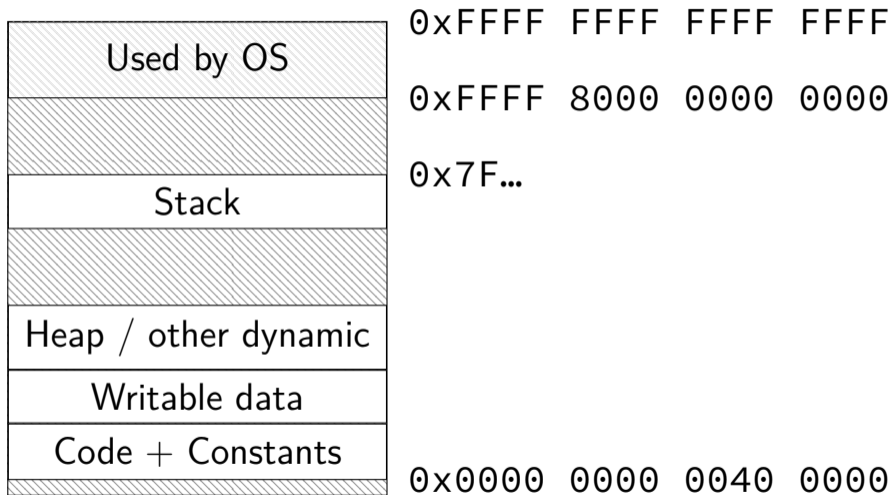


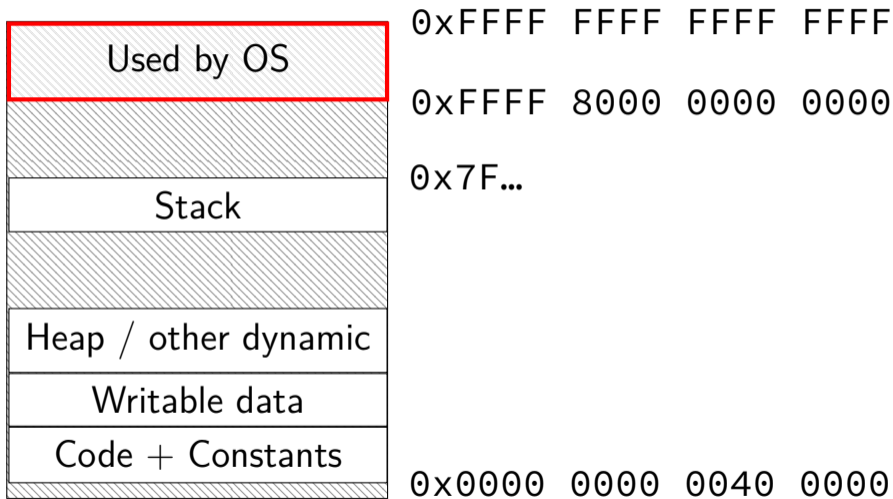
changelog

20 Feb 2025: correct error in transcription of 0×109 in solution slide for '2-level exercise (2)' that was carried through rest of exercise

program memory

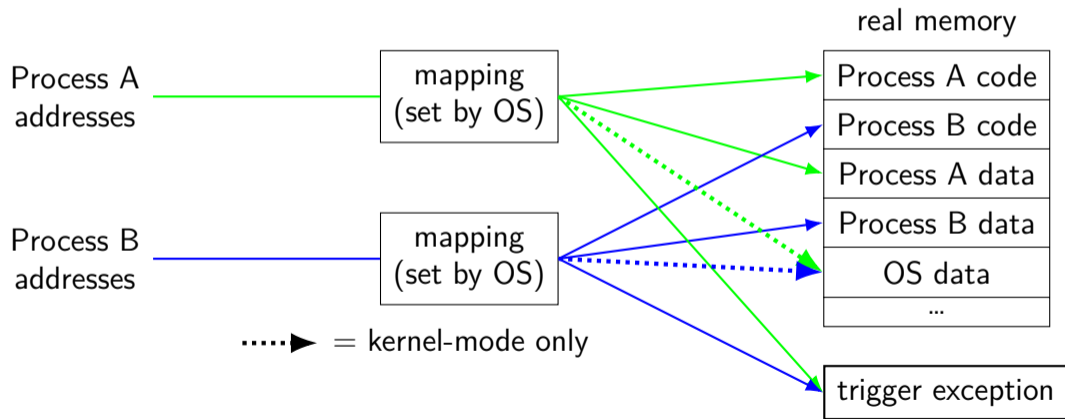


program memory



address spaces

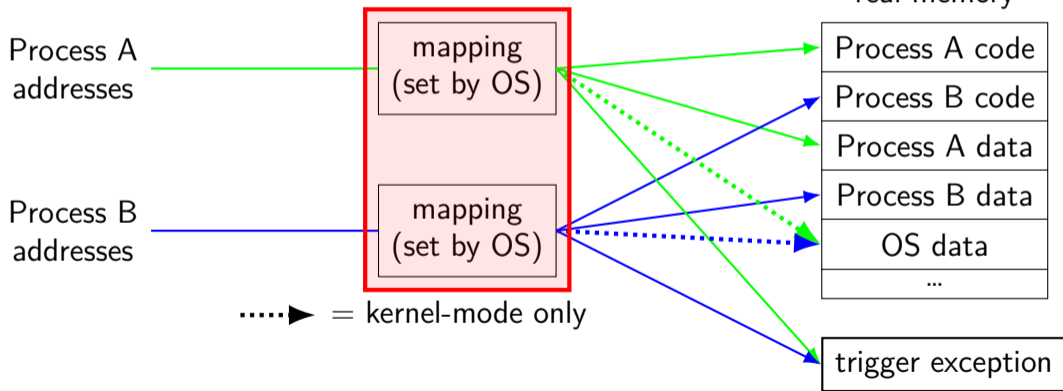
illusion of *dedicated memory*



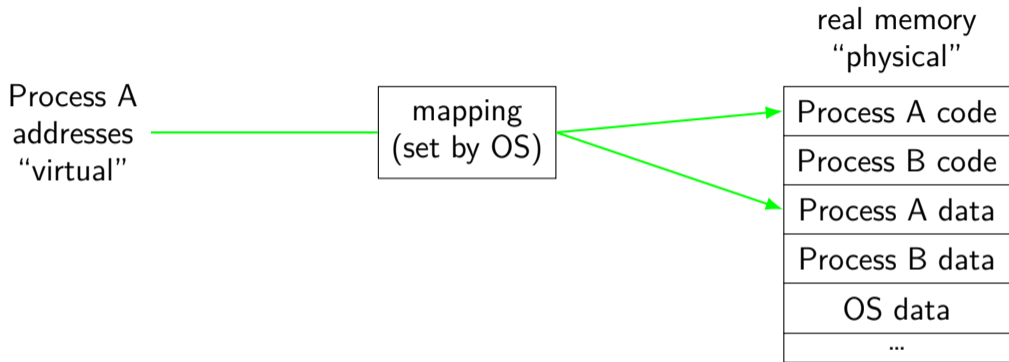
address spaces

illusion of *dedicated memory*

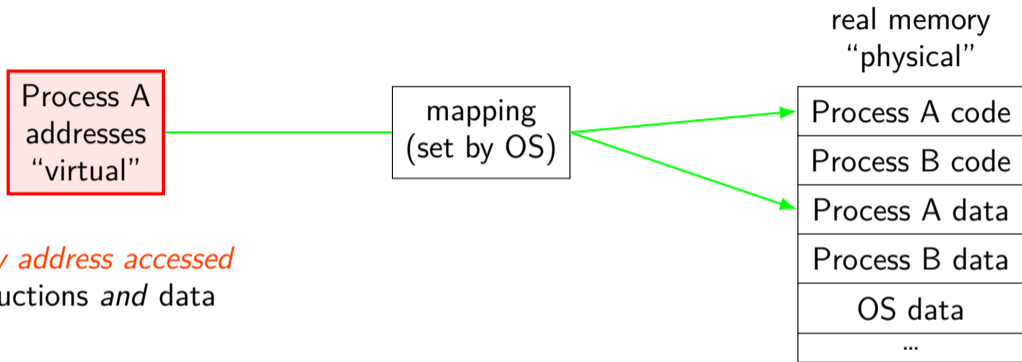
chose one during context switch



address translation

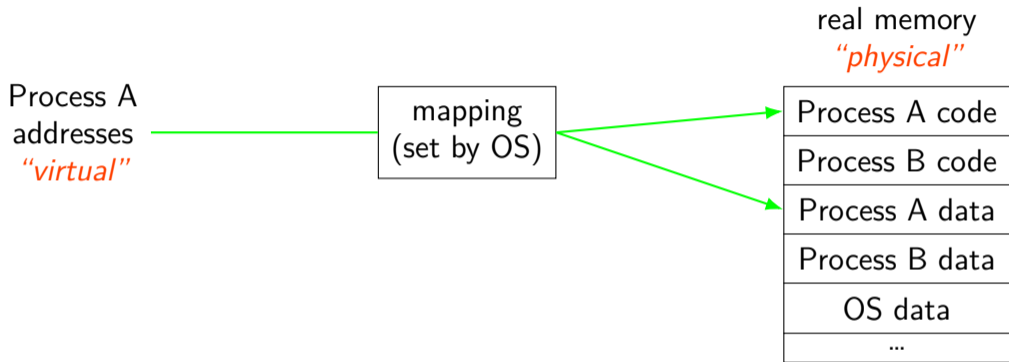


address translation



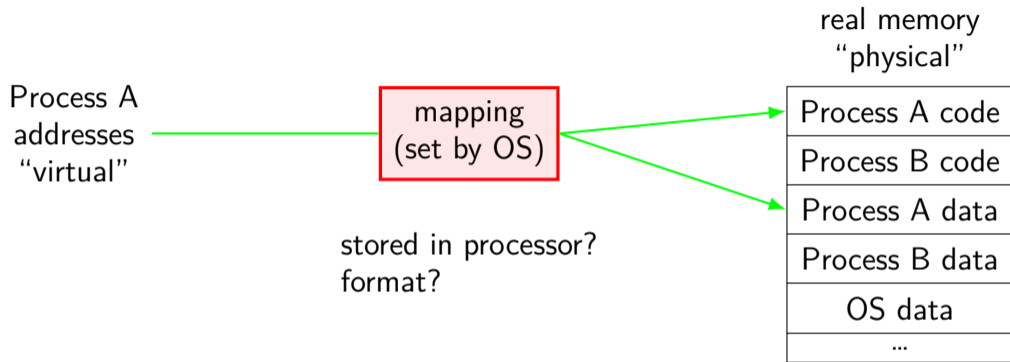
every address accessed
instructions *and* data

address translation

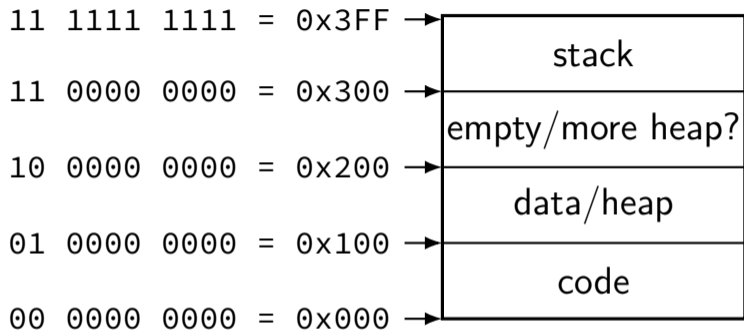


program addresses are 'virtual'
real addresses are 'physical'
can be *different sizes!*

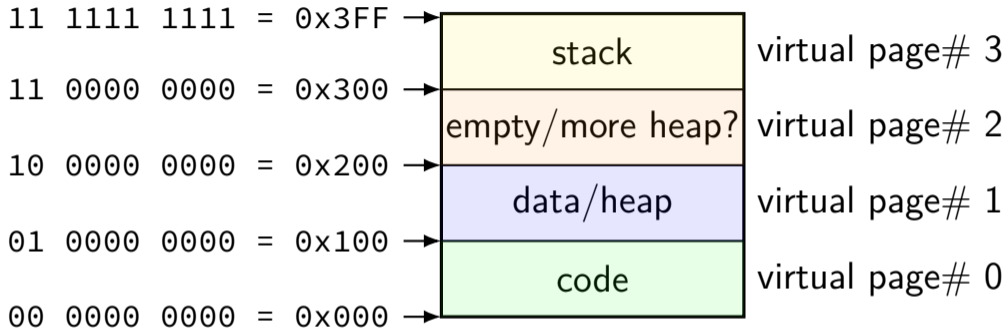
address translation



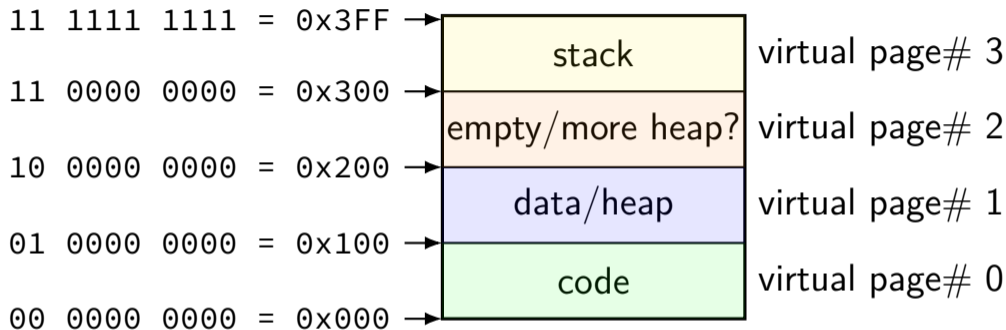
toy program memory



toy program memory

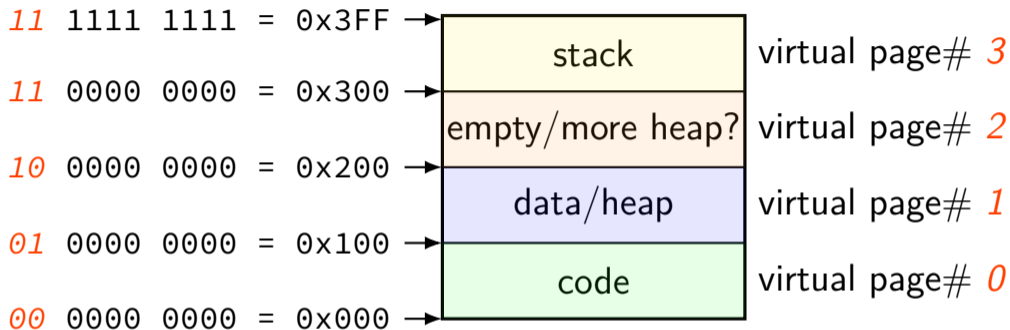


toy program memory



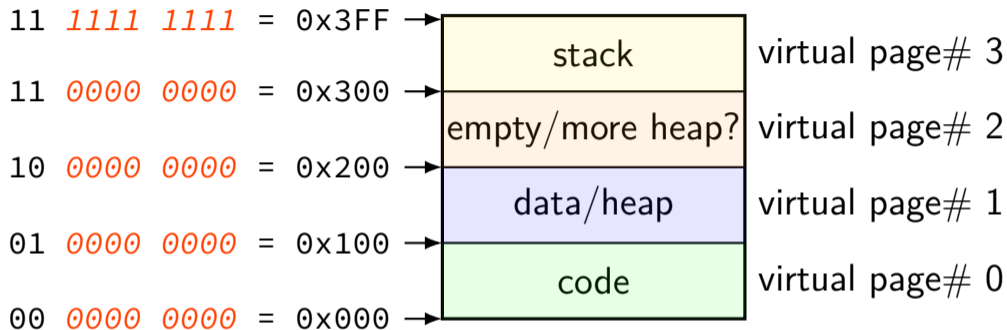
divide memory into *pages* (2^8 bytes in this case)
“virtual” = addresses the program sees

toy program memory



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

toy program memory



rest of address is called *page offset*

toy physical memory

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

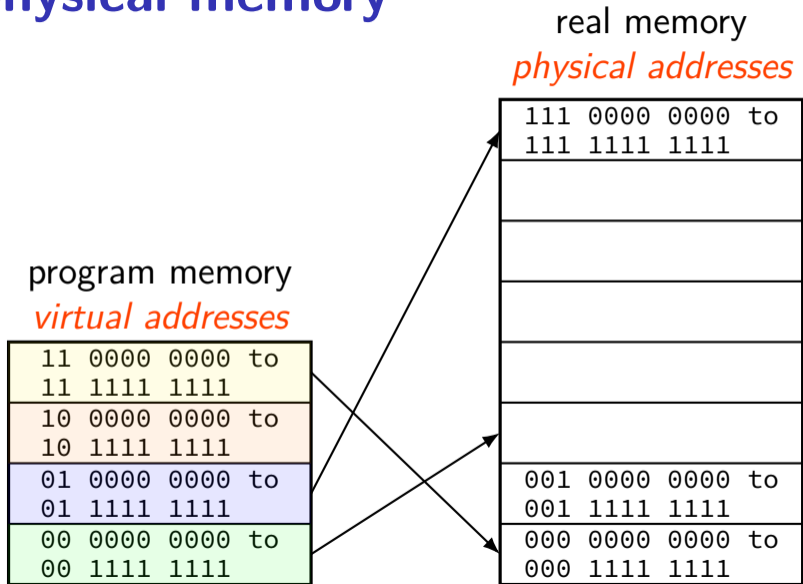
<i>111</i> 0000 0000 to <i>111</i> 1111 1111
<i>001</i> 0000 0000 to <i>001</i> 1111 1111
<i>000</i> 0000 0000 to <i>000</i> 1111 1111

physical page 7

physical page 1

physical page 0

toy physical memory



toy physical memory

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

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

toy physical memory

page
table! real memory

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

physical addresses

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

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

toy page table lookup

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

toy page table lookup

01 1101 0010 — address from CPU

virtual
page # valid? physical page #

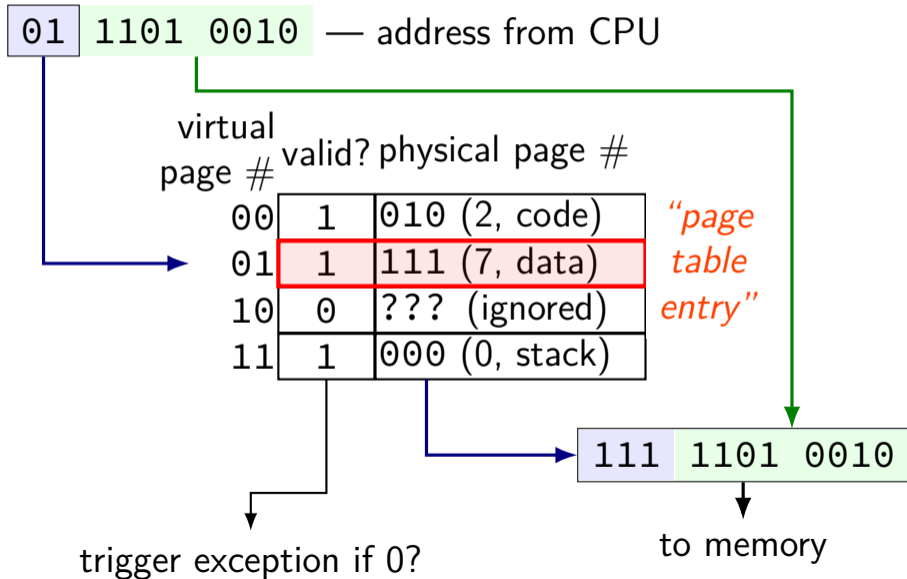
00	1	010 (2, code)
01	1	111 (7, data)
10	0	??? (ignored)
11	1	000 (0, stack)

111 1101 0010

trigger exception if 0?

to memory

toy page table lookup



t “virtual page number” lookup

01 1101 0010 — address from CPU

virtual
page # valid? physical page #

00	1	010 (2, code)
01	1	111 (7, data)
10	0	??? (ignored)
11	1	000 (0, stack)

111 1101 0010

trigger exception if 0?

to memory

toy page table lookup

01 1101 0010 — address from CPU

virtual
page # valid? physical page #

00	1	010 (2, code)
01	1	111 (7, data)
10	0	??? (ignored)
11	1	000 (0, stack)

“physical page number”

111 1101 0010

trigger exception if 0?

to memory

toy pag "page offset" lookup

01 1101 0010 — address from CPU

virtual
page # valid? physical page #

00	1	010 (2, code)
01	1	111 (7, data)
10	0	??? (ignored)
11	1	000 (0, stack)

"page offset"

111 1101 0010

trigger exception if 0?

to memory

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 address = 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 address = 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?

$$2^{32} / 2^{12} = 2^{20}$$

exercise: page table size

suppose 32-bit virtual (program) addresses

suppose 30-bit physical (hardware) addresses

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

page 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)

page table entries have physical page #, valid bit, bit

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

2^{20} entries \times (18 + 1) bits per entry

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:

page #: 0x12345; offset: 0x678

exercise: page table lookup

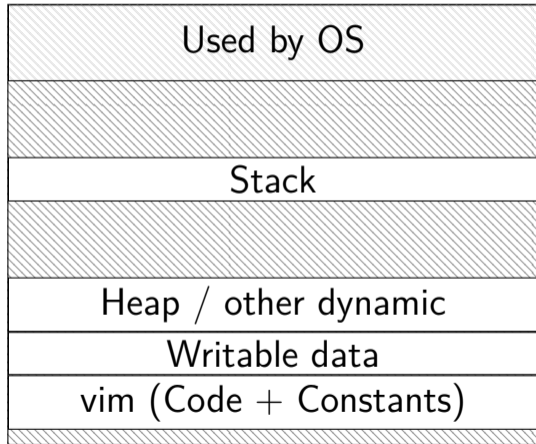
suppose 64-byte pages (= 6-bit page offsets), 9-bit virtual addresses

VPN	valid	PPN
000	1	0010
001	1	1010
010	0	---
011	0	---
100	1	1110
101	1	0100
110	1	0001
111	0	---

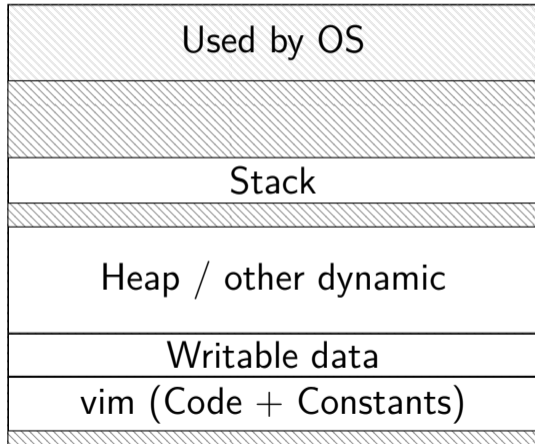
virtual address 0x024 (0 0010 0100) = physical address ???

vim (two copies)

Vim (run by user mst3k)

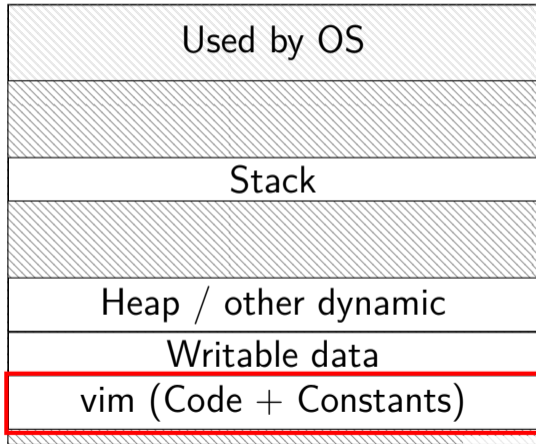


Vim (run by user xyz4w)

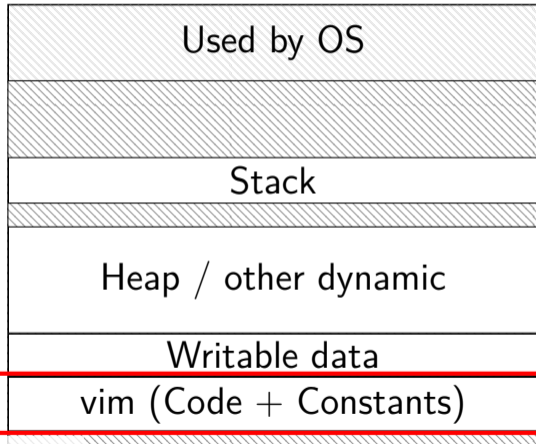


vim (two copies)

Vim (run by user mst3k)



Vim (run by user xyz4w)



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?

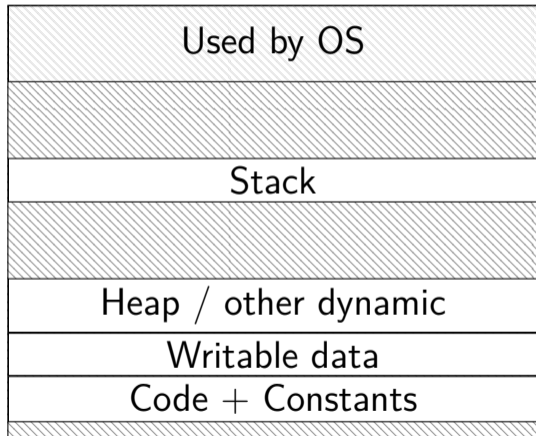
checked by hardware like valid bit

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

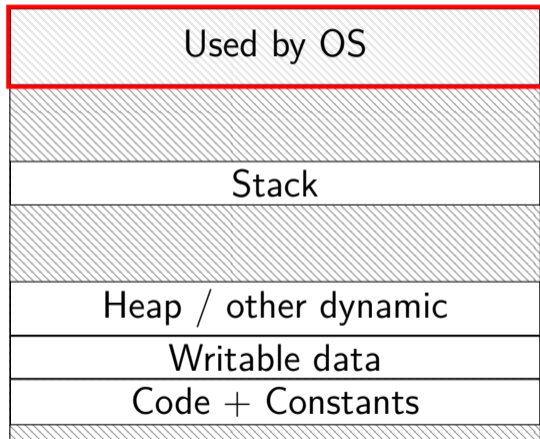
running a program

Some program



running a program

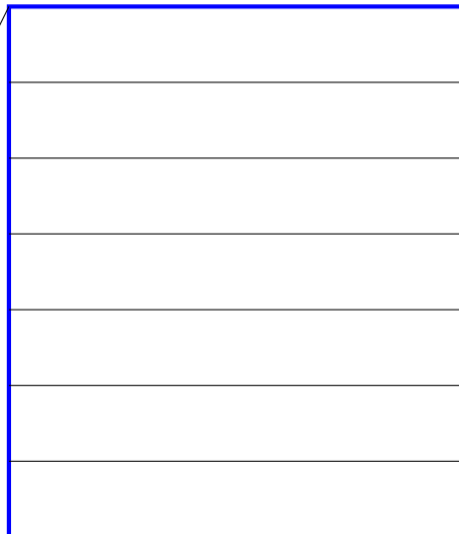
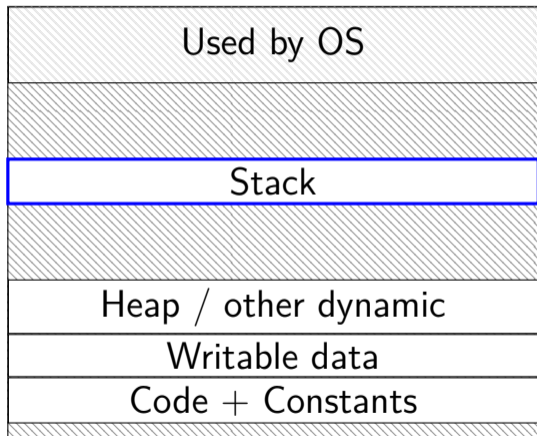
Some program



OS's memory

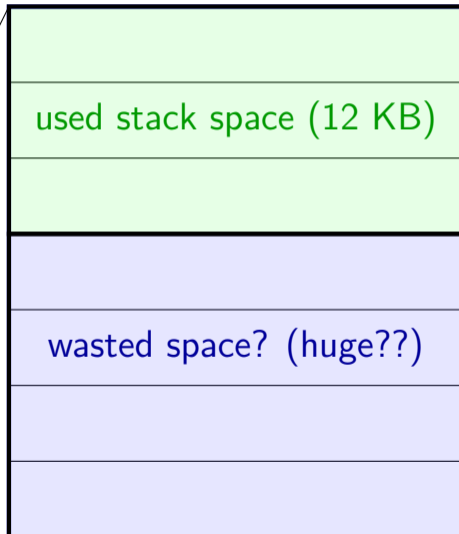
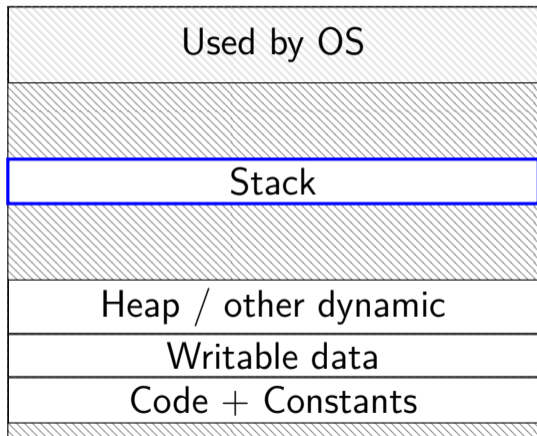
space on demand

Program Memory



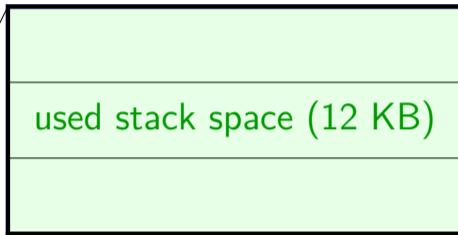
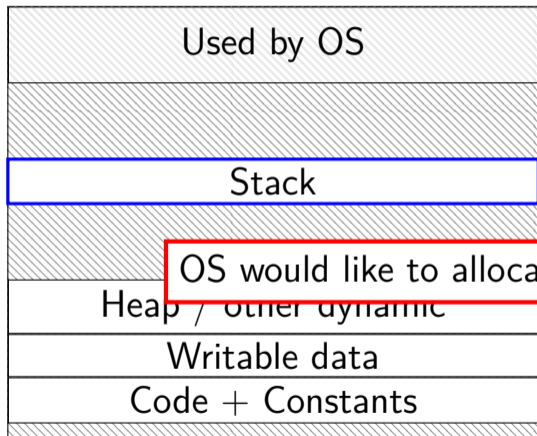
space on demand

Program Memory



space on demand

Program Memory



OS would like to allocate space only if needed

wasted space? (huge??)

allocating space on demand

`%rsp = 0x7FFFC000`

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

VPN

```
...  
0x7FFFB  
0x7FFFC  
0x7FFFD  
0x7FFFE  
0x7FFFF  
...
```

valid? physical
page

valid?	physical page
...	...
0	---
1	0x200DF
1	0x12340
1	0x12347
1	0x12345
...	...

allocating space on demand

`%rsp = 0x7FFFC000`

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

VPN

```
...  
0x7FFFB  
0x7FFFC  
0x7FFFD  
0x7FFFE  
0x7FFFF  
...
```

valid? physical
page

valid?	physical page
...	...
0	---
1	0x200DF
1	0x12340
1	0x12347
1	0x12345
...	...

pushq triggers exception
hardware says “accessing address 0x7FFBFF8”
OS looks up what’s should be there — “stack”

allocating space on demand

`%rsp = 0x7FFFC000`

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

VPN	valid?	physical page
...
<code>0x7FFFB</code>	1	<code>0x200D8</code>
<code>0x7FFFC</code>	1	<code>0x200DF</code>
<code>0x7FFFD</code>	1	<code>0x12340</code>
<code>0x7FFFE</code>	1	<code>0x12347</code>
<code>0x7FFFF</code>	1	<code>0x12345</code>
...

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

allocating space on demand

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

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

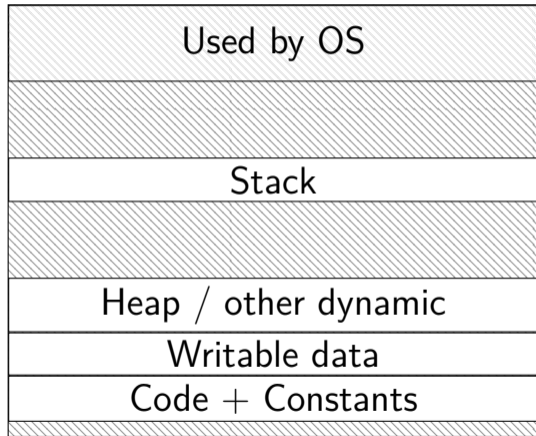
loading program can be *merely creating empty page table*

everything else can be handled *in response to page faults*

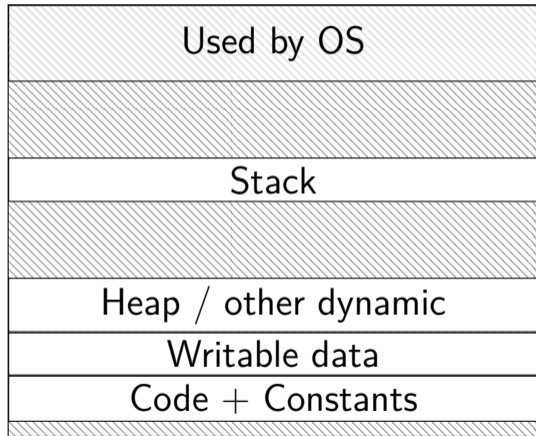
no time/space spent loading/allocating unneeded space

do we really need a complete copy?

bash

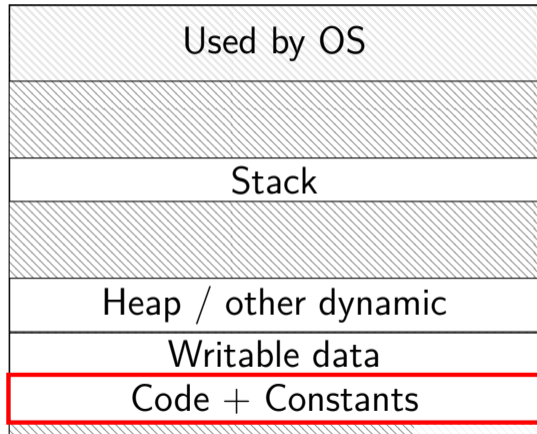


new copy of bash

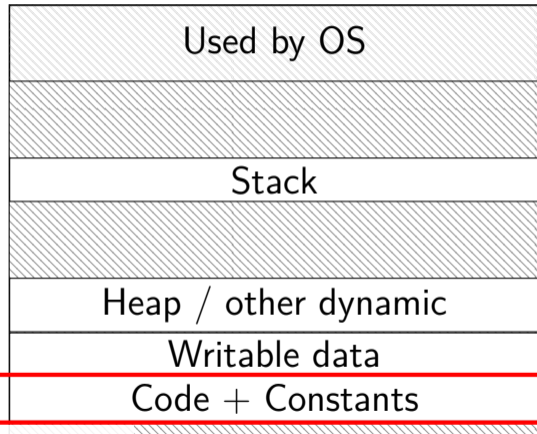


do we really need a complete copy?

bash



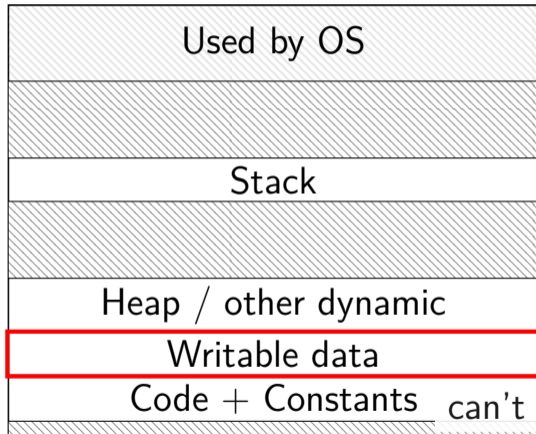
new copy of bash



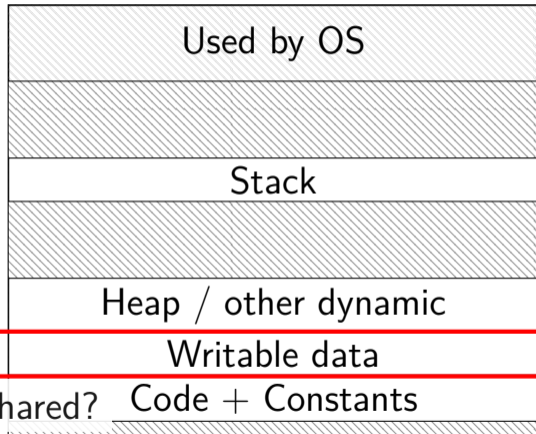
shared as read-only

do we really need a complete copy?

bash



new copy of bash



can't be shared?

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

copy-on-write and page tables

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

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

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

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

VPN	valid?	write?	physical page
...
0x00601	1	0	0x12345
0x00602	1	0	0x12347
0x00603	1	0	0x12340
0x00604	1	0	0x200DF
0x00605	1	1	0x300FD
...

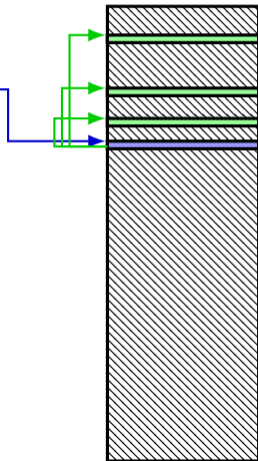
after allocating a copy, OS reruns the write instruction

fork (w/ copy-on-write, if parent writes first)

parent process info

user regs	rax (return val.)=42 child pid, rcx=133, ...
page tables	
open files	fd 0: ... fd 1: ...
...	...

memory

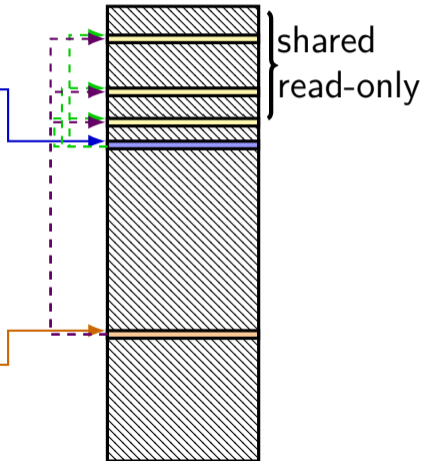


fork (w/ copy-on-write, if parent writes first)

parent process info

user regs	rax (return val.)=42 child pid, rcx=133, ...
page tables	
open files	fd 0: ... fd 1: ...
...	...

memory



child process info

user regs	rax (return val.)=420, rcx=133, ...
page tables	
open files	fd 0: ... fd 1: ...
...	...

fork (w/ copy-on-write, if parent writes first)

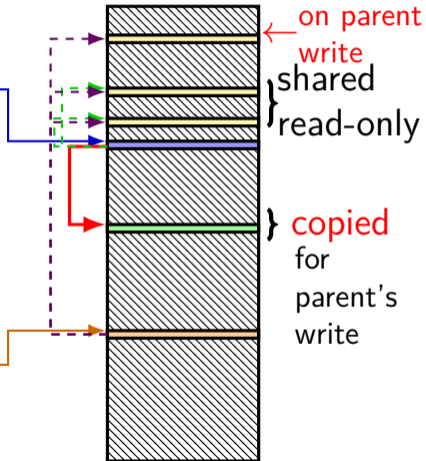
parent process info

user regs	rax (return val.)=42 child pid, rcx=133, ...
page tables	
open files	fd 0: ... fd 1: ...
...	...

child process info

user regs	rax (return val.)=420, rcx=133, ...
page tables	
open files	fd 0: ... fd 1: ...
...	...

memory



fork (w/ copy-on-write, if parent writes first)

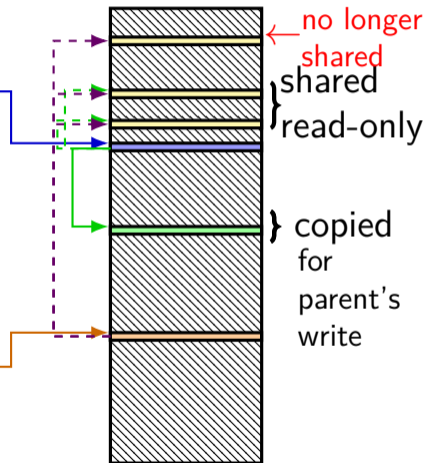
parent process info

user regs	rax (return val.)=42 child pid, rcx=133, ...
page tables	
open files	fd 0: ... fd 1: ...
...	...

child process info

user regs	rax (return val.)=420, rcx=133, ...
page tables	
open files	fd 0: ... fd 1: ...
...	...

memory



fork (w/ copy-on-write, if parent writes first)

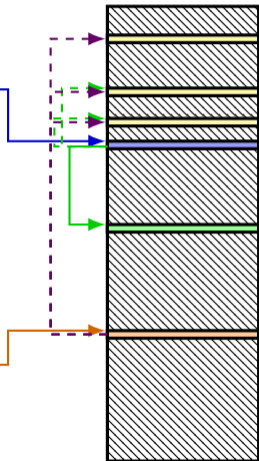
parent process info

user regs	rax (return val.)=42 ^{child pid} , rcx=133, ...
page tables	
open files	fd 0: ... fd 1: ...
...	...

child process info

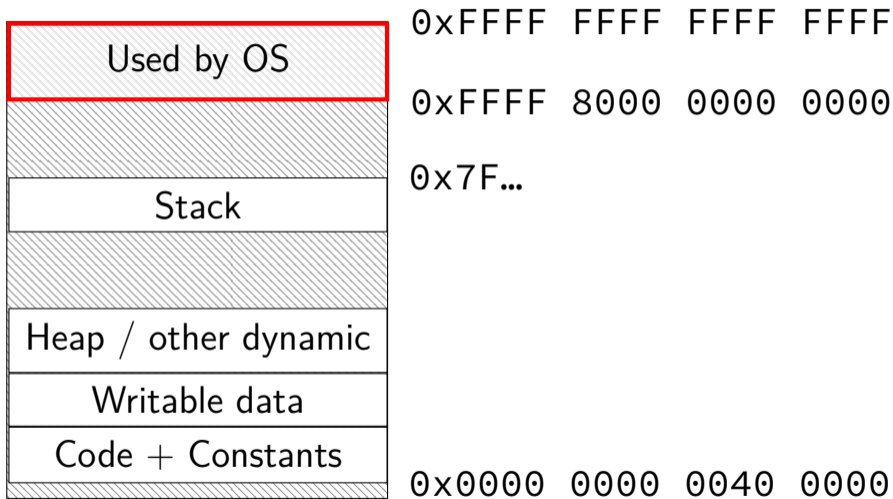
user regs	rax (return val.)=42 ⁰ , rcx=133, ...
page tables	
open files	fd 0: ... fd 1: ...
...	...

memory



} copied for parent's write

program memory



running OS code

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

running OS code

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

where in memory is this OS code?

running 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

running 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

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.s
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.s
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 [vdso]
ffffffffffff600000-ffffffffffff601000 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
0060b000-0
01974000-0
7f60c718b0
7f60c74900
7f60c764e0
7f60c784e0
7f60c78520
7f60c78540
7f60c78590
7f60c7a390
7f60c7a7a0
7f60c7a7b0
7f60c7a7c0
7f60c7a7d0
7ffc5d2b20
7ffc5d3b00
7ffc5d3b30
ffffffff600000-ffffffff601000 r-xp 00000000 00:00 0 [vsyscall]
```

OS tracks list of struct `vm_area_struct` with:

(shown in this output):

virtual address start, end

permissions

offset in backing file (if any)

pointer to backing file (if any)

(not shown):

info about sharing of non-file data ...

cafe-archive

gnu/libc-2.1

gnu/libc-2.1

gnu/libc-2.1

gnu/libc-2.1

gnu/ld-2.19.s

gnu/ld-2.19.s

gnu/ld-2.19.s

gnu/ld-2.19.s

gnu/ld-2.19.s

gnu/ld-2.19.s

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

copy-on-write

saving data temporarily to disk, reloading to memory on demand
“swapping”

detecting whether memory was read/written recently

stopping in a debugger when a variable is modified

sharing memory between programs on two different machines

example page table tricks

allocating space on demand

loading code/data from files on disk on demand

copy-on-write

saving data temporarily to disk, reloading to memory on demand

“swapping”

detecting whether memory was read/written recently

stopping in a debugger when a variable is modified

sharing memory between programs on two different machines

example page table tricks

allocating space on demand

loading code/data from files on disk on demand

copy-on-write

saving data temporarily to disk, reloading to memory on demand
“swapping”

detecting whether memory was read/written recently

stopping in a debugger when a variable is modified

sharing memory between programs on two different machines

example page table tricks

allocating space on demand

loading code/data from files on disk on demand

copy-on-write

saving data temporarily to disk, reloading to memory on demand
“swapping”

detecting whether memory was read/written recently

stopping in a debugger when a variable is modified

sharing memory between programs on two different machines

example page table tricks

allocating space on demand

loading code/data from files on disk on demand

copy-on-write

saving data temporarily to disk, reloading to memory on demand
“swapping”

detecting whether memory was read/written recently

stopping in a debugger when a variable is modified

sharing memory between programs on two different machines

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

exercise setup

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

page table

virtual page #	valid?	physical 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
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 addresses	bytes
0x20-3	D0 D1 D2 D3
0x24-7	D4 D5 D6 D7
0x28-B	89 9A AB BC
0x2C-F	CD DE EF F0
0x30-3	BA 0A BA 0A
0x34-7	CB 0B CB 0B
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

page table

virtual page #	valid?	physical 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
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 addresses	bytes
0x20-3	01 D2 D3
0x24-7	05 D6 D7
0x28-B	0A AB BC
0x2C-F	0E EF F0
0x30-3	0A 0A BA 0A
0x34-7	0B 0B CB 0B
0x38-B	0C 0C DC 0C
0x3C-F	0C 0C EC 0C

phys. page 0

phys. page 1

exercise

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
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
$0x0C-F$	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 addresses	bytes
$0x20-3$	D0 D1 D2 D3
$0x24-7$	D4 D5 D6 D7
$0x28-B$	89 9A AB BC
$0x2C-F$	CD DE EF F0
$0x30-3$	BA 0A BA 0A
$0x34-7$	CB 0B CB 0B
$0x38-B$	DC 0C DC 0C
$0x3C-F$	EC 0C EC 0C

exercise

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

(virtual addresses) $0x18 = 00$; $0x03 = ???$; $0x0A = ???$; $0x13 = ???$

page table

virtual page #	valid?	physical 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
$0x0C-F$	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 addresses	bytes
$0x20-3$	D0 D1 D2 D3
$0x24-7$	D4 D5 D6 D7
$0x28-B$	89 9A AB BC
$0x2C-F$	CD DE EF F0
$0x30-3$	BA 0A BA 0A
$0x34-7$	CB 0B CB 0B
$0x38-B$	DC 0C DC 0C
$0x3C-F$	EC 0C EC 0C

exercise

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

(virtual addresses) $0x18 = 00$; $0x03 = 0x4A$; $0x0A = ???$; $0x13 = ???$

page table

virtual page #	valid?	physical 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
$0x0C-F$	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 addresses	bytes
$0x20-3$	D0 D1 D2 D3
$0x24-7$	D4 D5 D6 D7
$0x28-B$	89 9A AB BC
$0x2C-F$	CD DE EF F0
$0x30-3$	BA 0A BA 0A
$0x34-7$	CB 0B CB 0B
$0x38-B$	DC 0C DC 0C
$0x3C-F$	EC 0C EC 0C

exercise

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

(virtual addresses) $0x18 = 00$; $0x03 = 0x4A$; $0x0A = 0xDC$; $0x13 = ???$

page table

virtual page #	valid?	physical 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
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 addresses	bytes
0x20-3	D0 D1 D2 D3
0x24-7	D4 D5 D6 D7
0x28-B	89 9A AB BC
0x2C-F	CD DE EF F0
0x30-3	BA 0A BA 0A
0x34-7	CB 0B CB 0B
0x38-B	DC 0C DC 0C
0x3C-F	EC 0C EC 0C

exercise

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

(virtual addresses) $0x18 = 00$; $0x03 = 0x4A$; $0x0A = 0xDC$; $0x13 = \text{fault}$

page table

virtual page #	valid?	physical 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
$0x0C-F$	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 addresses	bytes
$0x20-3$	D0 D1 D2 D3
$0x24-7$	D4 D5 D6 D7
$0x28-B$	89 9A AB BC
$0x2C-F$	CD DE EF F0
$0x30-3$	BA 0A BA 0A
$0x34-7$	CB 0B CB 0B
$0x38-B$	DC 0C DC 0C
$0x3C-F$	EC 0C EC 0C

page tables in memory

where can processor store megabytes of page tables? *in memory*

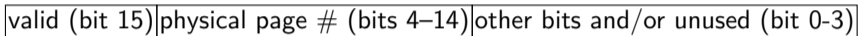
page table entry layout (chosen by processor)

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

page tables in memory


where can processor store megabytes of page tables? *in memory*

page table entry layout (chosen by processor)



page table
base register

0x00010000

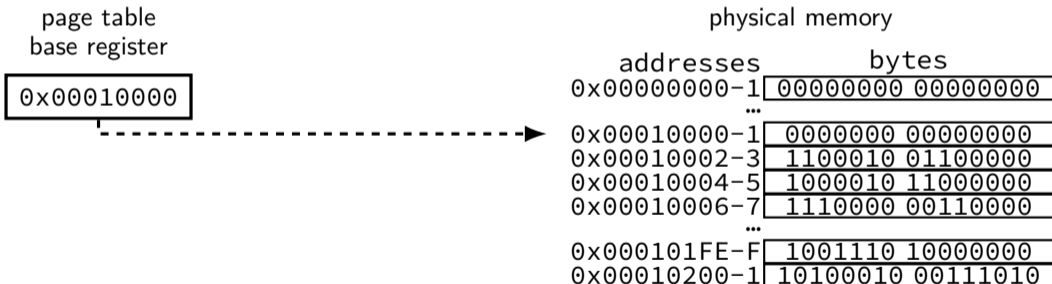


page tables in memory

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)
----------------	-----------------------------	------------------------------------



page tables in memory

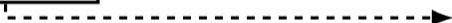
where can processor store megabytes of page tables? *in memory*

page table entry layout (chosen by processor)

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

page table
base register

0x00010000



physical memory

addresses	bytes
0x00000000-1	00000000 00000000
...	...
0x00010000-1	00000000 00000000
0x00010002-3	1100010 01100000
0x00010004-5	1000010 11000000
0x00010006-7	1110000 00110000
...	...
0x000101FE-F	1001110 10000000
0x00010200-1	10100010 00111010

page tables in memory

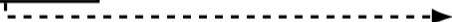
where can processor store megabytes of page tables? *in memory*

page table entry layout (chosen by processor)

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

page table
base register

0x00010000



physical memory

addresses	bytes
0x00000000-1	00000000 00000000
...	...
0x00010000-1	00000000 00000000
0x00010002-3	1100010 01100000
0x00010004-5	1000010 11000000
0x00010006-7	1110000 00110000
...	...
0x000101FE-F	1001110 10000000
0x00010200-1	10100010 00111010

page tables in memory

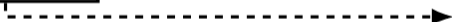
where can processor store megabytes of page tables? *in memory*

page table entry layout (chosen by processor)

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

page table
base register

0x00010000



physical memory

addresses	bytes
0x00000000-1	00000000 00000000
...	
0x00010000-1	00000000 00000000
0x00010002-3	1100010 01100000
0x00010004-5	1000010 11000000
0x00010006-7	1110000 00110000
...	
0x000101FE-F	1001110 10000000
0x00010200-1	10100010 00111010

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)
----------------	-----------------------------	------------------------------------

page table
base register

0x00010000

page table (logically)

virtual page #	valid?	...	physical page #
0000 0000	0	...	00 0000 0000
0000 0001	1	...	10 0010 0110
0000 0010	1	...	00 0000 1100
0000 0011	1	...	11 0000 0011
...			
1111 1111	1	...	00 1110 1000

physical memory

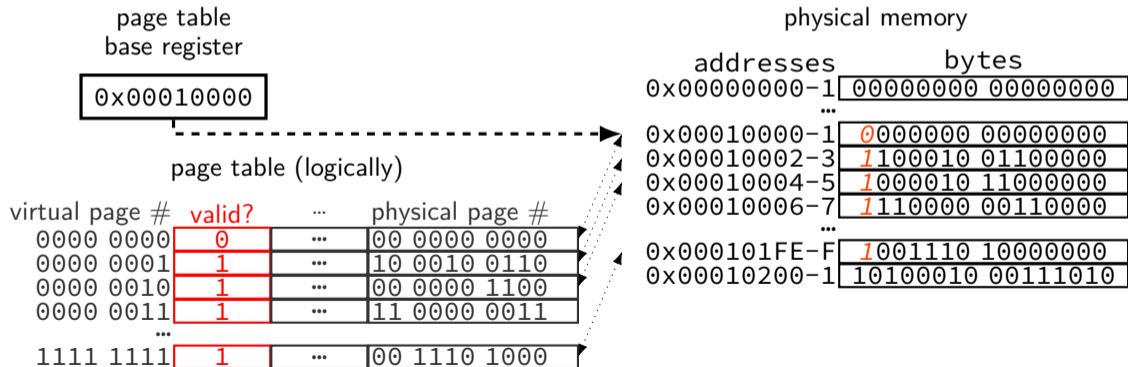
addresses	bytes
0x00000000-1	00000000 00000000
...	
0x00010000-1	00000000 00000000
0x00010002-3	1100010 01100000
0x00010004-5	1000010 11000000
0x00010006-7	1110000 00110000
...	
0x000101FE-F	1001110 10000000
0x00010200-1	10100010 00111010

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)

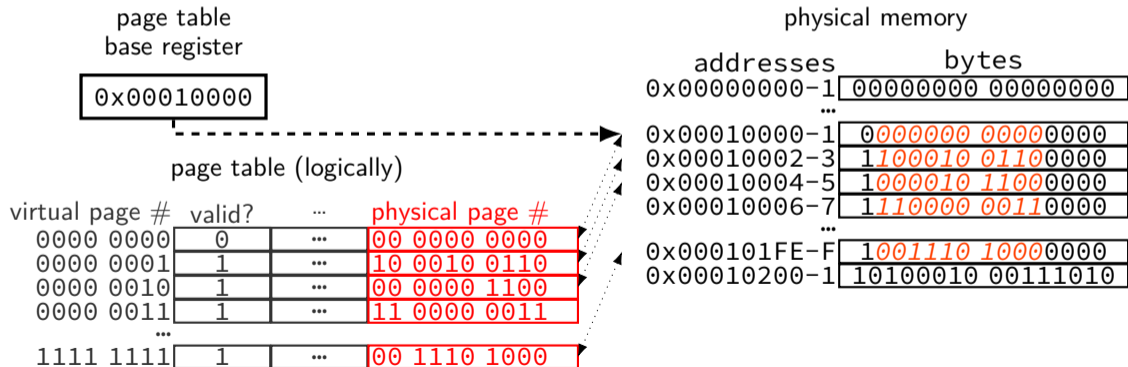


page tables in memory

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)

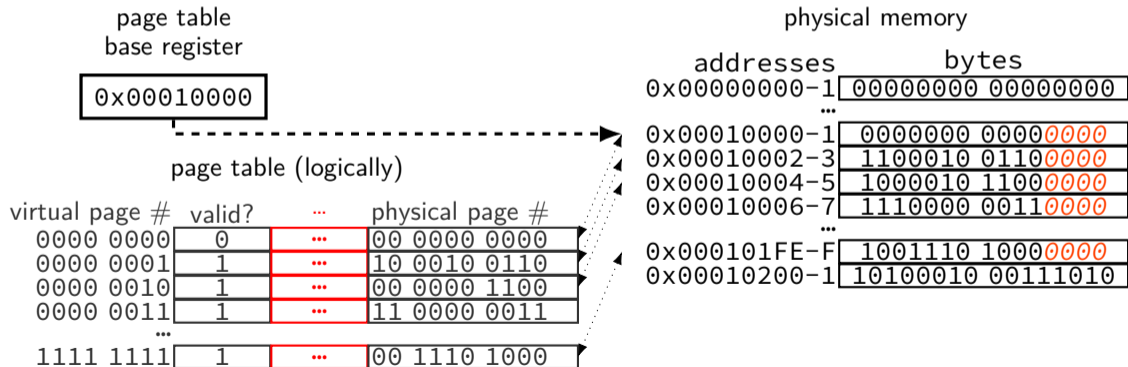


page tables in memory

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)

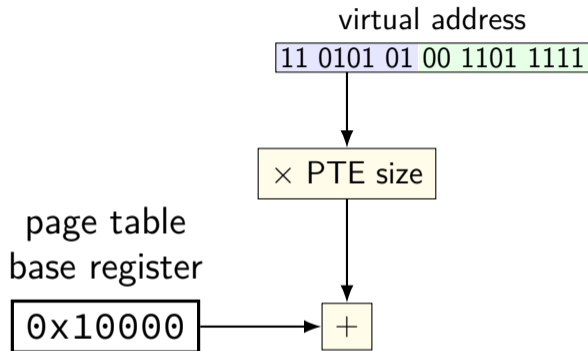


memory access with page table

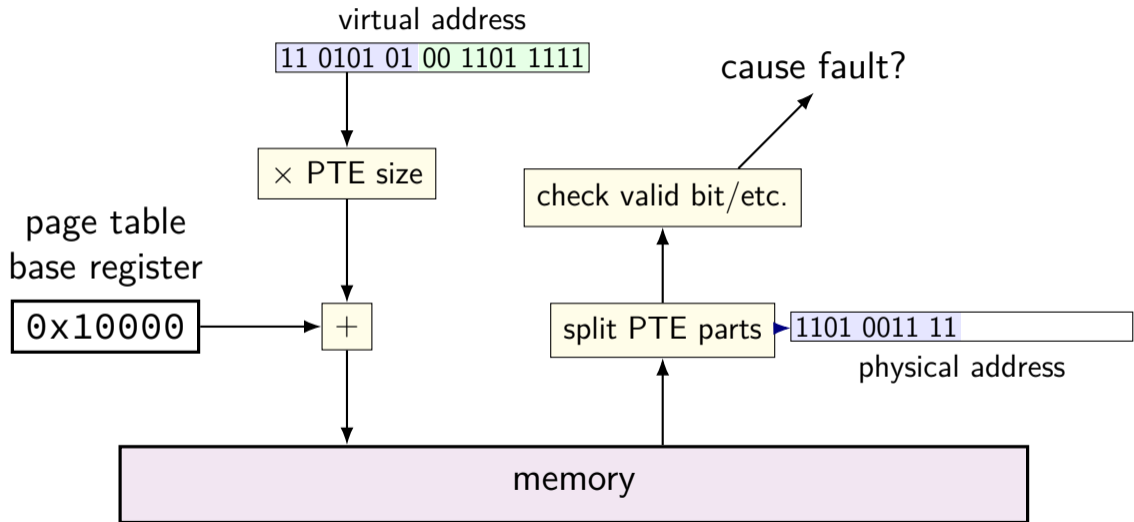
virtual address

11 0101 01 00 1101 1111

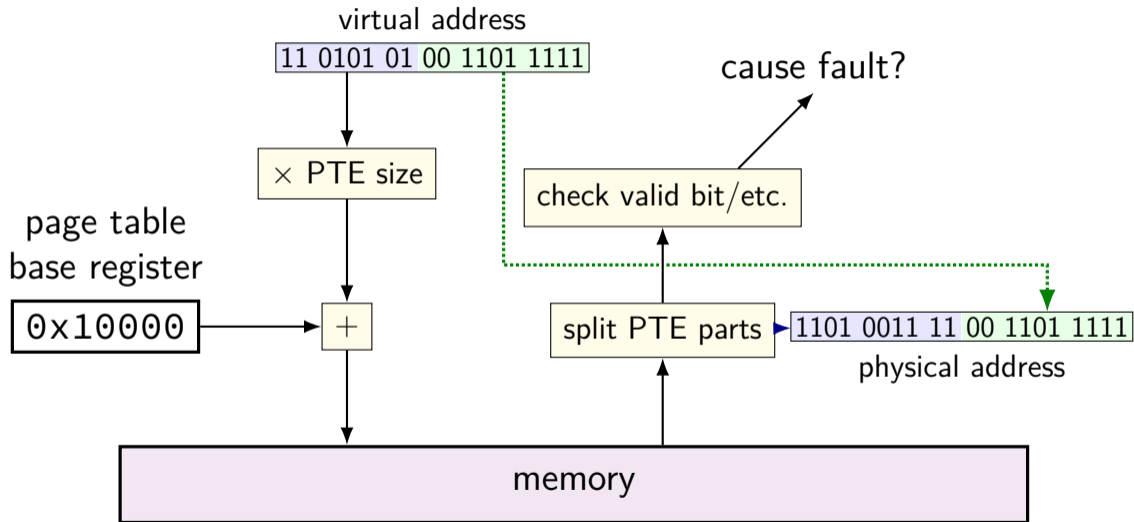
memory access with page table



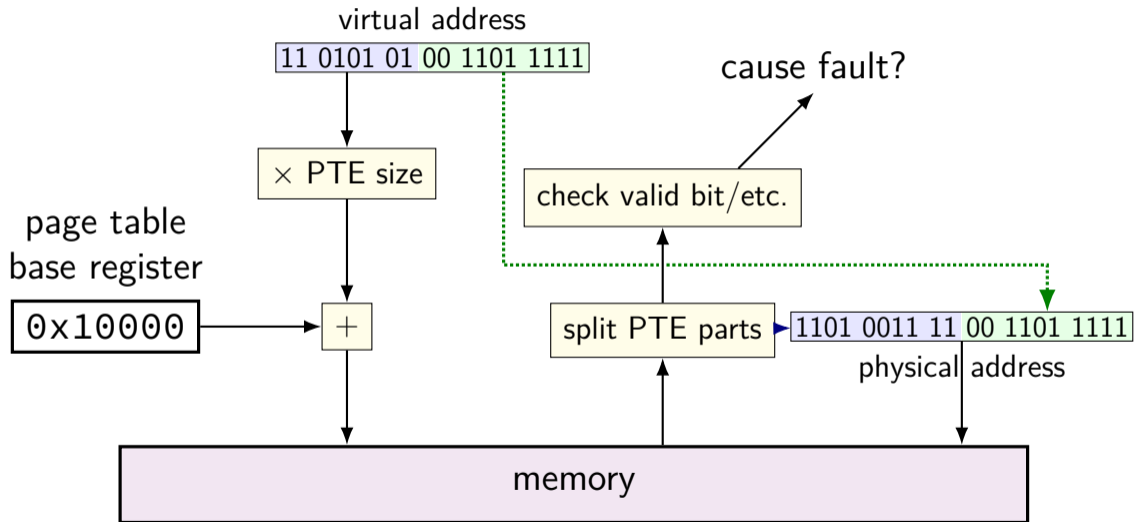
memory access with page table



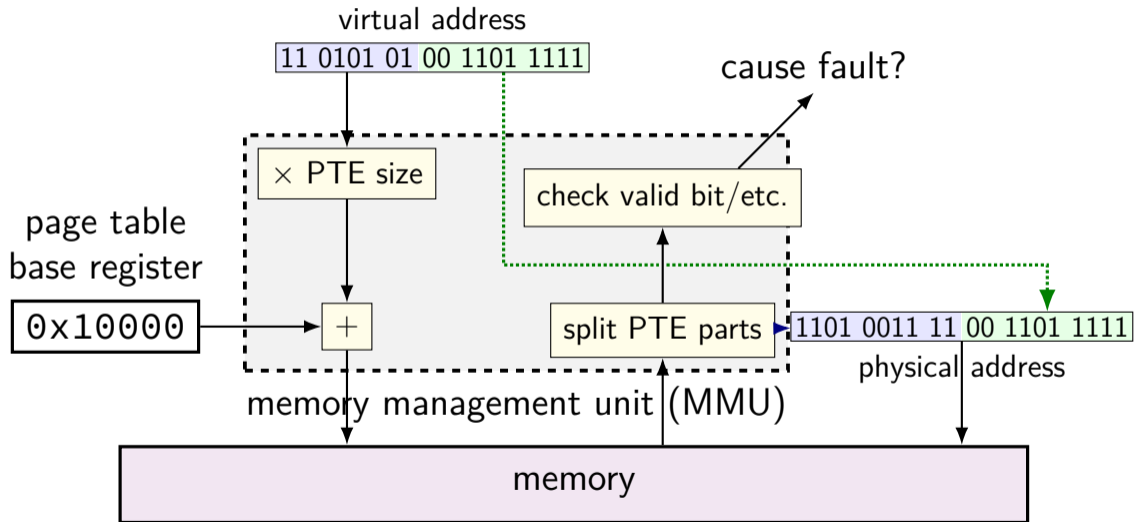
memory access with page table



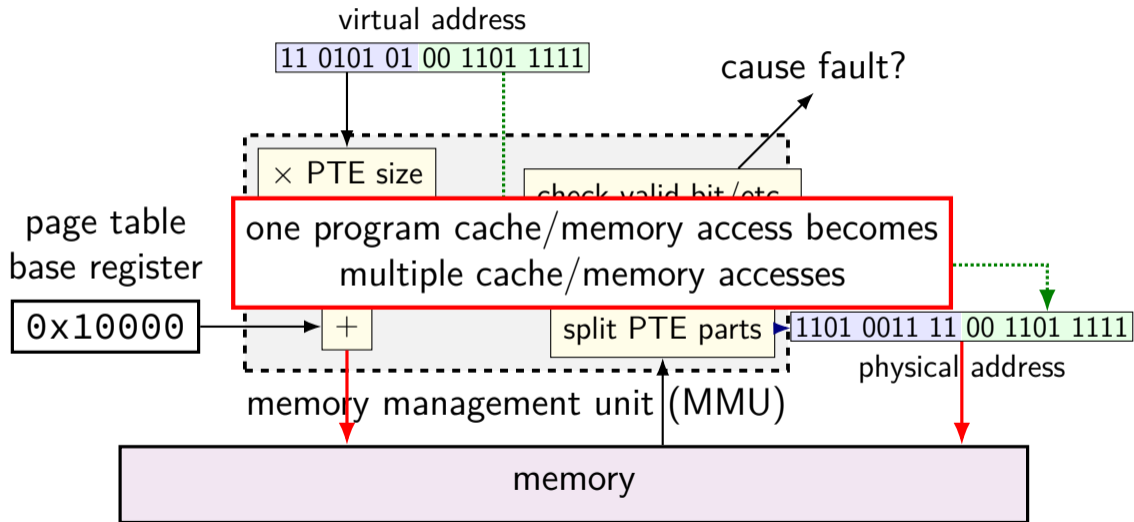
memory access with page table



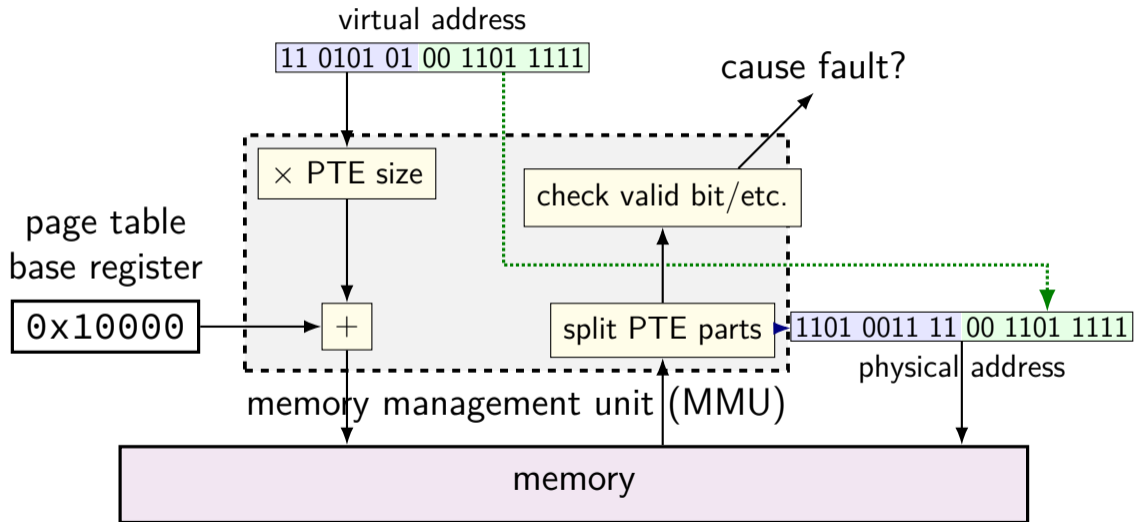
memory access with page table



memory access with page table



memory access with page table



1-level exercise (1)

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	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	1C 2C 3C 4C

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

1-level exercise (1)

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	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	1C 2C 3C 4C

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

0x31 = 11 0001

PTE addr:

0x20 + 110 × 1 = 0x26

PTE value:

0xF6 = 1111 0110

PPN 111, valid 1

M[111 001] = M[0x39]

→ 0x0C

1-level exercise (1)

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	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	1C 2C 3C 4C

physical addresses	bytes
0x20-3	D0 D1 D2 D3
0x24-7	E4 E5 F6 07
0x28-B	89 9A AB BC
0x2C-F	CD DE EF F0
0x30-3	BA 0A BA 0A
0x34-7	CB 0B CB 0B
0x38-B	DC 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

$M[111\ 001] = M[0x39]$

→ 0x0C

1-level exercise (1)

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	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	1C 2C 3C 4C

physical addresses	bytes
0x20-3	D0 D1 D2 D3
0x24-7	E4 E5 F6 07
0x28-B	89 9A AB BC
0x2C-F	CD DE EF F0
0x30-3	BA 0A BA 0A
0x34-7	CB 0B CB 0B
0x38-B	DC 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

$M[111\ 001] = M[0x39]$

→ 0x0C

1-level exercise (1)

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	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	1C 2C 3C 4C

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

0x31 = 11 0001

PTE addr:

0x20 + 110 × 1 = 0x26

PTE value:

0xF6 = 1111 0110

PPN 111, valid 1

M[111 001] = M[0x39]

→ 0x0C

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

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	1C 2C 3C 4C

physical addresses	bytes
0x20-3	A0 E2 D1 F3
0x24-7	E4 E5 F6 07
0x28-B	89 9A AB BC
0x2C-F	CD DE EF F0
0x30-3	BA 0A BA 0A
0x34-7	CB 0B CB 0B
0x38-B	DC 0C DC 0C
0x3C-F	EC 0C EC 0C

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

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	1C 2C 3C 4C

physical addresses	bytes
0x20-3	A0 E2 D1 F3
0x24-7	E4 E5 F6 07
0x28-B	89 9A AB BC
0x2C-F	CD DE EF F0
0x30-3	BA 0A BA 0A
0x34-7	CB 0B CB 0B
0x38-B	DC 0C DC 0C
0x3C-F	EC 0C EC 0C

0x12 = 01 0010

PTE addr:

0x20 + 2 × 1 = 0x22

PTE value:

0xD1 = 1101 0001

PPN 110, valid 1

M[110 010] = M[0x32]

→ 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 $0x12$

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
$0x1C-F$	1C 2C 3C 4C

physical addresses	bytes
$0x20-3$	A0 E2 D1 F3
$0x24-7$	E4 E5 F6 07
$0x28-B$	89 9A AB BC
$0x2C-F$	CD DE EF F0
$0x30-3$	BA 0A BA 0A
$0x34-7$	CB 0B CB 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

$M[110\ 010] = M[0x32]$

$\rightarrow 0xBA$

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

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	1C 2C 3C 4C

physical addresses	bytes
0x20-3	A0 E2 D1 F3
0x24-7	E4 E5 F6 07
0x28-B	89 9A AB BC
0x2C-F	CD DE EF F0
0x30-3	BA 0A BA 0A
0x34-7	CB 0B CB 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

$M[110 \ 010] = M[0x32]$

→ 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

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	1C 2C 3C 4C

physical addresses	bytes
0x20-3	A0 E2 D1 F3
0x24-7	E4 E5 F6 07
0x28-B	89 9A AB BC
0x2C-F	CD DE EF F0
0x30-3	BA 0A BA 0A
0x34-7	CB 0B CB 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

$M[110\ 010] = M[0x32]$

→ 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 ... /* page offset bits */
#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 (~0L) if invalid
size_t 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:

	VPN	valid?	physical
	0	0	—
ptbr = GetPointerToTable(1	1	0x9999
	2	0	—
	3	1	0x3333

translate(0x0FFF) == (void*) ~0L

translate(0x1000) == (void*) 0x9999000

translate(0x1001) == (void*) 0x9999001

translate(0x2000) == (void*) ~0L

translate(0x2001) == (void*) ~0L

translate(0x3000) == (void*) 0x3333000

translate()

with POBITS=12, LEVELS=1:

ptbr = GetPointerToTable(

VPN	valid?	physical
0	0	—
1	1	0x9999
2	0	—
3	1	0x3333
...

)

translate(0x0FFF) == (void*) ~0L

translate(0x1000) == (void*) 0x9999000

translate(0x1001) == (void*) 0x9999001

translate(0x2000) == (void*) ~0L

translate(0x2001) == (void*) ~0L

translate(0x3000) == (void*) 0x3333000

page_allocate()

with POBITS=12, LEVELS=1:

```
ptbr == 0
```

```
page_allocate(0x1000) or page_allocate(0x1001) or ...
```

page_allocate()

with POBITS=12, LEVELS=1:

ptbr == 0

page_allocate(0x1000) or page_allocate(0x1001) or ...

ptbr now == GetPointerToTable(

VPN valid? physical

0	0	—
1	1	(new)
2	0	—
3	0	—
...

allocated with posix_memalign

page_allocate()

with POBITS=12, LEVELS=1:

ptbr == 0

page_allocate(0x1000) or page_allocate(0x1001) or ...

ptbr now == GetPointerToTable(

VPN valid? physical

0	0	—
1	1	(new)
2	0	—
3	0	—
...

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 (28 Feb): LEVELS=1, POBITS=12 and
translate() OR
page_allocate()

part 2 (week after break): all LEVELS, both functions
in preparation for code review
due Weds BEFORE LAB

part 3 (week after break): final submission
Friday after code review
most of grade based on this
will test previous parts again

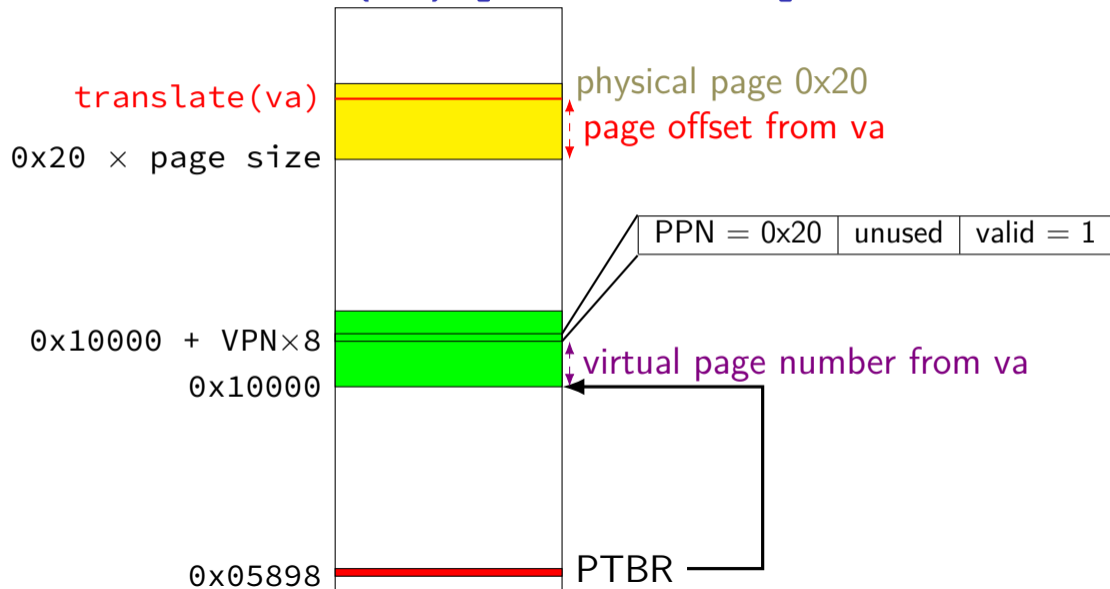
address/page table entry format

(with POBITS=12, LEVELS=1)

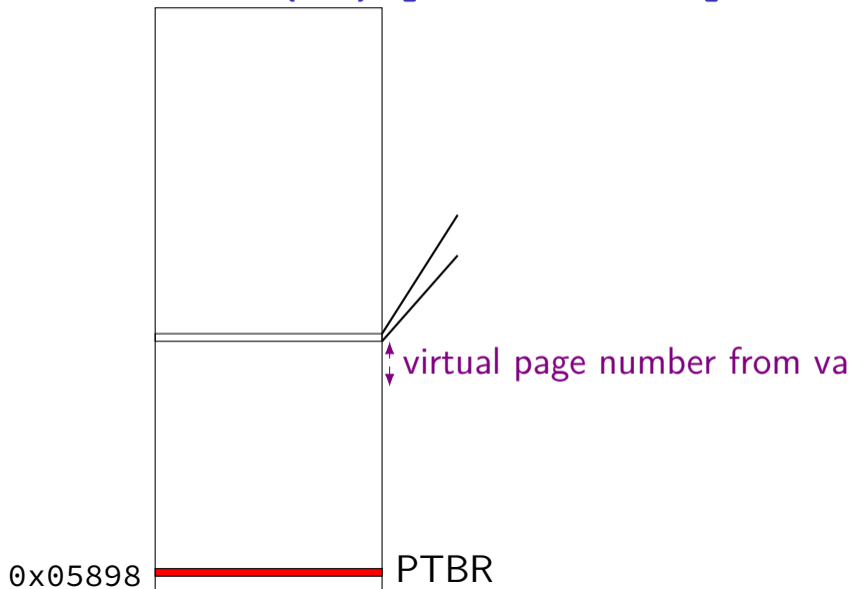
	bits 63-21	bits 20-12	bits 11-1	bit 0
page table entry	physical page number		unused	valid bit
virtual address	unused	virtual page number	page offset	
physical address	physical page number		page offset	

in assignment: value from `posix_memalign` = physical address

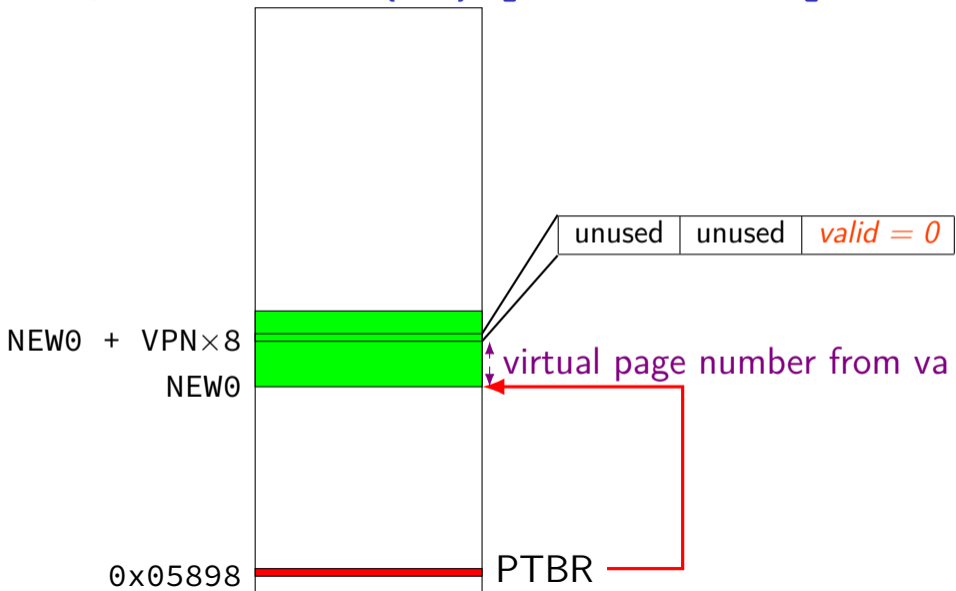
pa = translate(va) [LEVELS=1]



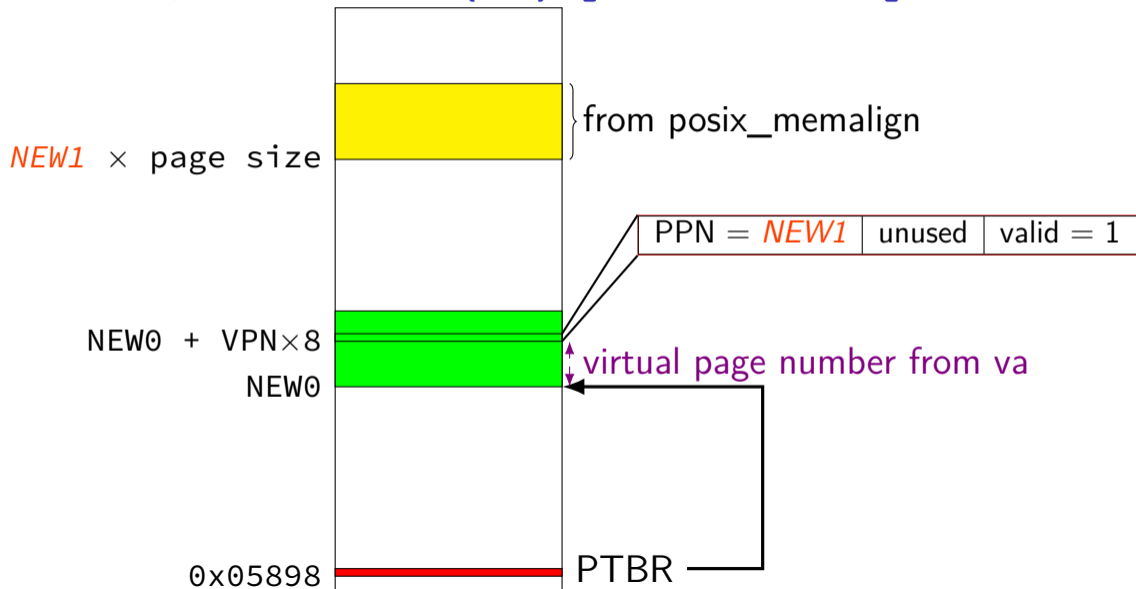
first page_allocate(va) [LEVELS=1]



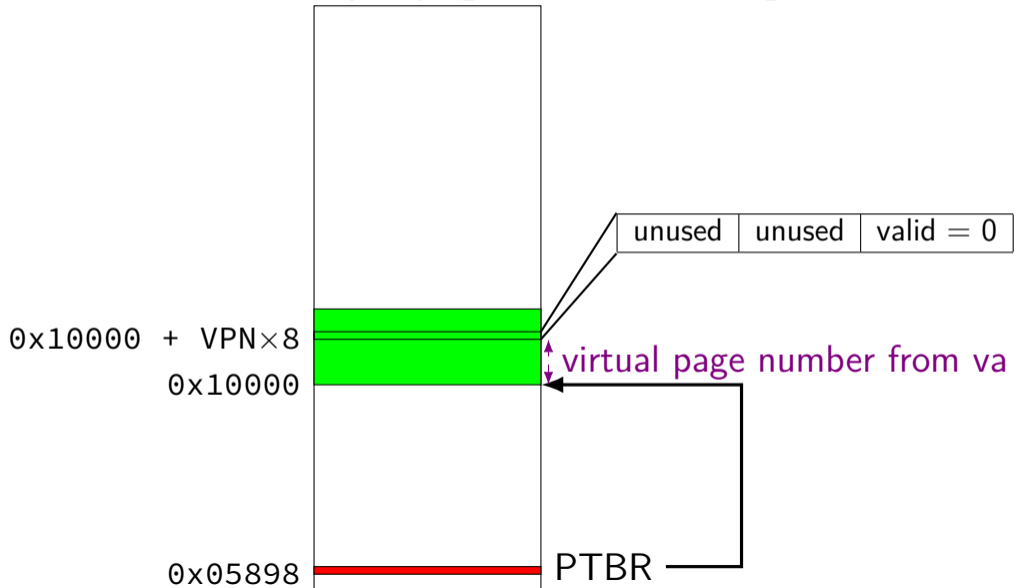
first_page_allocate(va) [LEVELS=1]



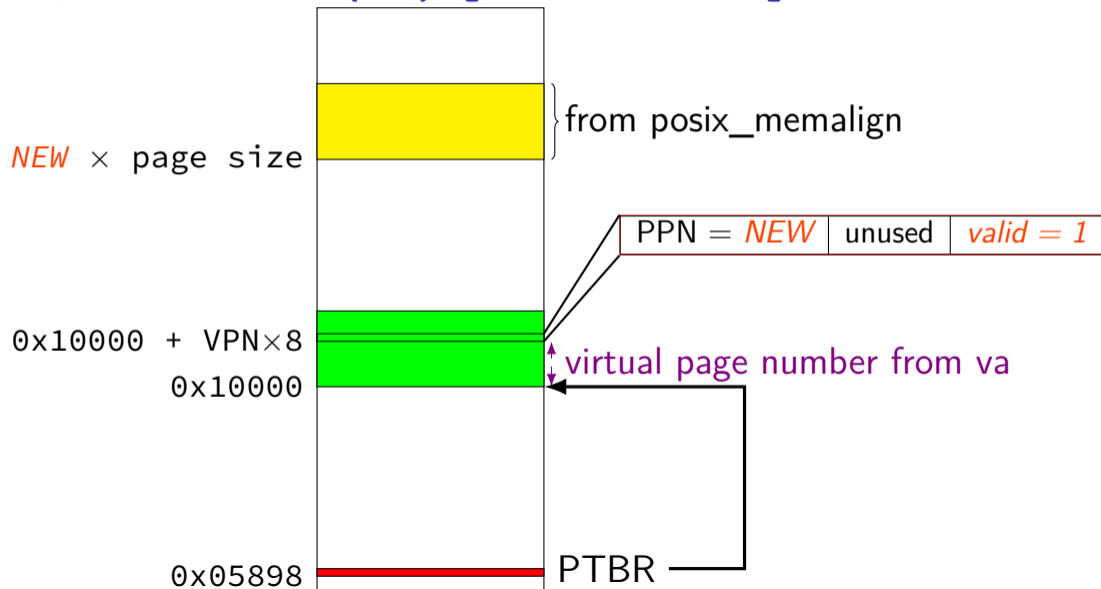
first page_allocate(va) [LEVELS=1]



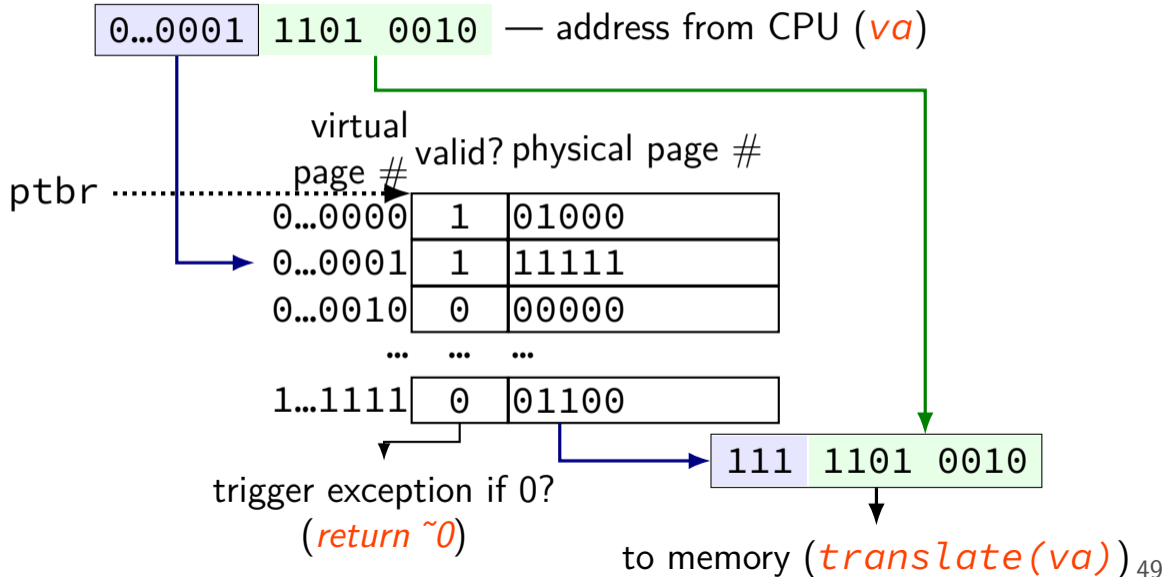
page_allocate(va) [LEVELS=1]



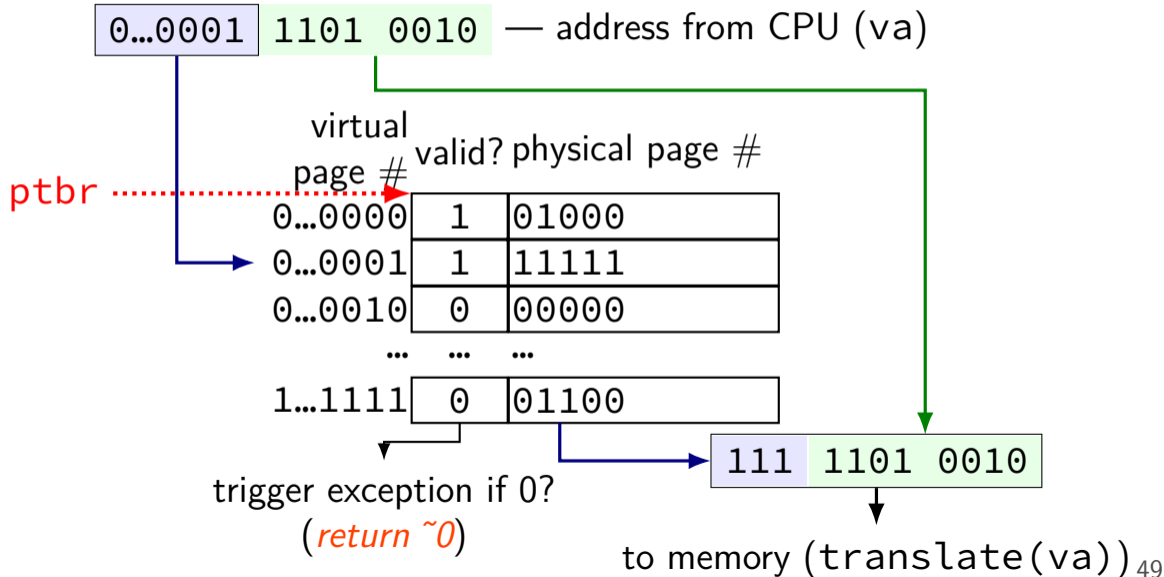
page_allocate(va) [LEVELS=1]



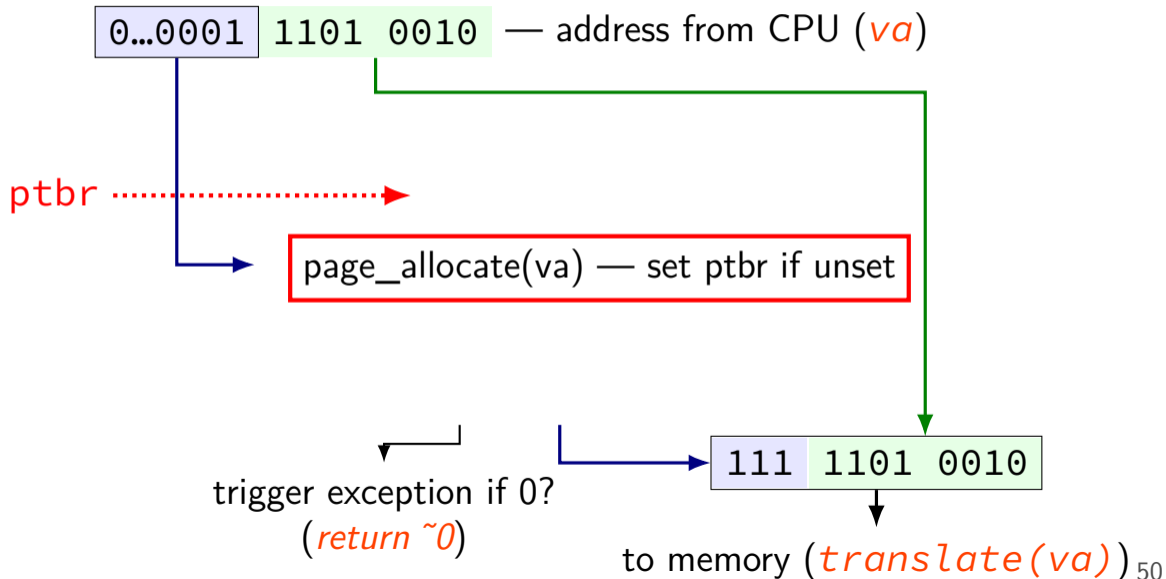
page table lookup (and translate())



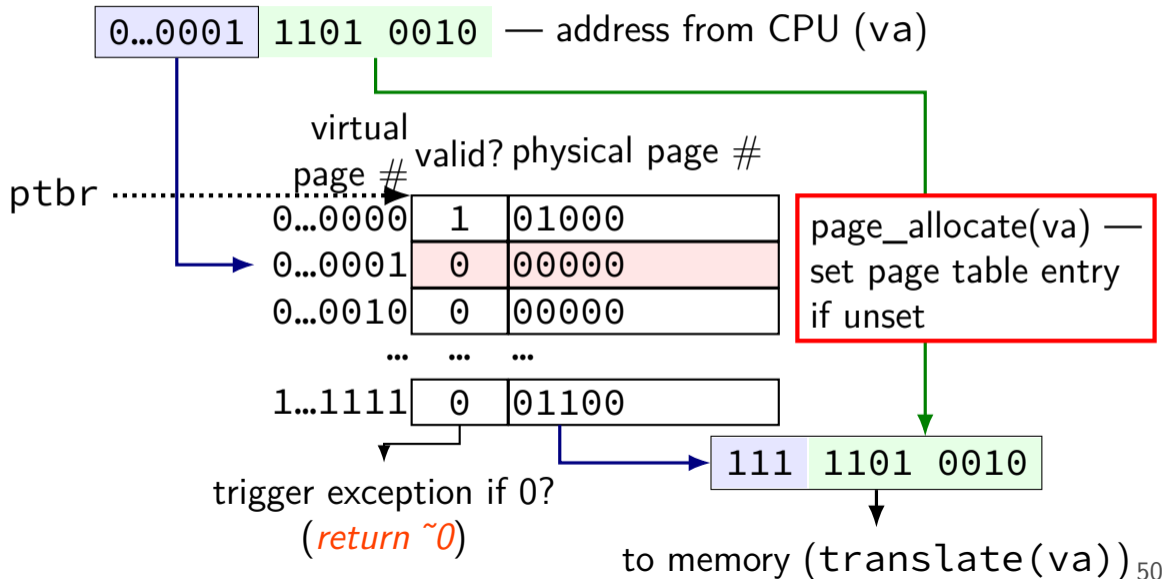
page table lookup (and translate())



page table lookup (and allocate)



page table lookup (and allocate)



exercise: 64-bit system

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

4096 byte pages

exercise: 64-bit system

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

4096 byte pages

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

exercise: 64-bit system

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?

exercise: 64-bit system

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) $2^{48}/2^{12} = 2^{36}$ entries

exercise: how large are physical page numbers? $39 - 12 = 27$ bits

exercise: 64-bit system

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) $2^{48}/2^{12} = 2^{36}$ entries

exercise: how large are physical page numbers? $39 - 12 = 27$ bits

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??)

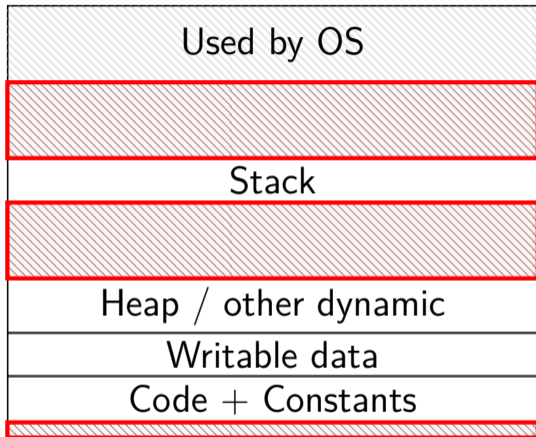
huge page tables

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

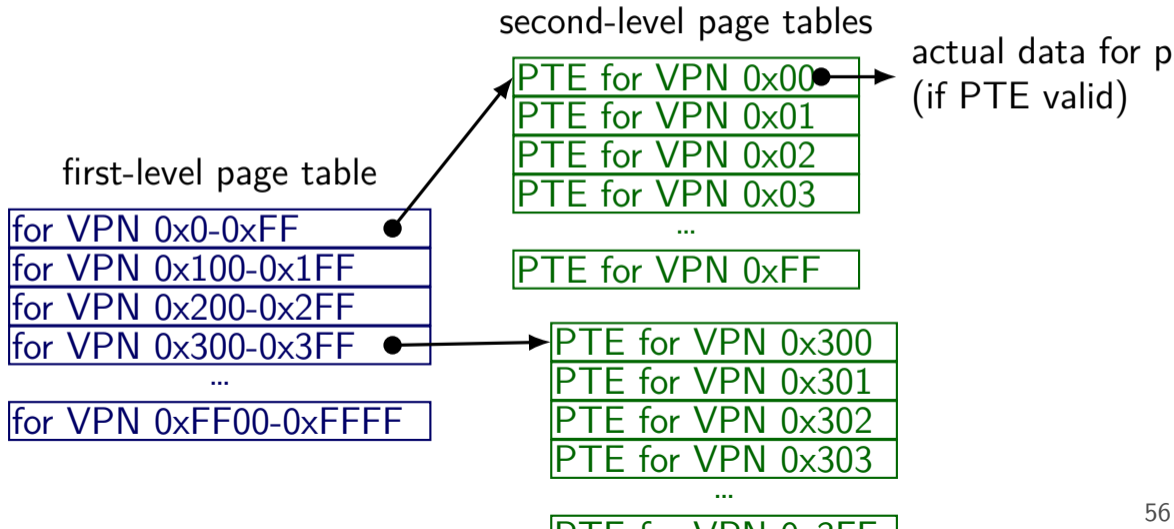
doing two memory accesses is already very slow

solution: tree with many children from each node

(far from binary tree's left/right child)

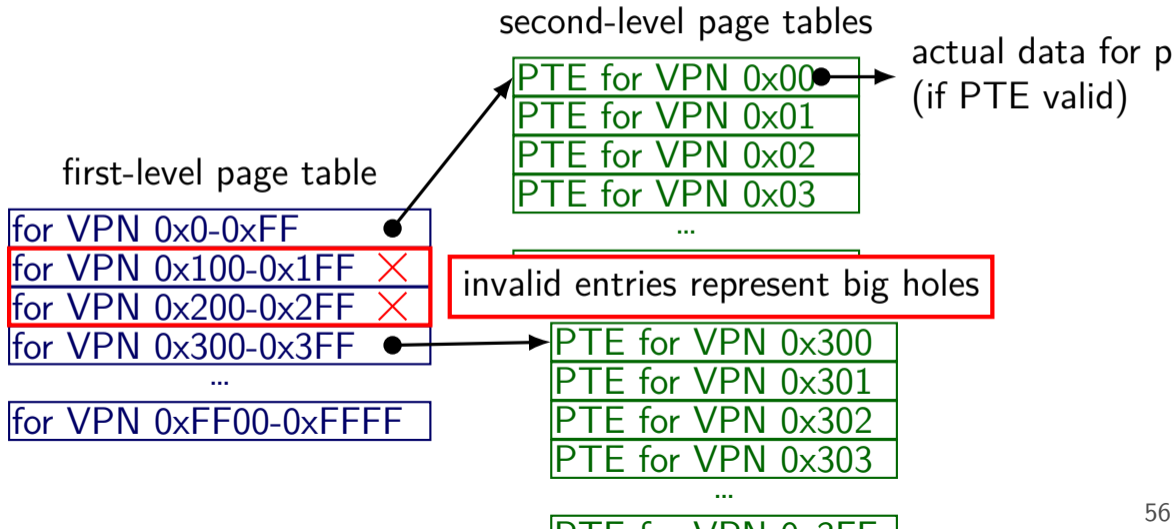
two-level page tables

two-level page tables for 65536 pages (16-bit VPN; 256 entries/table)



two-level page tables

two-level page tables for 65536 pages (16-bit VPN; 256 entries/table)



two-level page tables

two-level page tables for 65536 pages (16-bit VPN: 256 entries/table)

first-level page table

VPN range	valid	...	physical page # (of next page table)
0x0000-0x00FF	1	...	0x22343
0x0100-0x01FF	0	...	0x00000
0x0200-0x02FF	0	...	0x00000
0x0300-0x03FF	1	...	0x33454
0x0400-0x04FF	1	...	0xFF043
...
0xFF00-0xFFFF	1	...	0xFF045

for p
d)

first-level page table for VPN 0x0-0xFF

for VPN 0x100-0x1FF

for VPN 0x200-0x2FF

for VPN 0x300-0x3FF

...

for VPN 0xFF00-0xFFFF

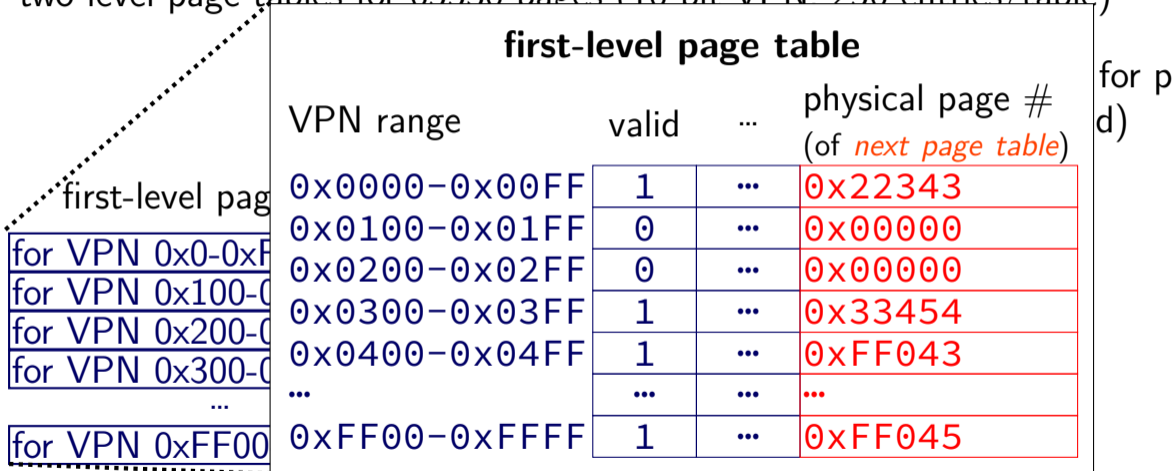
PTE for VPN 0x303

...

PTE for VPN 0x3FF

two-level page tables

two-level page tables for 65536 pages (16-bit VPN: 256 entries/table)



two-level page tables

two-level page tables for 65536 pages (16-bit VPN: 256 entries/table)

	first-level page table			for p d)
	VPN range	valid	... physical page # (of next page table)	
first-level page	0x0000-0x00FF	1	... 0x22343	
for VPN 0x0-0xF	0x0100-0x01FF	0	... 0x00000	
for VPN 0x100-0x10F	0x0200-0x02FF	0	... 0x00000	
for VPN 0x200-0x20F	0x0300-0x03FF	1	... 0x33454	
for VPN 0x300-0x30F	0x0400-0x04FF	1	... 0xFF043	
...	
for VPN 0xFF00-0xFF0F	0xFF00-0xFFFF	1	... 0xFF045	

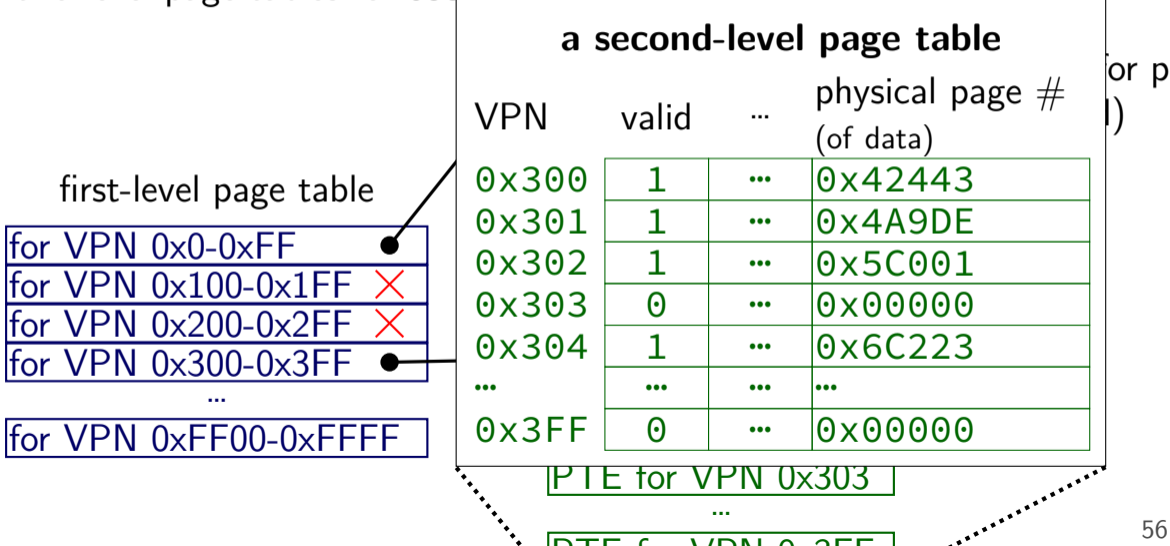
PTE for VPN 0x303

...

PTE for VPN 0x3FF

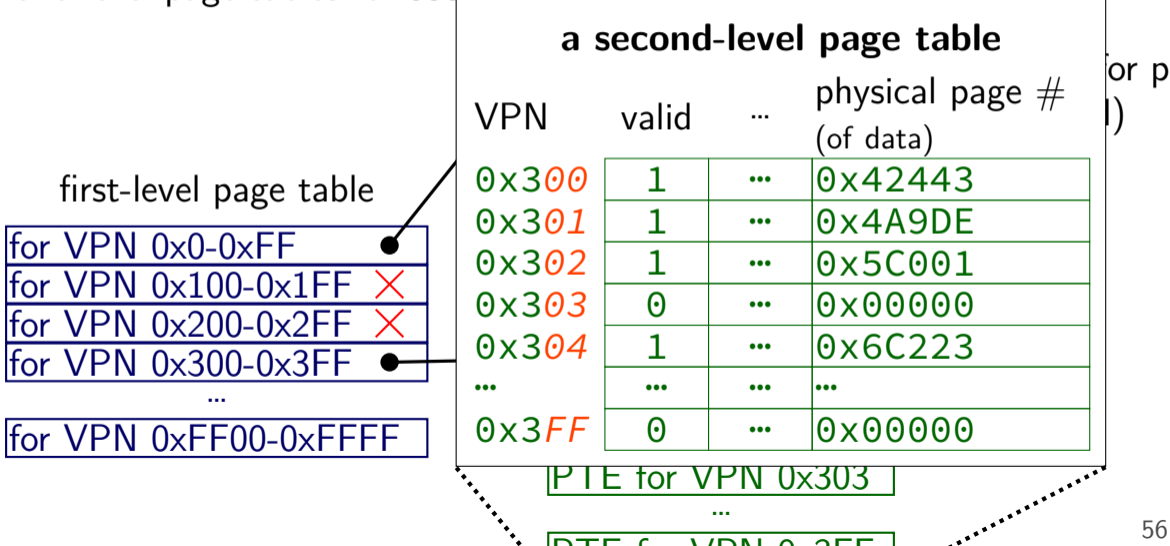
two-level page tables

two-level page tables for 65536 pages (16-bit VPN: 256 entries/table)



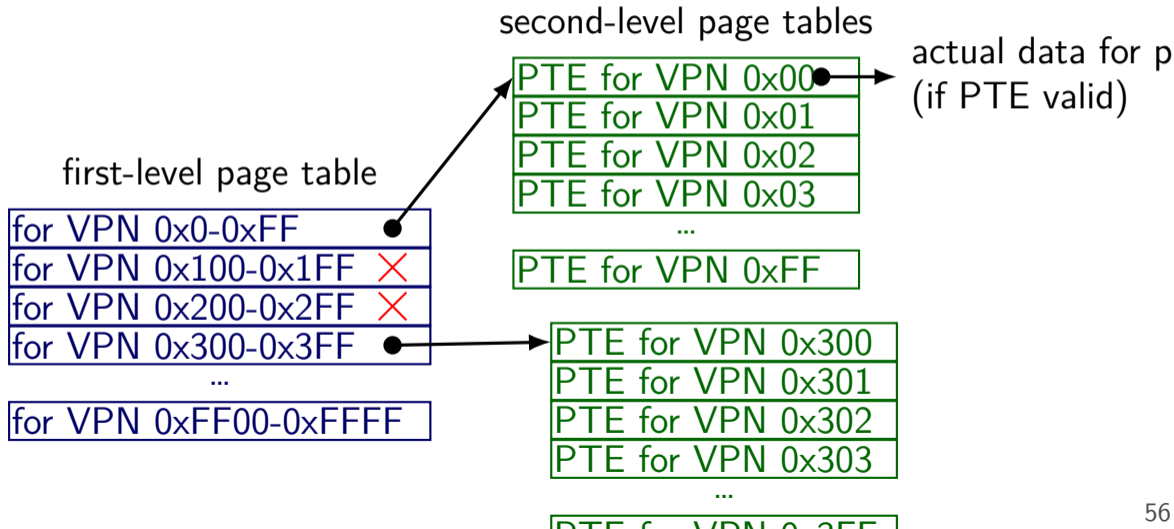
two-level page tables

two-level page tables for 65536 pages (16-bit VPN: 256 entries/table)



two-level page tables

two-level page tables for 65536 pages (16-bit VPN; 256 entries/table)



two-level page table lookup

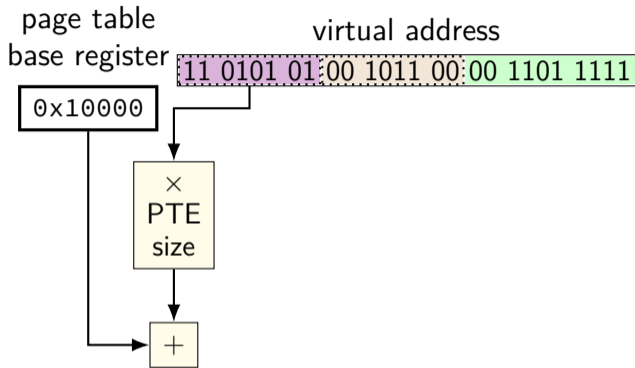
virtual address

11 0101 0100 1011 0000 1101 1111

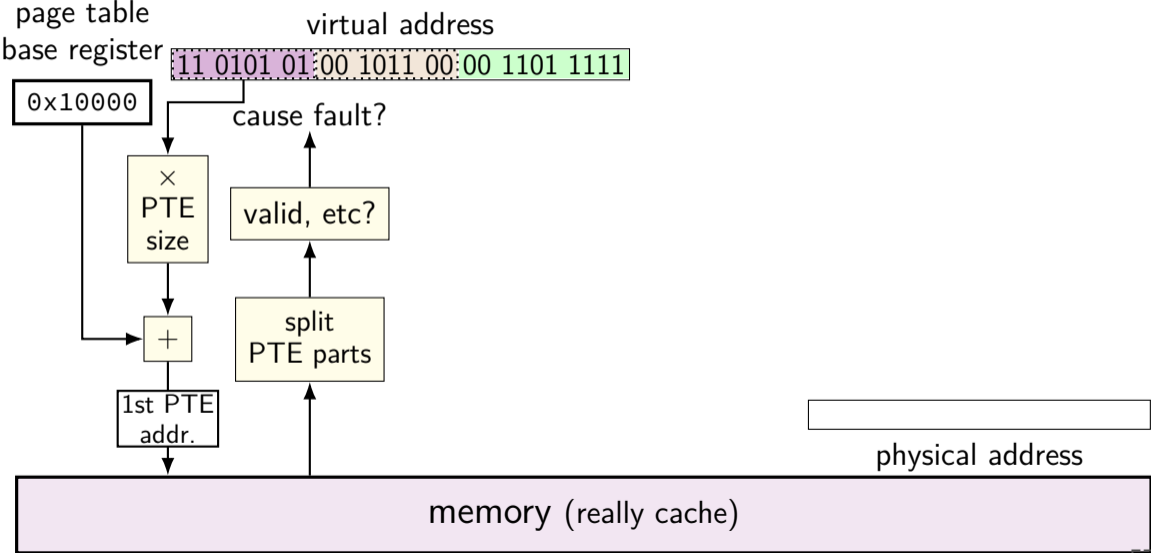
VPN — split into two parts (one per level)

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

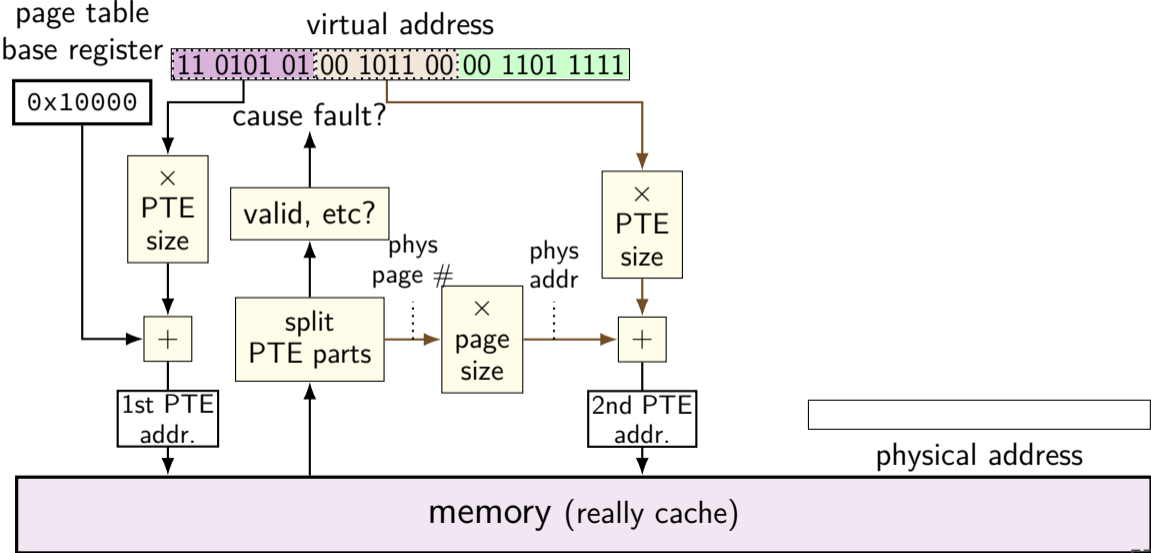
two-level page table lookup



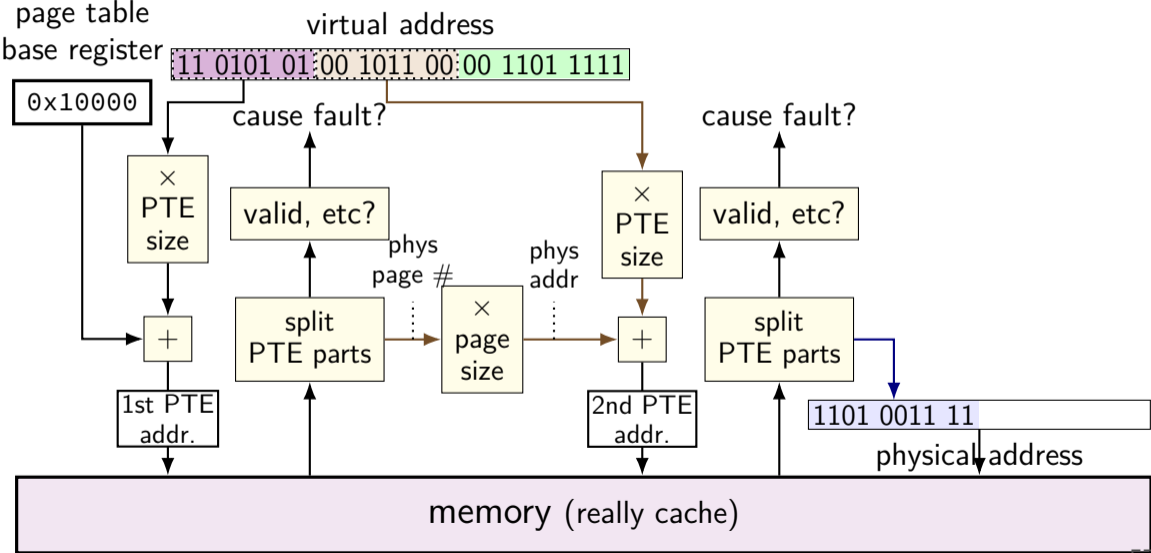
two-level page table lookup



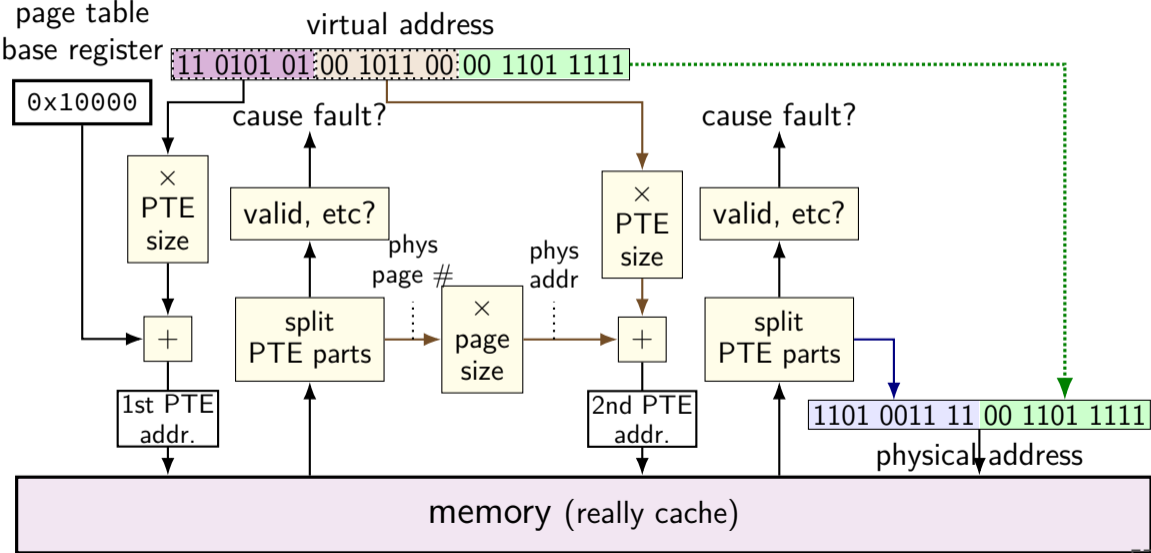
two-level page table lookup



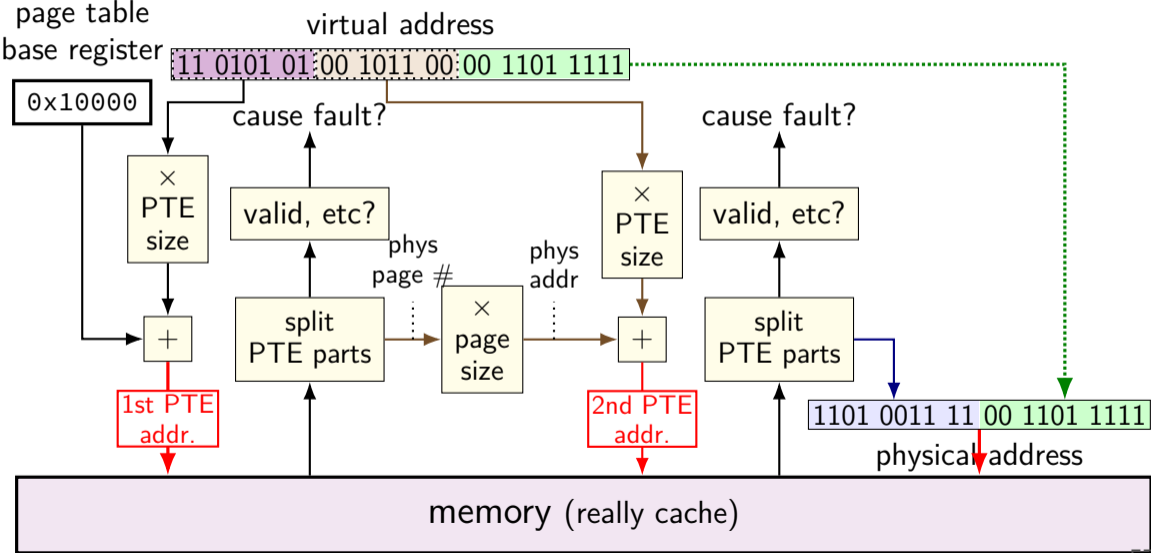
two-level page table lookup



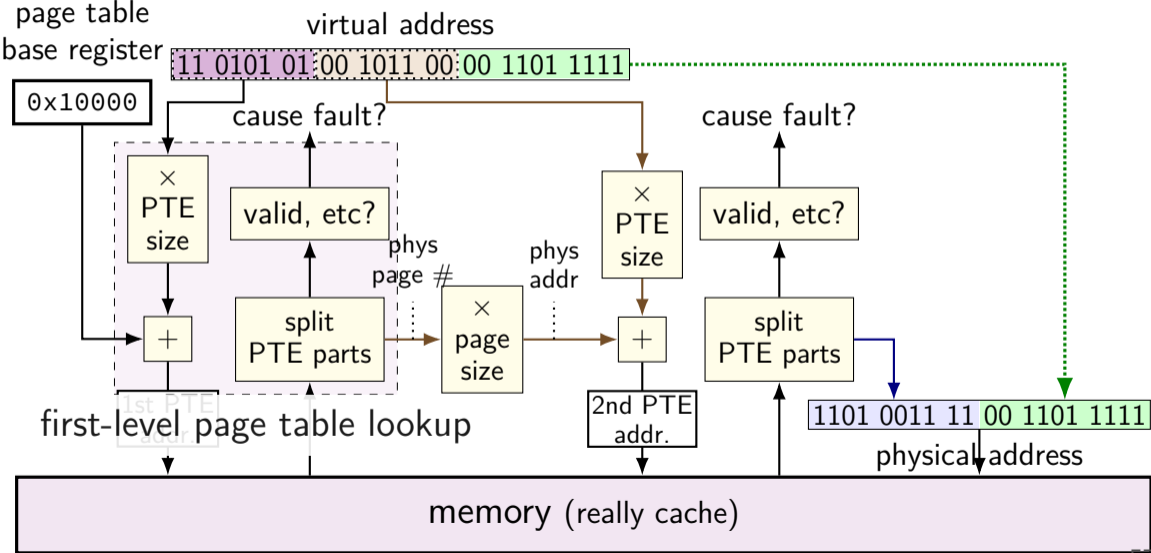
two-level page table lookup



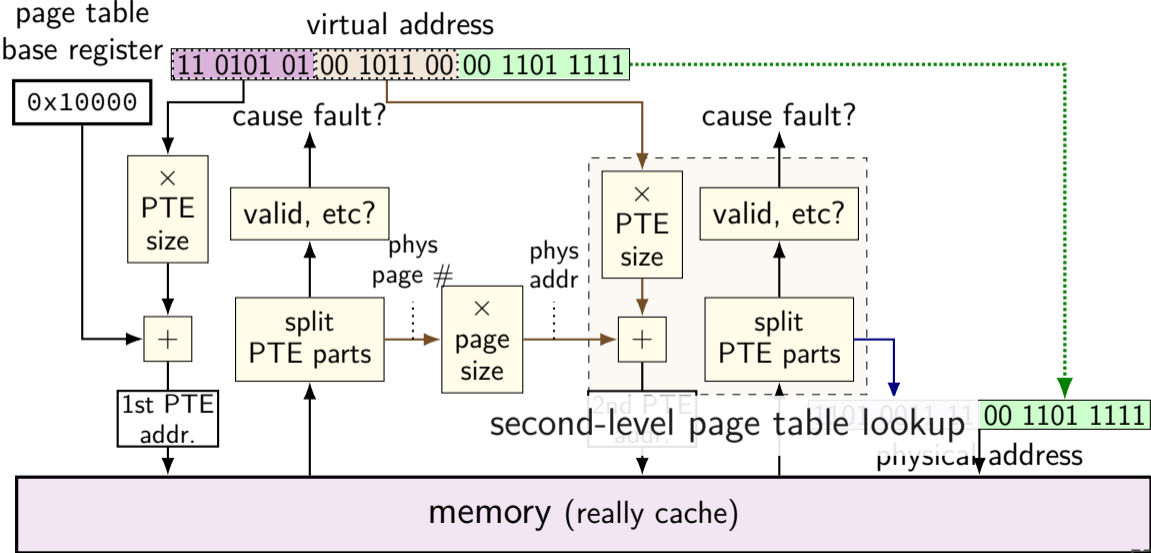
two-level page table lookup



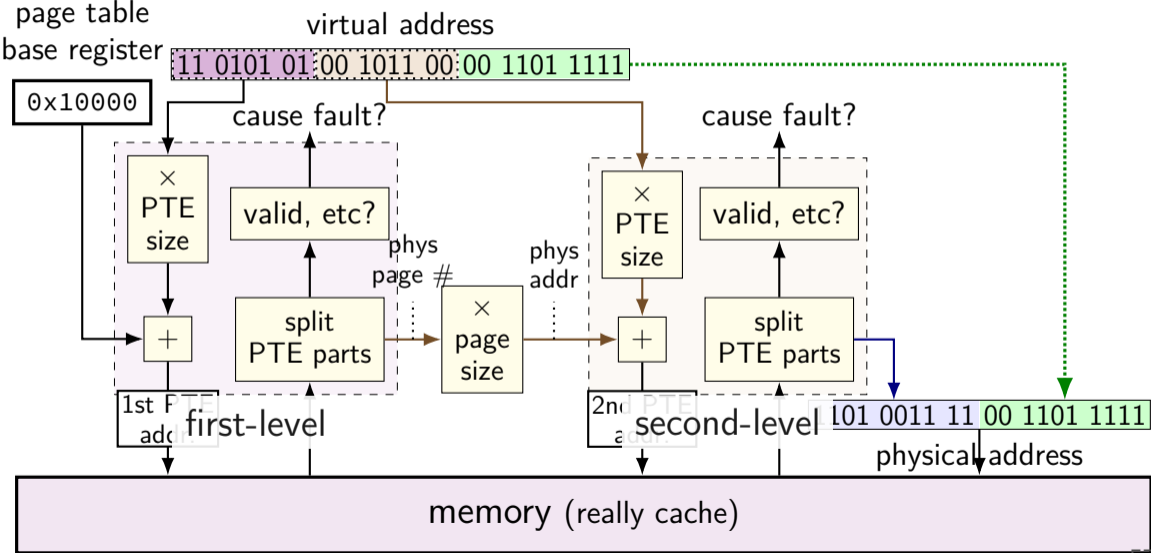
two-level page table lookup



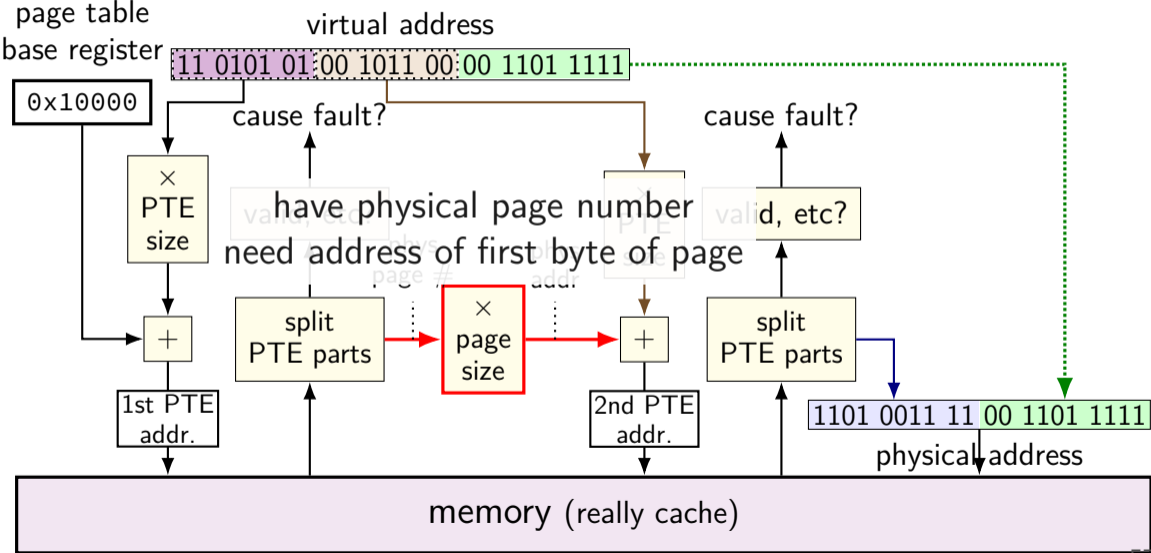
two-level page table lookup



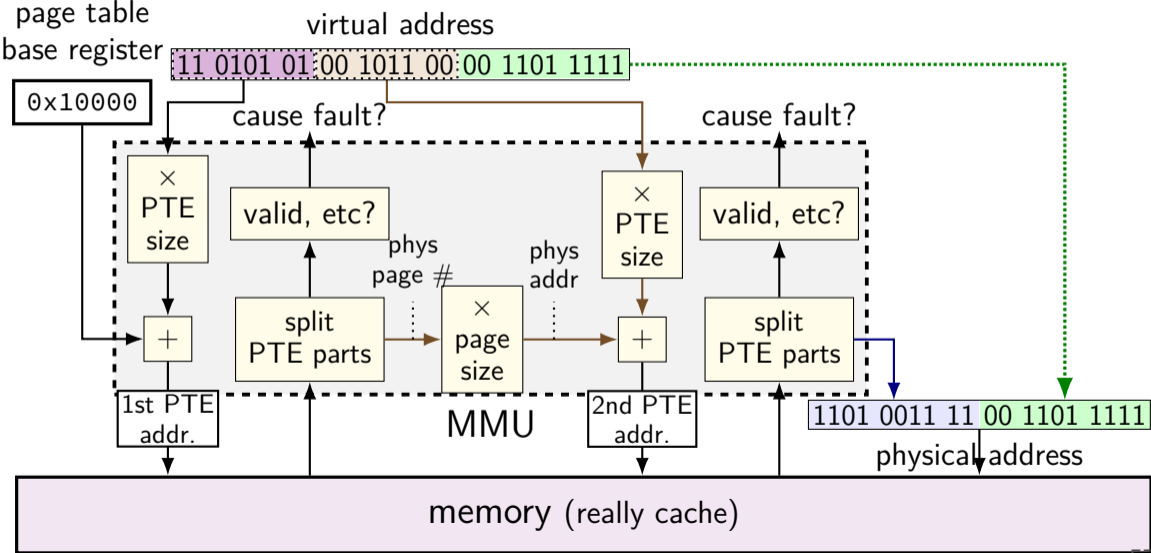
two-level page table lookup



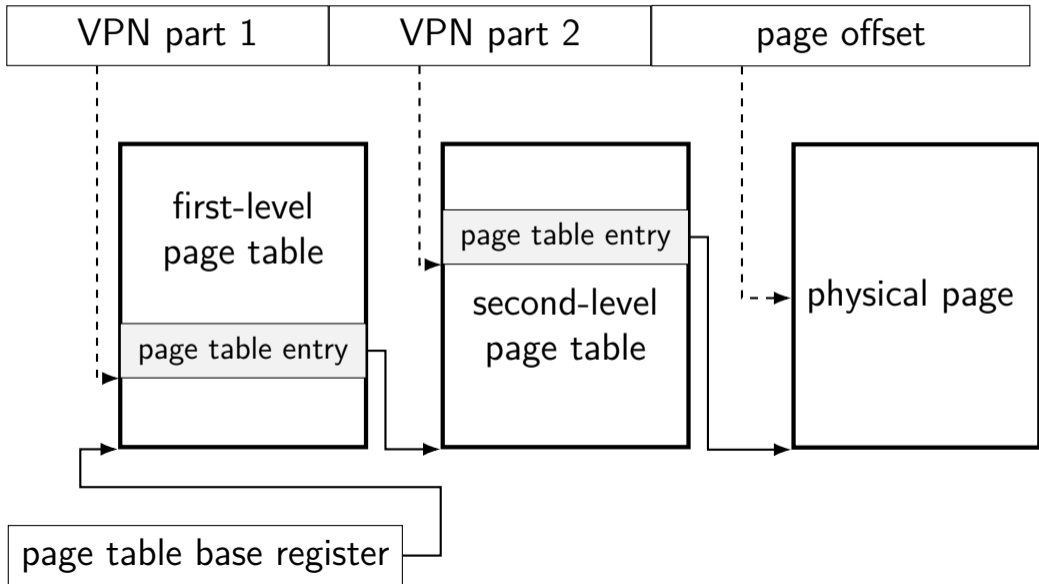
two-level page table lookup



two-level page table lookup



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

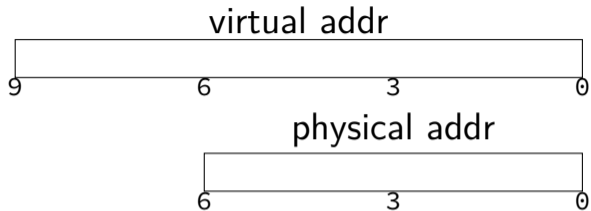
indexes used for lookup *parts of the virtual page number*
(there are not multiple VPNs)

assignment

2-level splitting

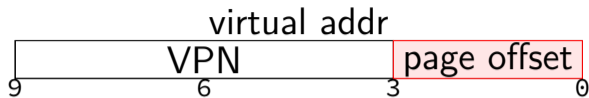
9-bit virtual address

6-bit physical address



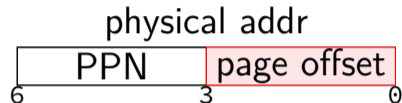
2-level splitting

9-bit virtual address



6-bit physical address

8-byte pages → 3-bit page offset (bottom)

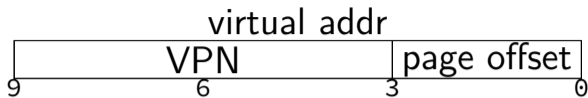


9-bit VA: 6 bit VPN + 3 bit PO

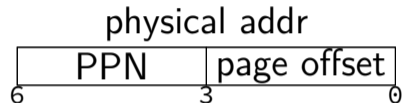
6-bit PA: 3 bit PPN + 3 bit PO

2-level splitting

9-bit virtual address



6-bit physical address



8-byte pages \rightarrow 3-bit page offset (bottom)

9-bit VA: 6 bit VPN + 3 bit PO

6-bit PA: 3 bit PPN + 3 bit PO

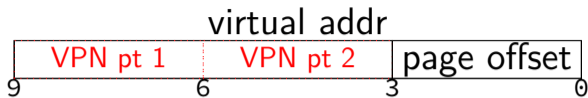
1 page page tables w/ 1 byte entry \rightarrow 8 entry PTs

page table (either level)

	valid? PPN	
0		
1		
2		
...
7		

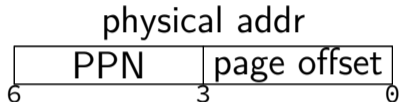
2-level splitting

9-bit virtual address



6-bit physical address

8-byte pages \rightarrow 3-bit page offset (bottom)



9-bit VA: 6 bit VPN + 3 bit PO

page table (either level)

6-bit PA: 3 bit PPN + 3 bit PO

1 page page tables w/ 1 byte entry \rightarrow 8 entry PTs

	valid? PPN	
0		
1		
2		
...
7		

8 entry page tables \rightarrow 3-bit VPN parts

9-bit VA: 3 bit VPN part 1; 3 bit VPN part 2

2-level example

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 0x20; translate virtual address 0x129

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	1C 2C 3C 4C

physical addresses	bytes
0x20-3	00 91 72 13
0x24-7	F4 A5 36 07
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	AC DC DC 0C

2-level example

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 $0x20$; translate virtual address $0x129$

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
$0x1C-F$	1C 2C 3C 4C

physical addresses	bytes
$0x20-3$	00 91 72 13
$0x24-7$	F4 A5 36 07
$0x28-B$	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
$0x3C-F$	AC DC DC 0C

$0x129 = 1\ 0010\ 1001$
 $0x20 + 0x4 \times 1 = 0x24$
PTE 1 value:
 $0xF4 = 1111\ 0100$
PPN 111, valid 1

2-level example

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 $0x20$; translate virtual address $0x129$

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
$0x1C-F$	1C 2C 3C 4C

physical addresses	bytes
$0x20-3$	00 91 72 13
$0x24-7$	F4 A5 36 07
$0x28-B$	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
$0x3C-F$	AC DC DC 0C

$0x129 = 1\ 0010\ 1001$
 $0x20 + 0x4 \times 1 = 0x24$

PTE 1 value:

$0xF4 = 1111\ 0100$

PPN 111, valid 1

PTE 2 addr:

$111\ 000 + 101 \times 1 = 0x3D$

PTE 2 value: $0xDC$

2-level example

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 $0x20$; translate virtual address $0x129$

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
$0x1C-F$	1C 2C 3C 4C

physical addresses	bytes
$0x20-3$	00 91 72 13
$0x24-7$	F4 A5 36 07
$0x28-B$	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
$0x3C-F$	AC DC DC 0C

$0x129 = 1\ 0010\ 1001$

$0x20 + 0x4 \times 1 = 0x24$

PTE 1 value:

$0xF4 = 1111\ 0100$

PPN 111, valid 1

PTE 2 addr:

$111\ 000 + 101 \times 1 = 0x3D$

PTE 2 value: $0xDC$

PPN **110**; valid 1

$M[110\ 001\ (0x31)] = 0x0A$

2-level example

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 $0x20$; translate virtual address $0x129$

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
$0x1C-F$	1C 2C 3C 4C

physical addresses	bytes
$0x20-3$	00 91 72 13
$0x24-7$	F4 A5 36 07
$0x28-B$	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
$0x3C-F$	AC DC DC 0C

$0x129 = 1\ 0010\ 1001$

$0x20 + 0x4 \times 1 = 0x24$

PTE 1 value:

$0xF4 = 1111\ 0100$

PPN 111, valid 1

PTE 2 addr:

$111\ 000 + 101 \times 1 = 0x3D$

PTE 2 value: $0xDC$

PPN 110; valid 1

$M[110\ 001\ (0x31)] = 0x0A$

2-level example

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 $0x20$; translate virtual address $0x129$

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
$0x1C-F$	1C 2C 3C 4C

physical addresses	bytes
$0x20-3$	00 91 72 13
$0x24-7$	F4 A5 36 07
$0x28-B$	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
$0x3C-F$	AC DC DC 0C

$0x129 = 1\ 0010\ 1001$

$0x20 + 0x4 \times 1 = 0x24$

PTE 1 value:

$0xF4 = 1111\ 0100$

PPN 111, valid 1

PTE 2 addr:

$111\ 000 + 101 \times 1 = 0x3D$

PTE 2 value: $0xDC$

PPN 110; valid 1

$M[110\ 001\ (0x31)] = 0x0A$

2-level exercise (1)

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 0x0FB

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	1C 2C 3C 4C

physical addresses	bytes
0x20-3	D0 D1 D2 D3
0x24-7	D4 D5 D6 D7
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

2-level exercise (1)

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 $0x0FB$

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
$0x1C-F$	1C 2C 3C 4C

physical addresses	bytes
$0x20-3$	D0 D1 D2 D3
$0x24-7$	D4 D5 D6 D7
$0x28-B$	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

$0x0F3 = 011\ 111\ 011$
(PTE 1 addr: $0x08 +$
PTE size times $011\ (3)$)
PTE 1: $0xBB$ at $0x0B$
PTE 1: PPN $101\ (5)$ valid 1
PTE 2: $0xF0$ at $0x2F$
PTE 2: PPN $111\ (7)$ valid 1
 $111\ 011 = 0x3B \rightarrow 0x0C$

2-level exercise (1)

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 $0x0FB$

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
$0x1C-F$	1C 2C 3C 4C

physical addresses	bytes
$0x20-3$	D0 D1 D2 D3
$0x24-7$	D4 D5 D6 D7
$0x28-B$	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

$0x0F3 = 011\ 111\ 011$
(PTE 1 addr: $0x08 +$
PTE size times $011\ (3)$)
*PTE 1: **0xBB** at $0x0B$*
PTE 1: PPN $101\ (5)$ valid 1
PTE 2: $0xF0$ at $0x2F$
PTE 2: PPN $111\ (7)$ valid 1
 $111\ 011 = 0x3B \rightarrow 0x0C$

2-level exercise (1)

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 $0x0FB$

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
$0x1C-F$	1C 2C 3C 4C

physical addresses	bytes
$0x20-3$	D0 D1 D2 D3
$0x24-7$	D4 D5 D6 D7
$0x28-B$	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

$0x0F3 = 011\ 111\ 011$
(PTE 1 addr: $0x08 +$
PTE size times $011\ (3)$)
PTE 1: $0xBB$ at $0x0B$
PTE 1: PPN $101\ (5)$ valid 1
PTE 2: $0xF0$ at $0x2F$
PTE 2: PPN $111\ (7)$ valid 1
 $111\ 011 = 0x3B \rightarrow 0x0C$

2-level exercise (1)

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 0x0FB

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	1C 2C 3C 4C

physical addresses	bytes
0x20-3	D0 D1 D2 D3
0x24-7	D4 D5 D6 D7
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

0x0F3 = 011 111 011
(PTE 1 addr: 0x08 +
PTE size times 011 (3))
PTE 1: 0xBB at 0x0B
PTE 1: PPN 101 (5) valid 1
PTE 2: 0xF0 at 0x2F
PTE 2: PPN 111 (7) valid 1
111 011 = 0x3B → 0x0C

2-level exercise (2)

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 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 5A 4A
0x14-7	1B 2B 3B 4B
0x18-B	1C 2C 3C 4C
0x1C-F	1C 2C 3C 4C

physical
addresses bytes

0x20-3	D0 D1 D2 D3
0x24-7	D4 D5 D6 D7
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

2-level exercise (2)

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	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 5A 4A
0x14-7	1B 2B 3B 4B
0x18-B	1C 2C 3C 4C
0x1C-F	1C 2C 3C 4C

physical addresses	bytes
0x20-3	D0 D1 D2 D3
0x24-7	D4 D5 D6 D7
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

0x109 = 100 001 001
(PTE 1 at:
0x10 + PTE size times 4 (100))
PTE 1: 0x1B at 0x14
PTE 1: PPN 000 (0) valid 1
(second table at:
0 (000) times page size = 0x00)
PTE 2: 0x11 at 0x01
PTE 2: PPN 000 (0) valid 1
000 001 = 0x01 → 0x11

2-level exercise (2)

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	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 5A 4A
0x14-7	1B 2B 3B 4B
0x18-B	1C 2C 3C 4C
0x1C-F	1C 2C 3C 4C

physical addresses	bytes
0x20-3	D0 D1 D2 D3
0x24-7	D4 D5 D6 D7
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

0x109 = 100 001 001
(PTE 1 at:
0x10 + PTE size times 4 (100))
PTE 1: 0x1B at 0x14
PTE 1: PPN 000 (0) valid 1
(second table at:
0 (000) times page size = 0x00)
PTE 2: 0x11 at 0x01
PTE 2: PPN 000 (0) valid 1
000 001 = 0x01 → 0x11

2-level exercise (2)

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	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 5A 4A
0x14-7	1B 2B 3B 4B
0x18-B	1C 2C 3C 4C
0x1C-F	1C 2C 3C 4C

physical addresses	bytes
0x20-3	D0 D1 D2 D3
0x24-7	D4 D5 D6 D7
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

0x109 = 100 001 001
(PTE 1 at:
0x10 + PTE size times 4 (100))
PTE 1: 0x1B at 0x14
PTE 1: PPN 000 (0) valid 1
(second table at:
0 (000) times page size = 0x00)
PTE 2: 0x11 at 0x01
PTE 2: PPN 000 (0) valid 1
000 001 = 0x01 → 0x11

2-level exercise (2)

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	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 5A 4A
$0x14-7$	1B 2B 3B 4B
$0x18-B$	1C 2C 3C 4C
$0x1C-F$	1C 2C 3C 4C

physical addresses	bytes
$0x20-3$	D0 D1 D2 D3
$0x24-7$	D4 D5 D6 D7
$0x28-B$	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

$0x109 = 100\ 001\ 001$
(PTE 1 at:
 $0x10 + \text{PTE size times } 4 (100))$
PTE 1: $0x1B$ at $0x14$
PTE 1: PPN $000 (0)$ valid 1
(second table at:
 $0 (000)$ times page size = $0x00$)
PTE 2: $0x11$ at $0x01$
PTE 2: PPN $000 (0)$ valid 1
 $000\ 001 = 0x01 \rightarrow 0x11$

2-level exercise (3)

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
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	1C 2C 3C 4C

physical addresses	bytes
0x20-3	D0 D1 D2 D3
0x24-7	D4 D5 D6 D7
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

2-level exercise (3)

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
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	1C 2C 3C 4C

physical addresses	bytes
0x20-3	D0 D1 D2 D3
0x24-7	D4 D5 D6 D7
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

0x0F3 = 000 001 011

PTE 1: 0x88 at 0x08

PTE 1: PPN 100 (5) valid 0
page fault!

2-level exercise (3)

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
$0x00-3$	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
$0x1C-F$	1C 2C 3C 4C

physical addresses	bytes
$0x20-3$	D0 D1 D2 D3
$0x24-7$	D4 D5 D6 D7
$0x28-B$	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

$0x0F3 = 000\ 001\ 011$

PTE 1: $0x88$ at $0x08$

PTE 1: PPN 100 (5) valid 0
page fault!

2-level exercise (4)

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	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	1C 2C 3C 4C

physical addresses	bytes
0x20-3	D0 D1 D2 D3
0x24-7	D4 D5 D6 D7
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

2-level exercise (4)

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	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	1C 2C 3C 4C

physical addresses	bytes
0x20-3	D0 D1 D2 D3
0x24-7	D4 D5 D6 D7
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

0x1CB = 111 001 011
PTE 1: 0xFF at 0x0F
PTE 1: PPN 111 (7) valid 1
PTE 2: 0x0C at 0x39
PTE 2: PPN 000 (0) valid 0
page fault!

2-level exercise (4)

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	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	1C 2C 3C 4C

physical addresses	bytes
0x20-3	D0 D1 D2 D3
0x24-7	D4 D5 D6 D7
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

0x1CB = 111 001 011

PTE 1: 0xFF at 0x0F

PTE 1: PPN 111 (7) valid 1

PTE 2: 0x0C at 0x39

PTE 2: PPN 000 (0) valid 0
page fault!

2-level exercise (4)

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	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	1C 2C 3C 4C

physical addresses	bytes
0x20-3	D0 D1 D2 D3
0x24-7	D4 D5 D6 D7
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

0x1CB = 111 001 011
PTE 1: 0xFF at 0x0F
PTE 1: PPN 111 (7) valid 1
PTE 2: 0x0C at 0x39
PTE 2: PPN 000 (0) valid 0
page fault!

2-level exercise (5)

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

page table base register $0x10$; translate virtual address $0x376$

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
$0x1C-F$	AC BC DC EC

physical
addresses bytes

$0x20-3$	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

2-level exercise (5)

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

page table base register $0x10$; translate virtual address $0x376$

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
$0x1C-F$	AC BC DC EC

physical addresses	bytes
$0x20-3$	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

$0x376 = 110\ 111\ 0110$

PTE 1: $0x10 + 6 \times 2 = 0x1C$:
AC BC

PTE 1: PPN 10 valid 1

PTE 2: $0x20 + 7 \times 2 = 0x2E$:
EF F0

PTE 2: PPN 11 valid 1

11 0110 = $0x36 \rightarrow$ DB

2-level exercise (5)

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

page table base register $0x10$; translate virtual address $0x376$

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
$0x1C-F$	AC BC DC EC

physical addresses	bytes
$0x20-3$	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

$0x376 = 110\ 111\ 0110$

PTE 1: $0x10 + 6 \times 2 = 0x1C$:
AC BC

PTE 1: PPN 10 valid 1

PTE 2: $0x20 + 7 \times 2 = 0x2E$:
EF F0

PTE 2: PPN 11 valid 1

$11\ 0110 = 0x36 \rightarrow DB$

2-level exercise (5)

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

page table base register $0x10$; translate virtual address $0x376$

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
$0x1C-F$	AC BC DC EC

physical addresses	bytes
$0x20-3$	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

$0x376 = 110\ 111\ 0110$

PTE 1: $0x10 + 6 \times 2 = 0x1C$:

AC BC

PTE 1: PPN 10 valid 1

PTE 2: $0x20 + 7 \times 2 = 0x2E$:

EF F0

PTE 2: PPN 11 valid 1

11 0110 = $0x36 \rightarrow DB$

2-level exercise (5)

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

page table base register $0x10$; translate virtual address $0x376$

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
$0x1C-F$	AC BC DC EC

physical addresses	bytes
$0x20-3$	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

$0x376 = 110$ **111** 0110

PTE 1: $0x10 + 6 \times 2 = 0x1C$:
AC BC

PTE 1: PPN 10 valid 1

PTE 2: $0x20 + 7 \times 2 = 0x2E$:
EF F0

PTE 2: PPN 11 valid 1

11 $0110 = 0x36 \rightarrow$ DB

2-level exercise (5)

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

page table base register $0x10$; translate virtual address $0x376$

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
$0x1C-F$	AC BC DC EC

physical addresses	bytes
$0x20-3$	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

$0x376 = 110\ 111\ 0110$

PTE 1: $0x10 + 6 \times 2 = 0x1C$:
AC BC

PTE 1: PPN 10 valid 1

PTE 2: $0x20 + 7 \times 2 = 0x2E$:
EF F0

PTE 2: PPN 11 valid 1
 $11\ 0110 = 0x36 \rightarrow DB$

2-level exercise (5)

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

page table base register $0x10$; translate virtual address $0x376$

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
$0x1C-F$	AC BC DC EC

physical addresses	bytes
$0x20-3$	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

$0x376 = 110\ 111\ 0110$

PTE 1: $0x10 + 6 \times 2 = 0x1C$:
AC BC

PTE 1: PPN 10 valid 1

PTE 2: $0x20 + 7 \times 2 = 0x2E$:
EF F0

PTE 2: PPN 11 valid 1

11 $0110 = 0x36 \rightarrow$ DB

2-level exercise (5)

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

page table base register $0x10$; translate virtual address $0x376$

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
$0x1C-F$	AC BC DC EC

physical addresses	bytes
$0x20-3$	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

$0x376 = 110\ 111\ 0110$

PTE 1: $0x10 + 6 \times 2 = 0x1C$:
AC BC

PTE 1: PPN 10 valid 1

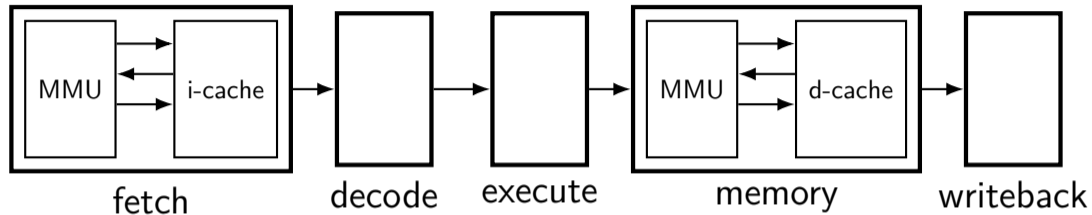
PTE 2: $0x20 + 7 \times 2 = 0x2E$:
EF F0

PTE 2: PPN 11 valid 1

$11\ 0110 = 0x36 \rightarrow$ **DB**

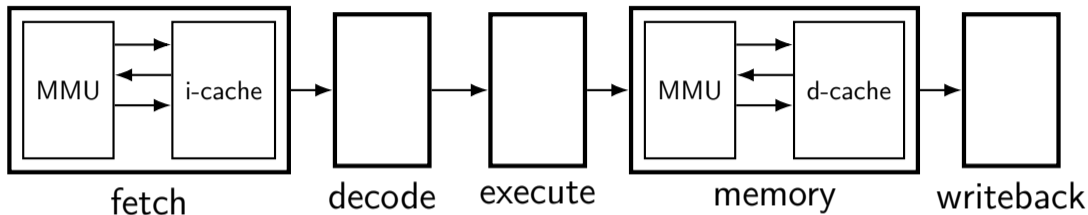
backup slides

MMUs in the pipeline



up to four memory accesses per instruction

MMUs in the pipeline

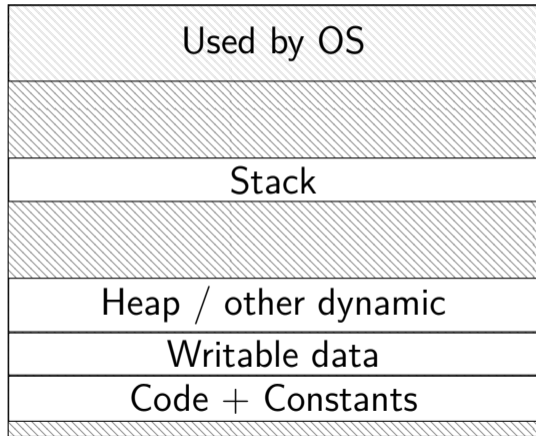


up to four memory accesses per instruction

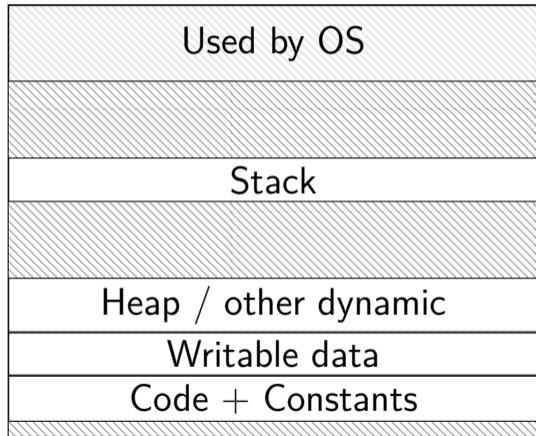
challenging to make this fast (topic for a future date)

do we really need a complete copy?

bash

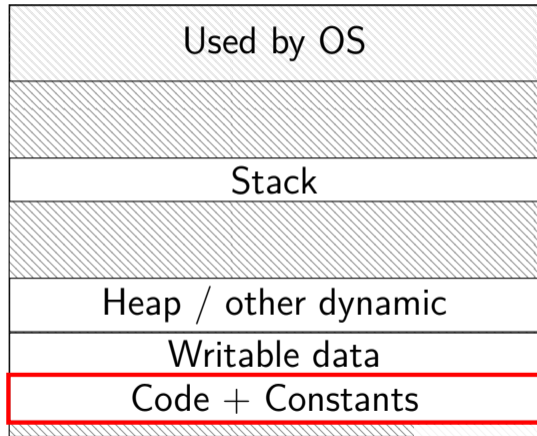


new copy of bash

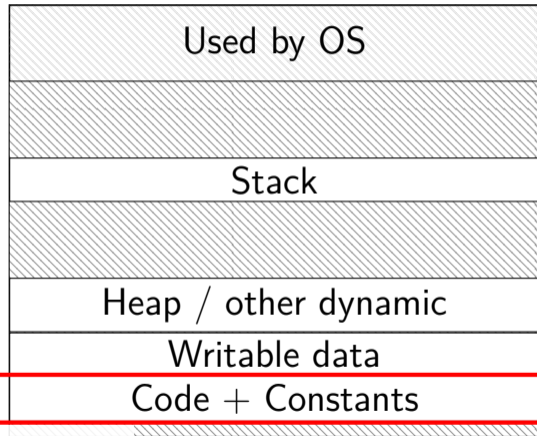


do we really need a complete copy?

bash



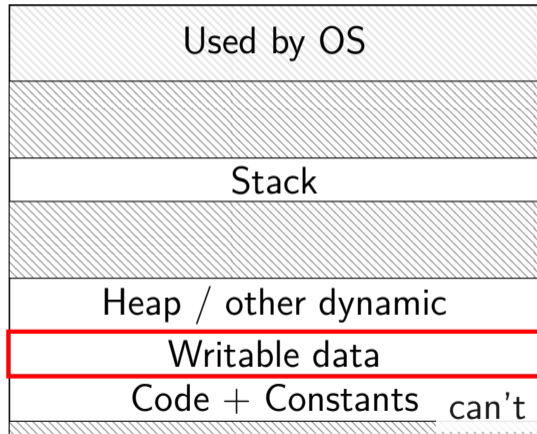
new copy of bash



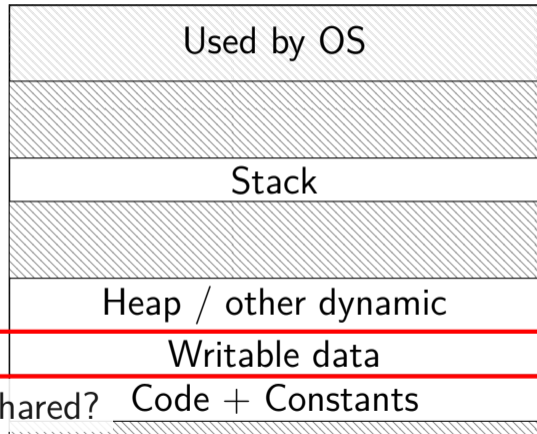
shared as read-only

do we really need a complete copy?

bash



new copy of bash



can't be shared?

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

copy-on-write and page tables

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

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

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

VPN	valid?	write?	physical page
...
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

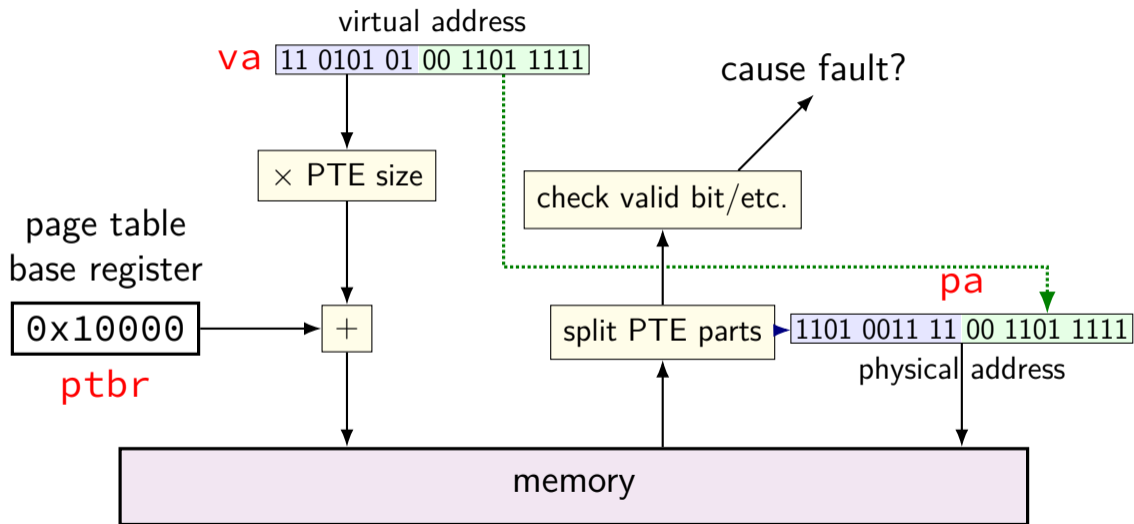
copy-on-write and page tables

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

VPN	valid?	write?	physical page
...
0x00601	1	0	0x12345
0x00602	1	0	0x12347
0x00603	1	0	0x12340
0x00604	1	0	0x200DF
0x00605	1	1	0x300FD
...

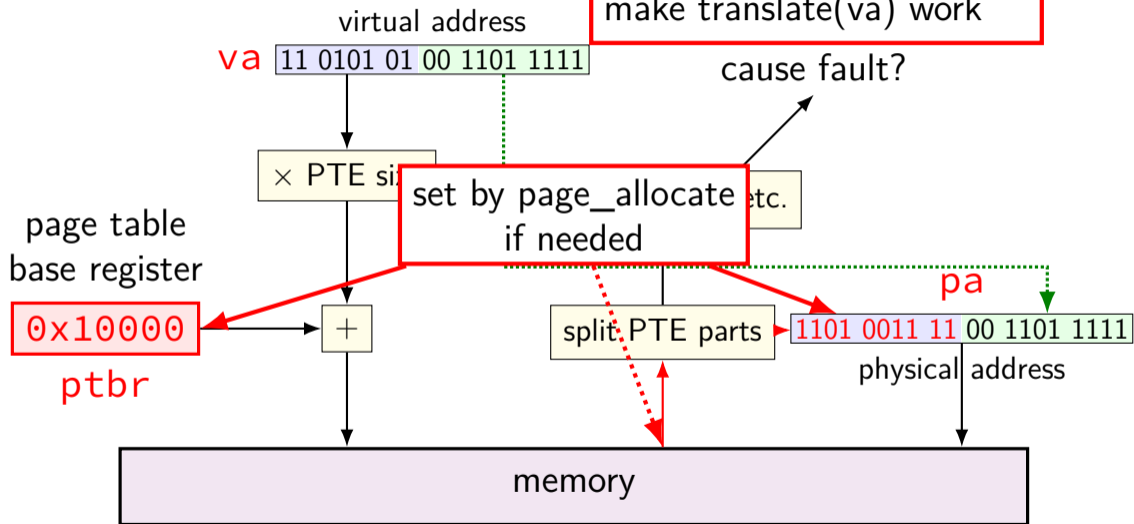
after allocating a copy, OS reruns the write instruction

pa=translate(va)



pa=translate(va)

page_allocate(va) needs to make translate(va) work



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/SDDs are slow

HDD reads and writes: milliseconds to tens of milliseconds

minimum size: 512 bytes

writing tens of kilobytes basically as fast as writing 512 bytes

SSD reads and writes: hundreds of microseconds

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

HDD/SDDs are slow

HDD reads and writes: *milliseconds to tens of milliseconds*

minimum size: 512 bytes

writing tens of kilobytes basically as fast as writing 512 bytes

SSD writes and reads: *hundreds of microseconds*

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

HDD/SDDs are slow

HDD reads and writes: *milliseconds to tens of milliseconds*

minimum size: 512 bytes

writing tens of kilobytes basically as fast as writing 512 bytes

SSD writes and reads: *hundreds of microseconds*

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

HDD/SDDs are slow

HDD reads and writes: milliseconds to tens of milliseconds

minimum size: 512 bytes

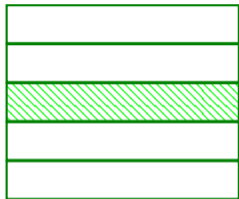
writing tens of *kilobytes* basically as fast as writing 512 bytes

SSD writes and reads: hundreds of microseconds

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

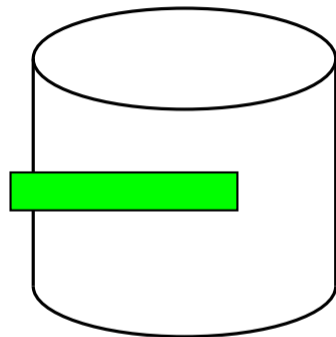
swapping timeline

program A pages



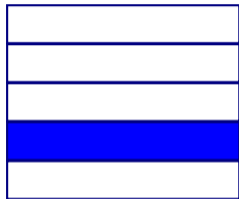
...

page fault



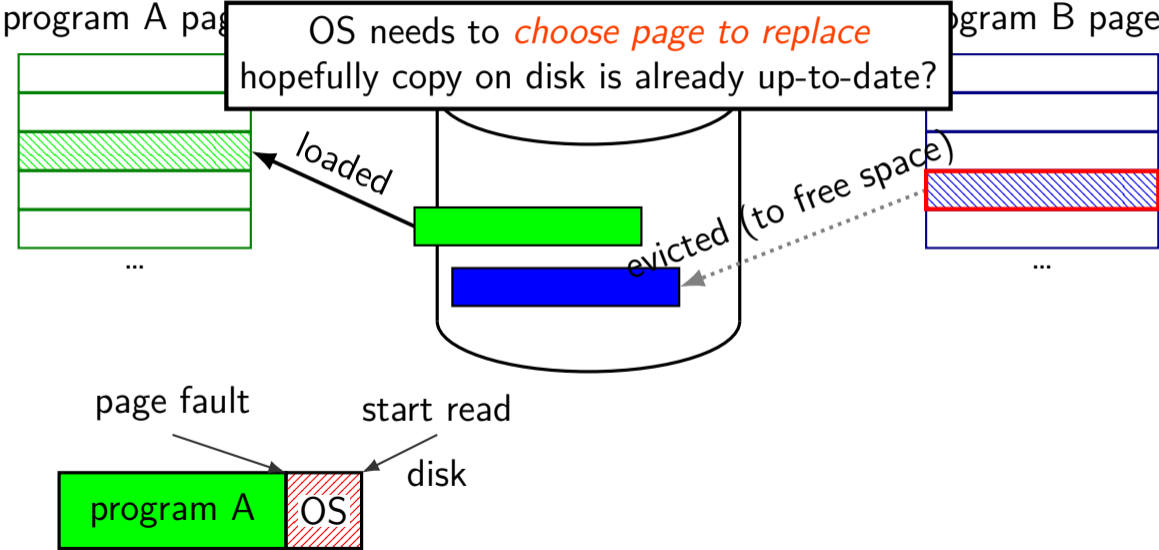
disk

program B page



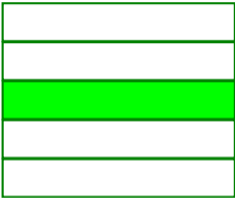
...

swapping timeline



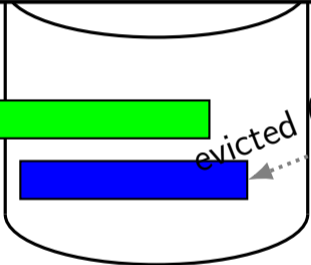
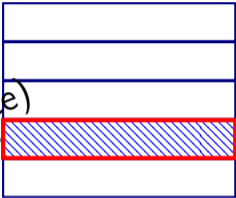
swapping timeline

program A pages



first step of replacement:
mark evicted page invalid in page table

program B page



loaded

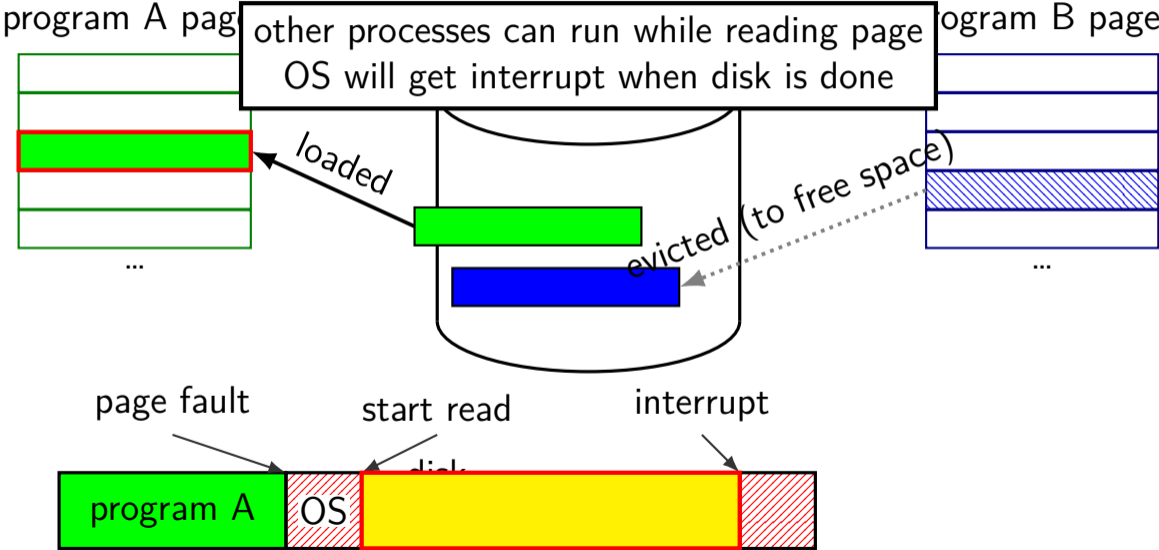
evicted (to free space)

page fault

start read



swapping timeline

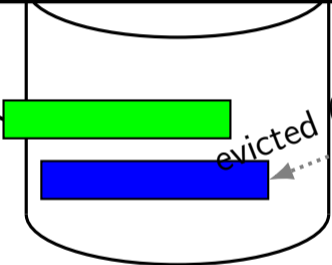


swapping timeline

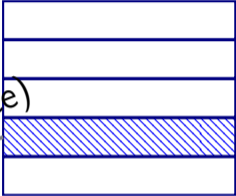
program A pages



process A's page table updated and restarted from point of fault



program B page



page fault

start read

interrupt



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*

exercise

40-bit physical addresses, 36-bit virtual addresses, 2^{14} byte (16384 byte) pages

Q1: number of virtual pages?

Q2: size of physical page numbers?

exercise: page table lookup (2)

suppose 32-byte pages (= 5-bit page offsets), 8-bit virtual addresses

VPN	valid	PPN
000	1	10
001	1	01
010	0	---
011	0	---
100	1	00
101	0	---
110	0	---
111	0	---

virtual address 0x97 = ???