last time

shell: redirection, pipelines

file descriptors

OS tracks array pointers to open files per process file descriptor = index into array dup2: copy pointer; close: NULL-out pointer

pipes:

two ends with separate file descriptors: write and read if too much data written and not read, writes will wait for reads reading from read end waits for some data to be available (or no write ends open)

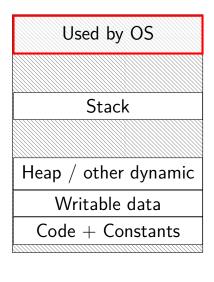
program memory

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

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

0x0000 0000 0040 0000

program memory

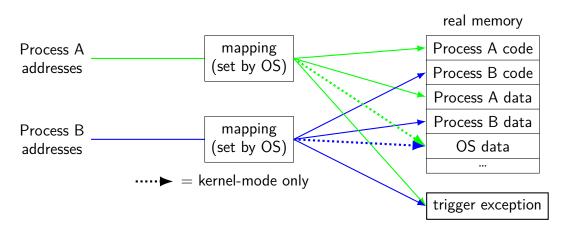


0x7F...

0x0000 0000 0040 0000

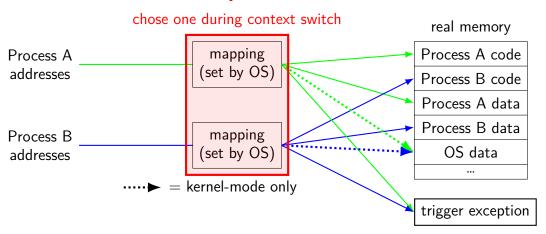
address spaces

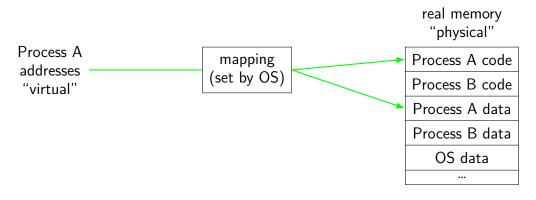
illuision of dedicated memory

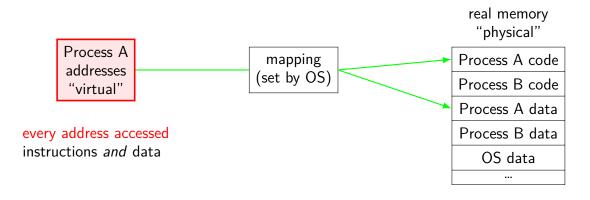


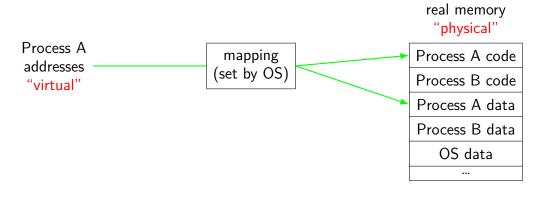
address spaces

illuision of dedicated memory

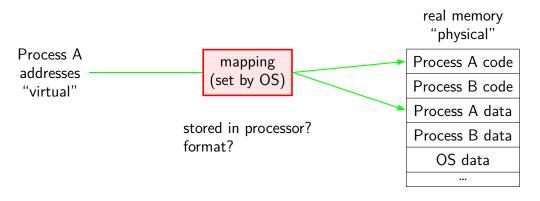


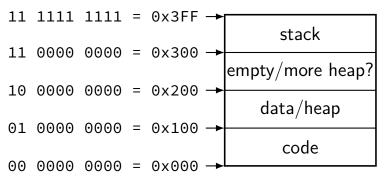


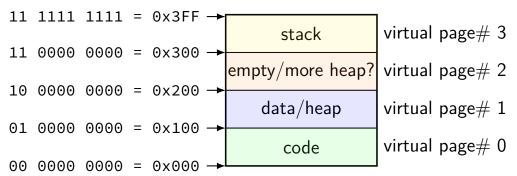


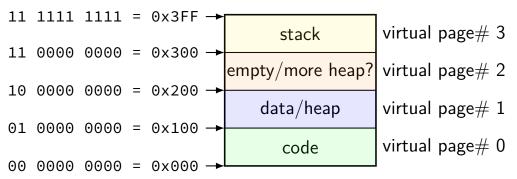


program addresses are 'virtual' real addresses are 'physical' can be different sizes!

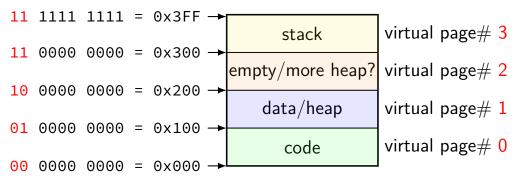




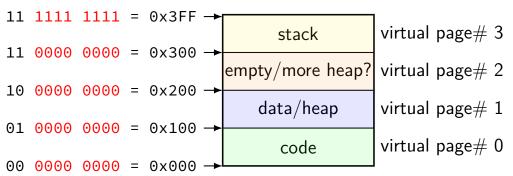




divide memory into pages (2^8 bytes in this case) "virtual" = addresses the program sees



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

	111	0000	0000	to
	111	1111	1111	
(901	0000	0000	to
(901	1111	1111	
(900	0000	0000	to
(900	1111	1111	

toy physical memory

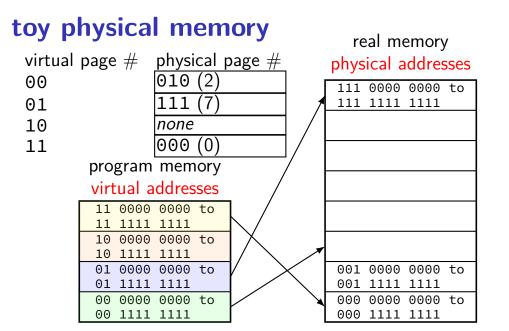
program memory virtual addresses

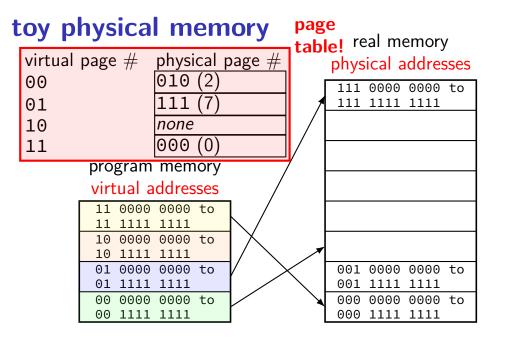
11	0000	0000	to
11	1111	1111	
10	0000	0000	to
10	1111	1111	
01	0000	0000	to
01	1111	1111	
00	0000	0000	to
00	1111	1111	

real memory physical addresses

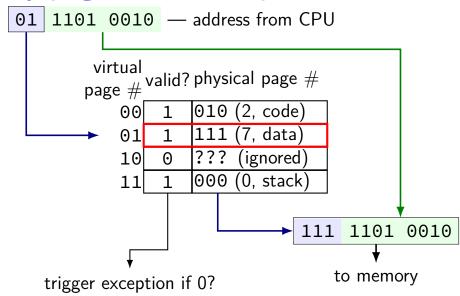
111	0000	0000	to	physical page 7
111	1111	1111		physical page i
001	0000	0000	to	
001	1111	1111	CO	physical page 1
000	0000	0000	to	
000	1111	1111	CO	physical page 0

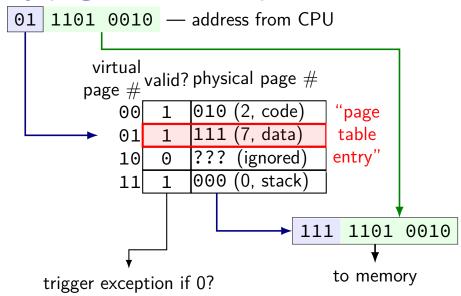
toy physical memory real memory physical addresses 111 0000 0000 to 111 1111 1111 program memory virtual addresses 11 0000 0000 to 1111 1111 10 0000 0000 to 1111 1111 01 0000 0000 to 0000 0000 to 1111 1111 001 1111 1111 0000 0000 to 000 0000 0000 1111 1111 000 1111 1111



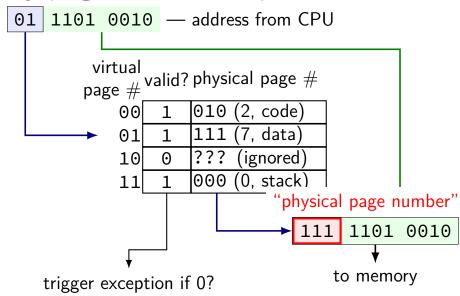


```
virtual page # valid? physical page # 00 1 010 (2, code) 01 1 111 (7, data) 10 0 ??? (ignored) 11 1 000 (0, stack)
```

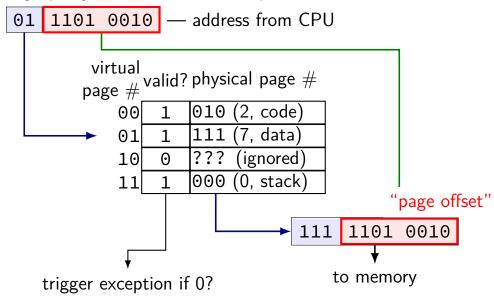




t "virtual page number" | ookup 1101 0010 — address from CPU virtual page # valid? physical page #010 (2, code) 00 (7, data) 01 10 (ignored) 000 (0, stack) 11 1101 0010 to memory trigger exception if 0?



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:

exercise: page table lookup

suppose 64-byte pages (= 6-bit page offsets), 9-bit virtual addresses

valid	PPN
1	0010
1	1010
0	
0	
1	1110
1	0100
1	0001
0	
	1 0 0 1 1

virtual address 0×024 (0 0010 0100) = physical address ???

vim (two copies)

Vim (run by user mst3k)

	Used by OS
	Stack
Не	ap / other dynamic
	Writable data
vim	(Code + Constants)
vim	(Code + Constants)

Vim (run by user xyz4w)

					,
	Use	ed by	OS		
		Stack	(
Не	eap / d	other	dyna	amic	
	Writ	able	data		
vim	(Code	e + (Const	ants	5)

vim (two copies)

Vim (run by user mst3k)	Vim (run by user xyz4w)
Used by OS	Used by OS
Stack	Stack
Heap / other dynamic	Heap / other dynamic
Writable data	Writable data
$vim\;(Code + Constants)$	$vim\;(Code + Constants)$

same data?

two copies of program

would like to only have one copy of program

what if mst3k's vim 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 hardware 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

running a program

Some program

U	Ised by OS
	Stack
Heap ,	other dynamic
W	ritable data
Code	e + Constants

running a program

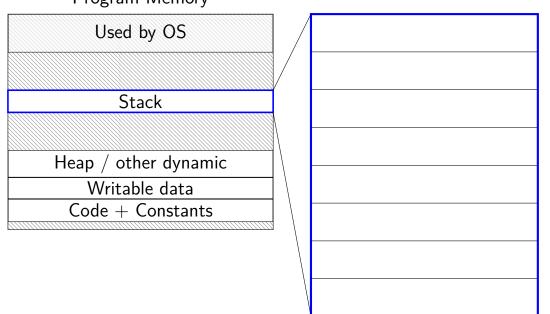
Some program

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

OS's memory

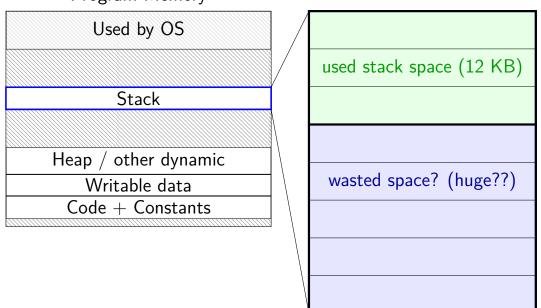
space on demand

Program Memory



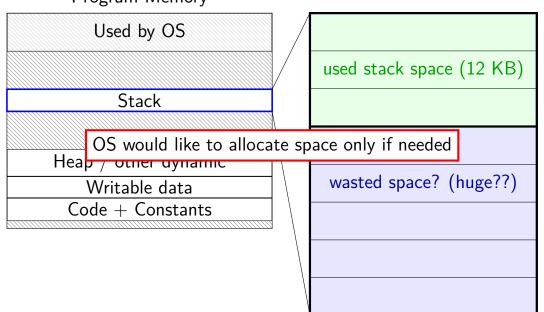
space on demand

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?	physical
VIIN	valiu:	page
•••	•••	•••
0x7FFFB	0	
0x7FFFC	1	0x200DF
0x7FFFD	1	0x12340
0x7FFFE	1	0x12347
0x7FFFF	1	0x12345
•••	•••	•••

%rsp = 0x7FFFC000

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

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

pushq triggers exception hardware says "accessing address 0x7FFBFF8" OS looks up what's should be there — "stack"

%rsp = 0x7FFFC000

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

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

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

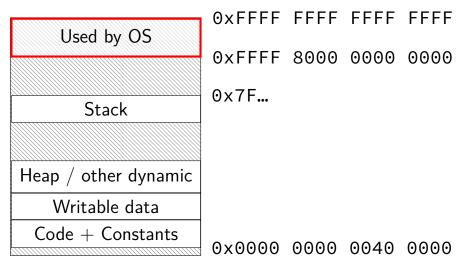
in exception handler, OS allocates more stack space OS updates the page table then returns to retry the instruction

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

program memory



system calls, I/O events, etc. run OS code in kernel mode

system calls, I/O events, etc. run OS code in kernel mode

where in memory is this OS code?

system calls, I/O events, etc. run OS code in kernel mode

where in memory is this OS code?

probably have a page table entry pointing to it marked not accessible in user mode

system calls, I/O events, etc. run OS code in kernel mode

where in memory is this OS code?

probably have a page table entry pointing to it marked not accessible in user mode

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

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
```

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
```

exercise setup

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

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	

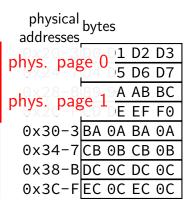
physical addresses	byt	es		
0x20-3	D0	D1	D2	D3
0x24-7				
0x28-B	89	9A	ΑB	ВС
0x2C-F				
0x30-3	ВА	0A	ВА	0Α
0x34-7				
0x38-B	DC	0C	DC	0C
0x3C-F	EC	0C	EC	0C

exercise setup

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

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

physical addresses	byt	es		
0x00-3	00	11	22	33
0x04-7				
0x08-B	88	99	ΑА	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 = ???
```

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

physical addresses	bytes
0x00-3	00 11 22 33
0x04-7	44 55 66 77
0x08-B	88 99 AA BB
	CC DD EE FF
0x10-3	1A 2A 3A 4A
0x14-7	1B 2B 3B 4B
0x18-B	
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	0Α
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 = ???
```

physical addresses	bytes
0x00-3	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	10.20.30.40

physical addresses	byt	es		
0x20-3	D0	D1	D2	D3
0x24-7			D6	
0x28-B	89	9A	ΑB	ВС
0x2C-F	CD	DE	EF	F0
0x30-3	ВА	0A	ВА	0A
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 1000

physical bytes
addresses
0x00-3 00 11 22 33
0x04-7 44 55 66 77
0x08-B88 99 AA BB
0x0C-FCC DD EE FF
0x10-3 1A 2A 3A 4A
0x14-7 1B 2B 3B 4B
0x18-B 1C 2C 3C 4C
0x1C-F1C 2C 3C 4C

physical addresses	byt	es		
0x20-3	D0	D1		
0x24-7	D4	D5	D6	D7
0x28-B				
0x2C-F	CD	DE	EF	F0
0x30-3	ВА	0A	ВА	0A
0x34-7	СВ	0B	СВ	0B
0x38-B				
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 page # valid? physical page # 00 1 010 011 111 10 0 000 11 1 1 1000

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

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?

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

copy-on-write and page tables

VPN

0x00601 0x00602 0x00603 0x00604 0x00605 valid? write?

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

copy-on-write and page tables

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

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

V 1 1 V
•••
0x00601
0x00602
0x00603
0x00604
0x00605
•••

VPN.

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

copy-on-write and page tables

nhygical

VPN	valid? write? page			
•••	•••	•••	•••	
0x00601	1	0	0x12345	
0x00602	1	0	0x12347	
0x00603	1	0	0x12340	
0x00604	1	0	0x200DF	
0x00605	1	0	0x200AF	
•••	•••	•••	•••	

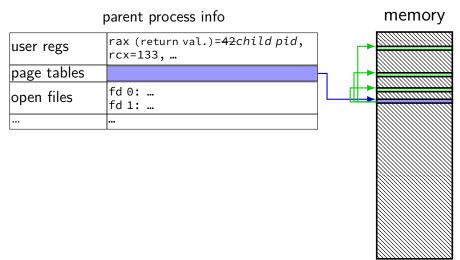
VPN	valid?	write	pnysicai
•••	•••	•••	
0x00601	1	0	0x12345
0x00602	1	0	0x12347
0x00603	1	0	0x12340
0x00604	1	0	0x200DF
0x00605	1	0	0x200AF
•••	•••	•••	•••

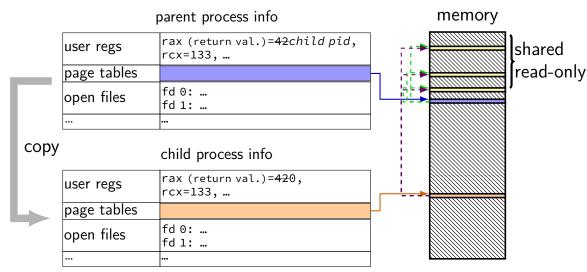
when either process tries to write read-only page triggers a fault — OS actually copies the page

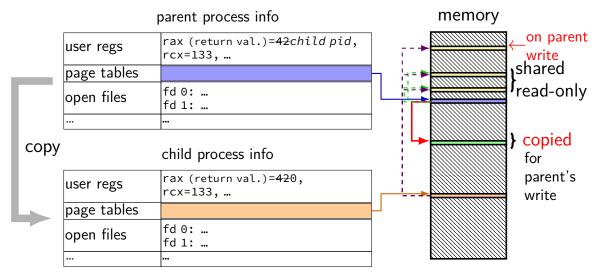
copy-on-write and page tables

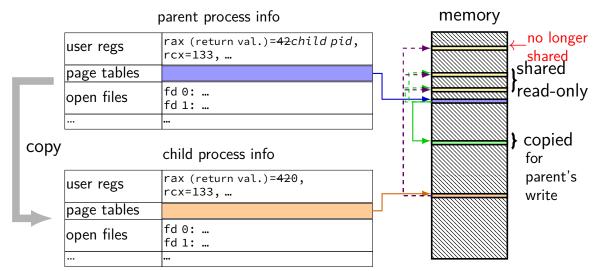
VPN	valid?	writo	physical Page	VPN	valid?	writo	physical page
VIIN	vallu:	WIILE	[:] page	VIIN	valiu: write:		page
•••	•••	•••	•••	•••	•••	•••	•••
0x00601	1	0	0x12345	0x00601	1	0	0x12345
0x00602	1	0	0x12347	0x00602	1	0	0x12347
0x00603	1	0	0x12340	0x00603	1	0	0x12340
0x00604	1	0	0x200DF	<u>0x00604</u>	1	0	0x200DF
0x00605	1	0	0x200AF	0x00605	1	1	0x300FD
•••	•••	•••	•••	•••	•••	•••	•••

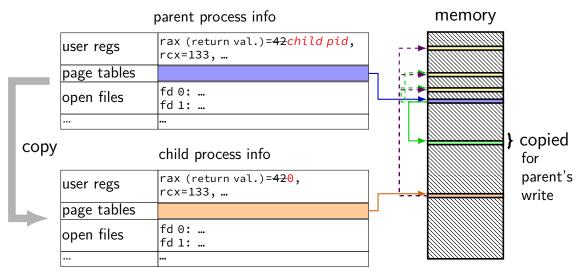
after allocating a copy, OS reruns the write instruction











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

example page table tricks

allocating space on demand

loading code/data from files on disk on demand

saving data temporarily to disk, reloading to memory on demand "swapping"

stopping in a debugger when a variable is modified detecting whether memory was read/written recently sharing memory between programs on two different machines "copy-on-write" (later)

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

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

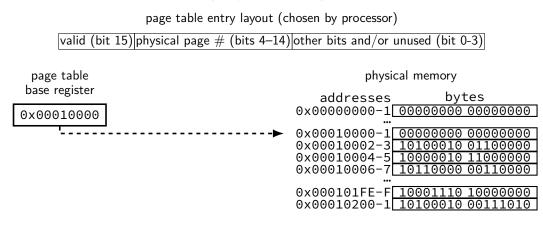
page table entry layout (chosen by processor)

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



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) physical memory page table base register addresses bytes 0x00000000-1 00000000 00000000 0x00010000 $0 \times 00010002 - 3 10100010$ $0 \times 00010004 - 5\Gamma$ 10000010 0x00010006-7 10110000 0x000101FE-F 10001110 0x00010200-1 10100010 0011101

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) physical memory page table base register addresses bytes 0x00000000-1 00000000 00000000 0x00010000 0x00010000-1 00000000 $0 \times 00010002 - 3 10100010$ $0 \times 00010004 - 5\Gamma$ 0x00010006-7 0x000101FE-F 10001110 0x00010200-1 10100010 0011101

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 0011101

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 $0 \times 00010002 - 3$ 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

page table entry layout (chosen by processor)

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

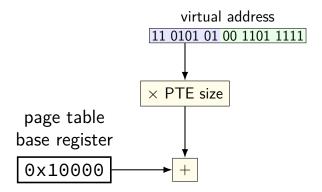
page table entry layout (chosen by processor) valid (bit 15) physical page # (bits 4–14) other bits and/or unused (bit 0-3) page table physical memory base register addresses bytes 0x0000000-1 00000000 00000000 0x00010000 0x00010000-1 00000000 $0 \times 00010002 - 3$ page table (logically) 0x00010004-5 0x00010006-7 virtual page # valid? physical page # 0000 0000 $0 \times 000101 FE - F 10001$ 0000 0001 0x00010200-1 10100010 001 0000 0010 0000 0011 0000 001 1111 1111 1110 1000

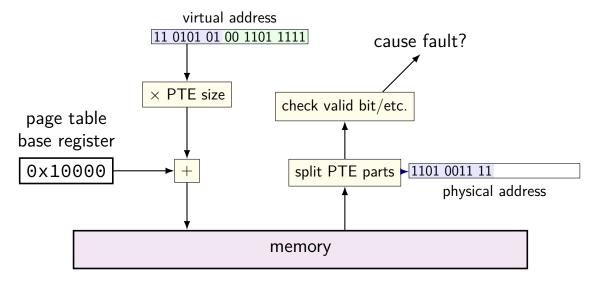
where can processor store megabytes of page tables? in memory

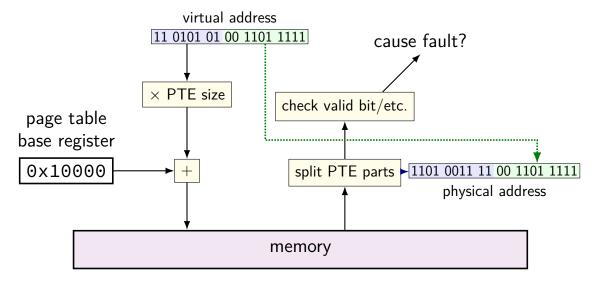
page table entry layout (chosen by processor) valid (bit 15) physical page # (bits 4–14) other bits and/or unused (bit 0-3) page table physical memory base register addresses bytes 0x0000000-1 00000000 00000000 0x00010000 0x00010000-1 00000000 000000000 $0 \times 00010002 - 3$ page table (logically) 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

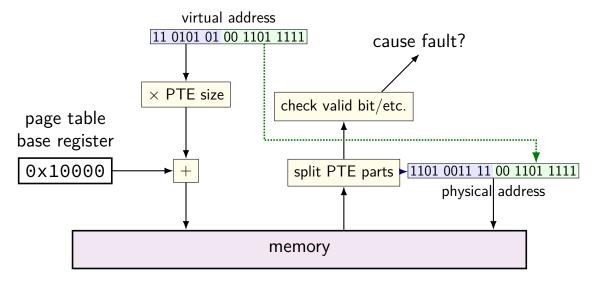
virtual address

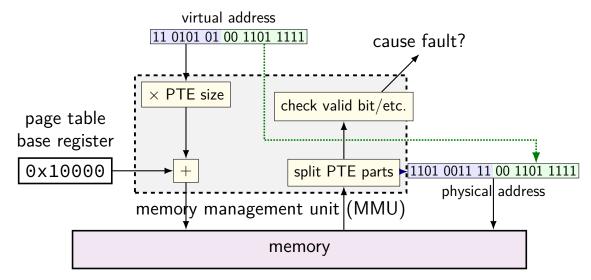
11 0101 01 00 1101 1111

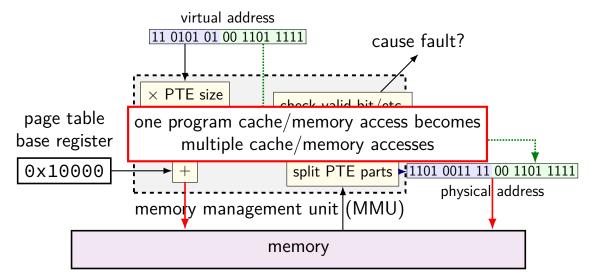


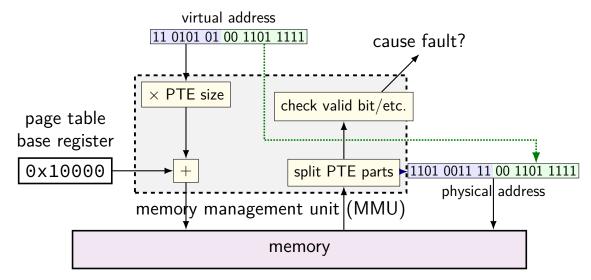












6-bit virtual addresses, 6-bit physical; 8 byte pages, 1 byte PTE page tables 1 page; PTE: 3 bit PPN (MSB), 1 valid bit, 4 other; page table base register 0x20; translate virtual address 0x31

physical addresses	byt	es		
0x00-3			22	33
0x04-7	44	55	66	77
0x08-B	88	99	AA	ВВ
0x0C-F	CC	DD	ΕE	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 ddresses	byt	es		
0x20-3	D0	D1	D2	D3
0x24-7	E4	E5	F6	07
0x28-B	89	9A	ΑB	ВС
0x2C-F	CD	DE	EF	F0
0x30-3	ВА	0A	ВА	0A
0x34-7	СВ	0B	СВ	0B
0x38-B	DC	0C	DC	0C
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;

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 + 110 \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-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;

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 + 110 \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
```

37

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 + 110 \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;

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

```
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-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-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-F|EC 0C EC 0C
```

```
0x31 = 11 0001
PTE \ addr:
0x20 + 110 \times 1 = 0x26
PTE \ value:
0xF6 = 1111 0110
PPN \ 111, \ valid \ 1
M[111 \ 001] = M[0x39]
\rightarrow 0x0C
```

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	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	1В	2B	3B	4B	
0x18-B	1C	2C	3C	4C	
0x1C-F	1 <u>C</u>	2C	3C	4C	

physical addresses	byt	es		
0x20-3	Α0	E2	D1	F3
0x24-7	E4	E5	F6	07
0x28-B	89	9A	AB	ВС
0x2C-F	CD	DE	EF	F0
0x30-3	ВА	0A	ВА	0A
0x34-7	СВ	0B	СВ	0B
0x38-B	DC	0C	DC	0C
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 page table base register 0x20; translate virtual address 0x12

addresses bytes
0x00-3 00 11 22 33
0x04-744 55 66 77
0x08-B88 99 AA BB
0x0C-FCC DD EE FF
0×10-3 1A 2A 3A 4A
0x14-7 1B 2B 3B 4B
0x18-B1C 2C 3C 4C
0x1C-F1C 2C 3C 4C

physical.

```
physical bytes
addresses
0x20-3|A0 E2 D1 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
```

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 bytes
addresses
0x00-300 11 22 33
0x04-744 55 66 77
0x08-B88 99 AA BB
0x0C-FCC DD EE FF
0x10-3 1A 2A 3A 4A
0x14-7 1B 2B 3B 4B
0x18-B1C 2C 3C 4C
0x1C-F1C 2C 3C 4C

physical addresses	byt	es			
0x20-3	_				
0x24-7	E4	E5	F6	07	
0x28-B	89	9A	AB	ВС	
0x2C-F	CD	DE	EF	F0	
0x30-3	ВА	0Α	ВА	0A	
0x34-7	СВ	0B	СВ	0B	
0x38-B	DC	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
```

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

bytes
addresses
0x00-300 11 22 33
0x04-744 55 66 77
0x08-B88 99 AA BB
0x0C-FCC DD EE FF
0x10-3 1A 2A 3A 4A
0x14-7 1B 2B 3B 4B
0x18-B1C 2C 3C 4C
0x1C-F1C 2C 3C 4C

nhysical.

```
physical bytes
addresses
0x20-3 A0 E2 D1 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
```

1-level exercise (2)

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

pnysical bytes					
addresses					
0x00-300 11 22 33					
0x04-744 55 66 77					
0x08-B88 99 AA BB					
0x0C-FCC DD EE FF					
0x10-3 1A 2A 3A 4A					
0x14-7 1B 2B 3B 4B					
0x18-B1C 2C 3C 4C					
0x1C-F1C 2C 3C 4C					

physical

```
physical bytes
addresses
0x20-3|A0 E2 D1 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
```

 \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:

	0	(
ptbr = GetPointerToTable(1	Ţ:
prof = Gerrolliter to rable(2	(
	3	Ţ.
		\neg

 VPN valid? physical

 0
 0

 1
 1

 2
 0

 3
 1

 0x3333

 ...
 ...

```
\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:

	ľ
${\sf ptbr} = {\sf GetPointerToTable}($	
	1

VPIN	valiu!	priysicai	
0	0		
1	1	0×9999	١
2	0)
3	1	0x3333	
	0 1 2 3	0 0 1 1 2 0 3 1 	2 0 —

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}
```

page_allocate()

page_allocate()

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

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

VLII	valiu!	priyar	Cai	
0	0			
1	1	(new))	١
2	0		,)
3	1			
	0 1 2	0 0 1 1 2 0	0 0 — 1 1 (new) 2 0 —	0 0 — 1 1 (new) 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 ...
```

 $ptbr\ \textit{now} == GetPointerToTable($

0	0		
1	1	(new)	\
2	0	— ↑] /
3	1	_	

VPN valid? 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
```

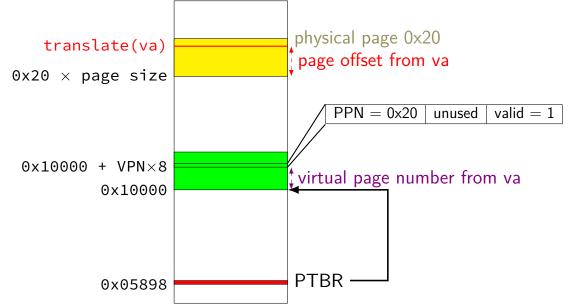
address/page table entry format

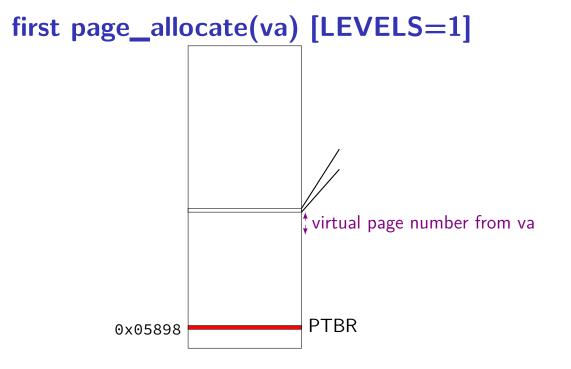
(with POBITS=12, LEVELS=1)

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

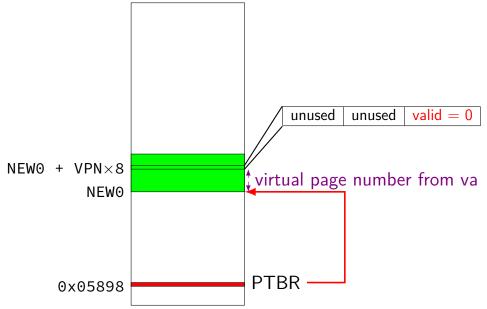
in assignment: value from posix_memalign = physical address

pa = translate(va) [LEVELS=1]

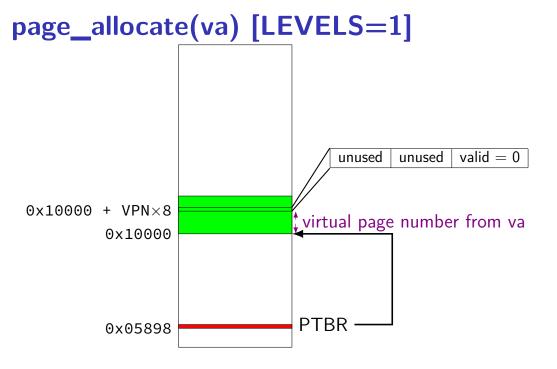


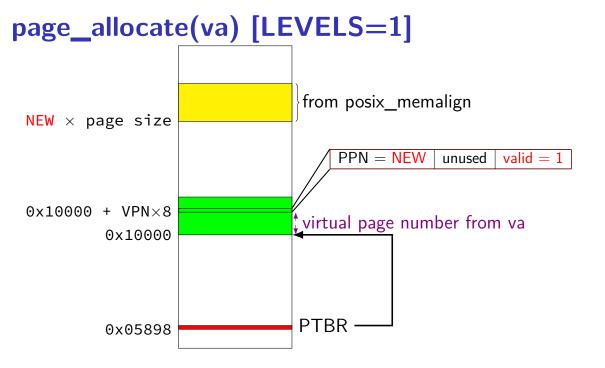


first page_allocate(va) [LEVELS=1]

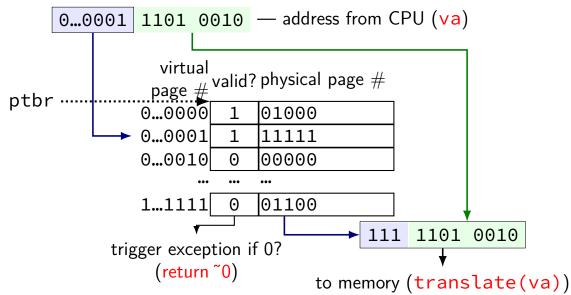


first page_allocate(va) [LEVELS=1] from posix_memalign $NEW1 \times page size$ PPN = NEW1valid = 1unused NEW0 + VPN×8 ‡virtual page number from va NEW₀ **PTBR** 0x05898

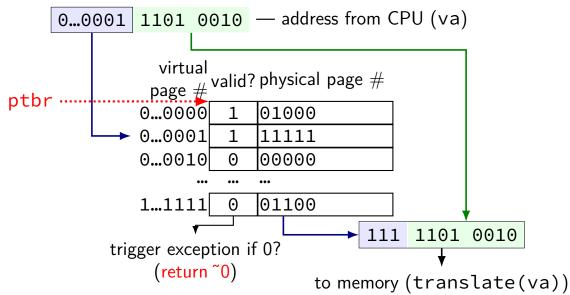




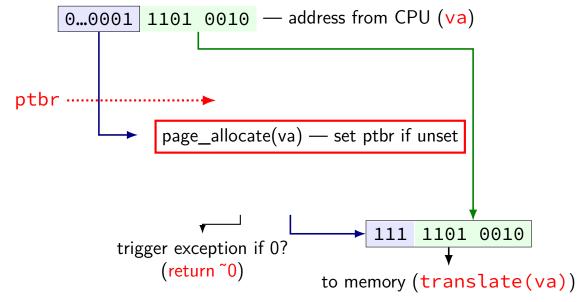
page table lookup (and translate())



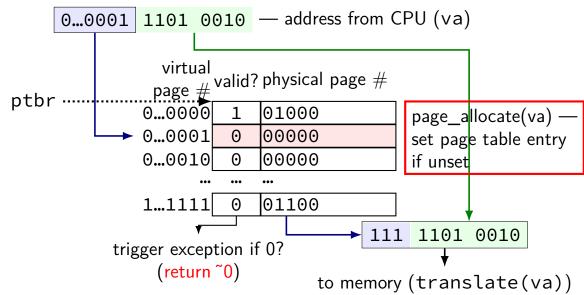
page table lookup (and translate())



page table lookup (and allocate)



page table lookup (and allocate)



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??)

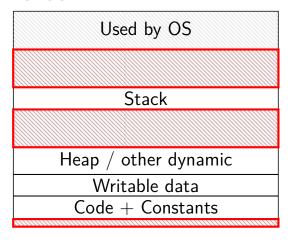
huge page tables

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)

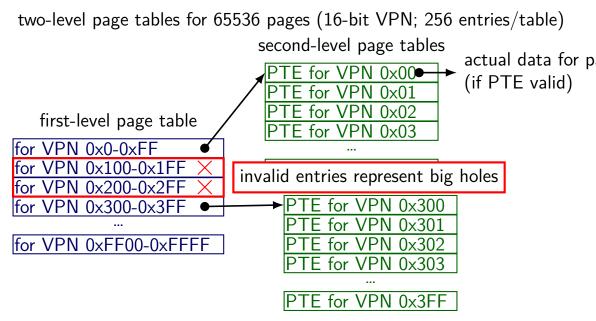
lookup should not involve many memory accesses

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 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

57

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 •••

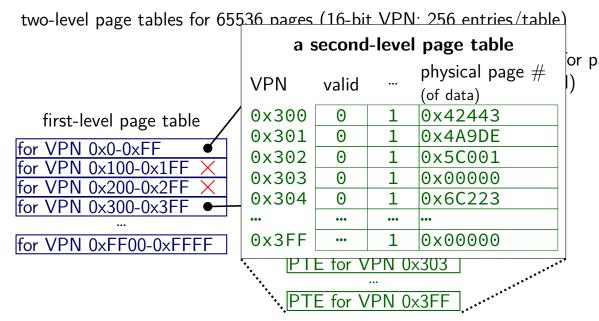
TE for VPN 0x3FF

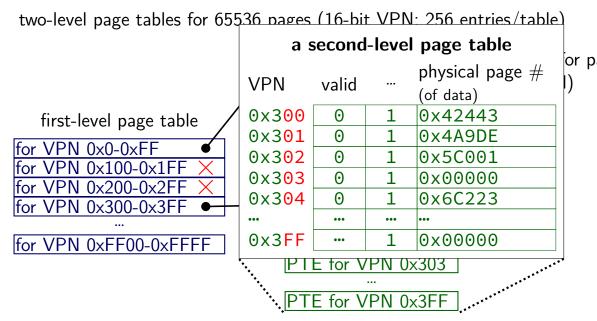
57

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 $0 \times FF00 - 0 \times FFFF$ •••

TE for VPN 0x3FF

57





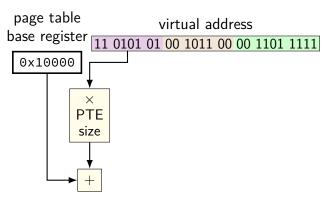
two-level page tables for 65536 pages (16-bit VPN; 256 entries/table) second-level page tables actual data for p for VPN 0x00 (if PTE valid) first-level page table for VPN $0 \times 0 - 0 \times FF$ tor VPN $0 \times 100 - 0 \times 1$ FF IPTE for VPN 0xFF VPN 0x200-0x2FF for VPN 0x300-0x3FF VPN 0x300 for VPN 0xFF00-0xFFFF VPN 0x302 TE for VPN 0x303 for VPN 0x3FF

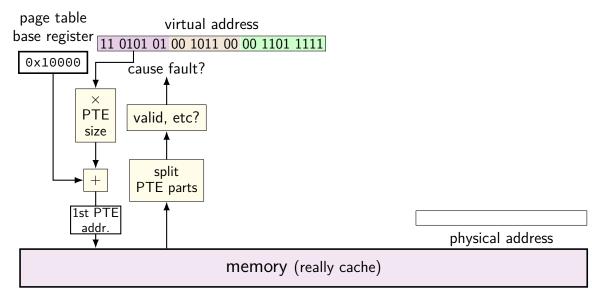
virtual address

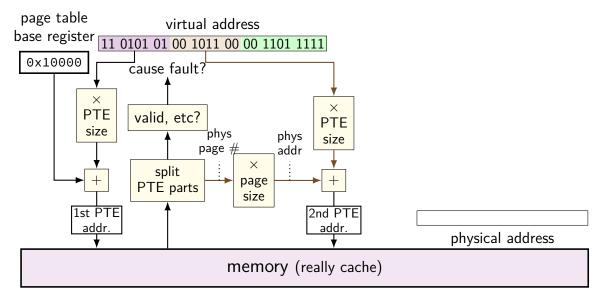
11 0101 01 00 1011 00 00 1101 1111

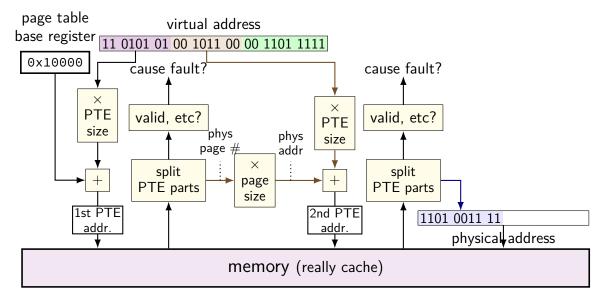
VPN — split into two parts (one per level)

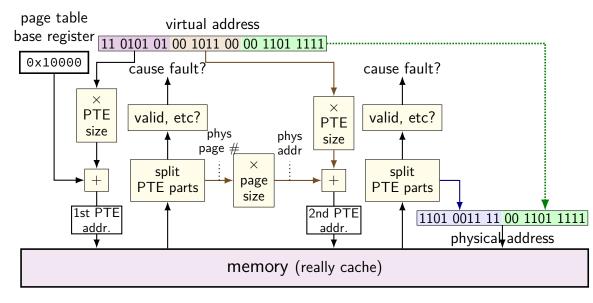
this example: parts equal sized — common, but not required

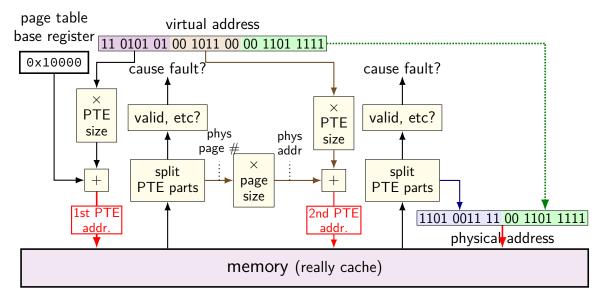


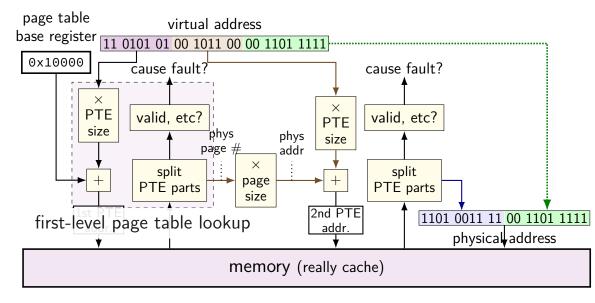


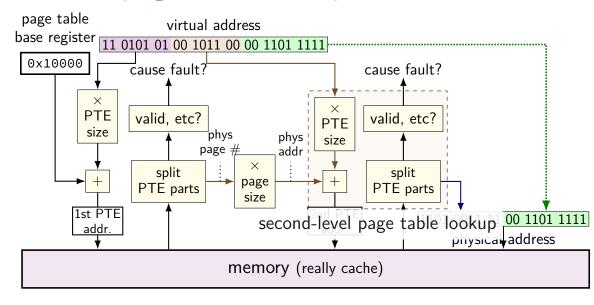


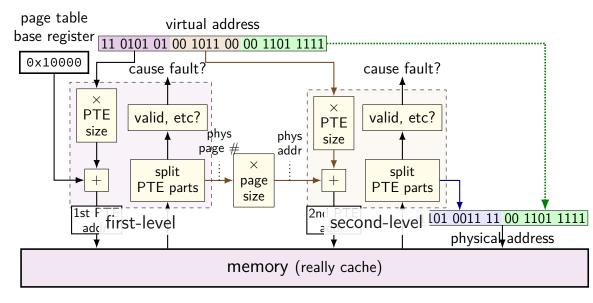


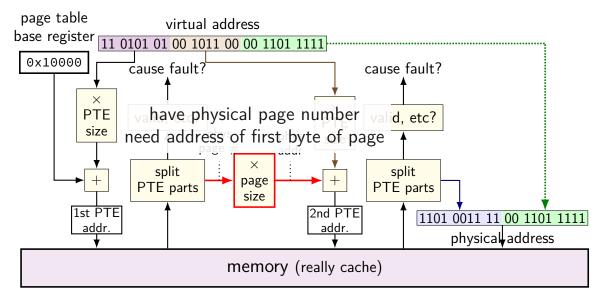


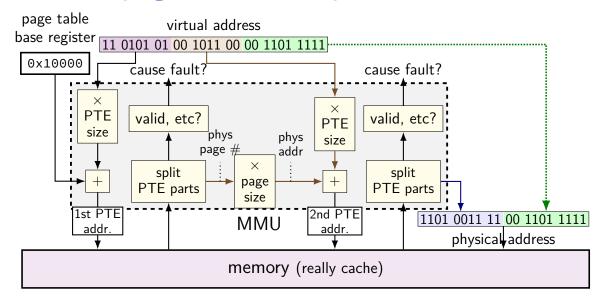




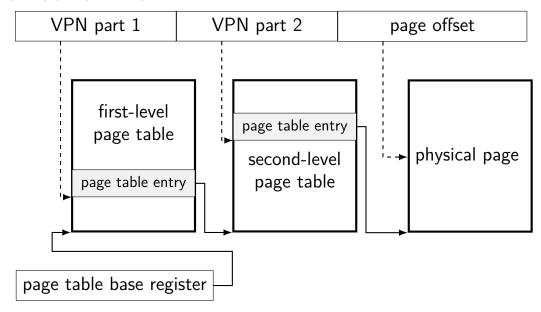








another view



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

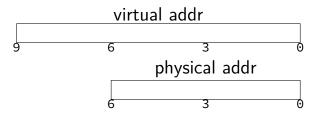
note on VPN splitting

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

assignment

9-bit virtual address

6-bit physical address



9-bit virtual address

virtual addr

VPN page offset

6 3

6-bit physical address

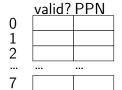
- physical addr
 PPN page offset
- 8-byte pages \rightarrow 3-bit page offset (bottom) ⁶
- 9-bit VA: 6 bit VPN + 3 bit PO
- 6-bit PA: 3 bit PPN + 3 bit PO

9-bit virtual address

virtual addr
VPN page offset
6 3

6-bit physical address

- physical addr
 PPN page offset
- 8-byte pages \rightarrow 3-bit page offset (bottom) 6
- 9-bit VA: 6 bit VPN + 3 bit PO
- 6-bit PA: 3 bit PPN + 3 bit PO
- 1 page page tables w/ 1 byte entry ightarrow 8 entry PTs



page table (either level)

9-bit virtual address

virtual addr

VPN pt 1 VPN pt 2 page offset

9 6 3

6-bit physical address

- physical addr
 PPN page offset
- 8-byte pages \rightarrow 3-bit page offset (bottom) 6
- 9-bit VA: 6 bit VPN + 3 bit PO
- 6-bit PA: 3 bit PPN + 3 bit PO
- 1 page page tables w/ 1 byte entry ightarrow 8 entry PTs
- valid? PPN
 0
 1
 2

 7

page table (either level)

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

physical addresses	byte	es			physical addresses	byt	es		
0x00-3			22	33	0x20-3			72	13
0x04-7	44	55	66	77	0x24-7	F4	Α5	36	07
0x08-B	88	99	AA	ВВ	0x28-B	89	9A	AB	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-B	EC	0C	EC	0C
0x1C-F	1C	2C	3C	4C	0x3C-F	AC	DC	DC	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	F4	Α5	36	07
0x08-B	88	99	AA	ВВ	0x28-B	89	9A	AB	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-B	EC	0C	EC	0C
0x1C-F	1C	2C	3C	4C	0x3C-F	AC	DC	DC	0C

physical addresses	byte	es			physical bytes addresses
0x00-3			22	33	
0x04-7	44	55	66	77	0x24-7 F4 A5 36 07
0x08-B	88	99	AΑ	ВВ	0x28-B89 9A AB BC
0x0C-F	CC	DD	EE	FF	0x2C-FCD DE EF F0
0x10-3	1A	2A	3A	4A	0x30-3 BA 0A BA 0A
0x14-7	1B	2B	3B	4B	0x34-7 DB 0B DB 0B
0x18-B	1C	2C	3C	4C	0x38-BEC 0C EC 0C
0x1C-F	1C	2C	3C	4C	0x3C-FAC DC DC 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	F4	Α5	36	07
0x08-B	88	99	AA	ВВ	0x28-B	89	9A	AB	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	AC	DC	DC	0C

physical addresses	byte	es			physical bytes addresses_	
0x00-3			22	33	0x20-3 <mark>00 91 72</mark>	13
0x04-7	44	55	66	77	0x24-7F4 A5 36	07
0x08-B	88	99	AΑ	ВВ	0x28-B89 9A AB	ВС
0x0C-F	CC	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	0x34-7DB 0B DB	0B
0x18-B	1C	2C	3C	4C	0x38-BEC 0C EC	0C
0x1C-F	1C	2C	3C	4C	0x3C-FAC DC DC	0C

physical addresses	byte	es			physical 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-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	0C
0x1C-F	1C	2C	3C	4C	0x3C-F	FC	0C	FC	0C

physical addresses	byt	es			a	physical iddresses	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	AB	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-B	EC	0C	EC	0C
0x1C-F	1C	2C	3C	4C	(0x3C-F	FC	0C	FC	0C
0x18-B	1C	2C	3C	4C	(0х38-В	EC	0C	EC	0C

2 D3
5 D7
BC
F F 0
4 0A
3 0B
0 O C
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-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	1B	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 bytes addresses		physical addresses	hytes		
addresses		addresses			
0x00-3 00 11 22	2 33	0x20-3	D0 D1	. D2	D3
0x04-744 55 66	6 77	0x24-7	D4 D5	D6	D7
0x08-B88 99 A	A BB	0x28-B	89 9 <i>A</i>	ΑB	ВС
0x0C-FCC DD EI	E FF	0x2C-F	CD DE	EF	F0
0x10-3 1A 2A 3/	4 4A	0x30-3	BA 0A	ВА	0Α
0x14-7 1B 2B 3B	3 4B	0x34-7	DB 0E	DB	0B
0x18-B 1C 2C 30	C 4C	0x38-B	EC 00	EC	0C
0x1C-F1C 2C 30	C 4C	0x3C-F	FC 00	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 0x10; translate virtual address 0x109

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

physical bytes addresses					
0x20-3	D0	D1	D2	D3	
0x24-7	D4	D5	D6	D7	
0x28-B	89	9Α	ΑB	ВС	
0x2C-F	CD	DE	EF	F0	
0x30-3	ВА	0A	ВА	0Α	
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 0x00B

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	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

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 bytes addresses				
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
			3B	
0x18-B	1C	2C	3C	4C
0x1C-F	1C	2C	3C	4C

physical bytes addresses				
0x20-3			D2	D3
0x24-7				
0x28-B				
0x2C-F	CD	DE	EF	F0
0x30-3	ВА	0A	ВА	0Α
0x34-7	DΒ	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 0x00B

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

physical bytes ddresses					
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	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 bytes addresses					
0x00-3					
0x04-7	44	55	66	77	
0x08-B	88	99	AΑ	ВВ	
0x0C-F	CC	DD	EE	FF	
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			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	0A
0x34-7	DB	0B	DB	0B
0x38-B	EC	0C	EC	0C
0x3C-F	FC.	0 C	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 bytes addresses					
0x00-3	00	11	22	33	
0x04-7	44	55	66	77	
0x08-B	88	99	AΑ	ВВ	
0x0C-F					
0x10-3	1A	2A	3A	4A	
0x14-7	1В	2B	3B	4B	
0x18-B		2C			
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 bytes addresses					
0x00-300 1					
$0 \times 04 - 7 = 44 = 5$	55 66 77				
0x08-B88 9	9 AA BB				
0x0C-FCC D					
0x10-3 1A 2	2A 3A 4A				
0x14-7 1B 2	B 3B 4B				
	2C 3C 4C				
0x1C-FAC B	BC DC EC				

```
physical bytes
addresses
0x20-3D0 E1 D2 D3
0x24-7D4 E5 D6 E7
0x28-Bl89 9A AB BC
0x2C-FCD DE EF F0
0x30-3|BA 0A BA 0A
0x34-7DB 0B DB 0B
0x38-B|EC 0C EC 0C
0x3C-FIFC 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 bytes addresses					
0x00-300 1					
$0 \times 04 - 7 = 44 = 5$	55 66 77				
0x08-B88 9	9 AA BB				
0x0C-FCC D					
0x10-3 1A 2	2A 3A 4A				
0x14-71B2	B 3B 4B				
	2C 3C 4C				
0x1C-FAC B	BC DC EC				

```
physical bytes
addresses
0x20-3D0 E1 D2 D3
0x24-7D4 E5 D6 E7
0x28-Bl89 9A AB BC
0x2C-FCD DE EF F0
0x30-3|BA 0A BA 0A
0x34-7DB 0B DB 0B
0x38-B|EC 0C EC 0C
0x3C-FIFC 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		
addresses,	-) -			
0x00-3				
0x04-7	44	55	66	77
0x08-B	88	99	AΑ	ВВ
0x0C-F	CC	DD	EE	FF
0x10-3	1A	2A	ЗА	4A
0x14-7	1B	2B	3B	4B
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 bytes	;
0x00-300 1	
$0 \times 04 - 7 = 44 = 5$	55 66 77
0x08-B88 9	9 AA BB
0x0C-FCC D	
0x10-3 1A 2	2A 3A 4A
0x14-71B2	B 3B 4B
	2C 3C 4C
0x1C-FAC B	BC DC EC

```
physical bytes
addresses
0x20-3D0 E1 D2 D3
0x24-7D4 E5 D6 E7
0x28-Bl89 9A AB BC
0x2C-FCD DE EF F0
0x30-3|BA 0A BA 0A
0x34-7DB 0B DB 0B
0x38-B|EC 0C EC 0C
0x3C-FIFC 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	<u>مح</u>		
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	1B	2B	3B	4B
0x18-B				
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 bytes	;
0x00-300 1	
$0 \times 04 - 7 = 44 = 5$	55 66 77
0x08-B88 9	9 AA BB
0x0C-FCC D	
0x10-3 1A 2	2A 3A 4A
0x14-71B2	B 3B 4B
	2C 3C 4C
0x1C-FAC B	BC DC EC

```
physical bytes
addresses
0x20-3D0 E1 D2 D3
0x24-7D4 E5 D6 E7
0x28-Bl89 9A AB BC
0x2C-FCD DE EF F0
0x30-3|BA 0A BA 0A
0x34-7DB 0B DB 0B
0x38-B|EC 0C EC 0C
0x3C-FIFC 0C FC 0C
```

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

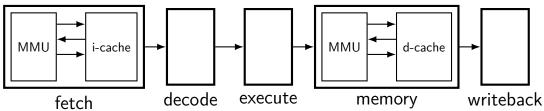
page tables 1 page; PTE 1st byte: (MSB) 2-bit PPN, valid bit; rest unused

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

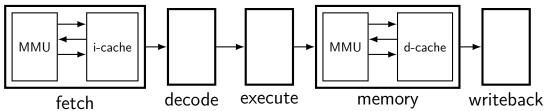
backup slides

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)

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
Heap $/$ other dynamic	Heap / other dynamic
Writable data	Writable data
Code + Constants can't be	e 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? page

•••

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?		
vviice.	page	
•••	•••	
0	0x12345	
0	0x12347	
0	0x12340	
0	0x200DF	
0	0x200AF	
•••	•••	
	 0 0 0	

V 1 1 V
•••
0x00601
0x00602
0x00603
0x00604
0x00605

VPN

valid?	write?	physical page

•••	•••	•••
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

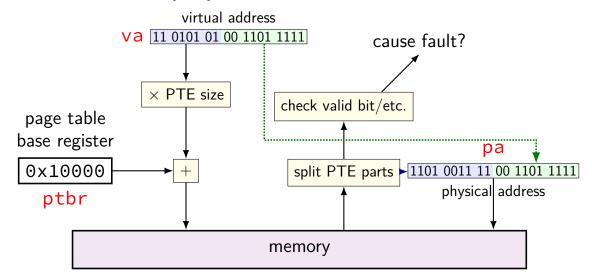
VPN	physical valid? write?		physical	VPN	physical valid? write? page		
VIIV	valiu:	VVIILC	page	V I I I I	valiu:	WIILC:	page
•••	•••	•••	•••	•••	•••	•••	•••
0x00601	1	0	0x12345	0x00601	1	0	0x12345
0x00602	1	0	0x12347	0x00602	1	0	0x12347
0x00603	1	0	0x12340	0x00603	1	0	0x12340
0x00604	1	0	0x200DF	<u>0x00604</u>	1	0	0x200DF
0x00605	1	0	0x200AF	0x00605	1	0	0x200AF
•••	•••	•••	•••	•••	•••	•••	•••

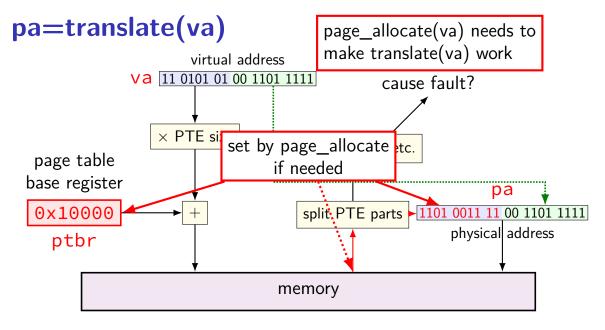
when either process tries to write read-only page triggers a fault — OS actually copies the page

VPN	valid?	write	physical page	VPN	valid?	write	physical page
			page	1			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	<u>0x00604</u>	1	0	0x200DF
0x00605	1	0	0x200AF	0x00605	1	1	0x300FD
•••	•••	•••	•••	•••	•••	•••	•••

after allocating a copy, OS reruns the write instruction

pa=translate(va)





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

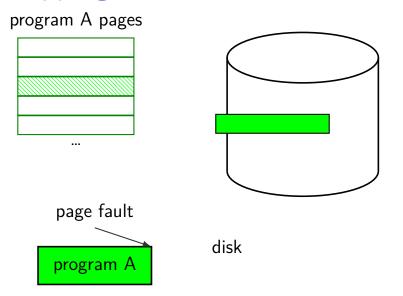
```
"swap in" a page — exactly like allocating on demand!
    OS gets page fault — invalid in page table
    check where page actually is (from virtual address)
    read from disk
    eventually restart process
"swap out" a page
    OS marks as invalid in the page table(s)
    copy to disk (if modified)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

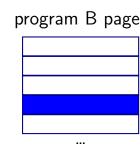
HDD reads and writes: milliseconds to tens of milliseconds minimum size: 512 bytes writing tens of kilobytes basically as fast as writing 512 bytes

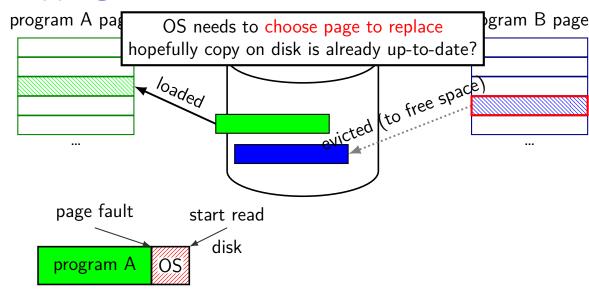
HDD reads and writes: milliseconds to tens of milliseconds minimum size: 512 bytes writing tens of kilobytes basically as fast as writing 512 bytes

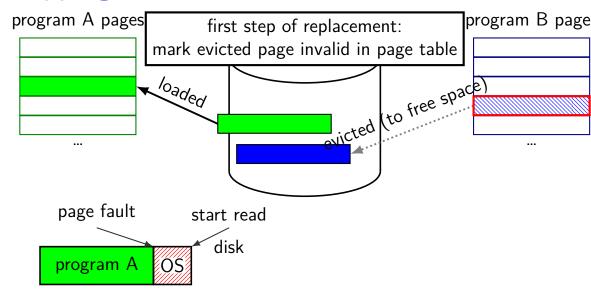
HDD reads and writes: milliseconds to tens of milliseconds minimum size: 512 bytes writing tens of kilobytes basically as fast as writing 512 bytes

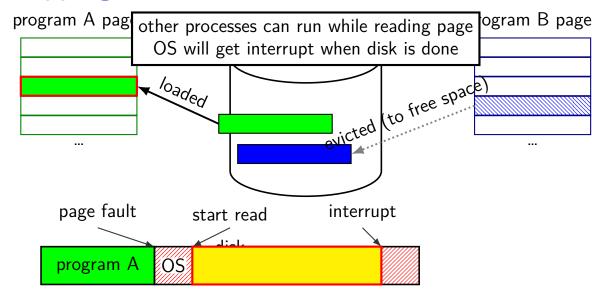
HDD reads and writes: milliseconds to tens of milliseconds minimum size: 512 bytes writing tens of kilobytes basically as fast as writing 512 bytes

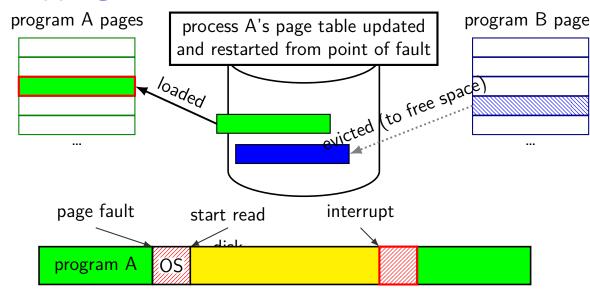












swapping almost mmap

```
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