Memory Management

memory management

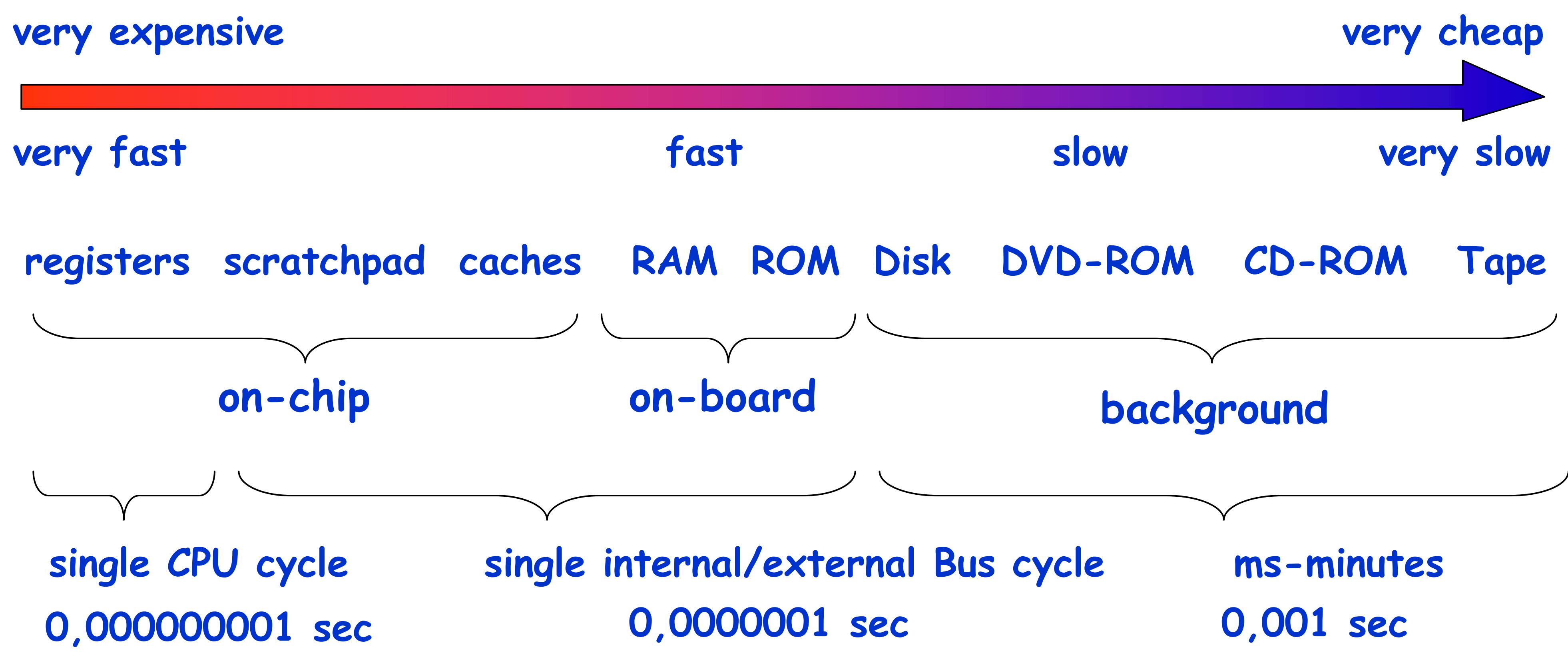
Why should we want memory management?
What characteristics do we want a memory should have?

- infinitely large,
- infinitely fast,
- infinitely cheap,
- nonvolatile.

Memory management aims to approximate these goals !

memory management

the cost/performance perspective:



memory management

registers
scratchpad

under explicit program/compiler control

caches
RAM
ROM
Disk
DVD-ROM
CD-ROM
Tape

hardware controlled

random access
memory abstraction
volatile

file abstraction

persistent

under operating
system control

memory management

Issues in memory management:

- ☆ transparent access through the hierarchy of memories
  - logical organization
  - physical organization

- ☆ Relocation
- ☆ Sharing
- ☆ Protection

} results from multiple programs/processes

multiprogramming with memory partitions

static memory partitioning:

today used in (some) embedded systems, palmtops, PDAs, etc.

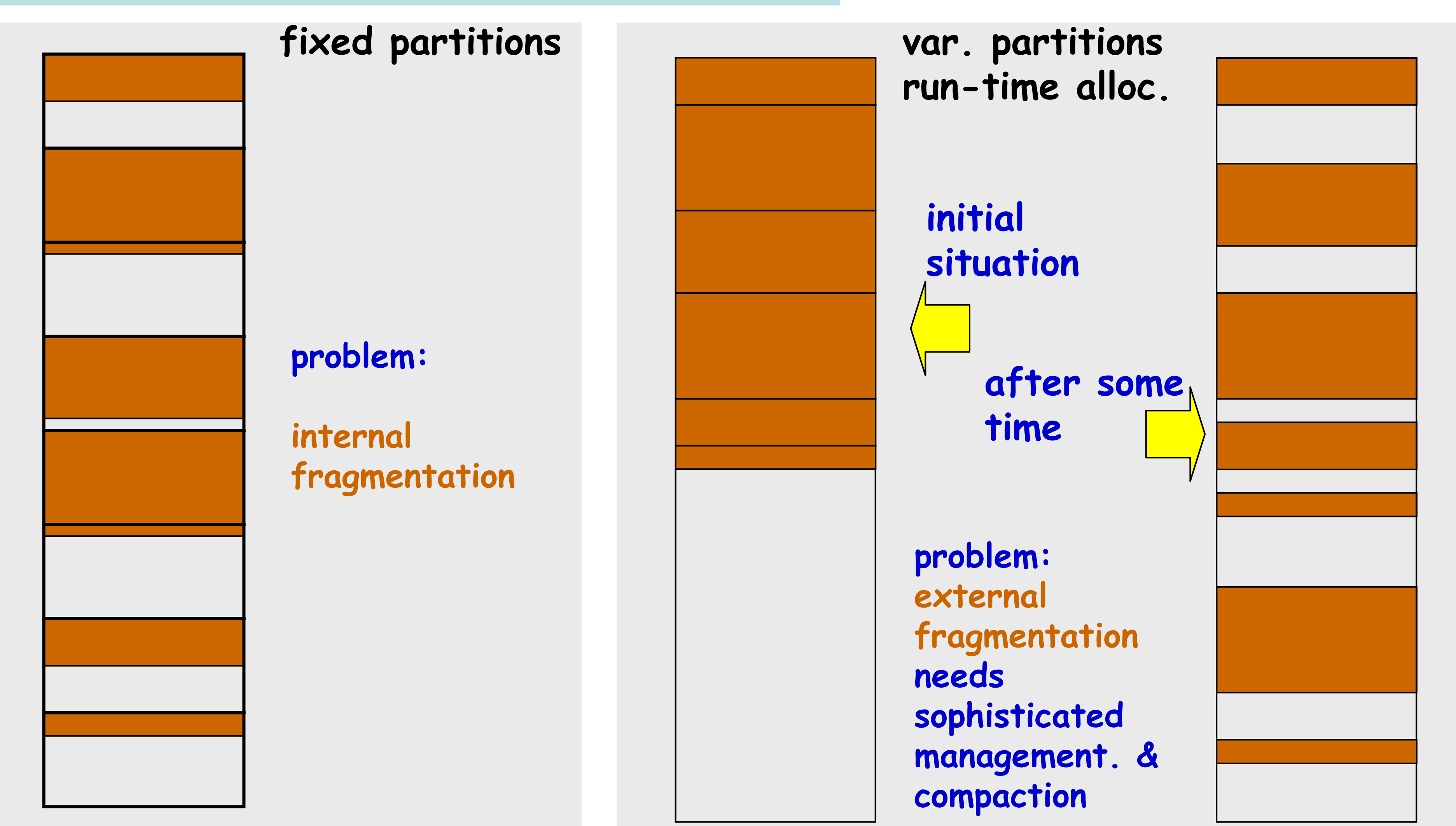
user program	OS in ROM	device drivers in ROM
	user program	user program
OS in RAM		OS in ROM

0xFFF...

0x00...

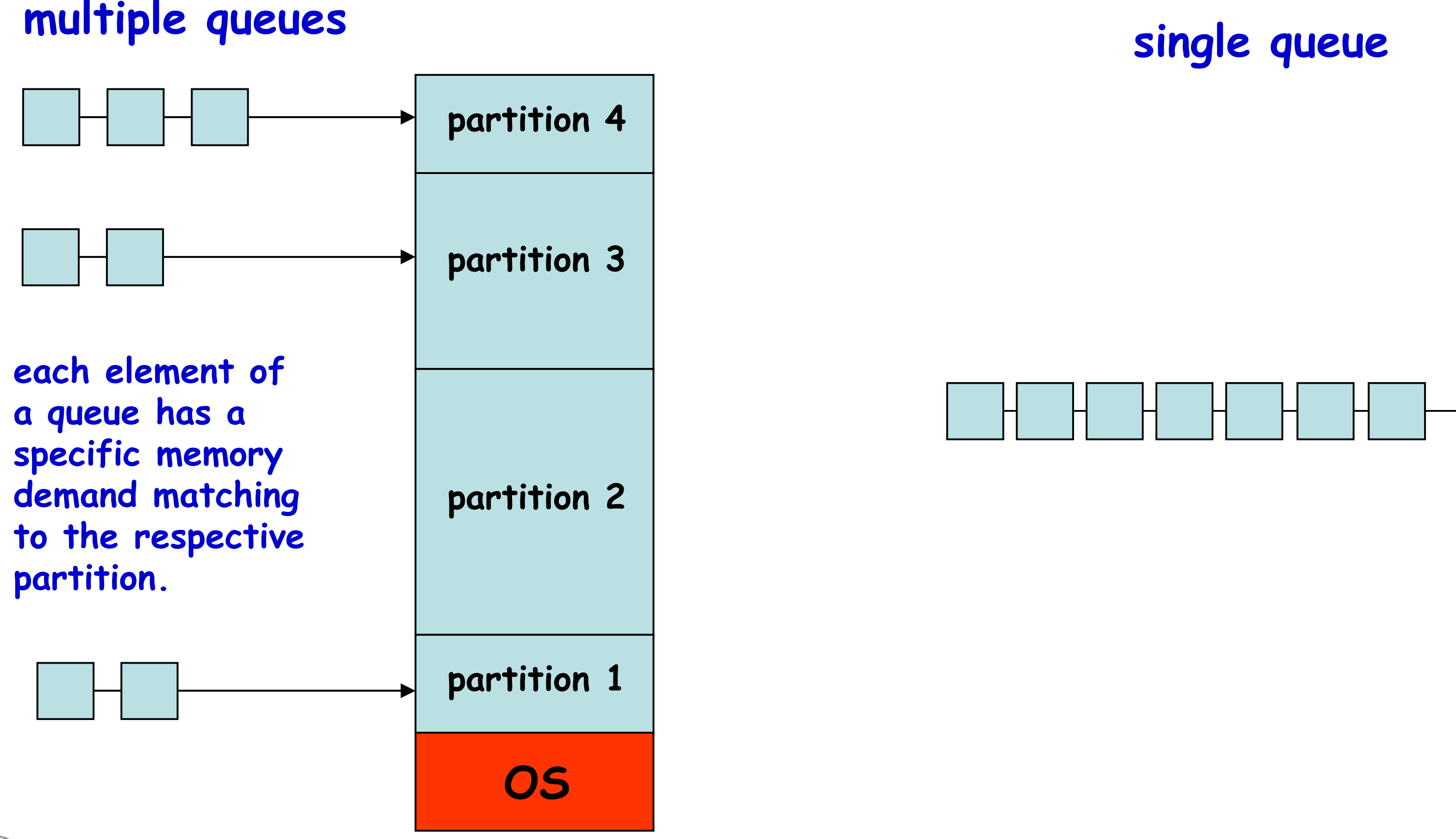
issues:	size of memory blocks	time of allocation
	fixed size variable size	at system generation (static) at run time (dynamic)

multiprogramming with memory partitions



multiprogramming with static memory partitions

how to manage fixed partitions?



multiprogramming with memory partitions

relocation and memory protection

Relocation:

Problem: Programs have to work on arbitrary memory addresses.

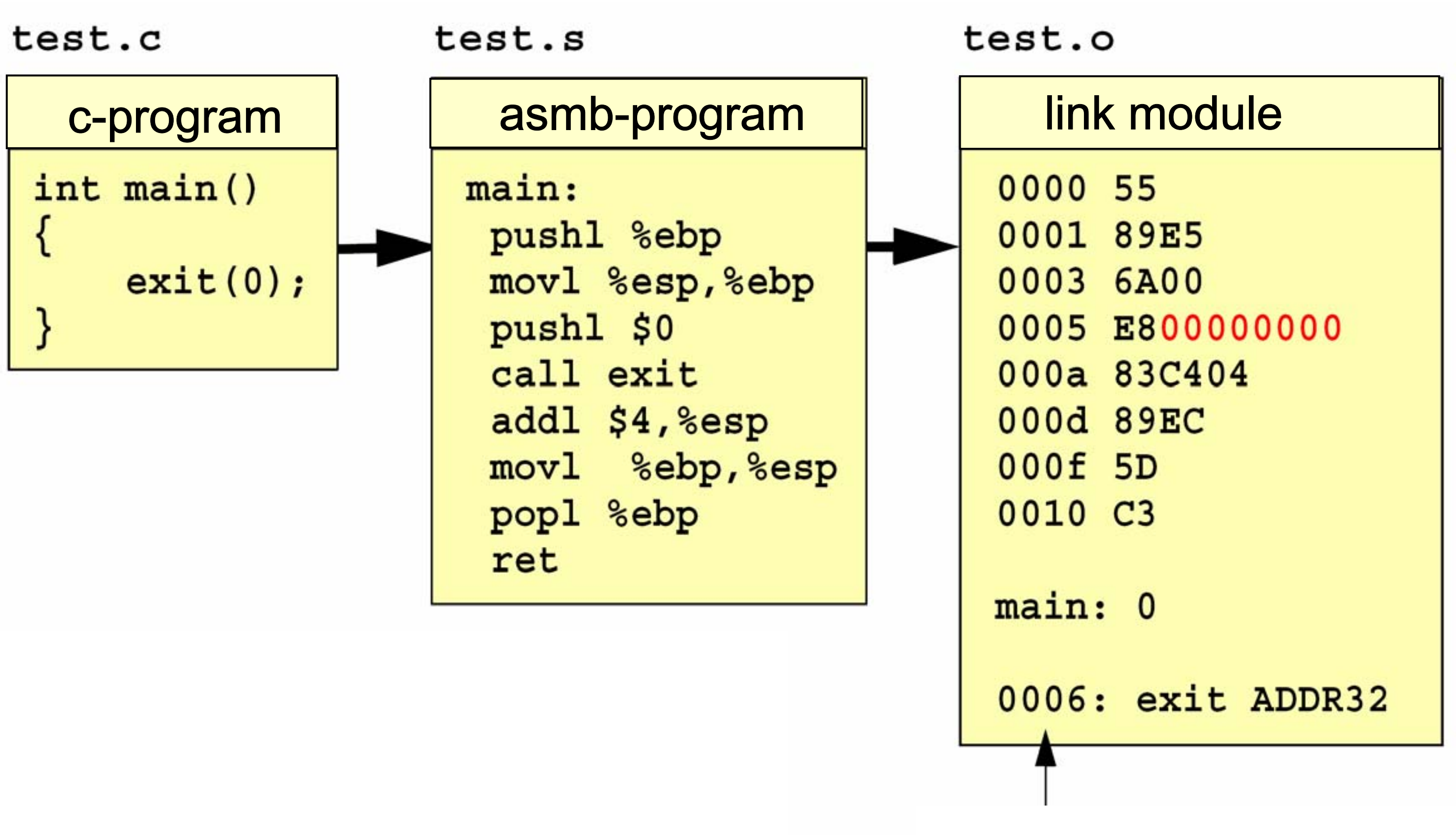
Mechanisms:
1. static binding of memory addresses during compile time ☹
2. Relocation when loading the program into memory → loader/linker
3. Relocation during run-time → needs position independent code → usually needs architectural support.

Protection:

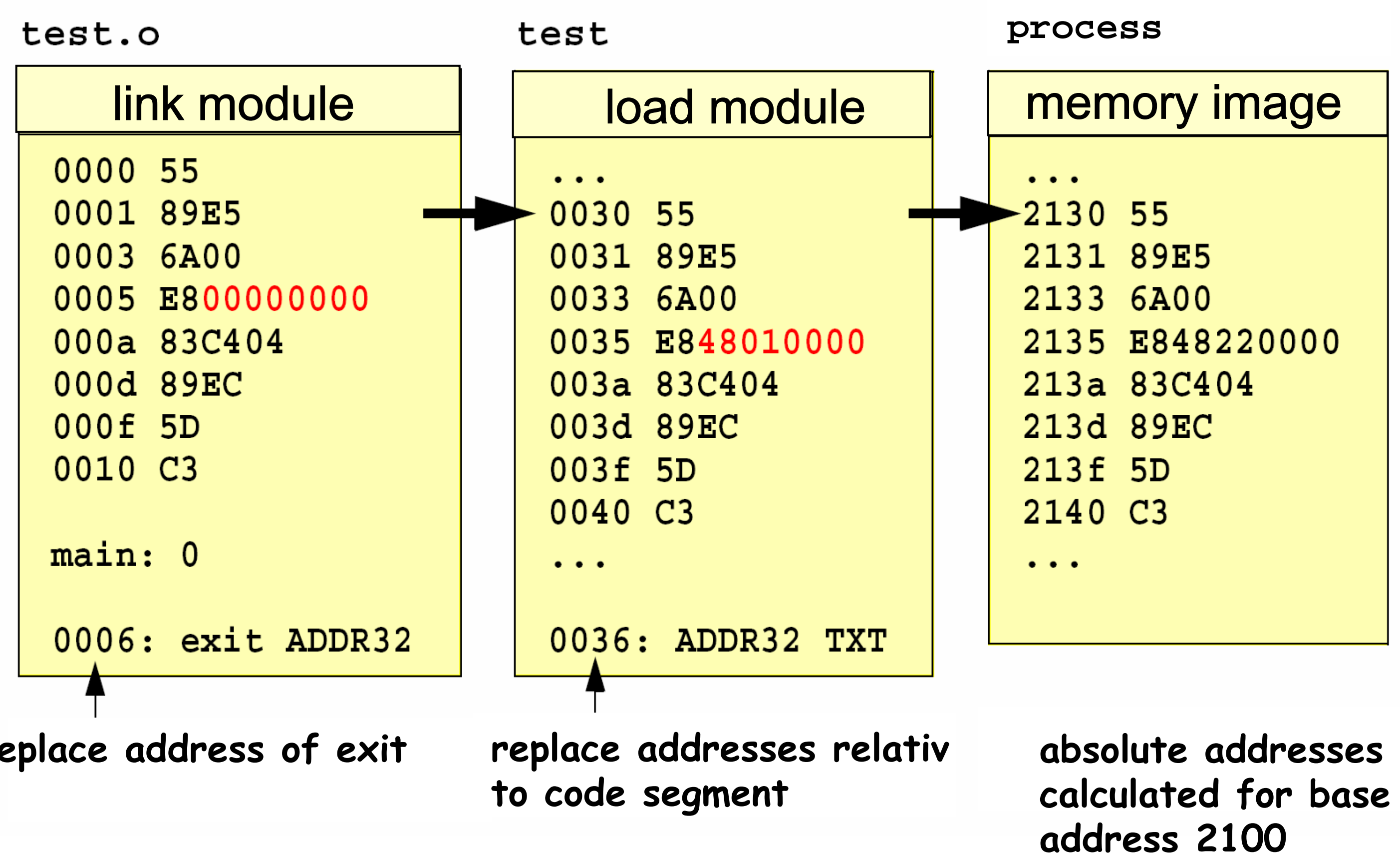
Problem: Arbitrary references to outside of the partition

Mechanisms:
1. Fixed length memory blocks tagged with 4-bit protection code. Check p-code against a field in the program-status-register (IBM 360)
2. Base and bound registers (CDC 6600)

relocation

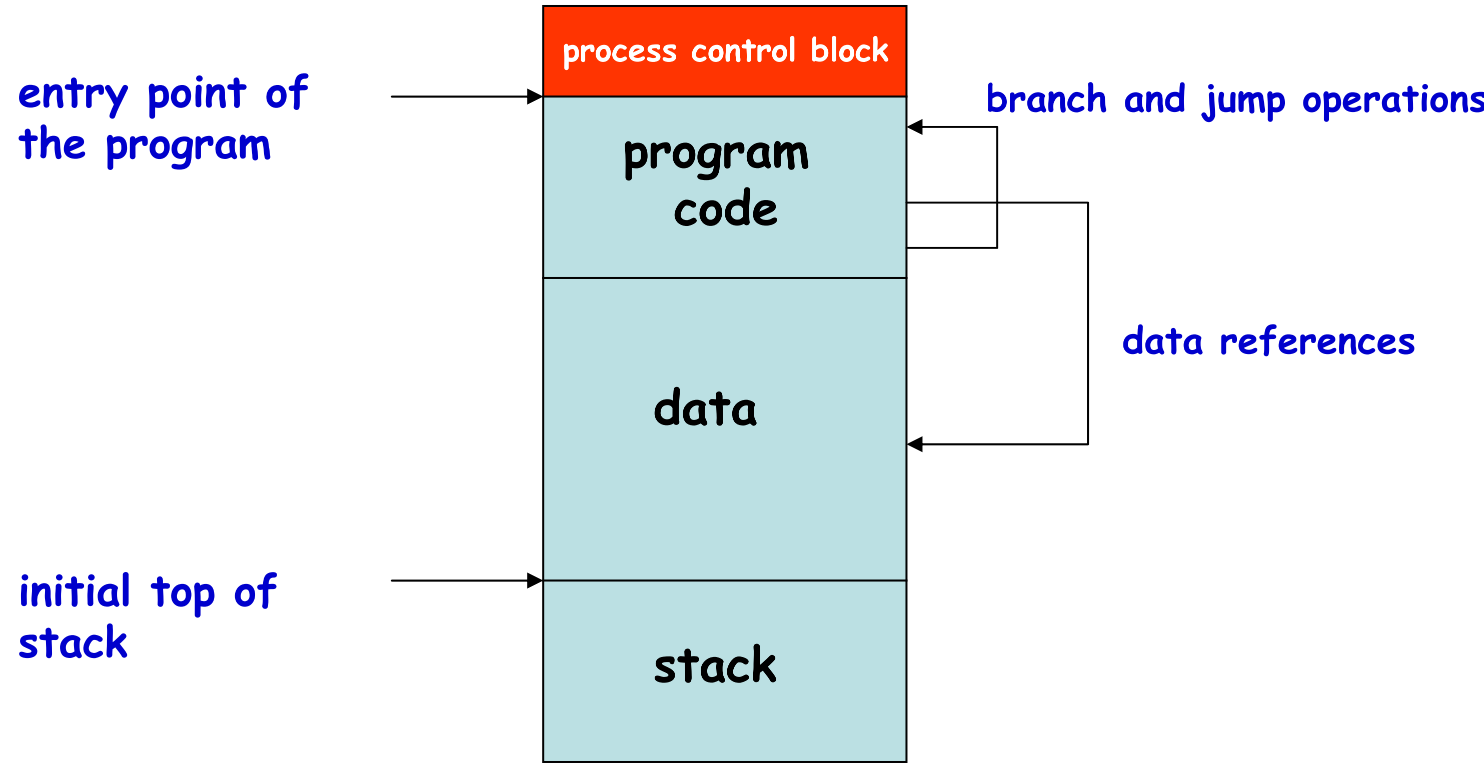


relocation

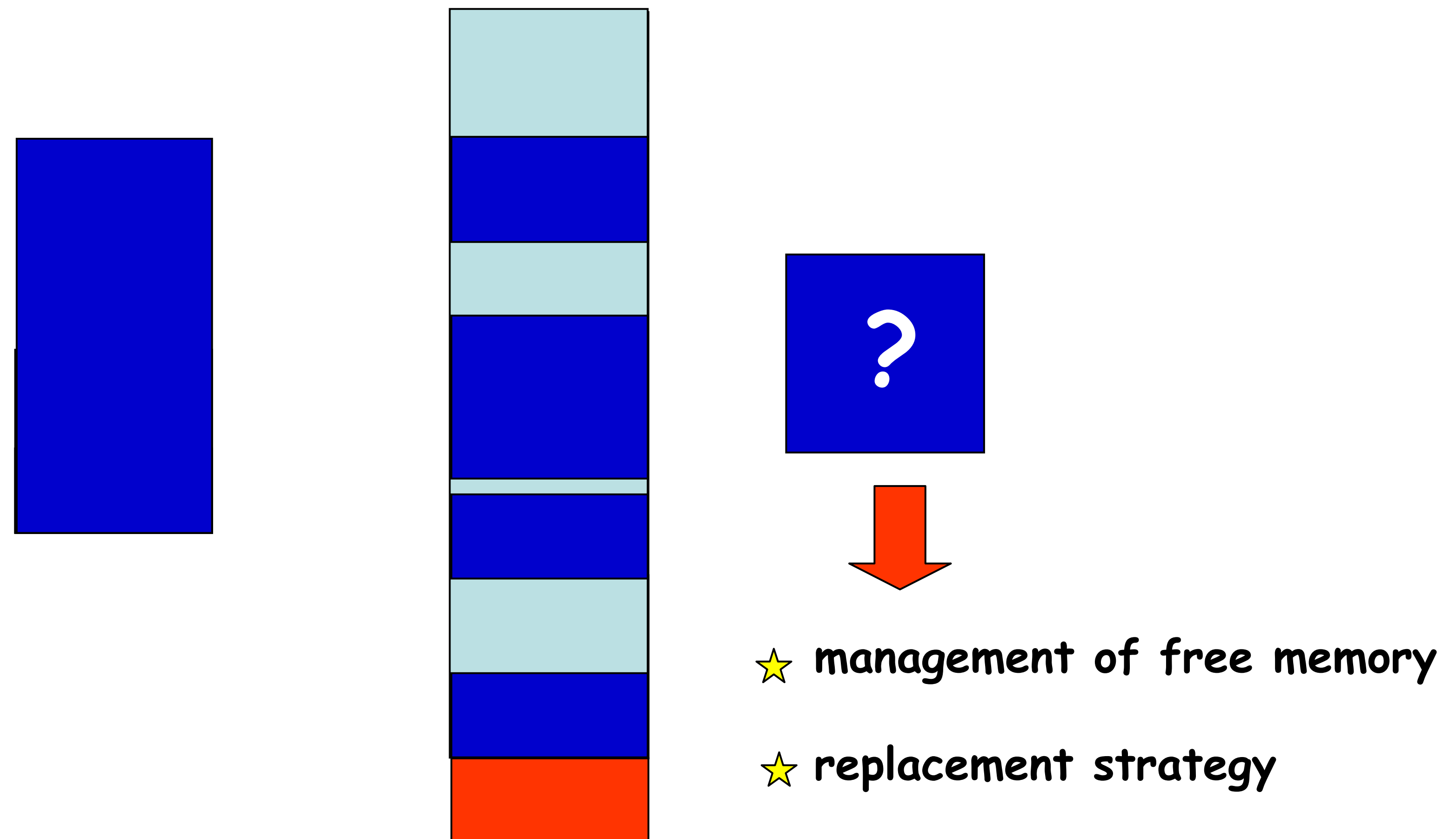


dynamic partitions and swapping

swapping: each proces is loaded into memory or swapped out to disk completely with its code, data and stack parts.

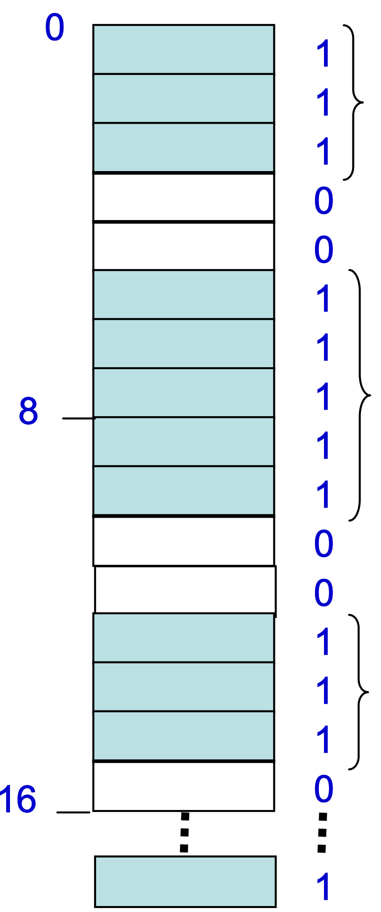


dynamic partitions and swapping



dynamic partition management

basic problem: where are free memory blocks of what size ?

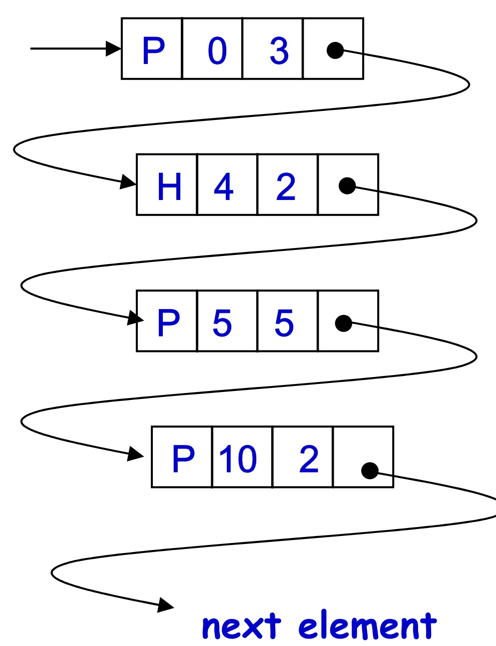


bitmap representation:

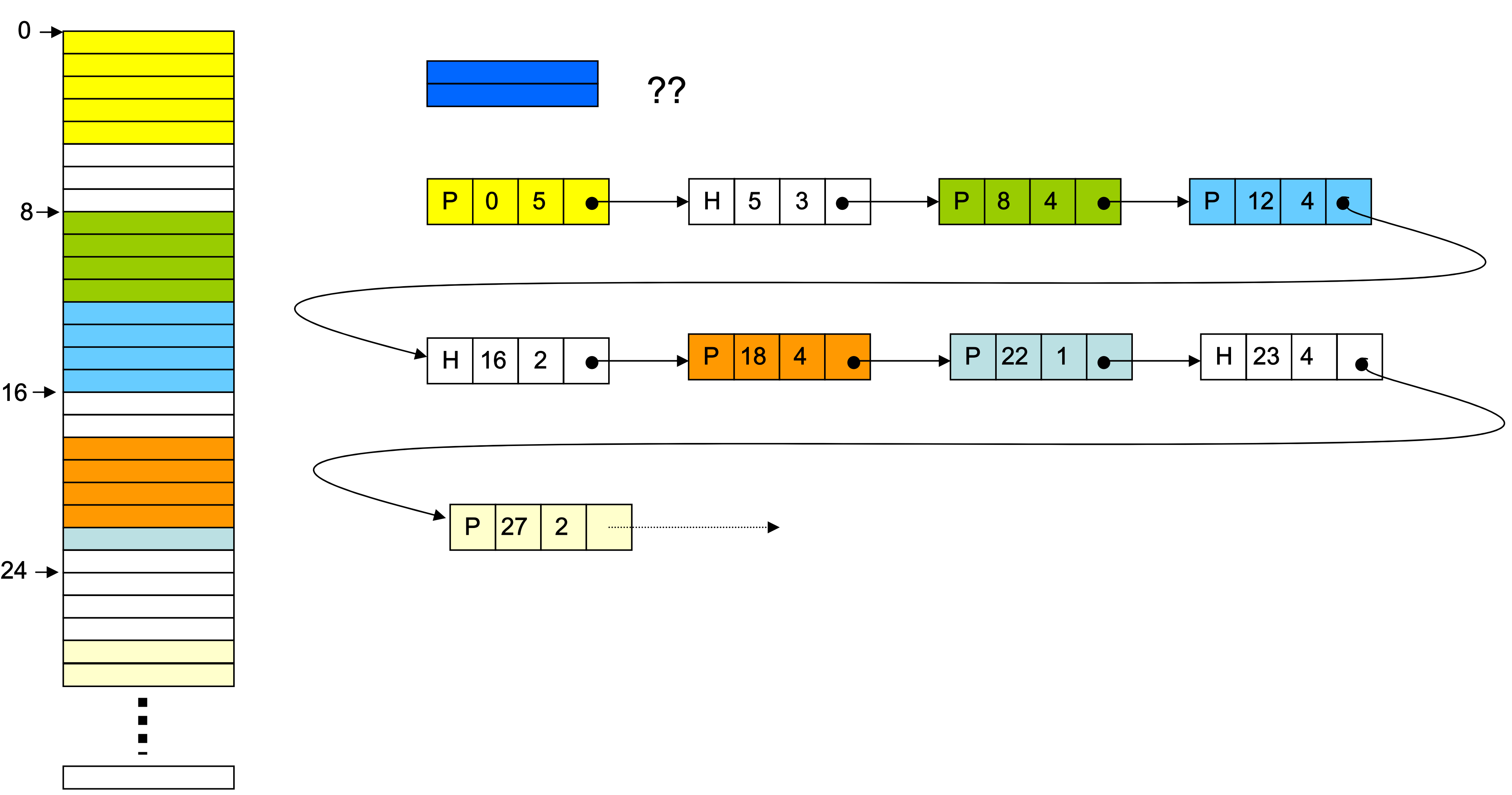
1110011
1110011
10.........

1 = allocated block
0 = hole

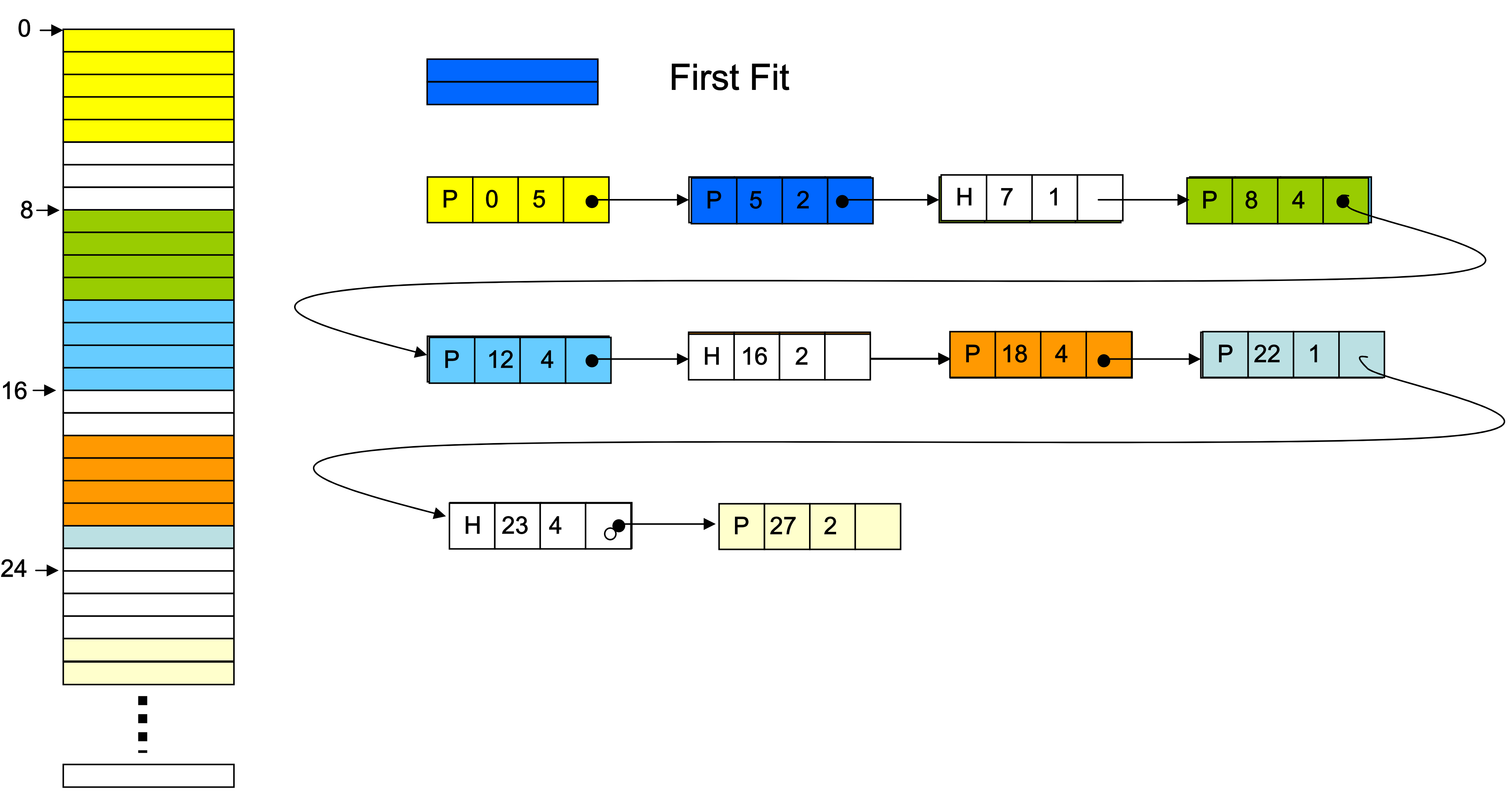
linked list representation:



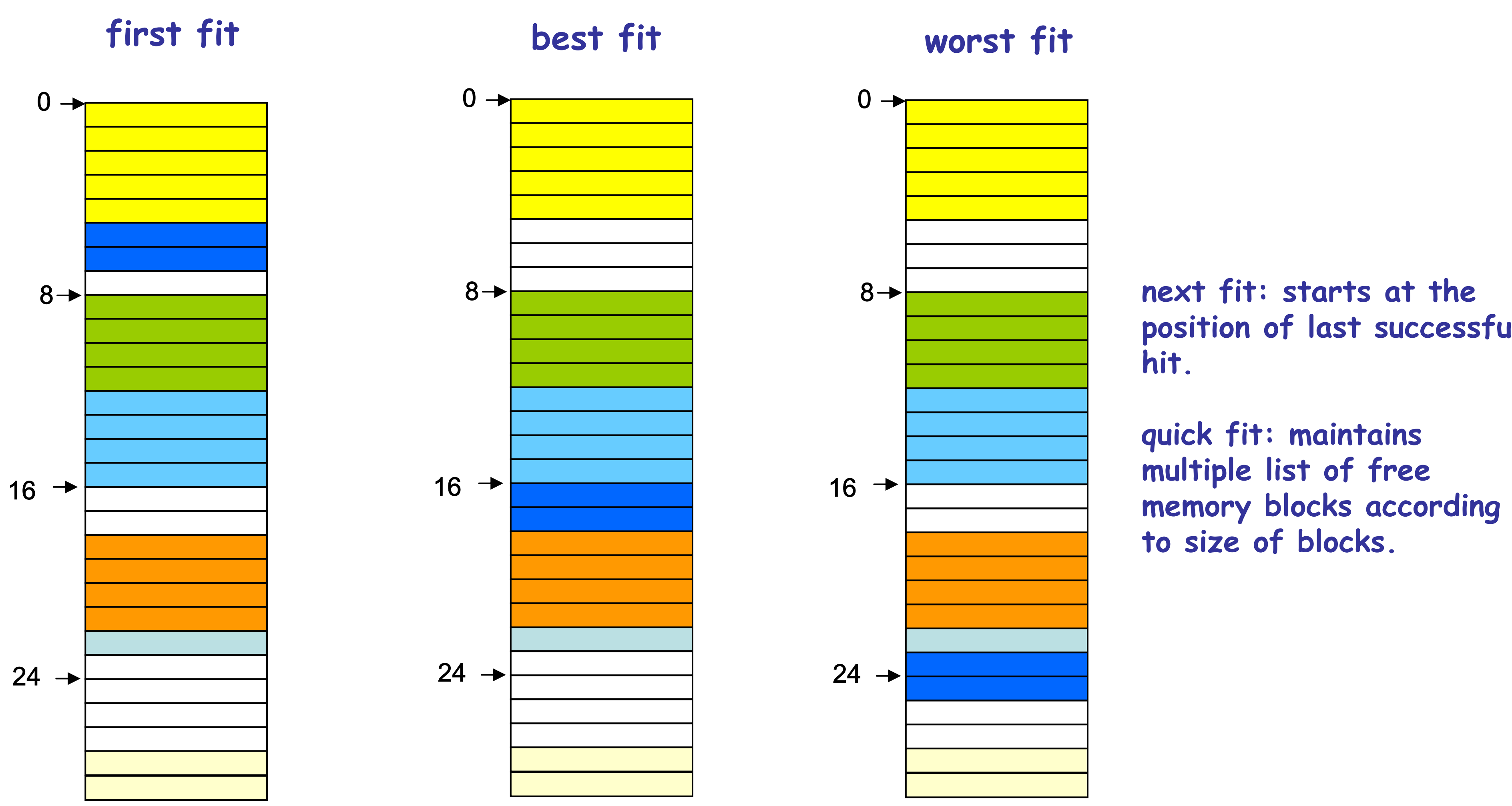
dynamic partition management



dynamic partition management



dynamic partition management



next fit: starts at the position of last successful hit.

quick fit: maintains multiple list of free memory blocks according to size of blocks.

the Buddy-System

Operation	Memory blocks
initial 1M block	1 M
request 100 k	128 k, 128 k, 256 k, 512 k
request 240 k	128 k, 128 k, 256 k, 512 k
request 60 k	128 k, 64 k, 64 k, 256 k, 512 k
request 250 k	128 k, 64 k, 64 k, 256 k, 256 k, 256 k
release	128 k, 64 k, 64 k, 256 k, 256 k, 256 k
release	128 k, 64 k, 64 k, 256 k, 256 k, 256 k
request 75 k	128 k, 64 k, 64 k, 256 k, 256 k, 256 k
release	128 k, 128 k, 256 k, 256 k, 256 k
release	512 k, 256 k, 256 k
release	1 M

discussion:

Mechanisms so far are characterized by:

1. Management of the real, physical memory.
2. Address space covers the size of physical memory.
3. Swapping to background disk is managed explicitely by application/OS.
4. Size of entities which are swapped are determined by application specs.

Still a problem:

- Programs which are larger than the available memory.
- Protection when multiple processes running in a single memory space.

discussion:

Address space of processor:

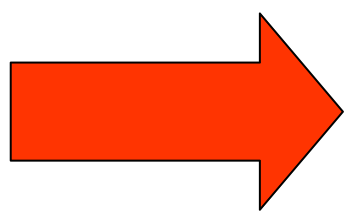
32 Bit = 4.294.967.296

64 Bit = 18.446.821.383.201.879.616 ~ 2 x 10**19

Idea:
View physical memory as a window to a much larger memory space.
Separation between physical address space and logical address space.

Desirable goals for a memory management scheme are:

1. Transparent mechanism of loading/replacing of memory blocks.
2. Address space larger than physical memory space.
3. Transparent relocation mechanism.
4. Better protection by separating address spaces.



virtual memory

The Computer Journal

Vol. 4, Issue 3, October 1961

virtual memory also described in:

John Fotheringham: Dynamic Storage Allocation in the ATLAS Including Automatic Use of Backing Store

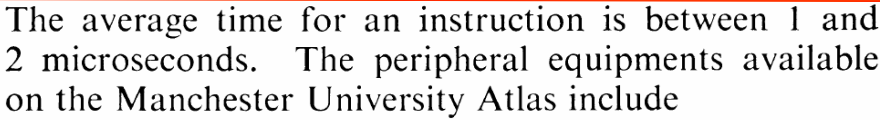
Communications of the ACM, Volume 4 , Issue 10 (October 1961)

The Manchester University Atlas Operating System Part I: Internal Organization

By T. Kilburn, D. J. Howarth, R. B. Payne and F. H. Sumner

Introduction

Atlas* is the name given to a comprehensive computer system designed by a joint team of Ferranti Ltd. and Manchester University engineers. The computer system comprises the central computer, fixed store, core store, magnetic drum store, magnetic tapes, and a large quantity and variety of peripheral equipments for input and output. The Manchester University Atlas has 32 blocks of core store each of 512 forty-eight bit words. There is also a magnetic drum store, and transfers between core and drum stores are performed automatically, giving an effective one-level store* of over two hundred blocks. The average time for an instruction is between 1 and 2 microseconds. The peripheral equipments available on the Manchester University Atlas include



8 magnetic tape decks	90,000 characters per second
4 paper tape readers	300 characters per second
4 paper tape punches	110 characters per second
1 line printer	600 lines per minute
1 card reader	600 cards per minute
1 card punch	100 cards per minute

virtual memory - history

The Atlas team at the University of Manchester created the first version of virtual memory in 1959. They called it One-Level Storage System.

There major achievements were:

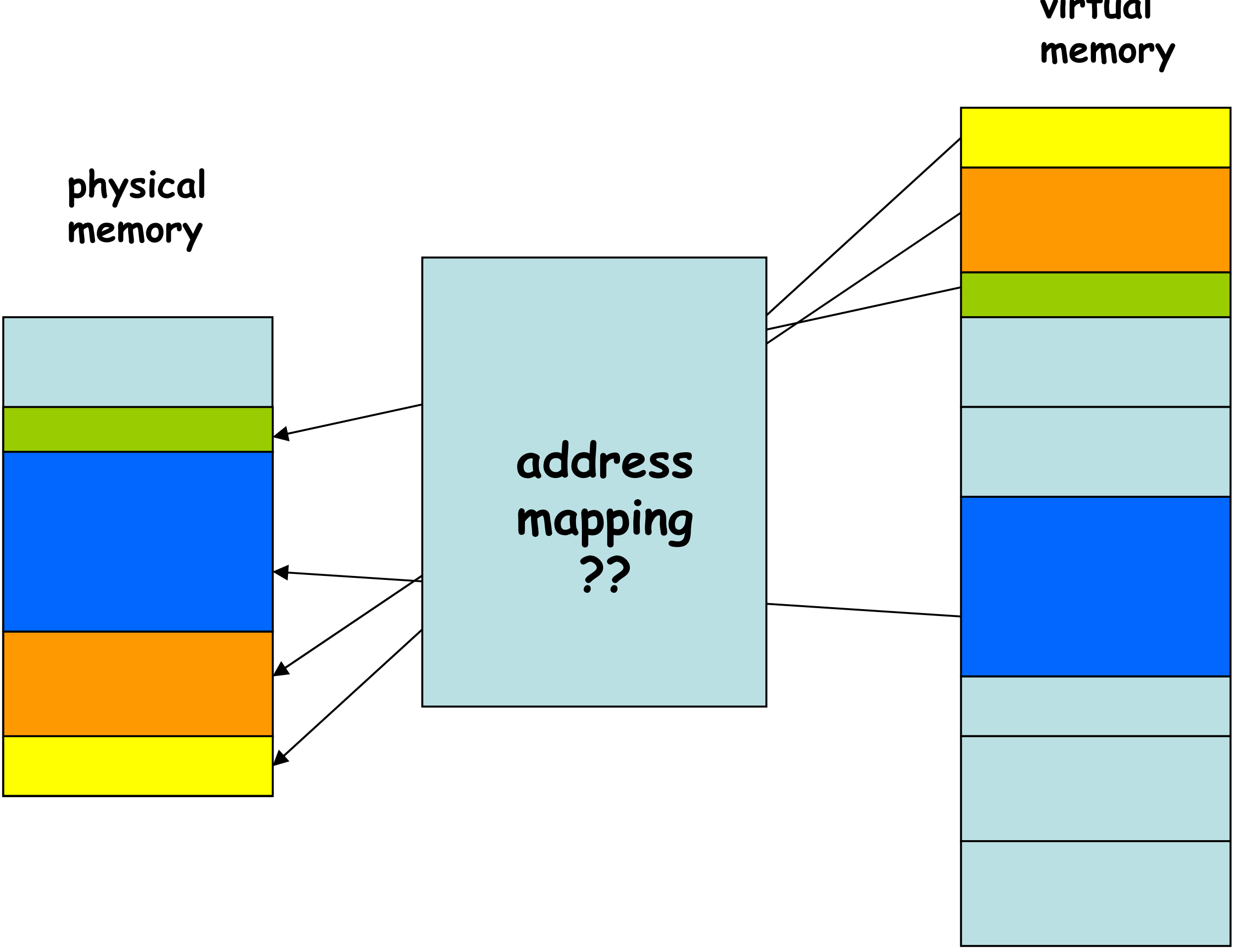
- Separation between Logical and Physical address space.
- Introduction of fixed size pages and frames.
- Hardware support for address translation.
- Demand paging
- A replacement algorithm

1960's: all popular commercial operating systems used virtual memory e.g. (Multics), IBM 360/67, CDC 7600, GE 645, RCA Spectra/70, Burroughs 6500.

1970s: IBM 370, DEC VMS, DEC TENEX, and Unix .

1995: Microsoft included virtual memory in Windows 95.

virtual memory



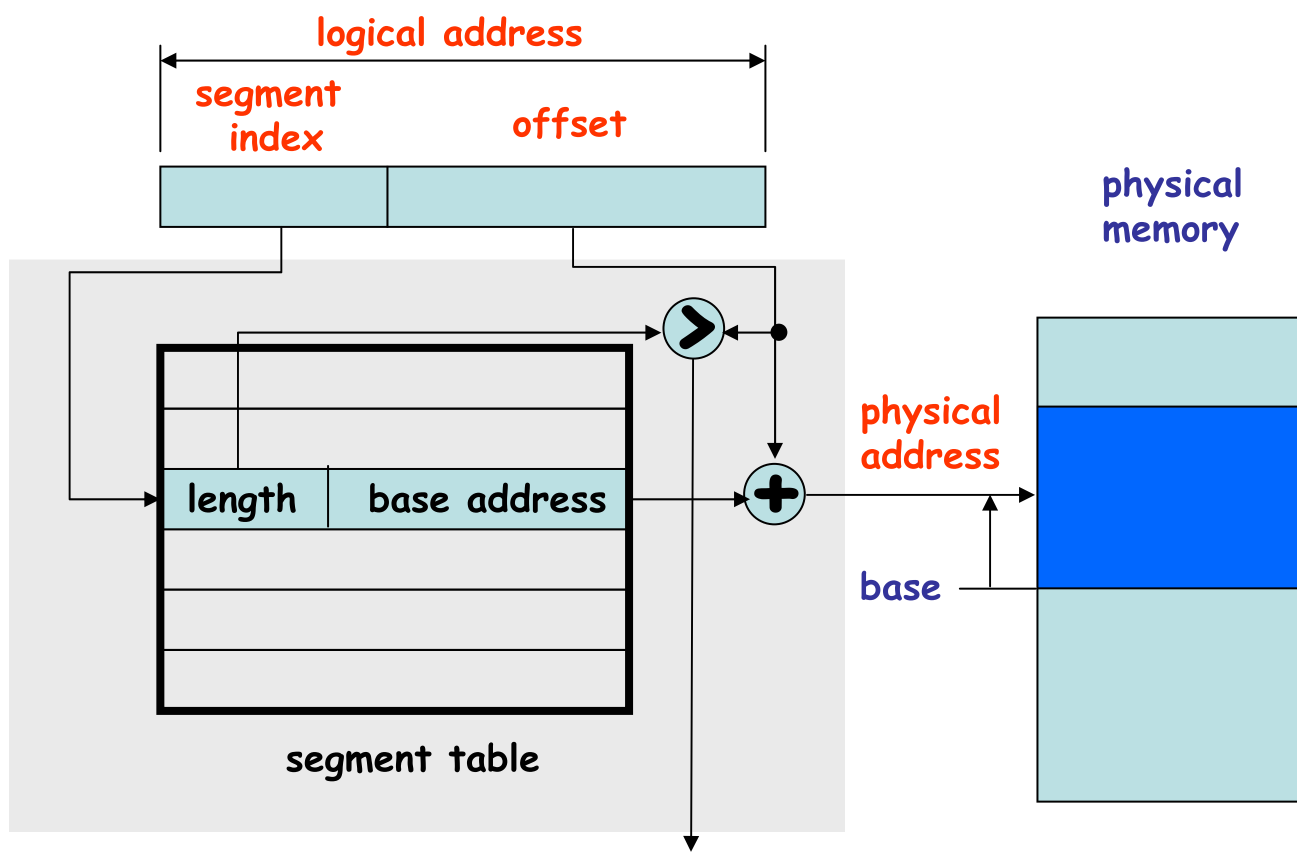
"segmented" virtual memory

virtual address space

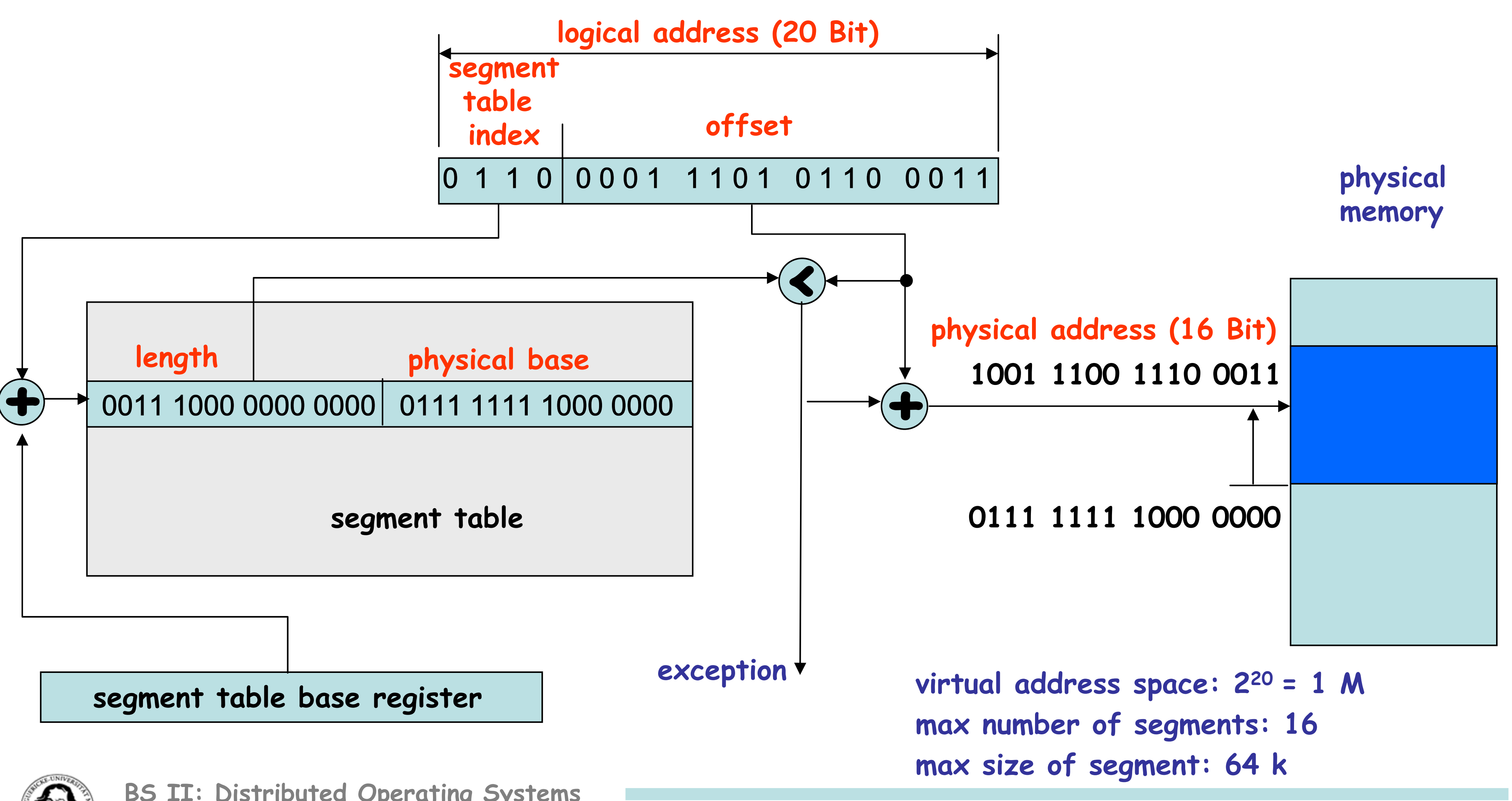
segments

offset

0



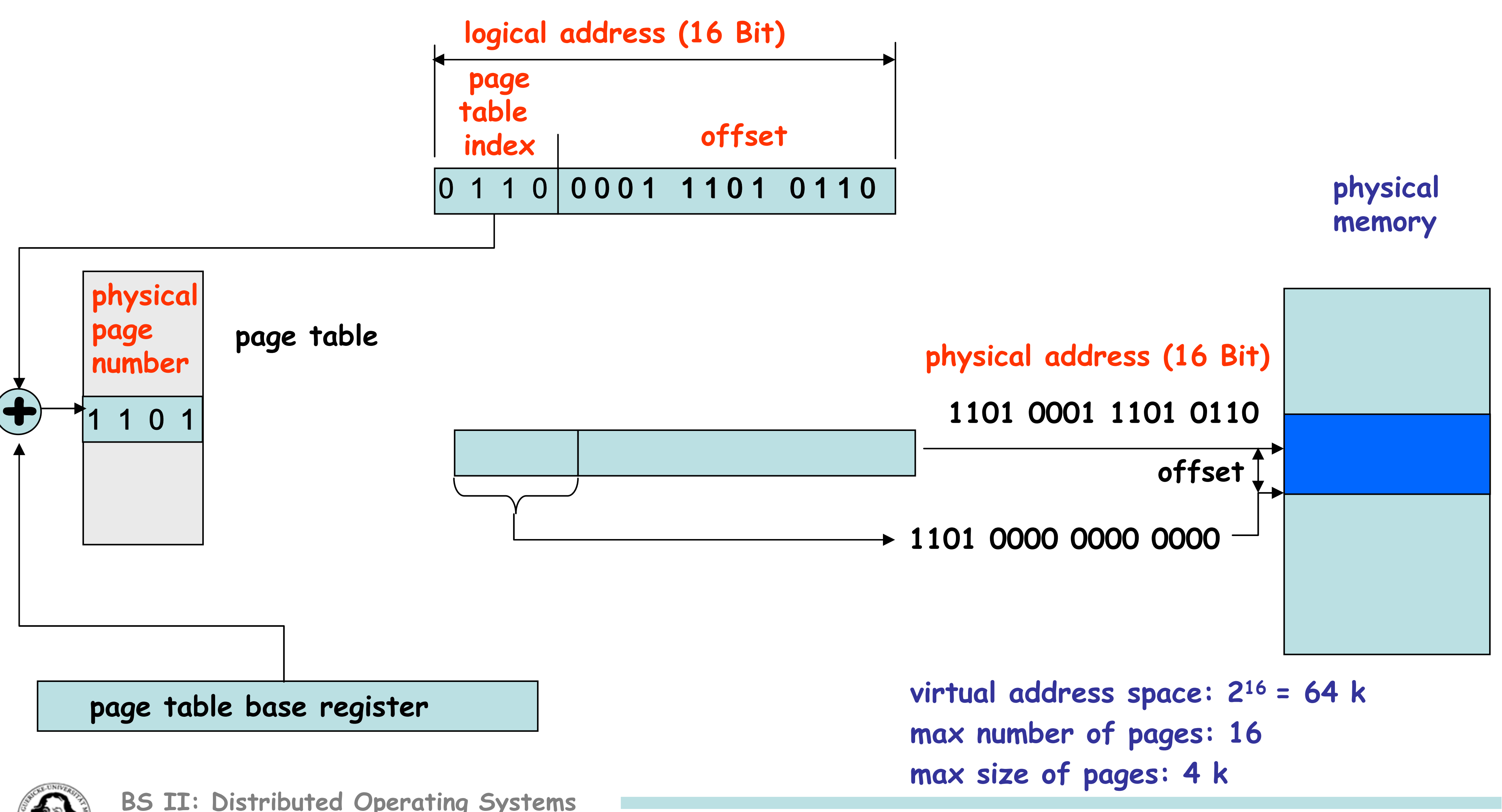
virtual memory - address translation



page-based virtual memory

Idea: fixed size virtual pages are mapped to fixed size physical frames.

virtual memory - page translation



virtual address space: 2^{16} = 64 k
max number of pages: 16
max size of pages: 4 k

structure of table entries

status	type	protection	privilege	size/limit	segment base address

- status:
  - caching
  - referenced
  - dirty
- type:
  - code segment
  - data segment
  - special
- protection:
  - read/write/exec
- privilege:
  - privilege level
- size/limit:
  - size in multiples of bytes or in pages (e.g. Pentium II)

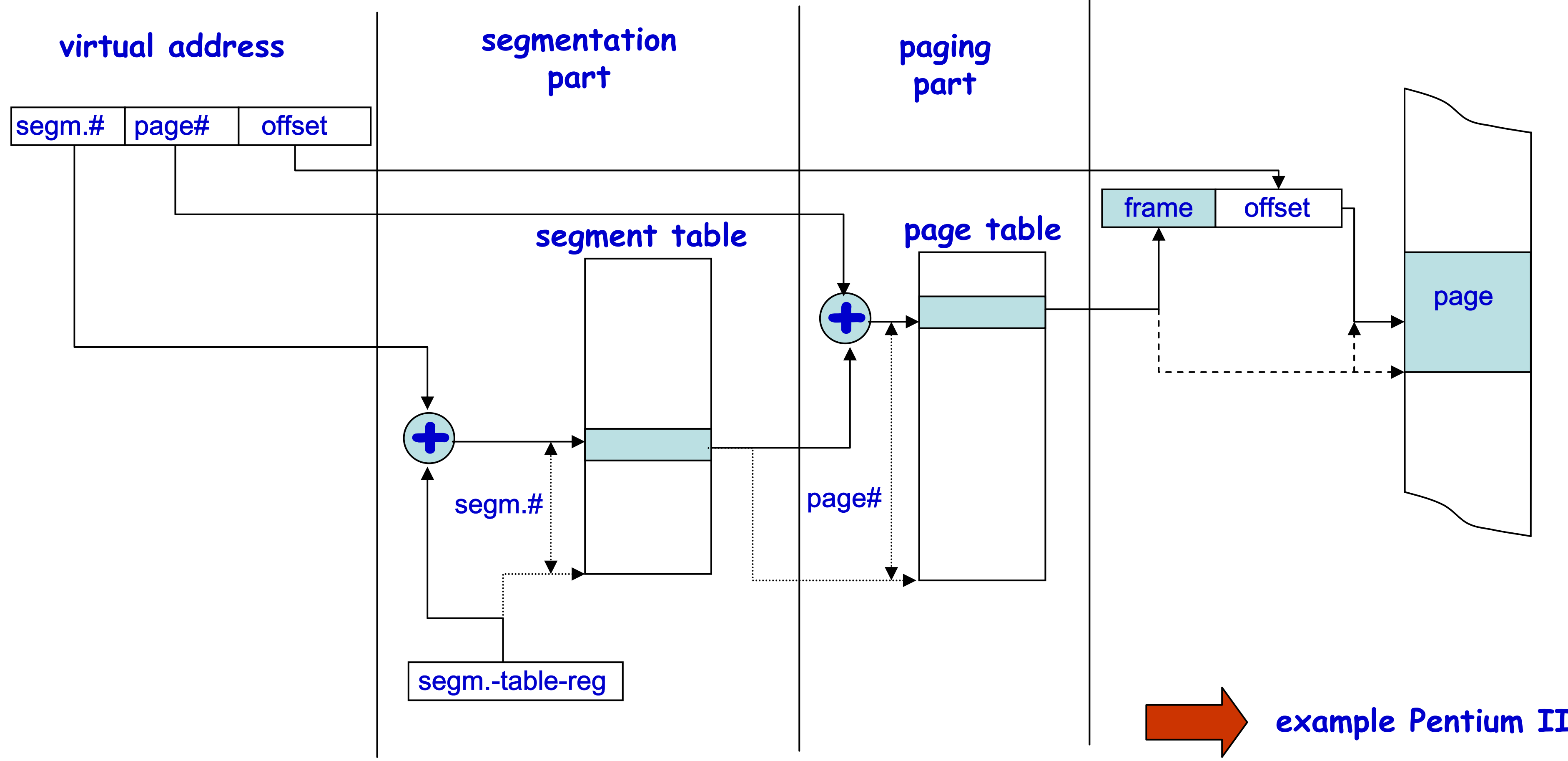
segment descriptor

discussion: segmentation vs. paging

	transparent for progr.	number of addr. spaces	virtual addr. space > real memory	variable size obj.	frag- ment.	mgnt overhead	main reason for invention
paging	☺	single	☺	☹	internal	☺	infinite memory
segmentation	☹	many	☺	☺	external	☹	multiple addr. spaces

Why not combining the advantages by:
- exploiting segmentation to define variable size objects and protect them;
- exploiting paging to provide a large, easy to manage address space?

combining segmentation and paging



paging - discussion

- size of the page table?
- are all pages of a process needed in physical memory all the time ?
- what pages then are in and out of physical memory?
- what could be a good scheme to replace pages in physical memory?

size of the page table

32-Bit Address Space: 4G Addresses — Page table size @ 4k: 1M @ 4k entries

64-Bit Address Space: 4G·4G Addresses — Page table size @ 4k: 4G·1M entries

1. increase page size: e.g. UltraSPARC II supports 8k, 64k, 512k and 4M pages
   rational: less pages to map
   problem: internal fragmentation; still a problem with 64-Bit-Addr.Space.

2. page the page tables: multi-level page table structure
   rational: a.)virtual memory is cheap, b.) only a very small set of page tables are needed at a time (→ working set)
   problem: another level of indirection and management

3. map physical to virtual memory not virtual to physical: inverted page tables
   rational: physical memory is small compared to virtual address space.
   problem: much more pages than entries in the table→mapping is ambiguous

multi-level hierachical page tables

principle

root page table, always in physical memory

page tables, may not be present in phys. mem.

pages

organization

10 bit	10 Bit	12 Bit

root

1024 entries

secondary

1024 entries

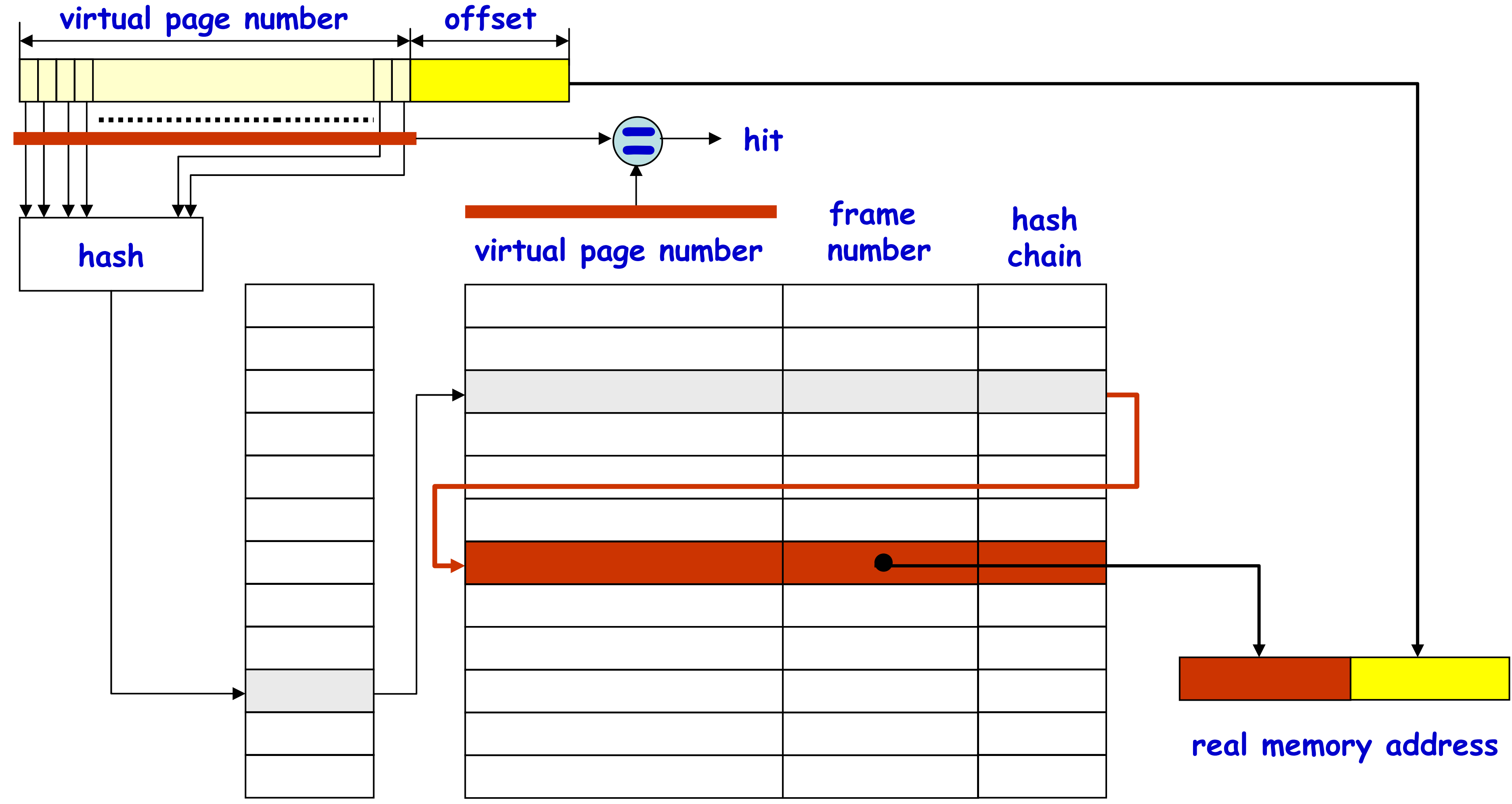
frame | offset

page

pt-base-reg

inverted page tables

used in e.g. PPC, AS/400



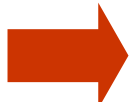
inverted/non inv. page tables - discussion

pro inverted:
page table is proportional to the physical memory:
e.g. for 1 G @ 8k pages : 128k entries independent of virtual addr.space

con inverted:
- hashing has to be performed with every memory access.
- chaining may incure multiple table accesses.
- misses lead to replacement of entries, complicated management needed.

but:
- using hierarchical page tables also require multiple table accesses.
- if secondary page tables are not in real memory, disk access needed.

➡ **architectural/harware support is needed in both schemes !**



virtual memory - address translation

virtual to physical address translation incurs some levels of indirection

- impact on performance !
- needs hardware support to speed up.

the locality principle

Locality principle:

1. Programs exhibit locality of references within a temporal window.
2. From the history of references one can predict future behaviour.

The locality principle is the basis for all chaching techniques!

Translation Lookaside Buffer

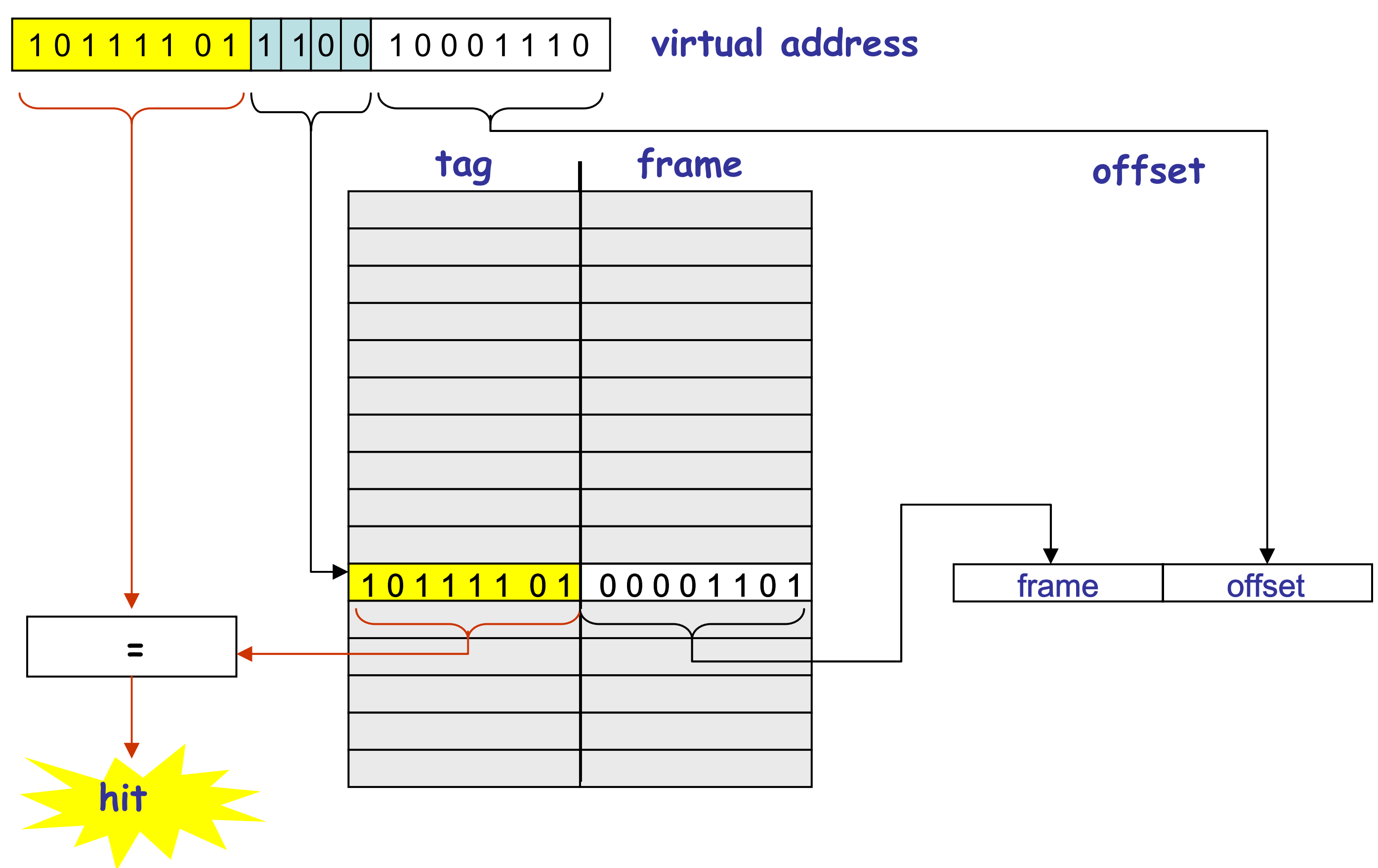
The Translation Lookaside Buffer TLB (sometimes also termed Address Translation Cache: ATC) is a cache for page or segment descriptors. It is in the critical address path of a CPU to memory including, in most cases, cache memory and is accessed with every memory instruction.

miss: exception handled by OS

key characteristics:

- size of TLB: 32 -4096 entries
- hash: full associative, set associative
- hit rate (typical): 99% - 99,99 %
- hit penalty: ~1 cycle
- miss penalty: 10 - 30 cycles

simple direct mapped TLB



simple 4-way set-associative TLB

virtual address

1 0 1 1 1 1 0 1	1	1	0	0	1 0 0 0 1 1 1 0

0 0 0 0 1 1 0 1	1 0 0 0 1 1 1 0

tag	frame	tag	frame	tag	frame	tag	frame
1 0 0 0 1 1 0 1	1 1 0 0 0 1 0 1	1 0 1 1 1 1 0 1	0 0 0 0 1 1 0 1	1 0 0 1 0 0 0 1	0 1 1 1 1 1 1 1	x x x x x x x x	x x x x x x x x

= = = =

hit

Combining segmentation and paging: The Pentium II address translation

0=GDT
1=LDT
privilege level

index
15
4
0

LDT-Reg.
GDT-Reg.

logical (segment) address
31 0

x 4096

bound
violation
exception

63
0
base addr.
32 bit
limit
20 bit
. . . . .

linear address
31 0

31
segment descriptor*
0

Base 0-15						Limit 0-15			
Bs. 24-31	G	D	0		L. 16-19	P	DPL	TYPE	Bs. 16-23

- G — 0: Limit in Bytes; 1: Limit in pages
- D — 0: 16 bit segm.; 1: 32 bit segm.
- P — 0: segm. not in mem.; 1: segm. in mem.
- DPL — privilege level 0-3
- TYPE — segm. type and protection

* for code segment

Combining segmentation and paging: The Pentium II address translation

31 22 21 12 11 0

page table dir.	page number	offset

linear address

page table

page base

page table dir. reg.

31 0

page base	offset

physical (real) address

Virtual memory organization in the SUN UltraSparc II

How to manage a 64-bit virtual address space?

64 bits

bits	51	13	48	16	45	19	42	22
virtual address	virt.8k pg-nr	offset	virt.64k pg-nr	offset	virt.512k pg-nr	offset	virt.4M pg-nr	offset

41 bits ~ 2200 Gbyte

bits	28	13	25	16	22	19	19	22
physical address	8k frame	offset	64k frame	offset	512k frame	offset	4M frame	offset

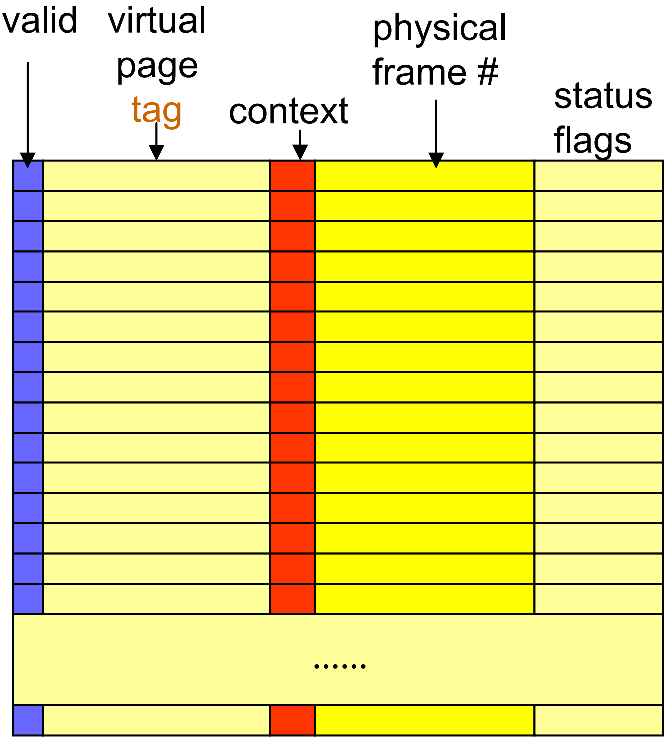
Data structures for translation in the SUN UltraSparc II

TLB (MMU-Hardware)

valid | virtual page # | context | physical frame # | status flags

......

TSB: Translation Storage Buffer (MMU+Software)



Translation Table

structure of entries is defined by the OS

Managing virtual memory

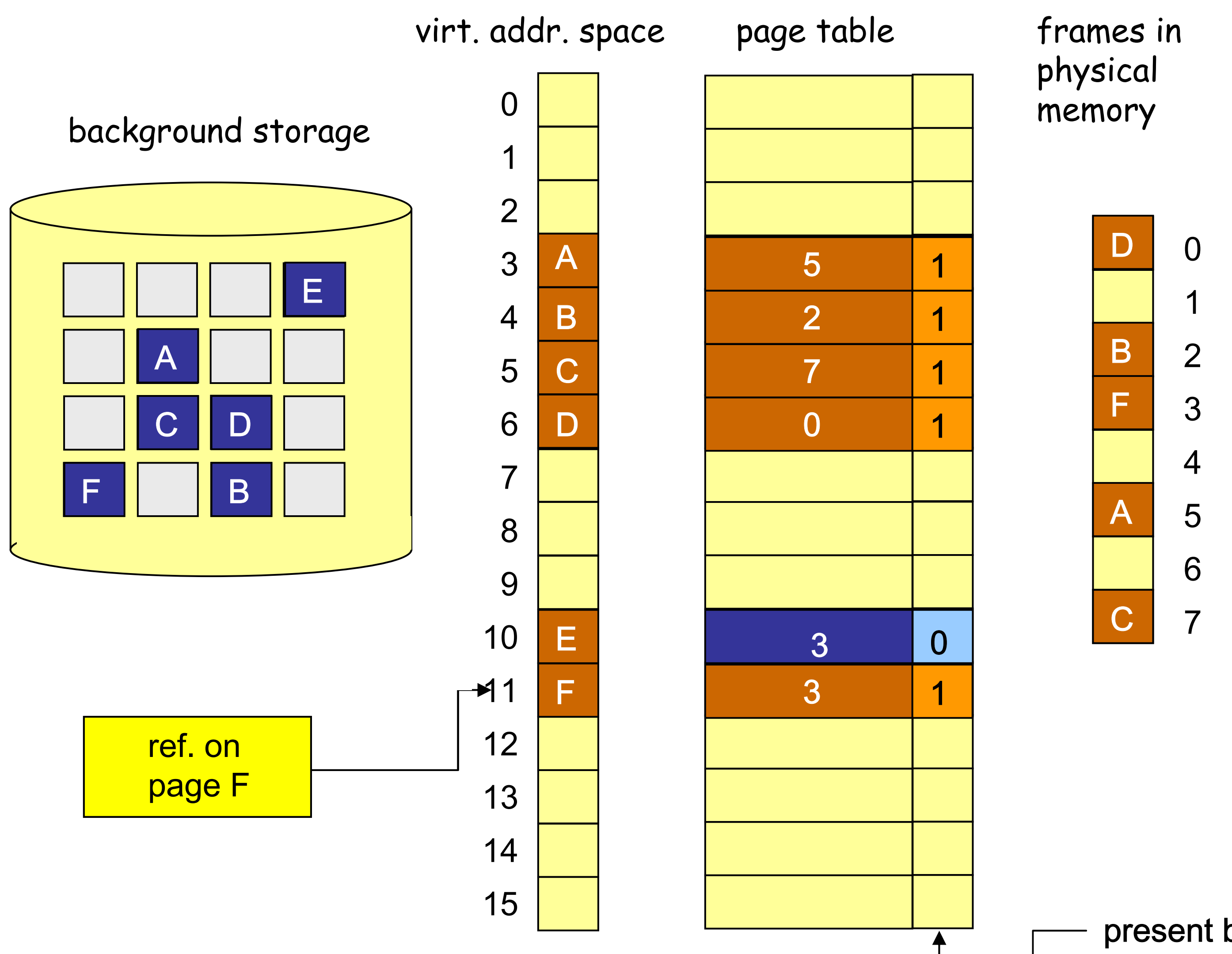
How to know which pages are needed in memory?

➡ dynamically swapping pages in real memory on demand

What to do if there are more pages needed as frames in memory are available?

➡ strategies to replace pages

demand paging



demand paging

hardware mechanism

operating system

reference to a memory location

page is mapped in TLB ?

yes → access phys. mem.

no ↓

exception: trap to OS
check present bit in
page descriptor

page is physical memory ?

yes → check for free TLB entry
load TLB with transl. info → access phys. mem.

no ↓

check for free frame
load page from disk
update page table → (back to: check for free TLB entry / load TLB with transl. info)

access phys. mem.

demand paging: discussion

normal memory access, no page fault:

access time ~ 5 - 200 ns

page fault penalty:

assumptions:

p: probability for a page fault (close to 0)

normal effective access time: 100 ns

load a page from disk: ~ 20 ms

Effective access time:

$(1-p) \cdot 100 + p \cdot 20000000 = 100 + 19999900 \cdot p$ für p=0,01: 199999 ns ~ 200 μs

to be in the range of the normal access time, the page fault probability is in the order of 0,000005 --> 1 page of 200000 pages should only lead to a page miss!

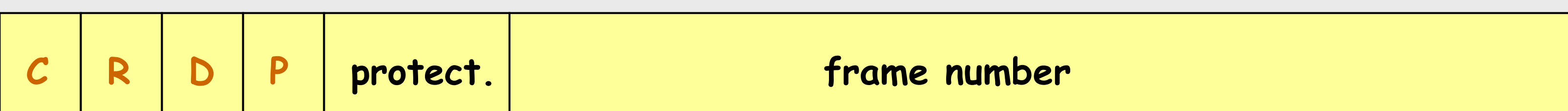
page replacement policies

What we need is to predict the page to which the next reference is the most distant in the chain of future references.

What we have to predict future behaviour:

- was the page accessed in the past?
- was the page modified in the past?
- what are the currently active processes?

page descriptor



C	R	D	P	protect.	frame number

C: Caching, R: Referenced, D: Dirty (modified), P: Present

optimal page replacement policies

3 frames

7 page faults

ref. sequence		1	2	3	4	1	2	5	1	2	3	4	5
frame assignment in phys. memory	frame 1	1	1	1	1	1	1	1	1	1	3	4	4
	frame 2		2	2	2	2	2	2	2	2	2	2	2
	frame 3			3	4	4	4	5	5	5	5	5	5
control state: distance to next reference	frame 1	4	3	2	1	3	2	1	∞	∞	∞	∞	∞
	frame 2	∞	4	3	2	1	3	2	1	∞	∞	∞	∞
	frame 3	∞	∞	7	7	6	5	5	4	3	2	2	∞
		P	P	P	P			P			P	P	

4 frames

6 page faults

ref. sequence		1	2	3	4	1	2	5	1	2	3	4	5
frame assignment in phys. memory	frame 1	1	1	1	1	1	1	1	1	1	1	4	4
	frame 2		2	2	2	2	2	2	2	2	2	2	2
	frame 3			3	3	3	3	3	3	3	3	3	3
	frame 4				4	4	4	5	5	5	5	5	5
control state: distance to next reference	frame 1	4	3	2	1	3	2	1	∞	∞	∞	∞	∞
	frame 2	∞	4	3	2	1	3	2	1	∞	∞	∞	∞
	frame 3	∞	∞	7	6	5	4	3	2	1	∞	∞	∞
	frame 4	∞	∞	∞	7	6	5	5	4	3	2	1	∞
		P	P	P	P	-	-	P	-	-	-	P	-

page replacement policies

Not-recently-used → distinguishes 4 classes of pages:

- class 0: R=0, D=0
- class 1: R=0, D=1
- class 2: R=1, D=0
- class 3: R=1, D=1

NRU deletes an arbitrary page from the lowest, non-empty class

problem (→ arbitrary)

					sweep ↓				sweep ↓				
ref. sequence		1	2	3	4	1	2	5	1	2	3	4	5
frame assignment in phys. memory	frame 1	1	1	1	1	1	1	5	1	1	3	3	5
	frame 2		2	2	2	2	2	2	2	2	2	2	5
	frame 3			3	4	4	4	4	4	4	4	4	4
control state: page class	frame 1	2	3	3	3	1	1	2	2	0	2	2	2
	frame 2	-	2	2	2	0	2	3	3	1	1	1	2
	frame 3	-	-	2	2	0	2	3	3	1	3	2	2
		P	P	P	P			P	P		P		P

8 page faults

page replacement policies

FIFO: replaces the page which has been the longest time in memory

ref. sequence		1	2	3	4	1	2	5	1	2	3	4	5
frame assignment in phys. memory	frame 1	1	1	1	4	4	4	5	5	5	5	5	5
	frame 2		2	2	2	1	1	1	1	1	3	3	3
	frame 3			3	3	3	2	2	2	2	2	4	4
control state: age of frame	frame 1	0	1	2	0	1	2	0	1	2	3	4	5
	frame 2	-	0	1	2	0	1	2	3	4	0	1	2
	frame 3	-	-	0	1	2	0	1	2	3	4	0	1
		P	P	P	P	P	P	P			P	P	

9 page faults

page replacement policies

FIFO: Belady's anomaly

ref. sequence		1	2	3	4	1	2	5	1	2	3	4	5
frame assignment in phys. memory	frame 1	1	1	1	1	1	1	5	5	5	5	4	4
	frame 2		2	2	2	2	2	2	1	1	1	1	5
	frame 3			3	3	3	3	3	3	2	2	2	2
	frame 4				4	4	4	4	4	4	3	3	3
control state: age of frame	frame 1	0	1	2	3	4	5	0	1	2	3	0	1
	frame 2	-	0	1	2	3	4	5	0	1	2	3	0
	frame 3	-	-	0	1	2	3	4	5	0	1	2	3
	frame 4	-	-	-	0	1	2	3	4	5	0	1	2
		P	P	P	P	-	-	P	P	P	P	P	P

10 page faults

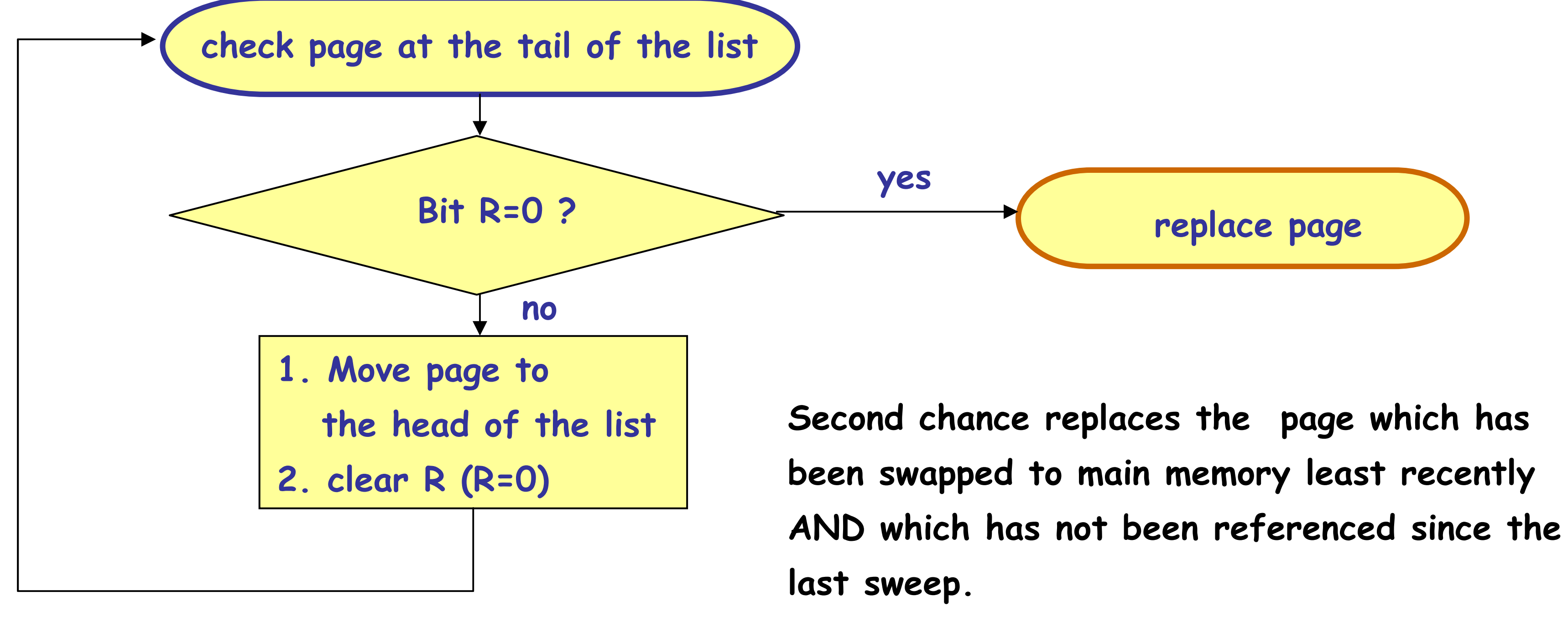
Although there are more frames, FIFO generates more page faults!

Basic problem: does not consider usage.

page replacement policies

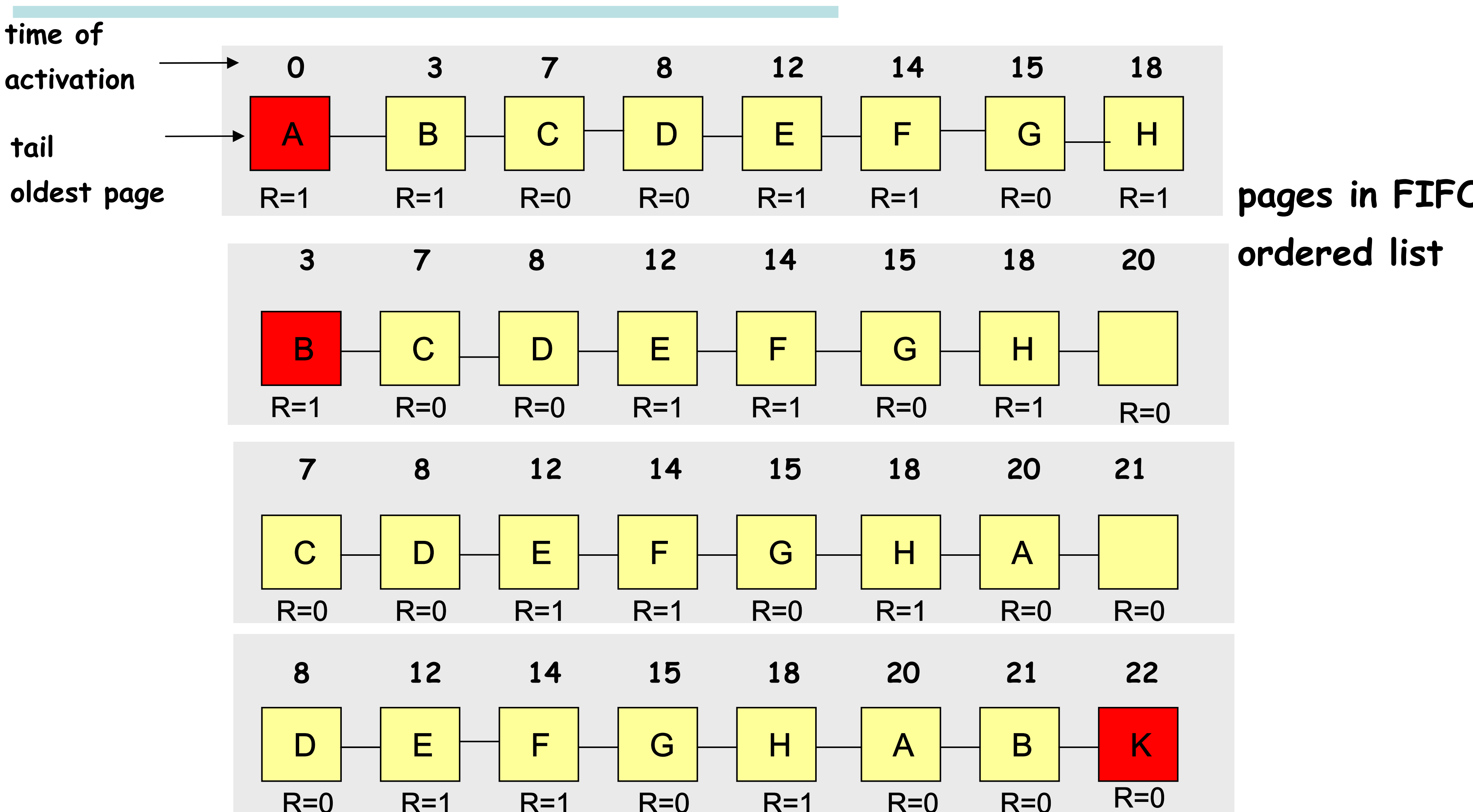
Variation of FIFO: The second chance algorithm

pages are organized in a FIFO ordered list

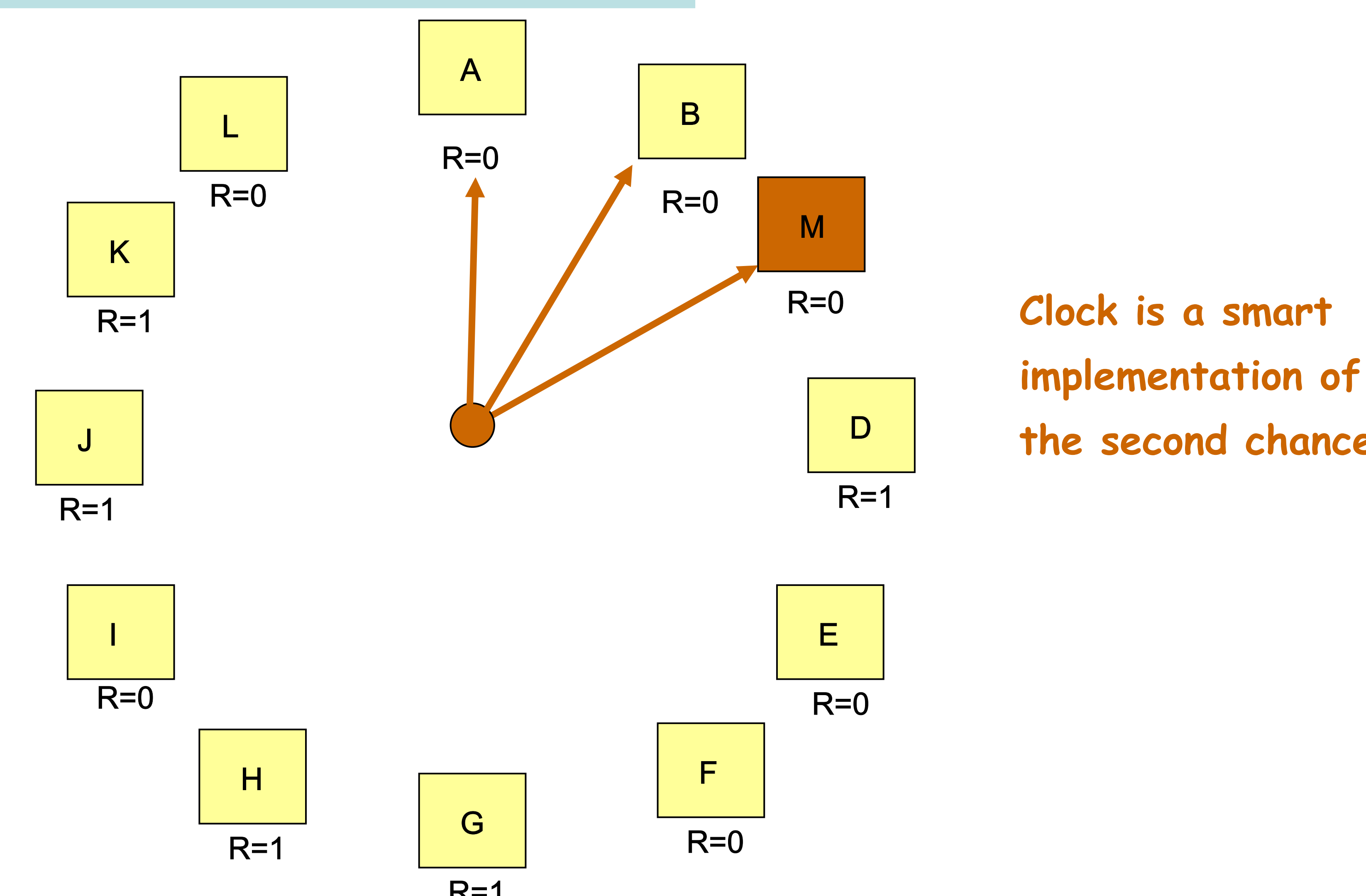


Second chance replaces the page which has been swapped to main memory least recently AND which has not been referenced since the last sweep.

The second chance algorithm



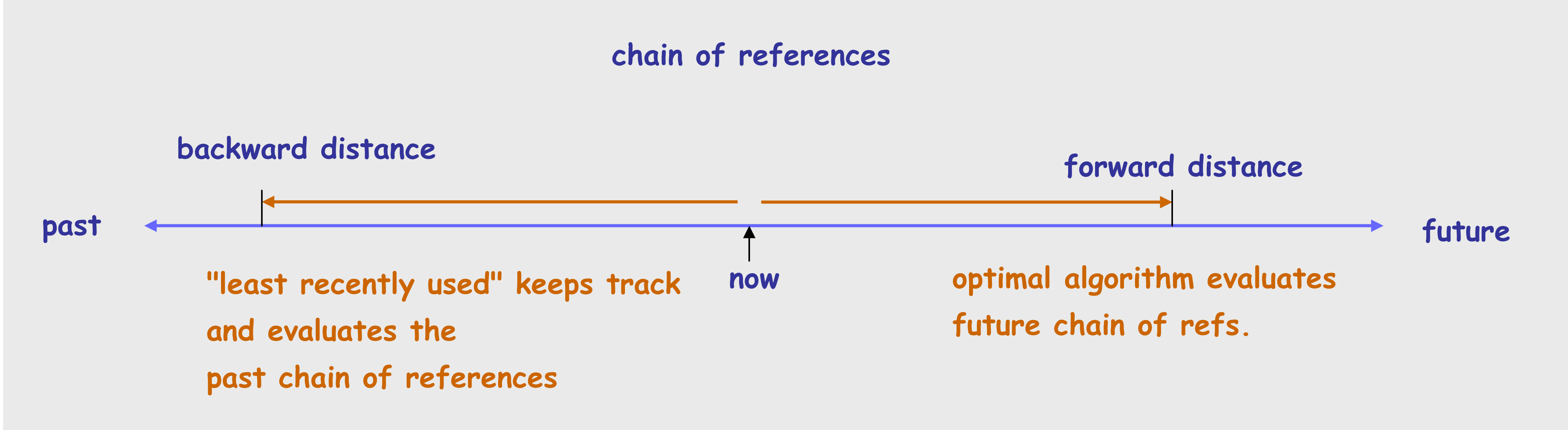
Variation of the "second chance" algorithm: The Clock-Algorithm



Clock is a smart implementation of the second chance

page replacement policies

Least-Recently-Used: delete the page which was not used for the longest time.



page replacement policies

Least-Recently-Used

ref. sequence		1	2	3	4	1	2	5	1	2	3	4	5
frame assignment in phys. memory	frame 1	1	1	1	4	4	4	5	5	5	3	3	3
	frame 2		2	2	2	1	1	1	1	1	1	4	4
	frame 3			3	3	3	2	2	2	2	2	2	5
control state: backward distance	frame 1	0	1	2	0	1	2	0	1	2	0	1	2
	frame 2	-	0	1	2	0	1	2	0	1	2	0	1
	frame 3	-	-	0	1	2	0	1	2	0	1	2	0
		P	P	P	P	P	P	P			P	P	P

10 page faults

page replacement policies

Least-Recently-Used

ref. sequence		1	2	3	4	1	2	5	1	2	3	4	5
frame assignment in phys. memory	frame 1	1	1	1	1	1	1	1	1	1	1	1	5
	frame 2		2	2	2	2	2	2	2	2	2	2	2
	frame 3			3	3	3	3	5	5	5	5	4	4
	frame 4				4	4	4	4	4	4	3	3	3
control state: backward distance	frame 1	0	1	2	3	0	1	2	0	1	2	3	0
	frame 2	-	0	1	2	3	0	1	2	0	1	2	3
	frame 3	-	-	0	1	2	3	0	1	2	3	0	1
	frame 4	-	-	-	0	1	2	3	4	5	0	1	2
		P	P	P	P			P			P		P

8 page faults

page replacement policies

How to implement LRU?

1. LRU-ordered list of pages. With every memory access the list order is updated.
2. Counter field in every page table entry. With every page replacement the lowest countervalue must be found. Problem: Dedicated counter hardware, high overhead.
3. Access matrix hardware. Problem special hardware, very high overhead (n frames: n^2 bits).

Example: ref. sequence 0 2 1, row with the lowest binary number identifies least recently used page.

	0	1	2	3
0	0	0	0	0
1	0	0	0	0
2	0	0	0	0
3	0	0	0	0

0	1	2	3
0	1	1	1
0	0	0	0
0	0	0	0
0	0	0	0

0	1	2	3
0	1	0	1
0	0	0	0
1	1	0	1
0	0	0	0

0	1	2	3
0	0	0	1
1	0	1	1
1	0	0	1
0	0	0	0

page replacement policies

LRU implemented in Software.

Not frequently used: Software-counter for each page, initially "0", periodically incremented by R-Bit value. Page with lowest counter value will be replaced.
Problem: Pure NFU never forgets.

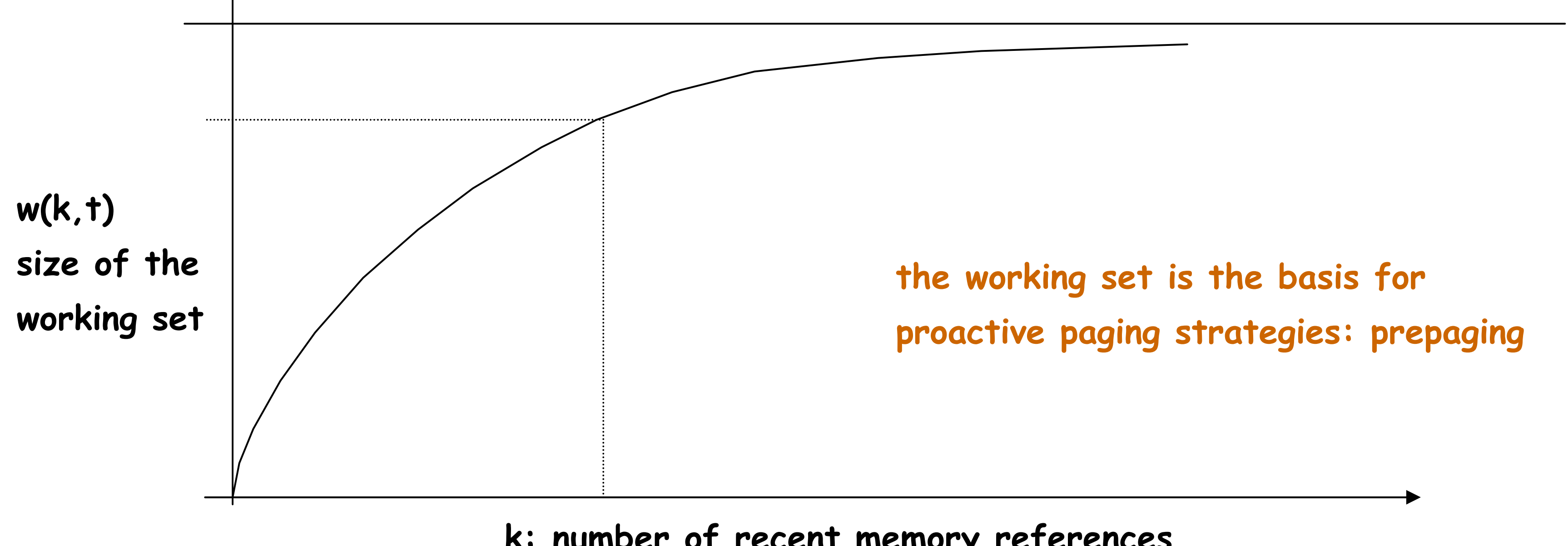
Aging: Variation of NFU. Software-counter is shifted right and R-Bit value is added to the leftmost (most significant) bit. Page with lowest counter value will be replaced.
Good approximation of LRU.

aging

timer interval	1	2	3	4	5
R-Bits (1 2 3 4 5 6)	1 0 1 0 1 1	1 1 0 0 1 0	1 1 0 1 0 1	1 0 0 0 1 0	0 1 1 0 0 0
0	1000 0000	1100 0000	1110 0000	1111 0000	0111 1000
1	0000 0000	1000 0000	1100 0000	0110 0000	1011 0000
2	1000 0000	0100 0000	0010 0000	0001 0000	1000 1000
3	0000 0000	0000 0000	1000 0000	0100 0000	0010 0000
4	1000 0000	1100 0000	0110 0000	1011 0000	0101 1000
5	1000 0000	0100 0000	1010 0000	0101 0000	0010 1000

page replacement policies

The working set algorithm



The set of pages which are used by a process in a certain time window is called the working set.

Peter Denning: The Working Set Model for Program Behaviour, CACM, May 1968

The working set algorithm

page descriptor

replacement algorithm:

- scan all page descriptors
- if R=1: set vt to cvt and set R=0;
- if R=0 $\wedge$ (cvt - vt) < t:
  replace page;
- if R=0 $\wedge$ (cvt - vt) > t:
  no change;
- if no page is found with
  R=0 $\wedge$ (cvt - vt) < t then
  replace oldest page;
- if all pages are referenced
  replace arbitrary page.

page table	
2083	1
2003	1
1981	1
1212	0
2014	1
2020	1
2032	1
1620	0
....	

R-Bit

virtual time: vt

field contains time of last access to the page

virtual time denotes some form of process local time representation, starts when process starts.

current virtual time: cvt

2204

The WSClock algorithm

R-Bit

current virtual time: cvt

2204

1620	0
2032	1
2204	0
2204	0
2204	1
1981	1
2003	1
2083	1

page replacement policies summary

algorithm	properties	impl.	comment
optimal	🙂 🙂 🙂	🙁 🙁 🙁	for comparison only, cannot be realized
NRU:	🙂	🙂 🙂	very simple but easy to implement
FIFO	🙂	🙂 🙂	simple; problem: important pages may be replaced + FIFO anomaly
2nd chance:	🙂 🙂	🙂	substantial improvment over FIFO
Clock:	🙂 🙂	🙂 🙂	smart implementation of second chance
LRU:	🙂 🙂 🙂	🙁 🙁	excellent, but implementation problems
NFU:	🙂 🙂 🙂	🙂	efficient algorithm, properties close to LRU
WS:	🙂 🙂 🙂	🙂	good, implementation problems
WSClock:	🙂 🙂 🙂	🙂 🙂	good + efficient

the class of stack algorithms

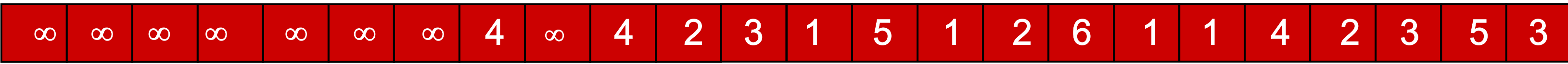
ref. string		0	2	1	3	5	4	6	3	7	4	7	3	3	5	5	3	1	1	1	7	1	3	4	1
pages in phys. memory (m)		0	2	1	3	5	4	6	3	7	4	7	3	3	5	5	3	1	1	1	7	1	3	4	1
			0	2	1	3	5	4	6	3	7	4	7	7	3	3	5	3	3	3	1	7	1	3	4
				0	2	1	3	5	4	6	3	3	4	4	7	7	7	5	5	5	3	3	7	1	3
					0	2	1	3	5	4	6	6	6	6	4	4	4	7	7	7	5	5	5	7	7
						0	2	1	1	5	5	5	5	5	6	6	6	4	4	4	4	4	4	5	5
pages out of phys. memory (n-m)							0	2	2	1	1	1	1	1	1	1	1	6	6	6	6	6	6	6	6
								0	0	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
										0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
		p	p	p	p	p	p	p		p					●			p						●	
distance string		∞	∞	∞	∞	∞	∞	∞	4	∞	4	2	3	1	5	1	2	6	1	1	4	2	3	5	3

important properties of stack algorithms: $M(m,r) \subseteq M(m+1,r)$

m: # of frames, r: distance index

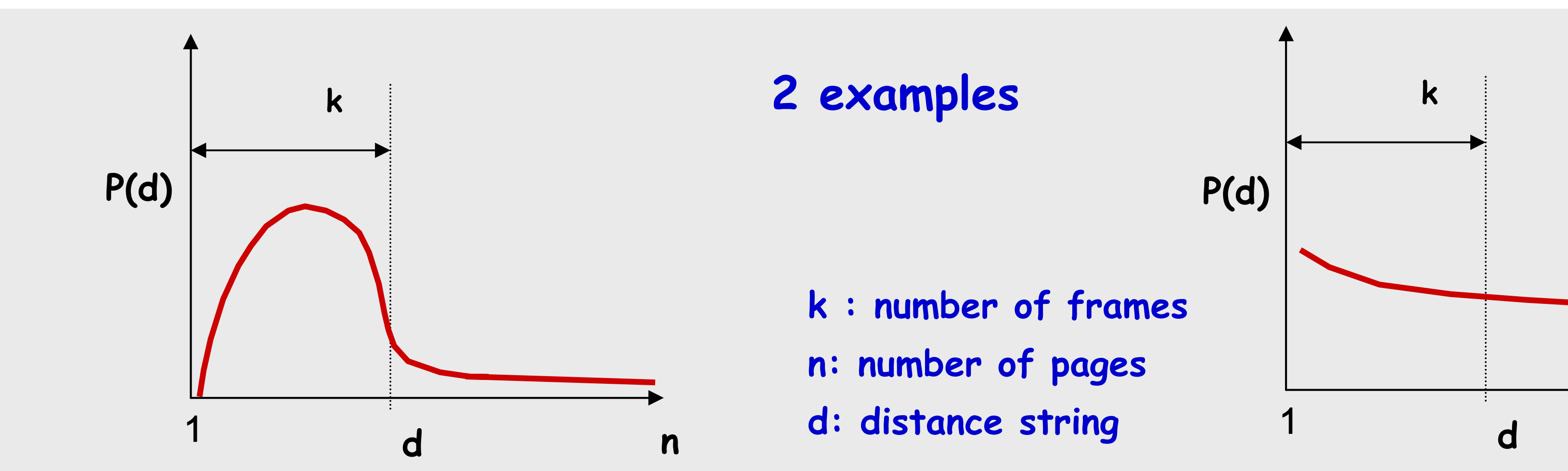
analysis of the distance chain

distance string



distance values:	1	2	3	4	5	6	7	8	∞
occurence:	4	3	3	3	2	1	0	0	8

the distance string depends:
1.) on the reference string
2.) on the page replacement strategy



predicting page fault rate

distance string

distance string	∞	∞	∞	∞	∞	∞	∞	4	∞	4	2	3	1	5	1	2	6	1	1	4	2	3	5	3

$C_1 \ldots C_\infty$

distance values:	1	2	3	4	5	6	7	8	∞
occurence:	4	3	3	3	2	1	0	0	8

$C_1 = 4$
$C_2 = 3$
$C_3 = 3$
$C_4 = 3$
$C_5 = 2$
$C_6 = 1$
$C_7 = 0$
$C_8 = 0$
$C_\infty = 8$

page fault rate F for n pages and m frames

$$F_m = \sum_{k=m+1}^{n} C_k + C_\infty$$

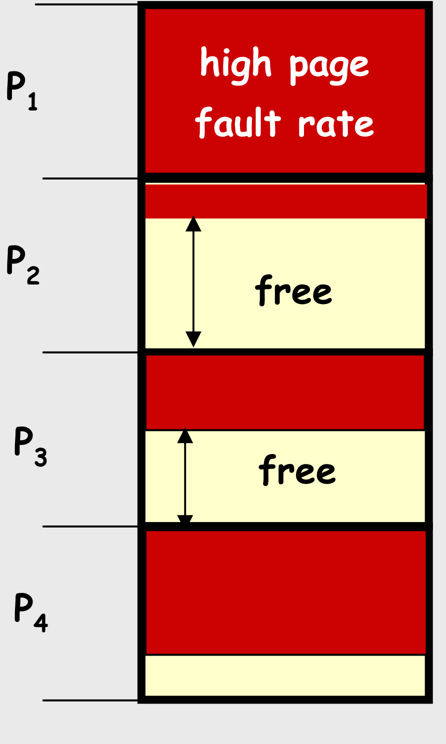
$F_1 = 20$	$C_2 + \ldots + C_\infty$
$F_2 = 17$	$C_3 + \ldots + C_\infty$
$F_3 = 14$	$C_4 + \ldots + C_\infty$
$F_4 = 11$	$C_5 + \ldots + C_\infty$
$F_5 = 9$	$C_6 + \ldots + C_\infty$
$F_6 = 8$	$C_7 + \ldots + C_\infty$
$F_7 = 8$	$C_8 + \ldots + C_\infty$
$F_8 = 8$	C_∞
$F_\infty = 8$	

design issues for paging

- local vs. global paging policies
- page size
- separating program and data pages
- sharing of pages
- release policies
- transparency issues and interface to the virtual memory

local paging policies

local policy: a fixed budget of frames is assigned to every process.



local policy: a variable budget of frames is assigned to every process.

- assignment proportional to process size.
- use distance chain analysis to determine number of frames needed (static).
- use page fault frequency measurements to determine number of frames (dynamic). (PFF-algorithm)

global paging policies

- global policies are more flexible and have more potential to balance memory requests.
- global policies do not work with all page replacement strategies, e.g. a global working set does not make sense.
- OS must prevent monopolization of memory by one or a few processes.
- Swapping processes to disk to reduce memory demand.

page size

for n segments with p bytes, the internal fragmentation is: np/2

trade-offs:

page size:	large	small
internal fragmentation ($n \cdot p/2$):	☹	☺
page table size+management overhead	☺	☹
load time from disk	☺	☹

Common page sizes are in the range between 512 Bytes and 64KB.

Today, page sizes of 4KB or 8KB are most common.

design issues for paging

- local vs. global paging policies
- page size
- separating program and data pages
- sharing of pages
- release policies
- transparency issues and interface to the virtual memory

separating policies and mechanisms

user address space

kernel address space

micro-kernel

- user process → (page fault) → page fault handler
- page fault handler → (requests page) → external pager
- external pager → (requests page) → disk
- disk → (retrieved page) → external pager
- external pager → (delivers page) → page fault handler
- page fault handler → (set up/update data structures) → MMU handler

finding pages on the disk

static swap space:

pages are easy to find by indexing the swap space.

swap space covers entire virtual memory space of a process.

finding pages on the disk

dynamic swap space:

finding pages needs extra mapping table.

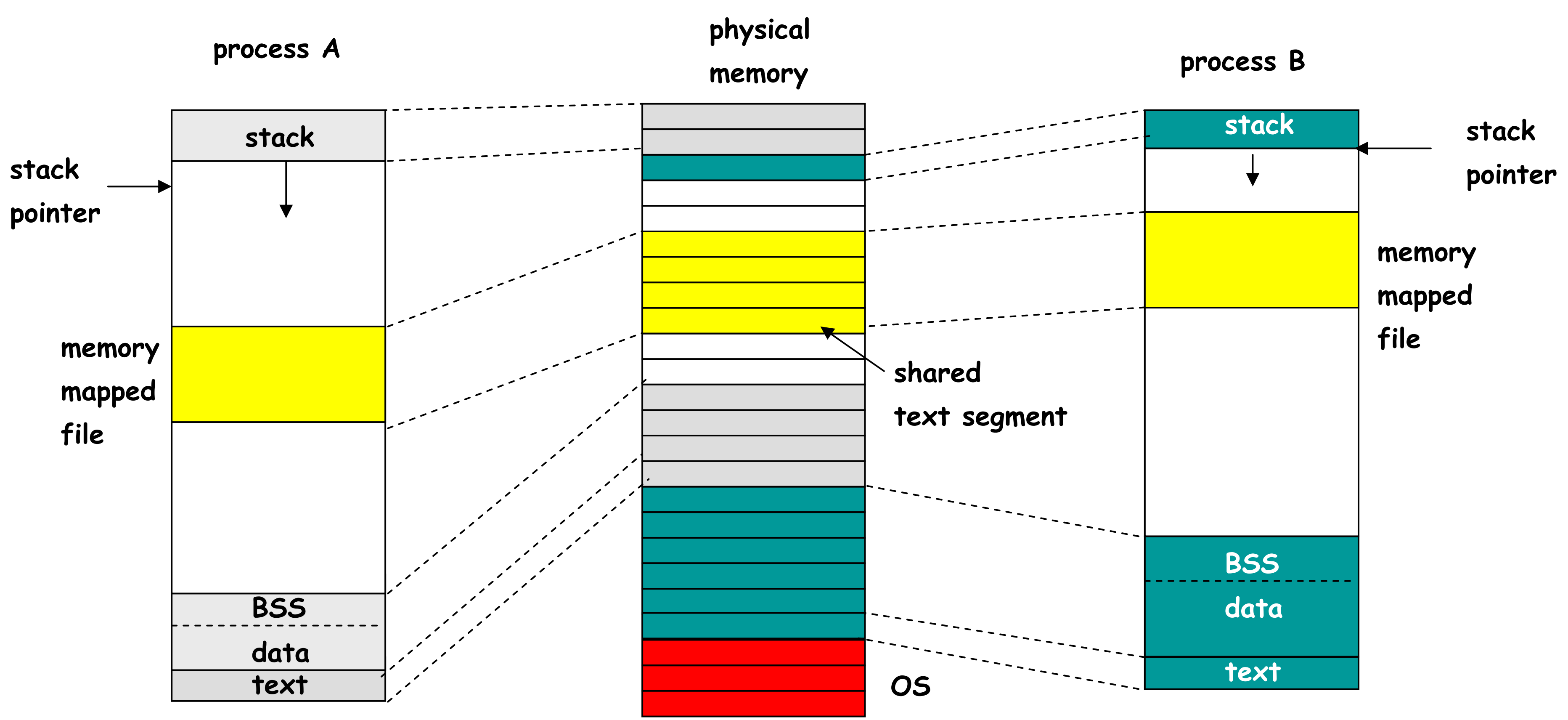
swap space covers only replaced (but assigned) pages.

pages in main memory may not have disk images.

Examples

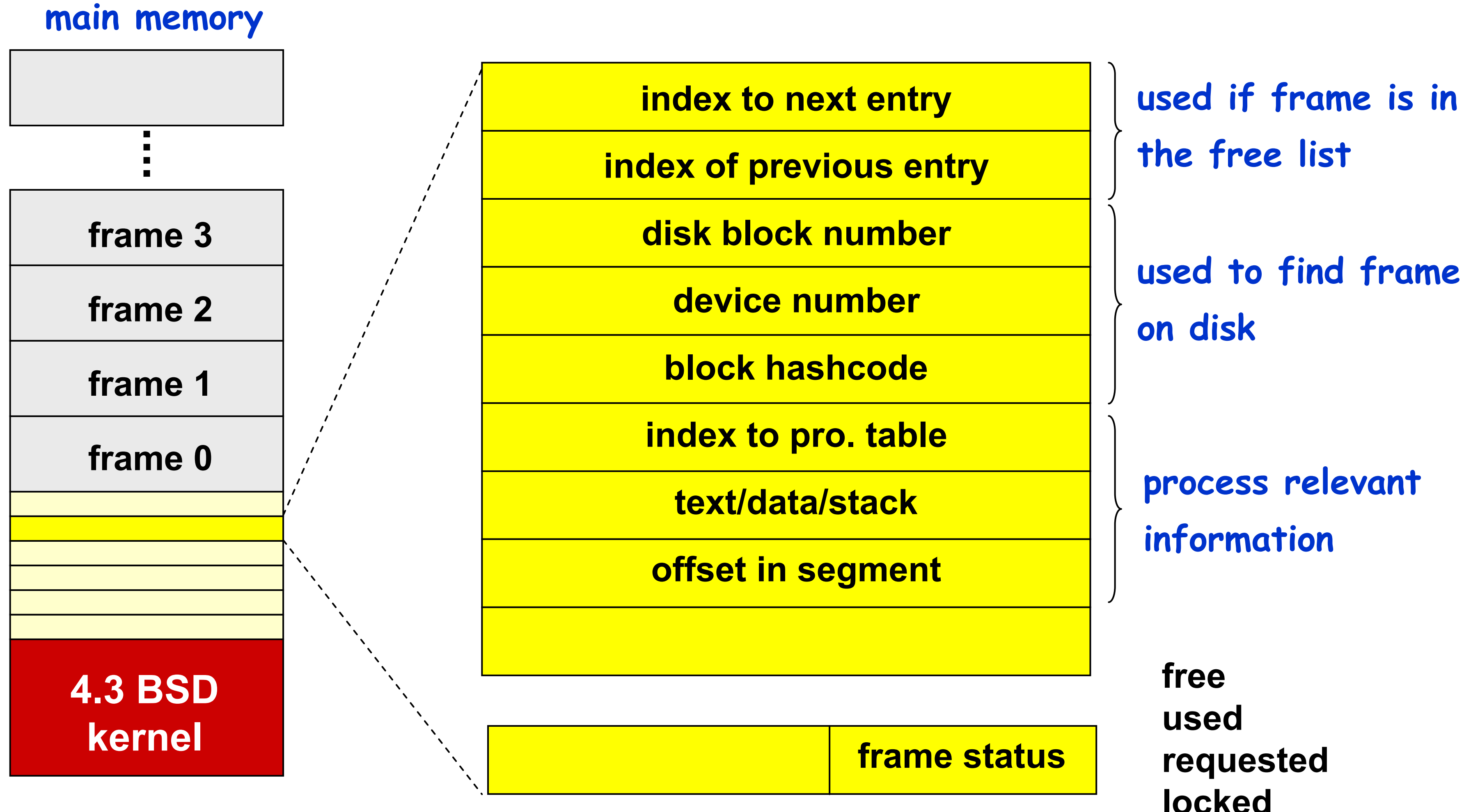
Unix-originated OS

Example: Memory management in Unix et al.



- supports processor architectures which separate program (read-only text) and data.
- stack contains context variables whihc define the execution environment for the process.

Example: Core Map in 4BSD



Example: Memory management in Unix et al.

Paging system

Data structures in System V Realease 4 (SVR4):

Page table

Disk block-descriptor table

Frame data table

Swap-use-table

Page table entry:

frame number	age	copy on write	D	R	V	protect.

Disk block-descriptor:

swap device number	dev. block number	storage type

frame data table entry:

fr. status	ref.cntr	log. device	block no.	pointer

pointer:
- other frame table pointers
- list of free pages
- hash queue

swap-use-table entry

ref.cntr	page-ID. on dev.

Example: Memory management in Unix et al.

System calls for memory management

s* = brk(addr)	change data segment size
a= mmap (addr, len, prot, flags, fd, offset)**	map file
s= munamap (addr, len)	unmap file

* s, a : return values (-1 if an error occurs)

** addr: start address of memory block

len: length in bytes

prot: protection bits

flags: controle privat or shared use

fd: file descriptor

offset: offset where the mapped section starts in file

Memory management in Unix et al.

less free pages ↔ more free pages

phys. memory

minfree | desfree | lotsfree | +deficit

process deactivation if >5sec below minfree

pageout expl. triggered. Proc. deact. if more than 30sec below desfree

pageout in regular intervals

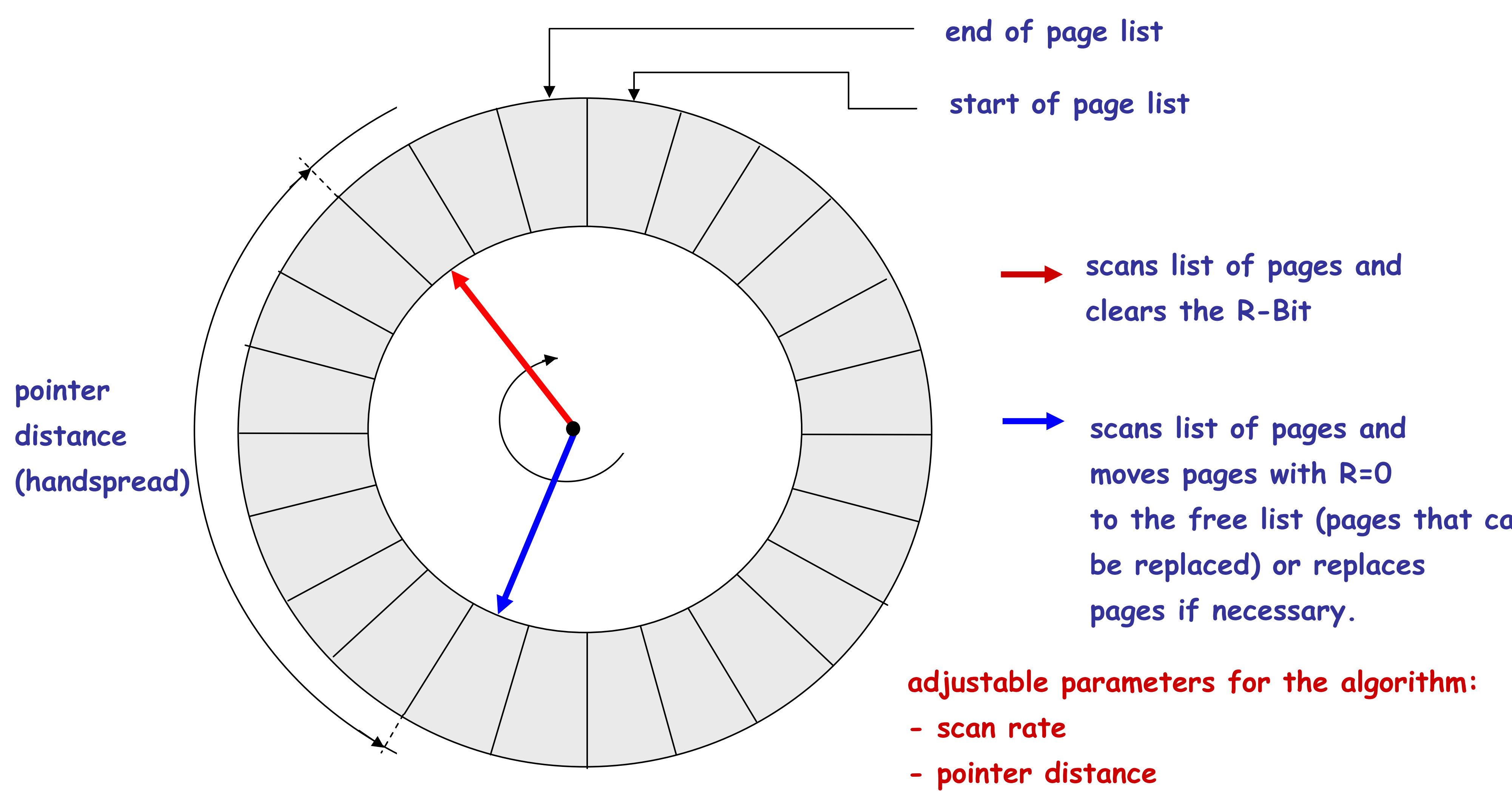
no *pageout*

fastscan

slowscan

scan rate

releasing pages: clock with two pointers

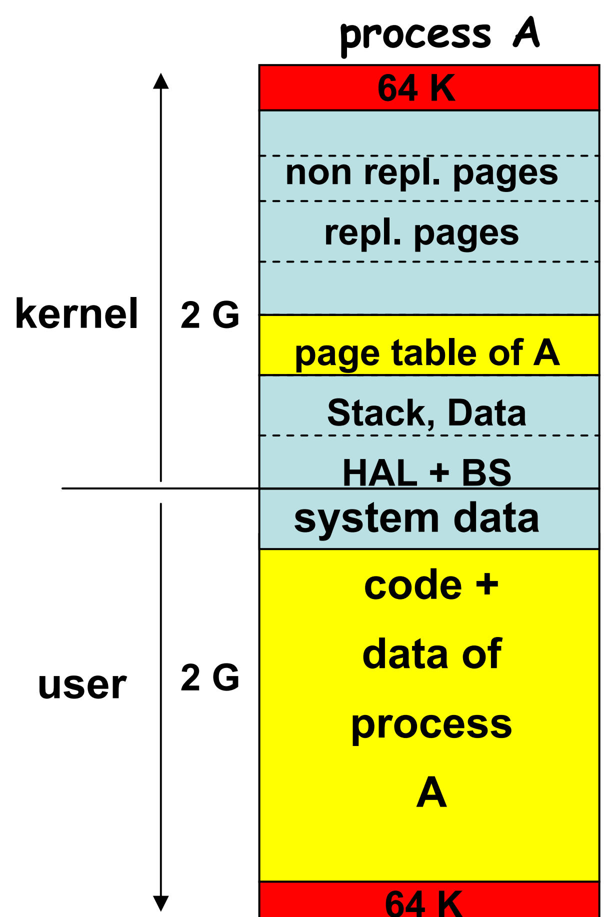


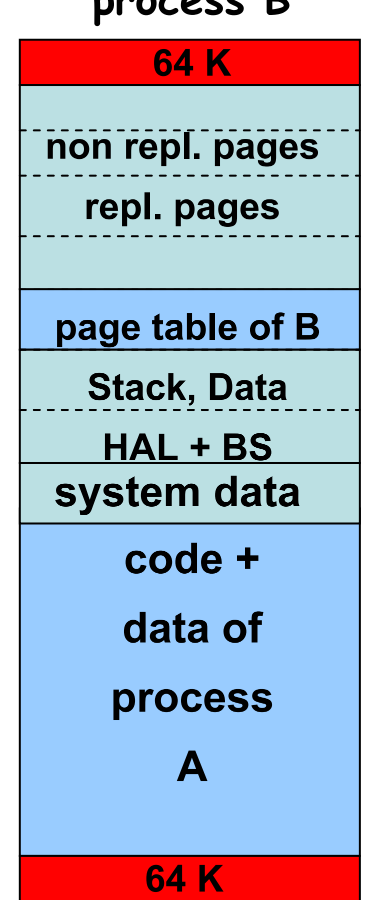
Examples

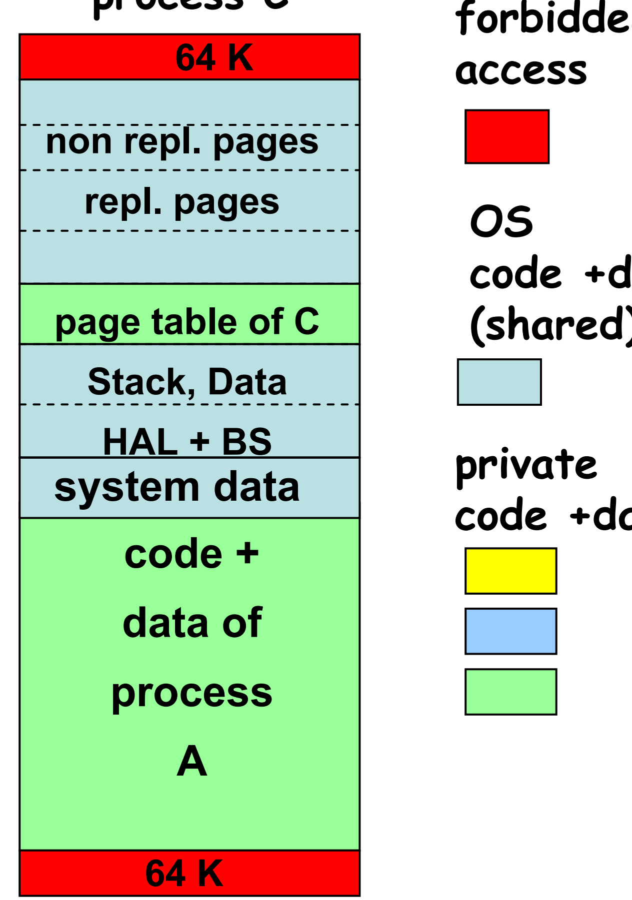
Windows 2000

Example: Memory management in Windows 2k

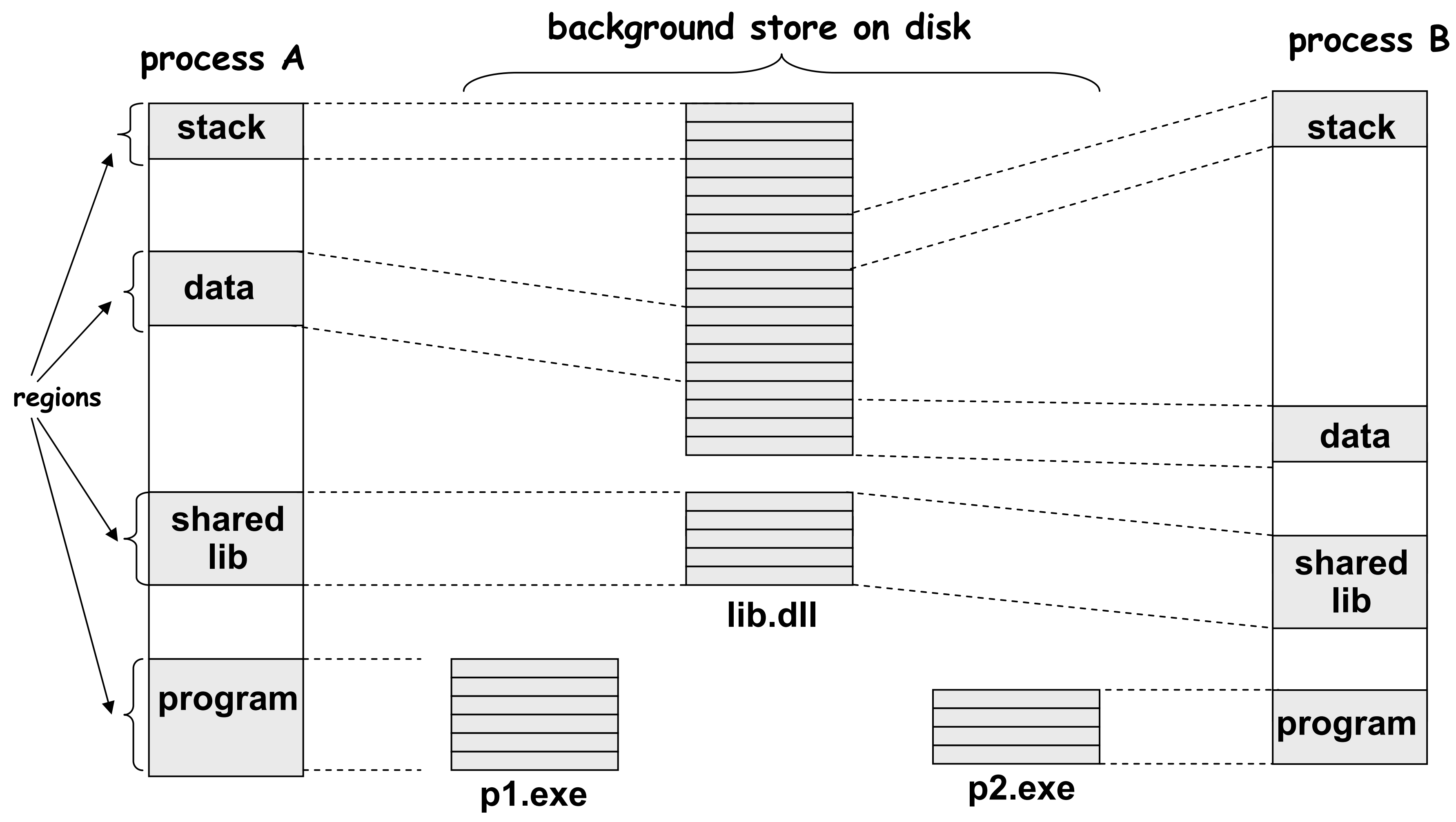
Every process has its own virtual address space of 4 GB







Example: Memory management in Windows 2k



Example: Memory management in Windows 2k

API for memory management in W2K

VirtualAlloc	reserve or allocate a region
VirtualFree	release a region
VirtualProtect	change the protection status of a region
VirtualQuery	check region status
VirtualLock	disable page replacement in this region
VirtualUnlock	enable page replacement in this region
CreateFileMapping	create a file representation and assign a name
MapViewOf File	map file to virtual address space
UnMapViewOfFile	unmap...
OpenFileMapping	open of an already mapped file

Example: Memory management in Windows 2k

Page replacement:

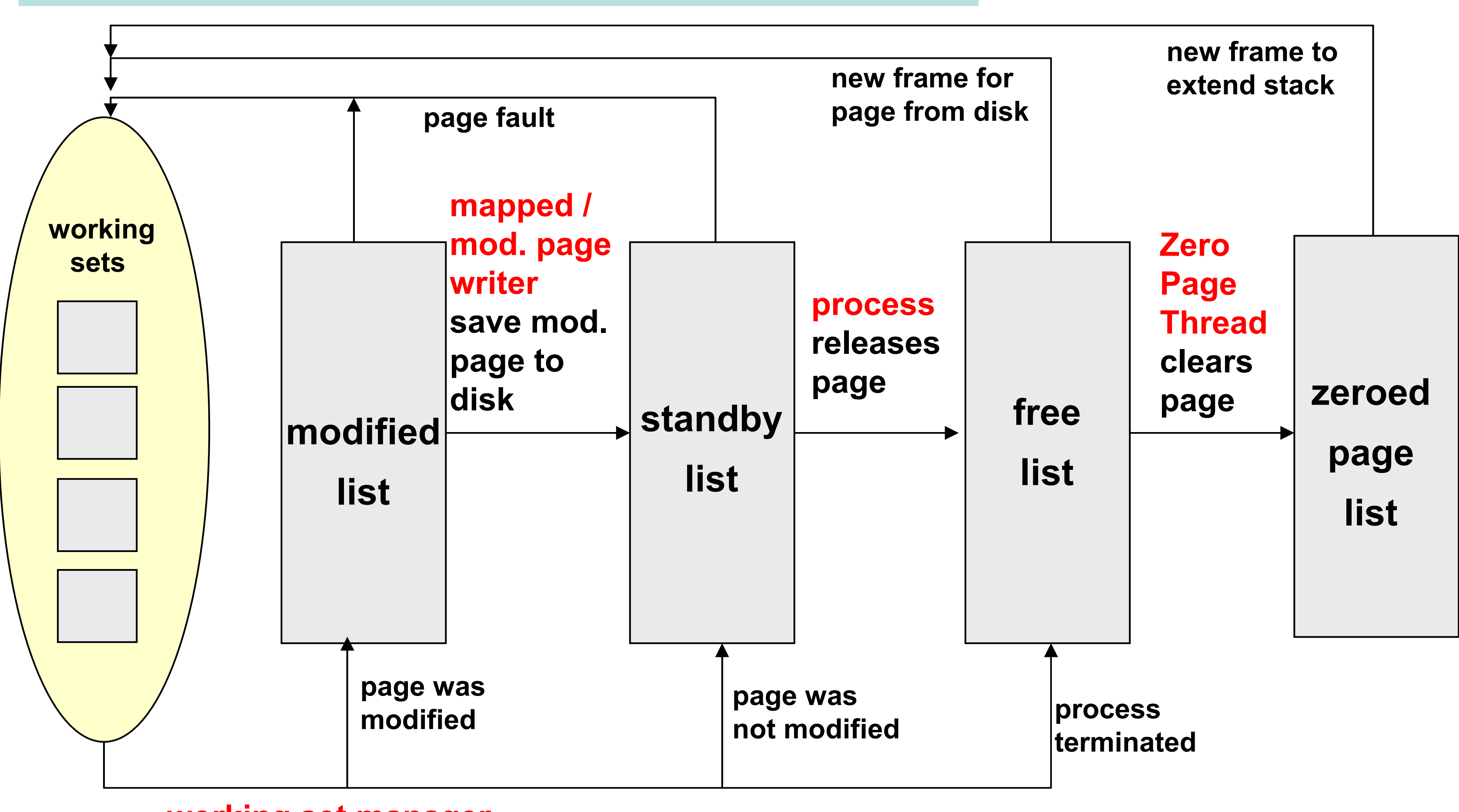
Basic algorithm: working set

parameter: min (20..50), max (45..345)

Balance set manager: checks for enough free pages.

Working set manager: checks the WS for replacable pgs.

Example: Memory management in Windows 2k



faulty frames list