last time (1)

mmap — process memory as list of mappings

mapping: read/write, private/shared, underlying file if any

memory as cache

- cached parts of files (for read *or* "mapped" into process's memory)
- "anonymous" data (like heap) *swapped* to disk if needed

forward mapping for hits: page tables (virtual → physical page)

forward mapping for misses: OS data structures

- virtual page → file + offset → cached pages/location on disk
- virtual page → temporary location on disk

last time (2)

memory as cache...

reverse mapping: physical page $\rightarrow$ page table entries
- needed to replace with some other data

Linux solution: data structure per physical page
- point to underlying file if any, file points to processes using it
- point to list of mappings (page table uses) for non-file data ("anon_vma")
  - data for heap, stack
  - copied-on-write parts of private mappings (e.g. initialized globals)
- space-saving: share lists between related pages (e.g. heap pages after fork)

started: page replacement goals

page replacement

step 1: evict a page to free a physical page

step 2: load new, more important in its place

evicting a page

find a ‘victim’ page to evict

remove victim page from page table, etc.

- every page table it is referenced by
- every list of file pages
- ...

if needed, save victim page to disk

page replacement goals

hit rate: minimize number of misses

throughput: minimize overhead/maximize performance

fairness: every process/user gets its ‘share’ of memory

will start with optimizing hit rate

max hit rate $\approx$ max throughput

optimizing hit rate almost optimizes throughput, but...

max hit rate $\approx$ max throughput

optimizing hit rate almost optimizes throughput, but…

cache miss costs are variable

- creating zero page versus reading data from slow disk?
- write back dirty page before reading a new one or not?
- reading multiple pages at a time from disk (faster per page read)?
- …

being proactive?

can avoid misses by "reading ahead"

- guess what's needed — read in ahead of time
- wrong guesses can have costs besides more cache misses

we will get back to this later

for now — only access/evict on demand

optimizing for hit-rate

assuming:

- we only bring in pages on demand (no reading in advance)
- we only care about maximizing cache hits

best possible page replacement algorithm: Belady's MIN

replace the page in memory accessed furthest in the future

- (never accessed again $=$ infinitely far in the future)

optimizing for hit-rate

assuming:

- we only bring in pages on demand (no reading in advance)
- we only care about maximizing cache hits

best possible page replacement algorithm: Belady's MIN

replace the page in memory accessed furthest in the future

- (never accessed again = infinitely far in the future)

impossible to implement in practice, but...

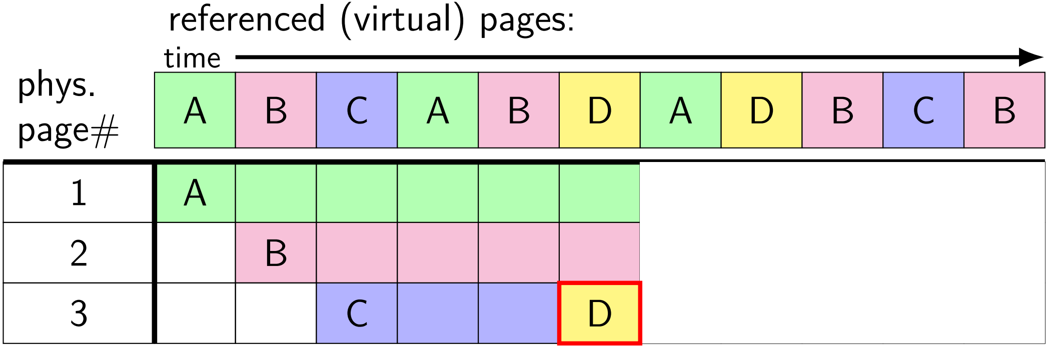
Belady's MIN

referenced (virtual) pages:

time →

phys. page#	A	B	C	A	B	D	A	D	B	C	B
1	A										
2		B									
3			C								

Belady's MIN



A next accessed in 1 time unit
B next accessed in 3 time units
C next accessed in 4 time units
choose to replace C

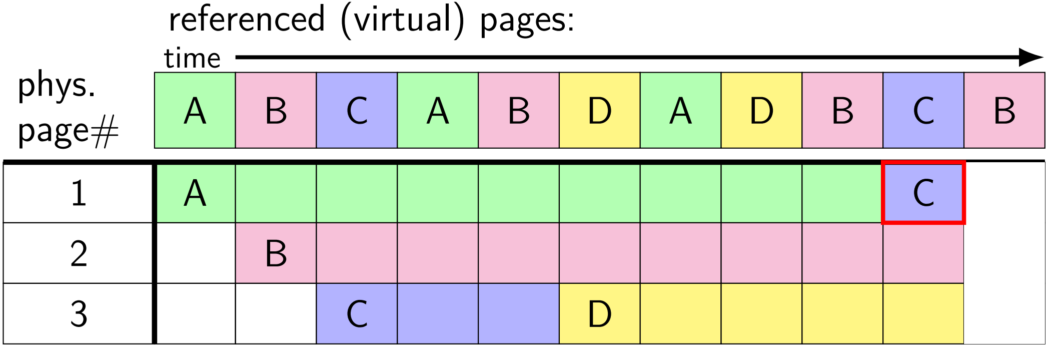
Belady's MIN

referenced (virtual) pages:

time →

phys. page#	A	B	C	A	B	D	A	D	B	C	B
1	A										
2		B									
3			C			D					

Belady's MIN



A next accessed in ∞ time units
B next accessed in 1 time units
D next accessed in ∞ time units
choose to replace A or D (equally good)

Belady's MIN

referenced (virtual) pages:

time →

phys. page#	A	B	C	A	B	D	A	D	B	C	B
1	A									C	
2		B									
3			C			D					

predicting the future?

can't really...

look for common patterns

the working set model

one common pattern: working sets

at any time, program is using a subset of its memory

- set of running functions
- their local variables, (parts of) global data structure

subset called its *working set*

rest of memory is inactive

cache size versus miss rate

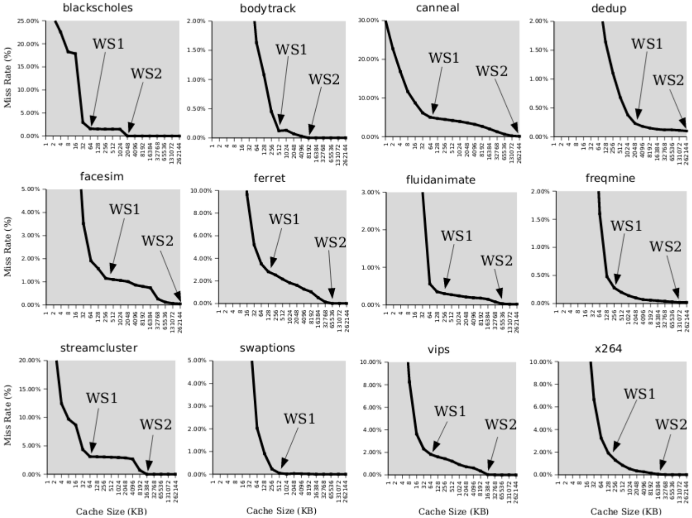


Figure 3: Miss rates versus cache size. Data assumes a shared 4-way associative cache with 64 byte lines. WS1 and WS2 refer to important working sets which we analyze in more detail in Table 2. Cache requirements of PARSEC benchmark programs can reach hundreds of megabytes.

Bienia et al, "The PARSEC Benchmark Suite: Characterization and Architectural Implications"

working sets and running many programs

give each program its working set

…and, to run as much as possible, not much more

inactive — won't be used

working sets and running many programs

give each program its working set

…and, to run as much as possible, not much more

inactive — won't be used

replacemnet policy: identify working sets (how?)

replace anything that's not in in it

working set model and phases

what happens when a program changes what it's doing?

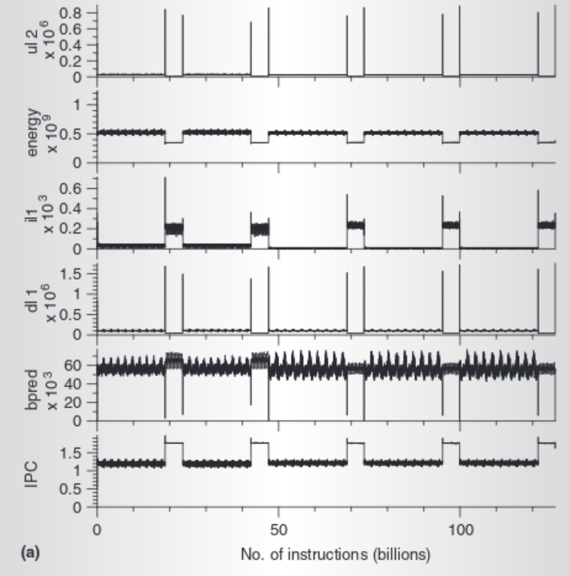
e.g. finish parsing input, now process it

phase change — discard one working set, give another

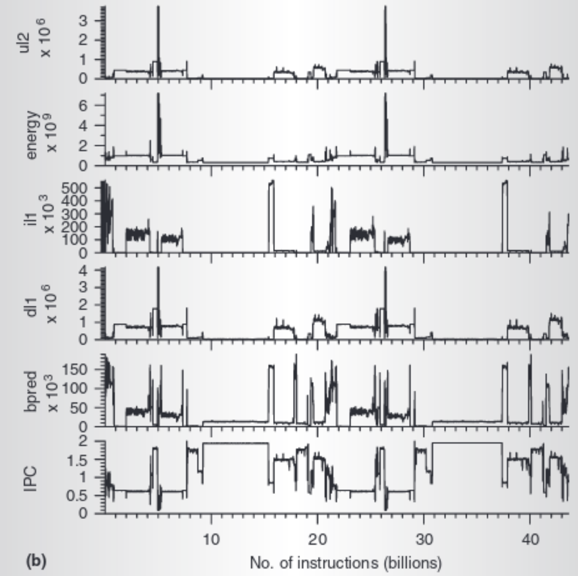
phase changes likely to have spike of cache misses

- whatever was cached, not what's being accessed anymore
- maybe along with change in kind of instructions being run

evidence of phases (gzip)



evidence of phases (gcc)



estimating working sets

working set $\approx$ what's been used recently

assuming not in phase change...

so, what a program recently used $\approx$ working set

can use this idea to estimate working set (from list of memory accesses)

using working set estimates

one idea: split memory into *part of working set* or *not*

using working set estimates

one idea: split memory into *part of working set* or *not*

not enough space for all working sets — stop whole program

- maybe a good idea, not done by common consumer/server OSes

using working set estimates

one idea: split memory into *part of working set* or *not*

not enough space for all working sets — stop whole program

- maybe a good idea, not done by common consumer/server OSes

allocating new memory: take from least recently used memory

- = not in a working set
- what most current OS try to do

practically optimizing for hit-rate

recall?: locality assumption

temporal locality: things accessed now will be accessed again soon

(for now: not concerned about spatial locality)

more possible policies: least recently used or least frequently used

practically optimizing for hit-rate

recall?: locality assumption

temporal locality: things accessed now will be accessed again soon

(for now: not concerned about spatial locality)

more possible policies: least recently used or least frequently used

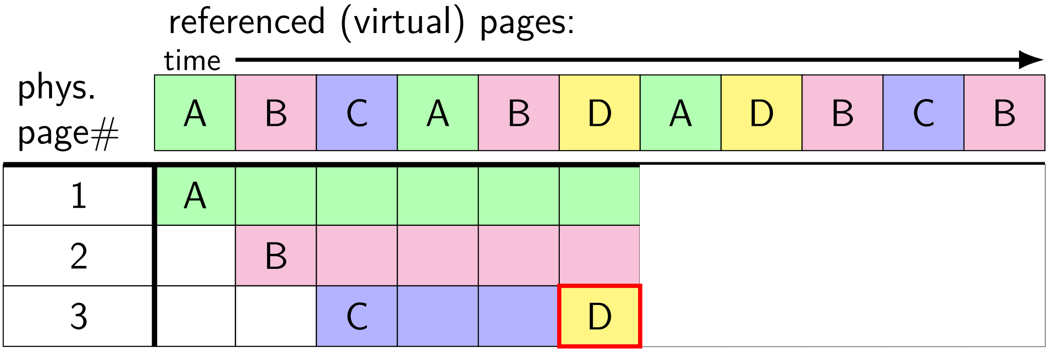
least recently used (the good case)

referenced (virtual) pages:

time →

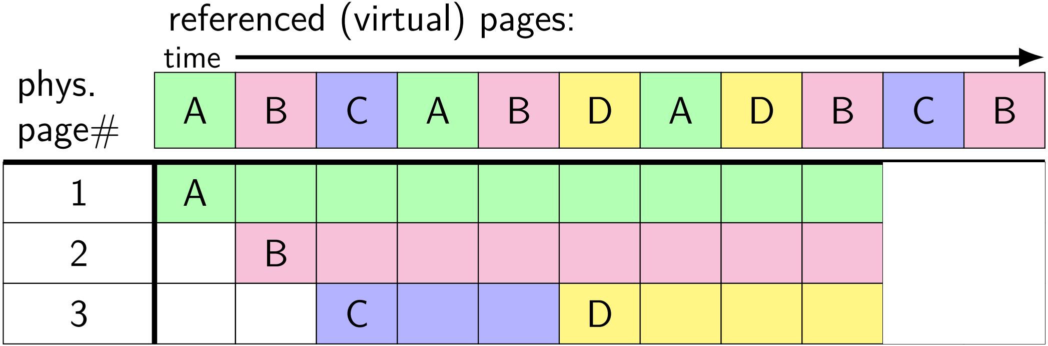
phys. page#	A	B	C	A	B	D	A	D	B	C	B
1	A										
2		B									
3			C								

least recently used (the good case)

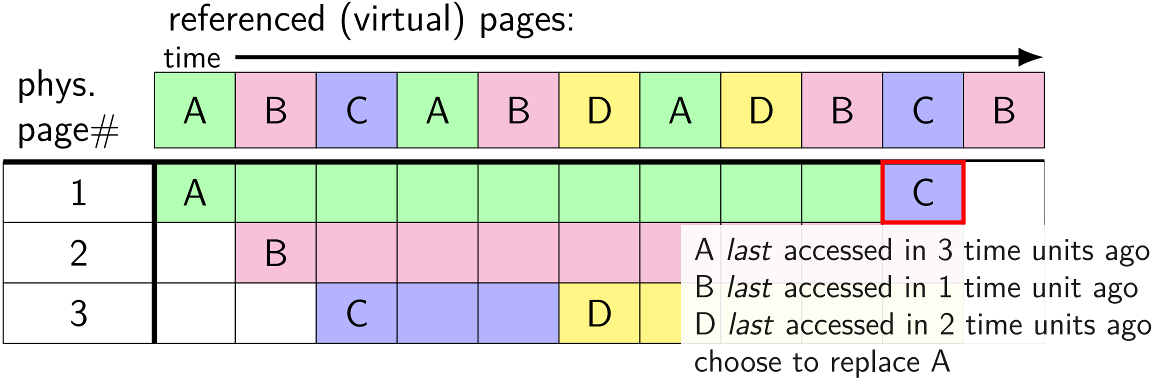


A *last* accessed 2 time units ago
B *last* accessed 1 time unit ago
C *last* accessed 3 time units ago
choose to replace C

least recently used (the good case)



least recently used (the good case)



least recently used (the good case)

referenced (virtual) pages:

time →

phys. page#	A	B	C	A	B	D	A	D	B	C	B
1	A									C	
2		B									
3			C			D					

least recently used (the worst case)

time →

phys. page#	A	B	C	D	A	B	C	D	A	B	C
1	A			D			C			B	
2		B			A			D			C
3			C			B			A		

least recently used (the worst case)

time →

phys. page#	A	B	C	D	A	B	C	D	A	B	C
1	A			D			C			B	
2		B			A			D			C
3			C			B			A		

8 replacements with LRU
versus 3 replacements with MIN:

1	A									B	
2		B					C				
3			C	D							

least recently used (exercise)

	A	B	A	D	C	B	D	B	C	D	A
1											
2											
3											

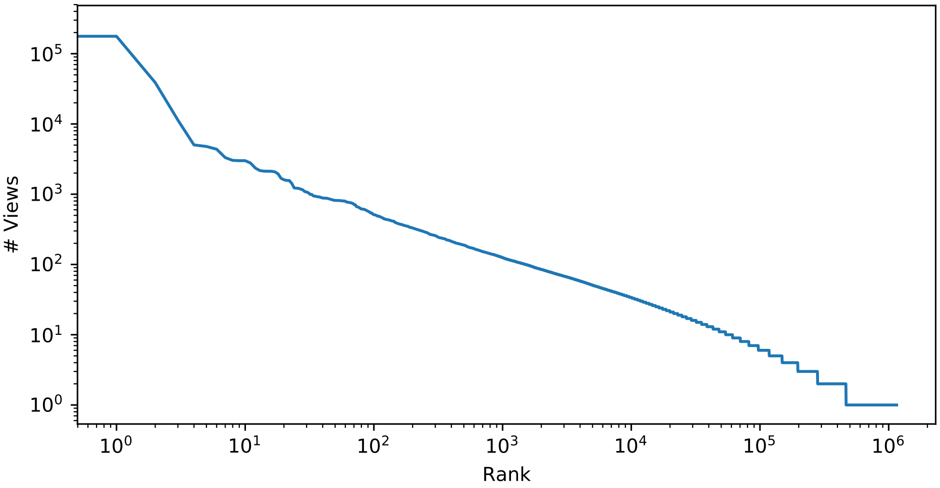
aside: Zipf model

working set model makes sense for programs

but not the only use of caches

example: Wikipedia — most popular articles

Wikipedia page views for 1 hour



NOTE: log-log-scale

Zipf distribution

Zipf distribution: straight line on log-log graph of rank v. count

a few items a much more popular than others

most caching benefit here

long tail: lots of items accessed a very small number of times

more cache less efficient — but does something

not like working set model, where there's just not more

good caching strategy for Zipf

keep the most recently popular things

up till what you have room for

- still benefit to caching things used 100 times/hour versus 1000

good caching strategy for Zipf

keep the most recently popular things

up till what you have room for

- still benefit to caching things used 100 times/hour versus 1000

LRU is okay — popular things always recently used

- seems to be what Wikipedia's caches do?

alternative policies for Zipf

least frequently used

- very simple policy
- if pure Zipf distribution — what you want
- practical problem: what about changes in popularity?

least frequently used + adjustments for 'recentness'

more?

pure LRU implementation

implementing LRU in software

maintain doubly-linked list of all physical pages

whenever a page is accessed:

- remove page from linked list, then
- add page to head of list

whenever a page needs to replaced:

- remove a page from the tail of the linked list, then
- evict that page from all page tables (and anything else)
- and use that page for whatever needs to be loaded

pure LRU implementation

implementing LRU in software

maintain doubly-linked list of all physical pages

whenever a page is accessed:

remove page from linked list, then

add

> need to run code on every access
> mechanism: make every access page fault
> which will make everything really slow

wheneve

rem

evict that page from all page tables (and anything else)

and use that page for whatever needs to be loaded

page fault for every access?

want every access to page fault? make every page invalid

...but want access to happen eventually

...which requires marking page as valid

...which makes future accesses not fault

page fault for every access?

want every access to page fault? make every page invalid

...but want access to happen eventually

...which requires marking page as valid

...which makes future accesses not fault

one solution: use debugging support to run one instruction

x86: “TF flag”

...then reset pages as invalid

page fault for every access?

want every access to page fault? make every page invalid

...but want access to happen eventually

...which requires marking page as valid

...which makes future accesses not fault

one solution: use debugging support to run one instruction

x86: “TF flag”

...then reset pages as invalid

okay, so I took something really slow and made it slower

so, what's practical

probably won't implement LRU — too slow

what can we practically do?

tools for tracking accesses

approximating LRU = "was this accessed recently"?

don't need to detect all accesses, only one recent one

tools for tracking accesses

approximating LRU = “was this accessed recently”?

don’t need to detect all accesses, only one recent one

ways to detect accesses:

- mark page invalid, if page fault happens make valid and record ‘accessed’
- ‘accessed’ or ‘referenced’ bit set by HW

tools for tracking accesses

approximating LRU = “was this accessed recently”?

don’t need to detect all accesses, only one recent one

ways to detect accesses:

- mark page invalid, if page fault happens make valid and record ‘accessed’
- ‘accessed’ or ‘referenced’ bit set by HW

usage: start detecting accesses,
if no access at all a little later — not recently accesssed

tools for tracking accesses

approximating LRU = "was this accessed recently"?

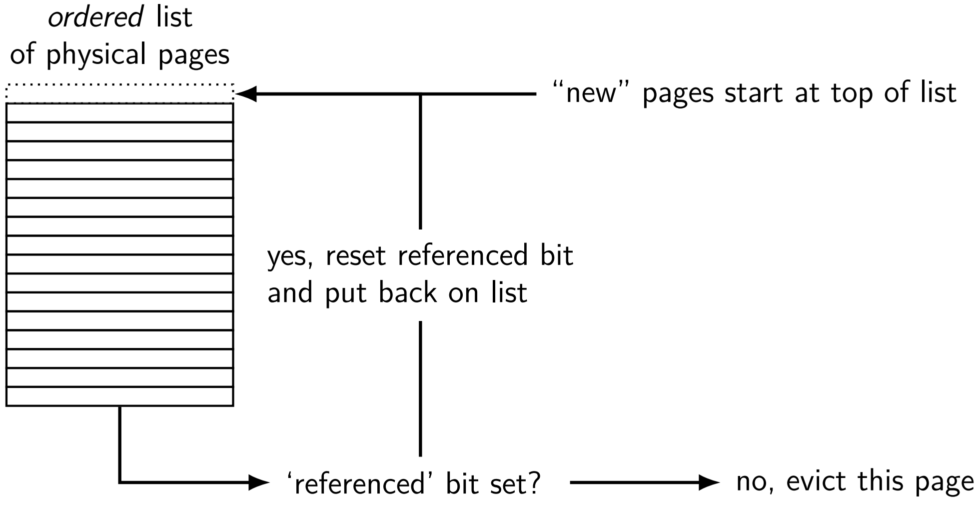
don't need to detect all accesses, only one recent one

ways to detect accesses:

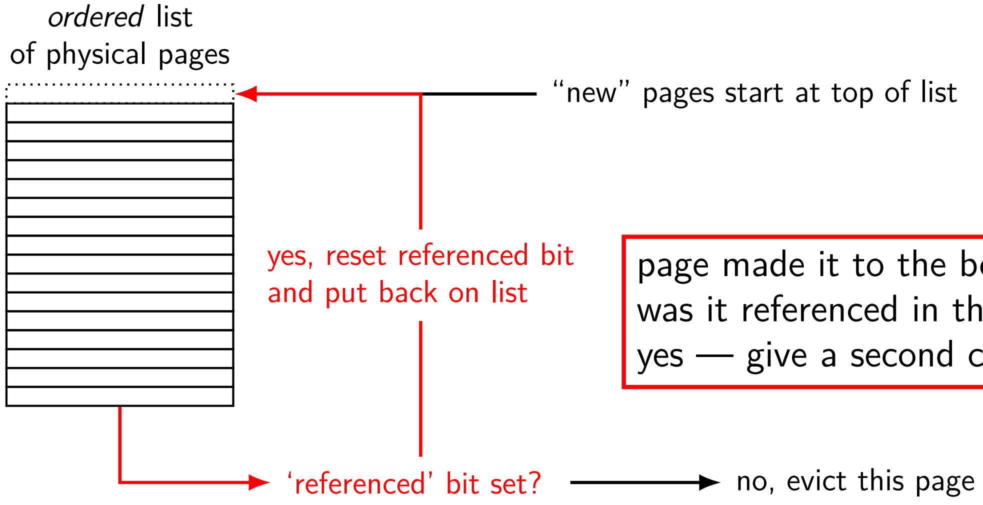
- mark page invalid, if page fault happens make valid and record 'accessed'
- 'accessed' or 'referenced' bit set by HW

usage: start detecting accesses,
if no access at all a little later — not recently accesssed

approximating LRU: second chance

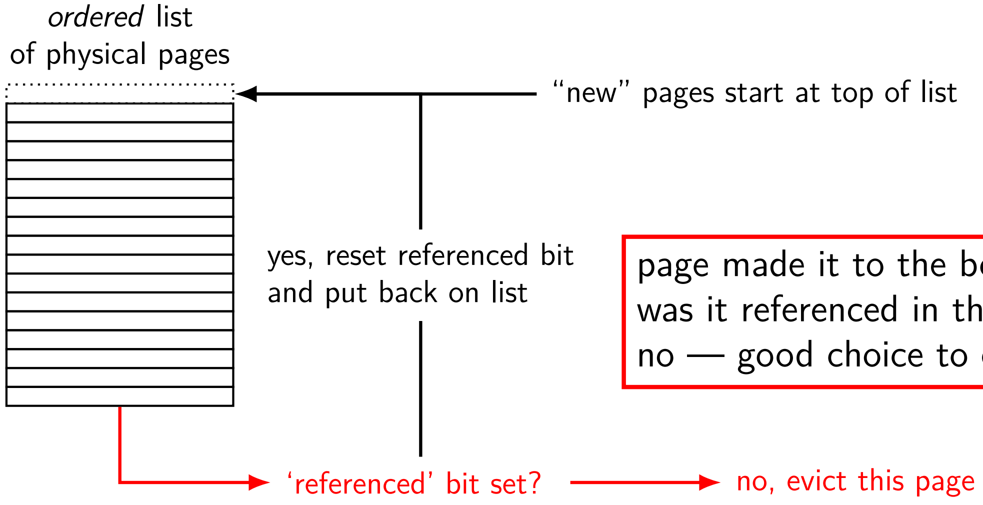


approximating LRU: second chance



page made it to the bottom
was it referenced in that time?
yes — give a second chance

approximating LRU: second chance



page made it to the bottom
was it referenced in that time?
no — good choice to evict

second chance example

	A	B	C	D	—	—	—	B	A	—	C	—
1	A						D					
2		B										C
3			C			C				A		
page list												
last added	*1R	*2R	*3R	1NR	2NR	3NR	*1R	1R	2NR	*3R	1NR	*2R
—	3NR	1R	2R	3R	1NR	2NR	3NR	3NR	1R	2NR	3R	1NR
end of list	2NR	3NR	1R	2R	3R	1NR	2NR	*2R	3NR	1R	2NR	3R

second chance example

page 2 was at bottom of list
is not referenced
okay to use

	A	B							A	—	C	—
1	A						D					
2		B										C
3			C			C				A		
page list												
last added	*1R	*2R	*3R	1NR	2NR	3NR	*1R	1R	2NR	*3R	1NR	*2R
—	3NR	1R	2R	3R	1NR	2NR	3NR	3NR	1R	2NR	3R	1NR
end of list	2NR	3NR	1R	2R	3R	1NR	2NR	*2R	3NR	1R	2NR	3R

second chance example

	A	B	C	D	—	—	—	B	A	—	C	—
1	A						D					
2		B										C
3			C			C				A		
page list												
last added	*1R	*2R	*3R	1NR	2NR	3NR	*1R	1R	2NR	*3R	1NR	*2R
—	3NR	1R	2R	3R	1NR	2NR	3NR	3NR	1R	2NR	3R	1NR
end of list	2NR	3NR	1R	2R	3R	1NR	2NR	*2R	3NR	1R	2NR	3R

second chance example

page 1 was at bottom of list
reference — give second chance
moves to top of list
clear referenced bit

	A	B							A	—	C	—
1	A						D					
2		B										C
3			C			C				A		
page list												
last added	*1R	*2R	*3R	1NR	2NR	3NR	*1R	1R	2NR	*3R	1NR	*2R
—	3NR	1R	2R	3R	1NR	2NR	3NR	3NR	1R	2NR	3R	1NR
end of list	2NR	3NR	1R	2R	3R	1NR	2NR	*2R	3NR	1R	2NR	3R

second chance example

eventually page 1 gets to bottom of list again
but now not referenced — use

	A									—	C	—
1	A						D					
2		B										C
3			C			C				A		
page list												
last added	*1R	*2R	*3R	1NR	2NR	3NR	*1R	1R	2NR	*3R	1NR	*2R
—	3NR	1R	2R	3R	1NR	2NR	3NR	3NR	1R	2NR	3R	1NR
end of list	2NR	3NR	1R	2R	3R	1NR	2NR	*2R	3NR	1R	2NR	3R

second chance example

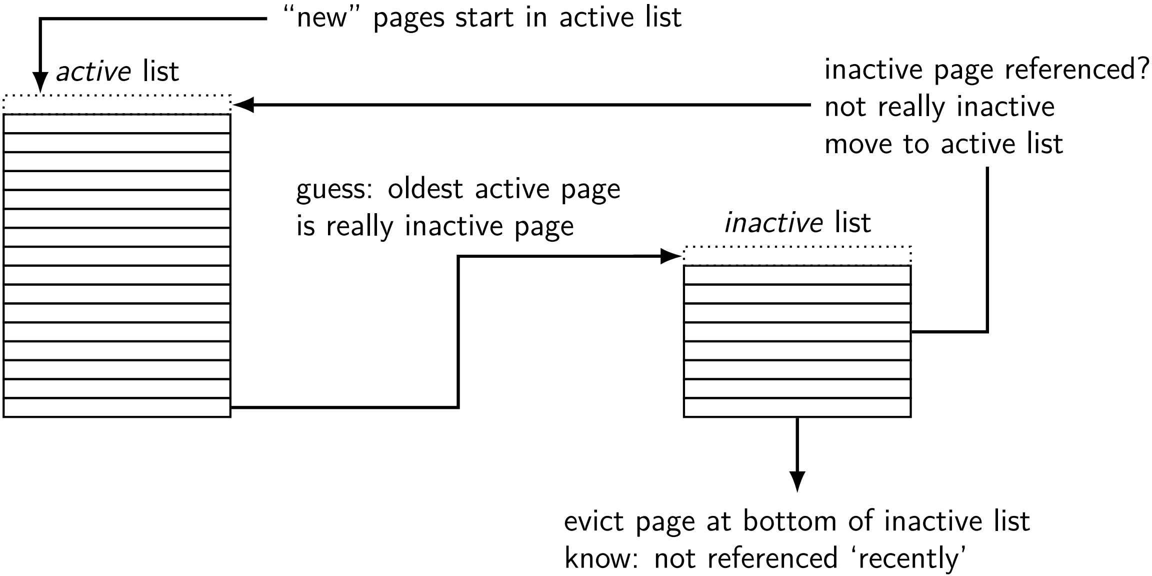
B referenced — flips referenced bit

	A									—	C	—
1	A						D					
2		B										C
3			C			C				A		
page list												
last added	*1R	*2R	*3R	1NR	2NR	3NR	*1R	1R	2NR	*3R	1NR	*2R
—	3NR	1R	2R	3R	1NR	2NR	3NR	3NR	1R	2NR	3R	1NR
end of list	2NR	3NR	1R	2R	3R	1NR	2NR	*2R	3NR	1R	2NR	3R

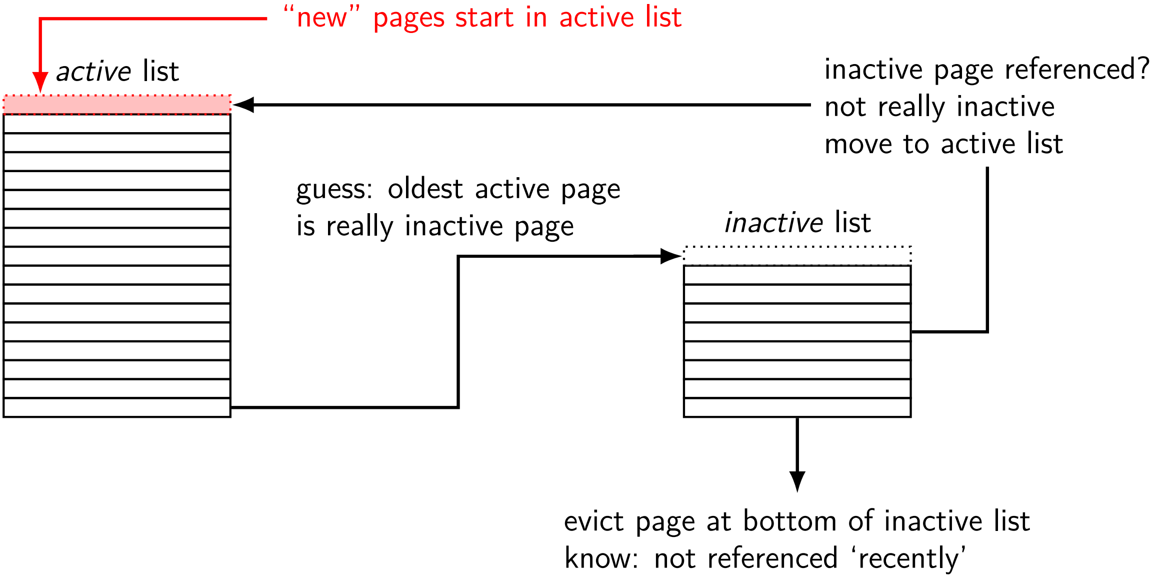
second chance example

	A	B	C	D	—	—	—	B	A	—	C	—
1	A						D					
2		B										C
3			C			C				A		
page list												
last added	*1R	*2R	*3R	1NR	2NR	3NR	*1R	1R	2NR	*3R	1NR	*2R
—	3NR	1R	2R	3R	1NR	2NR	3NR	3NR	1R	2NR	3R	1NR
end of list	2NR	3NR	1R	2R	3R	1NR	2NR	*2R	3NR	1R	2NR	3R

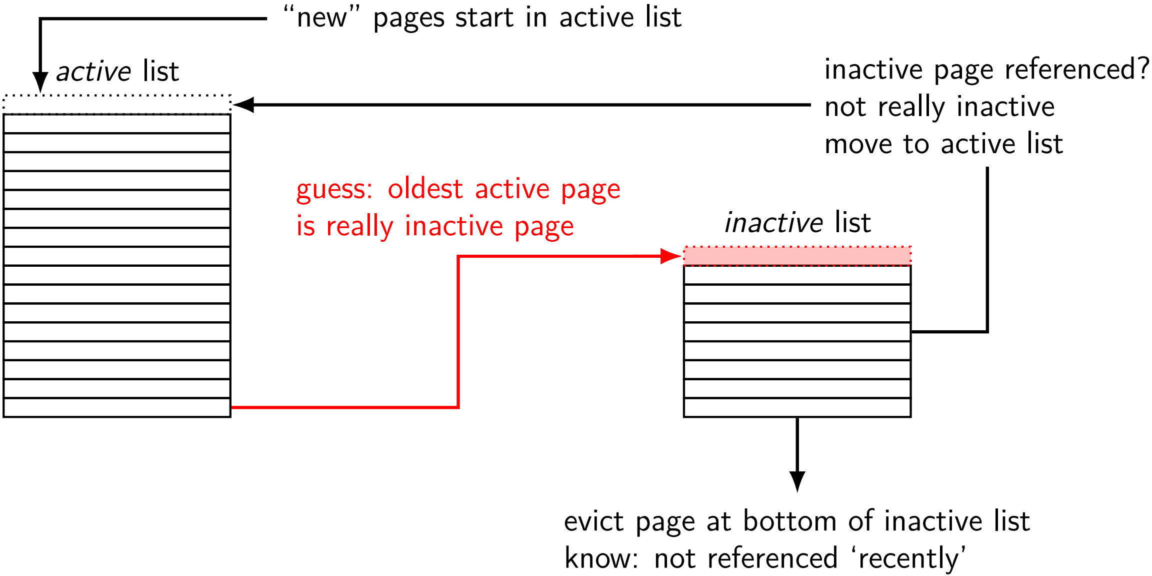
approximating LRU: SEQ



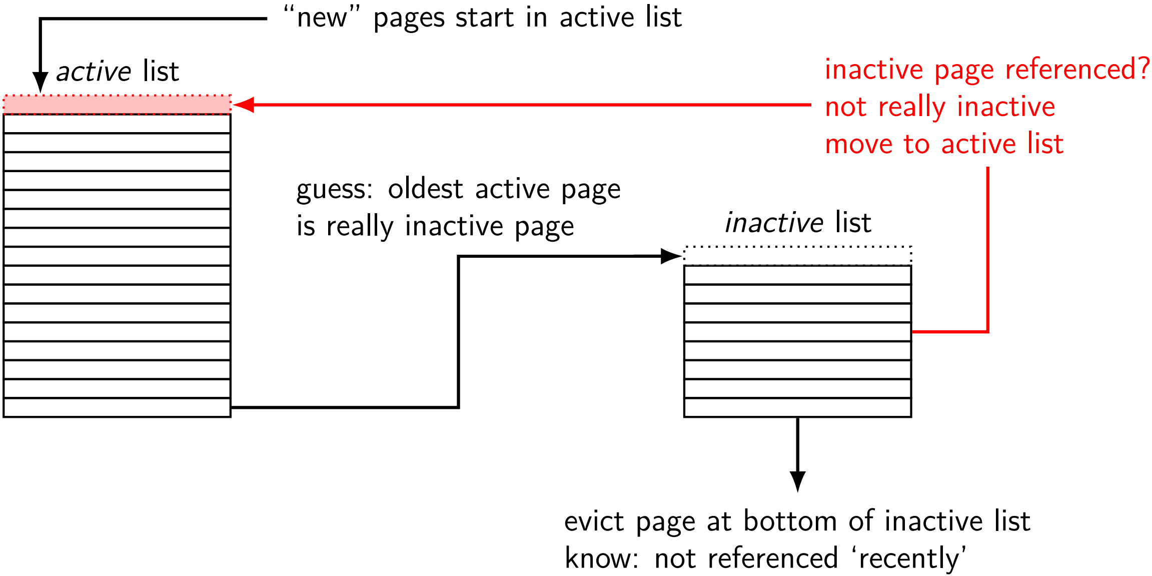
approximating LRU: SEQ



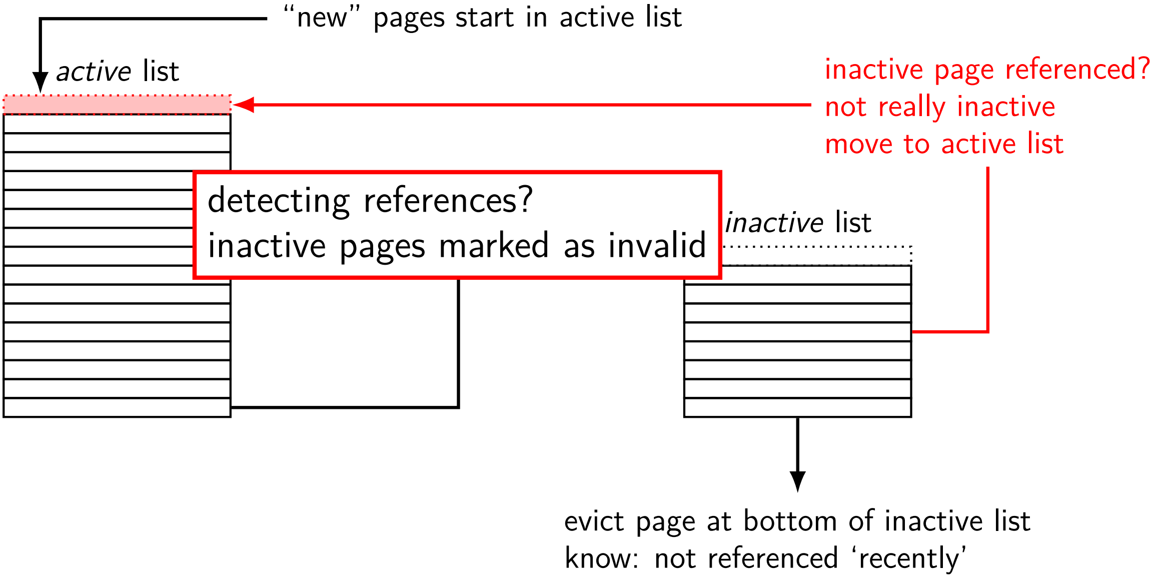
approximating LRU: SEQ



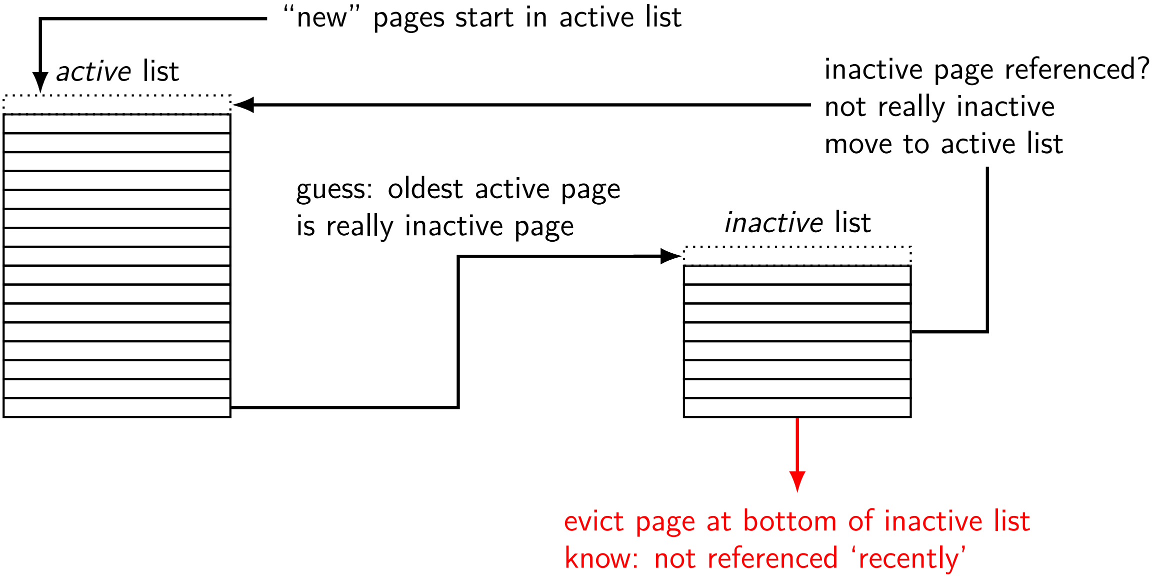
approximating LRU: SEQ



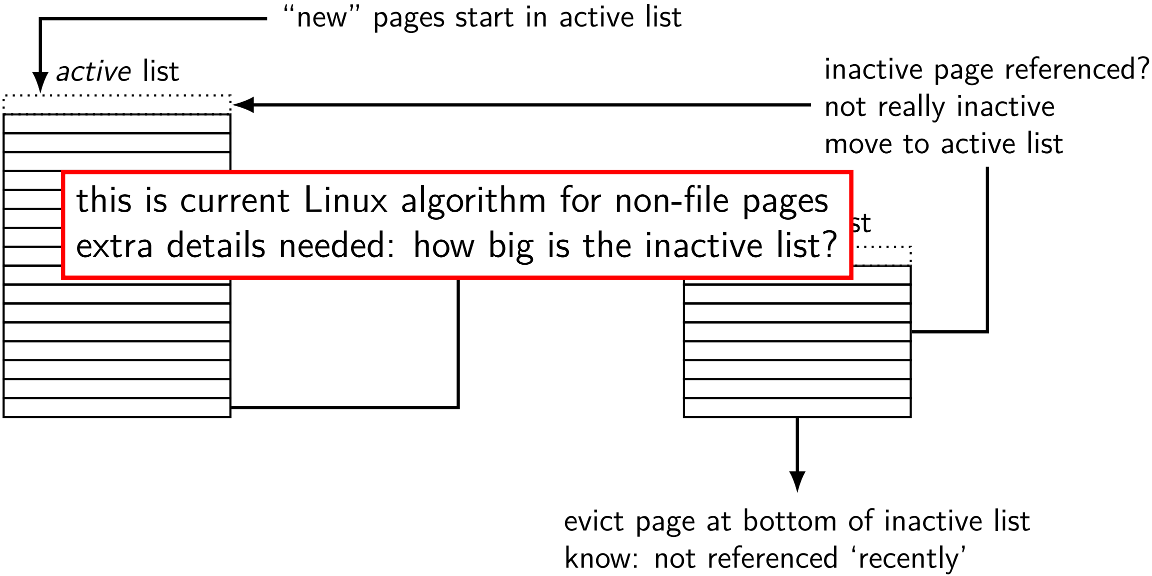
approximating LRU: SEQ



approximating LRU: SEQ



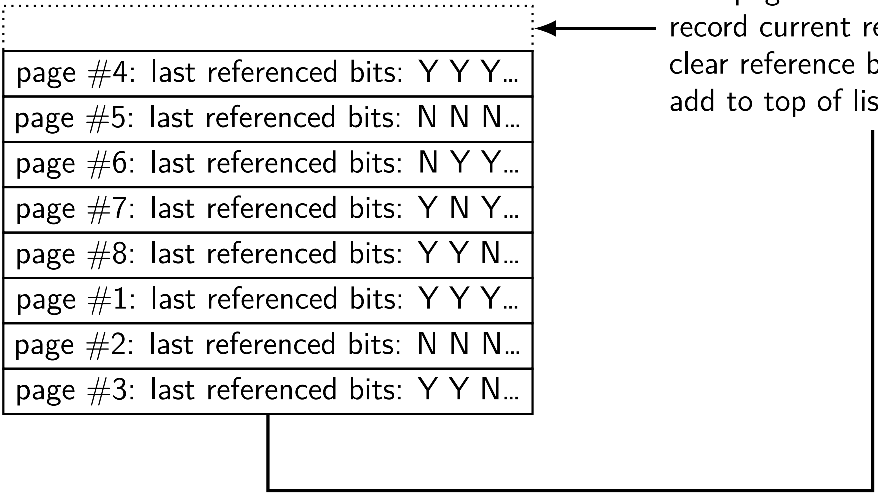
approximating LRU: SEQ



this is current Linux algorithm for non-file pages
extra details needed: how big is the inactive list?

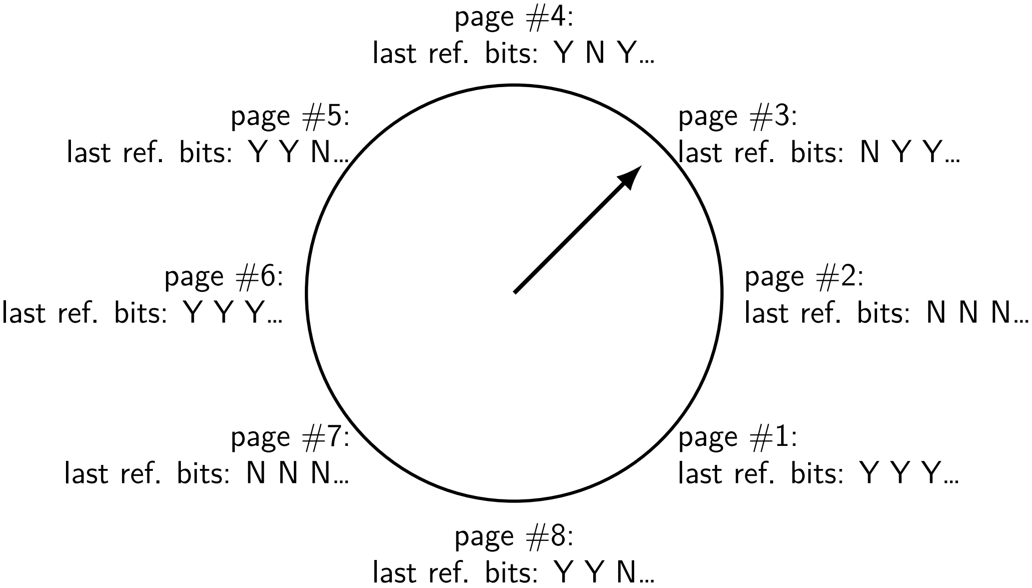
tracking usage: CLOCK (view 1)

ordered list
of physical pages



periodically:
take page from bottom of list
record current referenced bit
clear reference bit for next pass
add to top of list

tracking usage: CLOCK (view 2)



lazy replacement?

so far: don't do anything special until memory is full

only then is there a reason to writeback pages or evict pages

lazy replacement?

so far: don't do anything special until memory is full

only then is there a reason to writeback pages or evict pages

but real OSes are more proactive

non-lazy writeback

what happens when a computer loses power

how much data can you lose?

if we neve run out of memory…all of it?

    no changed data written back

solution: scan for dirty bits periodicially and writeback

non-lazy eviction

so far — allocating memory involves evicting pages

hopefully pages that haven't been used a long time anyways

could evict earlier "in the background" — means faster allocations

- probably wasn't using the CPU anyways

common strategy: maintain a small number of available pages

- might also make sure they start out pre-zeroed, etc.

problems with LRU

question: when does LRU perform poorly?

problems with LRU

question: when does LRU perform poorly?

only reading things once

repeated scans of large amounts of data

problems with LRU

question: when does LRU perform poorly?

only reading things once

repeated scans of large amounts of data

both common access patterns for files

CLOCK-Pro: special casing for one-use pages

by default, Linux tries to handle these patterns for file pages

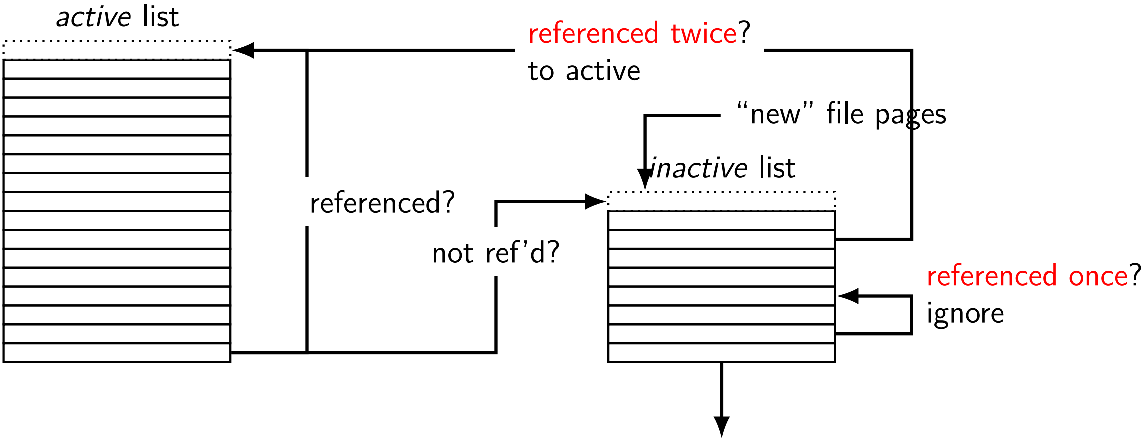
basic idea: don't consider pages active until the second access

single scans of file won't "pollute" cache

without this change: reading large files slows down other programs

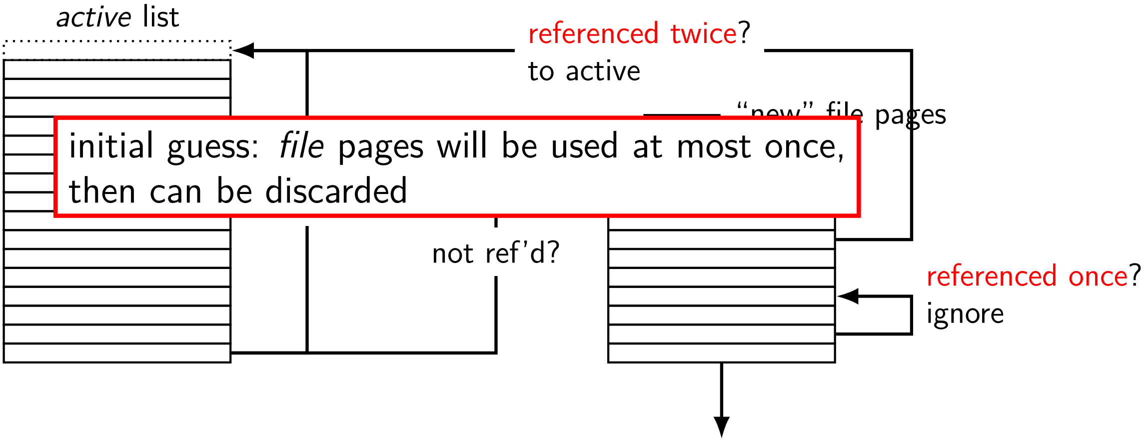
- recently read part of large file steals space from active programs

CLOCK-Pro: special casing for one-use pages



evict page at bottom of inactive list
either file page referenced once *or*
referenced multiple times, but not recently

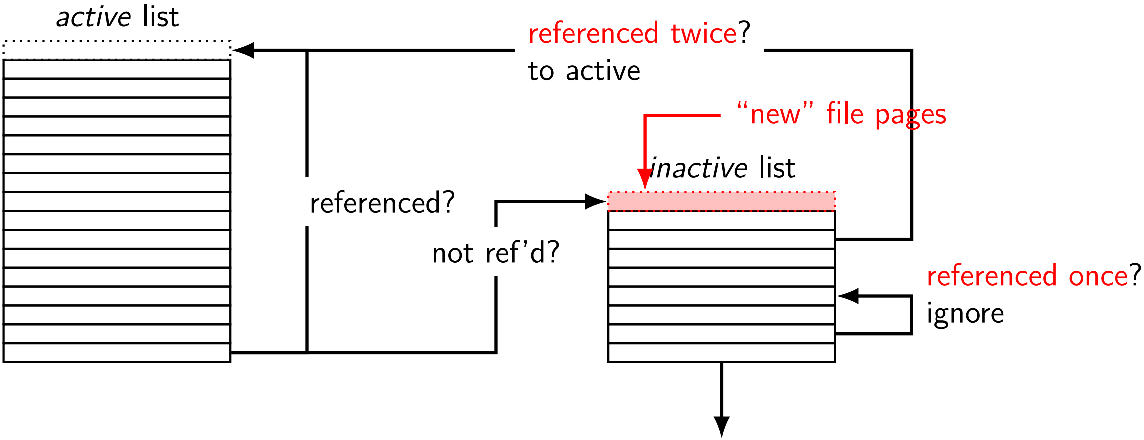
CLOCK-Pro: special casing for one-use pages



initial guess: *file* pages will be used at most once, then can be discarded

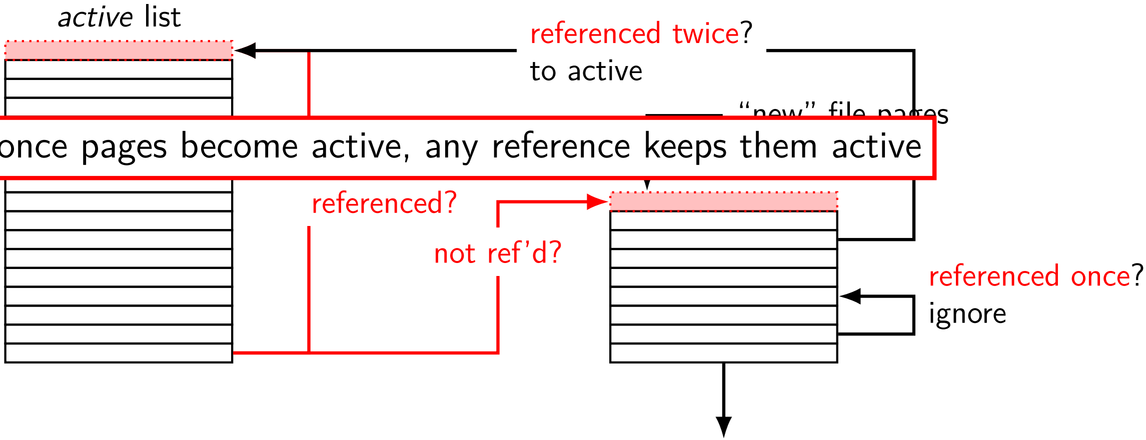
evict page at bottom of inactive list
either file page referenced once *or*
referenced multiple times, but not recently

CLOCK-Pro: special casing for one-use pages



evict page at bottom of inactive list
either file page referenced once *or*
referenced multiple times, but not recently

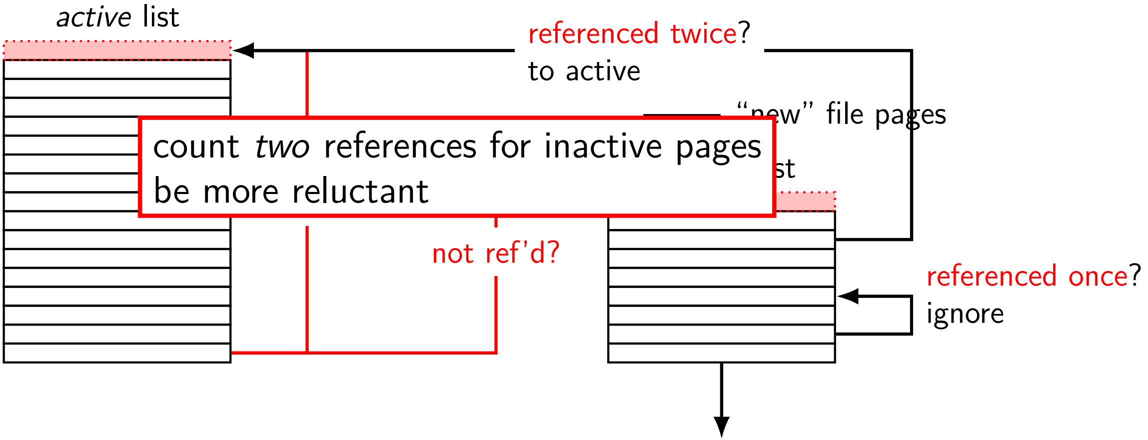
CLOCK-Pro: special casing for one-use pages



once pages become active, any reference keeps them active

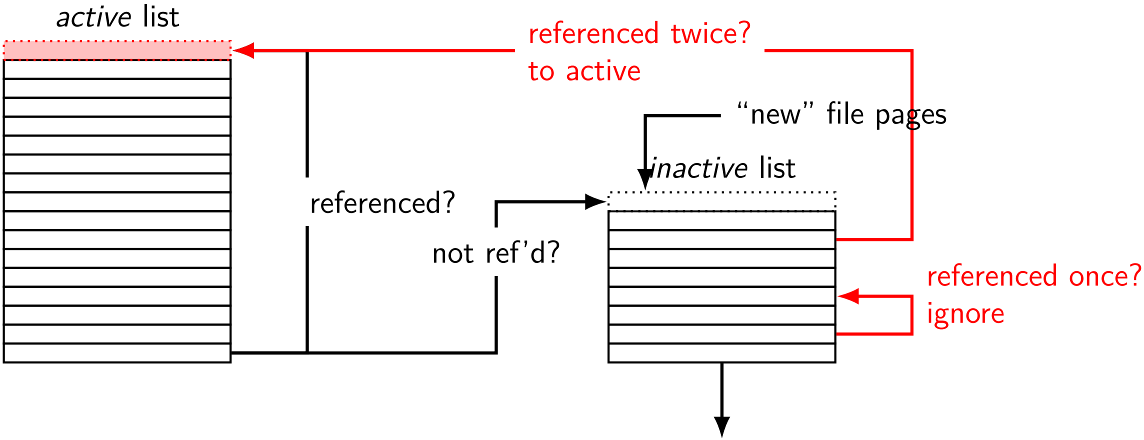
evict page at bottom of inactive list
either file page referenced once *or*
referenced multiple times, but not recently

CLOCK-Pro: special casing for one-use pages



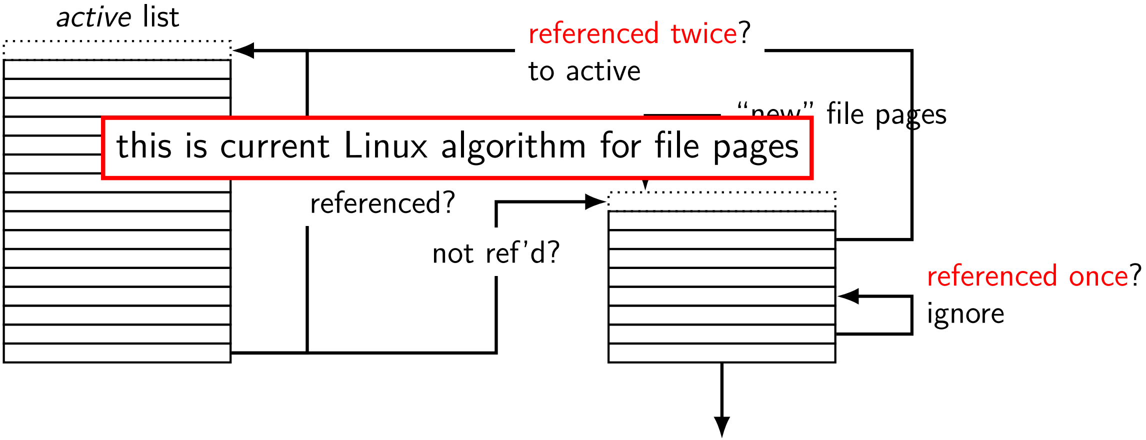
evict page at bottom of inactive list
either file page referenced once *or*
referenced multiple times, but not recently

CLOCK-Pro: special casing for one-use pages



evict page at bottom of inactive list
either file page referenced once *or*
referenced multiple times, but not recently

CLOCK-Pro: special casing for one-use pages



evict page at bottom of inactive list
either file page referenced once *or*
referenced multiple times, but not recently

default Linux page replacement summary

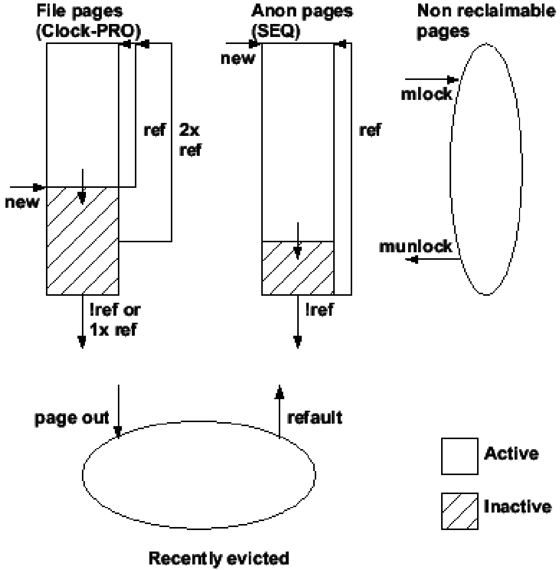


Figure: https://linux-mm.org/PageReplacementDesign

default Linux page replacement summary

identify *inactive* pages — guess: not going to be accessed soon

- file pages which haven't been accessed more than once, or
- any pages which haven't been accessed recently

some minimum threshold of inactive pages

- add to inactive list in background
- mark inactive pages as invalid to detect use quickly

oldest inactive page still not used → evict that one

- otherwise: give it a second chance

backup slides

swapping decisions

write policy

replacement policy

swapping decisions

write policy

replacement policy

swapping is writeback

implementing write-through is hard

- when fault happens — physical page not written
- when OS resumes process — no chance to forward write
- HW itself doesn't know how to write to disk

write-through would also be really slow

- HDD/SSD perform best if one writes at least a whole page at a time

implementing writeback

need a *dirty bit* per page ("was page modified")

x86: kept in the page table!

option 1 (most common): hardware sets dirty bit in page table entry (on write)

- bit means "physical page was modified using this PTE"

option 2: OS sets page read-only, flips read-only+dirty bit on fault

swapping decisions

write policy

replacement policy

replacement policies really matter

huge cost for "miss" on swapping (milliseconds!)

replacement policy implemented in software

- a lot more room for fancy policies

usualy goal: least-recently-used approximation

LRU replacement?

problem: need to identify when pages are used

- ideally every single time

not practical to do this exactly

- HW would need to keep a list of when each page was accessed, or
- SW would need to force every access to trigger a fault

second chance example

A	B	C	D	—	—	—	B	A	—	C	—

1	A						D					
2		B										C
3			C			C				A		
page list												
last added	*1R	*2R	*3R	1NR	2NR	3NR	*1R	1R	2NR	*3R	1NR	*2R
—	3NR	1R	2R	3R	1NR	2NR	3NR	3NR	1R	2NR	3R	1NR
end of list	2NR	3NR	1R	2R	3R	1NR	2NR	*2R	3NR	1R	2NR	3R

second chance example

page 2 was at bottom of list
is not referenced
okay to use

	A	B							A	—	C	—
1	A						D					
2		B										C
3			C			C				A		
page list												
last added	*1R	*2R	*3R	1NR	2NR	3NR	*1R	1R	2NR	*3R	1NR	*2R
—	3NR	1R	2R	3R	1NR	2NR	3NR	3NR	1R	2NR	3R	1NR
end of list	2NR	3NR	1R	2R	3R	1NR	2NR	*2R	3NR	1R	2NR	3R

second chance example

	A	B	C	D	—	—	—	B	A	—	C	—
1	A						D					
2		B										C
3			C			C				A		
page list												
last added	*1R	*2R	*3R	1NR	2NR	3NR	*1R	1R	2NR	*3R	1NR	*2R
—	3NR	1R	2R	3R	1NR	2NR	3NR	3NR	1R	2NR	3R	1NR
end of list	2NR	3NR	1R	2R	3R	1NR	2NR	*2R	3NR	1R	2NR	3R

second chance example

page 1 was at bottom of list
reference — give second chance
moves to top of list
clear referenced bit

	A	B							A	—	C	—
1	A						D					
2		B										C
3			C			C				A		
page list												
last added	*1R	*2R	*3R	1NR	2NR	3NR	*1R	1R	2NR	*3R	1NR	*2R
—	3NR	1R	2R	3R	1NR	2NR	3NR	3NR	1R	2NR	3R	1NR
end of list	2NR	3NR	1R	2R	3R	1NR	2NR	*2R	3NR	1R	2NR	3R

second chance example

eventually page 1 gets to bottom of list again
but now not referenced — use

											C	—
1	A						D					
2		B										C
3			C			C				A		
page list												
last added	*1R	*2R	*3R	1NR	2NR	3NR	*1R	1R	2NR	*3R	1NR	*2R
—	3NR	1R	2R	3R	1NR	2NR	3NR	3NR	1R	2NR	3R	1NR
end of list	2NR	3NR	1R	2R	3R	1NR	2NR	*2R	3NR	1R	2NR	3R

second chance example

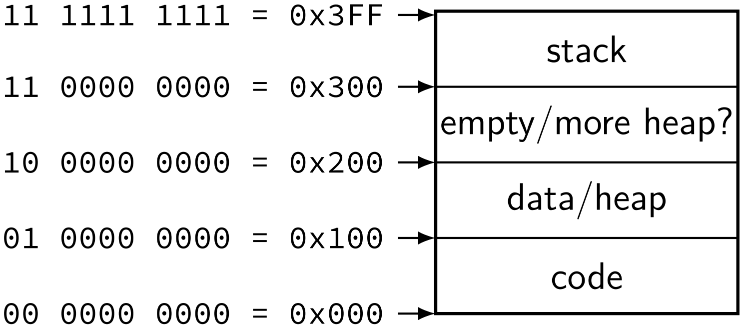
B referenced — flips referenced bit

	A									—	C	—
1	A						D					
2		B										C
3			C			C				A		
page list												
last added	*1R	*2R	*3R	1NR	2NR	3NR	*1R	1R	2NR	*3R	1NR	*2R
—	3NR	1R	2R	3R	1NR	2NR	3NR	3NR	1R	2NR	3R	1NR
end of list	2NR	3NR	1R	2R	3R	1NR	2NR	*2R	3NR	1R	2NR	3R

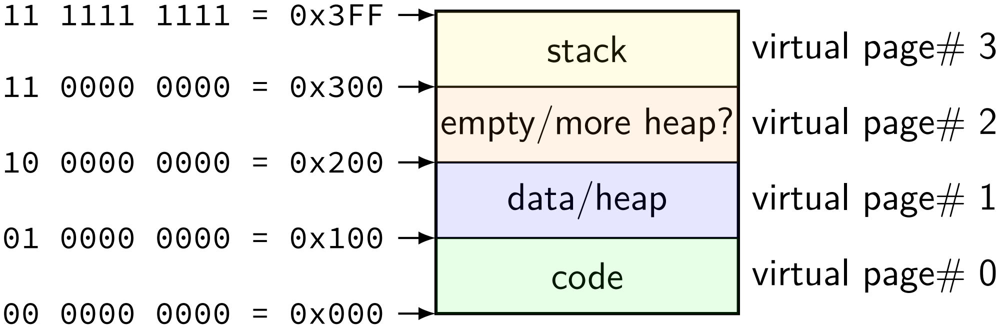
second chance example

	A	B	C	D	—	—	—	B	A	—	C	—
1	A						D					
2		B										C
3			C			C				A		
page list												
last added	*1R	*2R	*3R	1NR	2NR	3NR	*1R	1R	2NR	*3R	1NR	*2R
—	3NR	1R	2R	3R	1NR	2NR	3NR	3NR	1R	2NR	3R	1NR
end of list	2NR	3NR	1R	2R	3R	1NR	2NR	*2R	3NR	1R	2NR	3R

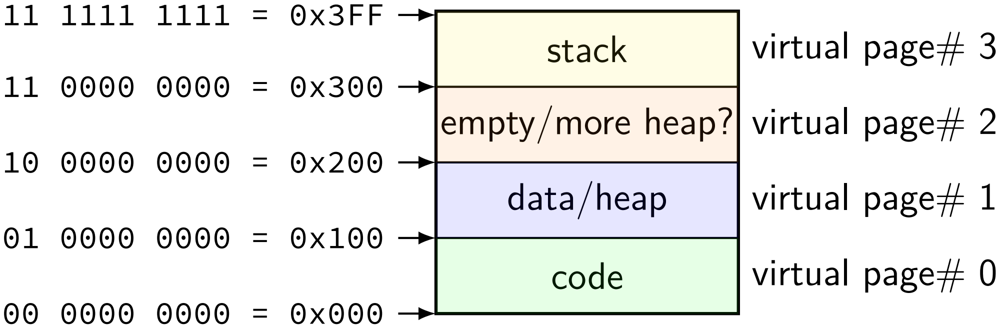
toy program memory



toy program memory

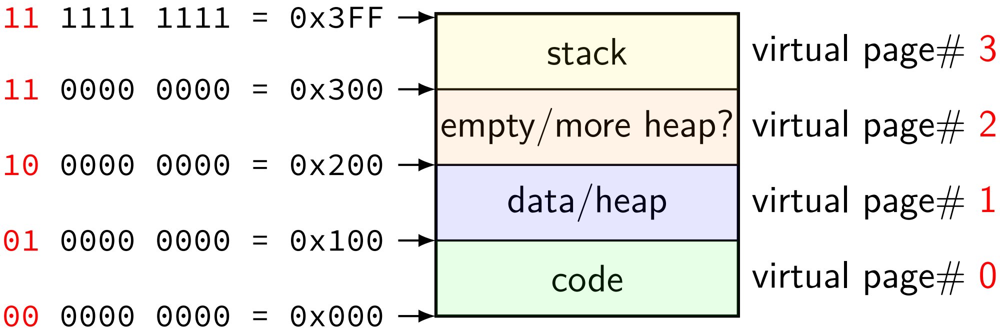


toy program memory



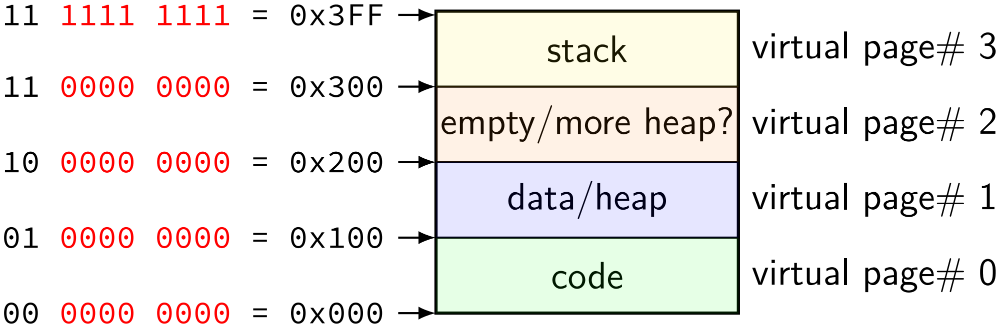
divide memory into pages (2^8 bytes in this case)
"virtual" = addresses the program sees

toy program memory



page number is upper bits of address
(because page size is power of two)

toy program memory



rest of address is called page offset

toy physical memory

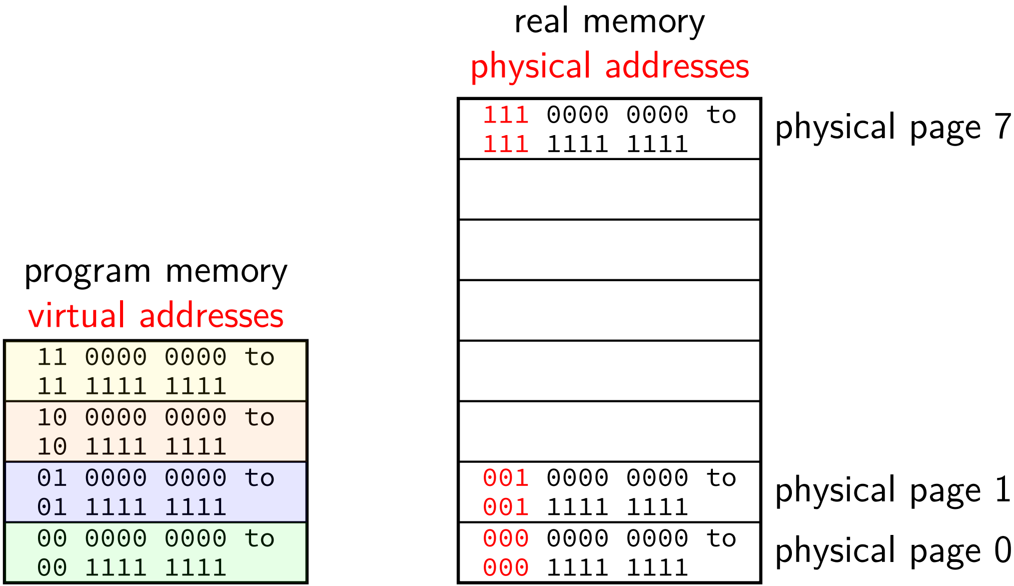
real memory
physical addresses

111 0000 0000 to 111 1111 1111
001 0000 0000 to 001 1111 1111
000 0000 0000 to 000 1111 1111

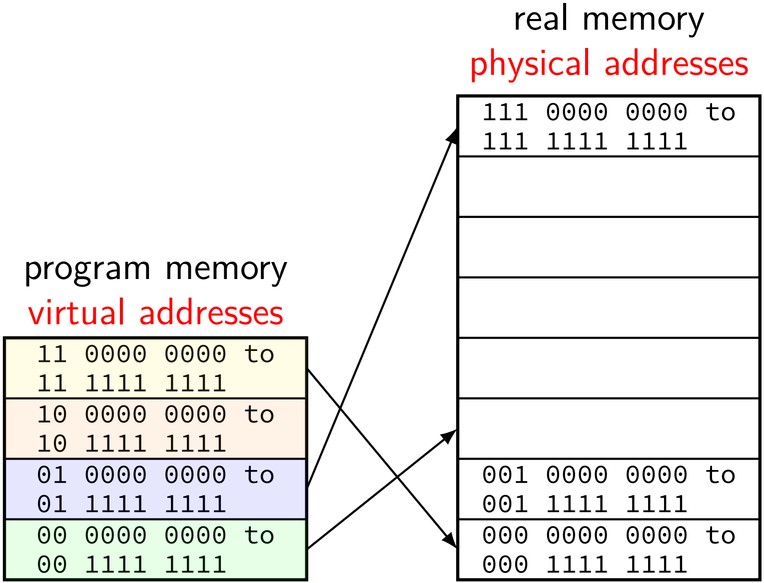
program memory
virtual addresses

11 0000 0000 to 11 1111 1111
10 0000 0000 to 10 1111 1111
01 0000 0000 to 01 1111 1111
00 0000 0000 to 00 1111 1111

toy physical memory

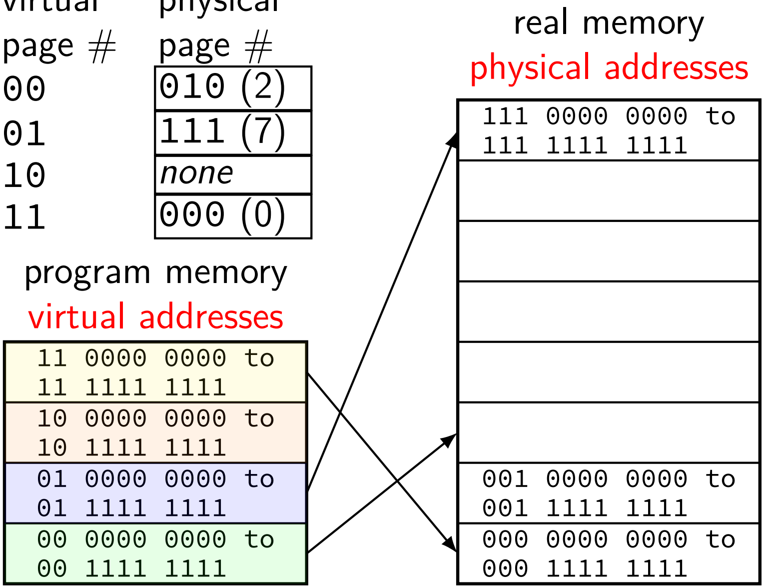


toy physical memory



toy physical memory

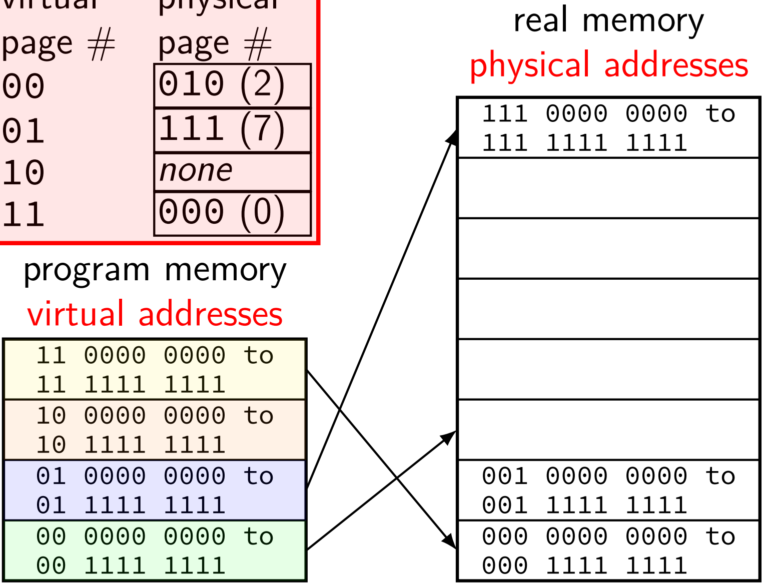
virtual page #	physical page #
00	010 (2)
01	111 (7)
10	*none*
11	000 (0)



toy physical memory

page table!

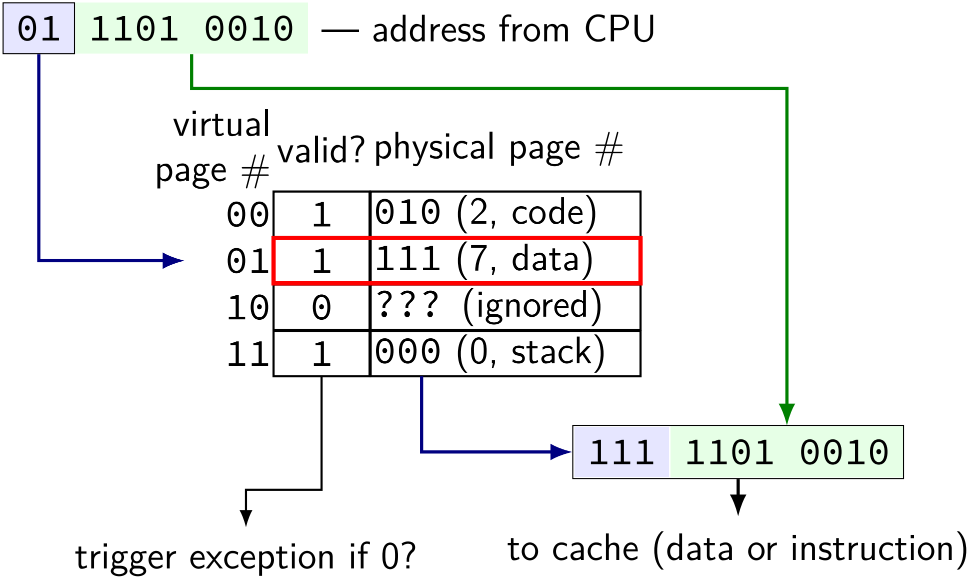
virtual page #	physical page #
00	010 (2)
01	111 (7)
10	*none*
11	000 (0)



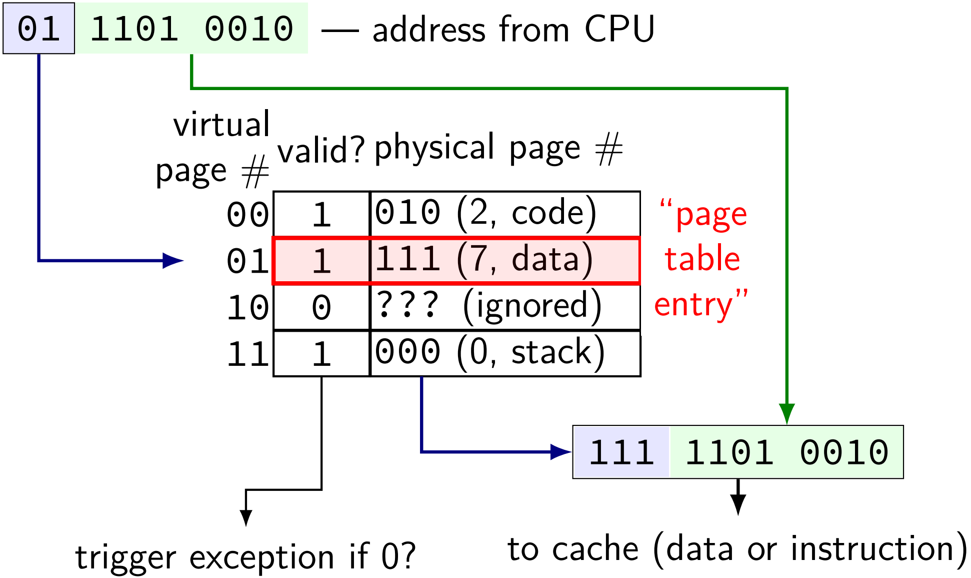
toy page table lookup

virtual page #	valid?	physical page #
00	1	010 (2, code)
01	1	111 (7, data)
10	0	??? (ignored)
11	1	000 (0, stack)

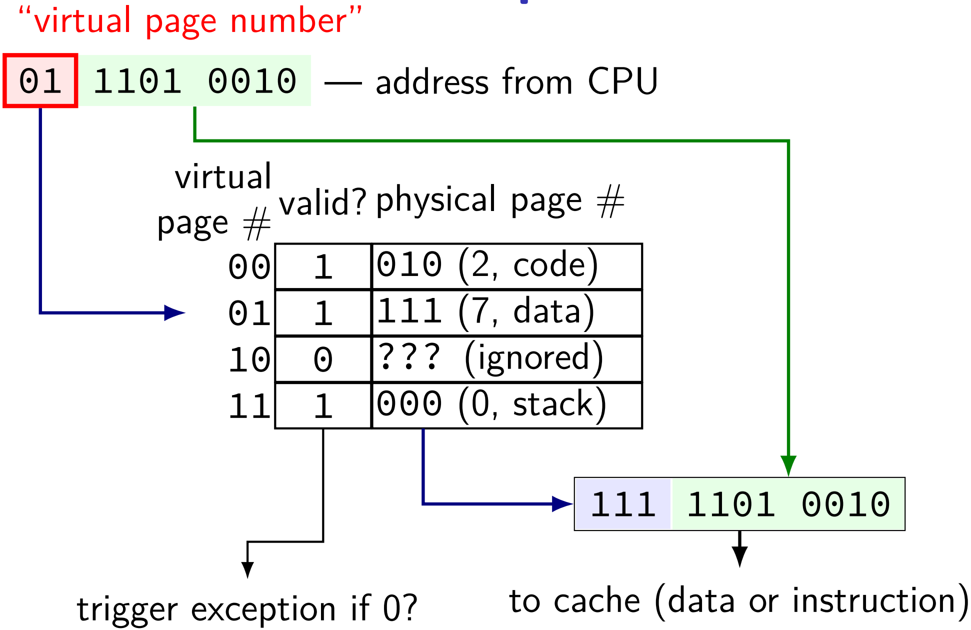
toy page table lookup



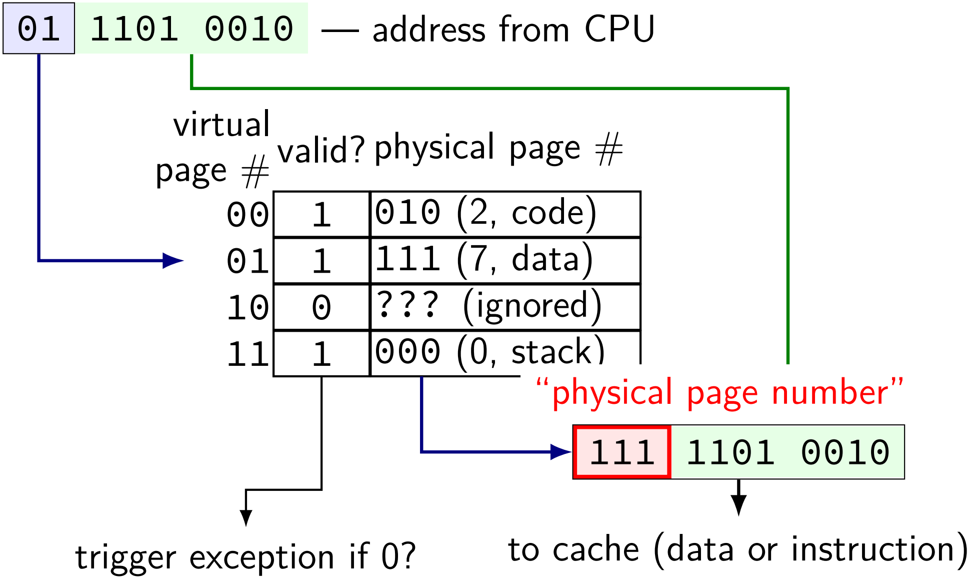
toy page table lookup



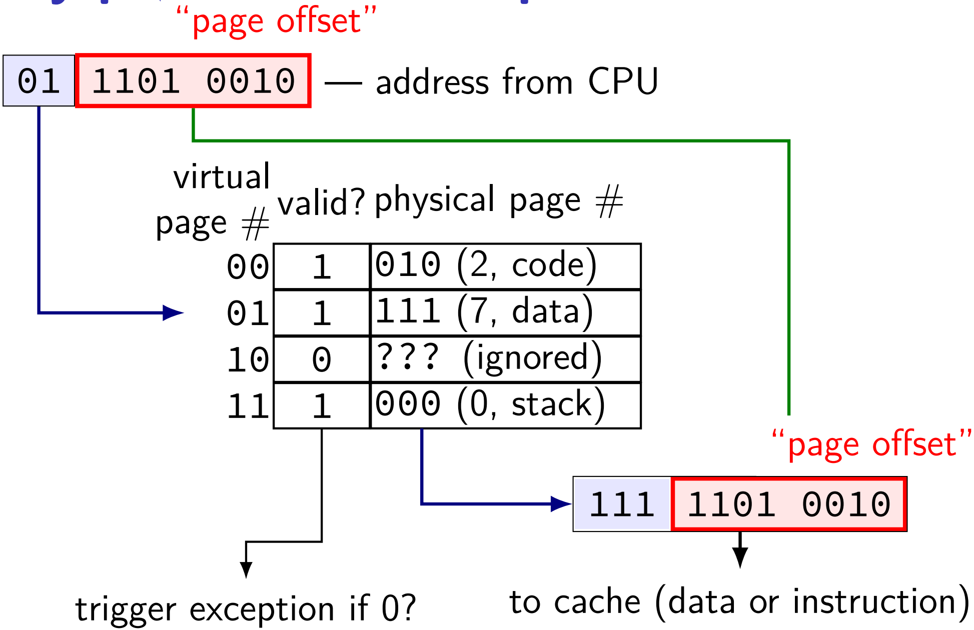
toy page table lookup



toy page table lookup

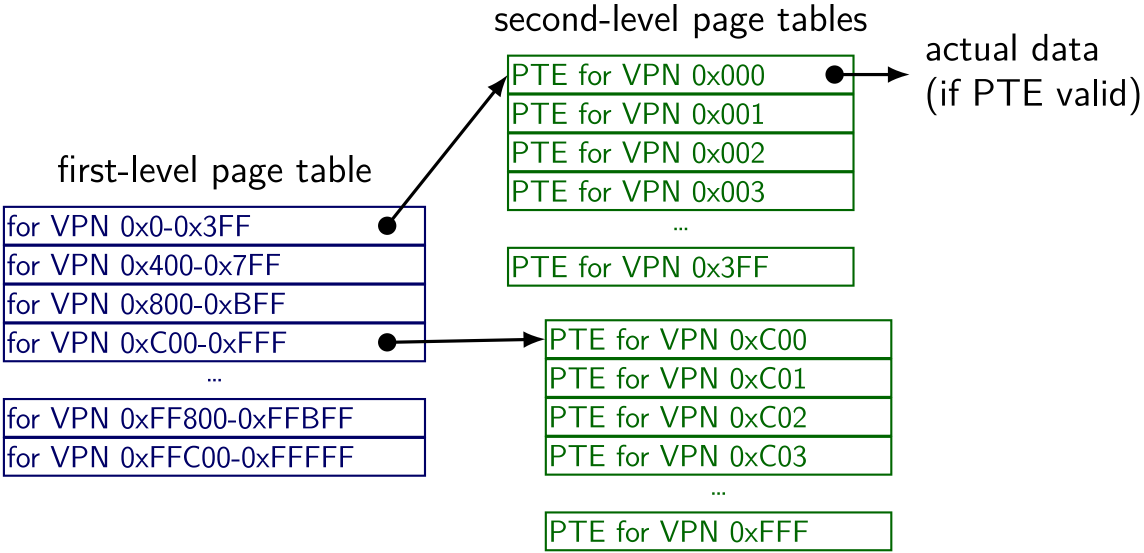


toy page table lookup



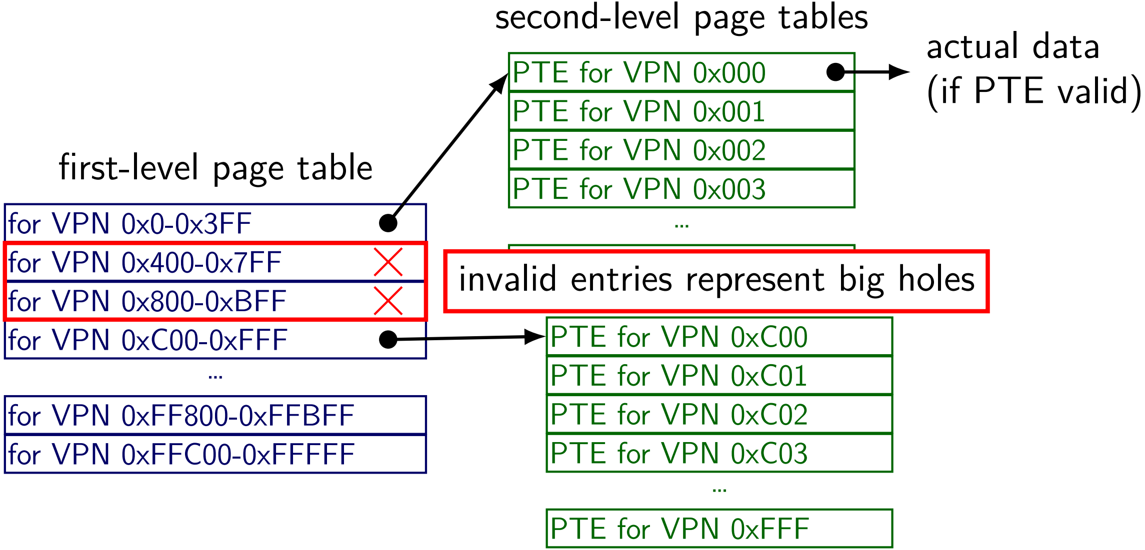
two-level page tables

two-level page table; 2^{20} pages total; 2^{10} entries per table



two-level page tables

two-level page table; 2^{20} pages total; 2^{10} entries per table



two-level page tables

two-level page table: 2^{20} pages total; 2^{10} entries per table

first-level pag

for VPN 0x0-0x3FF

for VPN 0x400-0x7F

for VPN 0x800-0xBF

for VPN 0xC00-0xFF

...

for VPN 0xFF800-0x

for VPN 0xFFC00-0xFFFFF

first-level page table

VPN range	valid	user?	write?	physical page # (of next page table)
0x0–0x3FF	1	1	1	0x22343
0x400–0x7FF	0	0	1	0x00000
0x800–0xBFF	0	0	0	0x00000
0xC00–0xFFF	1	1	0	0x33454
0x1000–0x13FF	1	1	0	0xFF043
…	…	…	…	…
0xFFC00–0xFFFFF	1	1	0	0xFF045

PTE for VPN 0xC03

...

PTE for VPN 0xFFF

two-level page tables

two-level page table: 2^{20} pages total; 2^{10} entries per table

first-level pag

for VPN 0x0-0x3FF

for VPN 0x400-0x7F

for VPN 0x800-0xBF

for VPN 0xC00-0xFF

...

for VPN 0xFF800-0x

for VPN 0xFFC00-0xFFFFF

PTE for VPN 0xC03

...

PTE for VPN 0xFFF

first-level page table

VPN range	valid	user?	write?	physical page # (of next page table)
0x0–0x3FF	1	1	1	0x22343
0x400–0x7FF	0	0	1	0x00000
0x800–0xBFF	0	0	0	0x00000
0xC00–0xFFF	1	1	0	0x33454
0x1000–0x13FF	1	1	0	0xFF043
…	…	…	…	…
0xFFC00–0xFFFFF	1	1	0	0xFF045

two-level page tables

two-level page table; 2^{20} pages total; 2^{10} entries per table

first-level page table

VPN range	valid	user?	write?	physical page # (of next page table)
0x0-0x3FF	1	1	1	0x22343
0x400-0x7FF	0	0	1	0x00000
0x800-0xBFF	0	0	0	0x00000
0xC00-0xFFF	1	1	0	0x33454
0x1000-0x13FF	1	1	0	0xFF043
…	…	…	…	…
0xFFC00-0xFFFFF	1	1	0	0xFF045

first-level pag

for VPN 0x0-0x3FF

for VPN 0x400-0x7F

for VPN 0x800-0xBF

for VPN 0xC00-0xFF

…

for VPN 0xFF800-0x

for VPN 0xFFC00-0xFFFFF

PTE for VPN 0xC03

…

PTE for VPN 0xFFF

two-level page tables

two-level page table; 2^{20} pages total; 2^{10} entries per table

first-level page table

for VPN 0x0-0x3FF
for VPN 0x400-0x7FF
for VPN 0x800-0xBFF
for VPN 0xC00-0xFFF
…
for VPN 0xFF800-0xFFBFF
for VPN 0xFFC00-0xFFFFF

a second-level page table

VPN	valid	user?	write?	physical page # (of data)
0xC00	1	1	0	0x42443
0xC01	1	1	0	0x4A9DE
0xC02	1	1	0	0x5C001
0xC03	0	0	0	0x00000
0xC04	1	1	0	0x6C223
…	…	…	…	…
0xFFF	0	0	0	0x00000

PTE for VPN 0xC03

…

PTE for VPN 0xFFF

two-level page tables

two-level page table; 2^{20} pages total; 2^{10} entries per table

first-level page table

- for VPN 0x0-0x3FF
- for VPN 0x400-0x7FF (✗)
- for VPN 0x800-0xBFF (✗)
- for VPN 0xC00-0xFFF

...

- for VPN 0xFF800-0xFFBFF
- for VPN 0xFFC00-0xFFFFF

a second-level page table

VPN	valid	user?	write?	physical page # (of data)
0xC00	1	1	0	0x42443
0xC01	1	1	0	0x4A9DE
0xC02	1	1	0	0x5C001
0xC03	0	0	0	0x00000
0xC04	1	1	0	0x6C223
...	...	...	...	...
0xFFF	0	0	0	0x00000

PTE for VPN 0xC03

...

PTE for VPN 0xFFF

two-level page tables

two-level page table; 2^{20} pages total; 2^{10} entries per table