virtual memory 4

last time

mmap: files appear as part of memory

shared mmap: physical pages assigned = pages read()/write() uses

private mmap: copy-on-write done for pages

processes memory as bunch of mmap calls

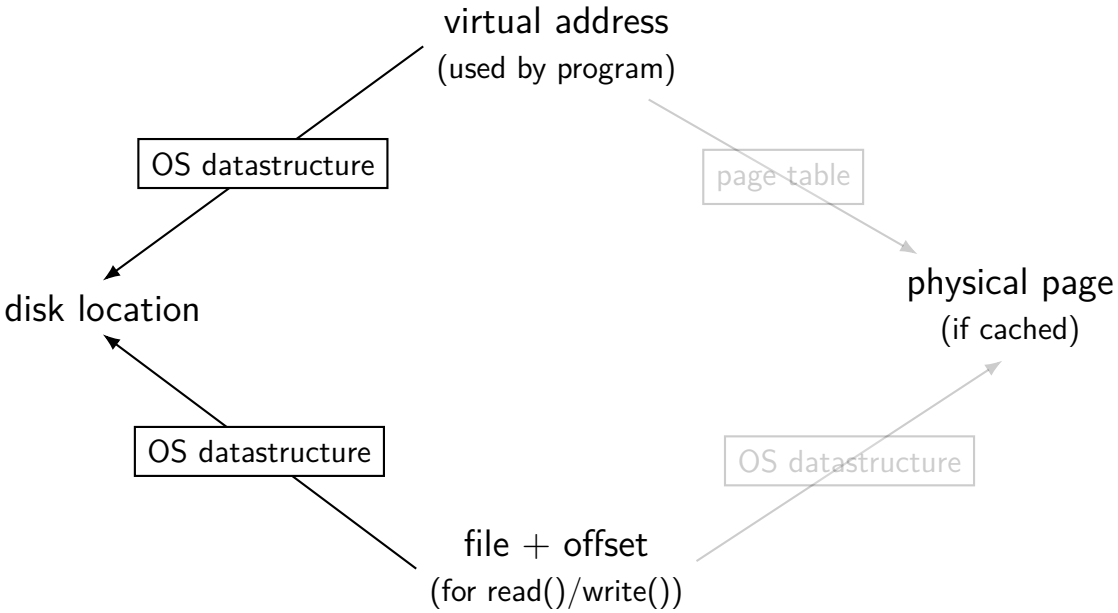
page cache: memory = cache for data on disk

- data on disk = all files + program data
- (program data (e.g. heap) assigned place on disk when needed)

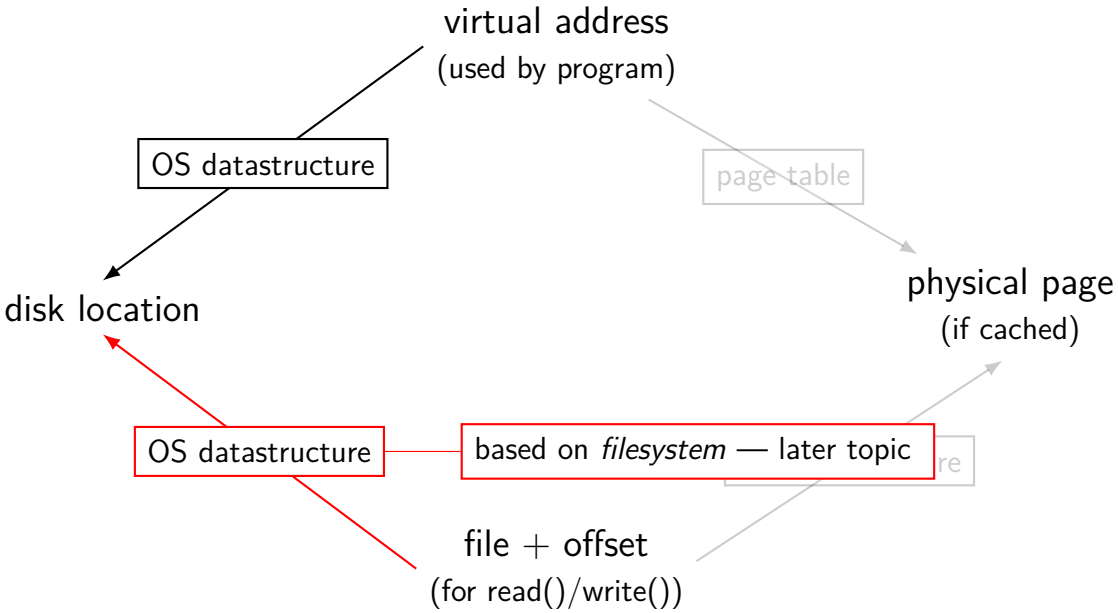
page cache data structures

- processor handles cache hits for virtual addresses via page tables
- OS handles cache hits for location in file
- need reverse lookup for page replacement
- need to choose not-going-to-be-used pages for replacement

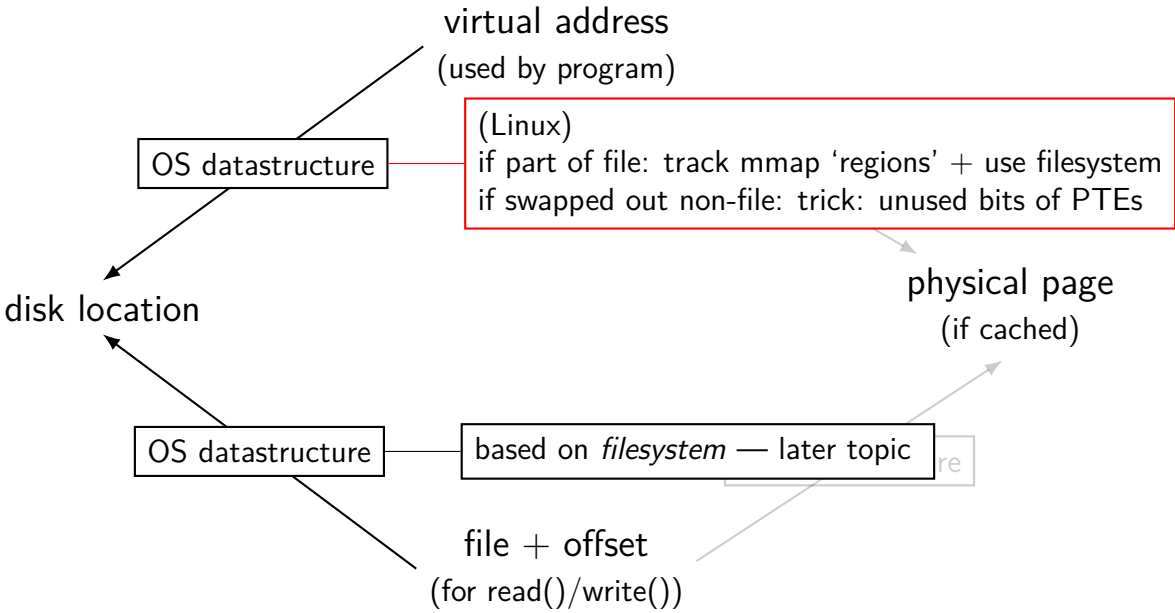
virtual address/file offset → location on disk



virtual address/file offset → location on disk



virtual address/file offset → location on 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

can save modified pages to disk in the background

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

referenced (virtual) pages:

time →

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

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

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					

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					

Belady's MIN exercise

referenced (virtual) pages:

time →

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

exercise: What does this access to D replace? (A, B, or C?)

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)

referenced (virtual) pages:

time →

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

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)

referenced (virtual) pages:

time →

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

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					

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

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) [intro]

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

least recently used (exercise)

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

least recently used (exercise) (2)

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

least recently used (exercise) (3)

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

least recently used (exercise) (4)

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

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 page

need to run code on every access
probably 100+x slowdown?

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

so, what's practical

probably won't implement LRU — too slow

what can we practically do?

practically tracking accesses

approximating LRU = "was this accessed recently"?

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

"was this accessed since we started looking a few seconds ago?"

practically tracking accesses

approximating LRU = "was this accessed recently"?

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

"was this accessed since we started looking a few seconds ago?"

one idea: track 'referenced' (or 'accessed') bit per page table entry

set to true when page table entry used for lookup

if OS clears periodically: indicates if accessed 'recently'

('recently' = since last time it was cleared)

implementing referenced bit

software: mark PTE invalid
if page fault happens, make valid and record 'referenced'

hardware: 'referenced' bit in page table entry
when hardware uses page table entry, sets bit

- x86: accessed flag in page table entries (PTE_A in xv6)
- not all hardware supports this

implementing referenced bit

software: mark PTE invalid
if page fault happens, make valid and record 'referenced'

hardware: 'referenced' bit in page table entry
when hardware uses page table entry, sets bit

- x86: accessed flag in page table entries (PTE_A in xv6)
- not all hardware supports this

same idea applies for detecting writes

- to know whether replaced page needs to be saved to disk
- called "dirty" bit

implementing referenced bit

software: mark PTE invalid
if page fault happens, make valid and record 'referenced'

hardware: 'referenced' bit in page table entry
when hardware uses page table entry, sets bit

- x86: accessed flag in page table entries (PTE_A in xv6)
- not all hardware supports this

same idea applies for detecting writes

- to know whether replaced page needs to be saved to disk
- called "dirty" bit

referenced bit and multiple mappings

suppose two processes map same physical page

- example: two processees are running ‘example.exe’
- physical pages holding that process’s code

was the page accessed recently?

yes, if referenced by either process

- need to check multiple page tables

referenced bit and multiple mappings

suppose two processes map same physical page

- example: two processees are running 'example.exe'
- physical pages holding that process's code

was the page accessed recently?

yes, if referenced by either process

- need to check multiple page tables

process A page table

...	...
VPN 0x40	not referenced, PPN 0x8332
VPN 0x41	referenced, PPN 0x8493
VPN 0x42	not referenced, PPN 0x8A31
...	...
VPN 0x50	referenced, PPN 0x8403
VPN 0x51	not referenced, PPN 0x8537
VPN 0x52	not referenced, PPN 0x8BCD
...	...

process B page table

...	...
VPN 0x40	not referenced, PPN 0x859A
VPN 0x41	referenced, PPN 0x8002
VPN 0x42	referenced, PPN 0x8004
...	...
VPN 0x50	referenced, PPN 0x8332
VPN 0x51	not referenced, PPN 0x8493
VPN 0x52	not referenced, PPN 0x8A31
...	...

referenced bit and multiple mappings

suppose two processes map same physical page

- example: two processees are running 'example.exe'
- physical pages holding that process's code

was the page accessed recently?

yes, if referenced by either process

- need to check multiple page tables

process A page table

...	...
VPN 0x40	not referenced, PPN 0x8332
VPN 0x41	referenced, PPN 0x8493
VPN 0x42	not referenced, PPN 0x8A31
...	...
VPN 0x50	referenced, PPN 0x8403
VPN 0x51	not referenced, PPN 0x8537
VPN 0x52	not referenced, PPN 0x8BCD
...	...

process B page table

...	...
VPN 0x40	not referenced, PPN 0x859A
VPN 0x41	referenced, PPN 0x8002
VPN 0x42	referenced, PPN 0x8004
...	...
VPN 0x50	referenced, PPN 0x8332
VPN 0x51	not referenced, PPN 0x8493
VPN 0x52	not referenced, PPN 0x8A31
...	...

referenced bit and multiple mappings

suppose two processes map same physical page

example: two processees are running 'example.exe'

physical pages holding that process's code

was the page accessed recently?

yes, if referenced by either process

need to check multiple page tables

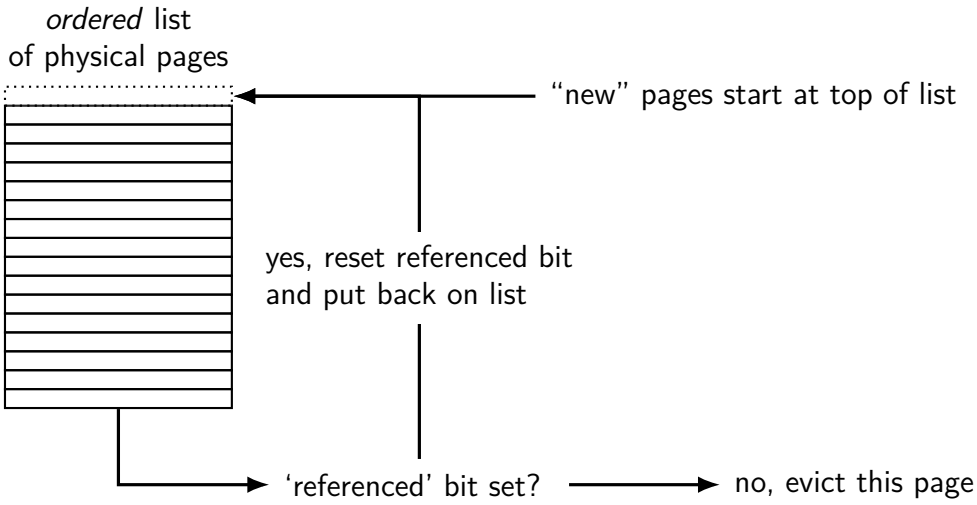
process A page table

...	...
VPN 0x40	not referenced, PPN 0x8332
VPN 0x41	referenced, PPN 0x8493
VPN 0x42	not referenced, PPN 0x8A31
...	...
VPN 0x50	referenced, PPN 0x8403
VPN 0x51	not referenced, PPN 0x8537
VPN 0x52	not referenced, PPN 0x8BCD
...	...

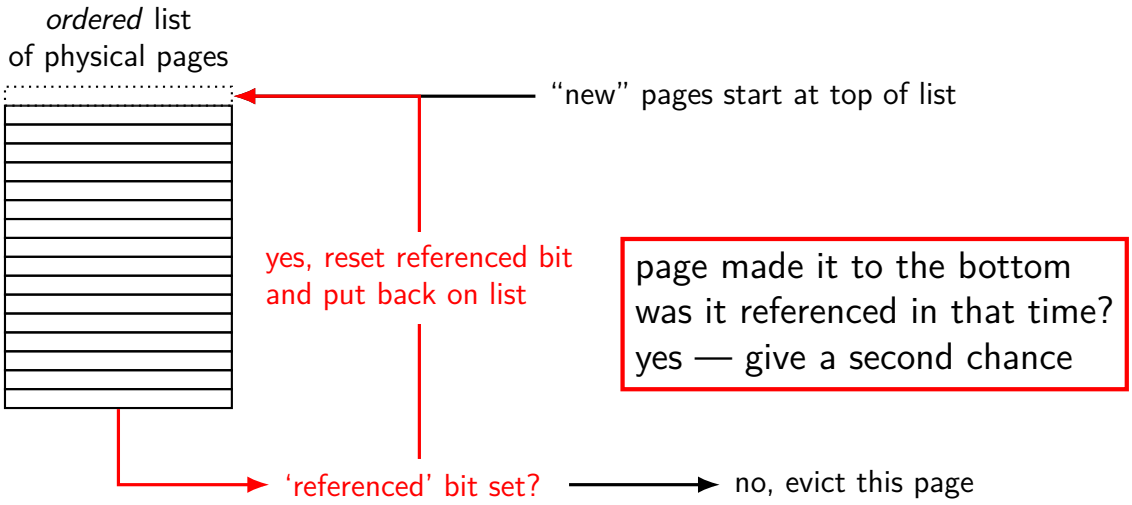
process B page table

...	...
VPN 0x40	not referenced, PPN 0x859A
VPN 0x41	referenced, PPN 0x8002
VPN 0x42	referenced, PPN 0x8004
...	...
VPN 0x50	referenced, PPN 0x8332
VPN 0x51	not referenced, PPN 0x8493
VPN 0x52	not referenced, PPN 0x8A31
...	...

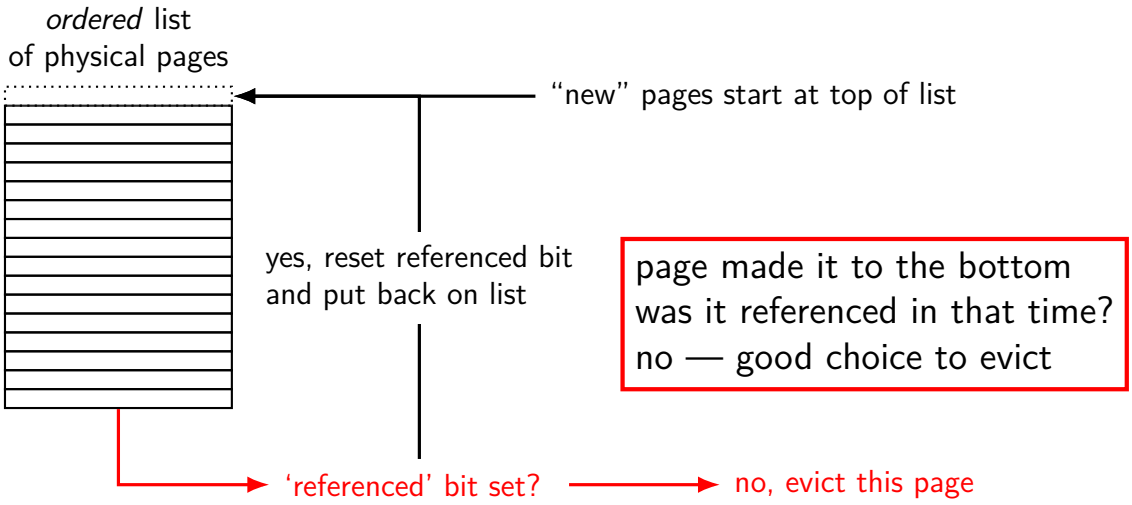
approximating LRU: second chance



approximating LRU: second chance



approximating LRU: second chance



second chance example (0)

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

second chance example (0)

		A		B

place A in physical page 1
accessed right after → becomes referenced

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

second chance example (0)

		A		B			

place B in physical page 2
accessed right after → becomes referenced

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

second chance example (0)

future slides:
going to skip writing
these intermediate steps
(just for space)

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

second chance example (1)

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

second chance example (1)

place A in page 1
not referenced on return from page fault handler
immediately referenced by program when page fault handler returns

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

second chance example (1)

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

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

second chance example (1)

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

second chance example (1)

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

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

second chance example (1)

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

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

second chance example (1)

B referenced — flips referenced bit

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

second chance example: exercise (1)

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

exercise: What does this access to A replace? (D, B, or C?)
what is at end of list after? (PP 1, 2, or 3?)

second chance example: exercise (2)

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

second chance example: exercise (2)

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

exercise: What does this access to C replace? (D, B, or A?)
what is at end of list after? (PP 1, 2, or 3?)

second chance example (2)

	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 cons

performs poorly with big memories...

may need to scan through lots of pages to find unaccessed

likely to count accesses from a long time ago

want some variation to tune its sensitivity

second chance cons

performs poorly with big memories...

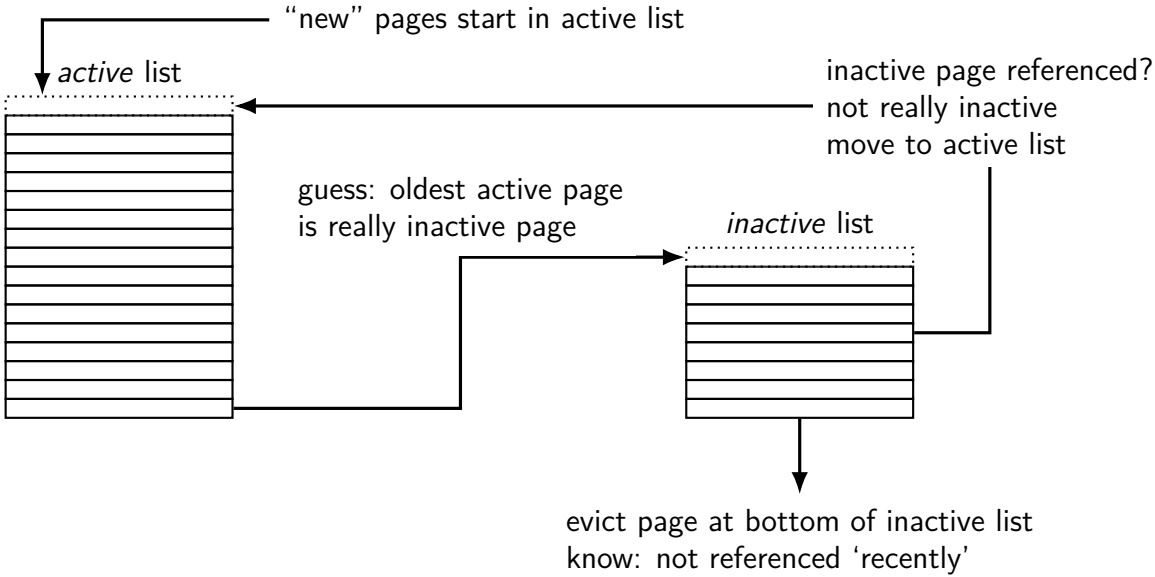
may need to scan through lots of pages to find unaccessed

likely to count accesses from a long time ago

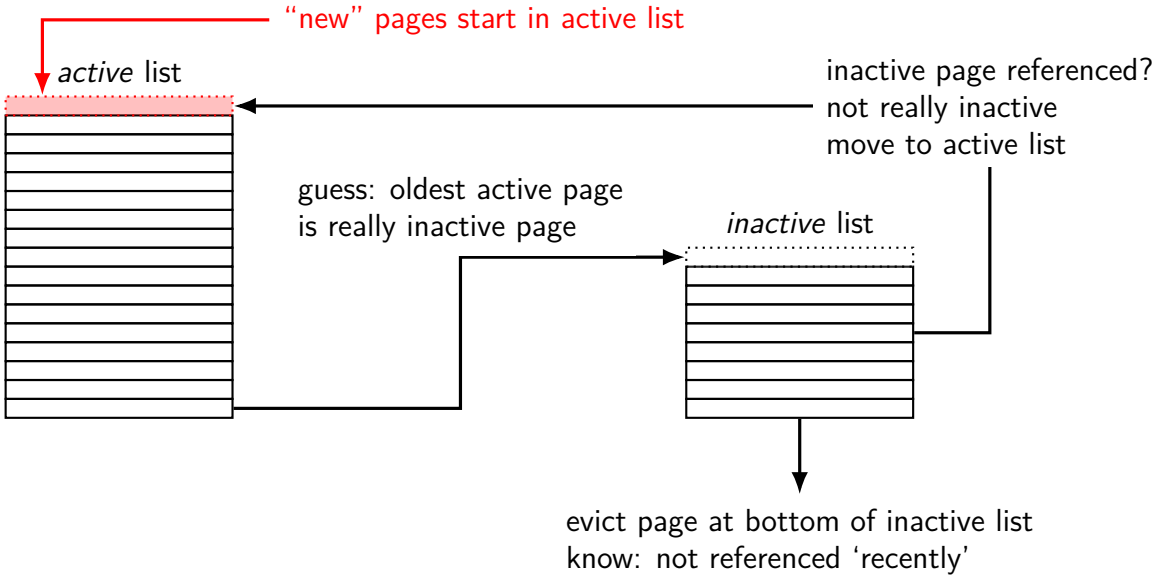
want some variation to tune its sensitivity

one idea: smaller list of pages to scan for accesses

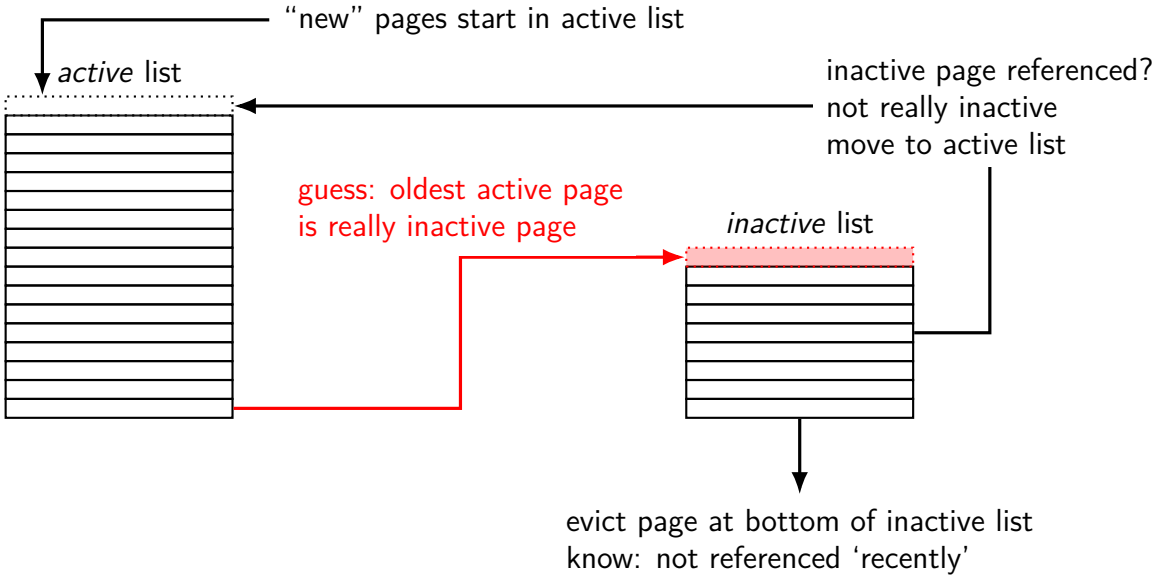
approximating LRU: SEQ



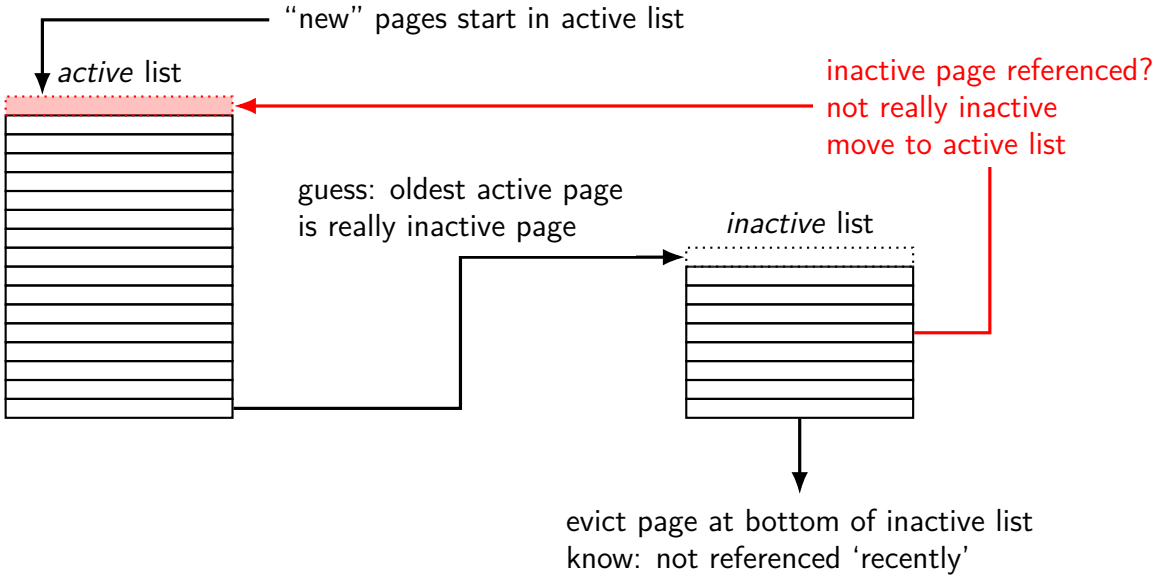
approximating LRU: SEQ



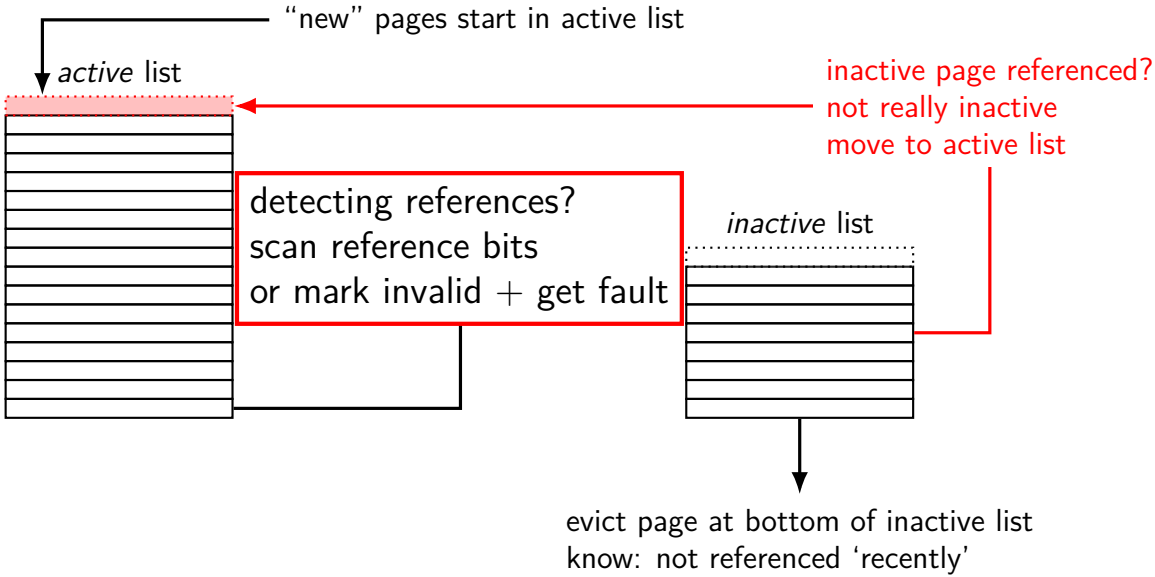
approximating LRU: SEQ



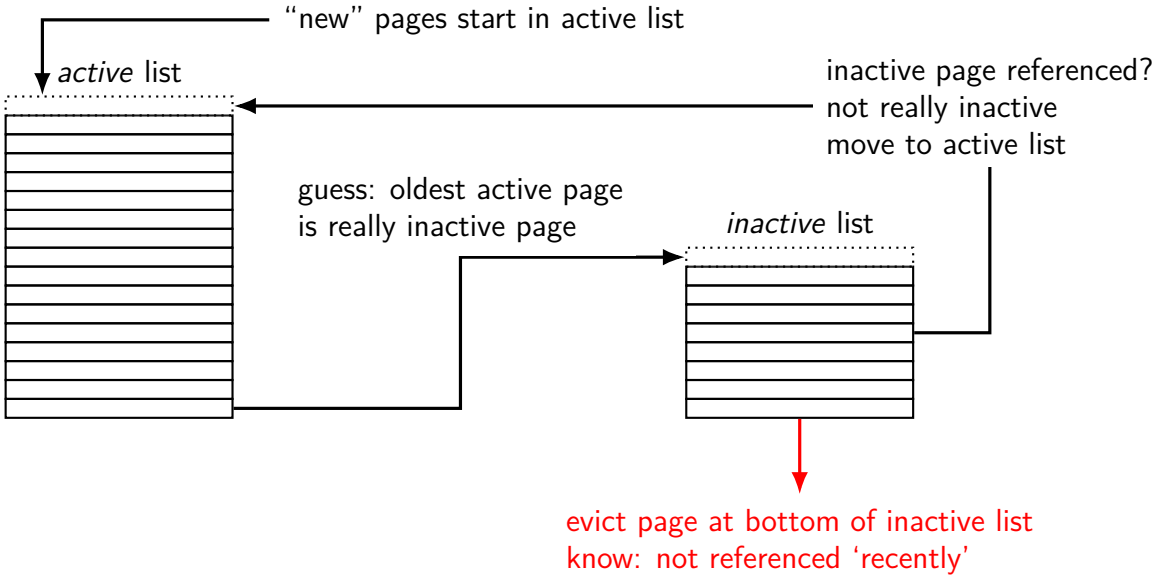
approximating LRU: SEQ



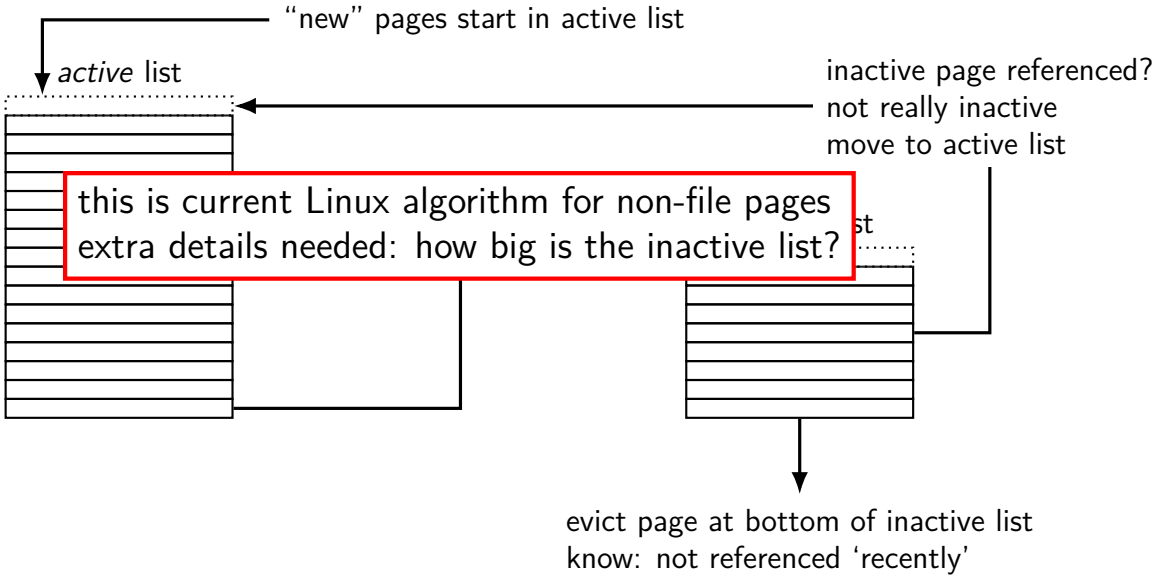
approximating LRU: SEQ



approximating LRU: SEQ



approximating LRU: SEQ



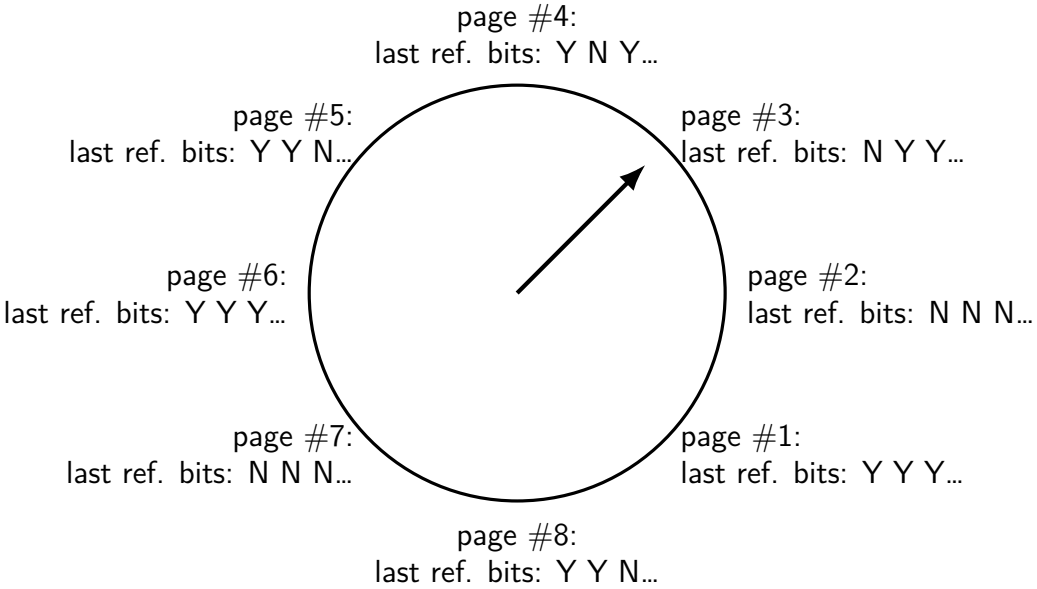
tracking usage: CLOCK (view 1)

ordered list
of physical pages

page #4: last referenced bits: Y Y Y...
page #5: last referenced bits: N N N...
page #6: last referenced bits: N Y Y...
page #7: last referenced bits: Y N Y...
page #8: last referenced bits: Y Y N...
page #1: last referenced bits: Y Y Y...
page #2: last referenced bits: N N N...
page #3: last referenced bits: Y Y N...

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)



problems with LRU

question: when does LRU perform poorly?

exercise: which of these is LRU bad for?

code in a text editor for handling out-of-disk-space errors

initial values of the shell's global variales

on a desktop, long movies that are too big to fit in memory and played from beginning to end

on web server, long movies that are too big to fit in memory and frequently downloaded by clients

files that are parsed when loaded and overwritten when saved

on web server, frequently requested HTML files

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

solution for LRU being bad?

one idea that Linux uses:

for *file data*, use different replacement policy

tries to avoid keeping around file data accessed only once

CLOCK-Pro: special casing for one-use pages

by default, Linux tries to handle scanning of files

- one read of file data — e.g. play a video, load file into memory

basic idea: delay considering pages active until second access

- second access = second scan of accessed bits/etc.

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

backup slides

x86-32 page table entries

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
Address of page directory[1]																				Ignored							PCD	PWT	Ignored			**CR3**
Bits 31:22 of address of 4MB page frame										Reserved (must be 0)					Bits 39:32 of address[2]				PAT	Ignored			G	1	D	A	PCD	PWT	U/S	R/W	1	**PDE: 4MB page**
Address of page table																				Ignored				0	Ign	A	PCD	PWT	U/S	R/W	1	**PDE: page table**
Ignored																															0	**PDE: not present**
Address of 4KB page frame																				Ignored			G	PAT	D	A	PCD	PWT	U/S	R/W	1	**PTE: 4KB page**
Ignored																															0	**PTE: not present**

Figure 4-4. Formats of CR3 and Paging-Structure Entries with 32-Bit Paging

x86-32 page table entries

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
Address of page directory[1]																				Ignored							PCD	PWT	Ignored			CR3
Bits 31:22 of address of 4MB page frame																			P							A	PCD	PWT	U/S	R/W	1	PDE: 4MB page
Address of page table																				Ignored				0	Ign	A	PCD	PWT	U/S	R/W	1	PDE: page table
Ignored																															0	PDE: not present
Address of 4KB page frame																				Ignored			G	PAT	D	A	PCD	PWT	U/S	R/W	1	PTE: 4KB page
Ignored																															0	PTE: not present

page table base register (CR3)

Figure 4-4. Formats of CR3 and Paging-Structure Entries with 32-Bit Paging

x86-32 page table entries

first-level page table entries

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
Address of pag																											PCD	PWT	Ignored			CR3
Bits 31:22 of address of 4MB page frame										Reserved (must be 0)					Bits 39:32 of address[2]				PAT	Ignored			G	1	D	A	PCD	PWT	U/S	R/W	1	PDE: 4MB page
Address of page table																				Ignored				0	Ign	A	PCD	PWT	U/S	R/W	1	PDE: page table
Ignored																															0	PDE: not present
Address of 4KB page frame																				Ignored			G	PAT	D	A	PCD	PWT	U/S	R/W	1	PTE: 4KB page
Ignored																															0	PTE: not present

Figure 4-4. Formats of CR3 and Paging-Structure Entries with 32-Bit Paging

x86-32 page table entries

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
Address of page directory[1]																				Ignored							PCD	PWT	Ignored			**CR3**
Bits 31:22 of address of 4MB page frame										Reserved (must be 0)					Bits 39:32 of address[2]				PAT	Ignored			G	1	D	A	PCD	PWT	U/S	R/W	1	**PDE: 4MB page**
Address of page table																				Ignored				0	Ign	A	PCD	PWT	U/S	R/W	1	**PDE: page table**
second-level page table entries																															0	**PDE: not present**
Address of 4KB page frame																				Ignored			G	PAT	D	A	PCD	PWT	U/S	R/W	1	**PTE: 4KB page**
Ignored																															0	**PTE: not present**

Figure 4-4. Formats of CR3 and Paging-Structure Entries with 32-Bit Paging

x86-32 page table entry v addresses

physical page number	zeros

phys. page byte addr

Address of 4KB page frame	Ignored	G	PAT	D	A	PCD	PWT	U/S	R/W	1	PTE: 4KB page
Ignored										0	PTE: not present

flags

trick: page table entry with lower bits zeroed = physical *byte* address of corresponding page

page # is address of page (2^{12} byte units)

makes constructing page table entries simpler:

```
physicalAddress | flagsBits
```

x86-32 pagetables: page table entries

xv6 header: mmu.h

```
// Page table/directory entry flags.
#define PTE_P           0x001   // Present
#define PTE_W           0x002   // Writeable
#define PTE_U           0x004   // User
#define PTE_PWT         0x008   // Write-Through
#define PTE_PCD         0x010   // Cache-Disable
#define PTE_A           0x020   // Accessed
#define PTE_D           0x040   // Dirty
#define PTE_PS          0x080   // Page Size
#define PTE_MBZ         0x180   // Bits must be zero

// Address in page table or page directory entry
#define PTE_ADDR(pte)   ((uint)(pte) & ~0xFFF)
#define PTE_FLAGS(pte)  ((uint)(pte) &  0xFFF)
```

xv6: extracting top-level page table entry

```
void output_top_level_pte_for(struct proc *p, void *address) {
  pde_t *top_level_page_table = p->pgdir;
  // PDX = Page Directory indeX
  // next level uses PTX(....)
  int index_into_pgdir = PDX(address);
  pde_t top_level_pte = top_level_page_table[index_into_pgdir];
  cprintf("top level PT for %x in PID %d\n", address, p->pid);
  if (top_level_pte & PTE_P) {
    cprintf("is present (valid)\n");
  }
  if (top_level_pte & PTE_W) {
    cprintf("is writable (may be overriden in next level)\n");
  }
  if (top_level_pte & PTE_U) {
    cprintf("is user-accessible (may be overriden in next level)\n")
  }
  cprintf("has base address %x\n", PTE_ADDR(top_level_pte));
}
```

xv6: extracting top-level page table entry

```
void output_top_level_pte_for(struct proc *p, void *address) {
  pde_t *top_level_page_table = p->pgdir;
  // PDX = Page Directory indeX
  // next level uses PTX(....)
  int index_into_pgdir = PDX(address);
  pde_t top_level_pte = top_level_page_table[index_into_pgdir];
  cprintf("top level PT for %x in PID %d\n", address, p->pid);
  if (top_level_pte & PTE_P) {
    cprintf("is present (valid)\n");
  }
  if (top_level_pte & PTE_W) {
    cprintf("is writable (may be overriden in next level)\n");
  }
  if (top_level_pte & PTE_U) {
    cprintf("is user-accessible (may be overriden in next level)\n")
  }
  cprintf("has base address %x\n", PTE_ADDR(top_level_pte));
}
```

xv6: extracting top-level page table entry

```
void output_top_level_pte_for(struct proc *p, void *address) {
  pde_t *top_level_page_table = p->pgdir;
  // PDX = Page Directory indeX
  // next level uses PTX(....)
  int index_into_pgdir = PDX(address);
  pde_t top_level_pte = top_level_page_table[index_into_pgdir];
  cprintf("top level PT for %x in PID %d\n", address, p->pid);
  if (top_level_pte & PTE_P) {
    cprintf("is present (valid)\n");
  }
  if (top_level_pte & PTE_W) {
    cprintf("is writable (may be overriden in next level)\n");
  }
  if (top_level_pte & PTE_U) {
    cprintf("is user-accessible (may be overriden in next level)\n")
  }
  cprintf("has base address %x\n", PTE_ADDR(top_level_pte));
}
```

xv6: extracting top-level page table entry

```
void output_top_level_pte_for(struct proc *p, void *address) {
  pde_t *top_level_page_table = p->pgdir;
  // PDX = Page Directory indeX
  // next level uses PTX(....)
  int index_into_pgdir = PDX(address);
  pde_t top_level_pte = top_level_page_table[index_into_pgdir];
  cprintf("top level PT for %x in PID %d\n", address, p->pid);
  if (top_level_pte & PTE_P) {
    cprintf("is present (valid)\n");
  }
  if (top_level_pte & PTE_W) {
    cprintf("is writable (may be overriden in next level)\n");
  }
  if (top_level_pte & PTE_U) {
    cprintf("is user-accessible (may be overriden in next level)\n")
  }
  cprintf("has base address %x\n", PTE_ADDR(top_level_pte));
}
```

xv6: extracting top-level page table entry

```
void output_top_level_pte_for(struct proc *p, void *address) {
  pde_t *top_level_page_table = p->pgdir;
  // PDX = Page Directory indeX
  // next level uses PTX(....)
  int index_into_pgdir = PDX(address);
  pde_t top_level_pte = top_level_page_table[index_into_pgdir];
  cprintf("top level PT for %x in PID %d\n", address, p->pid);
  if (top_level_pte & PTE_P) {
    cprintf("is present (valid)\n");
  }
  if (top_level_pte & PTE_W) {
    cprintf("is writable (may be overriden in next level)\n");
  }
  if (top_level_pte & PTE_U) {
    cprintf("is user-accessible (may be overriden in next level)\n")
  }
  cprintf("has base address %x\n", PTE_ADDR(top_level_pte));
}
```

xv6: manually setting page table entry

```
pde_t *some_page_table;  // if top-level table
pte_t *some_page_table;  // if next-level table
...
...
some_page_table[index] =
    PTE_P | PTE_W | PTE_U | base_physical_address;
/* P = present; W = writable; U = user-mode accessible */
```

skipping the guard page

```
void example() {
    int array[2000];
    array[0] = 1000;
    ...
}
```

```
example:
    subl    $8024, %esp // allocate 8024 bytes on stack
    movl    $1000, 12(%esp) // write near bottom of allocation
        // goes beyond guard page
        // since not all of array init'd
    ....
```

create new page table (kernel mappings)

```
pde_t*
setupkvm(void)
{
  pde_t *pgdir;
  struct kmap *k;

  if((pgdir = (pde_t*)kalloc()) == 0)
    return 0;
  memset(pgdir, 0, PGSIZE);
  if (P2V(PHYSTOP) > (void*)DEVSPACE)
    panic("PHYSTOP too high");
  for(k = kmap; k < &kmap[NELEM(kmap)]; k++)
    if(mappages(pgdir, k->virt, k->phys_end - k->phys_start,
                (uint)k->phys_start, k->perm) < 0) {
      freevm(pgdir);
      return 0;
    }
  return pgdir;
}
```

create new page table (kernel mappings)

allocate first-level page table ("page directory")

```
pde_t*
setupkvm(void)
{
  pde_t *pgdir;
  struct kmap *k;

  if((pgdir = (pde_t*)kalloc()) == 0)
    return 0;
  memset(pgdir, 0, PGSIZE);
  if (P2V(PHYSTOP) > (void*)DEVSPACE)
    panic("PHYSTOP too high");
  for(k = kmap; k < &kmap[NELEM(kmap)]; k++)
    if(mappages(pgdir, k->virt, k->phys_end - k->phys_start,
                (uint)k->phys_start, k->perm) < 0) {
      freevm(pgdir);
      return 0;
    }
  return pgdir;
}
```

create new page table (kernel mappings)

initialize to 0 — every page invalid

```
pde_t*
setupkvm(void)
{
  pde_t *pgdir;
  struct kmap *k;

  if((pgdir = (pde_t*)kalloc()) == 0)
    return 0;
  memset(pgdir, 0, PGSIZE);
  if (P2V(PHYSTOP) > (void*)DEVSPACE)
    panic("PHYSTOP too high");
  for(k = kmap; k < &kmap[NELEM(kmap)]; k++)
    if(mappages(pgdir, k->virt, k->phys_end - k->phys_start,
                (uint)k->phys_start, k->perm) < 0) {
      freevm(pgdir);
      return 0;
    }
  return pgdir;
}
```

create new page table (kernel mappings)

iterate through list of kernel-space mappings
for everything above address 0x8000 0000
(hard-coded table including flag bits, etc.
because some addresses need different flags
and not all physical addresses are usable)

```
pde_t*
setupkvm(void)
{
  pde_t *pgdir;
  struct kmap *k;

  if((pgdir = (pde_t*)kalloc()) == 0)
    return 0;
  memset(pgdir, 0, PGSIZE);
  if (P2V(PHYSTOP) > (void*)DEVSPACE)
    panic("PHYSTOP too high");
  for(k = kmap; k < &kmap[NELEM(kmap)]; k++)
    if(mappages(pgdir, k->virt, k->phys_end - k->phys_start,
                (uint)k->phys_start, k->perm) < 0) {
      freevm(pgdir);
      return 0;
    }
  return pgdir;
}
```

create new page table (kernel mappings)

on failure (no space for new second-level page tales)
free everything

```
pde_t*
setupkvm(void)
{
  pde_t *pgdir;
  struct kmap *k;

  if((pgdir = (pde_t*)kalloc()) == 0)
    return 0;
  memset(pgdir, 0, PGSIZE);
  if (P2V(PHYSTOP) > (void*)DEVSPACE)
    panic("PHYSTOP too high");
  for(k = kmap; k < &kmap[NELEM(kmap)]; k++)
    if(mappages(pgdir, k->virt, k->phys_end - k->phys_start,
                (uint)k->phys_start, k->perm) < 0) {
      freevm(pgdir);
      return 0;
    }
  return pgdir;
}
```

reading executables (headers)

xv6 executables contain list of sections to load, represented by:

```
struct proghdr {
  uint type;        /* <-- debugging-only or not? */
  uint off;         /* <-- location in file */
  uint vaddr;       /* <-- location in memory */
  uint paddr;       /* <-- confusing ignored field */
  uint filesz;      /* <-- amount to load */
  uint memsz;       /* <-- amount to allocate */
  uint flags;       /* <-- readable/writeable (ignored) */
  uint align;
};
```

reading executables (headers)

xv6 executables contain list of sections to load, represented by:

```
struct proghdr {
  uint type;        /* <-- debugging-only or not? */
  uint off;         /* <-- location in file */
  uint vaddr;       /* <-- location in memory */
  uint paddr;       /* <-- confusing ignored field */
  uint filesz;      /* <-- amount to load */
  uint memsz;       /* <-- amount to allocate */
  uint flags;       /* <-- readable/writeable (ignored) */
  uint align;
};

...
  if((sz = allocuvm(pgdir, sz, ph.vaddr + ph.memsz)) == 0)
    goto bad;
  ...
  if(loaduvm(pgdir, (char*)ph.vaddr, ip, ph.off, ph.filesz) < 0)
    goto bad;
```

reading executables (headers)

xv6 executables contain list of sections to load, represented by:

sz — top of heap of new program
name of the field in struct proc

```
struct proghdr {                                    */
  uint type;
  uint off;
  uint vaddr;        /* <-- location in memory */
  uint paddr;        /* <-- confusing ignored field */
  uint filesz;       /* <-- amount to load */
  uint memsz;        /* <-- amount to allocate */
  uint flags;        /* <-- readable/writeable (ignored) */
  uint align;
};

...
  if((sz = allocuvm(pgdir, sz, ph.vaddr + ph.memsz)) == 0)
    goto bad;
  ...
  if(loaduvm(pgdir, (char*)ph.vaddr, ip, ph.off, ph.filesz) < 0)
    goto bad;
```

loading user pages from executable

```
loaduvm(pde_t *pgdir, char *addr, struct inode *ip, uint offset, uin
{
  ...
  for(i = 0; i < sz; i += PGSIZE){
    if((pte = walkpgdir(pgdir, addr+i, 0)) == 0)
      panic("loaduvm: address should exist");
    pa = PTE_ADDR(*pte);
    if(sz - i < PGSIZE)
      n = sz - i;
    else
      n = PGSIZE;
    if(readi(ip, P2V(pa), offset+i, n) != n)
      return -1;
  }
  return 0;
}
```

loading user pages from executable

```
loaduvm(pde_t *pgdir, char *addr                                    , uin
{
  ...
  for(i = 0; i < sz; i += PGSIZE){
    if((pte = walkpgdir(pgdir, addr+i, 0)) == 0)
      panic("loaduvm: address should exist");
    pa = PTE_ADDR(*pte);
    if(sz - i < PGSIZE)
      n = sz - i;
    else
      n = PGSIZE;
    if(readi(ip, P2V(pa), offset+i, n) != n)
      return -1;
  }
  return 0;
}
```

get page table entry being loaded
already allocated earlier
look up address to load into

loading user pages from executable

get physical address from page table entry
convert back to (kernel) virtual address
for read from disk

```
loaduvm(pde_t *pgdir, ch                                    , uin
{
  ...
  for(i = 0; i < sz; i += PGSIZE){
    if((pte = walkpgdir(pgdir, addr+i, 0)) == 0)
      panic("loaduvm: address should exist");
    pa = PTE_ADDR(*pte);
    if(sz - i < PGSIZE)
      n = sz - i;
    else
      n = PGSIZE;
    if(readi(ip, P2V(pa), offset+i, n) != n)
      return -1;
  }
  return 0;
}
```

loading user pages from executable

exercise: why don't we just use `addr` directly? (instead of turning it into a physical address, then into a virtual address again)

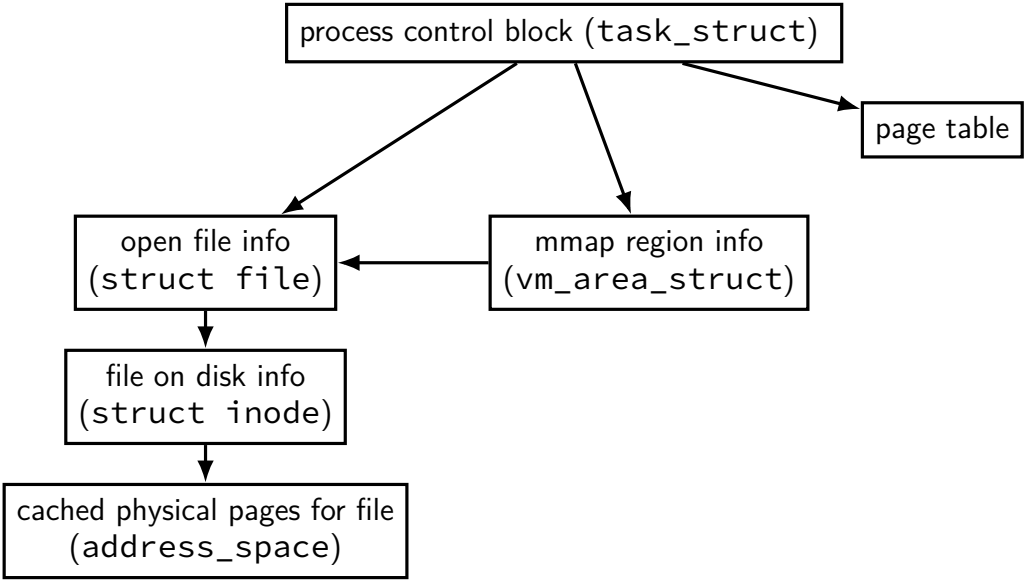
```
loaduvm(pde_t *pgdir                                    , uin
{
  ...
  for(i = 0; i < sz;
    if((pte = walkpgdir(pgdir, addr+i, 0)) == 0)
      panic("loaduvm: address should exist");
    pa = PTE_ADDR(*pte);
    if(sz - i < PGSIZE)
      n = sz - i;
    else
      n = PGSIZE;
    if(readi(ip, P2V(pa), offset+i, n) != n)
      return -1;
  }
  return 0;
}
```

loading user pages from executable

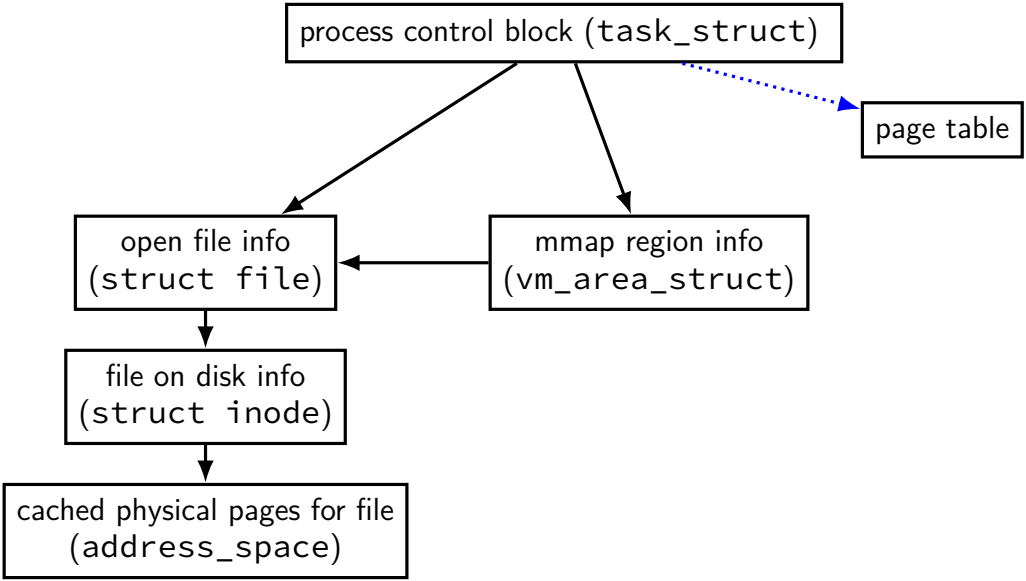
copy from file (represented by `struct inode`) into memory
P2V(pa) — mapping of physical addresss in kernel memory

```
loaduv                                                    , uin
{
  ...
  for(i = 0; i < sz; i += PGSIZE){
    if((pte = walkpgdir(pgdir, addr+i, 0)) == 0)
      panic("loaduvm: address should exist");
    pa = PTE_ADDR(*pte);
    if(sz - i < PGSIZE)
      n = sz - i;
    else
      n = PGSIZE;
    if(readi(ip, P2V(pa), offset+i, n) != n)
      return -1;
  }
  return 0;
}
```

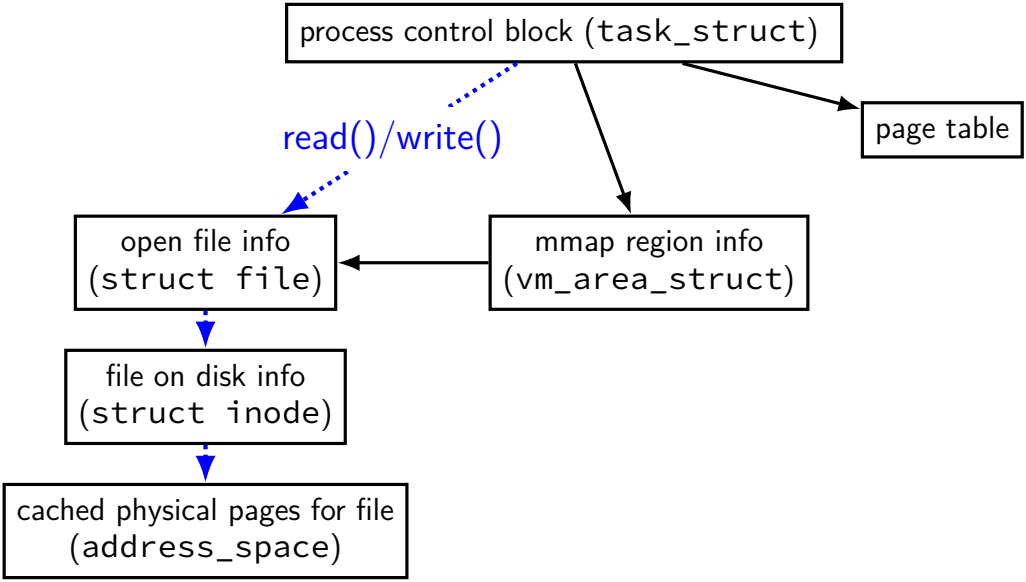
Linux: forward mapping



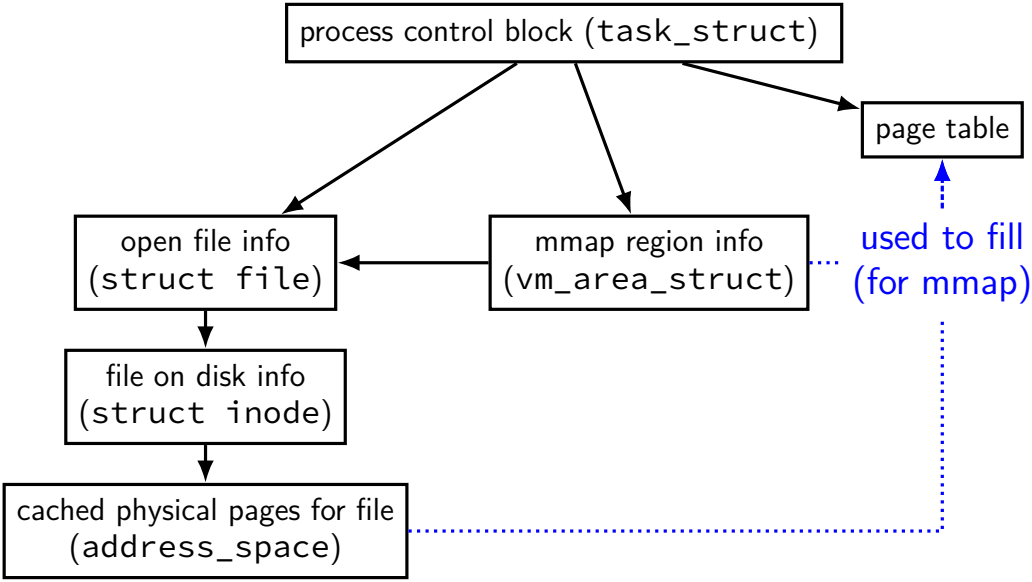
Linux: forward mapping



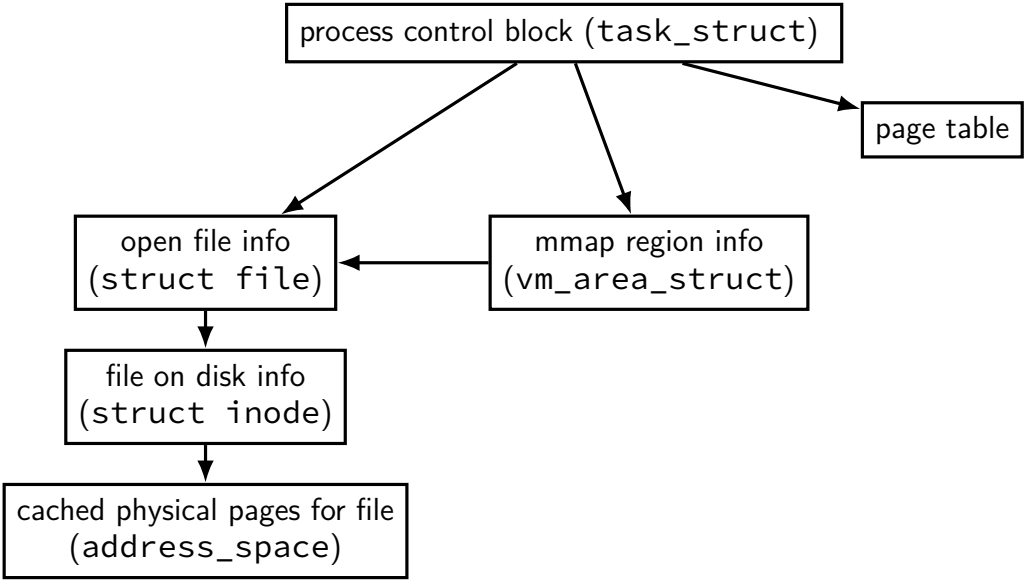
Linux: forward mapping



Linux: forward mapping



Linux: forward mapping



sketch: implementing mmap

access mapped file for first time, read from disk

- (like swapping when memory was swapped out)

write "mapped" memory, write to disk eventually

- need to detect whether writes happened
- usually hardware support: dirty bit

extra detail: other processes should see changes

- all accesses to file use same physical memory
- how? OS tracks copies of files in memory

xv6: setting process page tables (`exec()`)

exec step 1: create new page table with kernel mappings

- done in `setupkvm()`, which calls `mappages()`

exec step 2a: allocate memory for executable pages

- `allocuvm()` in loop
- new physical pages chosen by `kalloc()`

exec step 2b: load from executable file

- copying from executable file implemented by `loaduvm()`

exec step 3: allocate pages for heap, stack (`allocuvm()` calls)

xv6: setting process page tables (`exec()`)

exec step 1: create new page table with kernel mappings

- done in `setupkvm()`, which calls `mappages()`

exec step 2a: allocate memory for executable pages

- `allocuvm()` in loop
- new physical pages chosen by `kalloc()`

exec step 2b: load from executable file

- copying from executable file implemented by `loaduvm()`

exec step 3: allocate pages for heap, stack (`allocuvm()` calls)

minor and major faults

minor page fault

- page is already in memory ("page cache")
- just fill in page table entry

major page fault

- page not already in memory ("page cache")
- need to allocate space
- possibly need to read data from disk/etc.

Linux: reporting minor/major faults

```
$ /usr/bin/time --verbose some-command
        Command being timed: "some-command"
        User time (seconds): 18.15
        System time (seconds): 0.35
        Percent of CPU this job got: 94%
        Elapsed (wall clock) time (h:mm:ss or m:ss): 0:19.57
...
        Maximum resident set size (kbytes): 749820
        Average resident set size (kbytes): 0
        Major (requiring I/O) page faults: 0
        Minor (reclaiming a frame) page faults: 230166
        Voluntary context switches: 1423
        Involuntary context switches: 53
        Swaps: 0
...
        Exit status: 0
```

swapping

historical major use of virtual memory is supporting "swapping"

using disk (or SSD, ...) as the next level of the memory hierarchy

process is allocated space on disk/SSD

memory is a cache for disk/SSD

- only need keep 'currently active' pages in physical memory

swapping

historical major use of virtual memory is supporting "swapping"

using disk (or SSD, ...) as the next level of the memory hierarchy

process is allocated space on disk/SSD

memory is a cache for disk/SSD

- only need keep 'currently active' pages in physical memory

swapping $\approx$ mmap with "default" files to use

HDD/SDDs are slow

HDD reads and writes: milliseconds to tens of milliseconds

- minimum size: 512 bytes
- writing tens of kilobytes basically as fast as writing 512 bytes

SSD writes and writes: hundreds of microseconds

- designed for writes/reads of kilobytes (not much smaller)

HDD/SDDs are slow

HDD reads and writes: milliseconds to tens of milliseconds

- minimum size: 512 bytes
- writing tens of kilobytes basically as fast as writing 512 bytes

SSD writes and writes: hundreds of microseconds

- designed for writes/reads of kilobytes (not much smaller)

HDD/SDDs are slow

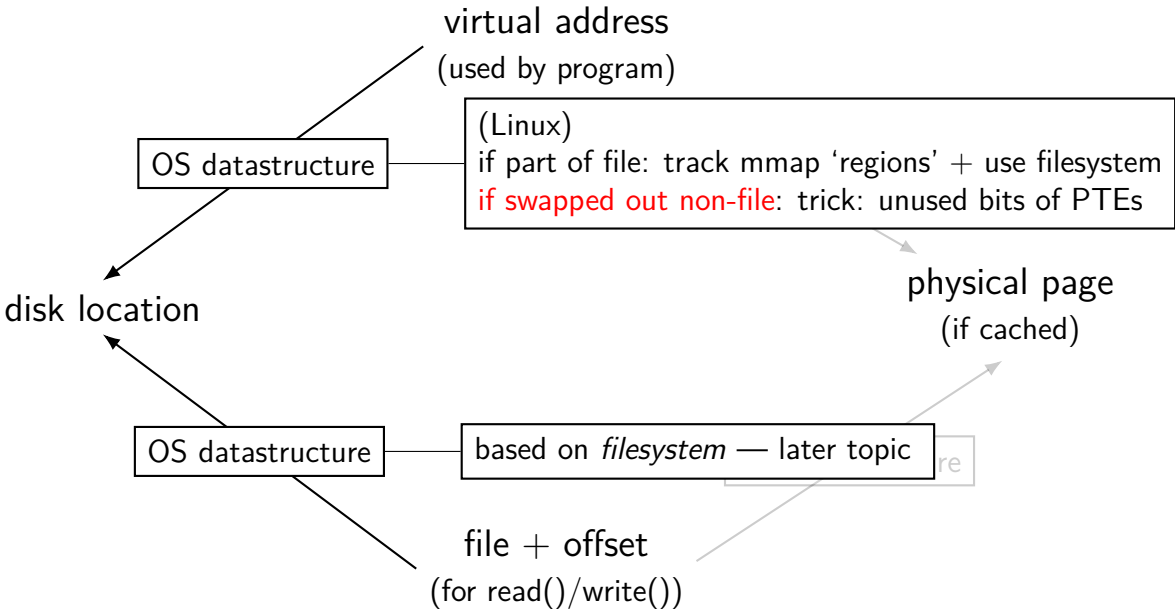
HDD reads and writes: milliseconds to tens of milliseconds

- minimum size: 512 bytes
- writing tens of kilobytes basically as fast as writing 512 bytes

SSD writes and writes: hundreds of microseconds

- designed for writes/reads of kilobytes (not much smaller)

virtual address/file offset → location on disk



Linux: tracking swapped out pages

need to lookup location on disk

potentially one location for every virtual page

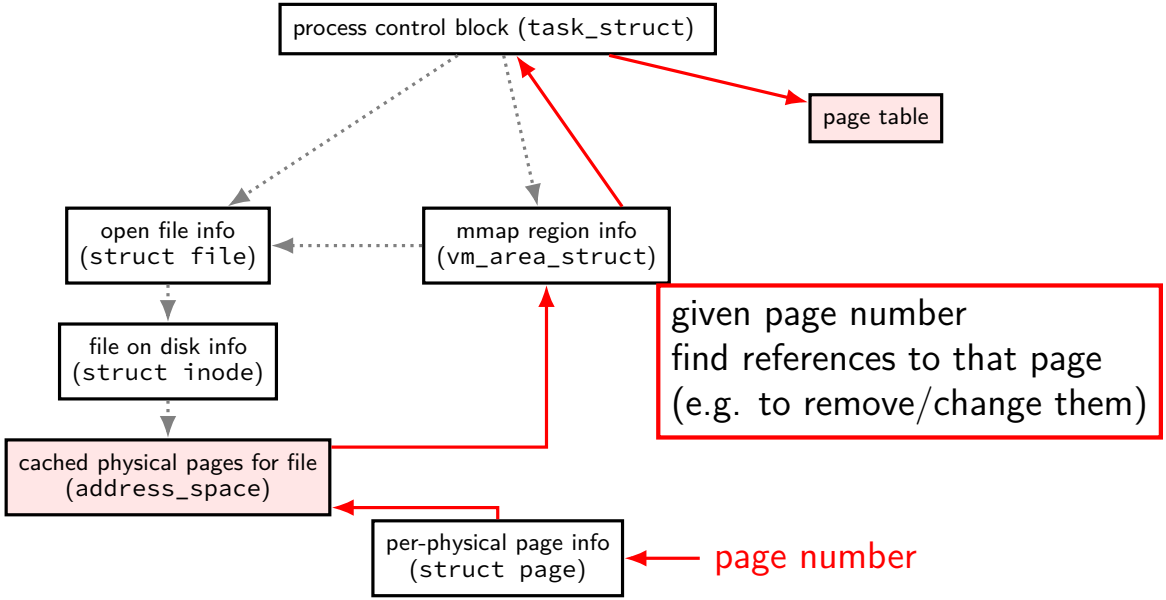
trick: store location in "ignored" part of page table entry

instead of physical page #, permission bits, etc., store offset on disk

Address of 4KB page frame	Ignored	G	PAT	D	A	PCD	PWT	U/S	R/W	1	PTE: 4KB page
Ignored										0	PTE: not present

Figure 4-4. Formats of CR3 and Paging-Structure Entries with 32-Bit Paging

Linux: reverse mapping (file pages)



tracking physical pages: finding free pages

Linux has list of "least recently used" pages:

```
struct page {
    ...
    struct list_head lru;   /* list_head ~ next/prev pointer */
    ...
};
```

how we're going to find a page to allocate

(and evict from something else)

later — what this list actually looks like (how many lists, ...)

predicting the future?

can't really...

look for common patterns

working set intuition

say we're executing a loop

what memory does this require?

code for the loop

code for functions called in the loop

- and functions they call

data structures used by the loop and functions called in it, etc.

only uses a subset of the program's memory

the working set model

one common pattern: working sets

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

...called its *working set*

rest of memory is inactive

...until program switches to different working set

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

replacement policy: identify working sets $\approx$ recently used data

replace anything that's not in in it

cache size versus miss rate

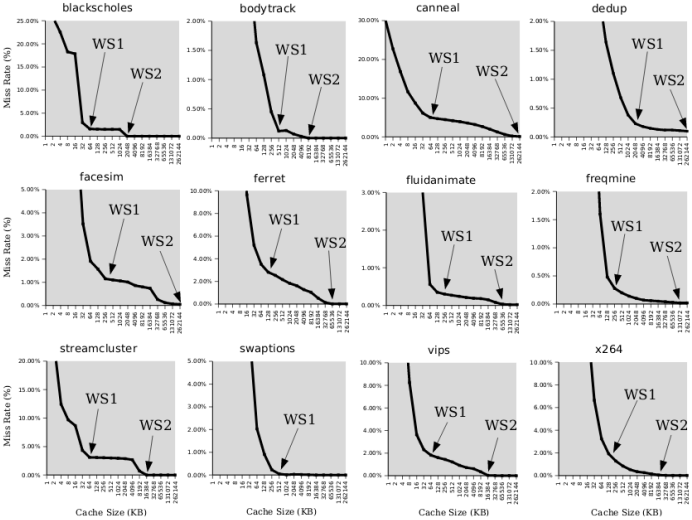


Figure 3: Miss rates versus cache size. Data assumes a shared 4-way associative cache with 64 byte lines. WS1 and WS2 refer to important working sets which we analyze in more detail in Table 2. Cache requirements of PARSEC benchmark programs can reach hundreds of megabytes.

estimating working sets

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

- except when program switching working sets

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

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

estimating working sets

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

- except when program switching working sets

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

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

recording accesses

goal: "check is this physical page still being used?"

software support: temporarily mark page table invalid

- use resulting page fault to detect "yes"

hardware support: accessed bits in page tables

- hardware sets to 1 when accessed

temporarily invalid PTE (software support)

program 1

```
mov 0x123456, %ecx
mov 0x123789, %ecx
...
...
mov 0x123300, %ecx
```

the kernel

...
(OS exception's handler)
...

page table for program 1

VPN	present?	writable?	...	PPN
0x00000	0	---	...	---
0x00001	0	---	...	---
...	...	...	...	...
0x00123	0	0	...	0x4442
...	...	...	...	...

OS page info

PPN	last known access?	...
...	...	...
0x04442	(never)	...
...	...	...

temporarily invalid PTE (software support)

program 1

```
mov 0x123456, %ecx
mov 0x123789, %ecx
...
...
mov 0x123300, %ecx
```

the kernel

```
...
(OS exception's handler)
...
```

oops! page fault

processor does lookup

page table for program 1

VPN	present?	writable?	...	PPN
0x00000	0	---	...	---
0x00001	0	---	...	---
...	...	...	...	...
0x00123	0	0	...	0x4442
...	...	...	...	...

OS page info

PPN	last known access?	...
...	...	...
0x04442	(never)	...
...	...	...

temporarily invalid PTE (software support)

program 1

```
mov 0x123456, %ecx
mov 0x123789, %ecx
...
...
mov 0x123300, %ecx
```

the kernel

...
(OS exception's handler)
...

update page info +
mark present

page table for program 1

VPN	present?	writable?	...	PPN
0x00000	0	---	...	---
0x00001	0	---	...	---
...	...	...	...	...
0x00123	1	0	...	0x4442
...	...	...	...	...

OS page info

PPN	last known access?	...
...	...	...
0x04442	at time X	
...	...	...

temporarily invalid PTE (software support)

program 1

```
mov 0x123456, %ecx
mov 0x123789, %ecx
...
...
mov 0x123300, %ecx
```

the kernel

...
(OS exception's handler)
...

processor does lookup
no page fault, not recorded in OS info

page table for program 1

VPN	present?	writable?	...	PPN
0x00000	0	---	...	---
0x00001	0	---	...	---
...	...	...	...	...
0x00123	1	0	...	0x4442
...	...	...	...	...

OS page info

PPN	last known access?	...
...	...	...
0x04442	at time X	...
...	...	...

temporarily invalid PTE (software support)

program 1

```
mov 0x123456, %ecx
mov 0x123789, %ecx
...
...
mov 0x123300, %ecx
```

the kernel

...
(OS exception's handler)
...

processor does lookup
no page fault, not recorded in OS info

page table for program 1

VPN	present?	writable?	...	PPN
0x00000	0	---	...	---
0x00001	0	---	...	---
...	...	...	...	...
0x00123	1	0	...	0x4442
...	...	...	...	...

OS page info

PPN	last known access?	...
...	...	...
0x04442	at time X	...
...	...	...

temporarily invalid PTE (software support)

program 1

```
mov 0x123456, %ecx
mov 0x123789, %ecx
...
...
mov 0x123300, %ecx
```

the kernel

...
(OS exception's handler)
...

OS clears present bit
to check for next access

page table for program 1

VPN	present?	writable?	...	PPN
0x00000	0	---	...	---
0x00001	0	---	...	---
...	...	...	...	...
0x00123	1	0	...	0x4442
...	...	...	...	...

OS page info

PPN	last known access?	...
...	...	...
0x04442	at time X	...
...	...	...

temporarily invalid PTE (software support)

program 1

```
mov 0x123456, %ecx
mov 0x123789, %ecx
...
...
mov 0x123300, %ecx
```

the kernel

...
(OS exception's handler)
...

OS clears present bit
to check for next access

page table for program 1

VPN	present?	writable?	...	PPN
0x00000	0	---	...	---
0x00001	0	---	...	---
...	...	...	...	...
0x00123	0	0	...	0x4442
...	...	...	...	...

OS page info

PPN	last known access?	...
...	...	...
0x04442	at time X	...
...	...	...

temporarily invalid PTE (software support)

program 1

```
mov 0x123456, %ecx
mov 0x123789, %ecx
...
...
mov 0x123300, %ecx
```

the kernel

```
...
(OS exception's handler)
...
```

oops! page fault

processor does lookup

page table for program 1

VPN	present?	writable?	...	PPN
0x00000	0	---	...	---
0x00001	0	---	...	---
...	...	...	...	...
0x00123	0	0	...	0x4442
...	...	...	...	...

OS page info

PPN	last known access?	...
...	...	...
0x04442	at time X	...
...	...	...

temporarily invalid PTE (software support)

program 1

```
mov 0x123456, %ecx
mov 0x123789, %ecx
...
...
mov 0x123300, %ecx
```

the kernel

...
(OS exception's handler)
...

update page info +
mark present

page table for program 1

VPN	present?	writable?	...	PPN
0x00000	0	---	...	---
0x00001	0	---	...	---
...	...	...	...	...
0x00123	1	0	...	0x4442
...	...	...	...	...

OS page info

PPN	last known access?	...
...	...	...
0x04442	at time Y	
...	...	...

accessed bit usage (hardware support)

program 1

```
mov 0x123456, %ecx
mov 0x123789, %ecx
...
...
mov 0x123300, %ecx
```

the kernel

```
...
(OS exception's handler)
...
```

page table for program 1

VPN	present?	accessed?	writable?	...	PPN
0x00000	0	---	---	...	---
0x00001	0	---	---	...	---
...	...	...	...	...	...
0x00123	1	0	0	...	0x4442
...	...	...	...	...	...

accessed bit usage (hardware support)

program 1

```
mov 0x123456, %ecx
mov 0x123789, %ecx
...
...
mov 0x123300, %ecx
```

the kernel

```
...
(OS exception's handler)
...
```

processor does lookup

sets accessed bit to 1

page table for program 1

VPN	present?	accessed?	writable?	...	PPN
0x00000	0	---	---	...	---
0x00001	0	---	---	...	---
...	...	...	...	...	...
0x00123	1	0	0	...	0x4442
...	...	...	...	...	...

accessed bit usage (hardware support)

program 1

```
mov 0x123456, %ecx
mov 0x123789, %ecx
...
...
mov 0x123300, %ecx
```

the kernel

```
...
(OS exception's handler)
...
```

processor does lookup

sets accessed bit to 1

page table for program 1

VPN	present?	accessed?	writable?	...	PPN
0x00000	0	---	---	...	---
0x00001	0	---	---	...	---
...	...	...	...	...	...
0x00123	1	1	0	...	0x4442
...	...	...	...	...	...

accessed bit usage (hardware support)

program 1

```
mov 0x123456, %ecx
mov 0x123789, %ecx
...
...
mov 0x123300, %ecx
```

the kernel

```
...
(OS exception's handler)
...
```

processor does lookup

keeps access bit set to 1

page table for program 1

VPN	present?	accessed?	writable?	...	PPN
0x00000	0	---	---	...	---
0x00001	0	---	---	...	---
...	...	...	...	...	...
0x00123	1	1	0	...	0x4442
...	...	...	...	...	...

accessed bit usage (hardware support)

program 1

```
mov 0x123456, %ecx
mov 0x123789, %ecx
...
...
mov 0x123300, %ecx
```

the kernel

```
...
(OS exception's handler)
...
```

processor does lookup

keeps access bit set to 1

page table for program 1

VPN	present?	accessed?	writable?	...	PPN
0x00000	0	---	---	...	---
0x00001	0	---	---	...	---
...	...	...	...	...	...
0x00123	1	1	0	...	0x4442
...	...	...	...	...	...

accessed bit usage (hardware support)

program 1

```
mov 0x123456, %ecx
mov 0x123789, %ecx
...
...
mov 0x123300, %ecx
```

the kernel

...
(OS exception's handler)
...

OS reads + records + clears access bit

page table for program 1

VPN	present?	accessed?	writable?	...	PPN
0x00000	0	---	---	...	---
0x00001	0	---	---	...	---
...	...	...	...	...	...
0x00123	1	1	0	...	0x4442
...	...	...	...	...	...

accessed bit usage (hardware support)

program 1

```
mov 0x123456, %ecx
mov 0x123789, %ecx
...
...
mov 0x123300, %ecx
```

the kernel

...
(OS exception's handler)
...

OS reads + records + clears access bit

page table for program 1

VPN	present?	accessed?	writable?	...	PPN
0x00000	0	---	---	...	---
0x00001	0	---	---	...	---
...	...	...	...	...	...
0x00123	1	0	0	...	0x4442
...	...	...	...	...	...

accessed bit usage (hardware support)

program 1

```
mov 0x123456, %ecx
mov 0x123789, %ecx
...
...
mov 0x123300, %ecx
```

the kernel

```
...
(OS exception's handler)
...
```

processor does lookup

sets accessed bit to 1 (again)

page table for program 1

VPN	present?	accessed?	writable?	...	PPN
0x00000	0	---	---	...	---
0x00001	0	---	---	...	---
...	...	...	...	...	...
0x00123	1	0	0	...	0x4442
...	...	...	...	...	...

accessed bit usage (hardware support)

program 1

```
mov 0x123456, %ecx
mov 0x123789, %ecx
...
...
mov 0x123300, %ecx
```

the kernel

```
...
(OS exception's handler)
...
```

processor does lookup

sets accessed bit to 1 (again)

page table for program 1

VPN	present?	accessed?	writable?	...	PPN
0x00000	0	---	---	...	---
0x00001	0	---	---	...	---
...	...	...	...	...	...
0x00123	1	1	0	...	0x4442
...	...	...	...	...	...

accessed bits: multiple processes

page table for program 1

VPN	present?	accessed?	writable?	···	PPN
0x00000	0	---	---	···	---
0x00001	0	---	---	···	---
···	···	···	···	···	···
0x00123	1	0	0	···	0x4442
···	···	···	···	···	···

page table for program 2

VPN	present?	accessed?	writable?	···	PPN
0x00000	0	---	---	···	---
0x00001	0	---	---	···	---
···	···	···	···	···	···
0x00483	1	1	0	···	0x4442
···	···	···	···	···	···

OS needs to clear+check **all** accessed bits for the physical page

dirty bits

"was this part of the mmap'd file changed?"

"is the old swapped copy still up to date?"

software support: temporarily mark read-only

hardware support: ***dirty bit*** set by hardware

- same idea as accessed bit, but only changed on writes

x86-32 accessed and dirty bit

Address of 4KB page frame	Ignored	G	PAT	D	A	PCD	PWT	U/S	R/W	1	PTE: 4KB page
Ignored										0	PTE: not present

Figure 4-4. Formats of CR3 and Paging-Structure Entries with 32-Bit Paging

A: acccessed — processor sets to 1 when PTE used

used = for read or write or execute

likely implementation: part of loading PTE into TLB

D: dirty — processor sets to 1 when PTE is used for write

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 never run out of memory…all of it?

- no changed data written back

solution: track or scan for dirty pages and writeback

example goals:

- lose no more than 90 seconds of data
- force writeback at file close
- …

non-lazy eviction

so far — allocating memory involves evicting pages

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

non-lazy eviction

so far — allocating memory involves evicting pages

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

alternative: evict earlier "in the background"

- "free": probably have some idle processor time anyways

allocation = remove already evicted page from linked list

- (instead of changing page tables, file cache info, etc.)

xv6 page table-related functions

`kalloc`/`kfree` — allocate physical page, return kernel address

`walkpgdir` — get pointer to second-level page table entry

- ...to check it/make it valid/invalid/point somewhere/etc.

`mappages` — set range of page table entries

- implementation: loop using `walkpgdir`

`allockvm` — create new set of page tables, set kernel (high) part

- entries for `0x8000 0000` and up set
- allocate new first-level table plus several second-level tables

`allocuvm` — allocate new user memory

- setup user-accessible memory
- allocate new second-level tables as needed

`deallocuvm` — deallocate user memory