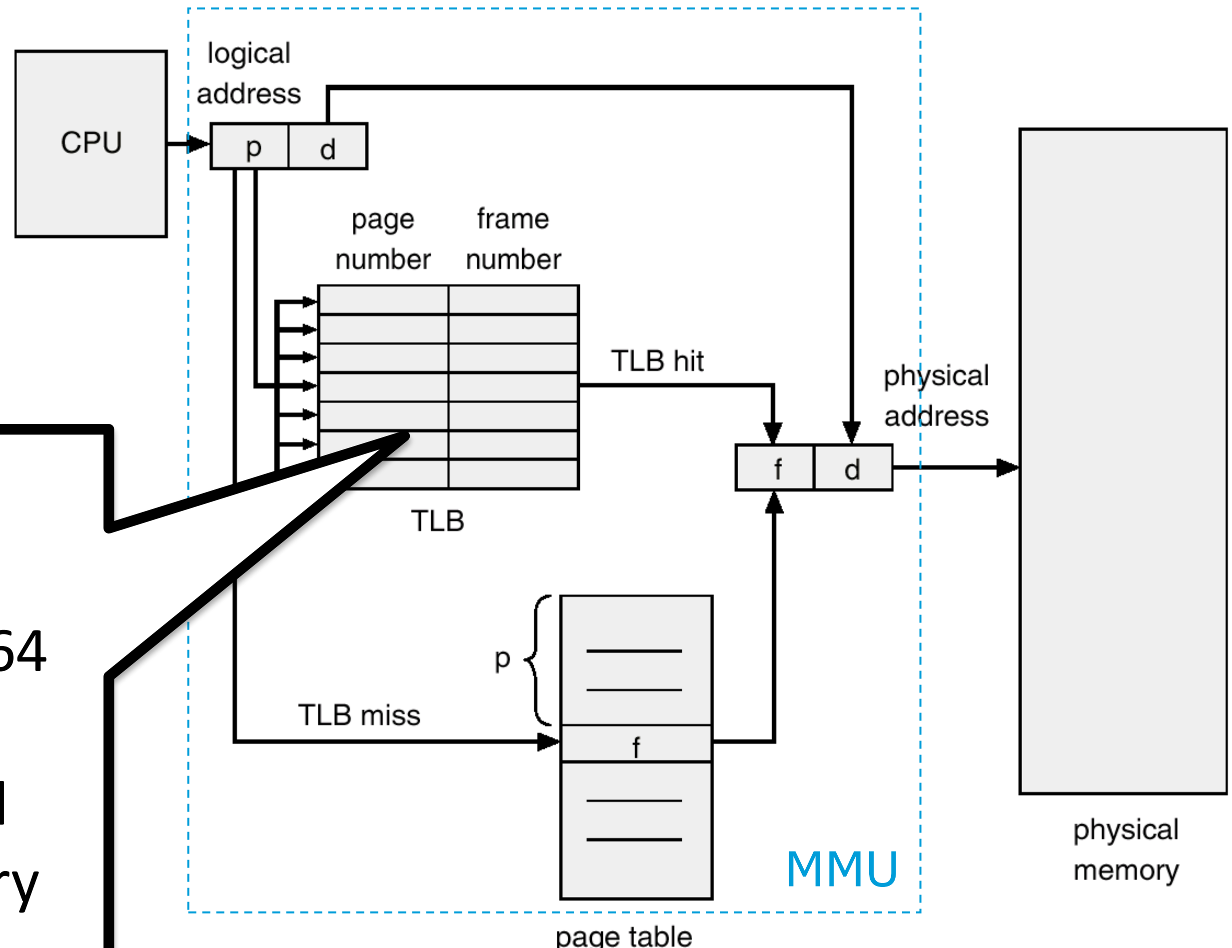


Solution: Cache recently used Translations
(special cache with a special name: Translation Lookaside Buffer – TLB)

Address Translation with a TLB



Address Translation with a TLB



- Usually fully associative
- Small, about 64 entries
- SRAM/Special hardware (very fast)

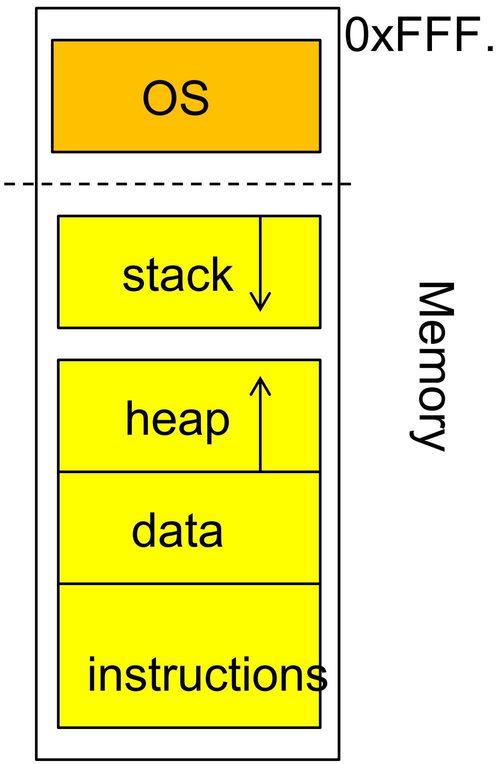
VIRTUAL MEMORY

Logical vs. Physical Memory

- Which is larger?

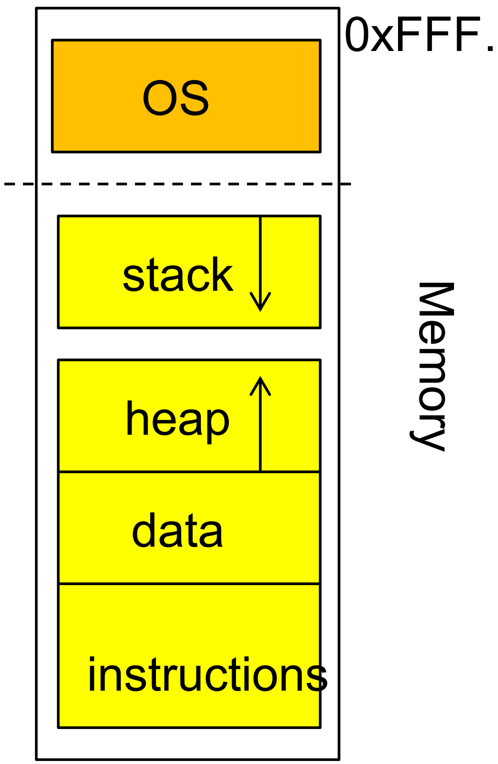
Logical vs. Physical Memory

- Which is larger?
- **Each process** thinks it runs alone
  - May access the entire physical memory



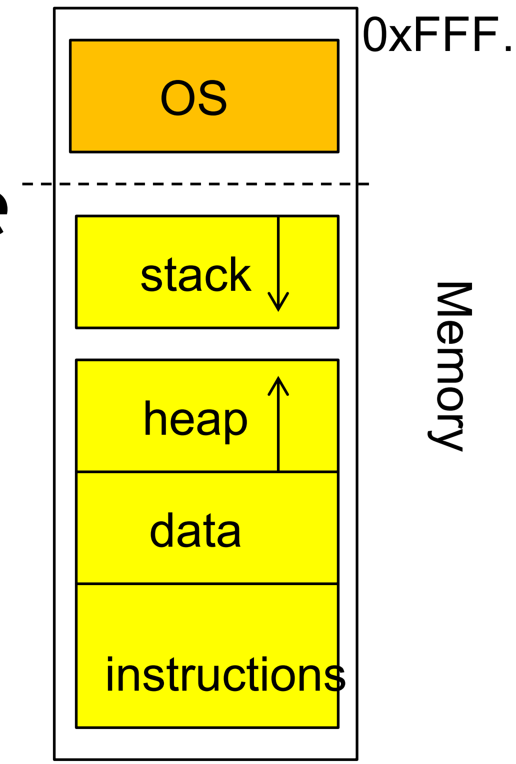
Logical vs. Physical Memory

- Which is larger?
- **Each process** thinks it runs alone
  - May access the entire physical memory
- The number of processes, in general, is not limited



Logical vs. Physical Memory

- Which is larger?
- **Each process** thinks it runs alone
  - May access the entire physical memory
- The number of processes, in general, is not limited



→ The required logical memory (sum of logical memory of all processes) is much larger than the actual physical memory

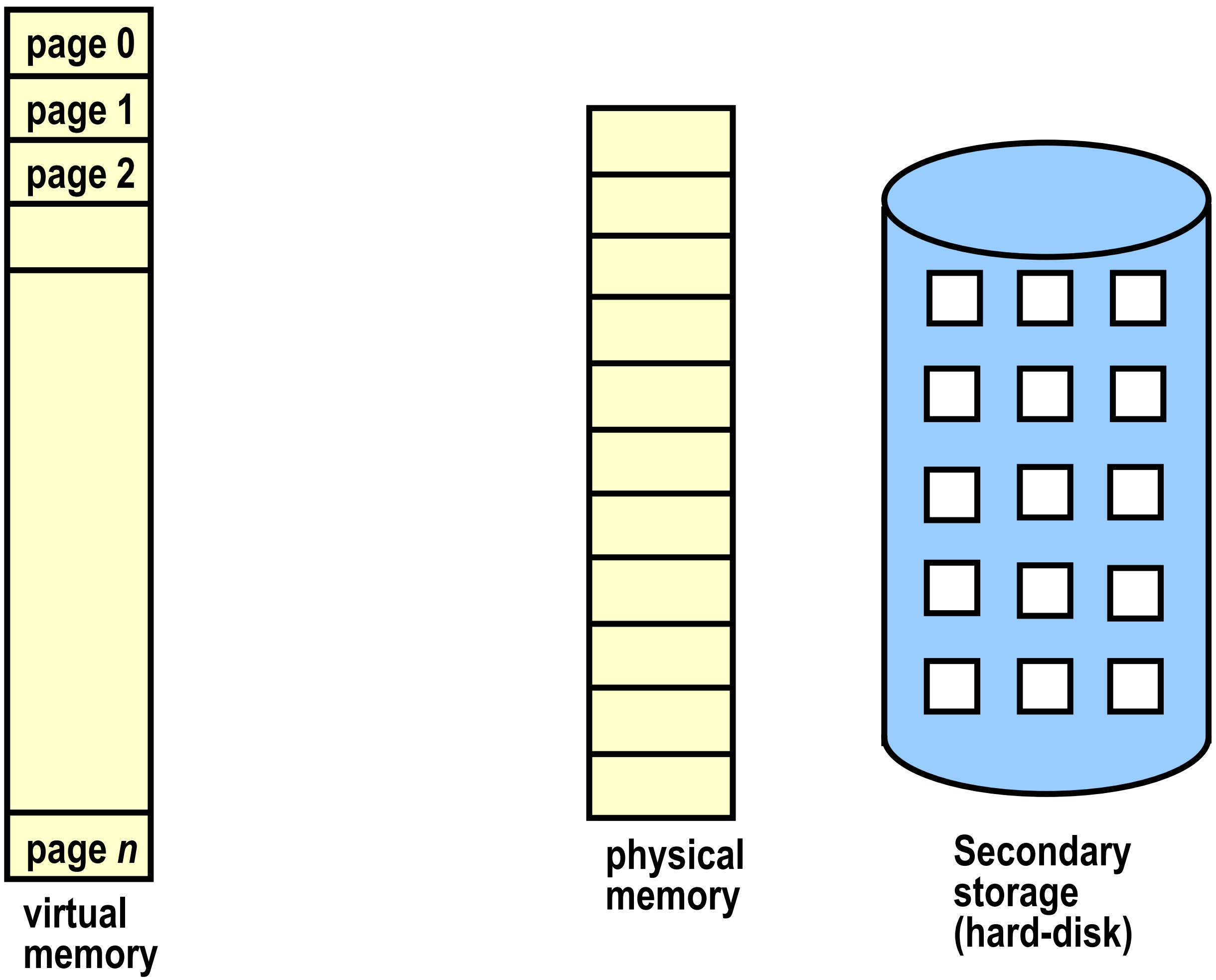
Important Observation

- Programs don't use all their address space all the time
  - Program phases: don't need initialization code after you finish it
  - May not need error handling code ever
  - Use one data structure then another
- Unused parts don't need to be mapped to memory
  - Can be stored on disk until needed
  - Reduces memory pressure
  - Allows more efficient process creation

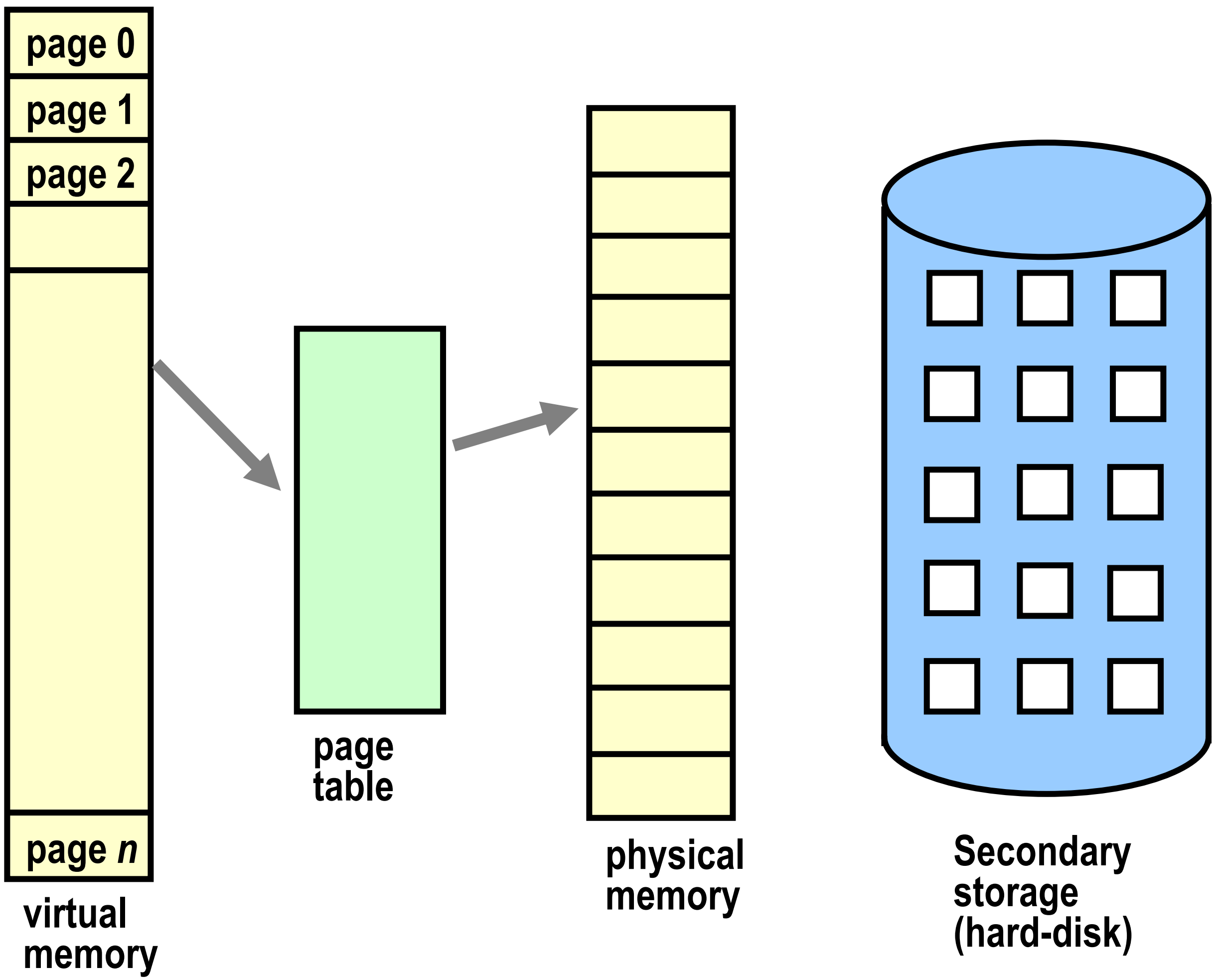
Virtual Memory

- The idea: VIRTUALIZATION
  - Disconnect from the limitations of our physical budget
  - Make it look as if we have all the memory we want
- The implementation: DEMAND PAGING
  - Bring pages to memory when we need them
  - Store them on disk when we don't

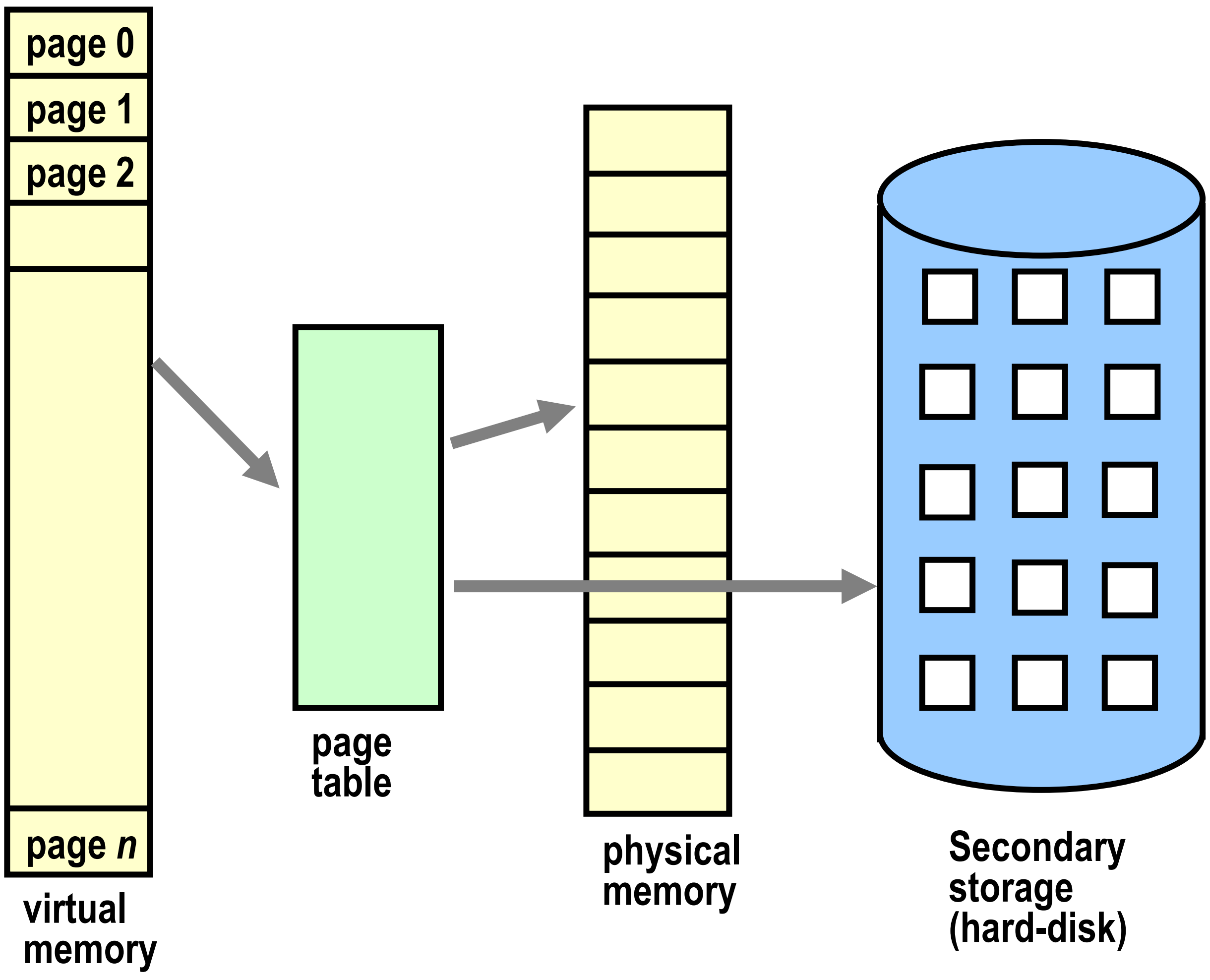
Dynamics of Demand Paging



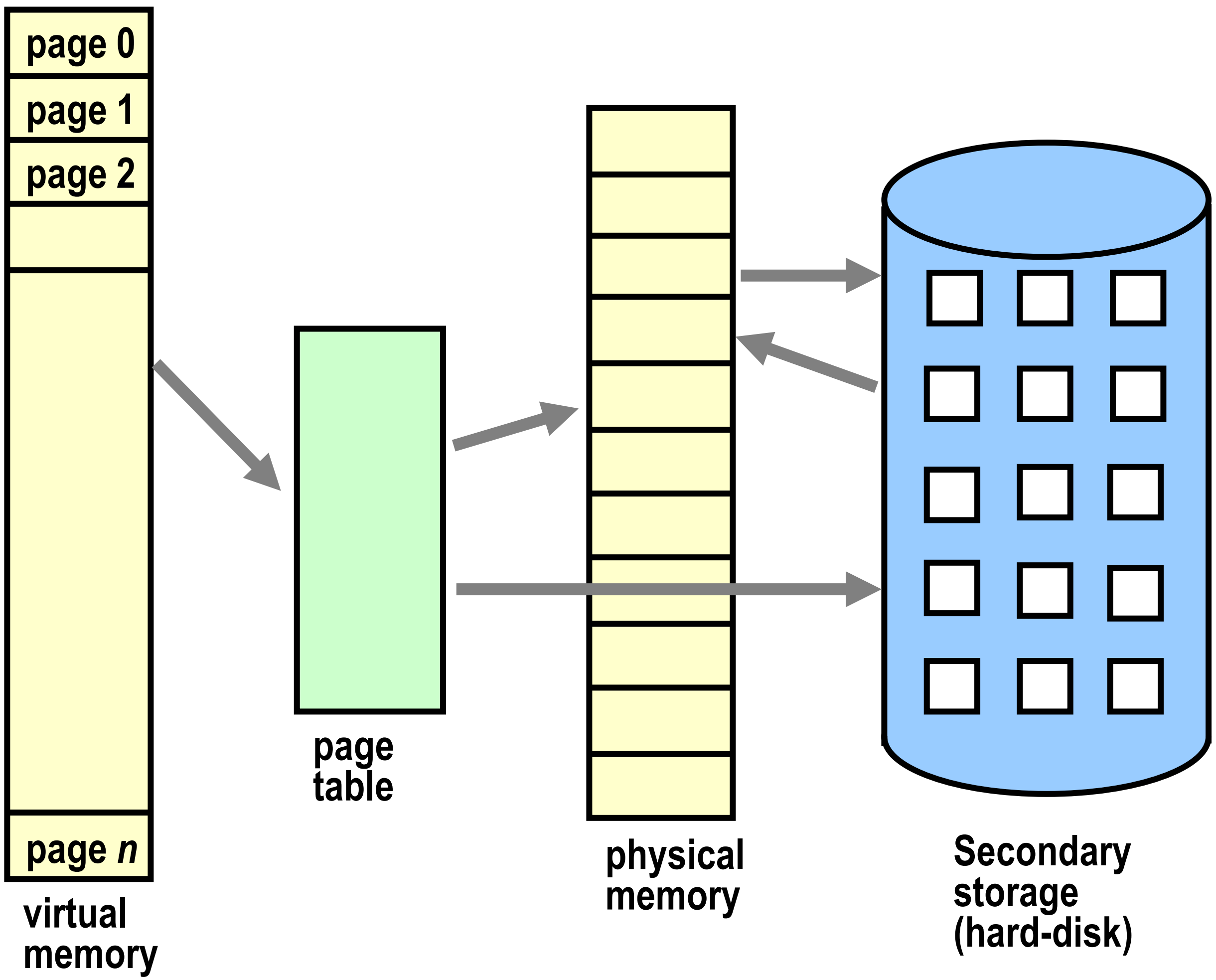
Dynamics of Demand Paging



Dynamics of Demand Paging



Dynamics of Demand Paging



Add Bit to Page Table

Virtual memory

a	0
b	1
c	2
d	3
e	4
f	5
g	6
h	7
i	8
j	9
k	10
l	11
m	12
n	13
o	14
p	15
q	16
r	17
s	18
t	19
u	20
v	21
w	22
x	23
y	24
z	25
aa	26

Page Table

Page	Frame	V
0	2	1
1	3	1
2	X	0
3	X	0
4	1	1
5	0	1
6	X	0

(per process)

Used by MMU

0	u
1	v
2	w
3	x
4	q
5	r
6	s
7	t
8	a
9	b
10	c
11	d
12	e
13	f
14	g
15	h

Physical memory

Add Bit to Page Table

Virtual memory

a	0
b	1
c	2
d	3
e	4
f	5
g	6
h	7
i	8
j	9
k	10
l	11
m	12
n	13
o	
p	
q	
r	
s	
t	
u	
v	
w	
x	
y	
z	25
aa	26

Page Table

Page	Frame	V
0	2	1
1	3	1
2	X	0
	X	0
4	1	1
5	0	1
6	X	0

(per process)

Used by MMU

Valid/invalid bit:

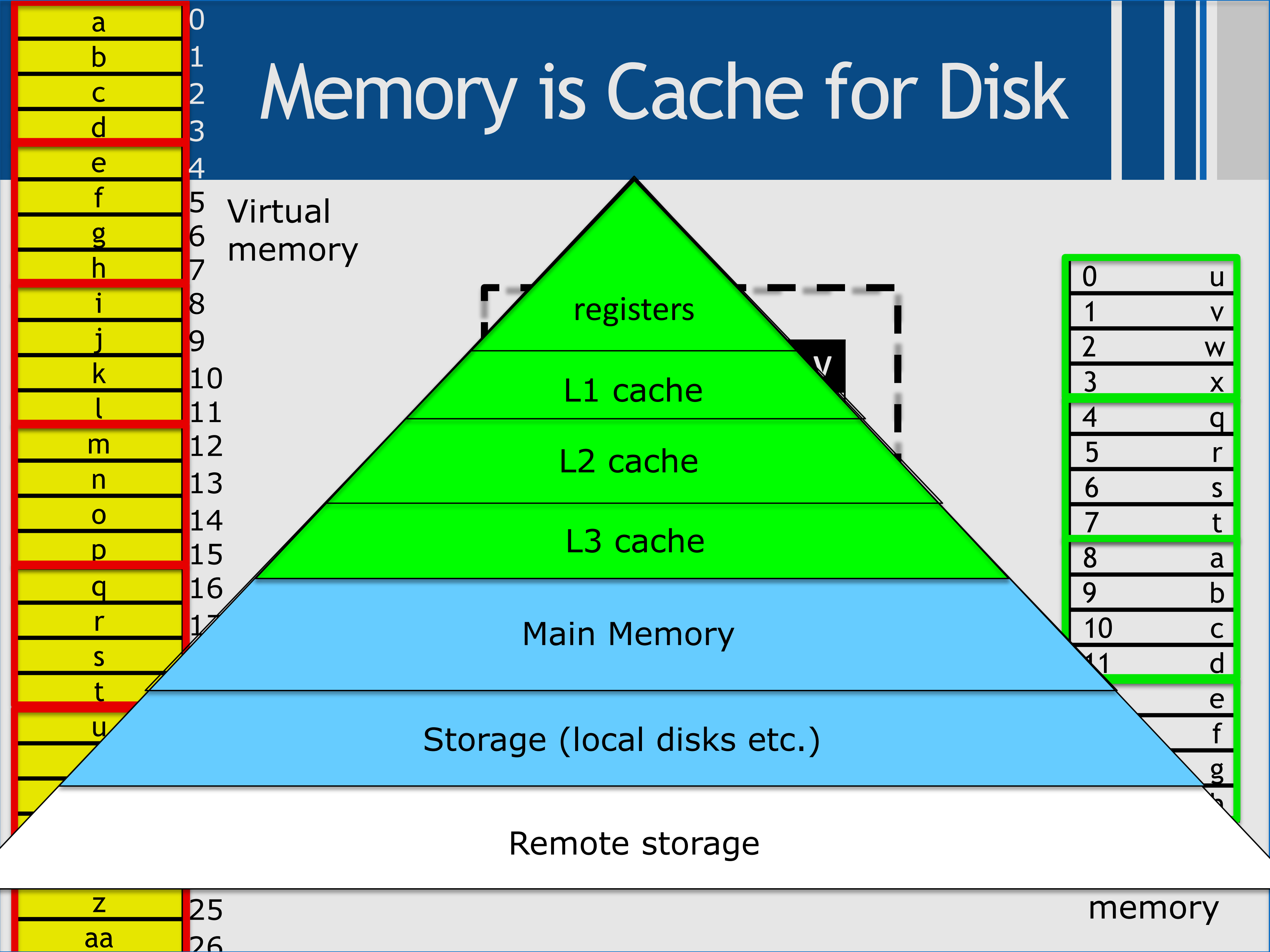
1: Page in memory

0: Page not in memory (the value in frame column is invalid)

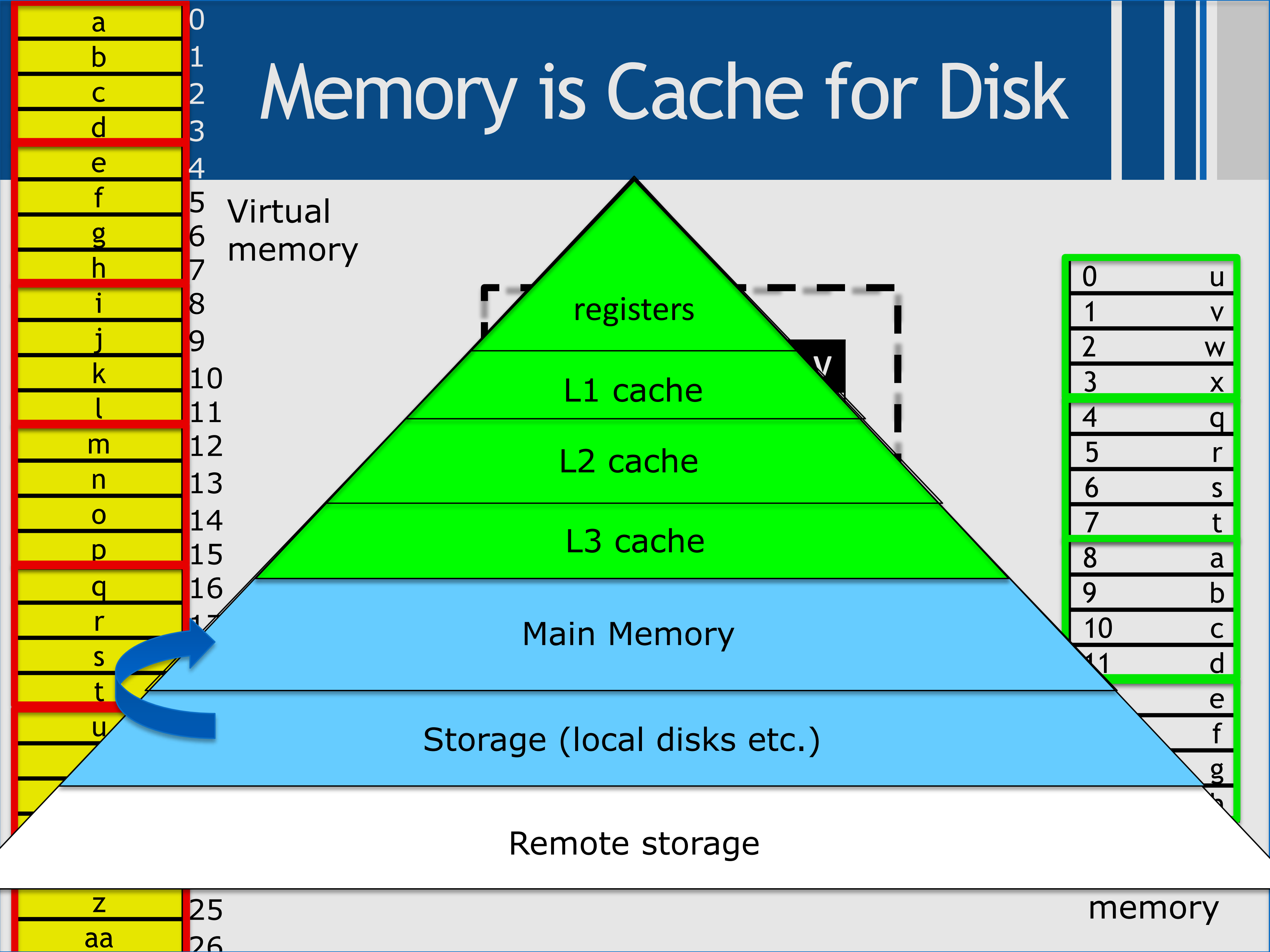
0	u
1	v
2	w
3	x
4	q
5	r
6	s
7	t
8	a
9	b
10	c
11	d
12	e
13	f
14	g
15	h

Physical memory

Memory is Cache for Disk



Memory is Cache for Disk



Demand Paging

- OS loads a page into memory only when it is needed
  - Less I/O needed
  - Less memory needed
  - Faster response
  - More users

Demand Paging

- OS loads a page into memory only when it is needed
  - Less I/O needed
  - Less memory needed
  - Faster response
  - More users
- **Another option:** Pre-paging
  - OS guesses in advance which pages the process will need and pre-loads them into memory
  - Save time if the OS guesses correctly
  - More overhead if the OS is wrong

What happens when the accessed page is not in memory?

- CPU issues a virtual address that is in an <u>unmapped</u> page

What happens when the accessed page is not in memory?

- CPU issues a virtual address that is in an <u>unmapped</u> page
- The data cannot be accessed!

What happens when the accessed page is not in memory?

- CPU issues a virtual address that is in an unmapped page
- The data cannot be accessed!
- MMU creates a PAGE FAULT exception

What happens when the accessed page is not in memory?

- CPU issues a virtual address that is in an <u>unmapped</u> page
- The data cannot be accessed!
- MMU creates a PAGE FAULT exception
- OS exception handler is invoked

What happens when the accessed page is not in memory?

- CPU issues a virtual address that is in an <u>unmapped</u> page
- The data cannot be accessed!
- MMU creates a PAGE FAULT exception
- OS exception handler is invoked
- Initiate disk operation to get required data

What happens when the accessed page is not in memory?

- CPU issues a virtual address that is in an unmapped page
- The data cannot be accessed!
- MMU creates a PAGE FAULT exception
- OS exception handler is invoked
- Initiate disk operation to get required data
- Process put to sleep until it arrives

What happens when the accessed page is not in memory?

- CPU issues a virtual address that is in an <u>unmapped</u> page
- The data cannot be accessed!
- MMU creates a PAGE FAULT exception
- OS exception handler is invoked
- Initiate disk operation to get required data
- Process put to sleep until it arrives
- Run other processes in this time

What happens when the accessed page is not in memory?

- CPU issues a virtual address that is in an <u>unmapped</u> page
- The data cannot be accessed!
- MMU creates a PAGE FAULT exception
- OS exception handler is invoked
- Initiate disk operation to get required data
- Process put to sleep until it arrives
- Run other processes in this time
- When data arrives, awaken process and re-issue the same instruction

Handling Page Faults

a	0
b	1
c	2
d	3
e	4
f	5
g	6
h	7
i	8
j	9
k	10
l	11
m	12
n	13
o	14
p	15
q	16
r	17
s	18
t	19
u	20
v	21
w	22
x	23
y	24
z	25
aa	26

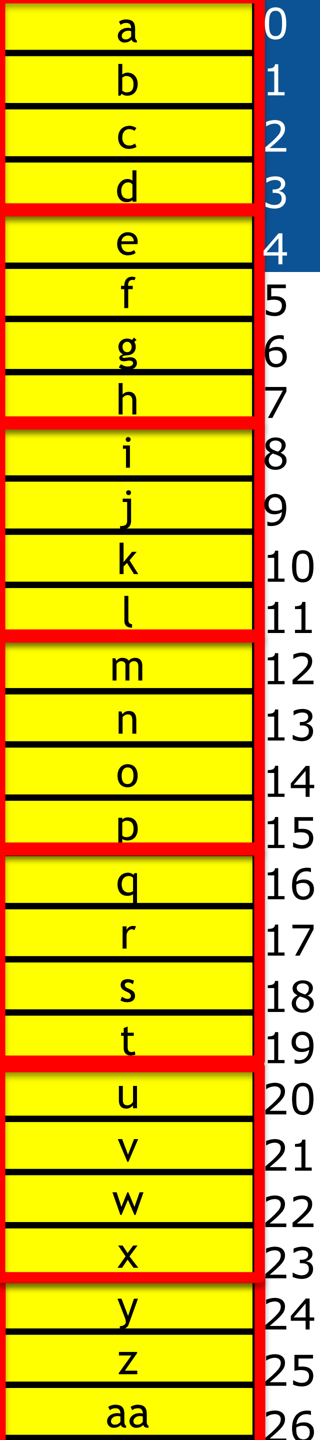
Handling Page Faults

a	0
b	1
c	2
d	3
e	4
f	5
g	6
h	7
i	8
j	9
k	10
l	11
m	12
n	13
o	14
p	15
q	16
r	17
s	18
t	19
u	20
v	21
w	22
x	23
y	24
z	25
aa	26

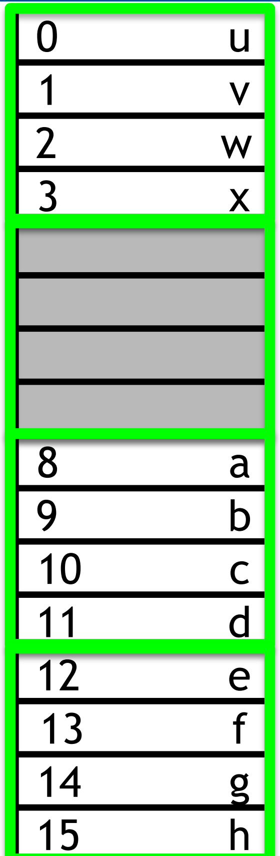
Page	Frame	
0	2	1
1	3	1
2	2	0
3	3	0
4	1	0
5	0	1
6	0	0

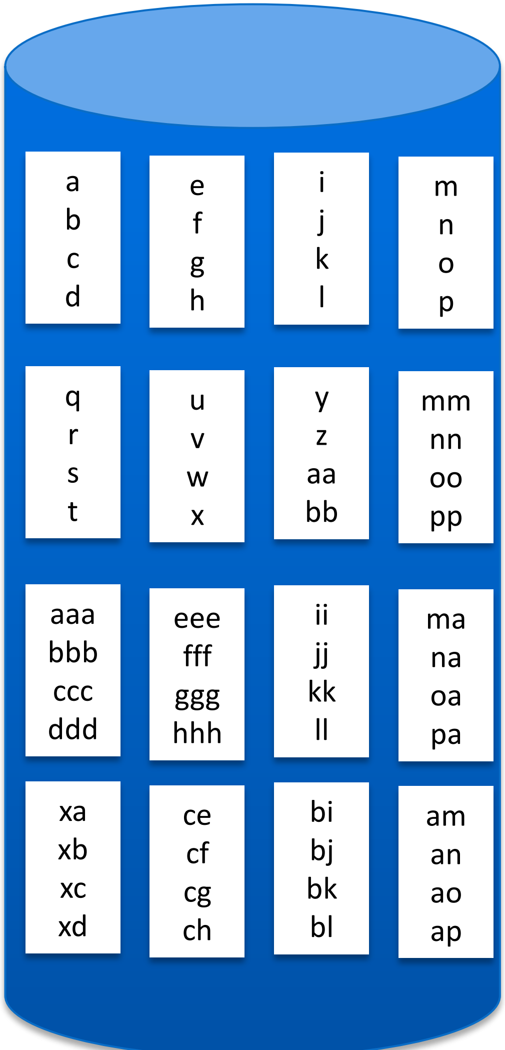
0	u
1	v
2	w
3	x
8	a
9	b
10	c
11	d
12	e
13	f
14	g
15	h

Handling Page Faults

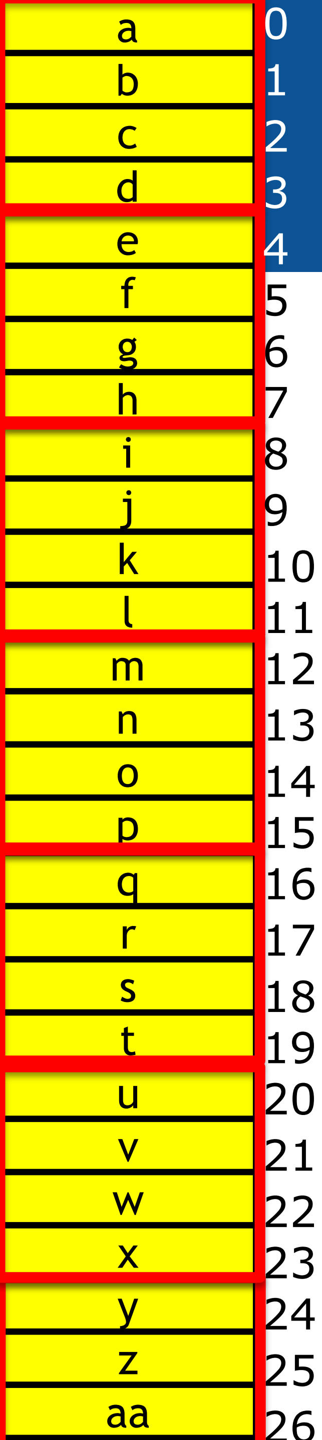


Page	Frame	
0	2	1
1	3	1
2	2	0
3	3	0
4	1	0
5	0	1
6	0	0

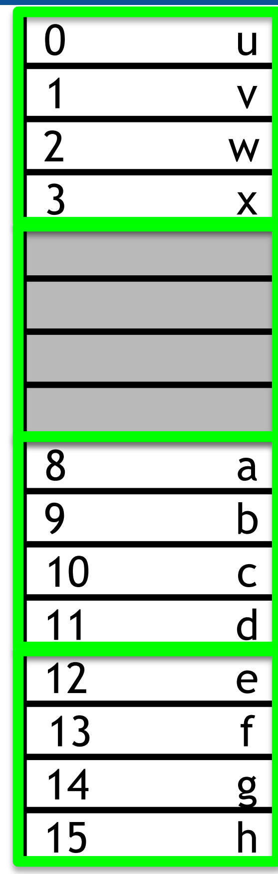


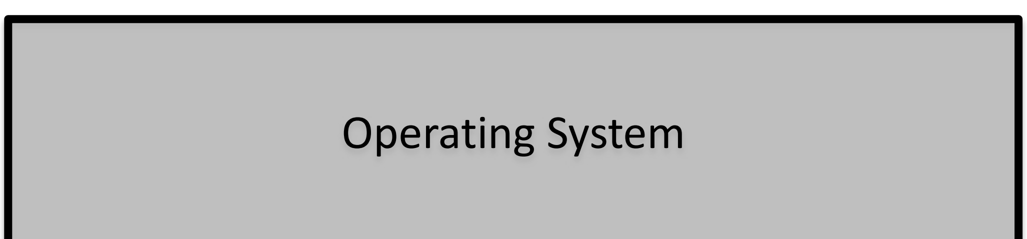


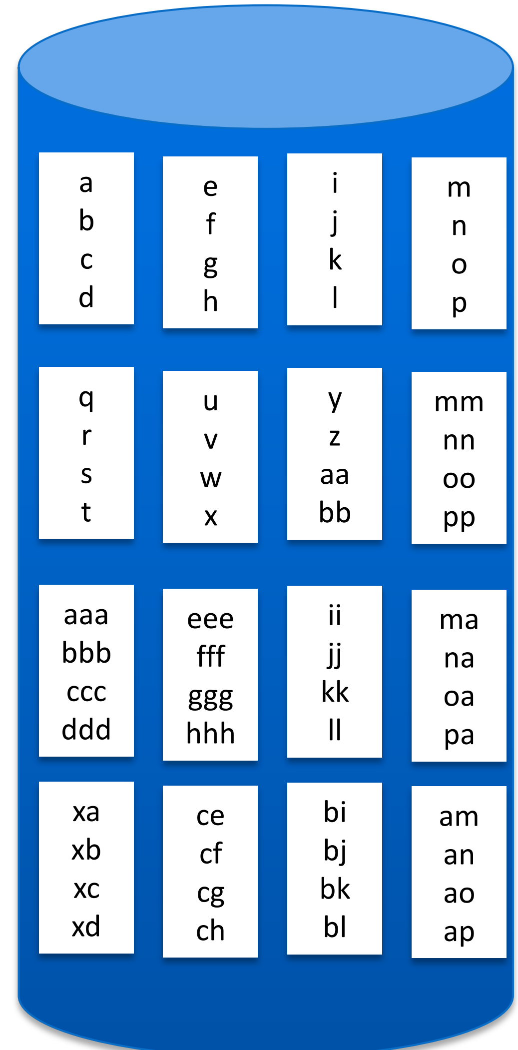
Handling Page Faults



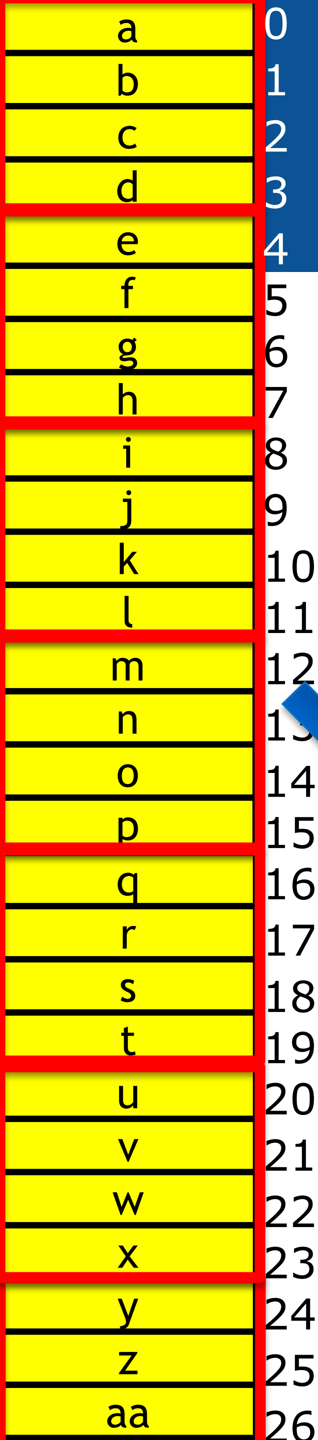
Page	Frame	
0	2	1
1	3	1
2	2	0
3	3	0
4	1	0
5	0	1
6	0	0





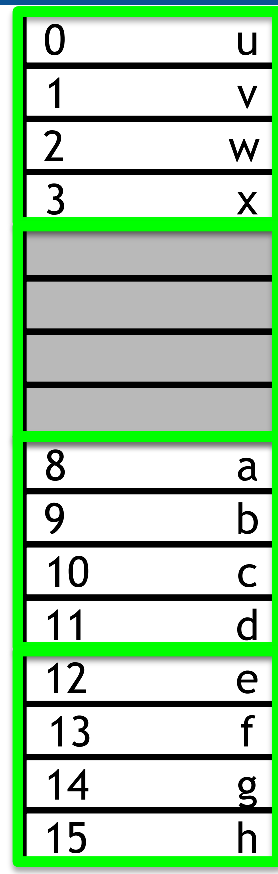


Handling Page Faults

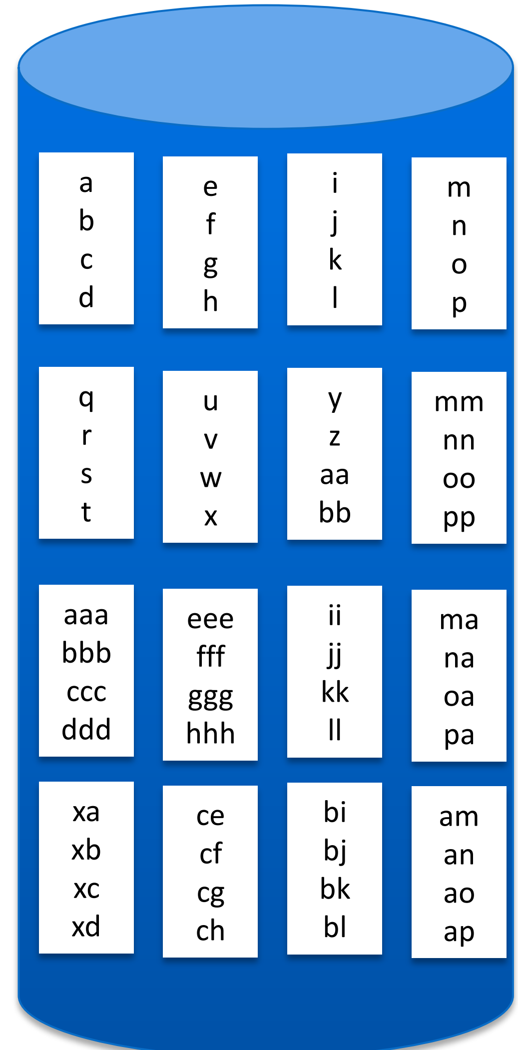


lw 12, R1

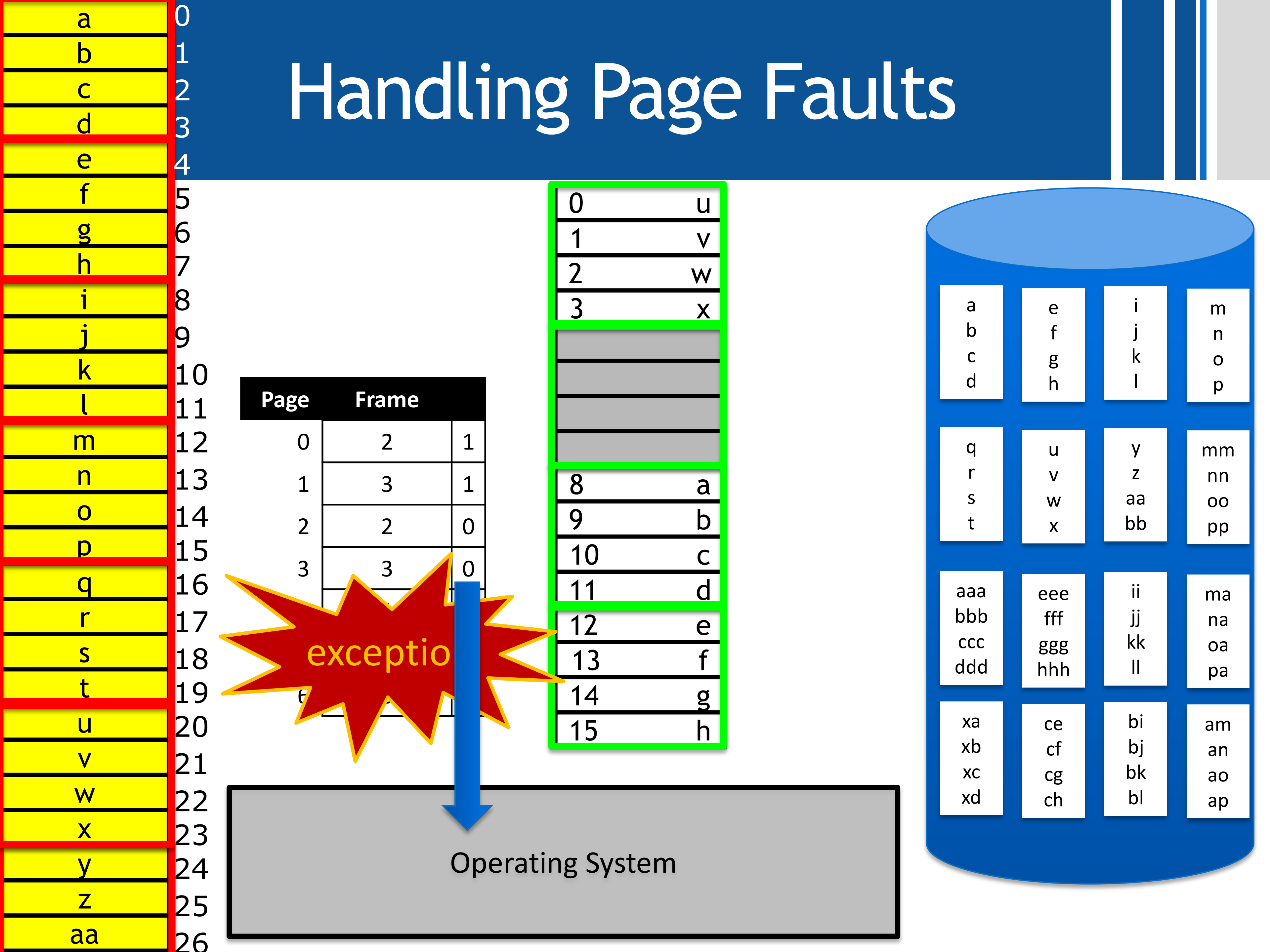
Page	Frame	
0	2	1
1	3	1
2	2	0
3	3	0
4	1	0
5	0	1
6	0	0



Operating System

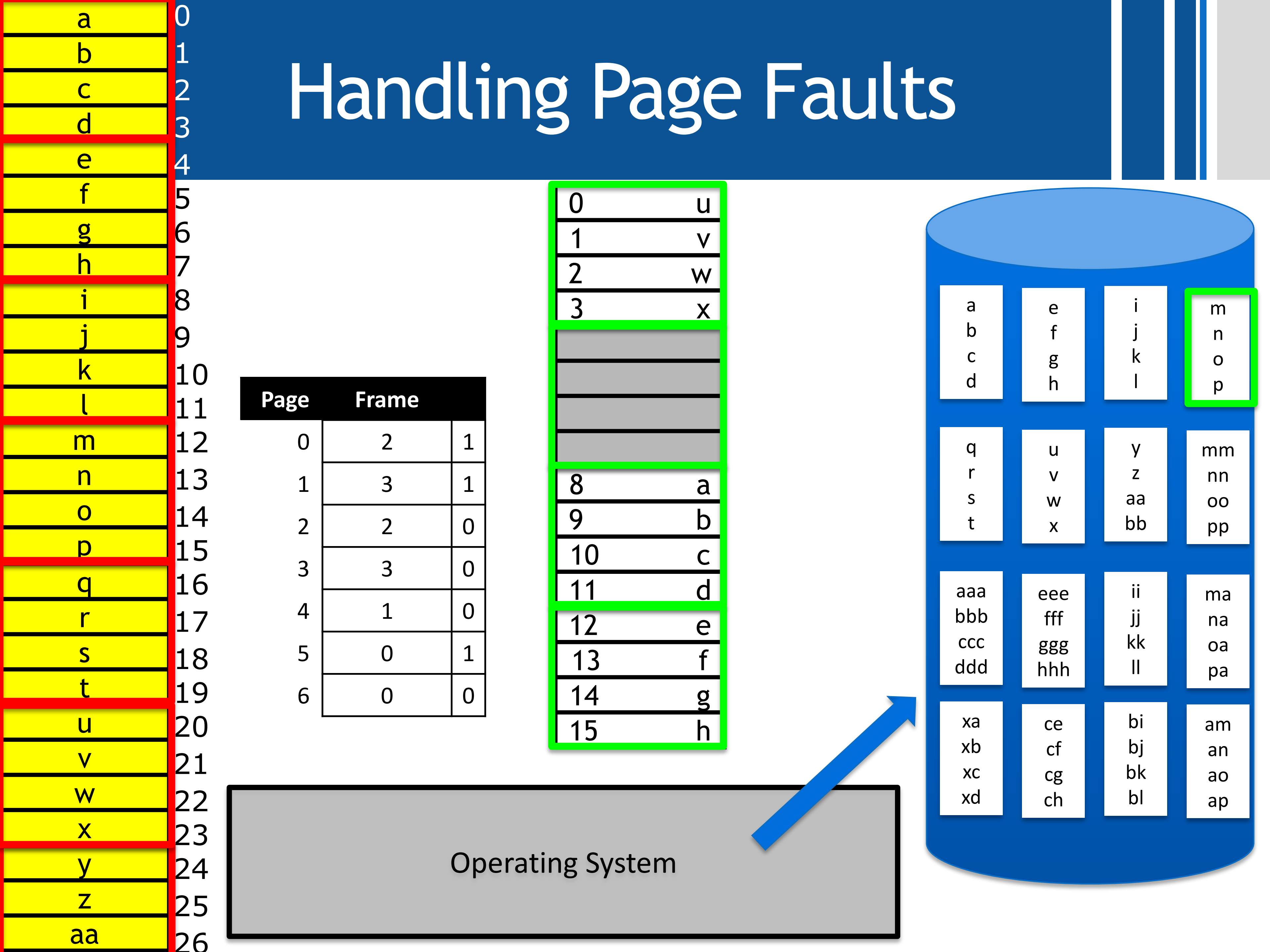


Handling Page Faults

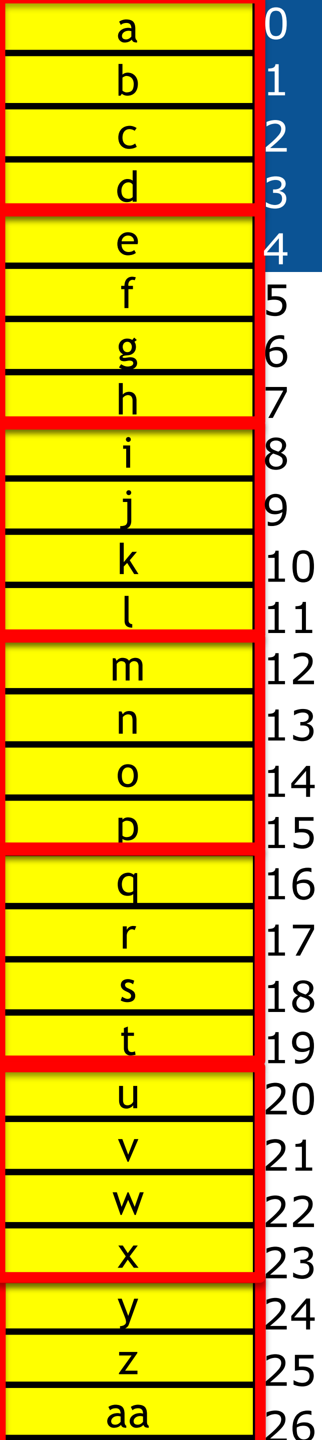


Handling Page Faults

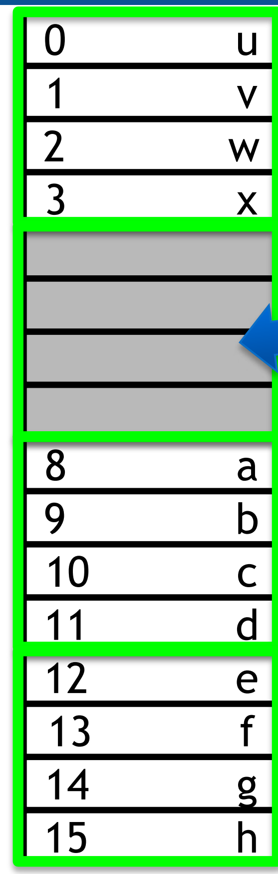
Page	Frame	
0	2	1
1	3	1
2	2	0
3	3	0
4	1	0
5	0	1
6	0	0

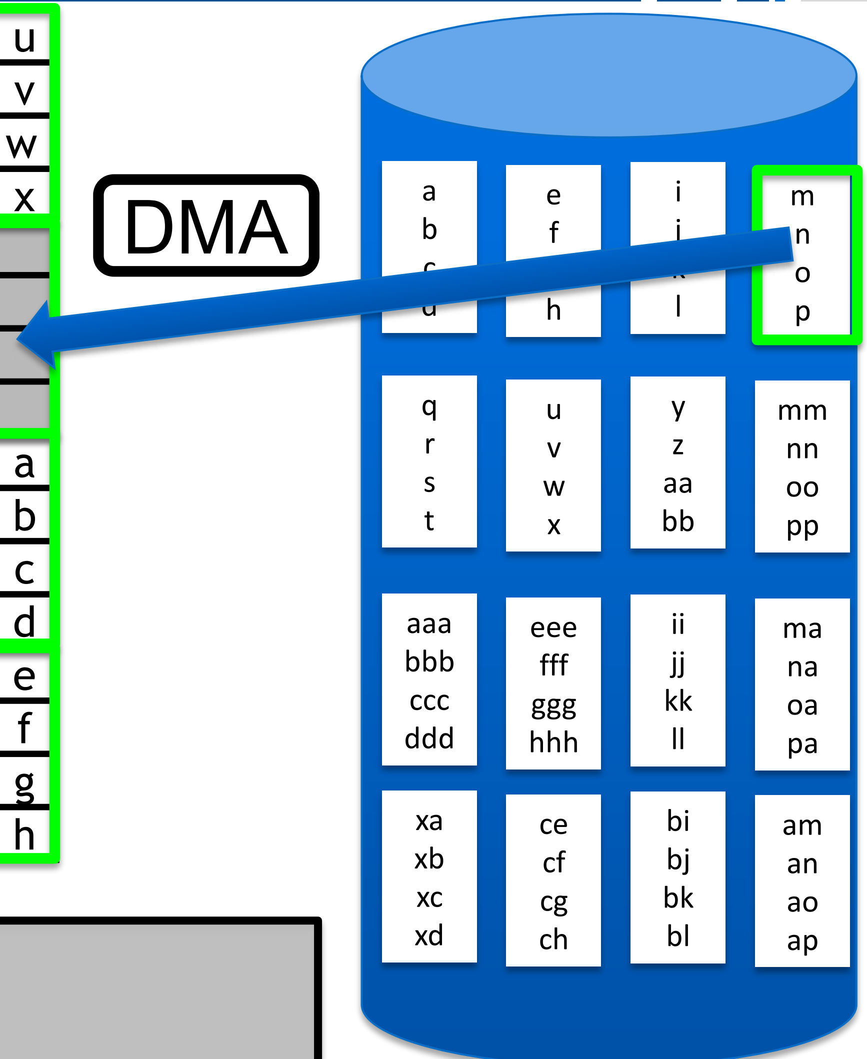


Handling Page Faults



Page	Frame	
0	2	1
1	3	1
2	2	0
3	3	0
4	1	0
5	0	1
6	0	0





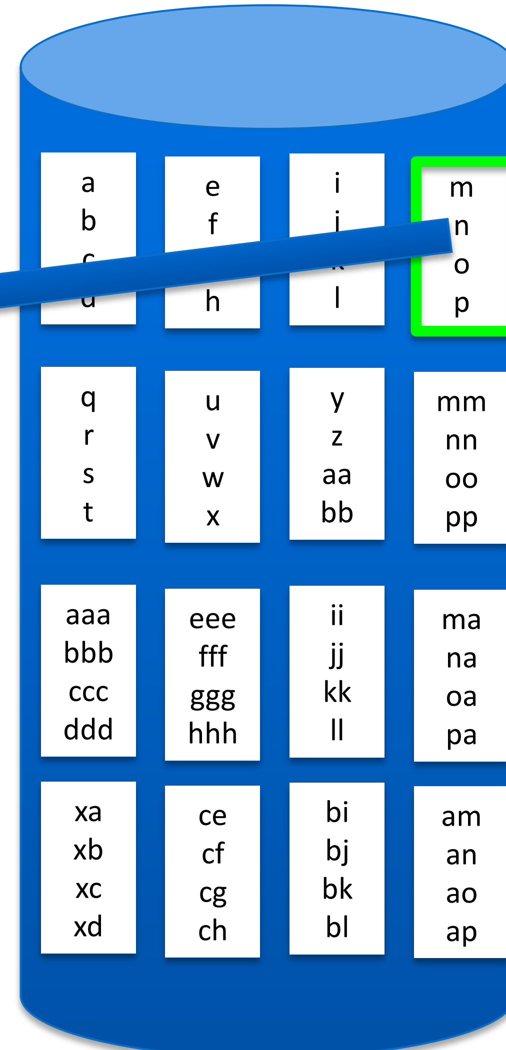
Operating System

Handling Page Faults

a	0
b	1
c	2
d	3
e	4
f	5
g	6
h	7
i	8
j	9
k	10
l	11
m	12
n	13
o	14
p	15
q	16
r	17
s	18
t	19
u	20
v	21
w	22
x	23
y	24
z	25
aa	26

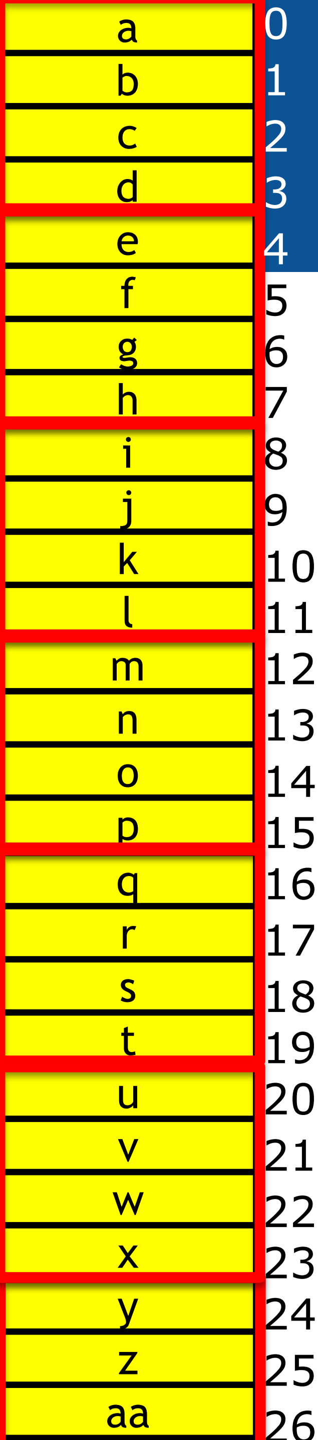
Page	Frame	
0	2	1
1	3	1
2	2	0
3	3	0
4	1	0
5	0	1
6	0	0

0	u
1	v
2	w
3	x
4	m
5	n
6	o
7	p
8	a
9	b
10	c
11	d
12	e
13	f
14	g
15	h

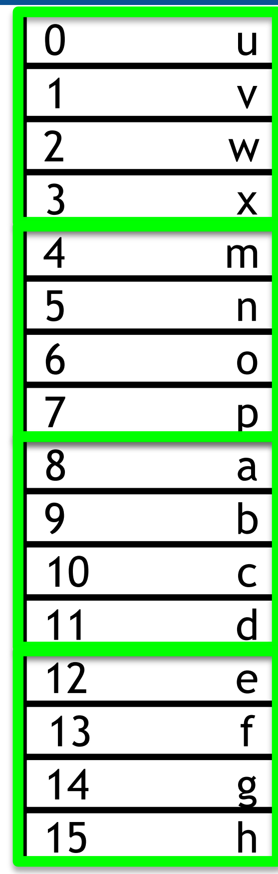


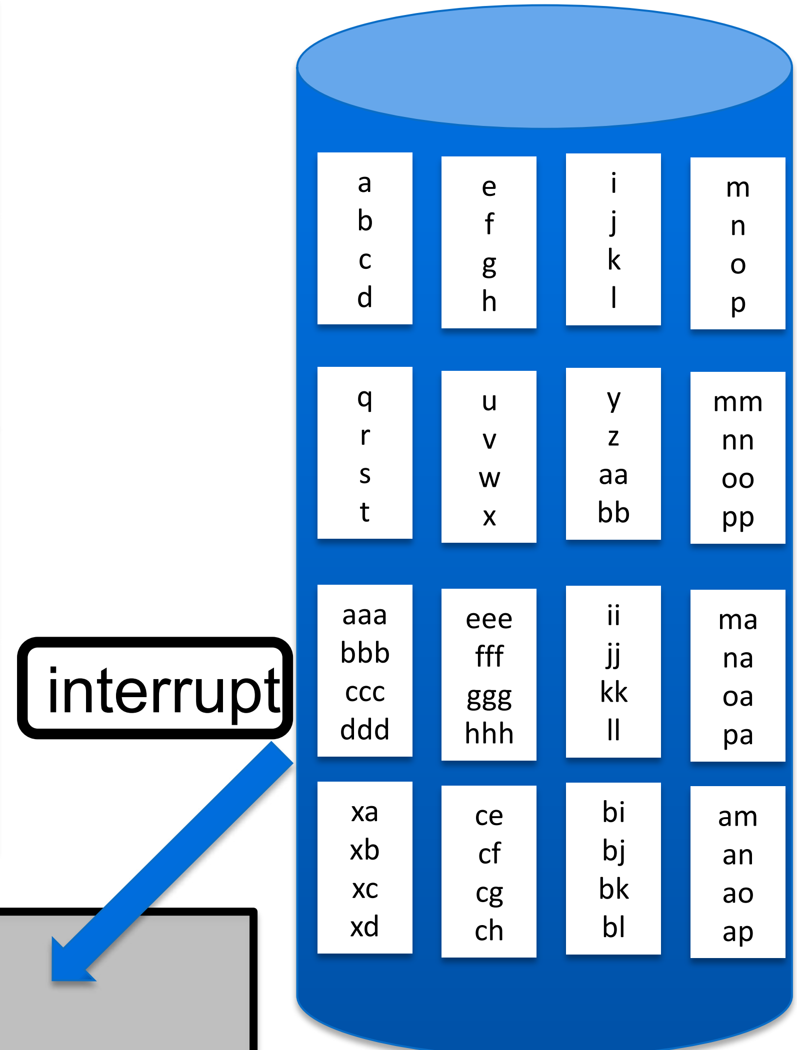
Operating System

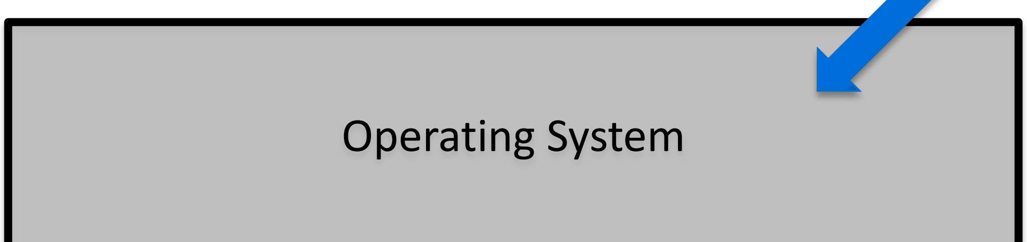
Handling Page Faults



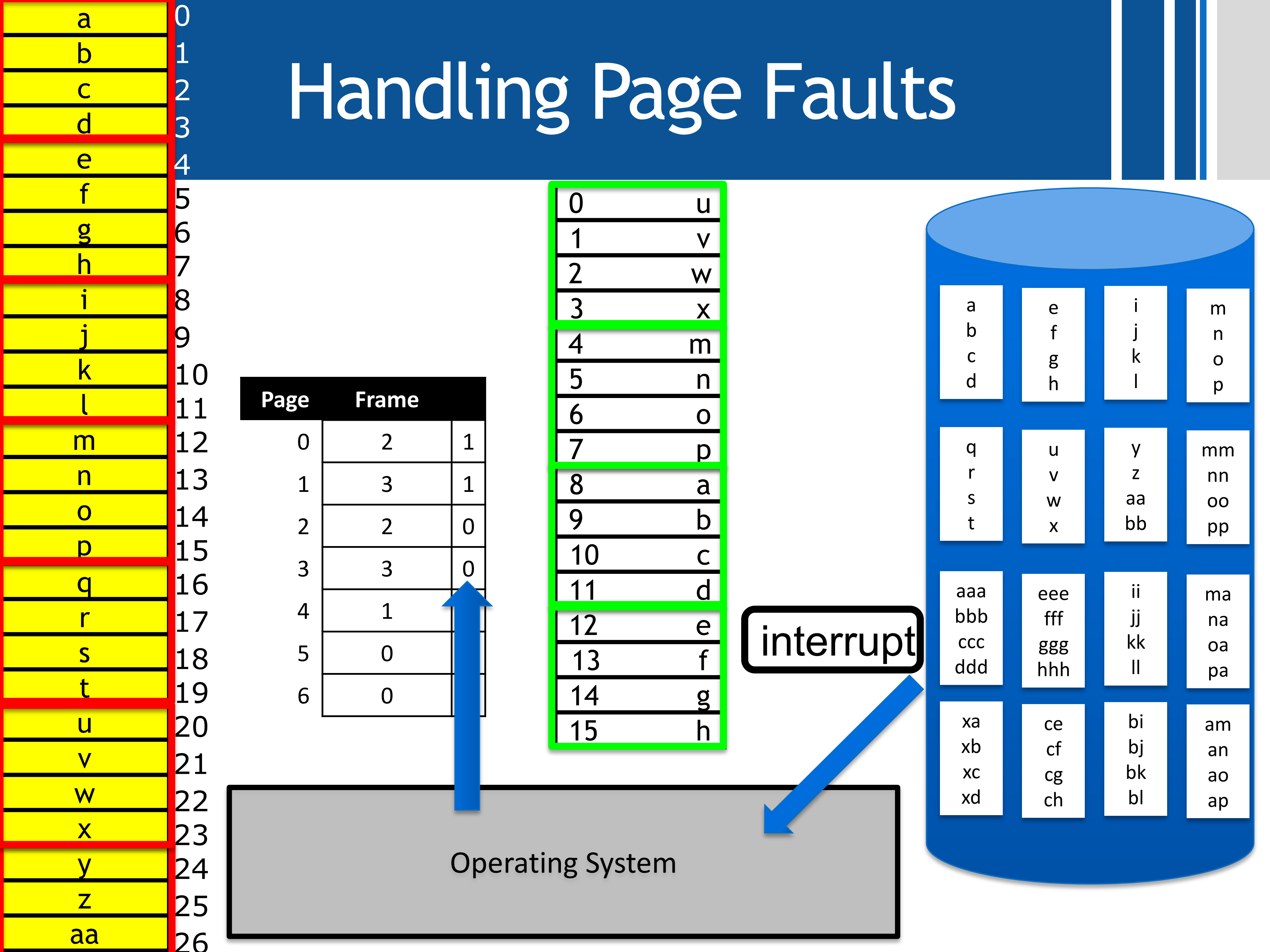
Page	Frame	
0	2	1
1	3	1
2	2	0
3	3	0
4	1	0
5	0	1
6	0	0





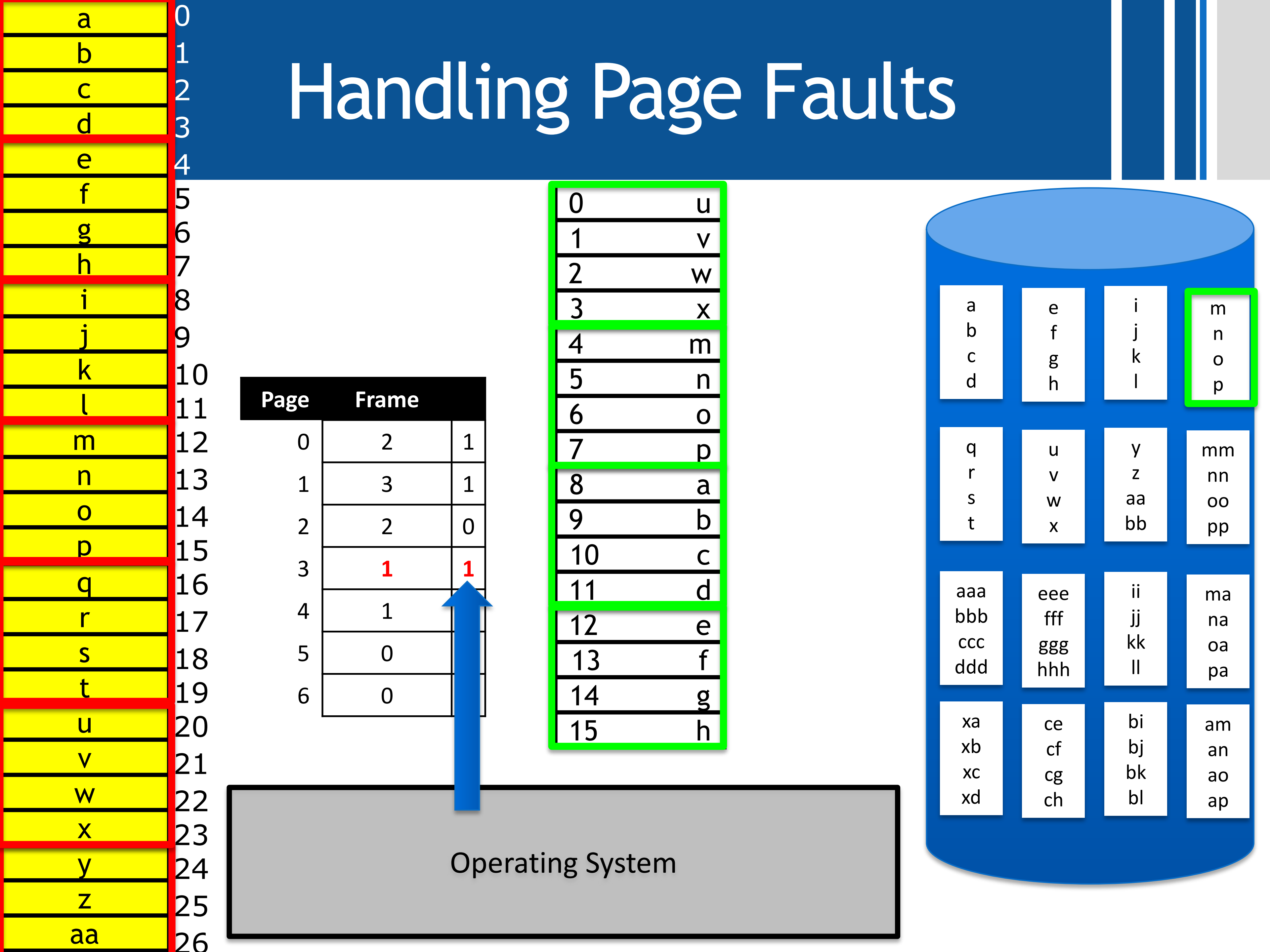


Handling Page Faults



Page	Frame	
0	2	1
1	3	1
2	2	0
3	3	0
4	1	
5	0	
6	0	

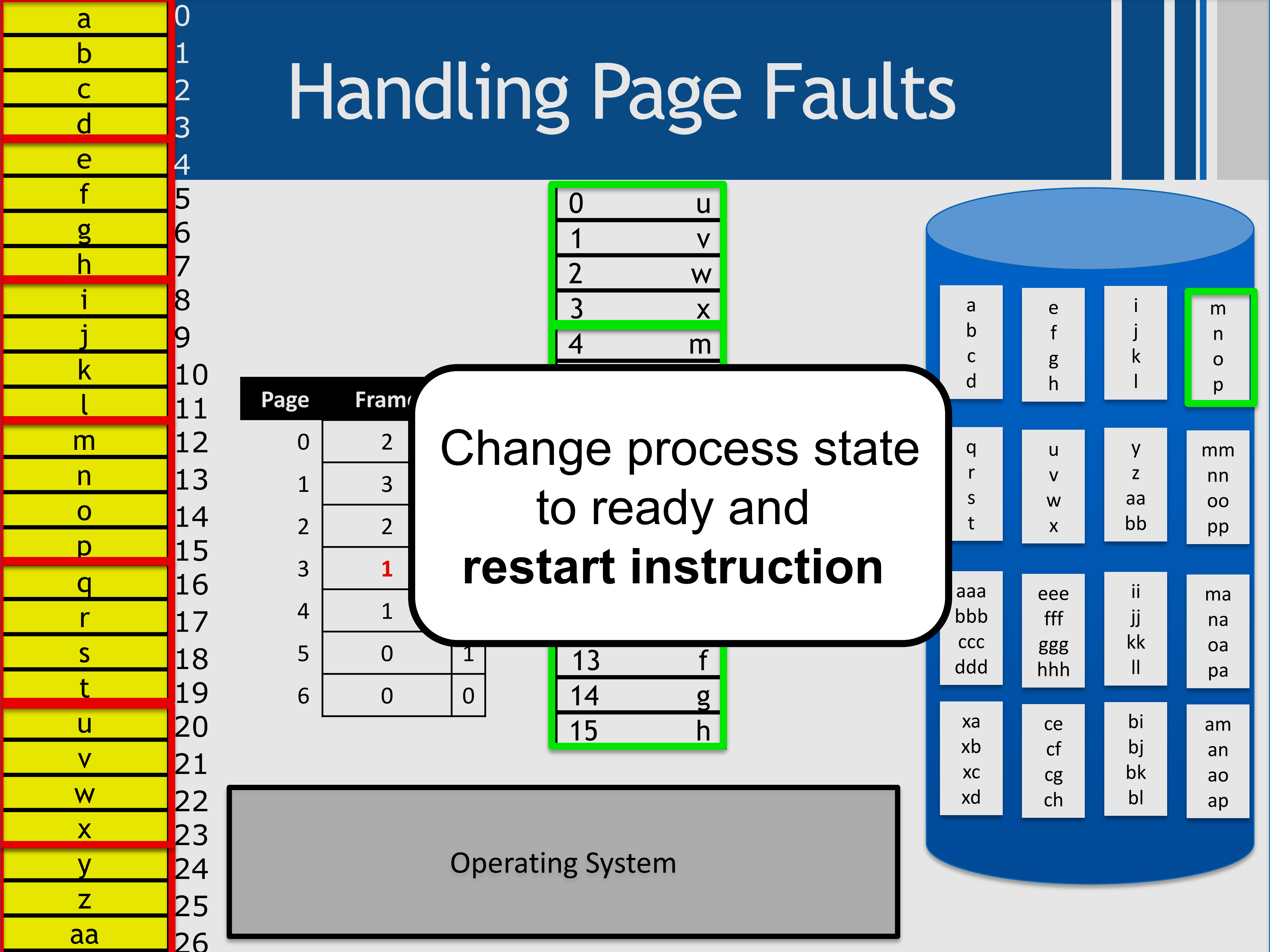
Handling Page Faults



Page	Frame	
0	2	1
1	3	1
2	2	0
3	1	1
4	1	
5	0	
6	0	

Handling Page Faults

Change process state to ready and **restart instruction**



Handling Page Faults

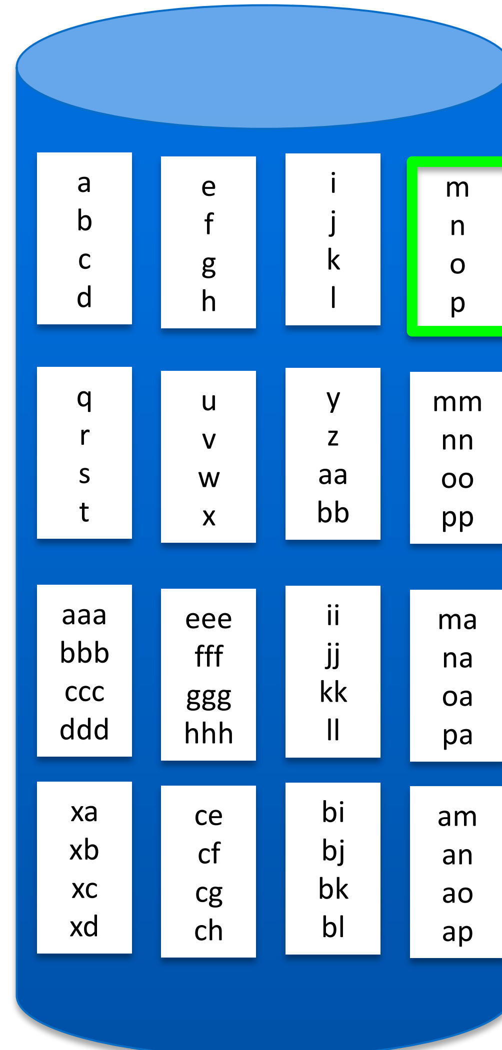
0	a
1	b
2	c
3	d
4	e
5	f
6	g
7	h
8	i
9	j
10	k
11	l
12	m
13	n
14	o
15	p
16	q
17	r
18	s
19	t
20	u
21	v
22	w
23	x
24	y
25	z
26	aa

lw 12, R1

Page	Frame	
0	2	1
1	3	1
2	2	
3	**1**	**1**
4	1	0
5	0	1
6	0	0

0	u
1	v
2	w
3	x
4	m
5	n
6	o
7	p
8	a
9	b
10	c
11	d
12	e
13	f
14	g
15	h

write m to R1



Operating System

But There’s A Slight Problem

But There's A Slight Problem

What if there is no free frame to map the page to?

Page Eviction

- Need to select some mapped page and evict it
  - Copy its data to disk
  - Use the frame for the new page
- Which page should it be?
  - Called “the victim”

Replacement Algorithms

- How do we choose the *victim*?

Replacement Algorithms

- How do we choose the *victim*?
  - We can just choose at random…

Replacement Algorithms

- How do we choose the *victim*?
  - We can just choose at random…
  - But better not to choose often used pages (will probably need to be brought back in soon)

Replacement Algorithms

- How do we choose the *victim*?
  - We can just choose at random…
  - But better not to choose often used pages (will probably need to be brought back in soon)
- Many policies are possible
  - Optimal
  - Random
  - FIFO (first-in-first-out), second chance FIFO
  - NRU (not recently used)
  - LRU (least recently used), pseudo-LRU
  - LFU (least frequently used)
  - Etc

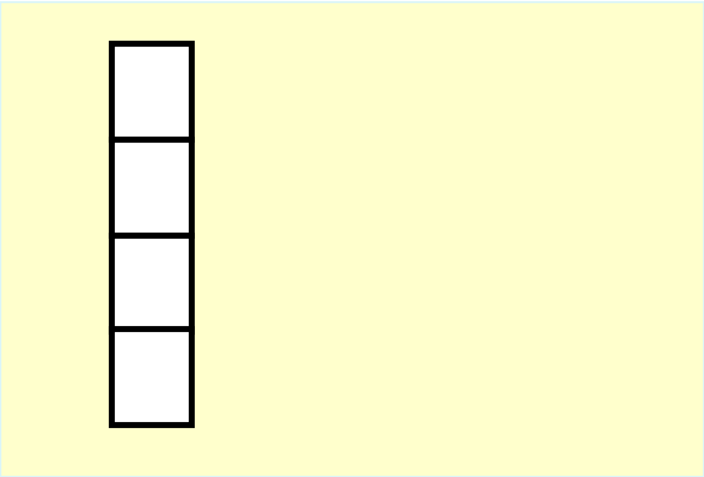
(Infeasible) Optimal Algorithm
a.k.a. Bélády's Algorithm, clairvoyant algorithm

- Replace the page that **will not be used** for the longest period of time

(Infeasible) Optimal Algorithm

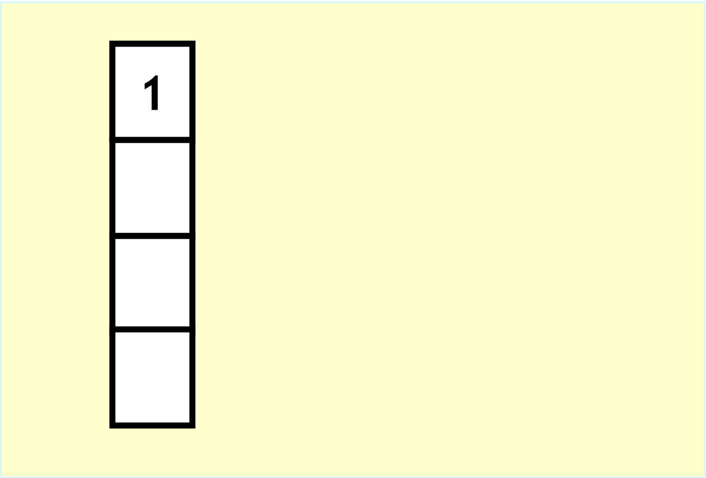
a.k.a. Bélády's Algorithm, clairvoyant algorithm

- Replace the page that **will not be used** for the longest period of time
- Page numbers: **1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5**



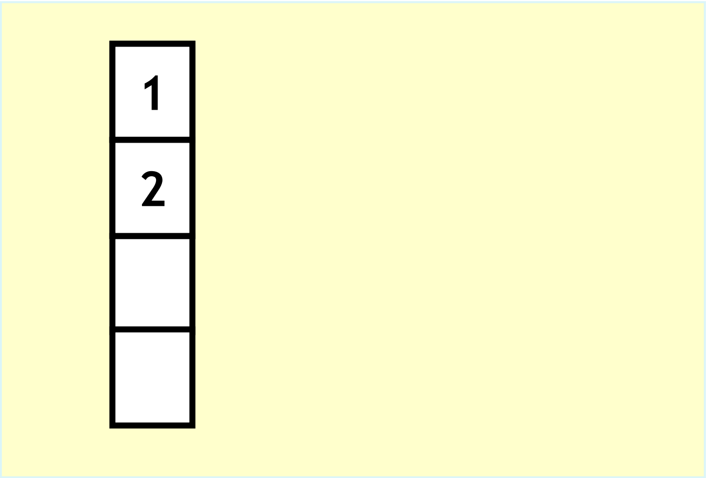
(Infeasible) Optimal Algorithm
a.k.a. Bélády's Algorithm, clairvoyant algorithm

- Replace the page that **will not be used** for the longest period of time
- Page numbers: **1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5**



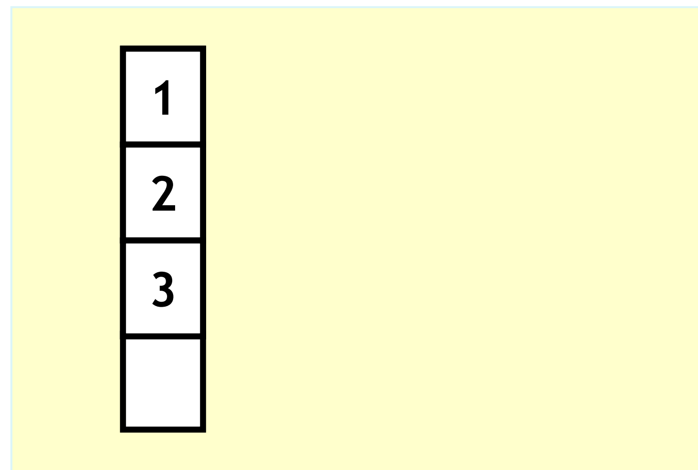
(Infeasible) Optimal Algorithm
a.k.a. Bélády's Algorithm, clairvoyant algorithm

- Replace the page that **will not be used** for the longest period of time
- Page numbers: **1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5**



(Infeasible) Optimal Algorithm
a.k.a. Bélády's Algorithm, clairvoyant algorithm

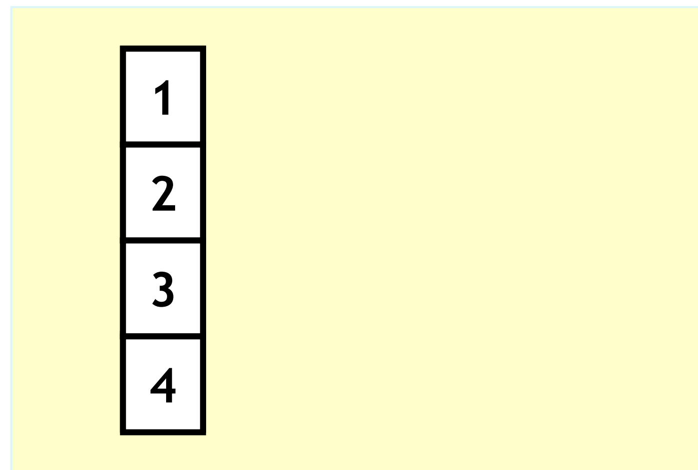
- Replace the page that **will not be used** for the longest period of time
- Page numbers: **1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5**



(Infeasible) Optimal Algorithm

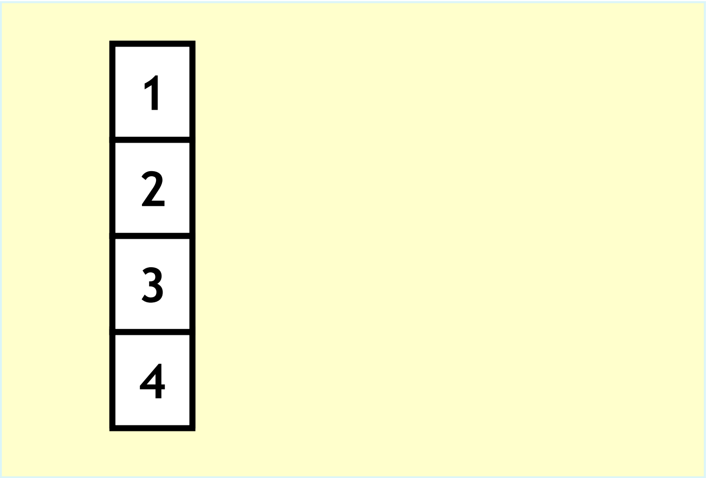
a.k.a. Bélády's Algorithm, clairvoyant algorithm

- Replace the page that **will not be used** for the longest period of time
- Page numbers: **1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5**



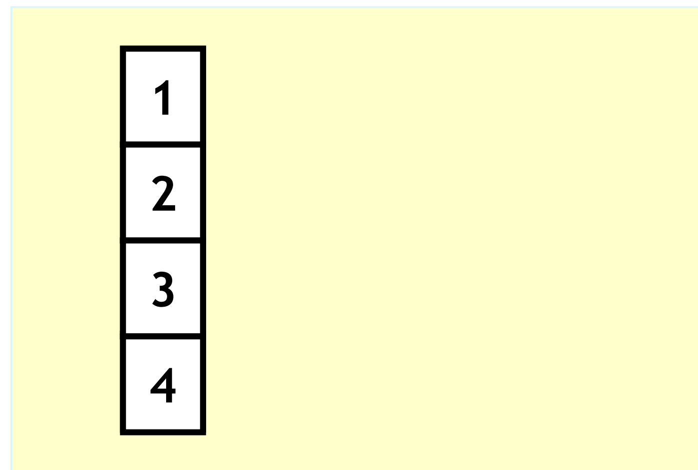
(Infeasible) Optimal Algorithm
a.k.a. Bélády's Algorithm, clairvoyant algorithm

- Replace the page that **will not be used** for the longest period of time
- Page numbers: **1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5**



(Infeasible) Optimal Algorithm
a.k.a. Bélády's Algorithm, clairvoyant algorithm

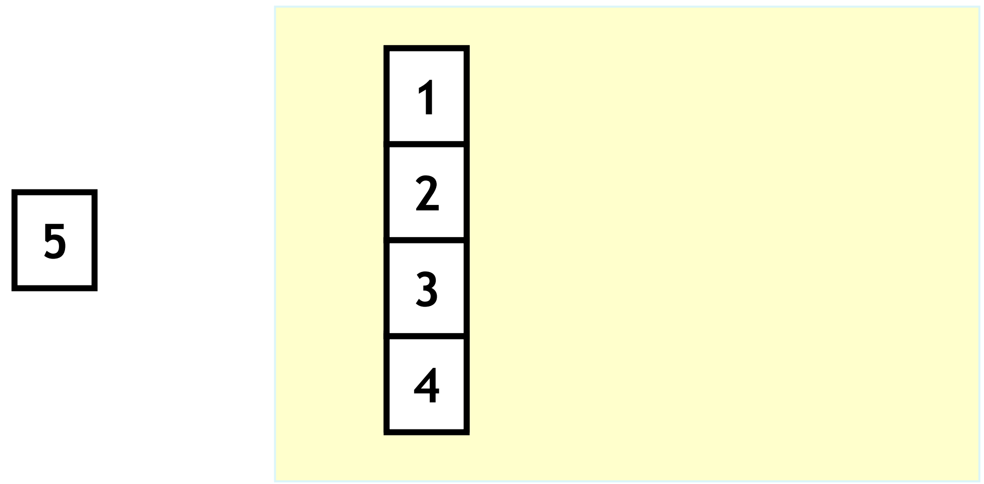
- Replace the page that **will not be used** for the longest period of time
- Page numbers: **1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5**



(Infeasible) Optimal Algorithm

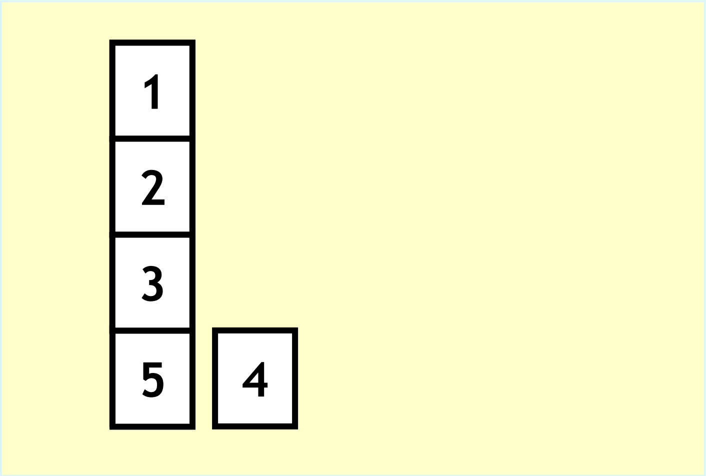
a.k.a. Bélády's Algorithm, clairvoyant algorithm

- Replace the page that **will not be used** for the longest period of time
- Page numbers: **1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5**



(Infeasible) Optimal Algorithm
a.k.a. Bélády's Algorithm, clairvoyant algorithm

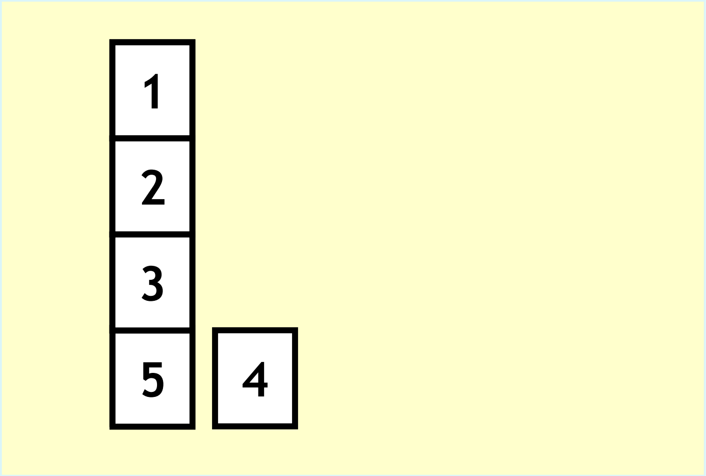
- Replace the page that **will not be used** for the longest period of time
- Page numbers: **1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5**



(Infeasible) Optimal Algorithm

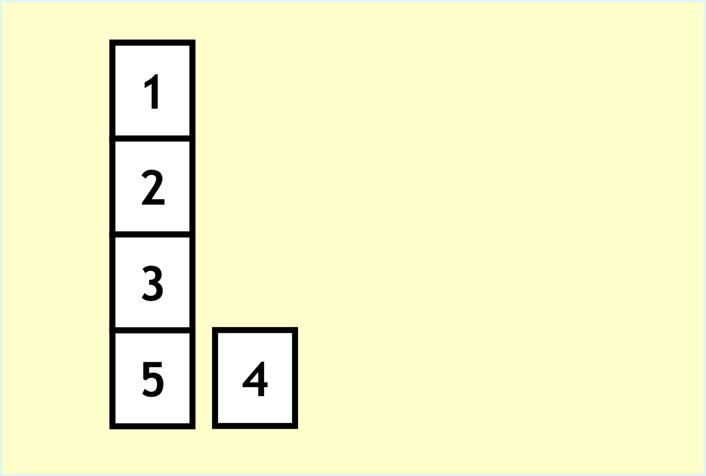
a.k.a. Bélády's Algorithm, clairvoyant algorithm

- Replace the page that **will not be used** for the longest period of time
- Page numbers: **1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5**



(Infeasible) Optimal Algorithm
a.k.a. Bélády's Algorithm, clairvoyant algorithm

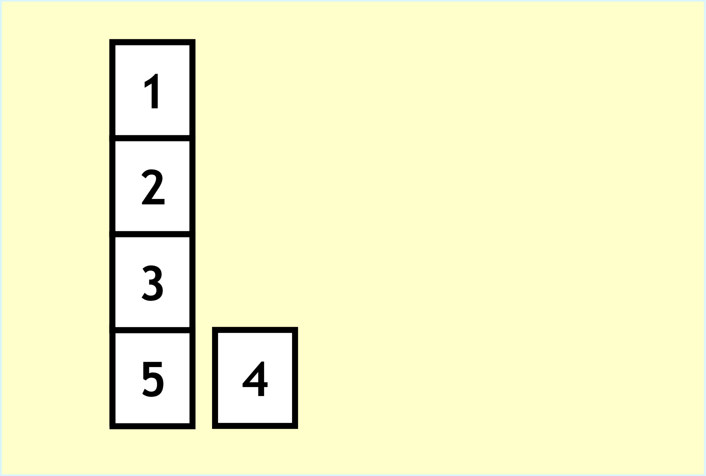
- Replace the page that **will not be used** for the longest period of time
- Page numbers: **1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5**



(Infeasible) Optimal Algorithm

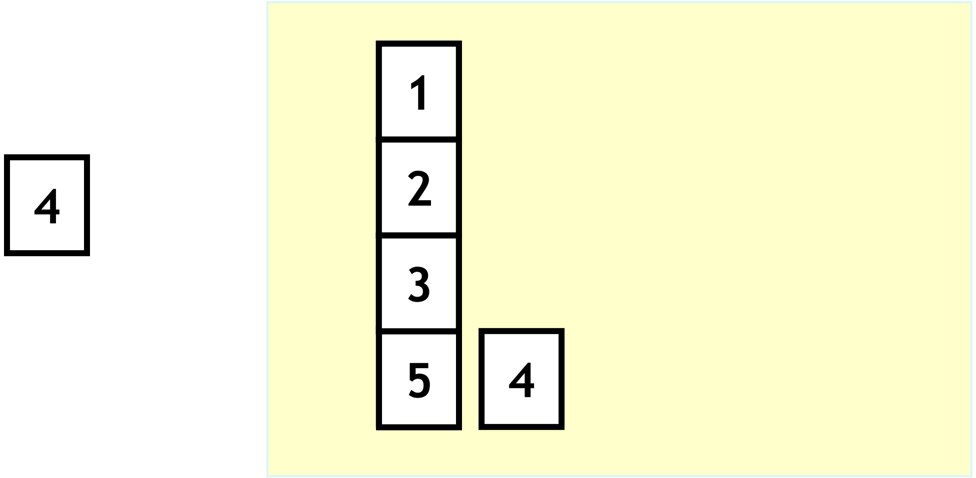
a.k.a. Bélády's Algorithm, clairvoyant algorithm

- Replace the page that **will not be used** for the longest period of time
- Page numbers: **1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5**



(Infeasible) Optimal Algorithm
a.k.a. Bélády's Algorithm, clairvoyant algorithm

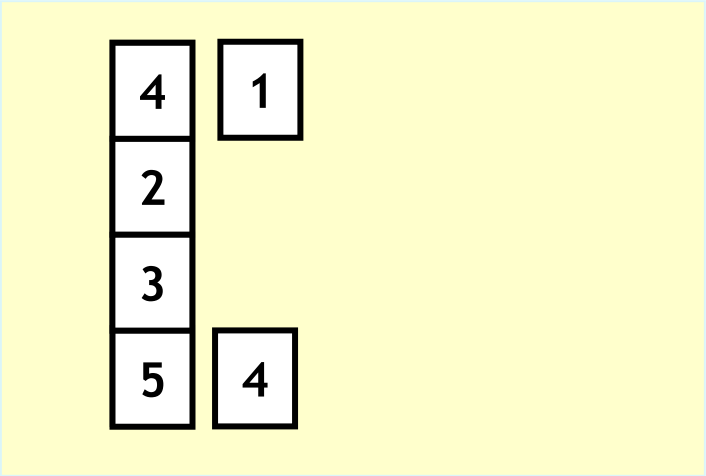
- Replace the page that **will not be used** for the longest period of time
- Page numbers: **1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5**



(Infeasible) Optimal Algorithm

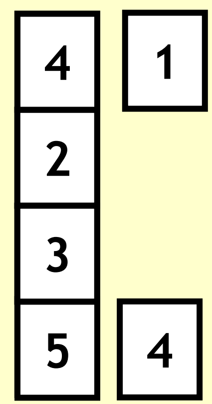
a.k.a. Bélády's Algorithm, clairvoyant algorithm

- Replace the page that **will not be used** for the longest period of time
- Page numbers: **1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5**



(Infeasible) Optimal Algorithm
a.k.a. Bélády's Algorithm, clairvoyant algorithm

- Replace the page that **will not be used** for the longest period of time
- Page numbers: **1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5**

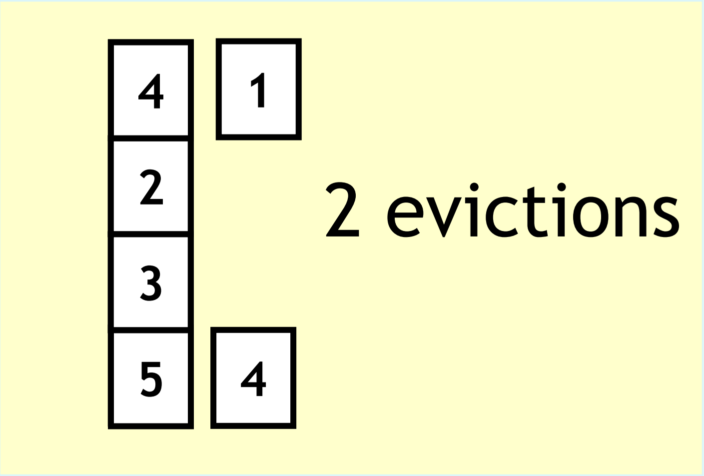


2 evictions

(Infeasible) Optimal Algorithm

a.k.a. Bélády's Algorithm, clairvoyant algorithm

- Replace the page that **will not be used** for the longest period of time
- Page numbers: **1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5**

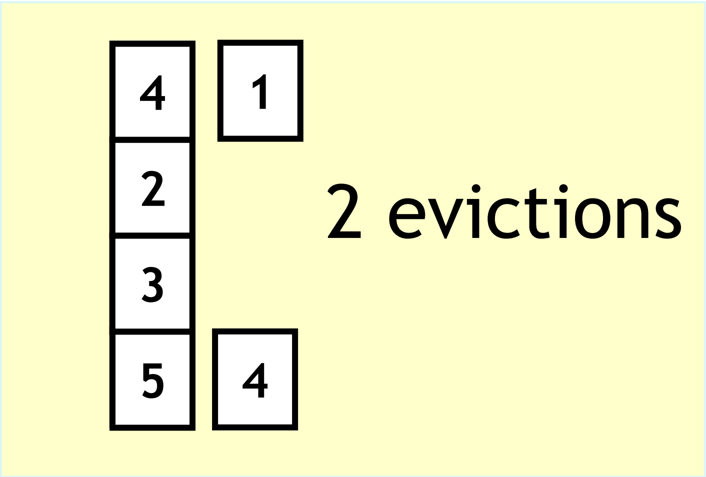


- **Infeasible**: need to know the future

(Infeasible) Optimal Algorithm

a.k.a. Bélády's Algorithm, clairvoyant algorithm

- Replace the page that **will not be used** for the longest period of time
- Page numbers: **1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5**



- **Infeasible**: need to know the future
- Used only for comparison
  - Given an algorithm, how close is it to optimal?

Random Replacement

- Just evict any page randomly
- The other extreme from optimal
  - Optimal uses full knowledge of everything, including the future
  - Random uses no knowledge of anything, including the past
- Also used for comparison
  - Given an algorithm, is it better than random?

First-In-First-Out (FIFO)

- What information can the OS use?

First-In-First-Out (FIFO)

- What information can the OS use?
- Sort the pages in the order they were loaded to memory
  - E.g., maintain the order as a link list

First-In-First-Out (FIFO)

- What information can the OS use?
- Sort the pages in the order they were loaded to memory
  - E.g., maintain the order as a link list
- The victim is the first page (in this order)
  - The “oldest” page

First-In-First-Out (FIFO)

- What information can the OS use?
- Sort the pages in the order they were loaded to memory
  - E.g., maintain the order as a link list
- The victim is the first page (in this order)
  - The “oldest” page
- Disadvantage: page in the memory the longest may actually be used often

First-In-First-Out (FIFO)

- What information can the OS use?
- Sort the pages in the order they were loaded to memory
  - E.g., maintain the order as a link list
- The victim is the first page (in this order)
  - The “oldest” page
- Disadvantage: page in the memory the longest may actually be used often
- Used in Windows NT
  - Independent of any hardware support

Belady's Anomaly

- Excepted behavior: Larger memory → less page faults

Belady’s Anomaly

- Excepted behavior: Larger memory → less page faults
- Consider: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

Belady’s Anomaly

- Excepted behavior: Larger memory → less page faults
- Consider: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

Memory with 3 frames

newest	1	2	3	4	1	2	5	5	5	3	4	4
		1	2	3	4	1	2	2	2	5	3	3
oldest			1	2	3	4	1	1	1	2	5	5

time →

Belady’s Anomaly

- Excepted behavior: Larger memory → less page faults
- Consider: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

Memory with 3 frames

newest	1	2	3	4	1	2	5	5	5	3	4	4
		1	2	3	4	1	2	2	2	5	3	3
oldest			1	2	3	4	1	1	1	2	5	5

time →

Memory with 4 frames

newest	1	2	3	4	4	4	5	1	2	3	4	5
		1	2	3	3	3	4	5	1	2	3	4
			1	2	2	2	3	4	5	1	2	3
oldest				1	1	1	2	3	4	5	1	2

Stop and Think:
What Do We Really Want?

Stop and Think: What Do We Really Want?

- Ideally, to know the future

Stop and Think: What Do We Really Want?

- Ideally, to know the future
  - But why? What information do we want?

Stop and Think: What Do We Really Want?

- Ideally, to know the future
  - But why? What information do we want?
- We actually want to identify the process's WORKING SET
- This is the set of pages it is currently using
  - Pages in the working set should be retained
  - Pages not in the working set can be evicted

Stop and Think: What Do We Really Want?

- Ideally, to know the future
  - But why? What information do we want?
- We actually want to identify the process's WORKING SET
- This is the set of pages it is currently using
  - Pages in the working set should be retained
  - Pages not in the working set can be evicted
- Based on the principle of locality

The Working Set

- So how can we identify the working set?

The Working Set

- So how can we identify the working set?
- How is it even defined formally?

The Working Set

- So how can we identify the working set?
- How is it even defined formally?
- Parametric definition: the set of pages accessed in the last k memory accesses
  - With random access, working set size $\approx k$
  - With locality, working set size $\ll k$

The Working Set

- So how can we identify the working set?
- How is it even defined formally?
- Parametric definition: the set of pages accessed in the last *k* memory accesses
  - With random access, working set size $\approx k$
  - With locality, working set size $\ll k$
- In theory: start with k=1, increase k until working set stabilizes

The Working Set

- So how can we identify the working set?
- How is it even defined formally?
- Parametric definition: the set of pages accessed in the last *k* memory accesses
  - With random access, working set size $\approx k$
  - With locality, working set size $\ll k$
- In theory: start with k=1, increase k until working set stabilizes
- In practice: (for large k) which pages have been used recently?

Hardware Support

- Memory access is done at clock speed
- So need hardware support to track it
- But need to also limit overhead

Reference bit: a.k.a. used bit	Turned on when a page is accessed
Dirty bit:	Turned on when a page is modified

- Done by MMU at each access
- Supported on Intel processors

Not Recently Used (NRU)

- We cannot know the future, so we try to estimate according to the past

Not Recently Used (NRU)

- We cannot know the future, so we try to estimate according to the past
- Use the pages' reference bits

Not Recently Used (NRU)

- We cannot know the future, so we try to estimate according to the past
- Use the pages' reference bits
- Periodically (on clock interrupt), all reference bits are cleared
  – Another version: cleared if all blocks are referenced

Not Recently Used (NRU)

- We cannot know the future, so we try to estimate according to the past
- Use the pages' reference bits
- Periodically (on clock interrupt), all reference bits are cleared
  - Another version: cleared if all blocks are referenced
- When need to choose a victim, choose randomly among the pages with zero reference bit
  - Logic: have not been accessed since last clearing

Not Recently Used (NRU)

- We cannot know the future, so we try to estimate according to the past
- Use the pages' reference bits
- Periodically (on clock interrupt), all reference bits are cleared
  - Another version: cleared if all blocks are referenced
- When need to choose a victim, choose randomly among the pages with zero reference bit
  - Logic: have not been accessed since last clearing
- Can be implemented, but very crude

Least Recently Used (LRU)

Least Recently Used (LRU)

- Temporal locality → pages that were recently used are likely to be used again

Least Recently Used (LRU)

- Temporal locality → pages that were recently used are likely to be used again
- So evict the one that was least recently used
  - Gives good approximation of working set
  - Same logic as NRU, but better accuracy

Least Recently Used (LRU)

- Temporal locality → pages that were recently used are likely to be used again
- So evict the one that was least recently used
  - Gives good approximation of working set
  - Same logic as NRU, but better accuracy
- LRU implementation is difficult/expensive

Least Recently Used (LRU)

- Temporal locality → pages that were recently used are likely to be used again
- So evict the one that was least recently used
  - Gives good approximation of working set
  - Same logic as NRU, but better accuracy
- LRU implementation is difficult/expensive
  - Timestamps? How many bits?
  - Must find minimal timestamp on each eviction

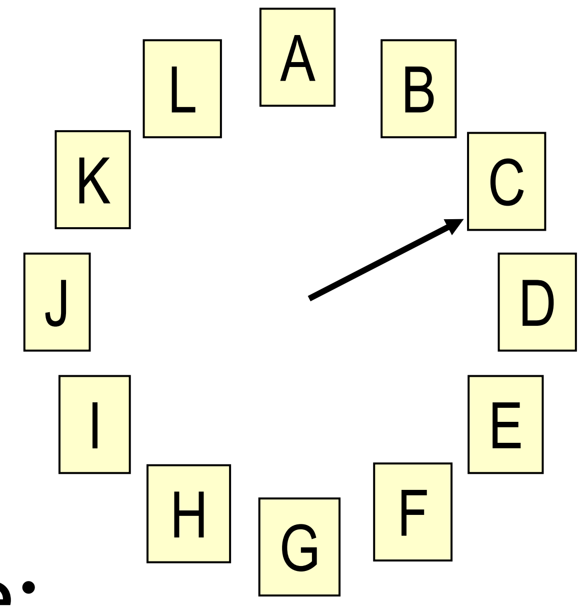
Least Recently Used (LRU)

- Temporal locality → pages that were recently used are likely to be used again
- So evict the one that was least recently used
  - Gives good approximation of working set
  - Same logic as NRU, but better accuracy
- LRU implementation is difficult/expensive
  - Timestamps? How many bits?
  - Must find minimal timestamp on each eviction
  - Sorted list? Re-sort on every access?
  - List overhead: $\log_2(n)$ bits / page

The Clock Algorithm

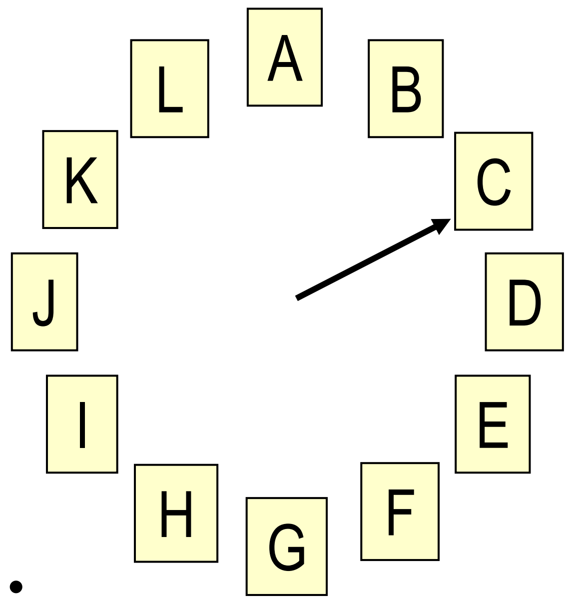
a.k.a. Second Chance Algorithm

- Think of all the frames as a circular list
- With a hand pointing at one of them
- Each page has a reference bit
- When you need to evict a page:
  1. As long as the pointed page has been referenced, clear the bit and move on
  2. Evict the first non-referenced page found



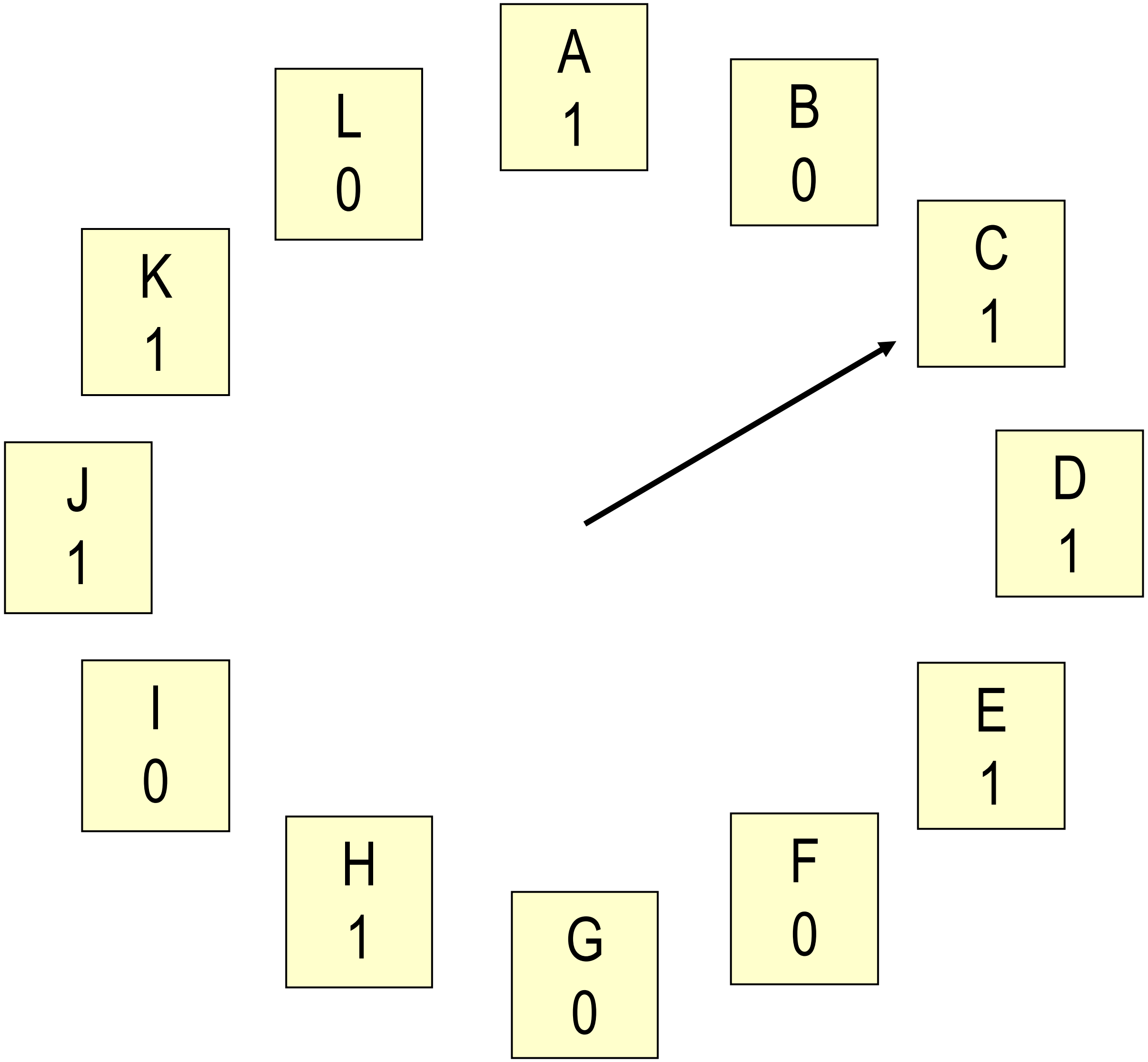
The Clock Algorithm
a.k.a. Second Chance Algorithm

- Think of all the frames as a circular list
- With a hand pointing at one of them
- Each page has a reference bit
- When you need to evict a page:
  1. As long as the pointed page has been referenced, clear the bit and move on
  2. Evict the first non-referenced page found

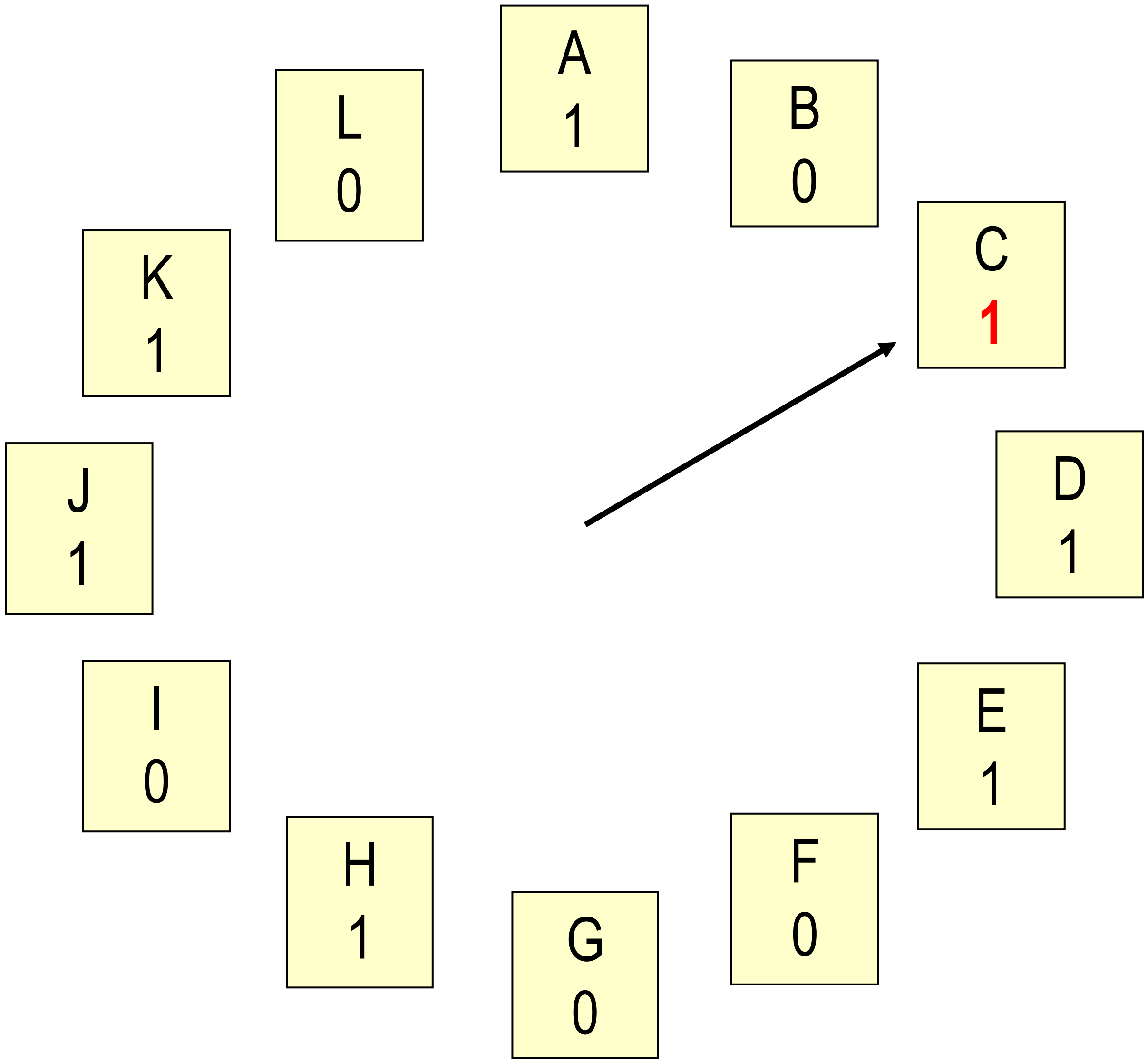


Give it a second chance

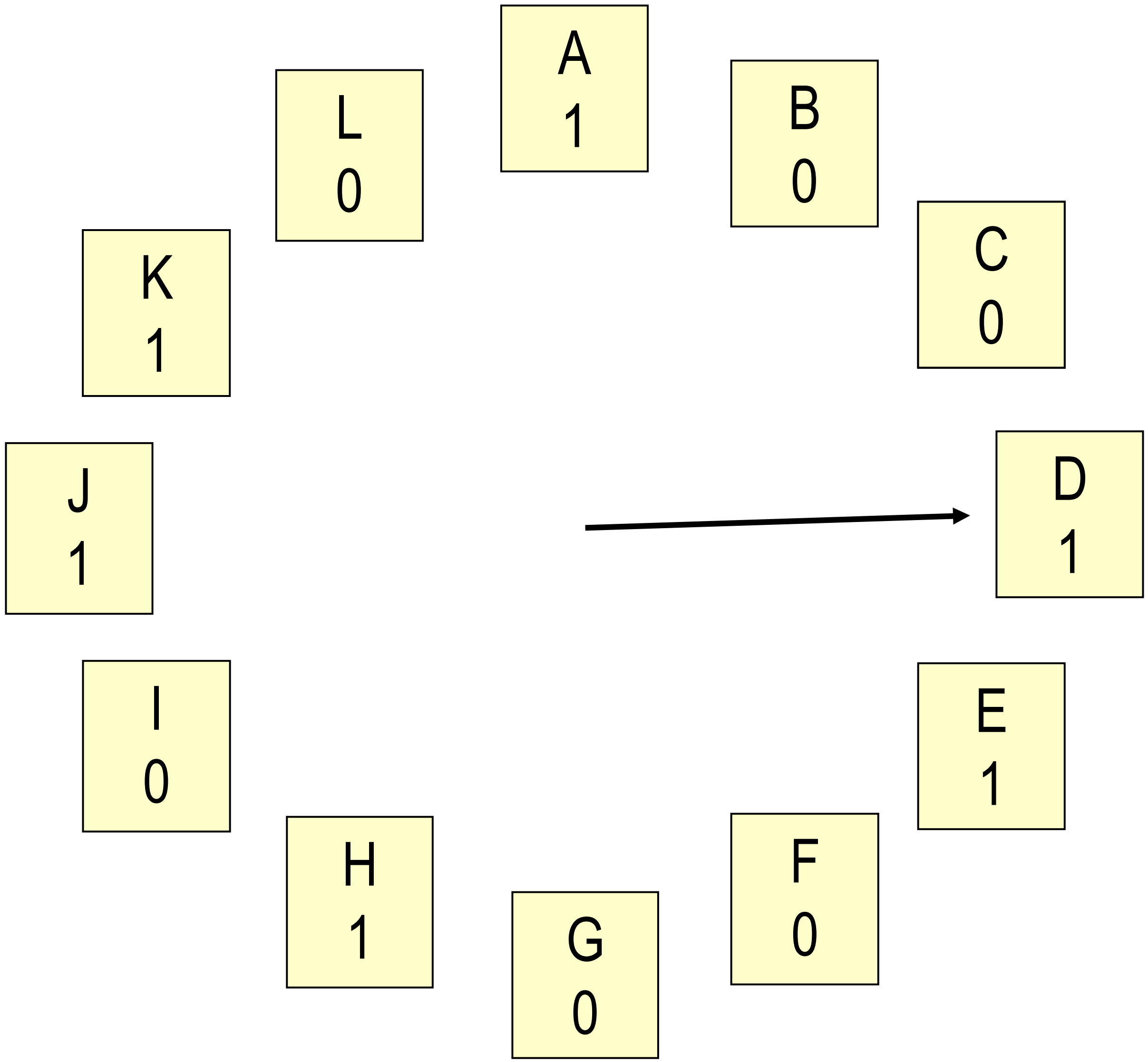
The Clock Algorithm



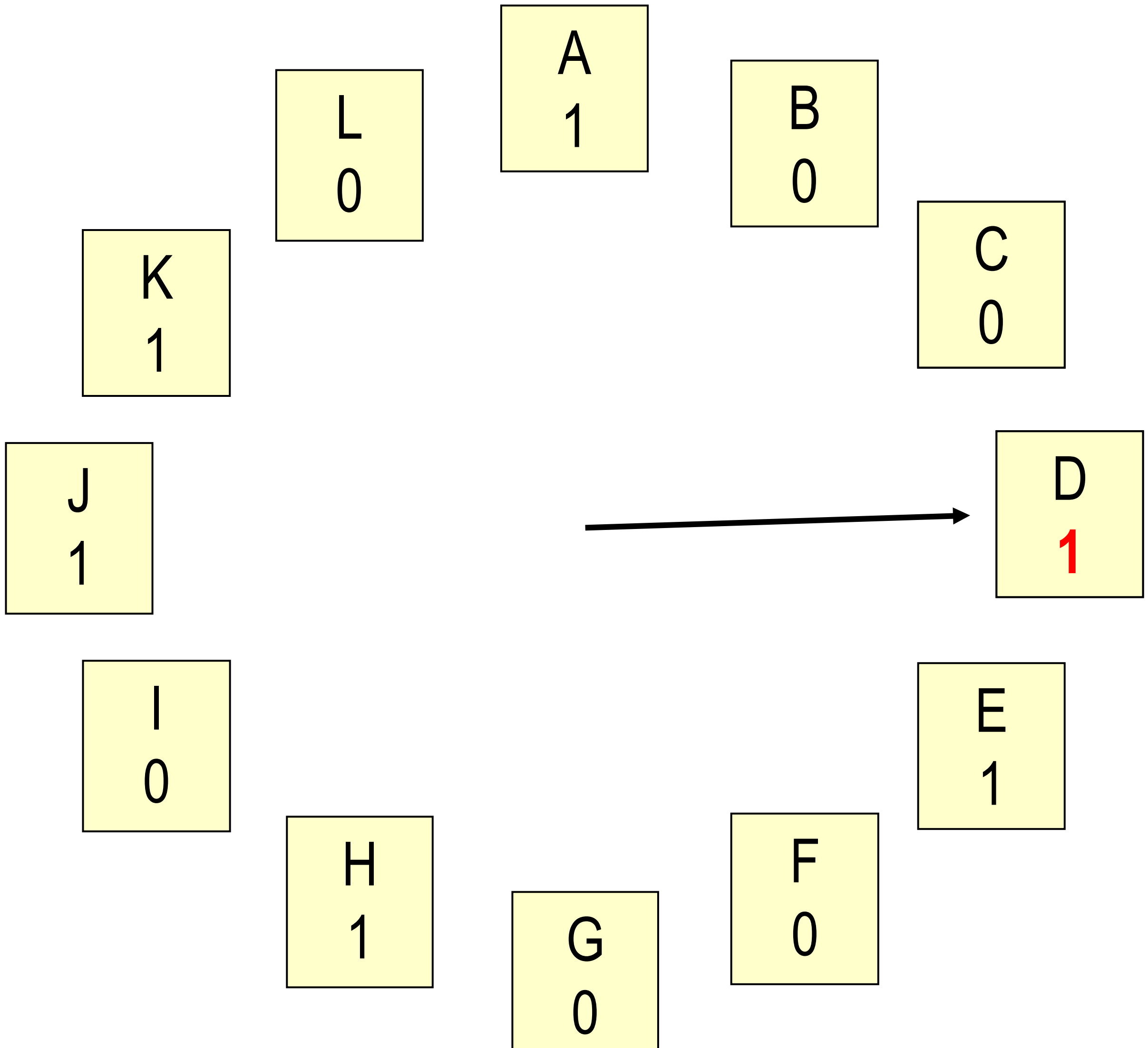
The Clock Algorithm



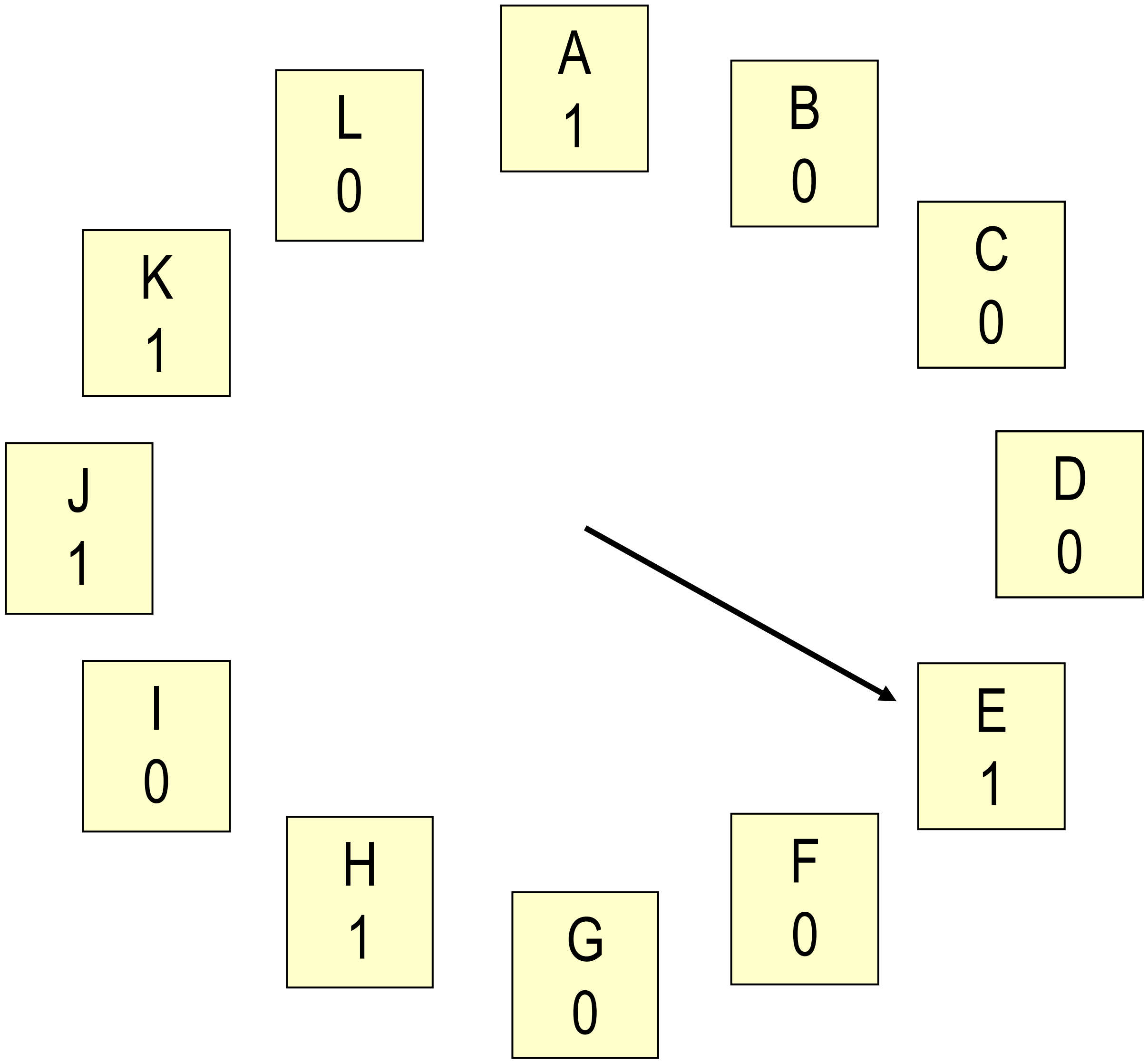
The Clock Algorithm



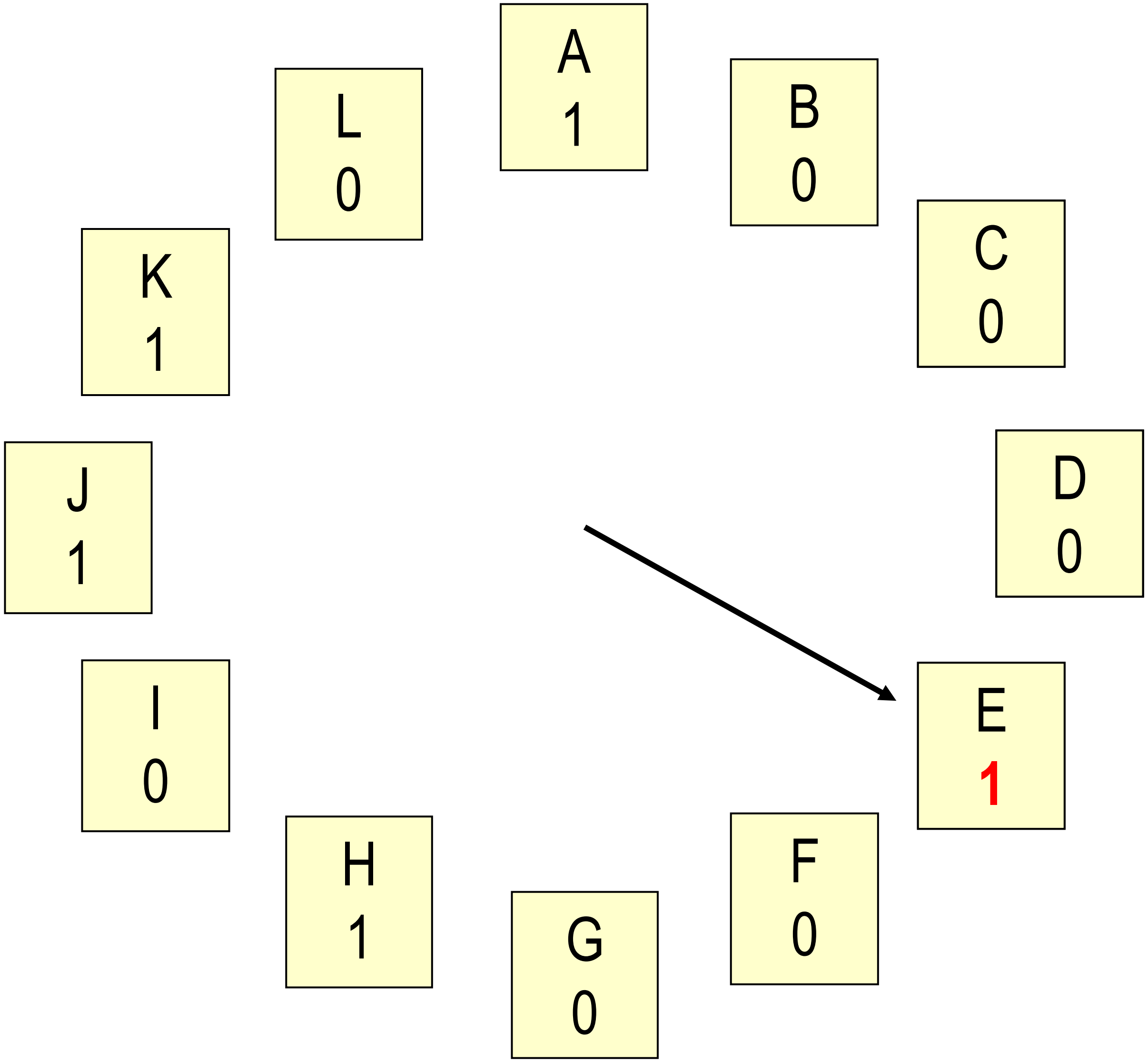
The Clock Algorithm



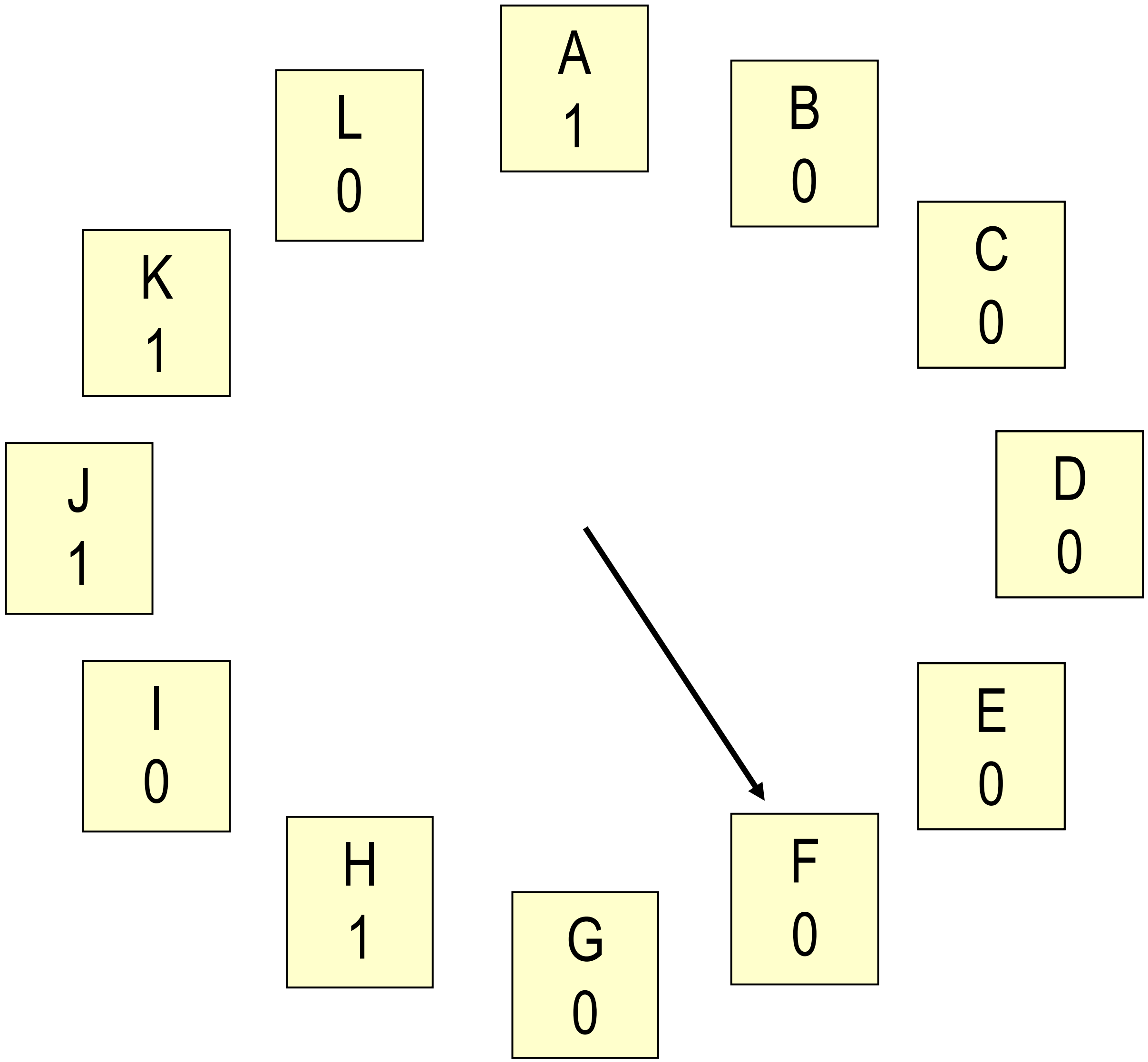
The Clock Algorithm



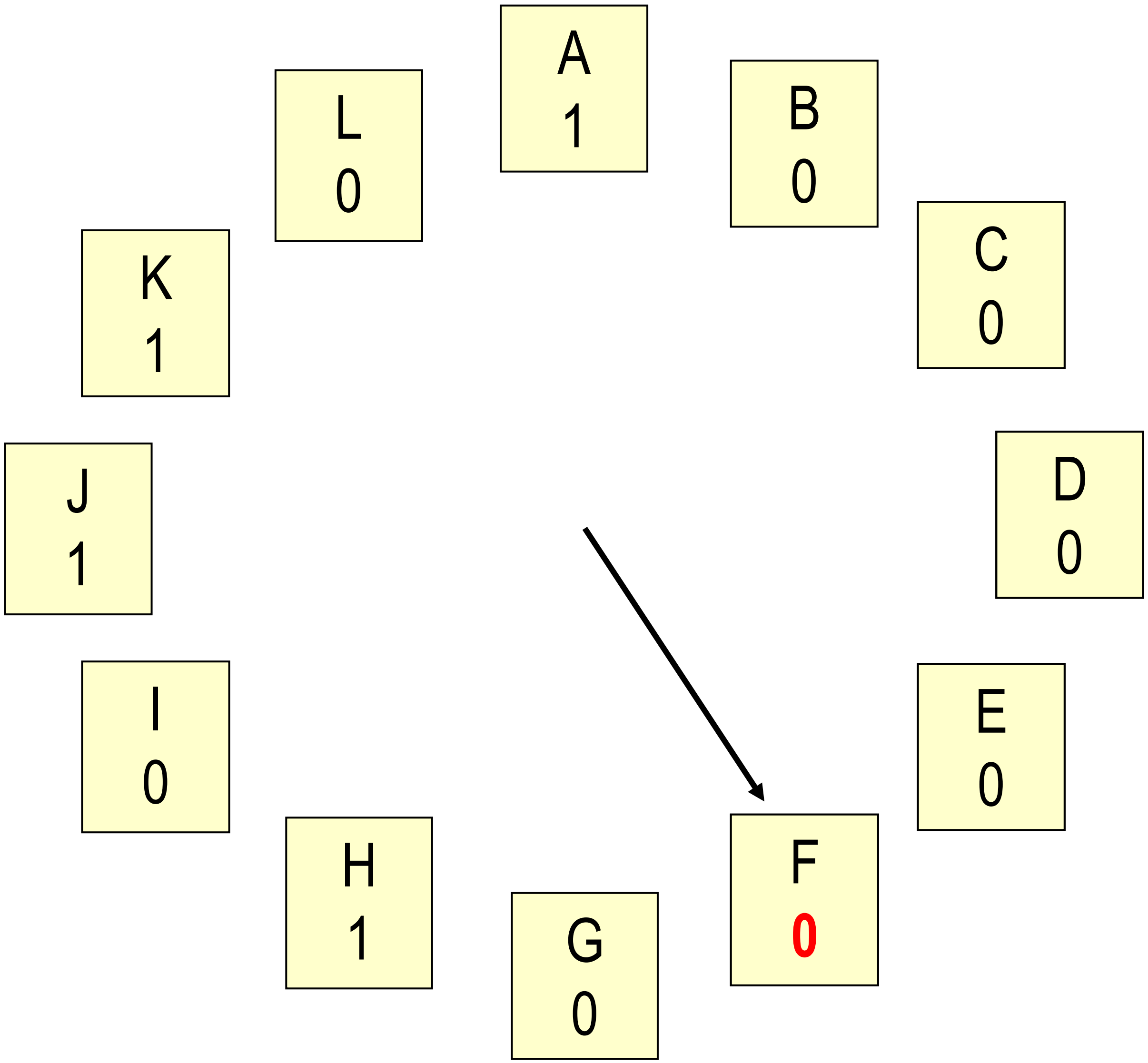
The Clock Algorithm



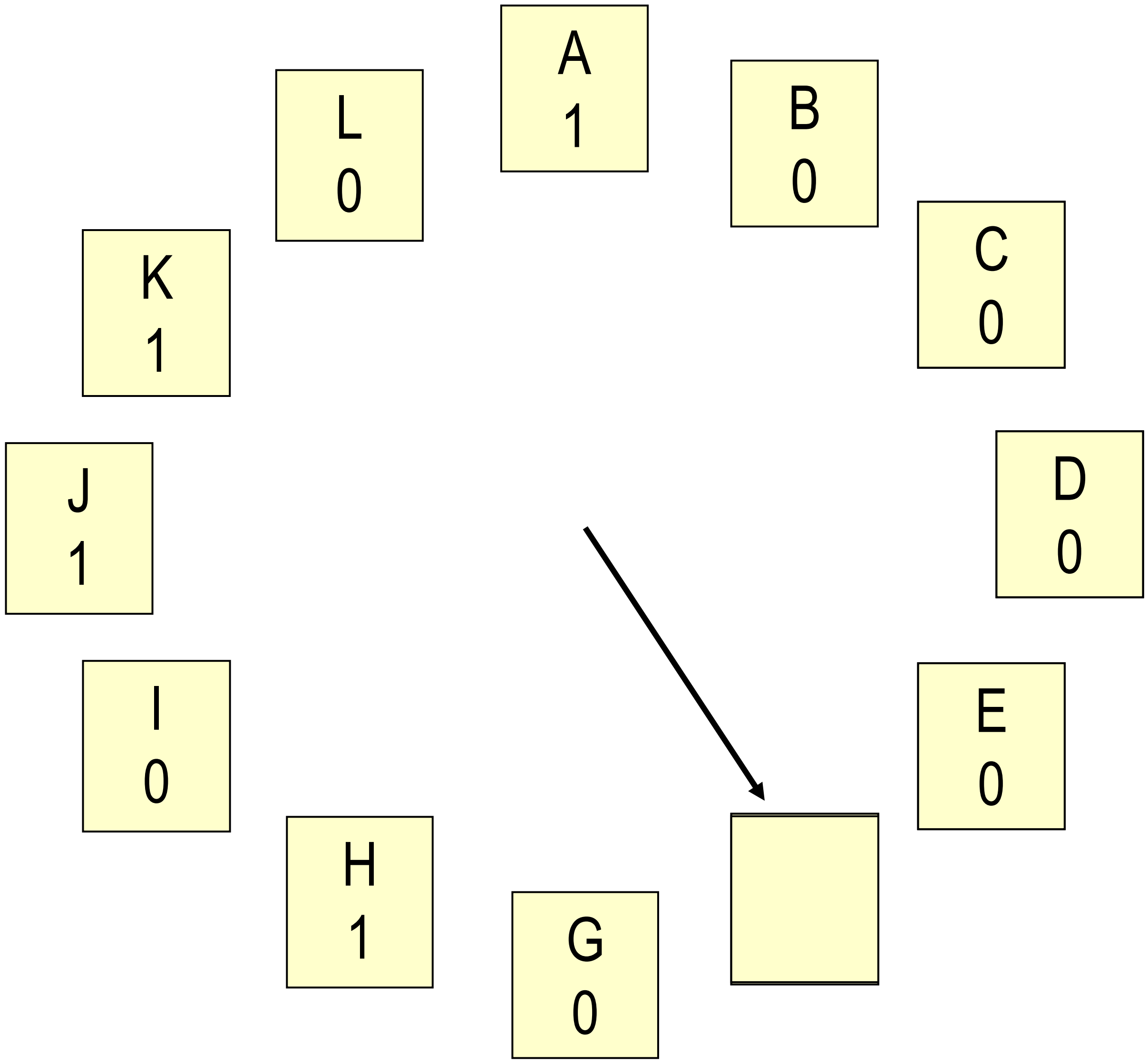
The Clock Algorithm



The Clock Algorithm



The Clock Algorithm

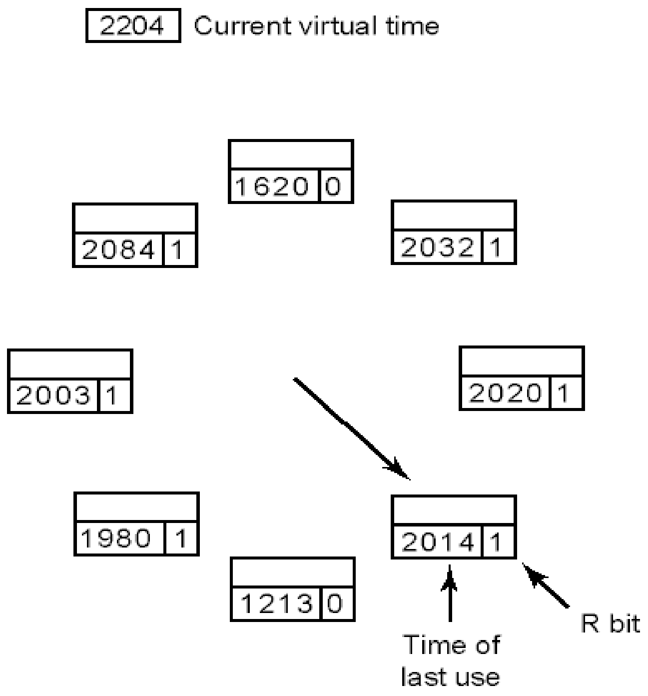


From References to milli-secs

- Clock algorithm is rather crude
- No real discrimination between pages with different activity patterns
- Improvement: try to track time since last reference
  - Not only that it existed
- Method: maintain crude timestamp per page

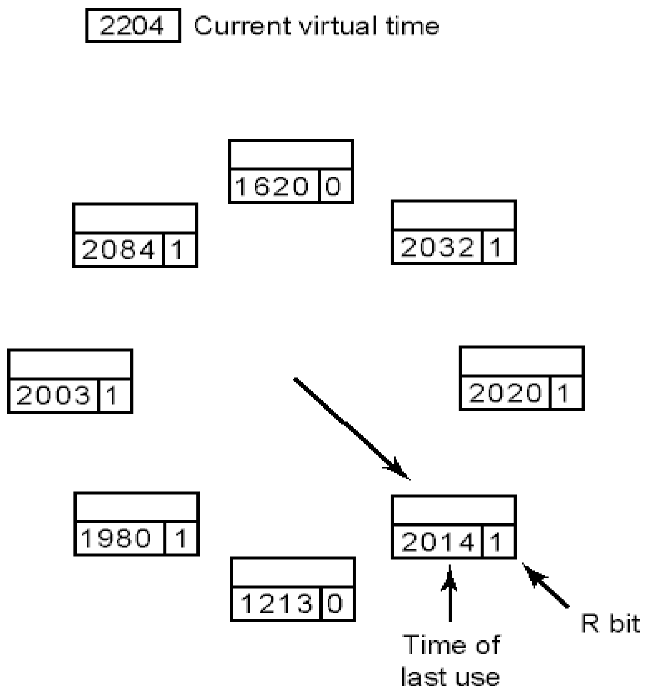
Clock+Time Algorithm

- Maintain all pages of a process in a circular list
- Maintain virtual time variable
- Every time a page is referenced, set R=1
- Upon a clock tick:
  – For each page with R=1, set time of last use to virtual time
  – Advance virtual time, reset R for all pages



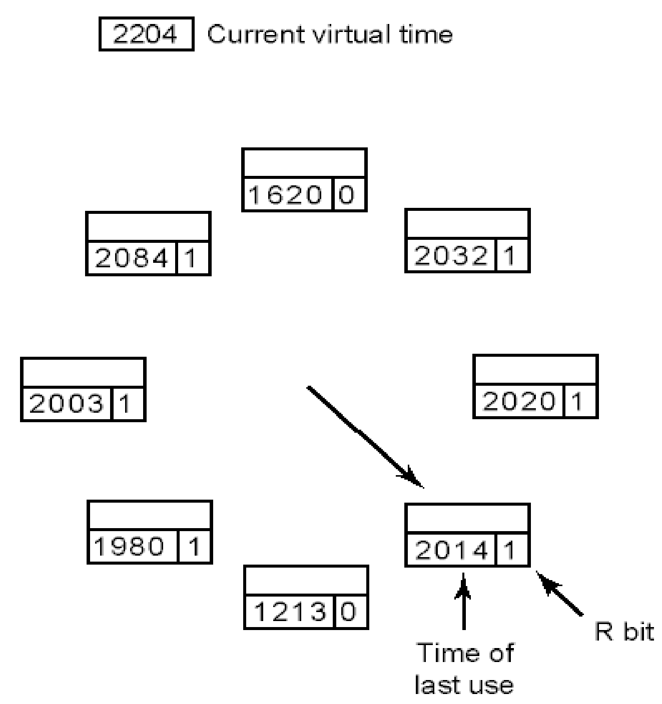
Clock+Time Algorithm

- Upon a page fault, if eviction is needed



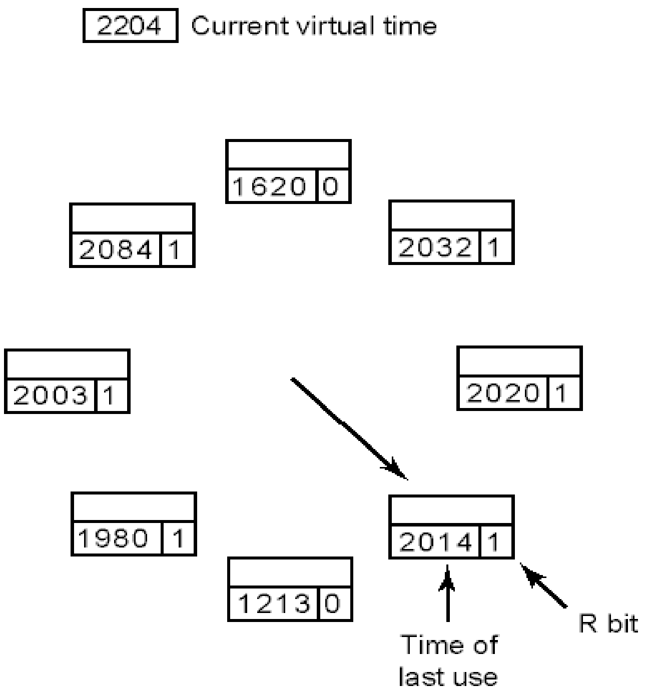
Clock+Time Algorithm

- Upon a page fault, if eviction is needed
  1. Look at the page with the arrow



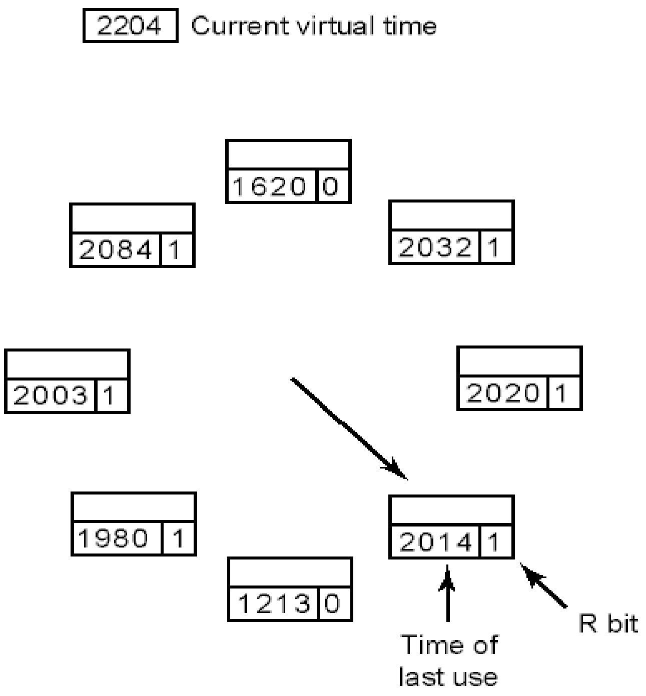
Clock+Time Algorithm

- Upon a page fault, if eviction is needed
  1. Look at the page with the arrow
  2. If R=1, advance arrow, go to Line 1



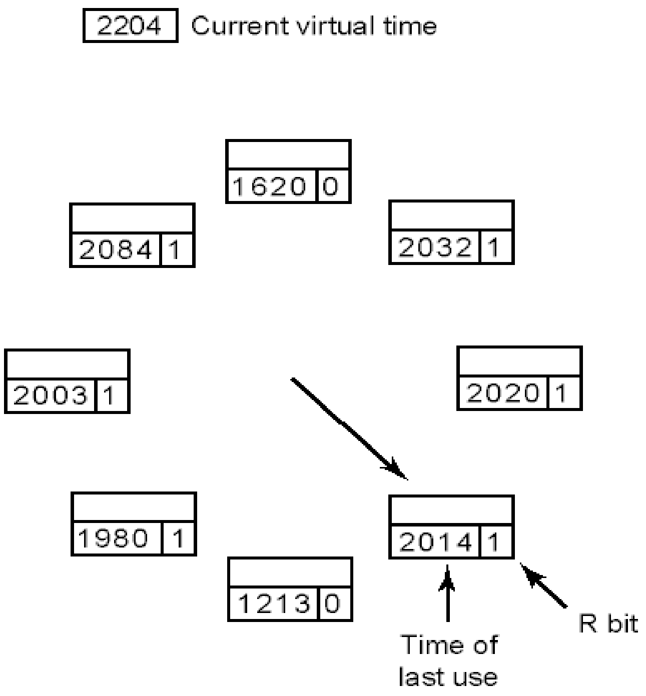
Clock+Time Algorithm

- Upon a page fault, if eviction is needed
  1. Look at the page with the arrow
  2. If R=1, advance arrow, go to Line 1
  3. If R=0 AND (current virtual time)-(time of last use)<k advance arrow, go to Line 1



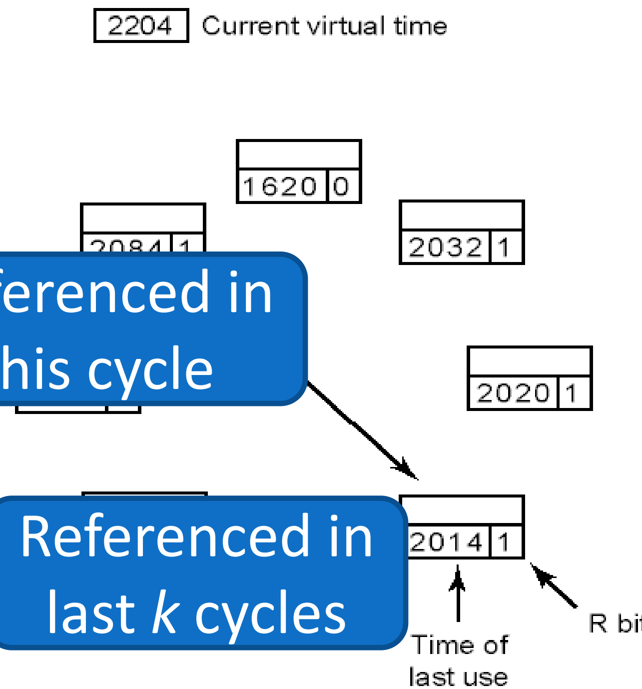
Clock+Time Algorithm

- Upon a page fault, if eviction is needed
  1. Look at the page with the arrow
  2. If R=1, advance arrow, go to Line 1
  3. If R=0 AND (current virtual time)-(time of last use)<*k* advance arrow, go to Line 1
  4. Otherwise, evict page



Clock+Time Algorithm

- Upon a page fault, if eviction is needed
  1. Look at the page with the arrow
  2. If R=1, advance arrow, go to Line 1
  3. If R=0 AND (current virtual time)-(time of last use)<*k* advance arrow, go to Line 1
  4. Otherwise, evict page



Evicting dirty and clean pages

Evicting dirty and clean pages

- Dirty page: A page that was modified since the last time it was written to the disk

Evicting dirty and clean pages

- Dirty page: A page that was modified since the last time it was written to the disk
  - The copy in the disk is different than the copy in the main memory (write back)

Evicting dirty and clean pages

- Dirty page: A page that was modified since the last time it was written to the disk
  - The copy in the disk is different than the copy in the main memory (write back)
  - Evicting dirty page → waiting to write the page to the disk and then waiting for the new page to the memory

Evicting dirty and clean pages

- Dirty page: A page that was modified since the last time it was written to the disk
  - The copy in the disk is different than the copy in the main memory (write back)
  - Evicting dirty page → waiting to write the page to the disk and then waiting for the new page to the memory
  - Evicting clean page → just waiting for the new page to the memory (safe to override)

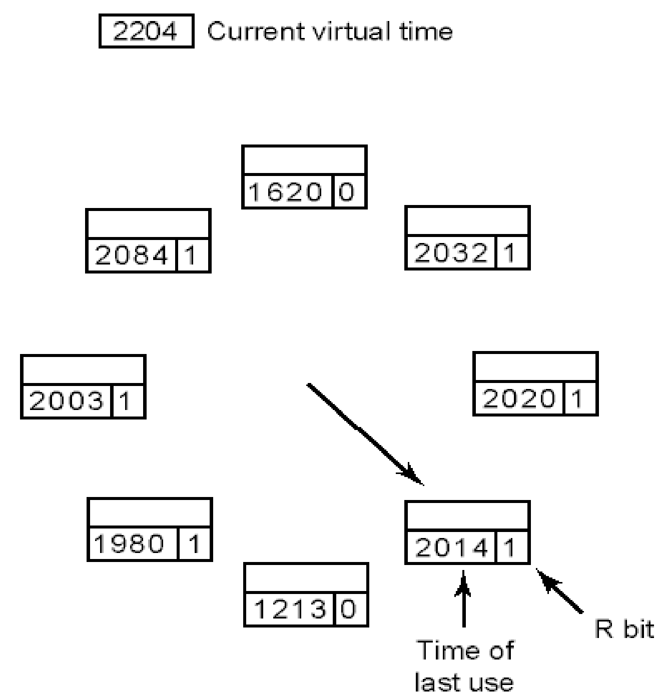
Evicting dirty and clean pages

- Dirty page: A page that was modified since the last time it was written to the disk
  - The copy in the disk is different than the copy in the main memory (write back)
  - Evicting dirty page → waiting to write the page to the disk and then waiting for the new page to the memory
  - Evicting clean page → just waiting for the new page to the memory (safe to override)

→ Prefer to replace clean pages

Improved Clock+Time Algorithm

- Upon a page fault, if eviction is needed
  1. Look at the page with the arrow
  2. If R=1, advance arrow, go to Line 1
  3. If R=0 AND (current virtual time)-(time of last use)<*k* advance arrow, go to Line 1
  4. If dirty, advance arrow, go to Line 1
  5. Otherwise, evict page

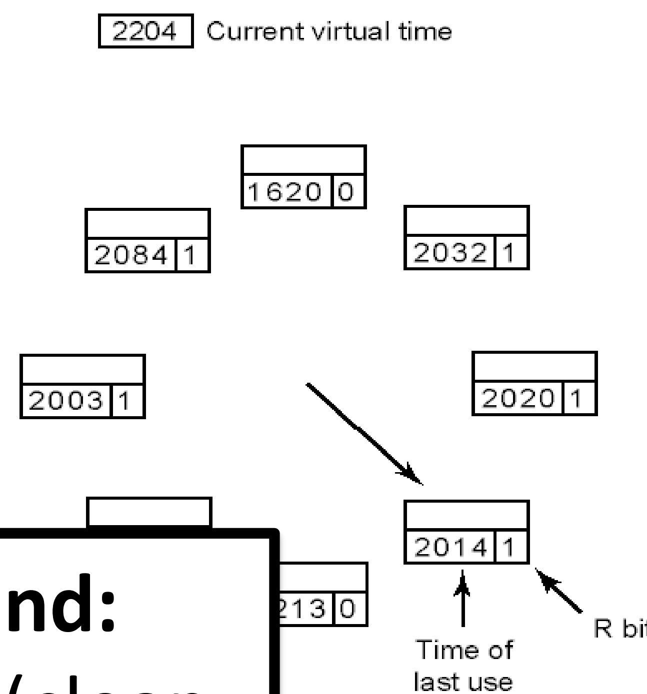


Improved Clock+Time Algorithm

- Upon a page fault, if eviction is needed
  1. Look at the page with the arrow
  2. If R=1, advance arrow, go to L
  3. If R (cu t use)<*k* adv
  4. If dirty, adv e arrow, go to Line 1
  5. Otherwise, evict page

If no candidate is found:

Evict the oldest page (clean or dirty)



Improved Clock+Time Algorithm

- Upon a pag
  eviction is

  1. Look at t
     the arro
  2. If R=1, a
     to Line 1
  3. If R=0 AND
     (current virtual -(time of last use)<*k*
     advance arrow, go Line 1
  4. If dirty, advance arrow, go to Line 1
  5. Otherwise, evict page

If dirty but old (more than k): schedule for eviction; the page is written to the disk in parallel to the process actions (using DMA)

20 1

R bit

last use

Global vs. Local Paging

- When process P1 causes a page fault, should the OS choose a victim from one of P1 pages or from any process?
- Some operating systems have local paging (only from P1 pages) and some global ("best" candidate to evict)