last time (1)

```
kill() signal sending timing
```

```
user ID/group IDs
```

used by kernel to determine what to do tracked for every process libraries/utilities map to names

permission checks in system call handlers

```
chmod permissions
```

user ID (owner) / one group ID / others — read/write/exec

access control list

list of users/groups — read/write/exec for each

last time (2)

user ID 0 ('root', 'superuser') — passes all permission checks

login program: runs as user ID 0 can access password database because user ID 0

set-user-ID programs

special bit says "run program with owner's user ID" system administrator can setup program to only do 'safe' things example: sudo: allow only users config file to do things as root example: allow users to shutdown only if no one logged in system tracks addt'l user ID to help with those checks

anonymous feedback (1)

"Could we have some more office hours at the beginning/middle of the week instead of the end? I feel that it would be more helpful in the case that we have questions concerning the homework. Thank you."

anonymous feedback (2)

"Hello, I feel that the following would be helpful for the entire class as they start/continue to work on the homework: Is it possible for you to explain why the overhead time with clock_gettime() is longer than some of the provided scenarios? Even after incorporating the tips of Section 1.3, it still is longer. Thank you for your help."

could be: system slow during overhead measurement, fast otherwise also can be task (e.g. empty function call) being optimized away also I think some are confused about what overhead is:



want to time task part, but timed measured includes extra stuff

anonymous feedback (3)

"Do you mind going over what exactly the read, write, and execute permissions allow?"

```
regular files:
read — open file for reading
write — open file to write/print (modify contents)
execute — run the file as a program
directories:
read — list directory contents write — add/move/rename files in
directory execute (search) — access file/subdirectory within directory (if
already know name)
```

quiz Q2

question: when will control-C terminate program?

answer: when signal handler not setup yet

once sigaction() called, signal handler remains setup until another sigaction/etc. call

quiz Q4

aaa1a/ccc1c: read-only

bbb1b: write3

make bbb1b owner, give owner rw permissions

have aaa1a+ccc1c (and maybe also bbb1b) in group

associate that group w/file + give group r permission only

make default permissions nothing

anonymous feedback (4)

"Do you think you could post the recordings on Panopto please? The video player doesn't allow us to have captions or skip/go back 10/sec intervals"

program memory

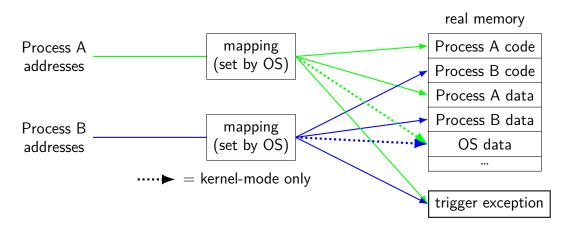
Used by OS
Stack
Heap / other dynamic
Writable data
Code + Constants

0xffff Ffff Ffff Ffff
0xffff 8000 0000 0000
0x7f...

0x0000 0000 0040 0000

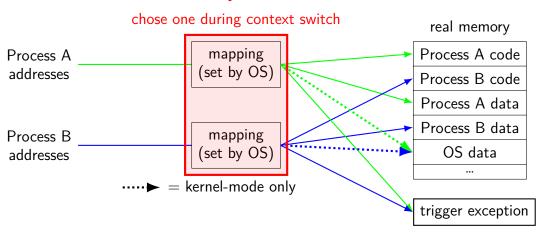
address spaces

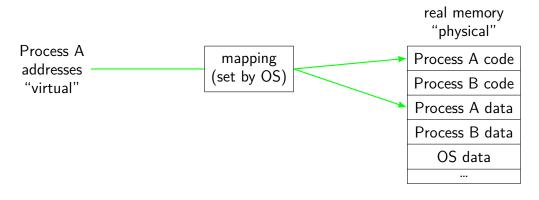
illuision of dedicated memory

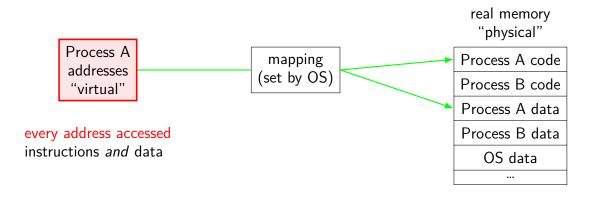


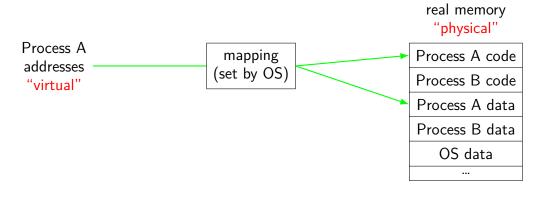
address spaces

illuision of dedicated memory



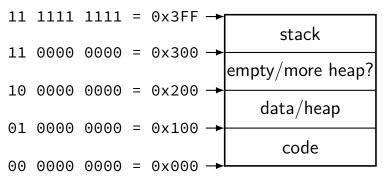


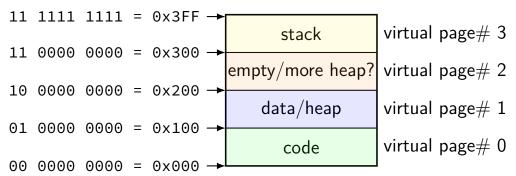


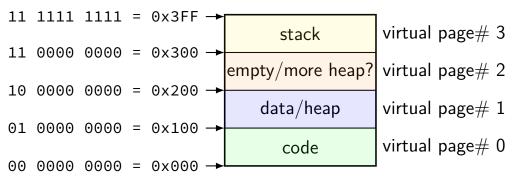


program addresses are 'virtual' real addresses are 'physical' can be different sizes!

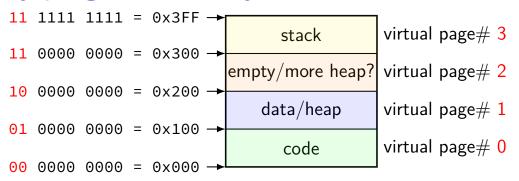




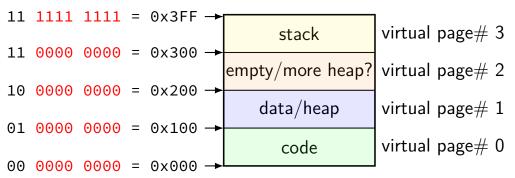




divide memory into pages (2^8 bytes in this case) "virtual" = addresses the program sees



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



rest of address is called page offset

toy physical memory

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	

real memory physical addresses

•				
1	.11	0000	0000	to
1	.11	1111	1111	
0	001	0000	0000	to
0	001	1111	1111	
0	000	0000	0000	to
0	00	1111	1111	

toy physical memory

program memory virtual addresses

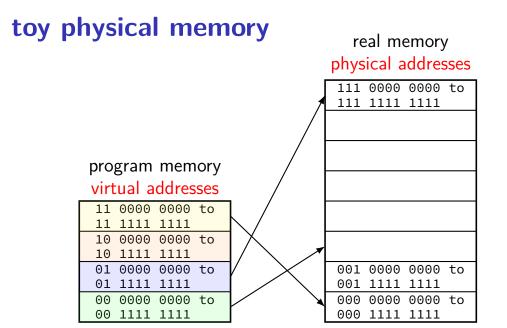
11	0000	0000	to
11	1111	1111	
10	0000	0000	to
10	1111	1111	
01	0000	0000	to
01	1111	1111	
00	0000	0000	to
00	1111	1111	

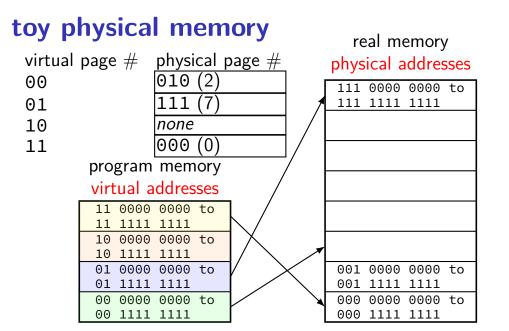
real memory physical addresses

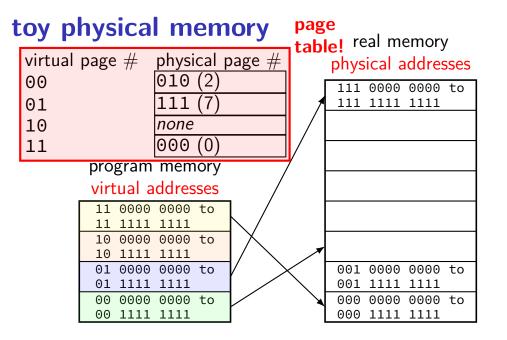
_				
l "	to	0000	0000	111
l b		1111	1111	111
ļ				
l		0000	0000	001
lр	to	0000		
"			1111	
lո	to	0000	0000	000
ı٢		1111	1111	000

physical page 7

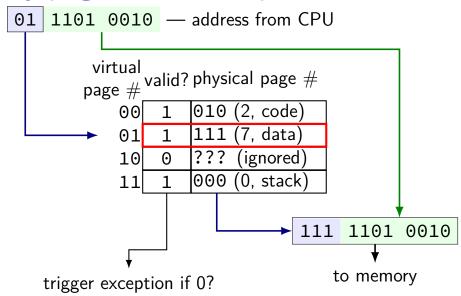
physical page 1 physical page 0

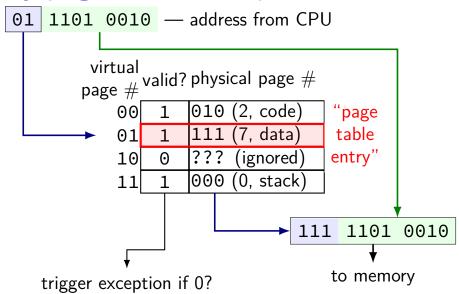


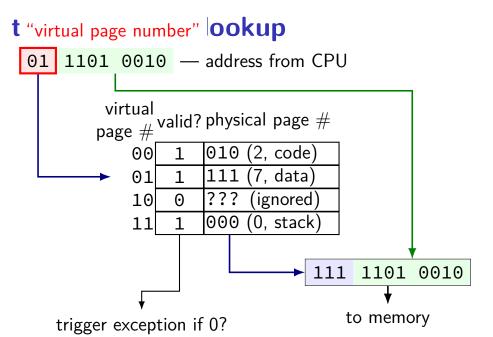


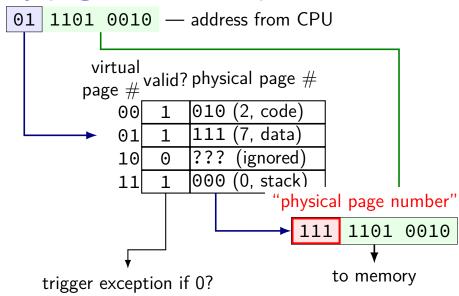


```
virtual page # valid? physical page # 00 1 010 (2, code) 01 1 111 (7, data) 10 0 ??? (ignored) 11 1 000 (0, stack)
```

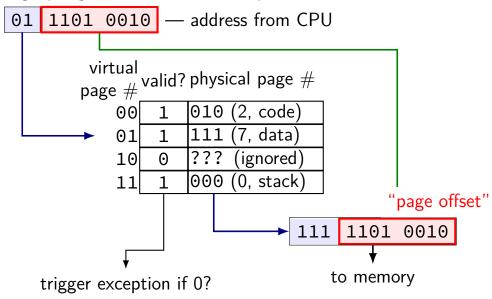








toy pag "page offset" ookup



on virtual address sizes

virtual address size = size of pointer?

often, but — sometimes part of pointer not used

example: typical x86-64 only use 48 bits rest of bits have fixed value

virtual address size is amount used for mapping

address space sizes

amount of stuff that can be addressed = address space size based on number of unique addresses

e.g. 32-bit virtual address = 2^{32} byte virtual address space

e.g. 20-bit physical addresss = 2^{20} byte physical address space

address space sizes

- amount of stuff that can be addressed = address space size based on number of unique addresses
- e.g. 32-bit virtual address = 2^{32} byte virtual address space
- e.g. 20-bit physical addresss = 2^{20} byte physical address space

what if my machine has 3GB of memory (not power of two)?

not all addresses in physical address space are useful
most common situation (since CPUs support having a lot of memory)

exercise: page counting

suppose 32-bit virtual (program) addresses

and each page is 4096 bytes (2^{12} bytes)

how many virtual pages?

exercise: page counting

suppose 32-bit virtual (program) addresses

and each page is 4096 bytes (2^{12} bytes)

how many virtual pages?

exercise: page table size

```
suppose 32-bit virtual (program) addresses suppose 30-bit physical (hardware) addresses each page is 4096 bytes (2^{12} bytes) pgae table entries have physical page \#, valid bit, bit
```

how big is the page table (if laid out like ones we've seen)?

exercise: page table size

```
suppose 32-bit virtual (program) addresses suppose 30-bit physical (hardware) addresses each page is 4096 bytes (2^{12} bytes) pgae table entries have physical page \#, valid bit, bit
```

how big is the page table (if laid out like ones we've seen)?

issue: where can we store that?

exercise: address splitting

and each page is 4096 bytes (2^{12} bytes)

split the address 0x12345678 into page number and page offset:

exercise: address splitting

and each page is 4096 bytes (2^{12} bytes)

split the address 0x12345678 into page number and page offset:

6-bit virtual addresses, 6-bit physical; 8 byte pages, 1 byte PTE page tables 1 page; PTE: 3 bit PPN (MSB), 1 valid bit, 4 other; page table base register 0x20; translate virtual address 0x31

physical addresses	byte	es			phys addres	sical sses	byt	es		
0x00-3			22	33	0x26				D2	D3
0x04-7	44	55	66	77	0x24	1-7	E4	E5	F6	07
0x08-B	88	99	AΑ	ВВ	0x28	3-B	89	9A	ΑB	ВС
0x0C-F	CC	DD	EE	FF	0x20	C-F	CD	DE	EF	F0
0x10-3	1A	2A	3A	4A	0x36	9-3	ВА	0A	ВА	0Α
0x14-7	1B	2B	3B	4B	0x34	1-7	СВ	0B	СВ	0B
0x18-B	1C	2C	3C	4C	0x38	3-B	DC	0C	DC	0C
0x1C-F	1C	2C	3C	4C	0x30	C-F	EC	0C	EC	0C

6-bit virtual addresses, 6-bit physical; 8 byte pages, 1 byte PTE page tables 1 page; PTE: 3 bit PPN (MSB), 1 valid bit, 4 other;

```
physical bytes
                       physical bytes
                                           0x31 = 11 0001
addresses
                      addresses
                                           PTE addr:
0x00-3|00 11 22 33
                      0x20-3|D0 D1 D2 D3
                                           0x20 + 6 \times 1 = 0x26
0x04-7|44 55 66 77
                      0x24-7|E4 E5 F6 07
0x08-B|88 99 AA BB
                      0x28-B|89 9A AB BC
                                          PTE value:
                      0x2C-FCD DE EF F0
0x0C-FICC DD EE FF
                                           0xF6 = 1111 0110
0x10-3|1A 2A 3A 4A
                      0x30-3|BA 0A BA 0A
                                           PPN 111, valid 1
0x14-7|1B 2B 3B 4B
                      0x34-7|CB 0B CB 0B
                                           M[111 \ 001] = M[0x39]
                      0x38-BDC 0C DC 0C
0x18-B|1C 2C 3C 4C
                                           \rightarrow 0x0C
0x1C-F|1C 2C 3C 4C
                      0x3C-FEC 0C EC 0C
```

6-bit virtual addresses, 6-bit physical; 8 byte pages, 1 byte PTE page tables 1 page; PTE: 3 bit PPN (MSB), 1 valid bit, 4 other;

```
physical bytes
                       physical bytes
                                           0x31 = 11 0001
                      addresses
addresses
                                           PTE addr:
0x00-3|00 11 22 33
                      0x20-3|D0 D1 D2 D3
                                           0x20 + 6 \times 1 = 0x26
0x04-7|44 55 66 77
                      0x24-7|E4 E5 F6 07
0x08-B|88 99 AA BB
                      0x28-B|89 9A AB BC
                                          PTE value:
                      0x2C-FCD DE EF F0
0x0C-FICC DD EE FF
                                           0xF6 = 1111 0110
0x10-3|1A 2A 3A 4A
                      0x30-3|BA 0A BA 0A
                                           PPN 111, valid 1
0x14-7|1B 2B 3B 4B
                      0x34-7|CB 0B CB 0B
                                           M[111 \ 001] = M[0x39]
                      0x38-BDC 0C DC 0C
0x18-Bl1C 2C 3C 4C
                                           \rightarrow 0x0C
0x1C-F|1C 2C 3C 4C
                      0x3C-FEC 0C EC 0C
```

6-bit virtual addresses, 6-bit physical; 8 byte pages, 1 byte PTE page tables 1 page; PTE: 3 bit PPN (MSB), 1 valid bit, 4 other;

```
physical bytes
addresses
0x00-3|00 11 22 33
0x04-7|44 55 66 77
0x08-B|88 99 AA BB
0x0C-FICC DD EE FF
0x10-3|1A 2A 3A 4A
0x14-7|1B 2B 3B 4B
0x18-Bl1C 2C 3C 4C
0x1C-F|1C 2C 3C 4C
```

```
physical bytes
addresses
0x20-3|D0 D1 D2 D3
0x24-7|E4 E5 F6 07
0x28-B|89 9A AB BC
0x2C-FCD DE EF F0
0x30-3|BA 0A BA 0A
0x34-7|CB 0B CB 0B
0x38-BDC 0C DC 0C
0x3C-FEC 0C EC 0C
```

```
0x31 = 11 0001
PTE \ addr:
0x20 + 6 \times 1 = 0x26
PTE \ value:
0xF6 = 1111 0110
PPN 111, \ valid 1
M[111 001] = M[0x39]
\rightarrow 0x0C
```

where can processor store megabytes of page tables? in memory

page table entry layout (chosen by processor)

valid (bit 15)|physical page # (bits 4–14)|other bits and/or unused (bit 0-3)|

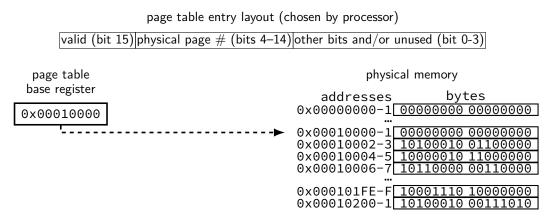
where can processor store megabytes of page tables? in memory

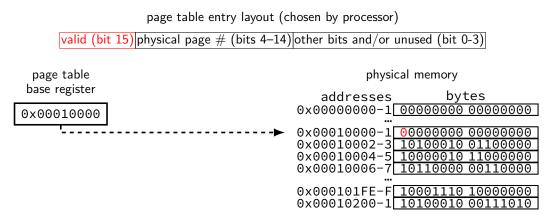
page table entry layout (chosen by processor)

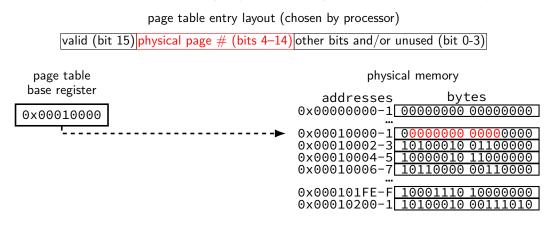
valid (bit 15) physical page # (bits 4–14) other bits and/or unused (bit 0-3)

page table base register

0x00010000







where can processor store megabytes of page tables? in memory

page table entry layout (chosen by processor) valid (bit $\overline{15}$) physical page # (bits 4–14) other bits and/or unused (bit 0-3) physical memory page table base register addresses bytes 0x00000000-1 00000000 00000000 0x00010000 0x00010000-1 00000000 00000000 $0 \times 00010002 - 3 10100010$ $0 \times 00010004 - 5\Gamma$ 10000010 0x00010006-7 10110000 0x000101FE-F 10001110 0x00010200-1 10100010 00111010

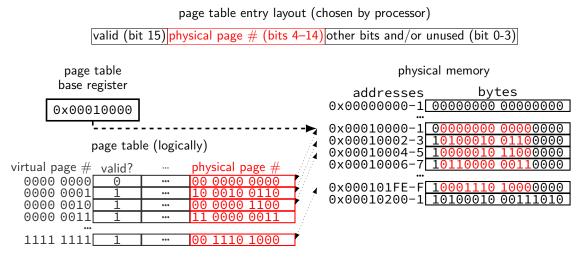
where can processor store megabytes of page tables? in memory

page table entry layout (chosen by processor) valid (bit 15) physical page # (bits 4–14) other bits and/or unused (bit 0-3) page table physical memory base register addresses bytes 0x0000000-1 00000000 00000000 0x00010000 0x00010000-1 00000000 00000000 $0 \times 00010002 - 3$ page table (logically) 0x00010004-5 10000010 0x00010006-7 10110000 00110000 virtual page # valid? physical page # 0000 0000 0000 0000 0x000101FE-F 10001110 0000 0001 0x00010200-1 10100010 00111010 0000 0010 0000 0011 0000 0011 1111 1111 00 1110 1000

where can processor store megabytes of page tables? in memory

valid (bit 15) physical page # (bits 4–14) other bits and/or unused (bit 0-3) page table physical memory base register addresses bytes 0x0000000-1 00000000 00000000 0x00010000 0x00010000-1 00000000 00000000 0x00010002-3 page table (logically) 0x00010004-5 0000010 0x00010006-7 0110000 00110000 virtual page # valid? physical page # 0000 0000 0000 0000 0x000101FE-F 10001110 0000 0001 0x00010200-1 10100010 00111010 0000 0010 0000 0011 0000 0011 1111 1111 00 1110 1000

page table entry layout (chosen by processor)

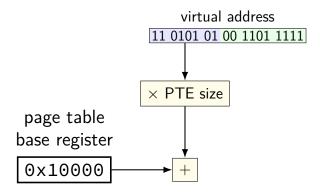


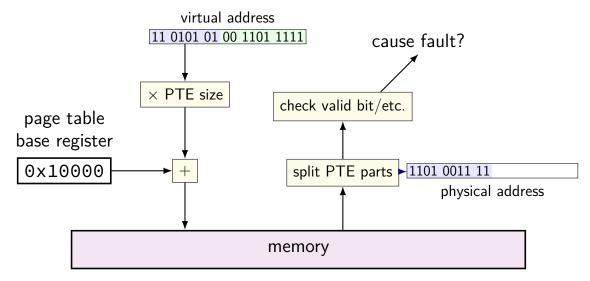
where can processor store megabytes of page tables? in memory

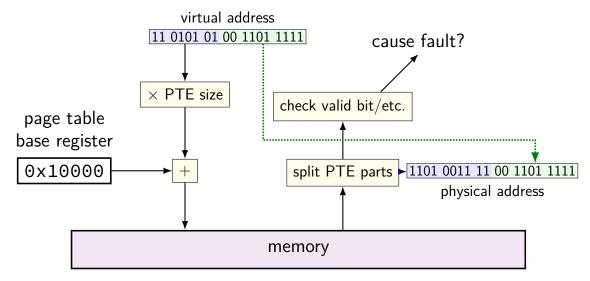
page table entry layout (chosen by processor) valid (bit 15) physical page # (bits 4–14) other bits and/or unused (bit 0-3) page table physical memory base register addresses bytes 0x0000000-1 00000000 00000000 0x00010000 0x00010000-1 00000000 000000000 $0 \times 00010002 - 3$ page table (logically) 0x00010004-5 10000010 0x00010006-7 10110000 001 virtual page # valid? physical page # 0000 0000 0000 0000 0x000101FE-F 10001110 0000 0001 0x00010200-1 10100010 00111010 0000 0010 0000 0011 0000 0011 1111 1111 00 1110 1000

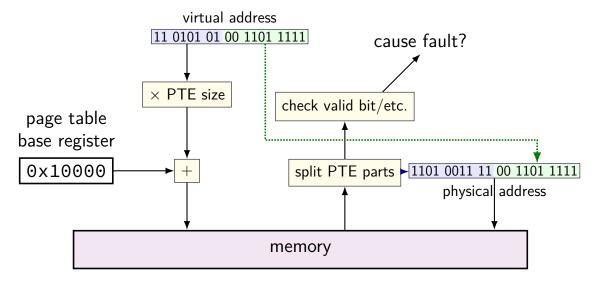
virtual address

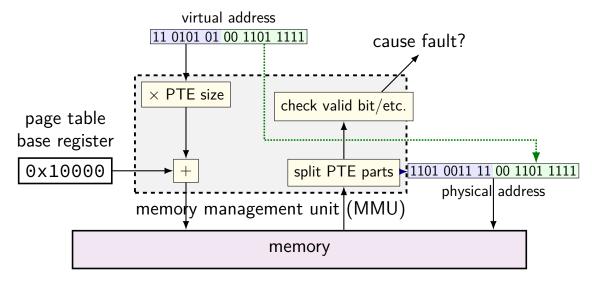
11 0101 01 00 1101 1111

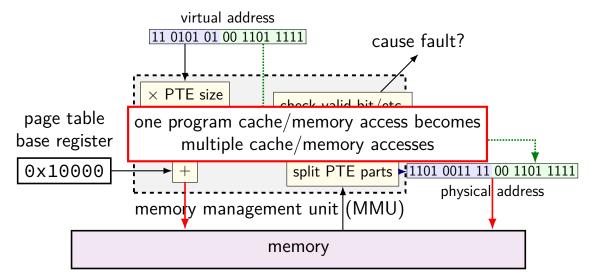


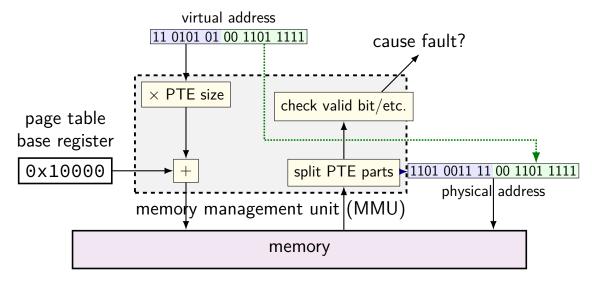












exercise setup

5-bit virtual addresses, 6-bit physical addresses, 8-byte pages

page table

virtual	valid?	physical		
page #	valid!	page #		
00	1	010		
01	1	111		
10	0	000		
11	1	000		

physical bytes addresses				
00	11	22	33	
44	55	66	77	
88	99	AΑ	ВВ	
CC	DD	EE	FF	
1A	2A	ЗА	4A	
1B	2B	3B	4B	
1C	2C	3C	4C	
1C	2C	3C	4C	
	00 44 88 CC 1A 1B	00 11 44 55 88 99 CC DD 1A 2A 1B 2B 1C 2C	bytes 00 11 22 44 55 66 88 99 AA CC DD EE 1A 2A 3A 1B 2B 3B 1C 2C 3C	

physical addresses	byte	es		
0x20-3	D0	D1	D2	D3
0x24-7	D4	D5	D6	D7
0x28-B	89	9A	ΑB	ВС
0x2C-F				
0x30-3	ВА	0A	ВА	0A
0x34-7	СВ	0B	СВ	0B
0x38-B				
0x3C-F	EC	0C	EC	0C

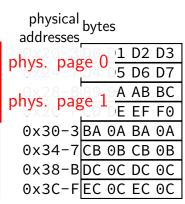
exercise setup

5-bit virtual addresses, 6-bit physical addresses, 8-byte pages

page table

virtual	2اء:اء،،	physical
page #	valid!	physical page #
00	1	010
01	1	111
10	0	000
11	1	000

physical bytes addresses					
0x00-3			22	33	
0x04-7	44	55	66	77	
0x08-B	88	99	AΑ	ВВ	
0x0C-F					
0x10-3	1A	2A	ЗА	4A	
0x14-7	1В	2B	3B	4B	
0x18-B	1C	2C	3C	4C	
0x1C-F	1C	2C	3C	4C	



5-bit virtual addresses, 6-bit physical addresses, 8-byte pages

(virtual addresses) 0x18 = ???; 0x03 = ???; 0x0A = ???; 0x13 = ???

page table

page # valid? _ 00 010 01 111 000 10 000 11

physical addresses	bytes
	00 11 22 33
0x04-7	44 55 66 77
0x08-B	88 99 AA BB
0x0C-F	CC DD EE FF
0x10-3	1A 2A 3A 4A
0x14-7	1B 2B 3B 4B
0x18-B	1C 2C 3C 4C
0x1C-F	1C 2C 3C 4C

physical bytes addresses 0x20-3 D0 D1 D2 D3 0x24-7 D4 D5 D6 D7 0x28-B|89 9A AB BC 0x2C-FCD DE EF F0 0x30-3|BA 0A BA 0A 0x34-7 CB 0B CB 0B 0x38-BDC 0C DC 0C 0x3C-FEC 0C EC 0C

5-bit virtual addresses, 6-bit physical addresses, 8-byte pages

page table

```
page # valid? ___
            1010
    001
    01
             111
             000
     10
    11
             000
```

physical addresses	byt	es		
0x00-3	00	11		
0x04-7	44	55	66	77
0x08-B	88	99	AA	ВВ
0x0C-F				
0x10-3	1A	2A	ЗА	4A
0x14-7	1B	2B	3B	4B
0x18-B	1C	2C	3C	4C
0x1C-F	1C	2C	3C	4C.

physical bytes addresses_					
0x20-3	D0	D1	D2	D3	
0x24-7					
0x28-B	89	9A	ΑB	ВС	
0x2C-F					
0x30-3	ВА	0A	ВА	0A	
0x34-7					
0x38-B	DC	0C	DC	0C	
0x3C-F	EC	0C	EC	0C	

5-bit virtual addresses, 6-bit physical addresses, 8-byte pages

```
(virtual addresses) 0x18 = ; 0x03 = ; 0x0A = ???; 0x13 = ??? page table
```

physical bytes					
addresses					
0x00-3					
0x04-7					
0x08-B					
0x0C-F	C	DD	EE	FF	
0x10-3	1A	2A	ЗА	4A	
0x14-7	1B	2B	3B	4B	
0x18-B	1C	2C	3C	4C	
0x1C-F	1C	2C	3C	4C	

physical addresses	byt	es		
0x20-3	D0	D1	D2	D3
0x24-7	D4	D5	D6	D7
0x28-B	89	9A	ΑB	ВС
0x2C-F	CD	DE	EF	F0
0x30-3	ВА	0A	ВА	ΘΑ
0x34-7	СВ	0B	СВ	0B
0x38-B	DC	0C	DC	0C
0x3C-F	EC	0C	EC	0C

5-bit virtual addresses, 6-bit physical addresses, 8-byte pages

```
(virtual addresses) 0x18 = ; 0x03 = ; 0x0A = ; 0x13 = ??? page table
```

virtual physical page # valid? page # 00 1 010 011 111 10 0 000 11 1 1 1000

physical bytes					
0x00-3	00 1	11	22	33	
0x04-7	44 5	55	66	77	
0x08-B	88 9	99	AA	ВВ	
0x0C-F	CC [DD	EE	FF	
0x10-3	1A 2	2A	ЗА	4A	
0x14-7	1B 2	2B	3B	4B	
0x18-B	1C 2	2C	3C	4C	
0x1C-F	1C 2	2C	3C	4C	

physical bytes addresses 0x20-3 D0 D1 D2 D3 0x24-7 D4 D5 D6 D7 0x28-B|89 9A AB BC 0x2C-FCD DE EF F0 0x30-3|BA 0A BA 0A 0x34-7 CB 0B CB 0B 0x38-BDC 0C DC 0C 0x3C-FEC 0C EC 0C

5-bit virtual addresses, 6-bit physical addresses, 8-byte pages

```
(virtual addresses) 0x18 = ; 0x03 = ; 0x0A = ; 0x13 = page table
```

physical addresses	byte	S		
0x00-3	00	11	22	33
0x04-7	44 !	55	66	77
0x08-B	88 9	99	AA	ВВ
0x0C-F	CC I	DD	EE	FF
0x10-3	1A 2	2A	ЗА	4A
0x14-7	1B 2	2B	3B	4B
0x18-B	1C 2	2C	3C	4C
0×1C-F	10 1	2 C	30	<u>4</u> C

physical bytes addresses 0x20-3 D0 D1 D2 D3 0x24-7 D4 D5 D6 D7 0x28-B|89 9A AB BC 0x2C-FCD DE EF F0 0x30-3|BA 0A BA 0A 0x34-7 CB 0B CB 0B 0x38-BDC 0C DC 0C 0x3C-FEC 0C EC 0C

6-bit virtual addresses, 6-bit physical; 8 byte pages, 1 byte PTE page tables 1 page; PTE: 3 bit PPN (MSB), 1 valid bit, 4 other; page table base register 0x20; translate virtual address 0x31

physical addresses	byt	es			physic address	al es	byt	es		
0x00-3			22	33	0x20-	-3	D0	D1	D2	D3
0x04-7	44	55	66	77	0x24-	-7	E4	E5	F6	07
0x08-B	88	99	AΑ	ВВ	0x28-	-B	89	9Α	ΑB	ВС
0x0C-F	CC	DD	EE	FF	0x2C-	-F	CD	DE	EF	F0
0x10-3	1A	2A	ЗА	4A	0x30-	-3	ВА	0Α	ВА	0A
0x14-7	1В	2B	3B	4B	0x34-	-7	СВ	0B	СВ	0B
0x18-B	1C	2C	3C	4C	0x38-	-B	DC	0C	DC	0C
0x1C-F	1C	2C	3C	4C	0x3C-	- F	EC	0C	EC	0C

6-bit virtual addresses, 6-bit physical; 8 byte pages, 1 byte PTE page tables 1 page; PTE: 3 bit PPN (MSB), 1 valid bit, 4 other;

```
physical bytes
addresses
0x00-3|00 11 22 33
0x04-7|44 55 66 77
0x08-B|88 99 AA BB
0x0C-FICC DD EE FF
0x10-3|1A 2A 3A 4A
0x14-7|1B 2B 3B 4B
0x18-Bl1C 2C 3C 4C
0x1C-F|1C 2C 3C 4C
```

```
physical bytes
addresses
0x20-3|D0 D1 D2 D3
0x24-7|E4 E5 F6 07
0x28-B|89 9A AB BC
0x2C-FCD DE EF F0
0x30-3|BA 0A BA 0A
0x34-7|CB 0B CB 0B
0x38-BDC 0C DC 0C
0x3C-FEC 0C EC 0C
```

```
0 \times 31 = 11 0001
PTE \ addr:
0 \times 20 + 6 \times 1 = 0 \times 26
PTE \ value:
0 \times F6 = 1111 0110
PPN 111, \ valid 1
M[111 001] = M[0 \times 39]
\rightarrow 0 \times 0 C
```

6-bit virtual addresses, 6-bit physical; 8 byte pages, 1 byte PTE page tables 1 page; PTE: 3 bit PPN (MSB), 1 valid bit, 4 other;

page table base register 0x20; translate virtual address 0x31

physical addresses	bvt	es			
					, ;
0x00-3					
0x04-7	44	55	66	77	
0x08-B	88	99	AΑ	ВВ	
0x0C-F	CC	DD	EE	FF	
0x10-3	1A	2A	3A	4A	
0x14-7	1В	2B	3B	4B	
0x18-B	1C	2C	3C	4C	
0x1C-F	1C	2C	3C	4C	

```
physical bytes
addresses
0x20-3|D0 D1 D2 D3
0x24-7 E4 E5 F6 07
0x28-Bl89 9A AB BC
0x2C-FCD DE EF F0
0x30-3|BA 0A BA 0A
0x34-7|CB 0B CB 0B
0x38-BDC 0C DC 0C
0x3C-F|EC 0C EC 0C
```

```
0 \times 31 = 11 0001
PTE \ addr:
0 \times 20 + 6 \times 1 = 0 \times 26
PTE \ value:
0 \times F6 = 1111 0110
PPN \ 111, \ valid \ 1
M[111 \ 001] = M[0 \times 39]
\rightarrow 0 \times 0 C
```

6-bit virtual addresses, 6-bit physical; 8 byte pages, 1 byte PTE page tables 1 page; PTE: 3 bit PPN (MSB), 1 valid bit, 4 other;

page table base register 0x20; translate virtual address 0x31

```
physical bytes
                       physical bytes
                                           0x31 = 11 \ 0001
addresses
                      addresses
                                           PTE addr:
0x00-3|00 11 22 33
                      0x20-3|D0 D1 D2 D3
                                           0x20 + 6 \times 1 = 0x26
0x04-7|44 55 66 77
                      0x24-7|E4 E5 F6 07
0x08-B|88 99 AA BB
                      0x28-B|89 9A AB BC
                                           PTE value:
                      0x2C-FCD DE EF F0
0x0C-FICC DD EE FF
                                           0xF6 = 1111 0110
0x10-3|1A 2A 3A 4A
                      0x30-3|BA 0A BA 0A
                                           PPN 111, valid 1
0x14-7|1B 2B 3B 4B
                      0x34-7|CB 0B CB 0B
                                           M[111 \ 001] = M[0x39]
                      0x38-BDC 0C DC 0C
0x18-Bl1C 2C 3C 4C
                                           \rightarrow 0x0C
0x1C-F|1C 2C 3C 4C
                      0x3C-FEC 0C EC 0C
```

26

physical byte addresses	es	physical bytes addresses
0x00-300		0x20-3A0 D1 E2 F3
0x04-744	55 66 77	0x24-7E4E5F6 07
0x08-B88	99 AA BB	0x28-B89 9A AB BC
0x0C-FCC	DD EE FF	0x2C-FCD DE EF F0
0x10-3 1A	2A 3A 4A	0x30-3BA 0A BA 0A
0x14-7 1B	2B 3B 4B	0x34-7 CB 0B CB 0B
0x18-B1C	2C 3C 4C	0x38-BDC 0C DC 0C
0x1C-F1C	2C 3C 4C	0x3C-FEC 0C EC 0C

physical addresses	byt	es			
0x00-3	00	11	22	33	
0x04-7	44	55	66	77	
0x08-B	88	99	AA	ВВ	
0x0C-F					
0x10-3	1A	2A	ЗА	4A	
0x14-7	1В	2B	3B	4B	
0x18-B	1C	2C	3C	4C	
0x1C-F	1C	2C	3C	4C	

```
physical bytes
addresses
0x20-3 A0 D1 E2 F3
0x24-7 E4 E5 F6 07
0x28-Bl89 9A AB BC
0x2C-FCD DE EF F0
0x30-3|BA 0A BA 0A
0x34-7|CB 0B CB 0B
0x38-BDC 0C DC 0C
0x3C-F|EC 0C EC 0C
```

```
0x12 = 01 0010

PTE addr:
0x20 + 2 \times 1 = 0x22

PTE value:
0xD1 = 1101 0001

PPN 110, valid 1

M[110 001] = M[0x32]

\rightarrow 0xBA
```

physical addresses	bytes	physica addresses
	00 11 22 33	0x20-3
0x04-7	44 55 66 77	0x24-7
0x08-B	88 99 AA BB	0x28-E
0x0C-F	CC DD EE FF	0x2C-F
0x10-3	1A 2A 3A 4A	0x30-3
0x14-7	1B 2B 3B 4B	0x34-7
0x18-B	1C 2C 3C 4C	0x38-E
0x1C-F	1C 2C 3C 4C	0x3C-F

```
al bytes
               0x12 = 01 0010
               PTE addr:
3|A0 D1 E2 F3
               0x20 + 2 \times 1 = 0x22
7|E4 E5 F6 07 |
B|89 9A <mark>AB</mark> BC | PTE value:
FCD DE EF FO
                0xD1 = 1101 0001
BIBA OA BA OA
               PPN 110, valid 1
 CB 0B CB 0B
               M[110 \ 001] = M[0x32]
BIDC OC DC OC
               \rightarrow 0xBA
FIEC OC EC OC
```

```
physical bytes
                       physical bytes
                                            0 \times 12 = 01 \ 0010
                      addresses
addresses
                                            PTE addr:
                       0x20-3 A0 D1 E2 F3
0x00-3|00 11 22 33
                                            0x20 + 2 \times 1 = 0x22
0x04-7|44 55 66 77
                       0x24-7|E4 E5 F6 07
0x08-B|88 99 AA BB
                       0x28-B|89 9A AB BC
                                           PTE value:
                       0x2C-FCD DE EF F0
0x0C-FICC DD EE FF
                                            0xD1 = 1101 0001
0x10-3|1A 2A 3A 4A
                       0x30-3|BA 0A BA 0A
                                            PPN 110, valid 1
0x14-7|1B 2B 3B 4B
                       0x34-7|CB 0B CB 0B
                                            M[110 \ 001] = M[0x32]
                       0x38-BDC 0C DC 0C
0x18-B|1C 2C 3C 4C
                                            \rightarrow 0xBA
0x1C-F|1C 2C 3C 4C
                       0x3C-FEC 0C EC 0C
```

6-bit virtual addresses, 6-bit physical; 8 byte pages, 1 byte PTE page tables 1 page; PTE: 3 bit PPN (MSB), 1 valid bit, 4 other page table base register 0x20; translate virtual address 0x12

physical addresses	bytes		physical addresses	bytes		0x12 =
0x00-3	00 11	22 33	0x20-3	A0 D1 E2	2 F3	PTE add
0x04-7	44 55	66 77	0x24-7	E4 E5 F6	6 07	0x20 +
0x08-B	88 99	AA BB	0x28-B	89 9A AE	в вс	PTE val
0x0C-F	CC DD	EE FF	0x2C-F	CD DE EF	F F0	0xD1 =
0x10-3	1A 2A	3A 4A	0x30-3	BA OA BA	A 0 A	PPN 11
0x14-7			1	CB 0B CE		
0x18-B	1C 2C	3C 4C	0x38-B	DC 0C DO	C 0C	M[110
0x1C-F	1C 2C	3C 4C	0x3C-F	EC 0C EC	C 0C	$\rightarrow 0xBA$

0x12 = 01 0010 $PTE \ addr$: $0x20 + 2 \times 1 = 0x22$ $PTE \ value$: 0xD1 = 1101 0001 PPN 110, valid 1 M[110 001] = M[0x32] $\rightarrow 0xBA$

pagetable assignment

```
pagetable assignment
```

simulate page tables (on top of normal program memory) alternately: implement another layer of page tables on top of the existing system's

in assignment:

virtual address \sim arguments to your functions

physical address \sim your program addresses (normal pointers)

pagetable assignment API

```
/* configuration parameters */
#define POBITS ...
#define LEVELS /* later /
size_t ptbr; // page table base register
    // points to page table (array of page table entries)
// lookup "virtual" address 'va' in page table ptbr points to
// return (void*) (~0L) if invalid
void *translate(size t va);
// make it so 'va' is valid, allocating one page for its data
// if it isn't already
void page_allocate(size_t va)
```

translate()

with POBITS=12, LEVELS=1:

ptbr = GetPointerToTable(

٧r	ıv vand	a i priysicai	
0	0		
1	1	0×9999	١,
2	0	_	7
3	1	0x3333	

VDN valid2 physical

```
\begin{array}{l} translate(0x0FFF) == (void^*) ~0L \\ translate(0x1000) == (void^*) ~0x9999000 \\ translate(0x1001) == (void^*) ~0x9999001 \\ translate(0x2000) == (void^*) ~0L \\ translate(0x2001) == (void^*) ~0L \\ translate(0x3000) == (void^*) ~0x3333000 \\ \end{array}
```

translate()

with POBITS=12, LEVELS=1:

ptbr = GetPointerToTable(

VPN Valid? physical					
0	0				
1	1	0×9999	١		
2	0)		
3	1	0x3333			

1/DM - 1:12 mby/size1

```
\begin{array}{l} translate(0x0\text{FFF}) == (void^*) ~0L \\ translate(0x1000) == (void^*) ~0x9999000 \\ translate(0x1001) == (void^*) ~0x9999001 \\ translate(0x2000) == (void^*) ~0L \\ translate(0x2001) == (void^*) ~0L \\ translate(0x3000) == (void^*) ~0x3333000 \\ \end{array}
```

page_allocate()

```
with POBITS=12, LEVELS=1:  ptbr == 0 \\ page\_allocate(0 \times 1000) \ \textit{or} \ page\_allocate(0 \times 1001) \ \textit{or} \ ... \\
```

page_allocate()

```
with POBITS=12, LEVELS=1: 
 ptbr == 0 page\_allocate(0x1000) \ or \ page\_allocate(0x1001) \ or \ ...
```

 $\mathsf{ptbr}\ \mathit{now} == \mathsf{GetPointerToTable}($

VEIN Vallu! pirysicai						
	0	0				
	1	1	(new))	١	
	2	0		,)	
	3	1				

V/DNI valid2 physical

allocated with posix_memalign

page_allocate()

```
with POBITS=12, LEVELS=1: ptbr == 0 page_allocate(0x1000) or page_allocate(0x1001) or ...
```

 $\mathsf{ptbr}\ \mathit{now} == \mathsf{GetPointerToTable}($

VI IN Vallu: physical							
	0	0					
	1	1	(new))	١		
	2	0	_ ^)		
	3	1					

VPN valid2 physical

allocated with posix_memalign

posix_memalign

```
void *result;
error code =
     posix_memalign(&result, alignment, size);
allocate size bytes
choosing address that is multiple of alignment
    can make sure allocation starts at beginning of page
error_code indicates if out-of-memory, etc.
fills in result (passed via pointer)
```

posix_memalign

```
void *result;
error code =
     posix_memalign(&result, alignment, size);
allocate size bytes
choosing address that is multiple of alignment
    can make sure allocation starts at beginning of page
error_code indicates if out-of-memory, etc.
fills in result (passed via pointer)
```

posix_memalign

```
void *result;
error code =
     posix_memalign(&result, alignment, size);
allocate size bytes
choosing address that is multiple of alignment
    can make sure allocation starts at beginning of page
error_code indicates if out-of-memory, etc.
fills in result (passed via pointer)
```

parts

```
part 1 (next week): LEVELS=1, POBITS=12 and
    translate() OR
     page_allocate()
part 2: all LEVELS, both functions
    in preparation for code review
     originally scheduled for lab on the 27th
    will move to lab just after reading day
     (might mean I need to cancel lab one week)
part 3: final submission
     Friday after code review
     most of grade based on this
     will test previous parts again
```

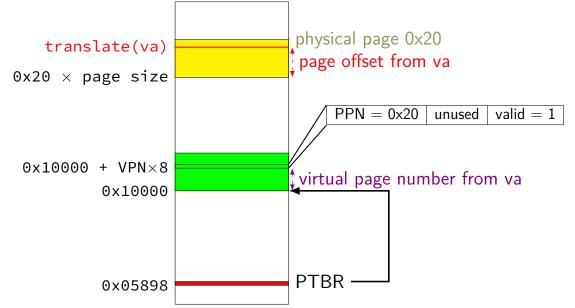
address/page table entry format

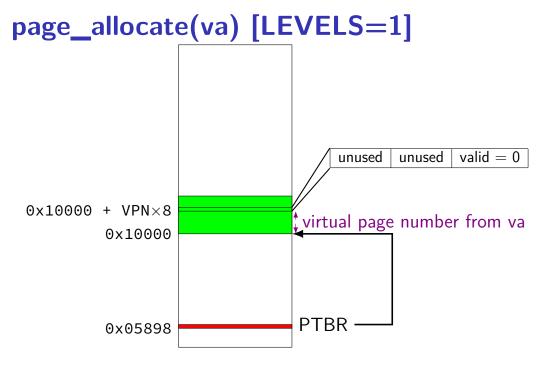
(with POBITS=12, LEVELS=1)

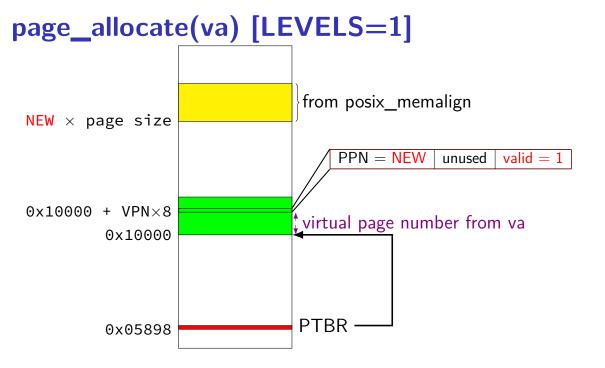
	bits 63–21	bits 20–12	bits 11–1	bit 0
page table entry	physic	al page number	unused	valid bit
virtual address	unused virtual page number		page o	offset
physical address	physical page number		page o	offset

in assignment: value from posix_memalign = physical address

pa = translate(va) [LEVELS=1]







my desktop: 39-bit physical addresses; 48-bit virtual addresses

4096 byte pages

my desktop: 39-bit physical addresses; 48-bit virtual addresses

4096 byte pages

top 16 bits of 64-bit addresses not used for translation

my desktop: 39-bit physical addresses; 48-bit virtual addresses

4096 byte pages

exercise: how many page table entries? (assuming page table like shown before)

exercise: how large are physical page numbers?

my desktop: 39-bit physical addresses; 48-bit virtual addresses

4096 byte pages

exercise: how many page table entries? (assuming page table like shown before)

exercise: how large are physical page numbers?

```
my desktop: 39-bit physical addresses; 48-bit virtual addresses
```

4096 byte pages

exercise: how many page table entries? (assuming page table like shown before)

exercise: how large are physical page numbers?

page table entries are 8 bytes (room for expansion, metadata) trick: power of two size makes table lookup faster

would take up 2^{39} bytes?? (512GB??)

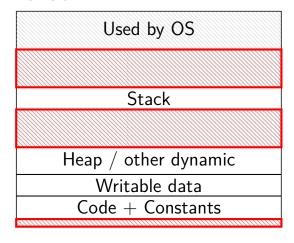
huge page tables

huge virtual address spaces!

impossible to store PTE for every page

how can we save space?

holes



most pages are invalid

saving space

basic idea: don't store (most) invalid page table entries
use a data structure other than a flat array
 want a map — lookup key (virtual page number), get value (PTE)
options?

saving space

```
basic idea: don't store (most) invalid page table entries
use a data structure other than a flat array
want a map — lookup key (virtual page number), get value (PTE)
options?
```

hashtable

actually used by some historical processors but never common

saving space

```
basic idea: don't store (most) invalid page table entries
use a data structure other than a flat array
want a map — lookup key (virtual page number), get value (PTE)
options?
```

hashtable

actually used by some historical processors but never common

tree data structure

but not quite a search tree

search tree tradeoffs

lookup usually implemented in hardware

lookup should be simple solution: lookup splits up address bits (no complex calculations)

lookup should not involve many memory accesses

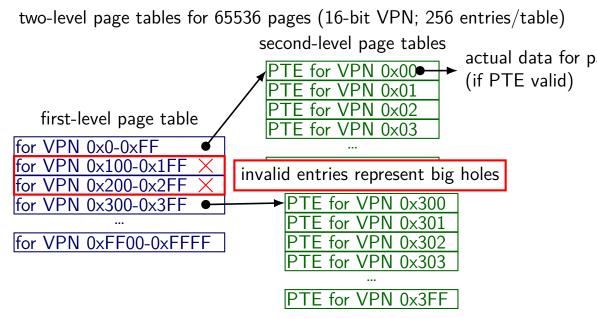
doing two memory accesses is already very slow solution: tree with many children from each node (far from binary tree's left/right child)

two-level page tables

two-level page tables for 65536 pages (16-bit VPN; 256 entries/table) second-level page tables actual data for p for VPN 0x00 (if PTE valid) first-level page table for VPN $0 \times 0 - 0 \times FF$ for VPN 0x100-0x1FF PTE for VPN 0xFF VPN 0x200-0x2FF VPN 0x300 for VPN 0x300-0x3FF for VPN 0xFF00-0xFFFF ΓE for VPN 0x302 TE for VPN 0x303

for VPN 0x3FF

two-level page tables



two-level page tables

two-level page tables for 65536 pages (16-bit VPN: 256 entries/table) first-level page table for p physical page # VPN range valid d) (of next page table) 0x0000-0x00FF 0x22343 first-level pag $0 \times 0100 - 0 \times 01 FF$ 0 0×00000 VPN 0x0-0xF $0 \times 0200 - 0 \times 02FF$ 0 0×00000 VPN 0x100-0 $0 \times 0300 - 0 \times 03FF$ 0x33454 VPN 0x200- $0 \times 0400 - 0 \times 04FF$ 0xFF043 0xFF045 $0 \times FF00 - 0 \times FFFF$ •••

TE for VPN 0x3FF

43

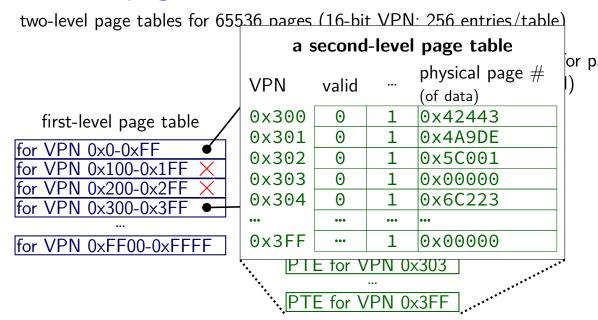
two-level page tables for 65536 pages (16-bit VPN: 256 entries/table) first-level page table for p physical page # VPN range valid d) (of next page table) 0x0000-0x00FF 0x22343 first-level pag $0 \times 0100 - 0 \times 01 FF$ 0 0×00000 VPN 0x0-0xF $0 \times 0200 - 0 \times 02FF$ 0 000000 VPN 0x100-0 $0 \times 0300 - 0 \times 03FF$ 0x33454 VPN 0x200- $0 \times 0400 - 0 \times 04FF$ 0xFF043 $0 \times FF00 - 0 \times FFFF$ 0xFF045 •••

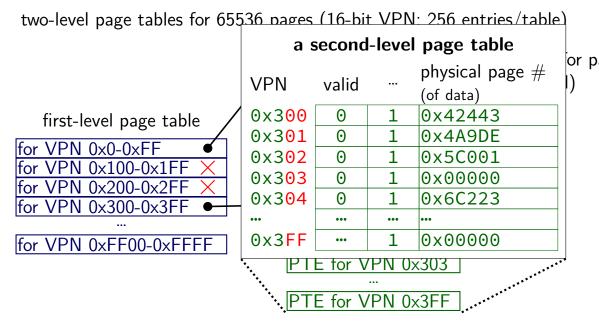
TE for VPN 0x3FF

43

two-level page tables for 65536 pages (16-bit VPN: 256 entries/table) first-level page table for p physical page # VPN range valid d) (of next page table) $0 \times 0 0 0 0 - 0 \times 0 0 FF$ 0x22343 first-level pag $0 \times 0100 - 0 \times 01FF$ 0 0×00000 VPN 0x0-0xF 0 0×00000 VPN 0x100-0 $0 \times 0300 - 0 \times 03FF$ 0x33454 VPN 0x200- $0 \times 0400 - 0 \times 04FF$ 0xFF043 0xFF045 $0 \times FF00 - 0 \times FFFF$ •••

PTE for VPN 0x3FF





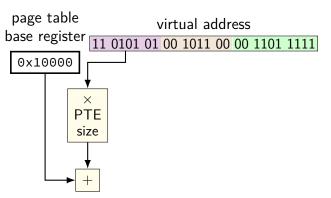
two-level page tables for 65536 pages (16-bit VPN; 256 entries/table) second-level page tables actual data for p for VPN 0x00 (if PTE valid) first-level page table for VPN $0 \times 0 - 0 \times FF$ tor VPN 0x100-0x1FFIPTE for VPN 0xFF VPN 0x200-0x2FF for VPN 0x300-0x3FF VPN 0x300 for VPN 0xFF00-0xFFFF VPN 0x302 TE for VPN 0x303 for VPN 0x3FF

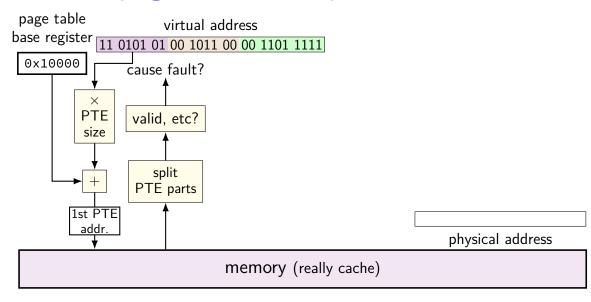
virtual address

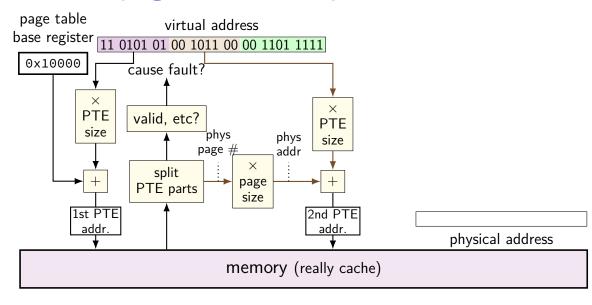
11 0101 01 00 1011 00 00 1101 1111

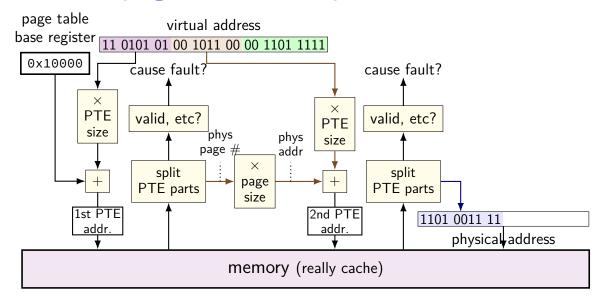
VPN — split into two parts (one per level)

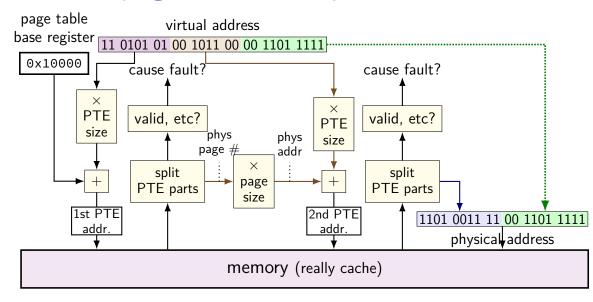
this example: parts equal sized — common, but not required

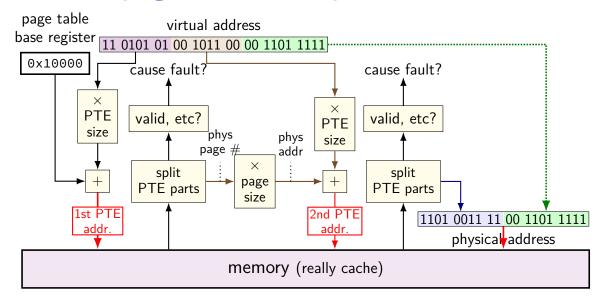


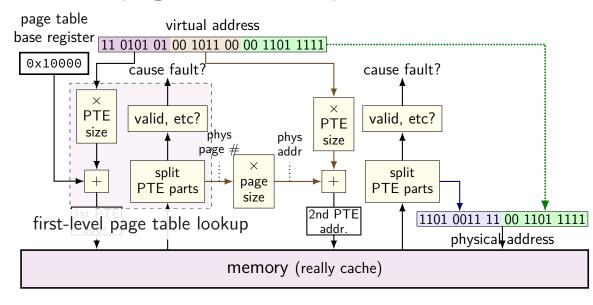


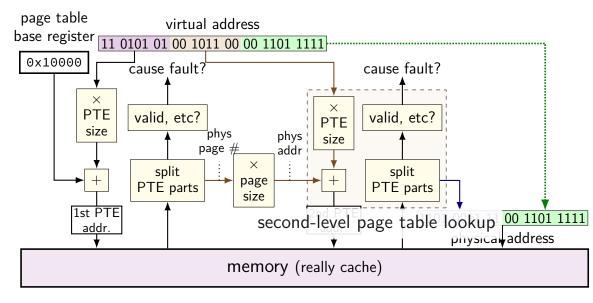


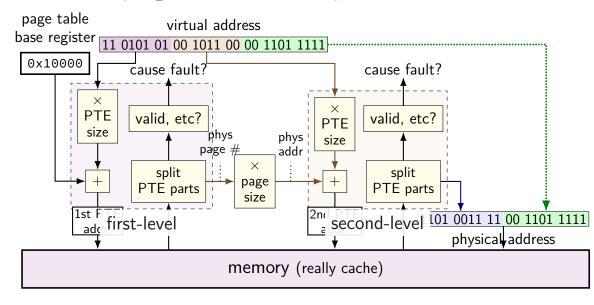


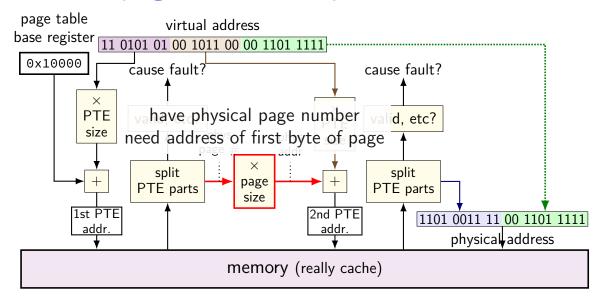


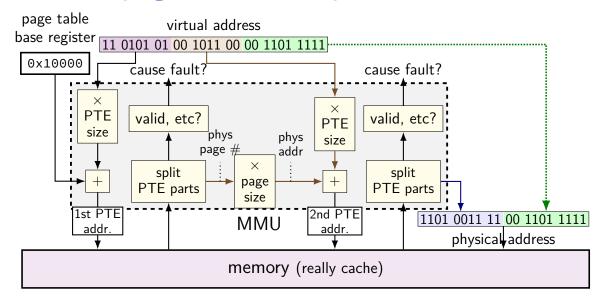




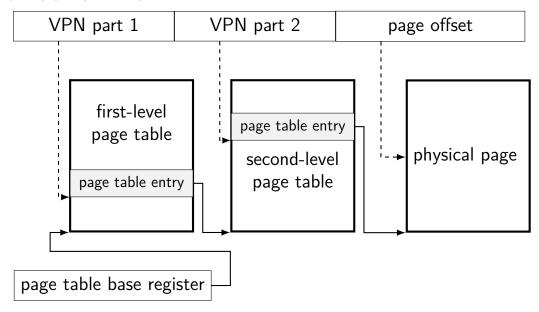








another view



multi-level page tables

VPN split into pieces for each level of page table

top levels: page table entries point to next page table usually using physical page number of next page table

bottom level: page table entry points to destination page

validity checks at each level

x86-64 page table splitting

48-bit virtual address

12-bit page offset (4KB pages)

36-bit virtual page number, split into four 9-bit parts

page tables at each level: 2^9 entries, 8 bytes/entry deliberate choice: each page table is one page

note on VPN splitting

indexes used for lookup parts of the virtual page number (there are not multiple VPNs)

assignment

physical addresses	bytes	
	00 11 22 33	7
	44 55 66 77	
0x08-B	88 99 AA BB	
0x0C-F	CC DD EE FF	1
0x10-3	1A 2A 3A 4A	
0x14-7	1B 2B 3B 4B	
0x18-B	1C 2C 3C 4C	
0x1C-F	1C 2C 3C 4C	

physical addresses	byt	es		
0x20-3	00	91	72	13
0x24-7	D4	F5	36	07
0x28-B				
0x2C-F	CD	DE	EF	F0
0x30-3	ВА	0A	ВА	ΘΑ
0x34-7	DB	0B	DB	0B
0x38-B	EC	0C	EC	0C
0x3C-F	FC	0C	FC.	0C

physical addresses	byte	S			phys addres	ical sses	byt	es		
0x00-3			22	33	0x20				72	13
0x04-7	44 !	55	66	77	0x24	-7	D4	F5	36	07
0x08-B	88 9	99	AA	ВВ	0x28	-В	89	9A	ΑB	ВС
0x0C-F	CC I	DD	EE	FF	0x20	-F	CD	DE	EF	F0
0x10-3	1A 2	2A	ЗА	4A	0x30	-3	ВА	0A	ВА	0A
0x14-7	1B 2	2B	3B	4B	0x34	-7	DB	0B	DB	0B
0x18-B	1C 2	2C	3C	4C	0x38	-В	EC	0C	EC	0C
0x1C-F	1C 2	2C	3C	4C	0x30	-F	FC	0C	FC	0C

physical addresses	byte	es			physical addresses	byt	es		
0x00-3			22	33	0x20-3			72	13
0x04-7	44	55	66	77	0x24-7	D4	F5	36	07
0x08-B	88	99	AA	ВВ	0x28-B	89	9A	ΑB	ВС
0x0C-F	CC	DD	EE	FF	0x2C-F	CD	DE	EF	F0
0x10-3	1A	2A	ЗА	4A	0x30-3	ВА	0A	ВА	0Α
0x14-7	1B	2B	3B	4B	0x34-7	DB	0B	DB	0B
0x18-B	1C	2C	3C	4C	0x38-B	EC	0C	EC	оC
0x1C-F	1C	2C	3C	4C	0x3C-F	FC	0C	FC	0C

physical addresses	byte	es			physica addresse	al byt	es		
0x00-3			22	33	0x20-			72	13
0x04-7	44	55	66	77	0x24-	7 D4	F5	36	07
0x08-B	88	99	AA	ВВ	0x28-	B 89	9A	AB	ВС
0x0C-F	CC	DD	EE	FF	0x2C-	FCD	DE	EF	F0
0x10-3	1A	2A	ЗА	4A	0x30-	3ВА	0A	ВА	0A
0x14-7	1В	2B	3B	4B	0x34-	7DB	0B	DB	0B
0x18-B	1C	2C	3C	4C	0x38-	BEC	0C	EC	0C
0x1C-F	1C	2C	3C	4C	0x3C-	FFC	0C	FC	0C

physical addresses	byte	es			physica addresse	al byt	es		
0x00-3			22	33	0x20-			72	13
0x04-7	44	55	66	77	0x24-	7 D4	F5	36	07
0x08-B	88	99	AA	ВВ	0x28-	B 89	9A	AB	ВС
0x0C-F	CC	DD	EE	FF	0x2C-	FCD	DE	EF	F0
0x10-3	1A	2A	ЗА	4A	0x30-	3ВА	0A	ВА	0A
0x14-7	1В	2B	3B	4B	0x34-	7DB	0B	DB	0B
0x18-B	1C	2C	3C	4C	0x38-	BEC	0C	EC	0C
0x1C-F	1C	2C	3C	4C	0x3C-	FFC	0C	FC	0C

physical addresses	byte	es			physical addresses	byt	es		
0x00-3			22	33	0x20-3			72	13
0x04-7	44	55	66	77	0x24-7	D4	F5	36	07
0x08-B	88	99	AA	ВВ	0x28-B	89	9A	ΑB	ВС
0x0C-F	CC	DD	EE	FF	0x2C-F	CD	DE	EF	F0
0x10-3	1A	2A	ЗА	4A	0x30-3	ВА	0A	ВА	0Α
0x14-7	1В	2B	3B	4B	0x34-7	DB	0B	DB	0B
0x18-B	1C	2C	3C	4C	0x38-B	EC	0C	EC	0C
0x1C-F	1C	2C	3C	4C	0x3C-F	FC	0C	FC	0C

2-level splitting

- 9-bit virtual address
- 6-bit physical address

- 8-byte pages \rightarrow 3-bit page offset (bottom bits)
- 9-bit VA: 6 bit VPN + 3 bit PO
- 6-bit PA: 3 bit PPN + 3 bit PO

- 8 entry page tables \rightarrow 3-bit VPN parts
- 9-bit VA: 3 bit VPN part 1; 3 bit VPN part 2

physical addresses	byt	es			physical addresses	byt	es		
0x00-3			22	33	0x20-3			D2	D3
0x04-7	44	55	66	77	0x24-7	D4	D5	D6	D7
0x08-B	88	99	AA	ВВ	0x28-B	89	9A	AB	ВС
0x0C-F	CC	DD	ΕE	FF	0x2C-F	CD	DE	EF	F0
0x10-3	1A	2A	ЗА	4A	0x30-3	ВА	0A	ВА	0A
0x14-7	1В	2B	3B	4B	0x34-7	DB	0B	DB	0B
0x18-B	1C	2C	3C	4C	0x38-B	EC	0C	EC	0C
0x1C-F	1C	2C	3C	4C	0x3C-F	FC	0C	FC	0C

physical background	ytes			r ad	ohysical Idresses	byt	es		
0x00-30	0 11	22	33	0	x20-3	D0	D1	D2	D3
0x04-74	4 55	66	77	0	x24-7	D4	D5	D6	D7
0x08-B8	8 99	AA	ВВ	0	x28-B	89	9A	AB	ВС
0x0C-FC	C DD	EE	FF	0	x2C-F	CD	DE	EF	F0
0x10-31	A 2A	3A	4A	0	x30-3	ВА	0A	ВА	0Α
0x14-71	B 2B	3B	4B	0	x34-7	DB	0B	DB	0B
0x18-B	C 2C	3C	4C	0	x38-B	EC	0C	EC	0C
0x1C-F[1	C 2C	3C	4C	0	x3C-F	FC	0C	FC	0C

physical addresses	byt	es			physic address	al es	byt	es		
0x00-3			22	33	0x20-	-3[D0	D1	D2	D3
0x04-7	44	55	66	77	0x24-	-7[D4	D5	D6	D7
0x08-B	88	99	AA	ВВ	0x28-	-B[89	9A	ΑB	ВС
0x0C-F	CC	DD	EE	FF	0x2C-	- F[CD	DE	EF	F0
0x10-3	1A	2A	ЗА	4A	0x30-	-3[ВА	0A	ВА	0A
0x14-7	1В	2B	3B	4B	0x34-	-7[DB	0B	DB	0B
0x18-B	1C	2C	3C	4C	0x38-	-B[EC	0C	EC	0C
0x1C-F	1C	2C	3C	4C	0x3C-	-F[FC	0C	FC	0C

physical by addresses_	tes			a	physical ddresses	byt	es		
0×00-300	11	22	33	(9x20-3	D0	D1	D2	D3
$0 \times 04 - 744$	1 55	66	77	(9x24-7	D4	D5	D6	D7
0x08-B88	3 99	AA	ВВ	(9х28-В	89	9A	AB	ВС
0×0C-FC	C DD	EE	FF	(9x2C-F	CD	DE	EF	F0
0×10-3	\ 2A	ЗА	4A	(9x30-3	ВА	0A	ВА	0A
0×14-7 1	3 2B	3B	4B	(9x34-7	DB	0B	DB	0B
0x18-B	2C	3C	4C	(9х38-В	EC	0C	EC	0C
0x1C-F10	2C	3C	4C	(0x3C-F	FC	0C	FC	0C

physical bytes addresses
0x20-3D0 D1 D2 D3
0x24-7D4 D5 D6 D7
0x28-B89 9A AB B0
0x2C-FCD DE EF F
0x30-3BA 0A BA 0A
0x34-7DB 0B DB 0E
0x38-BEC 0C EC 00
0x3C-FFC 0C FC 00

physical addresses	byte	es		
0x00-3	00	11	22	33
0x04-7	44	55	66	77
0x08-B	88	99	AΑ	ВВ
0x0C-F	CC	DD	EE	FF
0x10-3	1A	2A	5A	4A
0x14-7	1В	2B	3B	4B
0x18-B	1C	2C	3C	4C
0x1C-F	1C	2C	3C	4C

physical addresses	byt	es		
0x20-3	D0	D1		
0x24-7	D4	D5	D6	D7
0x28-B	89	9Α	ΑB	ВС
0x2C-F	CD	DE	EF	F0
0x30-3	ВА	0A	ВА	0A
0x34-7	DB	0B	DB	0B
0x38-B	EC	0C	EC	0C
0x3C-F	FC	0C	FC	0C

physical addresses	byte	es			physica addresses	byt	es		
0x00-3	00	11	22	33	0x20-3			D2	D3
0x04-7	44	55	66	77	0x24-7	'D4	D5	D6	D7
0x08-B	88	99	AΑ	ВВ	0x28-E	89	9A	ΑB	ВС
0x0C-F	CC	DD	EE	FF	0x2C-F	CD	DE	EF	F0
0x10-3	1A	2A	ЗА	4A	0x30-3	ВА	0A	ВА	0A
0x14-7	1B	2B	3B	4B	0x34-7	'DB	0B	DB	0B
0x18-B	1C	2C	3C	4C	0x38-E	EC	0C	EC	0C
0x1C-F	1C	2C	3C	4C	0x3C-F	FC	0C	FC	0C

9-bit virtual addresses, 6-bit physical; 8 byte pages, 1 byte PTE page tables 1 page; PTE: 3 bit PPN (MSB), 1 valid bit, 4 unused page table base register 0x08; translate virtual address 0x00B

physical addresses	bytes	S			phy addr	/sical esses	byt	es		
0x00-3			22	33		20-3			D2	D3
0x04-7	44 5	55	66	77	0x2	4-7	D4	D5	D6	D7
0x08-B	88 9	99	AΑ	ВВ	0x2	8-B	89	9A	ΑB	ВС
0x0C-F	CC [DD	EE	FF	0x2	C-F	CD	DE	EF	F0
0x10-3	1A 2	2A	ЗА	4A	0x3	80-3	ВА	0A	ВА	0Α
0x14-7	1B 2	2B	3B	4B	0x3	4-7	DB	0B	DB	0B
0x18-B	1C 2	2C	3C	4C	0x3	8-B	EC	0C	EC	оC
0x1C-F	1C 2	2C	3C	4C	0x3	C-F	FC	0C	FC	0C

9-bit virtual addresses, 6-bit physical; 8 byte pages, 1 byte PTE page tables 1 page; PTE: 3 bit PPN (MSB), 1 valid bit, 4 unused page table base register 0x08; translate virtual address 0x00B

physical addresses	byt	es		
0x00-3	00	11	22	33
0x04-7	44	55	66	77
0x08-B	88	99	AΑ	ВВ
0x0C-F				
0x10-3	1A	2A	ЗА	4A
0x14-7	1В	2B	3B	4B
0x18-B	1C	2C	3C	4C
0x1C-F	1C	2C	3C	4C

physical addresses	byt	es		
0x20-3			D2	D3
0x24-7				
0x28-B	89	9A	AB	ВС
0x2C-F	CD	DE	EF	F0
0x30-3	ВА	0A	ВА	0A
0x34-7	DB	0B	DB	0B
0x38-B	EC	0C	EC	0C
0x3C-F	FC	0C	FC	0C

9-bit virtual addresses, 6-bit physical; 8 byte pages, 1 byte PTE page tables 1 page; PTE: 3 bit PPN (MSB), 1 valid bit, 4 unused page table base register 0x08; translate virtual address 0x1CB

physical addresses	byte	es			physic address	cal ses	byte	es		
0x00-3			22	33	0x20-				D2	D3
0x04-7	44	55	66	77	0x24-	-7	D4	D5	D6	D7
0x08-B	88	99	AΑ	ВВ	0x28-	-в	89	9A	ΑB	ВС
0x0C-F	CC	DD	EE	FF	0x2C-	-F	CD	DE	EF	F0
0x10-3	1A	2A	3A	4A	0x30-	-3	ВА	0Α	ВА	0Α
0x14-7	1B	2B	3B	4B	0x34-	-7	DB	0B	DB	0B
0x18-B	1C	2C	3C	4C	0x38-	-В	EC	0C	EC	0C
0x1C-F	1C	2C	3C	4C	0x3C-	-F	FC	0C	FC	0C

10-bit virtual addresses, 6-bit physical; 16 byte pages, 2 byte PTE

page tables 1 page; PTE 1st byte: (MSB) 2-bit PPN, valid bit; rest unused

physical addresses	byt	es		
0x00-3	00	11		
0x04-7	44	55	66	77
0x08-B				
0x0C-F				
0x10-3	1A	2A	ЗА	4A
0x14-7			3B	
0x18-B	1C	2C	3C	4C
0x1C-F	AC	ВС	DC	EC

```
physical addresses

0x20-3 D0 E1 D2 D3

0x24-7 D4 E5 D6 E7

0x28-B 89 9A AB BC

0x2C-F CD DE EF F0

0x30-3 BA 0A BA 0A

0x34-7 DB 0B DB 0B

0x38-B EC 0C EC 0C

0x3C-F FC 0C FC 0C
```

10-bit virtual addresses, 6-bit physical; 16 byte pages, 2 byte PTE

page tables 1 page; PTE 1st byte: (MSB) 2-bit PPN, valid bit; rest unused

physical addresses	byt	es		
0x00-3	00	11	22	33
0x04-7	44	55	66	77
0x08-B				
0x0C-F				
0x10-3	1A	2A	3A	4A
0x14-7	1В	2B	3B	4B
0x18-B				
0x1C-F	AC	ВС	DC	EC

```
physical addresses

0x20-3 D0 E1 D2 D3

0x24-7 D4 E5 D6 E7

0x28-B 89 9A AB BC

0x2C-F CD DE EF F0

0x30-3 BA 0A BA 0A

0x34-7 DB 0B DB 0B

0x38-B EC 0C EC 0C

0x3C-F FC 0C FC 0C
```

10-bit virtual addresses, 6-bit physical; 16 byte pages, 2 byte PTE

page tables 1 page; PTE 1st byte: (MSB) 2-bit PPN, valid bit; rest unused

physical addresses	byt	es		
0x00-3	00	11	22	33
0x04-7	44	55	66	77
0x08-B				
0x0C-F				
0x10-3	1A	2A	3A	4A
0x14-7	1В	2B	3B	4B
0x18-B				
0x1C-F	AC	ВС	DC	EC

```
physical addresses

0x20-3 D0 E1 D2 D3

0x24-7 D4 E5 D6 E7

0x28-B 89 9A AB BC

0x2C-F CD DE EF F0

0x30-3 BA 0A BA 0A

0x34-7 DB 0B DB 0B

0x38-B EC 0C EC 0C

0x3C-F FC 0C FC 0C
```

10-bit virtual addresses, 6-bit physical; 16 byte pages, 2 byte PTE

page tables 1 page; PTE 1st byte: (MSB) 2-bit PPN, valid bit; rest unused

physical addresses	bvt	es		
0x00-3	00	11	22	33
0x04-7	44	55	66	77
0x08-B	88	99	AΑ	ВВ
0x0C-F	CC	DD	EE	FF
0x10-3	1A	2A	3A	4A
0x14-7	1В	2B	3B	4B
0x18-B	1C	2C	3C	4C
0x1C-F	AC	BC	DC	EC

```
physical bytes addresses 0x20-3 D0 E1 D2 D3 0x24-7 D4 E5 D6 E7 0x28-B 89 9A AB BC 0x2C-F CD DE EF F0 0x30-3 BA 0A BA 0A 0x34-7 DB 0B DB 0B 0x38-B EC 0C EC 0C 0x3C-F FC 0C FC 0C
```

10-bit virtual addresses, 6-bit physical; 16 byte pages, 2 byte PTE

page tables 1 page; PTE 1st byte: (MSB) 2-bit PPN, valid bit; rest unused

physical addresses	byt	es		
0x00-3	00	11	22	33
0x04-7	44	55	66	77
0x08-B				
0x0C-F				
0x10-3	1A	2A	3A	4A
0x14-7	1В	2B	3B	4B
0x18-B				
0x1C-F	AC	ВС	DC	EC

```
physical addresses

0x20-3 D0 E1 D2 D3

0x24-7 D4 E5 D6 E7

0x28-B 89 9A AB BC

0x2C-F CD DE EF F0

0x30-3 BA 0A BA 0A

0x34-7 DB 0B DB 0B

0x38-B EC 0C EC 0C

0x3C-F FC 0C FC 0C
```

10-bit virtual addresses, 6-bit physical; 16 byte pages, 2 byte PTE

page tables 1 page; PTE 1st byte: (MSB) 2-bit PPN, valid bit; rest unused

physical addresses	byt	es			
0x00-3	00	11	22	33	
0x04-7	44	55	66	77	
0x08-B	88	99	AΑ	ВВ	
0x0C-F	CC	DD	EE	FF	
0x10-3	1A	2A	3A	4A	
0x14-7	1B	2B	3B	4B	
0x18-B	1C	2C	3C	4C	
0x1C-F	AC	ВС	DC	EC	

```
physical bytes addresses 0x20-3 D0 E1 D2 D3 0x24-7 D4 E5 D6 E7 0x28-B 89 9A AB BC 0x2C-F CD DE EF F0 0x30-3 BA 0A BA 0A 0x34-7 DB 0B DB 0B 0x38-B EC 0C EC 0C 0x3C-F FC 0C FC 0C
```

10-bit virtual addresses, 6-bit physical; 16 byte pages, 2 byte PTE

page tables 1 page; PTE 1st byte: (MSB) 2-bit PPN, valid bit; rest unused

physical addresses	byt	es		
0x00-3	00	11	22	33
0x04-7	44	55	66	77
0x08-B				
0x0C-F				
0x10-3	1A	2A	3A	4A
0x14-7	1В	2B	3B	4B
0x18-B				
0x1C-F	AC	ВС	DC	EC

```
physical addresses

0x20-3 D0 E1 D2 D3

0x24-7 D4 E5 D6 E7

0x28-B 89 9A AB BC

0x2C-F CD DE EF F0

0x30-3 BA 0A BA 0A

0x34-7 DB 0B DB 0B

0x38-B EC 0C EC 0C

0x3C-F FC 0C FC 0C
```

10-bit virtual addresses, 6-bit physical; 16 byte pages, 2 byte PTE

page tables 1 page; PTE 1st byte: (MSB) 2-bit PPN, valid bit; rest unused

physical addresses	byt	es		
0x00-3				
0x04-7	44	55	66	77
0x08-B				
0x0C-F	CC	DD	EE	FF
0x10-3	1A	2A	3A	4A
0x14-7				
0x18-B				
0x1C-F	ΑC	ВС	DC	EC

```
physical bytes addresses 0x20-3 D0 E1 D2 D3 0x24-7 D4 E5 D6 E7 0x28-B 89 9A AB BC 0x2C-F CD DE EF F0 0x30-3 BA 0A BA 0A 0x34-7 DB 0B DB 0B 0x38-B EC 0C EC 0C 0x3C-F FC 0C FC 0C
```

emacs.exe

Emacs (run by user mst3k)

Used by OS	
Stack	
Heap / other dynamic	
Writable data	
${\sf emacs.exe} \; \big({\sf Code} + {\sf Constants}\big)$	

emacs.exe

Emacs (run by user mst3k)

Used by OS Stack Heap / other dynamic Writable data emacs.exe (Code + Constants)

OS's memory

part of context switch is changing the page table

extra privileged instructions

part of context switch is changing the page table

extra privileged instructions

where in memory is the code that does this switching?

part of context switch is changing the page table extra privileged instructions

where in memory is the code that does this switching? probably have a page table entry pointing to it hopefully marked kernel-mode-only

part of context switch is changing the page table extra privileged instructions

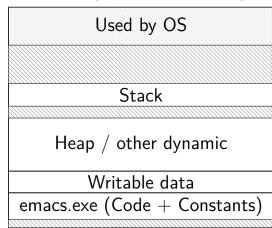
where in memory is the code that does this switching? probably have a page table entry pointing to it hopefully marked kernel-mode-only

code better not be modified by user program otherwise: uncontrolled way to "escape" user mode

emacs (two copies)

Emacs (run by user mst3k)

Used by OS Stack Heap / other dynamic Writable data emacs.exe (Code + Constants) Emacs (run by user xyz4w)



emacs (two copies)

Emacs (run by user mst3k)	Emacs (run by user xyz4w)
Used by OS	Used by OS
Stack	Stack
Heap / other dynamic	Heap / other dynamic
Writable data	Writable data
$emacs.exe\; (Code + Constants)$	emacs.exe (Code + Constants)

same data?

two copies of program

would like to only have one copy of program

what if mst3k's emacs tries to modify its code?

would break process abstraction:

"illusion of own memory"

permissions bits

```
page table entry will have more permissions bits can access in user mode? can read from? can write to? can execute from?
```

checked by MMU like valid bit

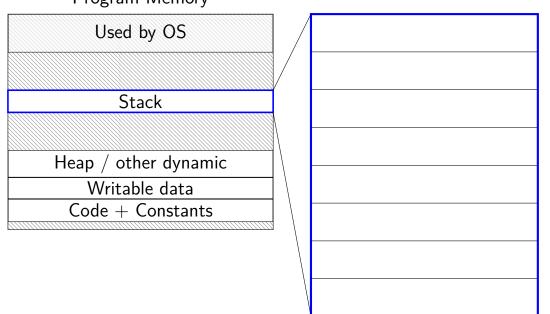
page table (logically)

virtual page #	valid?	user?	write?	exec?	physical page #
0000 0000	0	0	0	0	00 0000 0000
0000 0001	1	1	1	0	10 0010 0110
0000 0010	1	1	1	0	00 0000 1100
0000 0011	1	1	0	1	11 0000 0011

1111 1111[1	0	1	0	00 1110 1000

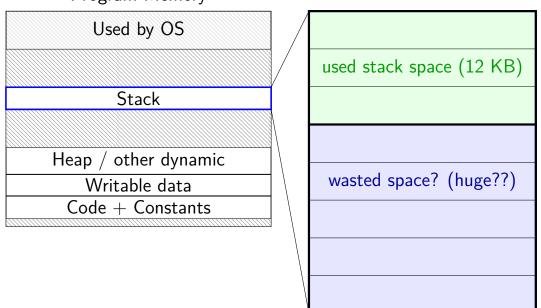
space on demand

Program Memory



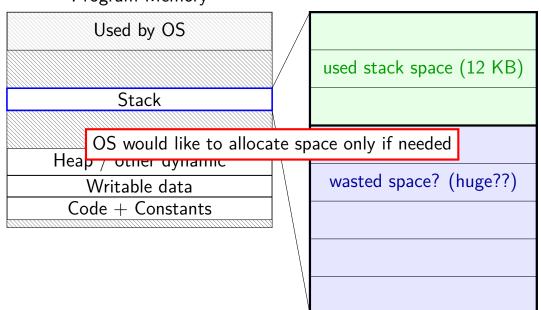
space on demand

Program Memory



space on demand

Program Memory



%rsp = 0x7FFFC000

```
...
// requires more stack space
A: pushq %rbx

B: movq 8(%rcx), %rbx
C: addq %rbx, %rax
...
```

VPN	valid?	page
		page
•••	•••	•••
0x7FFFB	0	
0x7FFFC	1	0x200DF
0x7FFFD	1	0x12340
0x7FFFE	1	0x12347
0x7FFFF	1	0x12345
•••	•••	•••

%rsp = 0x7FFC000

```
// requires more stack space
A: pushq %rbx
page fault!
B: movq 8(%rcx), %rbx
C: addq %rbx, %rax
...
```

VPN	valid?	physical page
VIIN	vallu!	page
•••	•••	•••
0x7FFFB	0	
0x7FFFC	1	0x200DF
0x7FFFD	1	0x12340
0x7FFFE	1	0x12347
0x7FFFF	1	0x12345
•••	•••	•••

pushq triggers exception hardware says "accessing address 0x7FFFBFF8" OS looks up what's should be there — "stack"

%rsp = 0x7FFC000

```
// requires more stack space
A: pushq %rbx restarted

B: movq 8(%rcx), %rbx
C: addq %rbx, %rax
...
```

VPN	valid?	physical page
VFIN	valiu!	page
•••	•••	•••
0x7FFFB	1	0x200D8
0x7FFFC	1	0x200DF
0x7FFFD	1	0x12340
0x7FFFE	1	0x12347
0x7FFFF	1	0x12345
•••	•••	•••

in exception handler, OS allocates more stack space OS updates the page table then returns to retry the instruction

note: the space doesn't have to be initially empty

only change: load from file, etc. instead of allocating empty page

loading program can be merely creating empty page table everything else can be handled in response to page faults no time/space spent loading/allocating unneeded space

mmap

```
Linux/Unix has a function to "map" a file to memory
int file = open("somefile.dat", O_RDWR);
    // data is region of memory that represents file
char *data = mmap(..., file, 0);
   // read byte 6 from somefile.dat
char seventh_char = data[6];
   // modifies byte 100 of somefile.dat
data[100] = 'x';
    // can continue to use 'data' like an array
```

swapping almost mmap

```
access mapped file for first time, read from disk (like swapping when memory was swapped out)
```

write "mapped" memory, write to disk eventually (like writeback policy in swapping) use "dirty" bit

extra detail: other processes should see changes all accesses to file use same physical memory

Linux maps: list of maps

```
$ cat /proc/self/maps
00400000-0040b000 r-xp 00000000 08:01 48328831
                                                         /bin/cat
0060a000-0060b000 r-p 0000a000 08:01 48328831
                                                         /bin/cat
0060b000-0060c000 rw-p 0000b000 08:01 48328831
                                                         /bin/cat
01974000-01995000 rw-p 00000000 00:00 0
                                                         [heap]
7f60c718b000-7f60c7490000 r-p 00000000 08:01 77483660
                                                         /usr/lib/locale/locale—archive
7f60c7490000-7f60c764e000 r-xp 00000000 08:01 96659129
                                                         /lib/x86_64-linux-gnu/libc-2.1
7f60c764e000-7f60c784e000 ----p 001be000 08:01 96659129
                                                         /lib/x86_64-linux-gnu/libc-2.1
7f60c784e000-7f60c7852000 r-p 001be000 08:01 96659129
                                                         /lib/x86_64-linux-gnu/libc-2.1
7f60c7852000-7f60c7854000 rw-p 001c2000 08:01 96659129
                                                         /lib/x86 64-linux-gnu/libc-2.1
7f60c7854000-7f60c7859000 rw-p 00000000 00:00 0
7f60c7859000-7f60c787c000 r-xp 00000000 08:01 96659109
                                                         /lib/x86_64-linux-gnu/ld-2.19.
7f60c7a39000-7f60c7a3b000 rw-p 00000000 00:00 0
7f60c7a7a000-7f60c7a7b000 rw-p 00000000 00:00 0
7f60c7a7b000-7f60c7a7c000 r-p 00022000 08:01 96659109
                                                         /lib/x86_64-linux-gnu/ld-2.19.
7f60c7a7c000-7f60c7a7d000 rw-p 00023000 08:01 96659109
                                                         /lib/x86_64-linux-gnu/ld-2.19.s
7f60c7a7d000-7f60c7a7e000 rw-p 00000000 00:00 0
7ffc5d2b2000-7ffc5d2d3000 rw-p 00000000 00:00 0
                                                         [stack]
7ffc5d3b0000-7ffc5d3b3000 r—p 00000000 00:00 0
                                                         [vvar]
7ffc5d3b3000-7ffc5d3b5000 r-xp 00000000 00:00 0
                                                         vdsol
fffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0
                                                         [vsyscall]
```

Linux maps: list of maps

```
$ cat /proc/self/maps
00400000-0040b000 r-xp 00000000 08:01 48328831
                                                        /bin/cat
0060a000-0060b000 r-p 0000a000 08:01 48328831
                                                         /bin/cat
0060b000-0060c000 rw-p 0000b000 08:01 48328831
                                                         /bin/cat
01974000 - 01995000 \text{ rw-p} 00000000 00:00 0
                                                         [heap]
7f60c718b000_7f60c7490000
                                                         <u>usr/lib/locale/lo</u>cale—archive
7f60c74900 OS tracks list of struct vm_area_struct with:
                                                                          gnu/libc-2.1
7f60c764e0
                                                                          gnu/libc-2.1
          (shown in this output):
7f60c784e0
                                                                          gnu/libc-2.1
7f60c78520
                                                                          gnu/libc-2.1
             virtual address start, end
7f60c78540
                                                                          gnu/ld-2.19.s
7f60c78590
             permissions
7f60c7a390
7f60c7a7a0
             offset in backing file (if any)
7f60c7a7b0
                                                                          gnu/ld-2.19.s
7f60c7a7c0
             pointer to backing file (if any)
                                                                          gnu/ld-2.19.s
7f60c7a7d0
7ffc5d2b20
7ffc5d3b00
           (not shown):
7ffc5d3b30
ffffffffff
             info about sharing of non-file data
```

page tricks generally

deliberately make program trigger page/protection fault

but don't assume page/protection fault is an error

have seperate data structures represent logically allocated memory e.g. "addresses 0x7FFF8000 to 0x7FFFFFFF are the stack"

page table is for the hardware and not the OS

hardware help for page table tricks

information about the address causing the fault
e.g. special register with memory address accessed
harder alternative: OS disassembles instruction, look at registers

(by default) rerun faulting instruction when returning from exception

precise exceptions: no side effects from faulting instruction or after e.g. pushq that caused did not change %rsp before fault e.g. can't notice if instructions were executed in parallel

swapping

early motivation for virtual memory: swapping

using disk (or SSD, ...) as the next level of the memory hierarchy how our textbook and many other sources presents virtual memory

OS allocates program space on disk own mapping of virtual addresses to location on disk

DRAM is a cache for disk

swapping

early motivation for virtual memory: swapping

using disk (or SSD, ...) as the next level of the memory hierarchy how our textbook and many other sources presents virtual memory

OS allocates program space on disk own mapping of virtual addresses to location on disk

DRAM is a cache for disk

swapping components

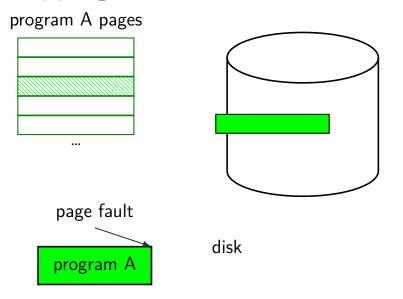
```
"swap in" a page — exactly like allocating on demand!
    OS gets page fault — invalid in page table
    check where page actually is (from virtual address)
    read from disk
    eventually restart process
"swap out" a page
    OS marks as invalid in the page table(s)
    copy to disk (if modified)
```

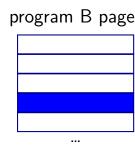
HDD reads and writes: milliseconds to tens of milliseconds minimum size: 512 bytes writing tens of kilobytes basically as fast as writing 512 bytes

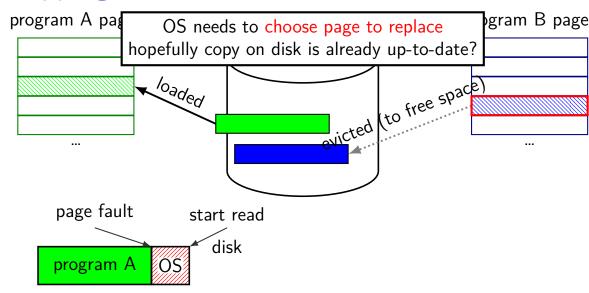
HDD reads and writes: milliseconds to tens of milliseconds minimum size: 512 bytes writing tens of kilobytes basically as fast as writing 512 bytes

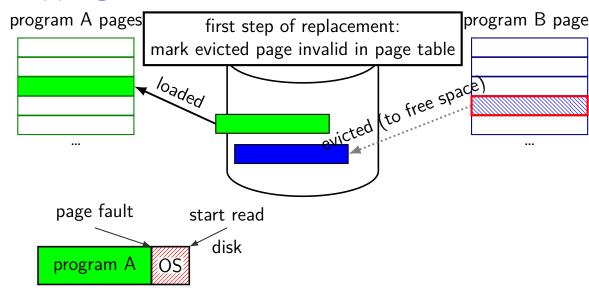
HDD reads and writes: milliseconds to tens of milliseconds minimum size: 512 bytes writing tens of kilobytes basically as fast as writing 512 bytes

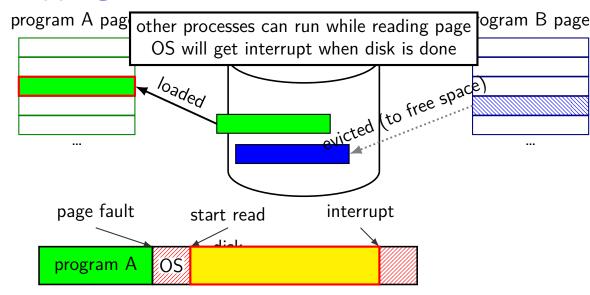
HDD reads and writes: milliseconds to tens of milliseconds minimum size: 512 bytes writing tens of kilobytes basically as fast as writing 512 bytes

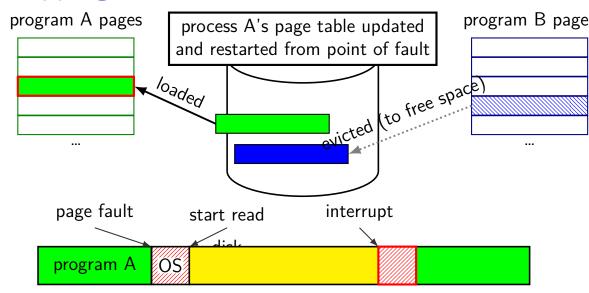












do we really need a complete copy?

bash	new copy of bash			
Used by OS	Used by OS			
Stack	Stack			
Heap / other dynamic	Heap / other dynamic			
Writable data	Writable data			
Code + Constants	Code + Constants			

do we really need a complete copy?

bash	new copy of bash			
Used by OS	Used by OS			
Stack	Stack			
Heap / other dynamic	Heap / other dynamic			
Writable data	Writable data			
Code + Constants	Code + Constants			

shared as read-only

do we really need a complete copy?

bash	new copy of bash			
Used by OS	Used by OS			
Stack	Stack			
Stack	Stack			
Heap / other dynamic	Heap / other dynamic			
Writable data	Writable data			
Code + Constants can't be	shared? Code + Constants			

trick for extra sharing

```
sharing writeable data is fine — until either process modifies it example: default value of global variables might typically not change (or OS might have preloaded executable's data anyways)
```

can we detect modifications?

trick for extra sharing

sharing writeable data is fine — until either process modifies it example: default value of global variables might typically not change (or OS might have preloaded executable's data anyways)

can we detect modifications?

trick: tell CPU (via page table) shared part is read-only processor will trigger a fault when it's written

VPN

valid? write?

•••

0x00601 0x00602 0x00603 0x00604 0x00605

Page				
•••	•••	•••		
1	1	0x12345		
1	1	0x12347		
1	1	0x12340		
1	1	0x200DF		
1	1	0x200AF		
•••	•••	•••		

VPN
•••
0x00601
0x00602
0x00603
0x00604
0x00605
•••

valid? write? page						
•••	••• •••					
1	0	0x12345				
1	0	0x12347				
1	0	0x12340				
1	0	0x200DF				
1	0	0x200AF				
•••	•••	•••				

•••
0x00601
0x00602
0x00603
0x00604
0x00605

VPN

valid? write?		ph	ysical	
vana.	********	pa	ge	

•••	•••	•••
1	0	0x12345
1		0x12347
1	0	0x12340
1		0x200DF
1	0	0x200AF
•••	•••	•••

copy operation actually duplicates page table both processes share all physical pages but marks pages in both copies as read-only

VPN	valid?	write	physical page	VPN	valid?	write	? physical page
•••	•••	•••	page 		•••		page I
0x00601	1	0	0x12345	0x00601	1	0	0x1234
0x00601	1	0	0x12343	0x00601 0x00602	1	0	0x1234
0x00602	1	0	0x12347	0x00602	1	0	0x1234
0x00603	1	0	0x12340 0x200DF	0x00603	1	0	0x1234
0x00605	1	0	0x200AF	0x00605	1	0	0x200A
						•••	
	1						

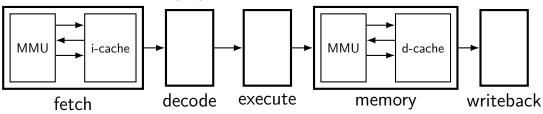
when either process tries to write read-only page triggers a fault — OS actually copies the page

VPN	valid?	writa'	physical page	VPN	valid?	writa	physical page
VIIV	valiu:	VVIIC	page	VIIV	valiu:	WITEC	page
•••	•••	•••	•••	•••	•••	•••	•••
0x00601	1	0	0x12345	0x00601	1	0	0x12345
0x00602	1	0	0x12347	0x00602	1	0	0x12347
0x00603	1	0	0x12340	0x00603	1	0	0x12340
0x00604	1	0	0x200DF	0x00604	1	0	0x200DF
0x00605	1	0	0x200AF	0x00605	1	1	0x300FD
•••	•••	•••	•••	•••	•••	•••	•••

after allocating a copy, OS reruns the write instruction

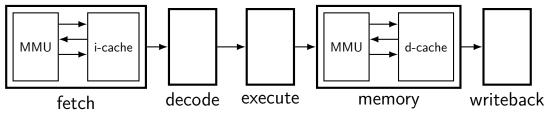
backup slides

MMUs in the pipeline



up to four memory accesses per instruction

MMUs in the pipeline



up to four memory accesses per instruction challenging to make this fast (topic for a future date)