alternatives to threads (con't) / virtual memory 1

last time (1)

deadlock, definition + conditions

- hold and wait
- circular waiting
- not just locks, any shared resource with waiting

preventing deadlock

- abort and retry
- revoke resources
- *acquire resources in consistent order*

detecting deadlock

- create directed graph (edges for have resource or waiting)
- search graph for cycles
- alternative algorithm for resources with variable quantity

last time (2)

alternatives to threads: event loops

- e.g. web server loop: what is next input from any browser?
- register to receive event when read/write/etc. ready
- main loop: get next event
- can avoid threads entirely — no synchronization issues
- code reorganization — move local variables to tracked state

beyond threads: event based programming

writing server that servers multiple clients?

- e.g. multiple web browsers at a time

maybe don't really need multiple processors/cores

- one network, not that fast

idea: one thread handles multiple connections

beyond threads: event based programming

writing server that servers multiple clients?

- e.g. multiple web browsers at a time

maybe don't really need multiple processors/cores

- one network, not that fast

idea: one thread handles multiple connections

issue: read from/write to multiple streams at once?

event loops

```
while (true) {
    event = WaitForNextEvent();
    switch (event.type) {
    case NEW_CONNECTION:
        handleNewConnection(event); break;
    case CAN_READ_DATA_WITHOUT_WAITING:
        connection = LookupConnection(event.fd);
        handleRead(connection);
        break;
    case CAN_WRITE_DATA_WITHOUT_WAITING:
        connection = LookupConnection(event.fd);
        handleWrite(connection);
        break;
        ...
    }
}
```

some single-threaded processing code

```
void ProcessRequest(int fd) {
  while (true) {
    char command[1024] = {};
    size_t command_length = 0;
    do {
      ssize_t read_result =
          read(fd, command + command_length,
               sizeof(command) - command_length);
      if (read_result <= 0) handle_error();
      command_length += read_result;
    } while (command[command_length - 1] != '\n');
    if (IsExitCommand(command)) { return; }
    char response[1024];
    computeResponse(response, commmand);
    size_t total_written = 0;
    while (total_written < sizeof(response)) {
      ...
    }
  }
}
```

some single-threaded processing code

original code: loop to handle one request
reads/writes multiple times; each read/write can block

```
void Pro
  while
    char command[1024] = {};
    size_t command_length = 0;
    do {
      ssize_t read_result =
          read(fd, command + command_length,
               sizeof(command) - command_length);
      if (read_result <= 0) handle_error();
      command_length += read_result;
    } while (command[command_length - 1] != '\n');
    if (IsExitCommand(command)) { return; }
    char response[1024];
    computeResponse(response, commmand);
    size_t total_written = 0;
    while (total_written < sizeof(response)) {
      ...
    }
  }
}
```

some single-threaded processing code

```
void ProcessRequest(int fd) {
  while (true) {
    char command[1024] = {};
    size_t command_length = 0;
    do {
      ssize_t read_result =
          read(fd, command + command_l
               sizeof(command) - comma
      if (read_result <= 0) handle_err
      command_length += read_result;
    } while (command[command_length - 1] != '\n');
    if (IsExitCommand(command)) { return; }
    char response[1024];
    computeResponse(response, commmand);
    size_t total_written = 0;
    while (total_written < sizeof(response)) {
      ...
    }
  }
}
```

```
struct Connection {
  int fd;
  char command[1024];
  size_t command_length;
  char response[1024];
  size_t total_written;
  ...
};
```

as event code

```
handleRead(Connection *c) {
    ssize_t read_result =
        read(fd, c->command + command_length,
             sizeof(command) - c->command_length);
    if (read_result <= 0) handle_error();
    c->command_length += read_result;

    if (c->command[c->command_length - 1] == '\n') {
        computeResponse(c->response, c->command);
        StopWaitingToRead(c->fd);
        StartWaitingToWrite(c->fd);
    }
}
```

new code: one read step per handleRead call
Connection struct: info between write calls

as event code

```
handleRead(Connection *c) {
    ssize_t read_result =
        read(fd, c->command + command_length,
             sizeof(command) - c->command_length);
    if (read_result <= 0) handle_error();
    c->command_length += read_result;

    if (c->command[c->command_length - 1] == '\n') {
        computeResponse(c->response, c->command);
        StopWaitingToRead(c->fd);
        StartWaitingToWrite(c->fd);
    }
}
```

POSIX support for event loops

`select` and `poll` functions

- take list(s) of file descriptors to read and to write
- wait for them to be read/writeable without waiting
- (or for new connections associated with them, etc.)

many OS-specific extensions/improvements/alternatives:

- examples: Linux epoll, Windows IO completion ports
- better ways of managing list of file descriptors
- enqueue read/write instead of learning when read/write okay

message passing

instead of having variables, locks between threads...

send messages between threads/processes

what you need anyways between machines

- big 'supercomputers' = really many machines together

arguably an easier model to program

- can't have locking issues

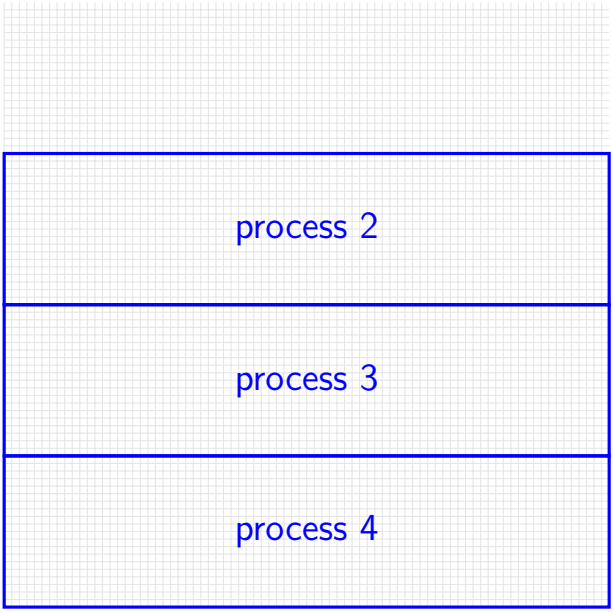
message passing API

core functions: Send(toId, data)/Recv(fromId, data)

simplest(?) version: functions wait for other processes/threads

```
if (thread_id == 0) {
    for (int i = 1; i < MAX_THREAD; ++i) {
        Send(i, getWorkForThread(i));
    }
    for (int i = 1; i < MAX_THREAD; ++i) {
        WorkResult result;
        Recv(i, &result);
        handleResultForThread(i, result);
    }
} else {
    WorkInfo work;
    Recv(0, &work);
    Send(0, ComputeResultFor(work));
}
```

message passing game of life

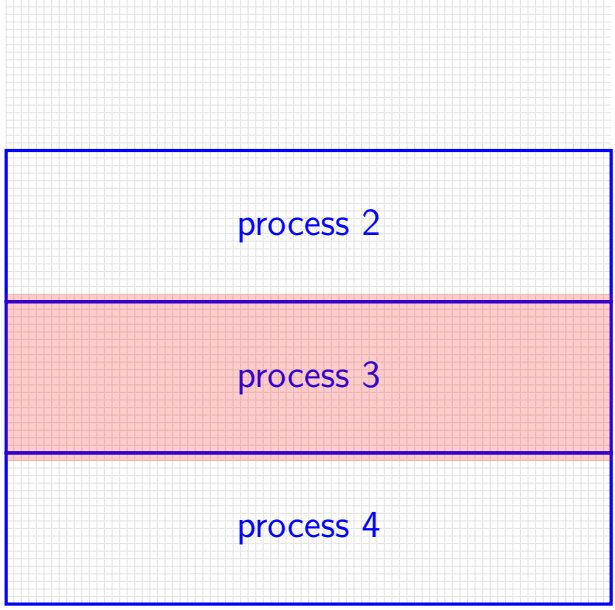


divide grid
like you would for normal threads

each process stores cells
in that part of grid

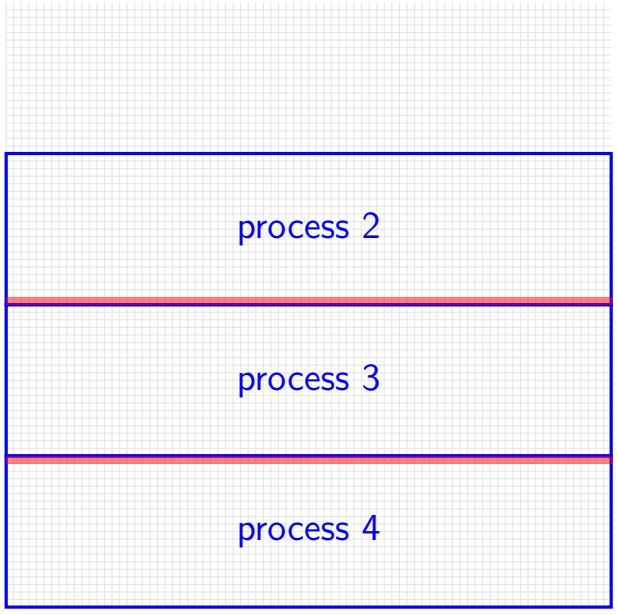
(no shared memory!)

message passing game of life



process 3 only needs values of cells around its area (values of cells adjacent to the ones it computes)

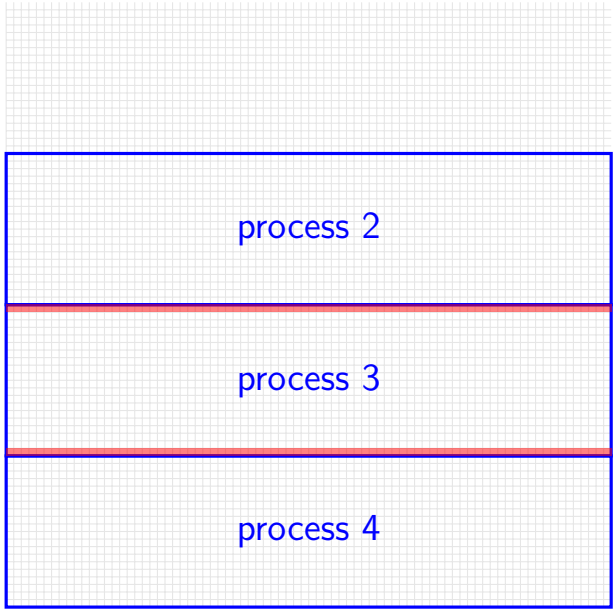
message passing game of life



small slivers of
other process's cells needed

solution: process 2, 4
send messages with cells every iterat

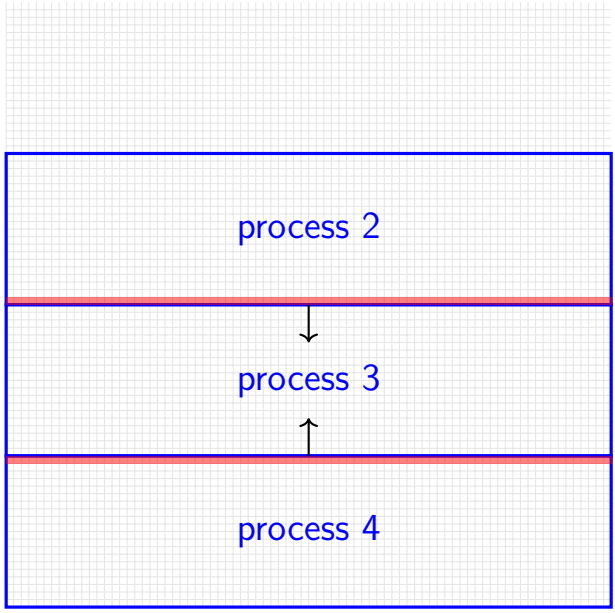
message passing game of life



some of process 3's cells
also needed by process 2/4

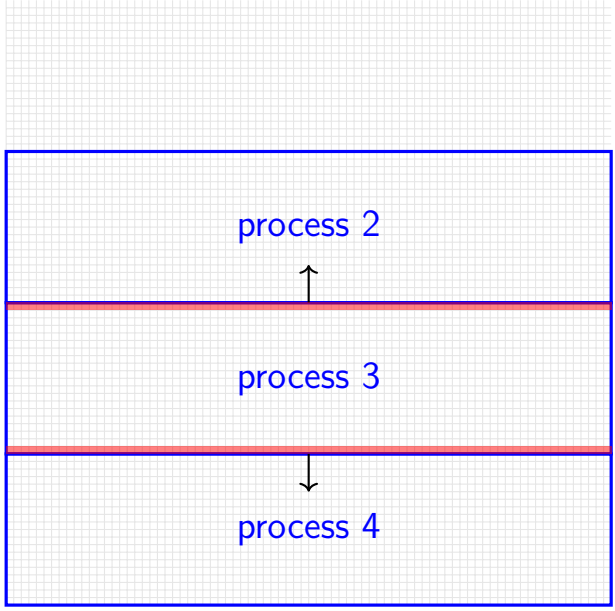
so process 3 also sends messages

message passing game of life



one possible pseudocode:
all even processes send messages
(while odd receives), then
all odd processes send messages
(while even receives)

message passing game of life



one possible pseudocode:
all even processes send messages
(while odd receives), then
all odd processes send messages
(while even receives)

a prereq note

in CS 3330 or CoA 2, we cover virtual memory for several days

- CS3330 = Computer Architecture
- CoA2 = Computer Organization and Architecture 2 in the CS 2020 curriculum pilot

for CpEs: the prereq for this class is ECE's *embedded* class

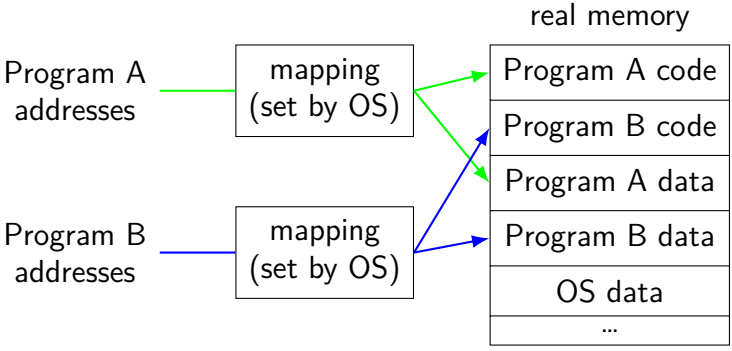
(and *not the CpE architecture class*)

- inexcusably, I've been confused about this
- and falsely insinuated that students might be missing a prereq
- and I'm really sorry about that

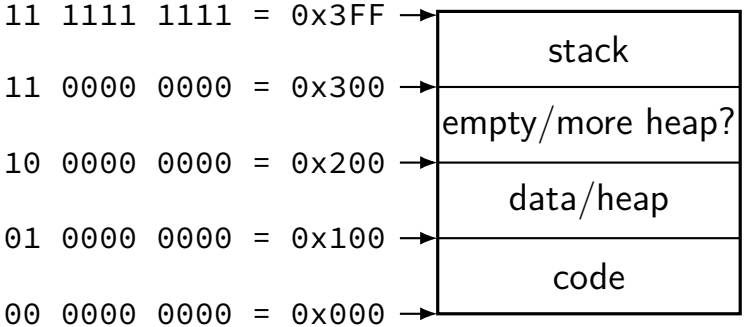
I think virtual memory barely covered in CpE embedded *or* architecture

but I don't really know what happened in those classes

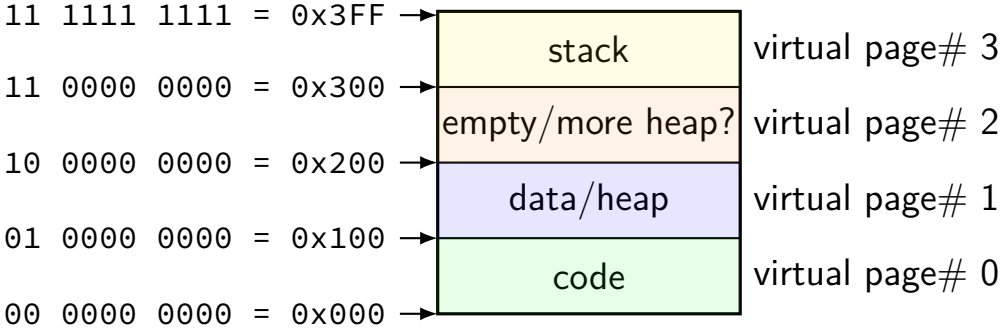
address translation



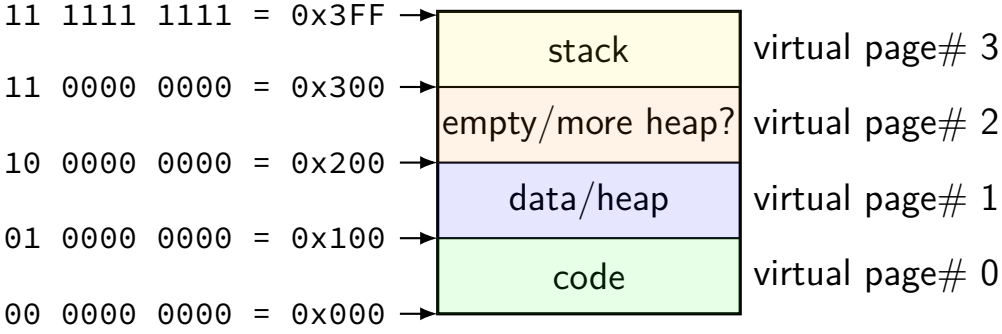
toy program memory



toy program memory

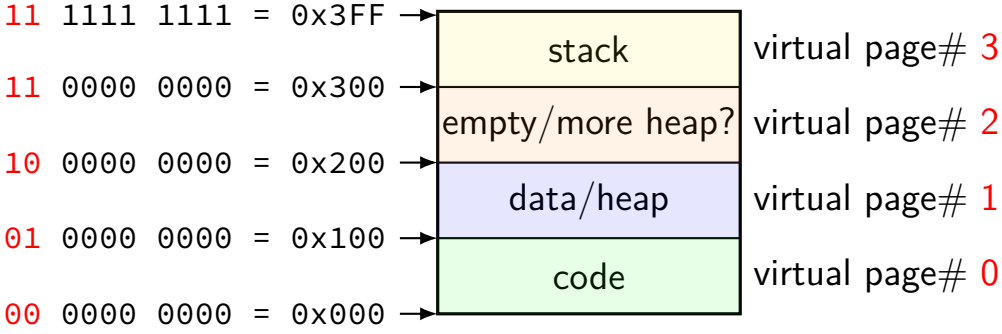


toy program memory



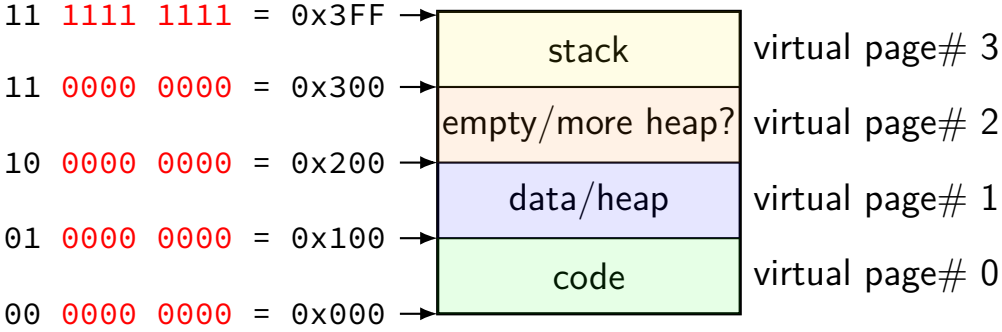
divide memory into pages (2^8 bytes in this case)
"virtual" = addresses the program sees

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

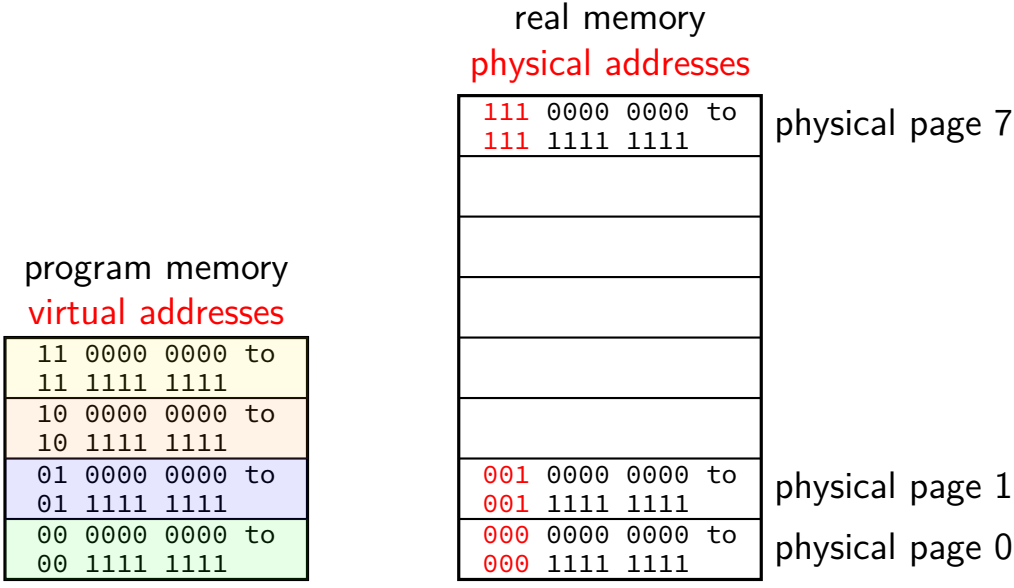
program memory
virtual addresses

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

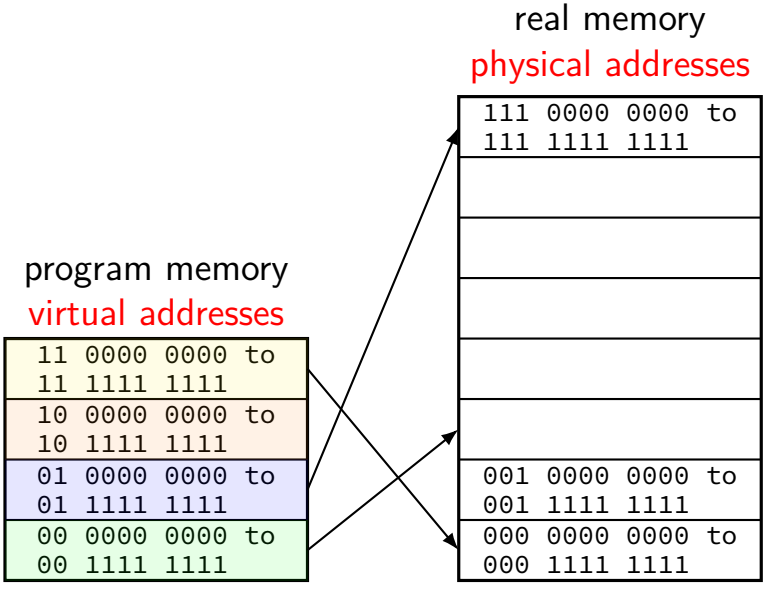
real memory
physical addresses

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

toy physical memory

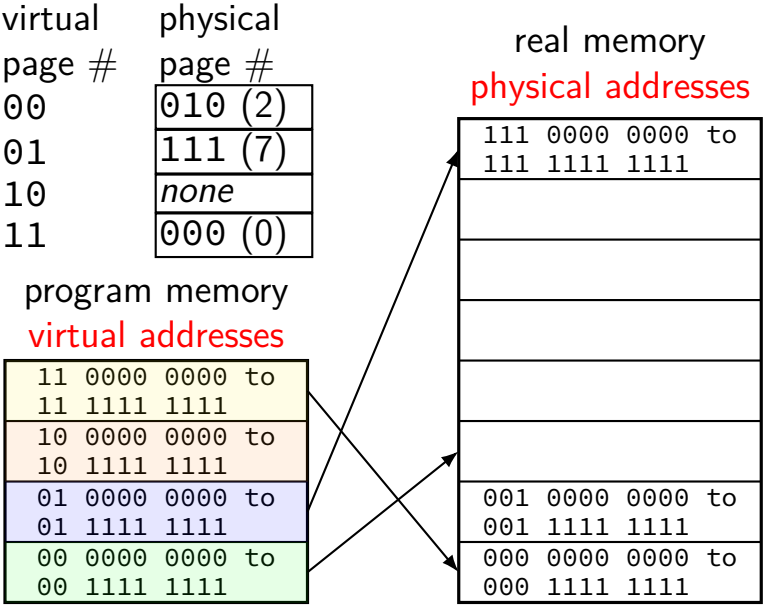


toy physical memory



toy physical memory

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



toy physical memory

page table!

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

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

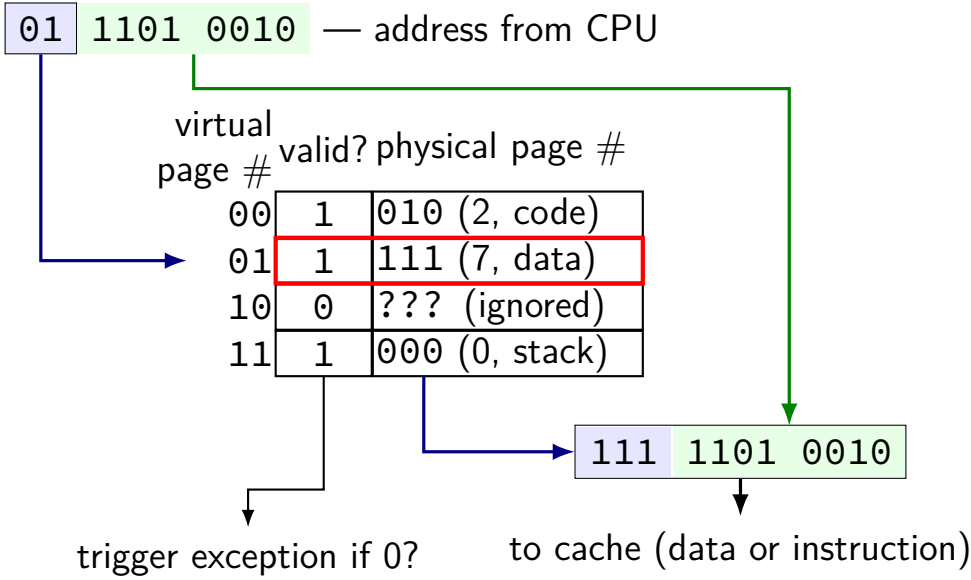
physical addresses

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

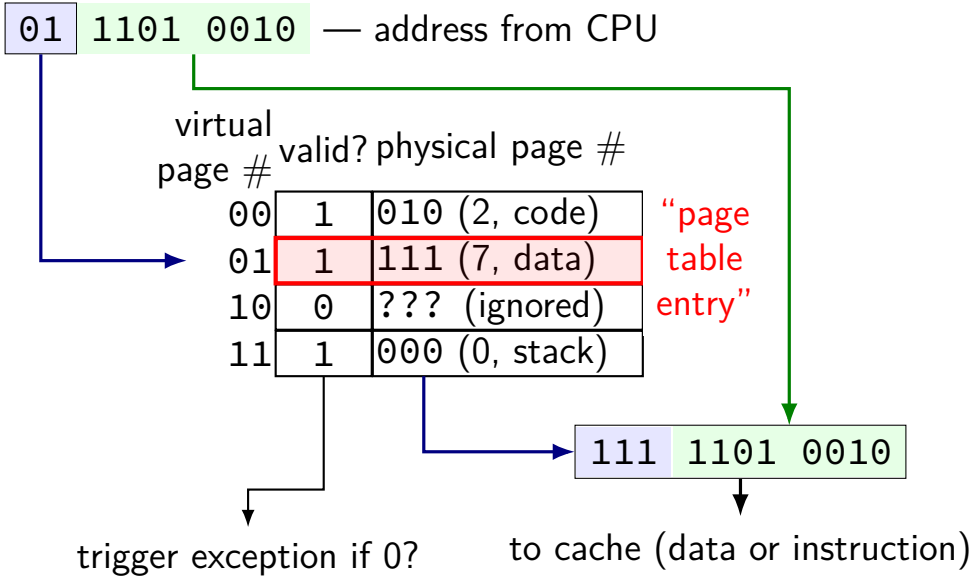
toy page table lookup

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

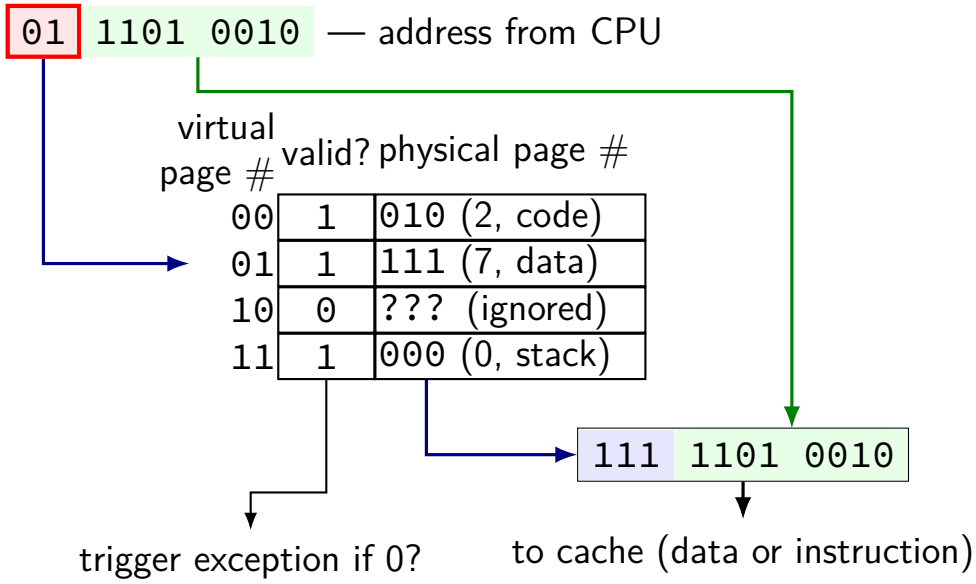
toy page table lookup



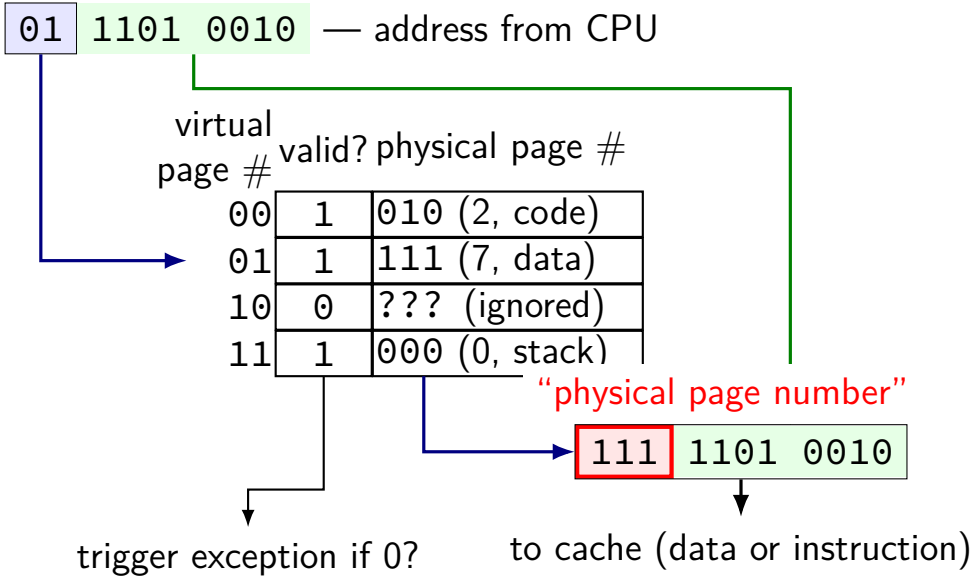
toy page table lookup



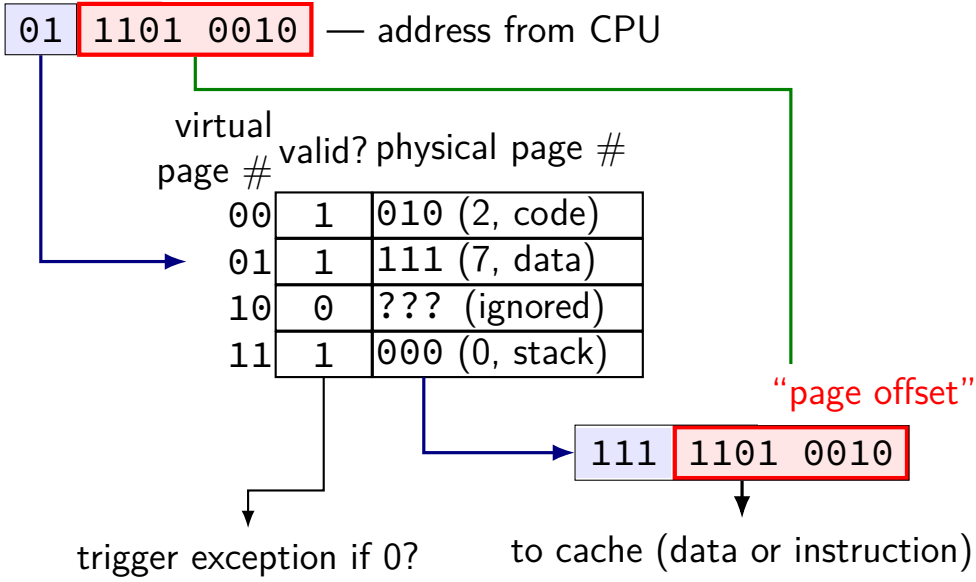
"virtual page number" **lookup**



toy page table lookup



toy pa "page offset" ookup



x86-32: VPN and PO

32-bit x86: 4096 byte (2^{12} byte) pages

given virtual address `0xABCD0123`

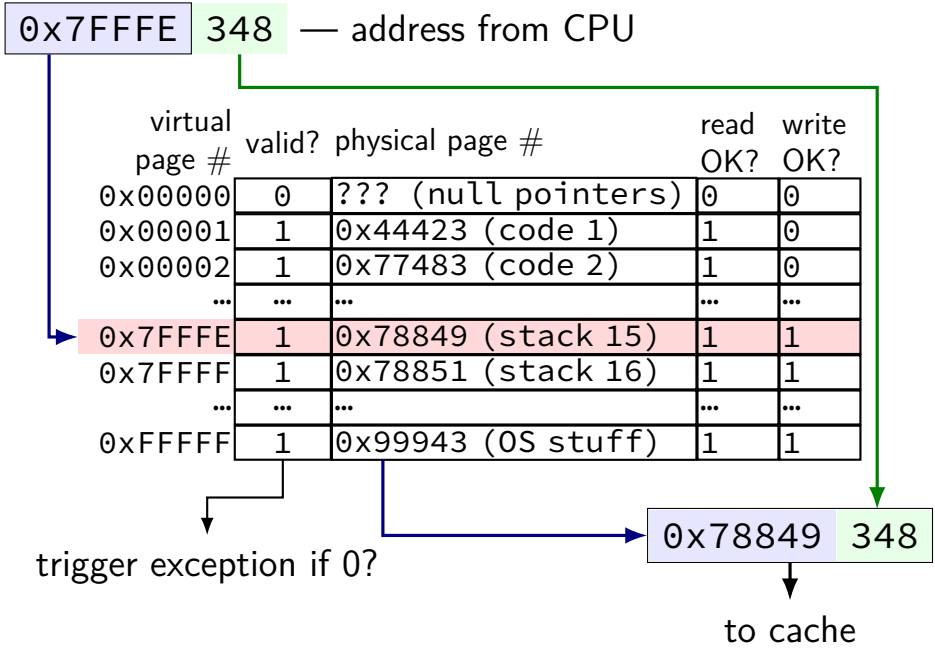
virtual page number = _____

page offset = _____

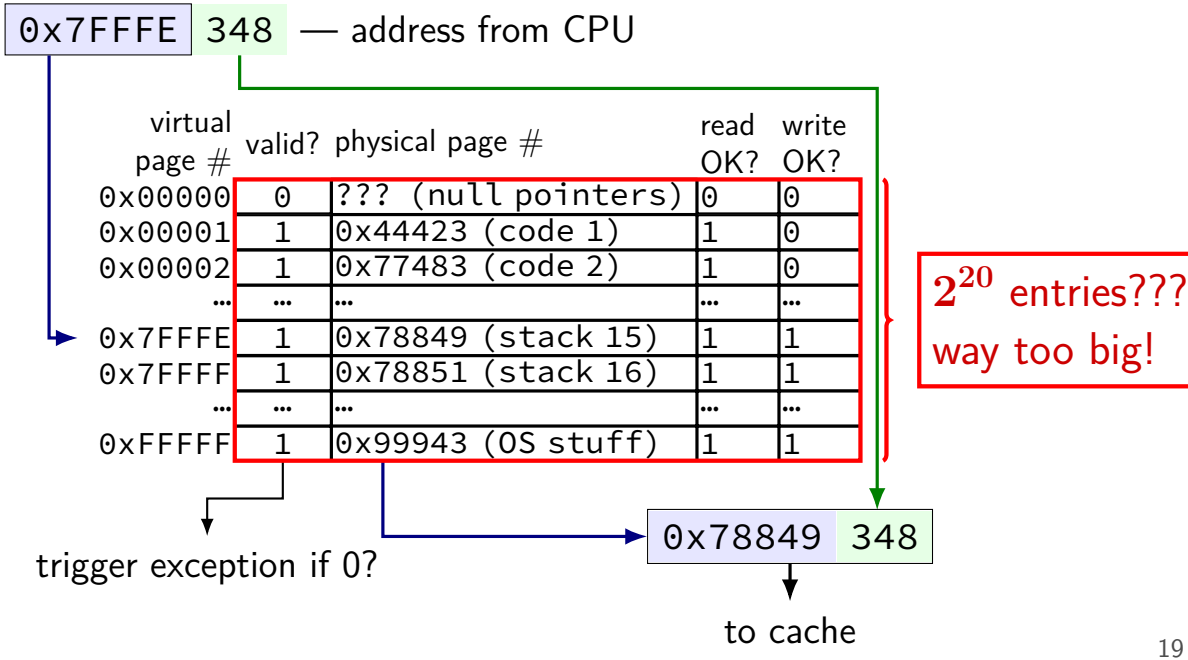
if that virtual page maps to physical page `0x998`

physical address = _____

32-bit x86 flat page table???



32-bit x86 flat page table???



storing huge page table?

keep it in memory

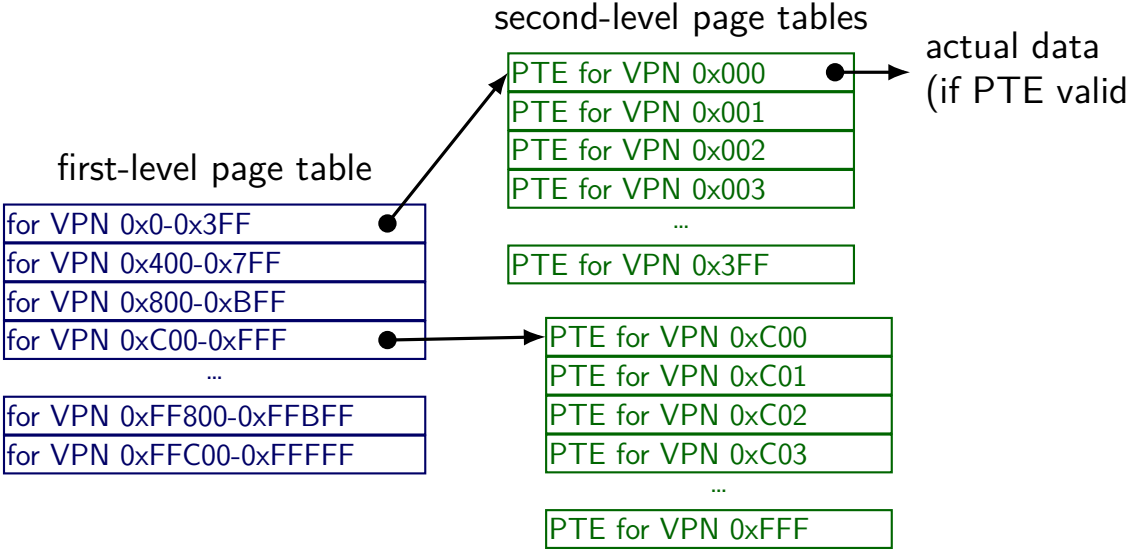
- add special cache for page table entries to handle memory being slow
- special cached called translation lookaside buffer (TLB)

use a tree and don't store most invalid page table entries

- take advantage of large contiguous invalid regions
- (between stack and heap, most high memory addresses, etc.)

two-level page tables

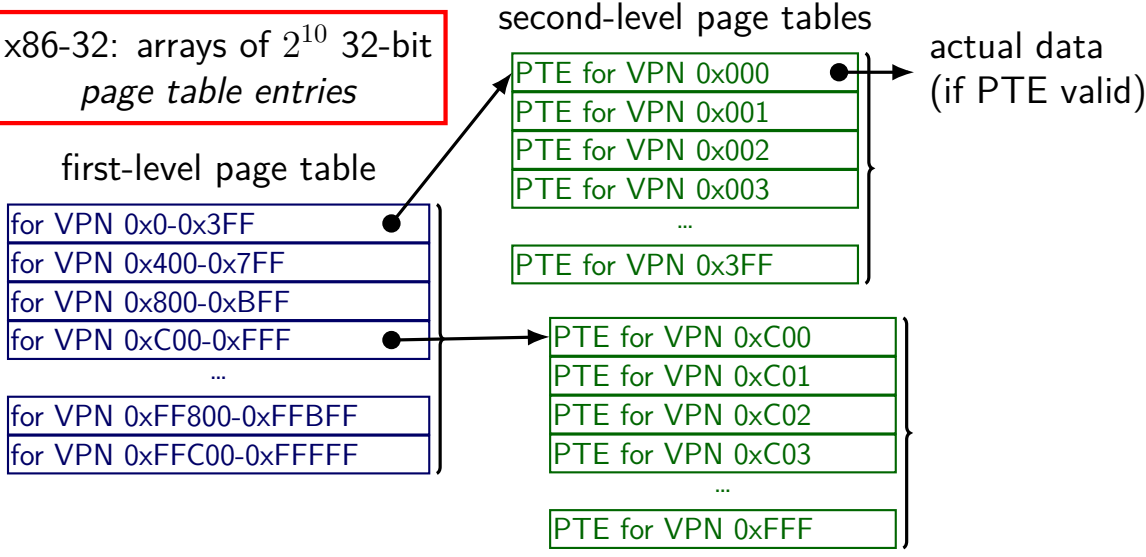
two-level page table; 2^{20} pages total; 2^{10} entries per table



two-level page tables

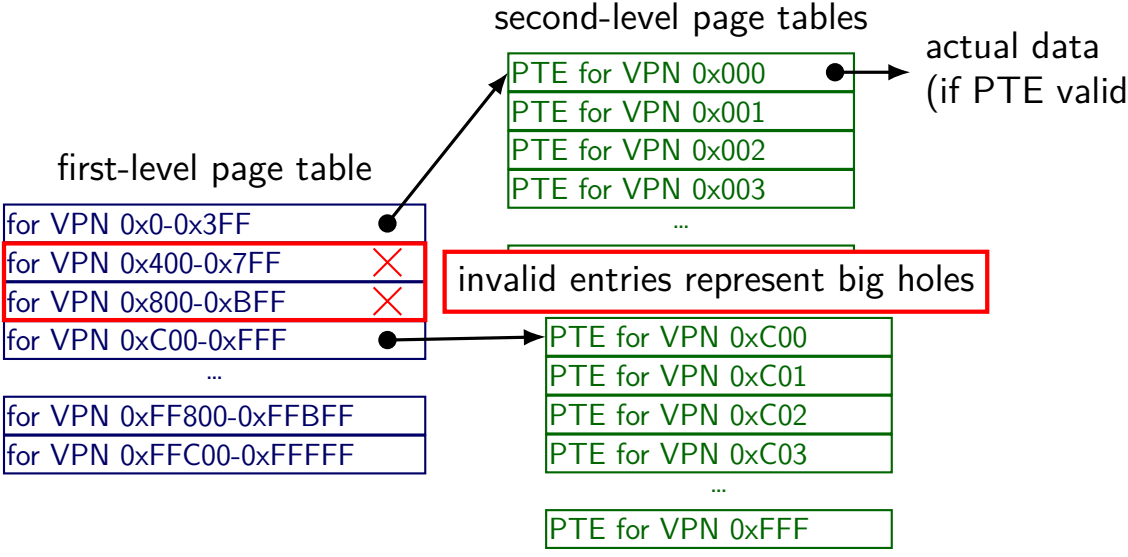
two-level page table; 2^{20} pages total; 2^{10} entries per table

x86-32: arrays of 2^{10} 32-bit *page table entries*



two-level page tables

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



two-level page tables

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

first-level pag

for VPN 0x0-0x3FF

for VPN 0x400-0x7F

for VPN 0x800-0xBF

for VPN 0xC00-0xFF

...

for VPN 0xFF800-0x

for VPN 0xFFC00-0xFFFFF

first-level page table

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

PTE for VPN 0xC03

...

PTE for VPN 0xFFF

two-level page tables

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

first-level pag

for VPN 0x0-0x3FF

for VPN 0x400-0x7F

for VPN 0x800-0xBF

for VPN 0xC00-0xFF

...

for VPN 0xFF800-0x

for VPN 0xFFC00-0xFFFFF

first-level page table

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

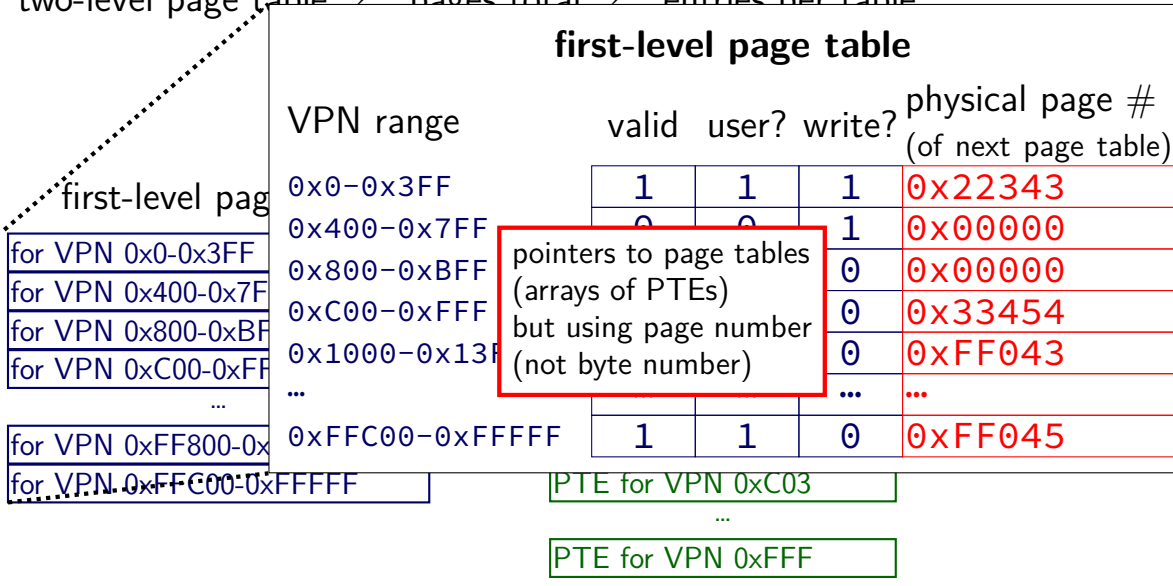
PTE for VPN 0xC03

...

PTE for VPN 0xFFF

two-level page tables

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



two-level page tables

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

first-level pag

for VPN 0x0-0x3FF

for VPN 0x400-0x7F

for VPN 0x800-0xBF

for VPN 0xC00-0xFF

...

for VPN 0xFF800-0x

for VPN 0xFFC00-0xFFFFF

first-level page table

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

valid bits indicate "holes"
note: physical page 0 is valid
so can't use NULL ptrs

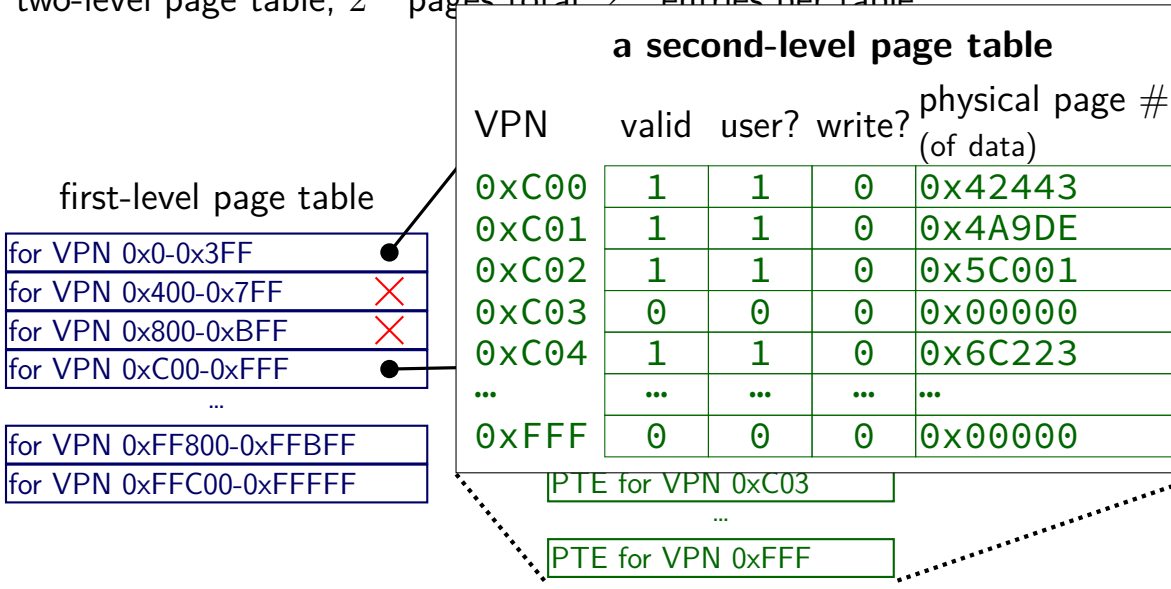
PTE for VPN 0xC03

...

PTE for VPN 0xFFF

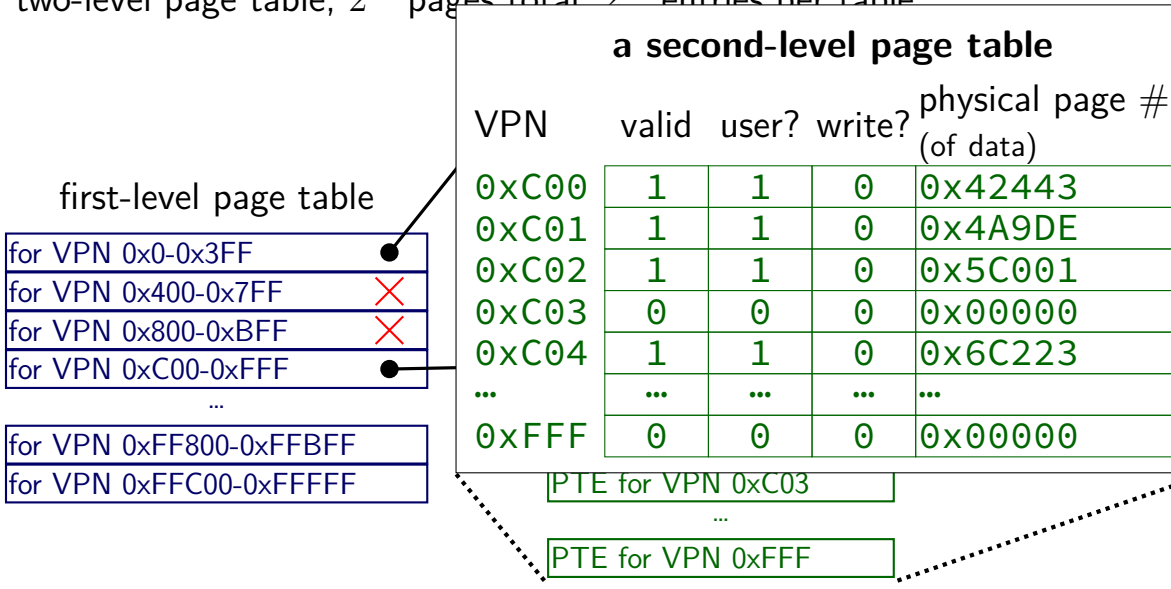
two-level page tables

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



two-level page tables

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



two-level page table naming

what the page table base register points to:

first-level page table

top-level page table

page directory (Intel's term, used in xv6 code)

what first-level page table entries point to

second-level page table

page table (Intel's term, used in xv6 code)

I'll avoid using this term unqualified...
but Intel manuals/xv6 do not

32-bit x86 paging

4096 ($= 2^{12}$) byte pages

4-byte page table entries — stored in memory

two-level table:

first 10 bits lookup in first level ("page directory")
second 10 bits lookup in second level

remaining 12 bits: which byte of 4096 in page?

32-bit x86 paging (in xv6)

xv6 header: mmu.h

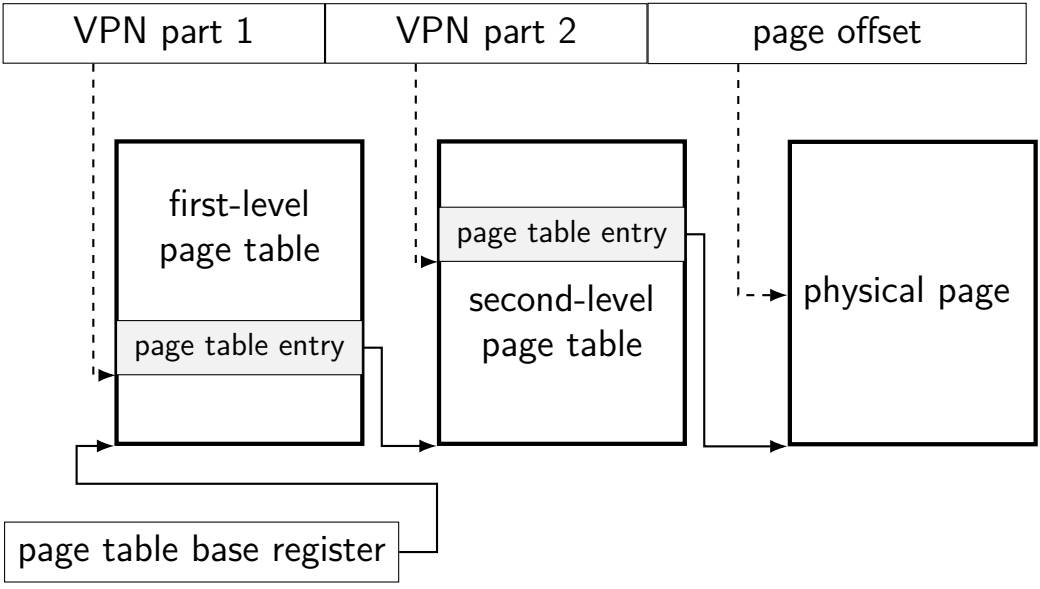
```
// A virtual address 'va' has a three-part structure as follows:
//
// +--------10------+-------10-------+---------12----------+
// | Page Directory |   Page Table    | Offset within Page  |
// |      Index     |      Index      |                     |
// +----------------+-----------------+---------------------+
//  \--- PDX(va) --/ \--- PTX(va) --/

// page directory index
#define PDX(va)         (((uint)(va) >> PDXSHIFT) & 0x3FF)

// page table index
#define PTX(va)         (((uint)(va) >> PTXSHIFT) & 0x3FF)

// construct virtual address from indexes and offset
#define PGADDR(d, t, o) ((uint)((d) << PDXSHIFT | (t) << PTXSHIFT |
```

another view



exercise (1)

4096 ($= 2^{12}$) byte pages

4-byte page table entries — stored in memory

two-level table:

first 10 bits lookup in first level ("page directory")
second 10 bits lookup in second level

exercise:

virtual address 0x12345678
base pointer 0x1000 (byte address)
first-level PTE: PPN 0x14; second-level PTE: PPN 0x15

address of 1st-level PTE? of second-level PTE?

exercise (2)

4096 ($= 2^{12}$) byte pages

4-byte page table entries — stored in memory

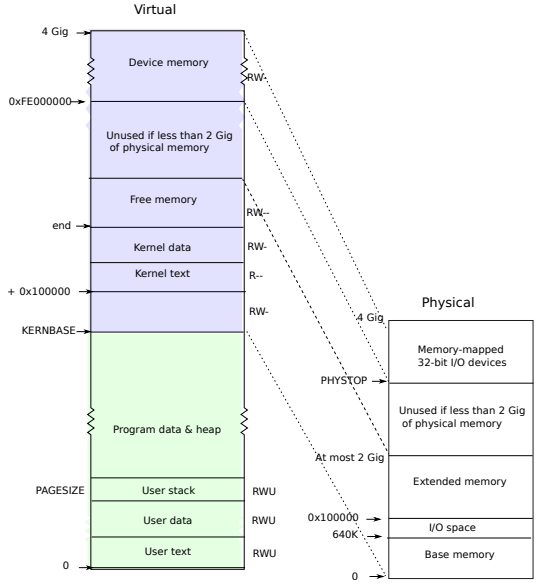
two-level table:

- first 10 bits lookup in first level ("page directory")
- second 10 bits lookup in second level

exercise: how big is...

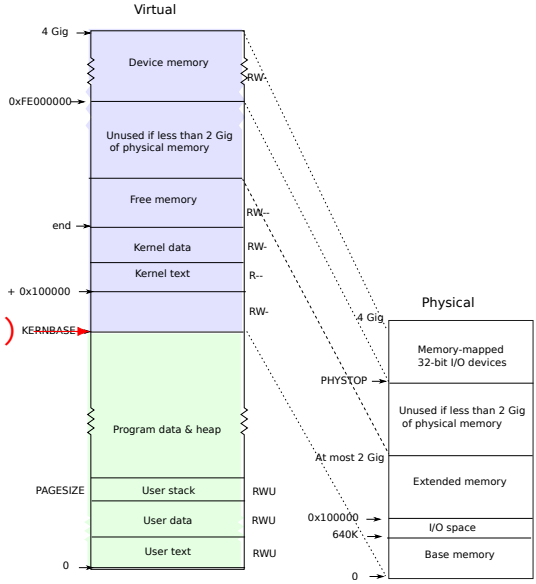
- a process's x86-32 page tables with 1 valid 4K page?
- a process's x86-32 page table with all 4K pages populated?

xv6 memory layout

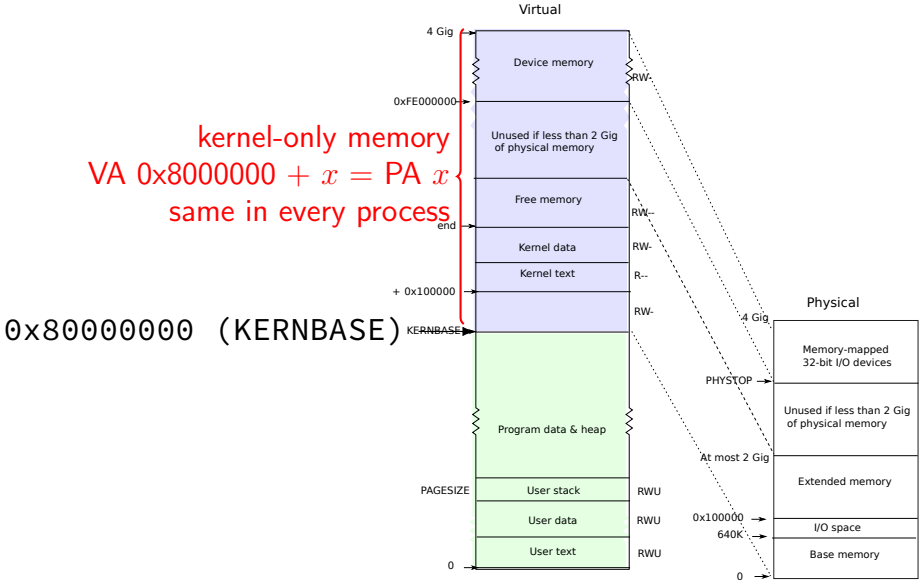


xv6 memory layout

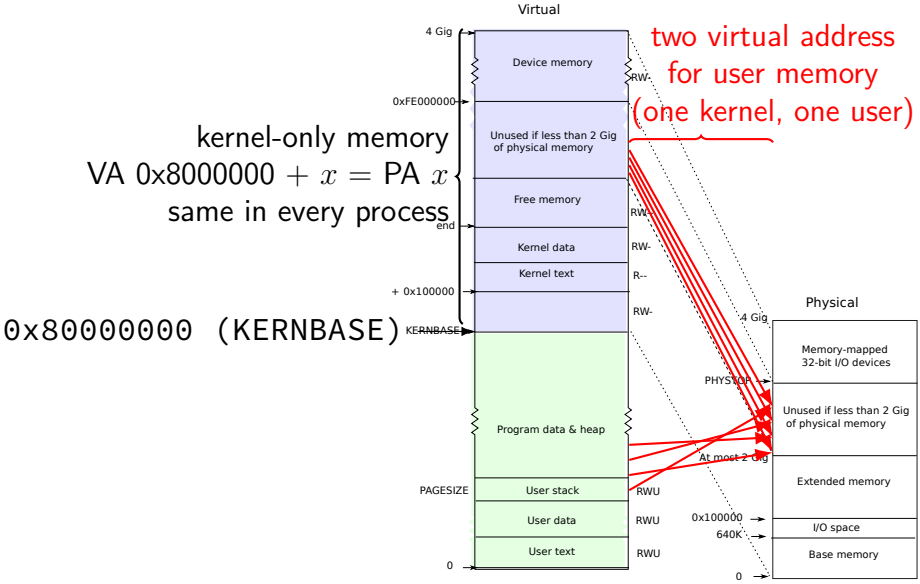
0x80000000 (KERNBASE)



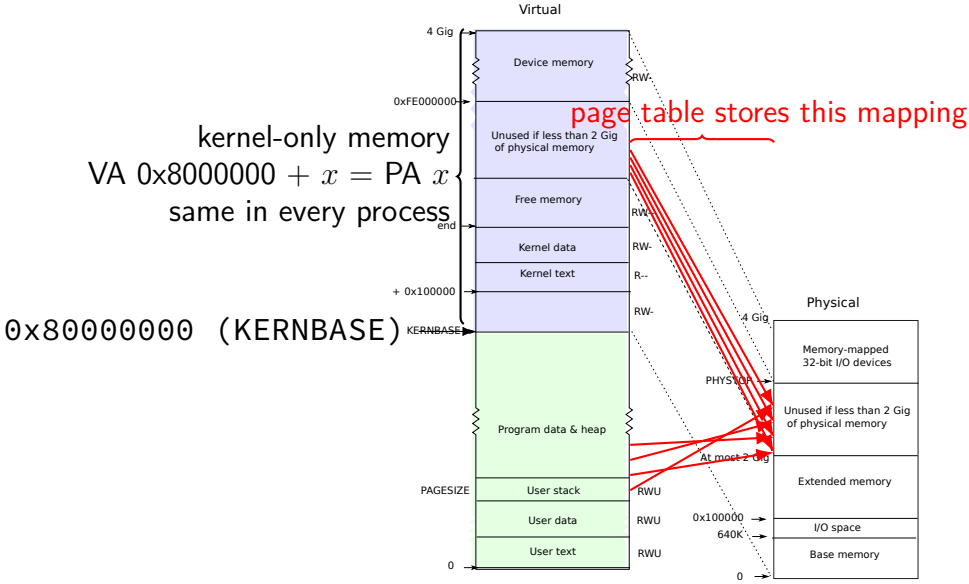
xv6 memory layout



xv6 memory layout



xv6 memory layout



xv6 kernel memory

virtual memory $>$ KERNBASE (`0x8000 0000`) is for kernel

always mapped as kernel-mode only

- protection fault for user-mode programs to access

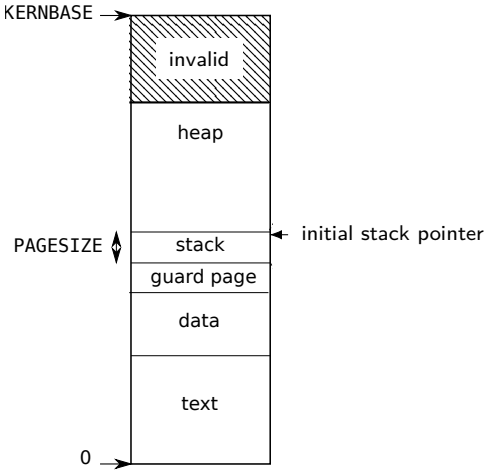
physical memory address 0 is mapped to KERNBASE$+0$

physical memory address N is mapped to KERNBASE$+N$

- not done by hardware — just page table entries OS sets up on boot
- very convenient for manipulating page tables with physical addresses

kernel code loaded into contiguous physical addresses

xv6 program memory



guard page

1 page after stack

- at lower addresses since stack grows towards lower addresses

marked as kernel-mode-only

idea: stack overflow $\rightarrow$ protection fault $\rightarrow$ kills program

skipping the guard page

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

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

xv6 types for paging (1)

virtual addresses: pointers (`void*`, etc.)

physical addresses: ints

P2V/V2P

V2P(x) (virtual to physical)
convert *kernel* address x to physical address

- subtract KERNBASE (0x8000 0000)
- assumes you pass a kernel address
- have user address? need full page table lookup instead

P2V(x) (physical to virtual)
convert *physical* address x to kernel address

- add KERNBASE (0x8000 0000)

xv6 convention: virtual addresses represented using pointers

xv6 convention: physical addresses represented using integers

P2V/V2P

V2P(x) (virtual to physical)
convert *kernel* address x to physical address

- subtract KERNBASE (0x8000 0000)
- assumes you pass a kernel address
- have user address? need full page table lookup instead

P2V(x) (physical to virtual)
convert *physical* address x to kernel address

- add KERNBASE (0x8000 0000)

xv6 convention: virtual addresses represented using pointers

xv6 convention: physical addresses represented using integers

P2V/V2P

V2P(x) (virtual to physical)
convert *kernel* address x to physical address

- subtract KERNBASE (0x8000 0000)
- assumes you pass a kernel address
- have user address? need full page table lookup instead

P2V(x) (physical to virtual)
convert *physical* address x to kernel address

- add KERNBASE (0x8000 0000)

xv6 convention: virtual addresses represented using pointers

xv6 convention: physical addresses represented using integers

xv6 types for paging (2)

x86-32 (as used by xv6) has 4-byte page table entries

page table entries, first-level: `pde_t`

> page directory entry
> alias for unsigned int

page table entries, second-level: `pte_t`

> page table entry
> alias for unsigned int

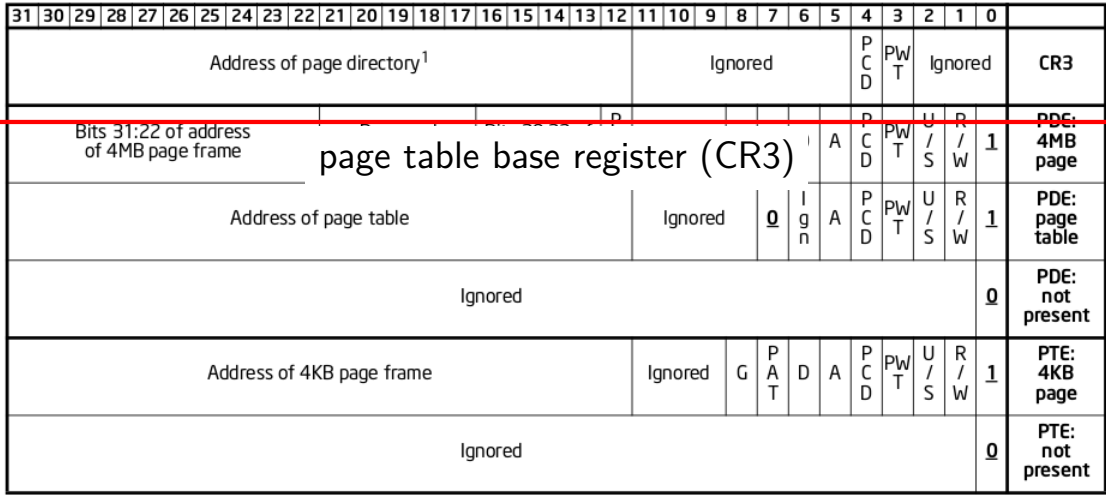
x86-32 page tables are 4096-byte *arrays of 1024 entries*

x86-32 page table entries

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

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

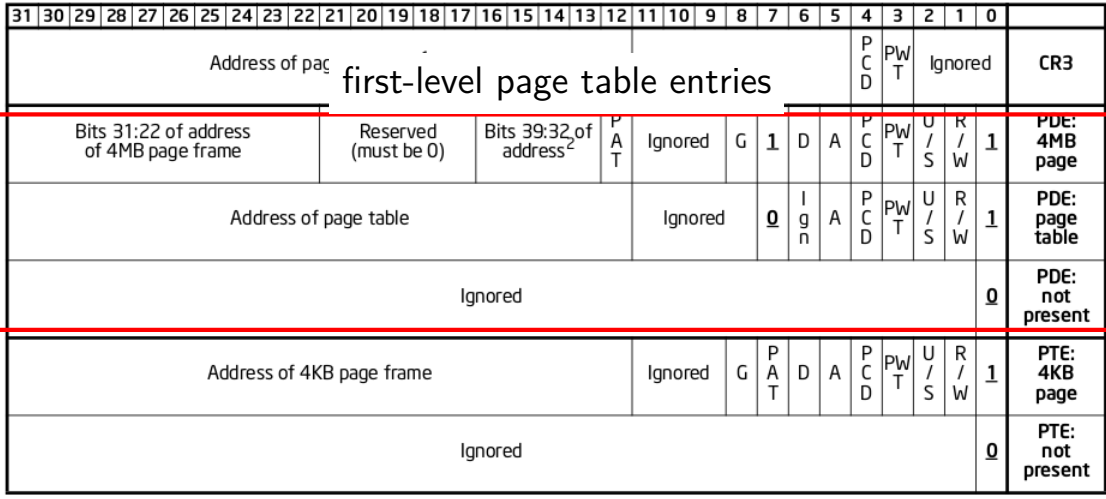
x86-32 page table entries



31–12	11–9	8	7	6	5	4	3	2	1	0	
Address of page directory[1]	Ignored					PCD	PWT	Ignored			CR3
Bits 31:22 of address of 4MB page frame					A	PCD	PWT	U/S	R/W	1	PDE: 4MB page
Address of page table	Ignored		0	Ign	A	PCD	PWT	U/S	R/W	1	PDE: page table
Ignored										0	PDE: not present
Address of 4KB page frame	Ignored	G	PAT	D	A	PCD	PWT	U/S	R/W	1	PTE: 4KB page
Ignored										0	PTE: not present

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

x86-32 page table entries



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

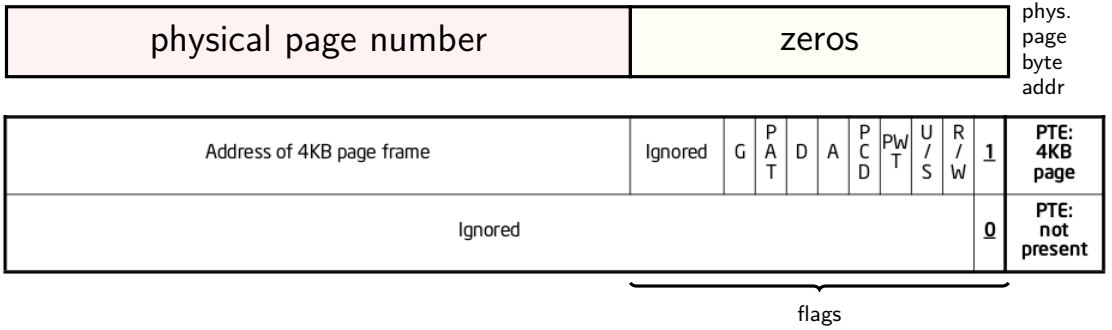
Figure 4-4. Formats of CR3 and Paging-Structure Entries with 32-Bit Paging

x86-32 page table entries

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

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

x86-32 page table entry v addresses



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

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

makes constructing page table entries simpler:

```
physicalAddress | flagsBits
```

x86-32 pagetables: page table entries

xv6 header: mmu.h

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

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

xv6: extracting top-level page table entry

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

xv6: manually setting page table entry

```
pde_t *some_page_table;  // if top-level table
pte_t *some_page_table;  // if next-level table
...
...
some_page_table[index] = PTE_P | PTE_W | PTE_U | base_physical_addre
```