Changelog

Changes made in this version not seen in first lecture:

2 November: Correct space on demand from $\leq$ to $<$ and $>$ to $\geq$ since sz is one past the end of the heap

Virtual Memory 1

last time

deadlock
- thread A holding a resource X...
- waits to get another resource Y...
- that is held by a thread...
- that needs thread A to give up resource X
- (directly or indirectly)

preventing deadlock
- ordering: avoid cycles
- have more copies of resource
- stop waiting (abort, steal resource, ...)

detecting deadlock — see if processes finish

avoiding threads: event loops

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 comamnd_length = 0;
        do {
            ssize_t read_result =
                read(fd, command + con
                    sizeof(command) –
            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)) {
            ...
        }
    }
```

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

some single-threaded processing code

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

```
void ProcessRequest(int fd) {
    while (true) {
        char command[1024] = {};
        size_t comamnd_length = 0;
        do {
            ssize_t read_result =
                read(fd, command + con
                    sizeof(command) -
            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)) {
            ...
        }
    }
```

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);
        if (IsExitCommand(command)) {
          FinishConnection(c);
        }
        StopWaitingToRead(c->fd);
        StartWaitingToWrite(c->fd);
    }
}
```

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);
        if (IsExitCommand(command)) {
          FinishConnection(c);
        }
        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
- do read/write when ready instead of just returning when reading/writing is 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

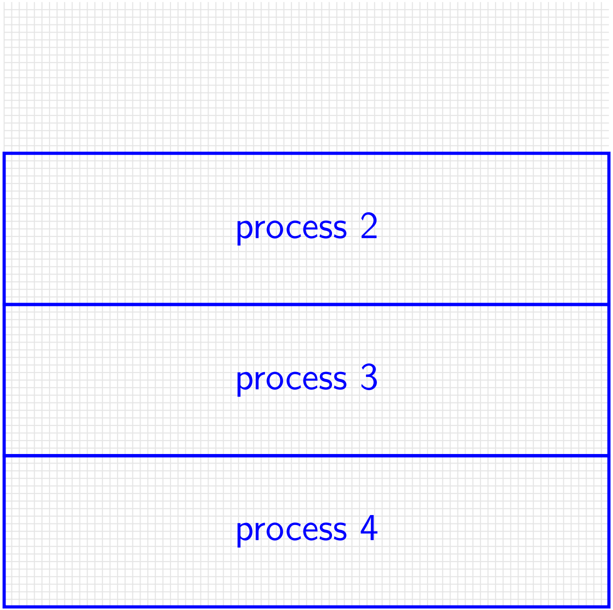
core functions: Send(toId, data)/Recv(fromId, data)

simplest version: functions wait for other processes/threads

extensions: send/recv at same time, multiple messages at once, don't wait, etc.

```
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

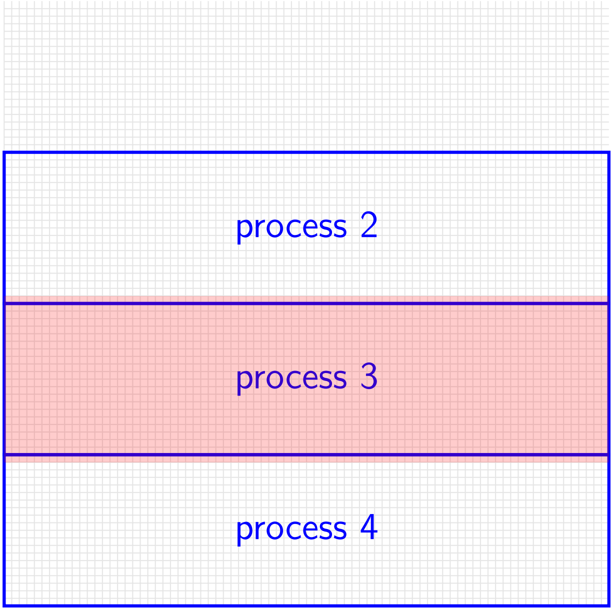


divide grid
like you would for normal threads

each process stores cells
in that part of grid

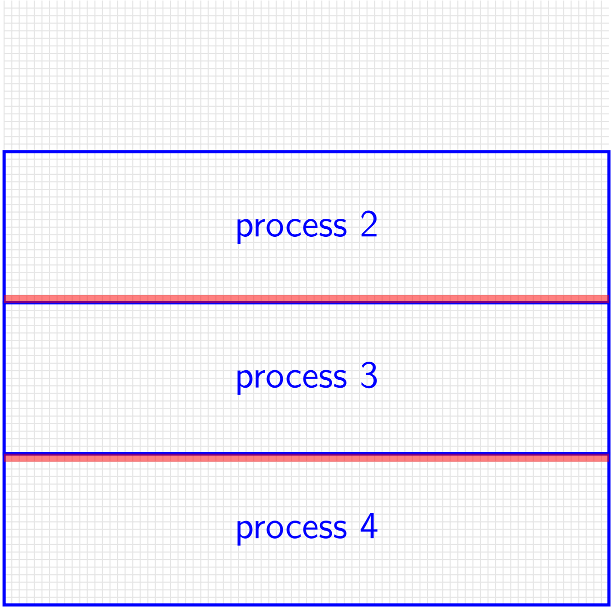
(no shared memory!)

message passing game of life



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

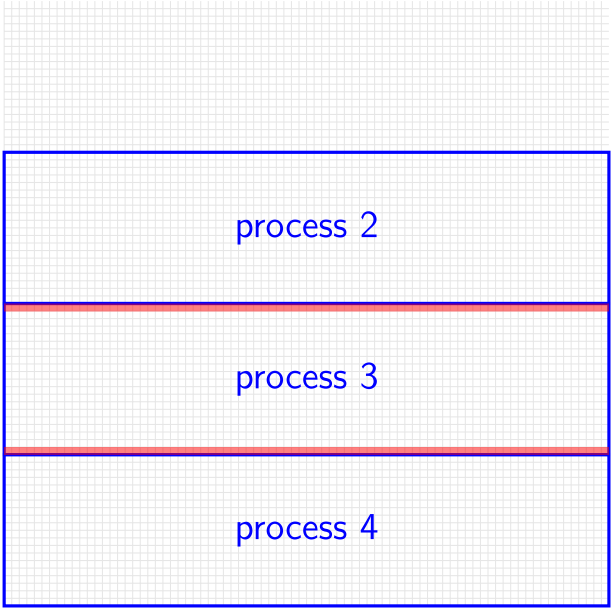
message passing game of life



small slivers of
other process's cells needed

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

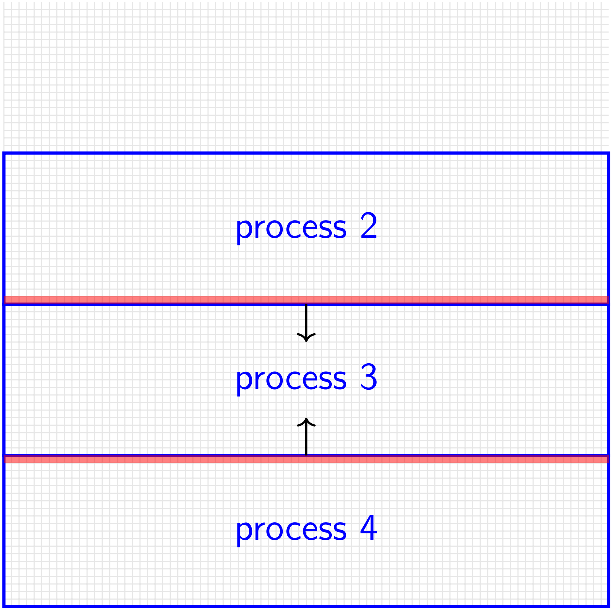
message passing game of life



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

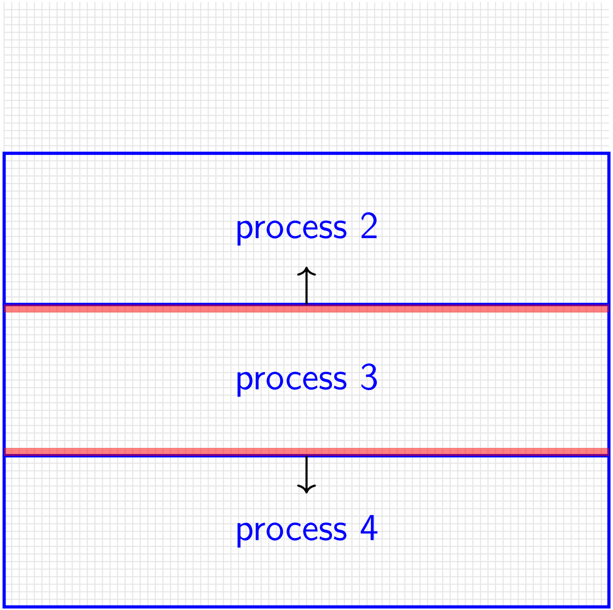
so process 3 also sends messages

message passing game of life



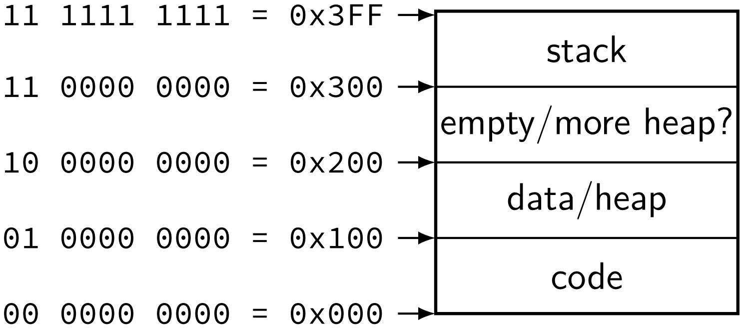
one possible pseudocode:
all even processes send messages
(while odd receives), then
all odd processes send messages
(while even receives)

message passing game of life

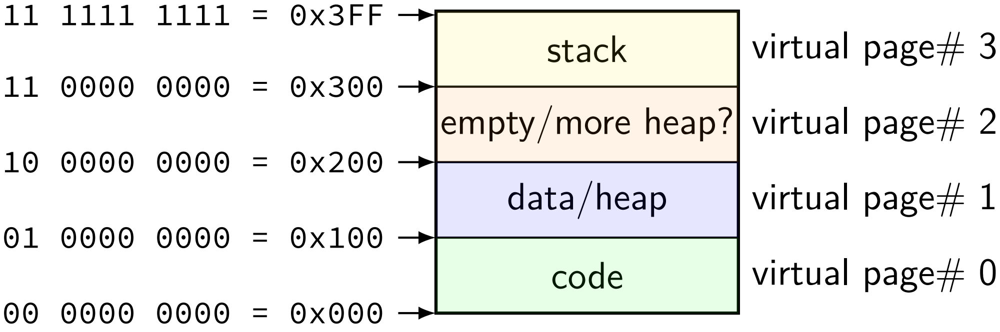


one possible pseudocode:
all even processes send messages (while odd receives), then
all odd processes send messages (while even receives)

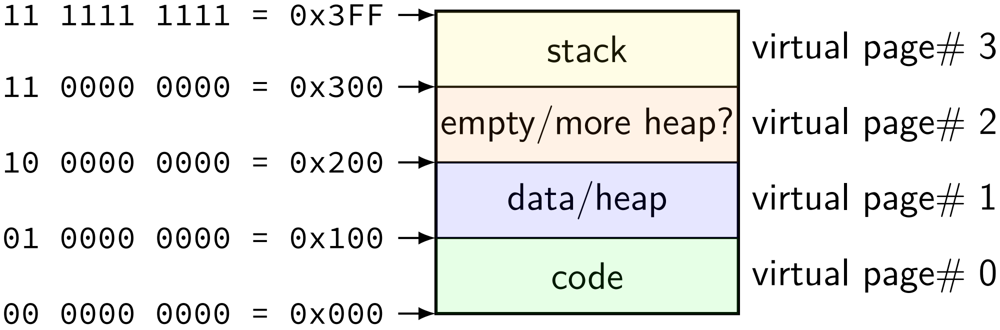
toy program memory



toy program memory

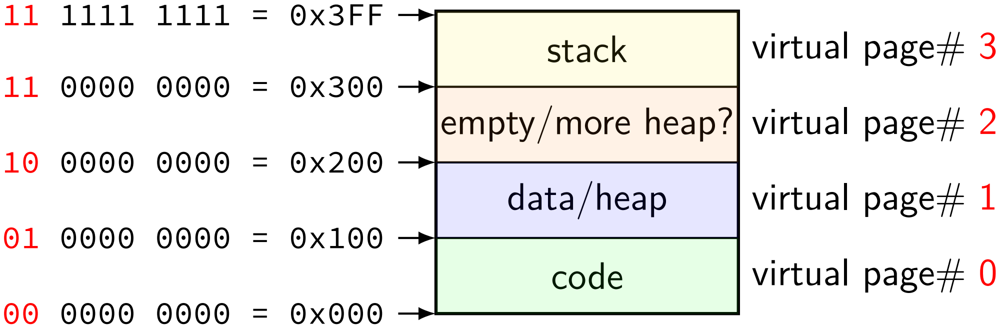


toy program memory



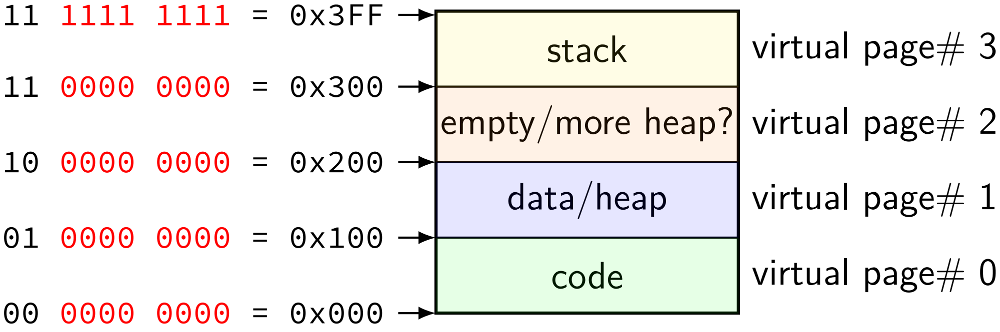
divide memory into pages (2^8 bytes in this case)
"virtual" = addresses the program sees

toy program memory



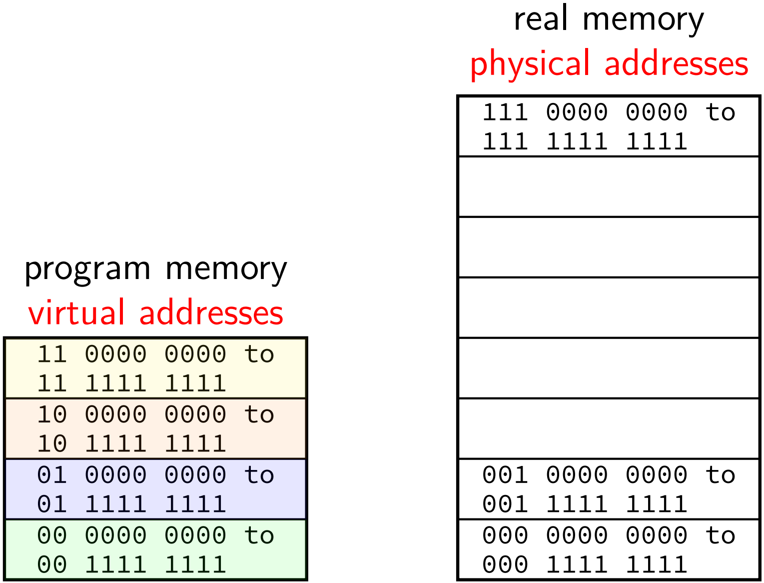
page number is upper bits of address
(because page size is power of two)

toy program memory

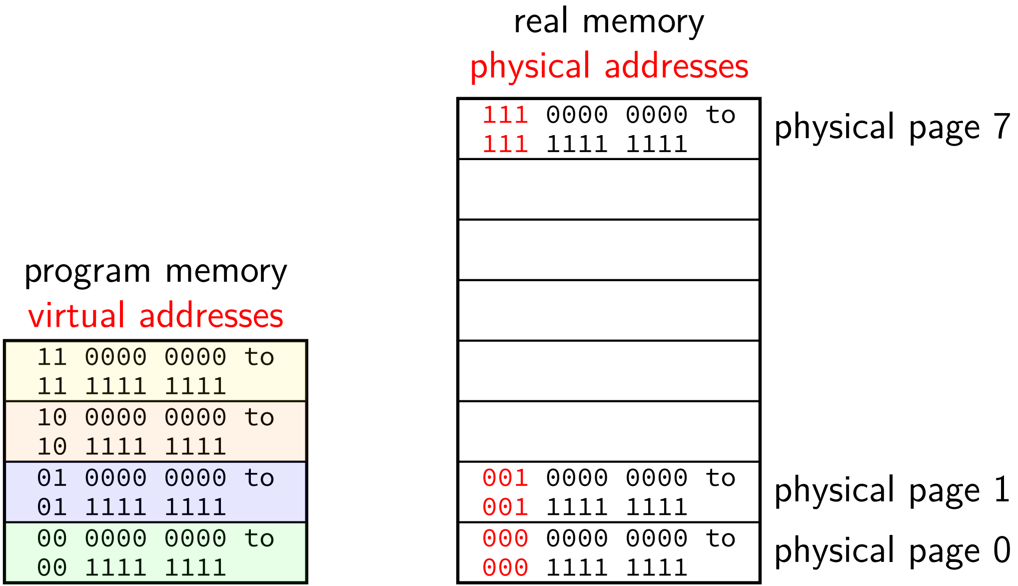


rest of address is called page offset

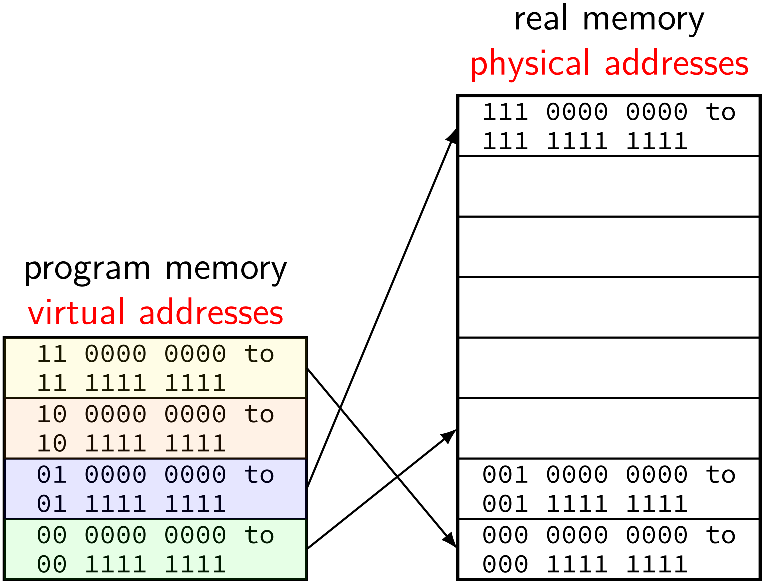
toy physical memory



toy physical memory

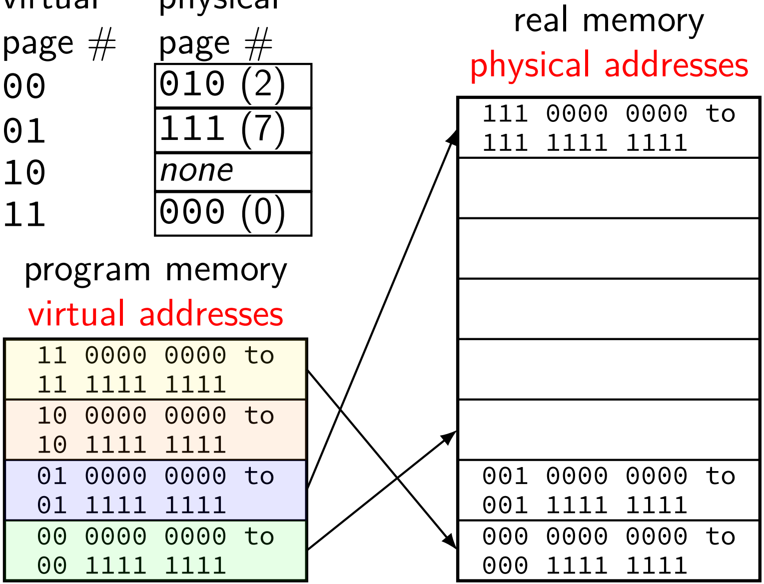


toy physical memory



toy physical memory

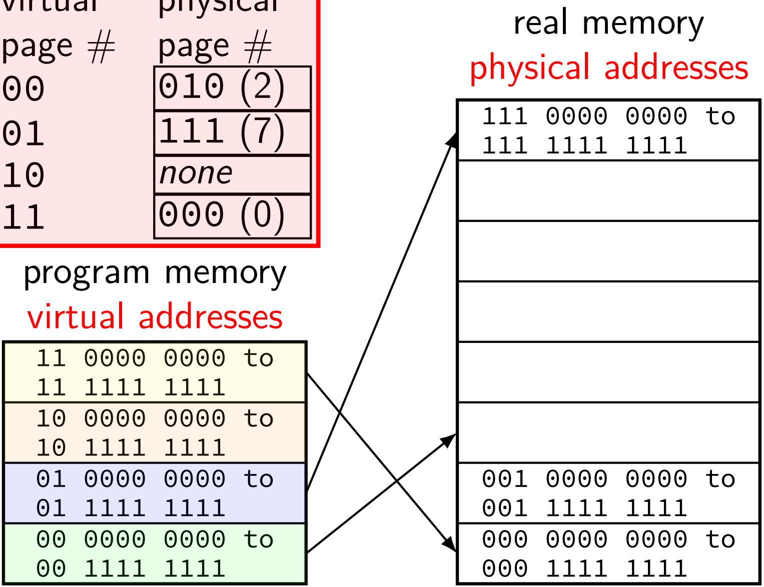
virtual page #	physical page #
00	010 (2)
01	111 (7)
10	*none*
11	000 (0)



toy physical memory

page table!

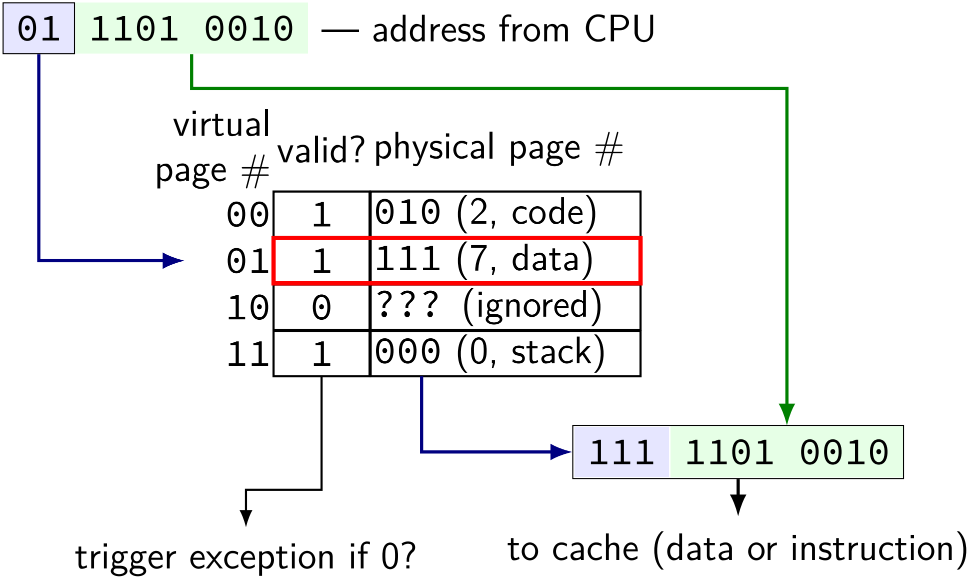
virtual page #	physical page #
00	010 (2)
01	111 (7)
10	*none*
11	000 (0)



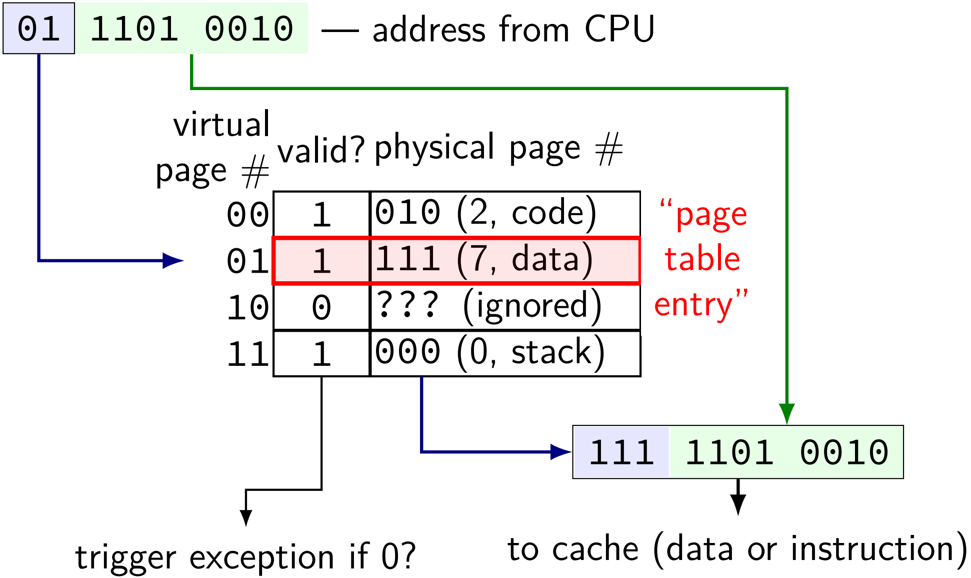
toy page table lookup

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

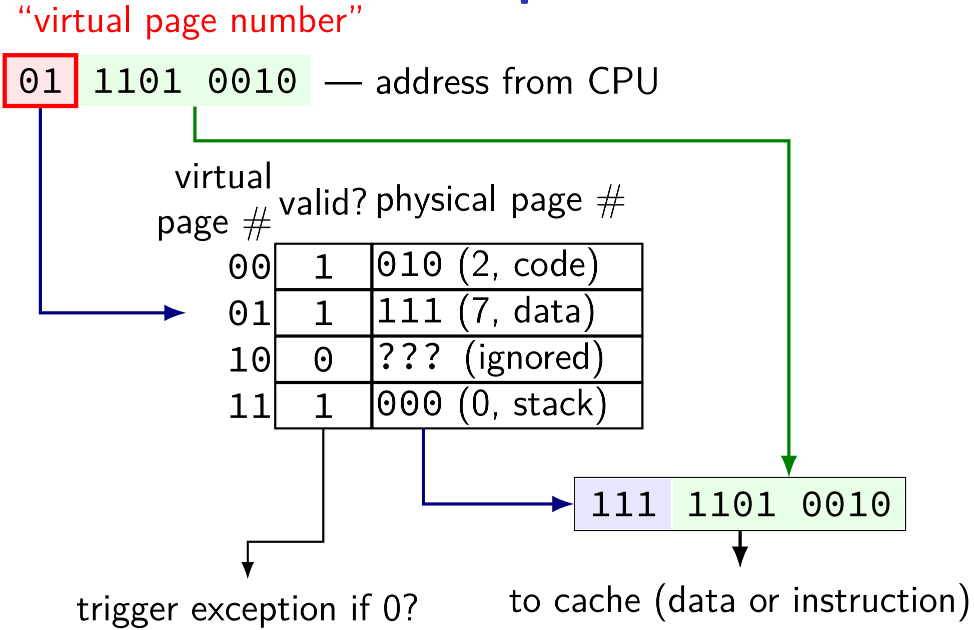
toy page table lookup



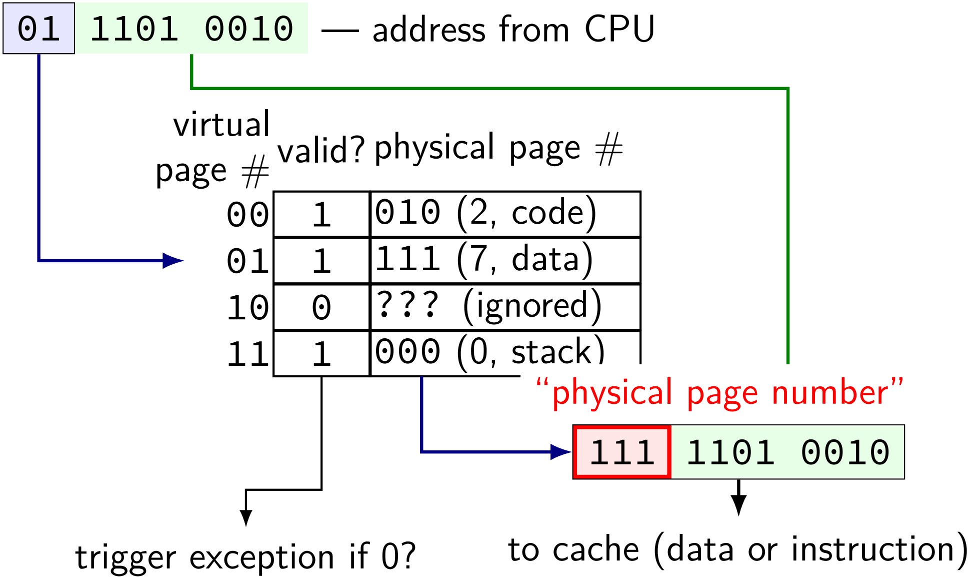
toy page table lookup



toy page table lookup

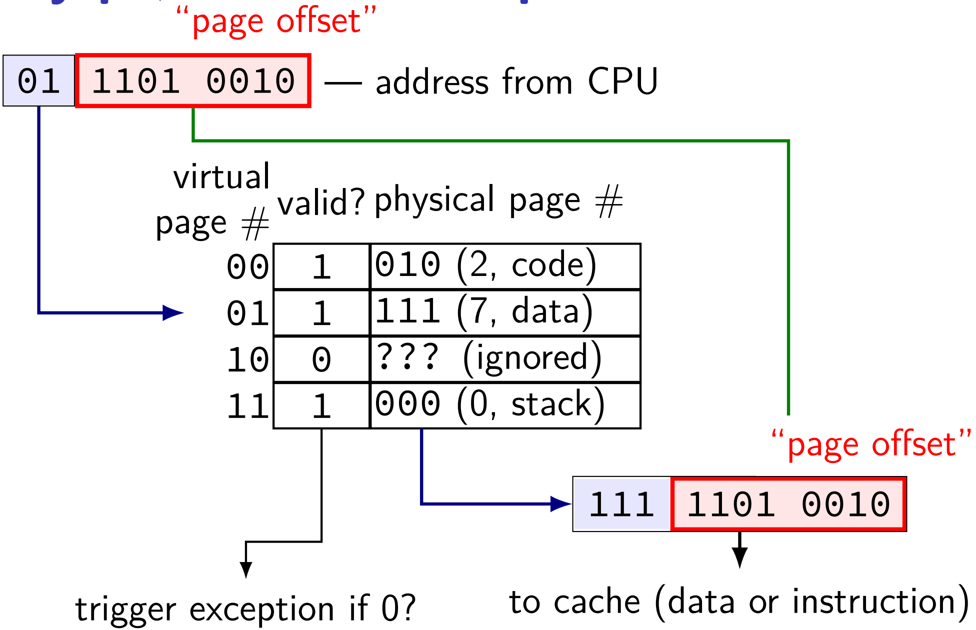


toy page table lookup



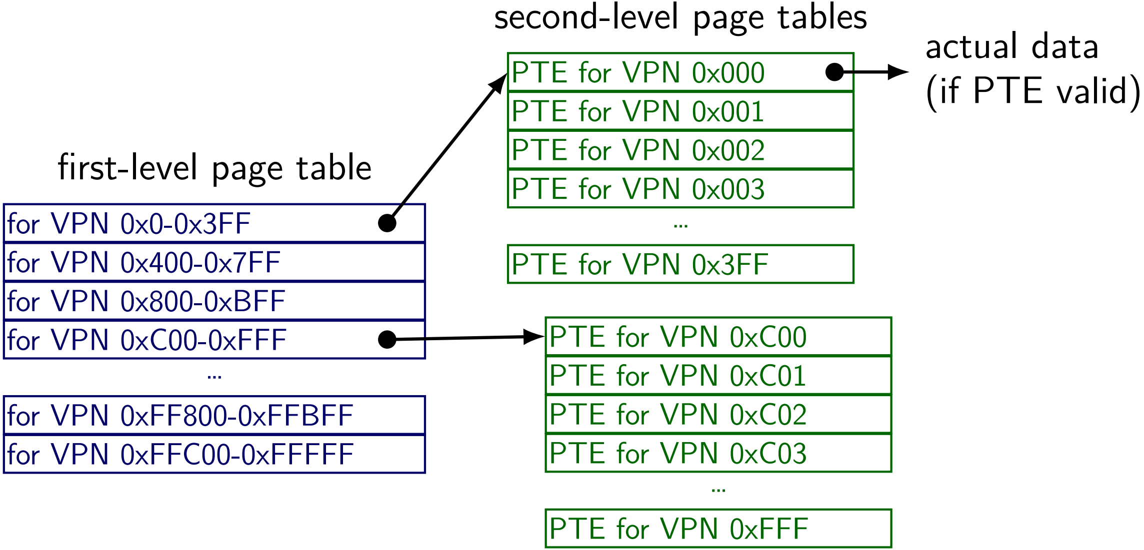
virtual page #	valid?	physical page #
00	1	010 (2, code)
01	1	111 (7, data)
10	0	??? (ignored)
11	1	000 (0, stack)

toy page table lookup



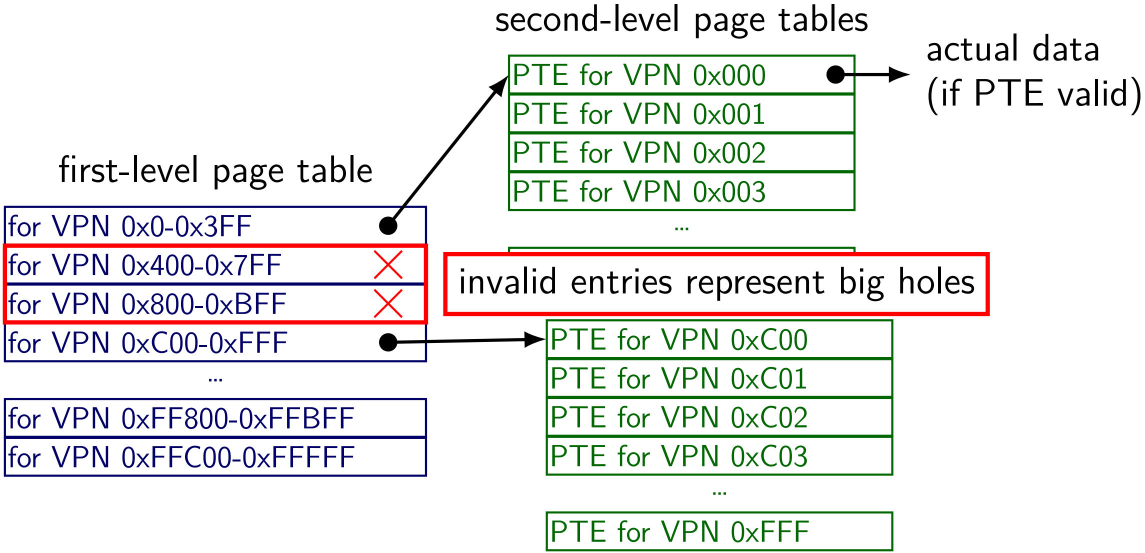
two-level page tables

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 tables

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

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`

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

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 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

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

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

first-level page table

for VPN 0x0-0x3FF	●
for VPN 0x400-0x7FF	✕
for VPN 0x800-0xBFF	✕
for VPN 0xC00-0xFFF	●
…	
for VPN 0xFF800-0xFFBFF	
for VPN 0xFFC00-0xFFFFF	

a second-level page table

VPN	valid	user?	write?	physical page # (of data)
0xC00	1	1	0	0x42443
0xC01	1	1	0	0x4A9DE
0xC02	1	1	0	0x5C001
0xC03	0	0	0	0x00000
0xC04	1	1	0	0x6C223
…	…	…	…	…
0xFFF	0	0	0	0x00000

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 page table

for VPN 0x0-0x3FF	●
for VPN 0x400-0x7FF	✕
for VPN 0x800-0xBFF	✕
for VPN 0xC00-0xFFF	●
...	
for VPN 0xFF800-0xFFBFF	
for VPN 0xFFC00-0xFFFFF	

a second-level page table

VPN	valid	user?	write?	physical page # (of data)
0xC00	1	1	0	0x42443
0xC01	1	1	0	0x4A9DE
0xC02	1	1	0	0x5C001
0xC03	0	0	0	0x00000
0xC04	1	1	0	0x6C223
...	...	...	...	...
0xFFF	0	0	0	0x00000

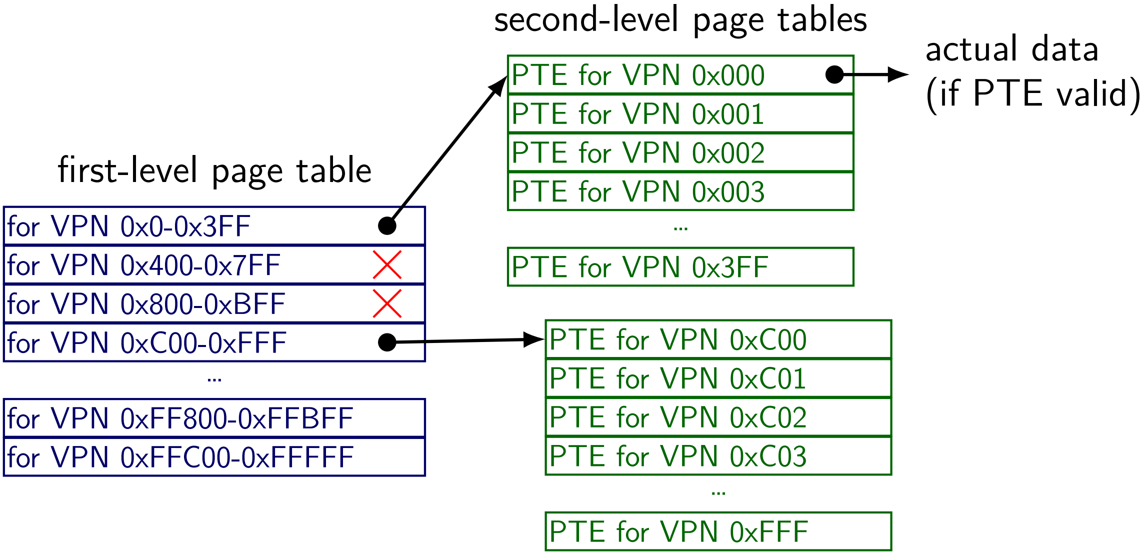
PTE for VPN 0xC03

...

PTE for VPN 0xFFF

two-level page tables

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



x86-32 pagetables: overall structure

xv6 header: mmu.h

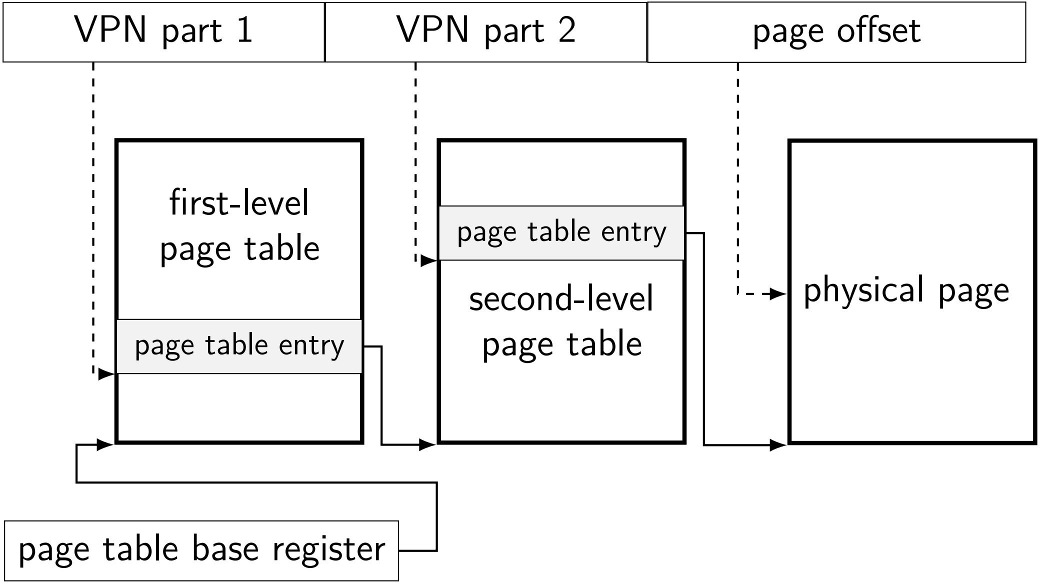
```
// A virtual address 'la' has a three-part structure as follows:
//
// +--------10------+-------10-------+---------12----------+
// | Page Directory |   Page Table   | Offset within Page  |
// |      Index     |      Index     |                     |
// +----------------+----------------+---------------------+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



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?

exercise

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?

x86-32 page table entries

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

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

x86-32 page table entries

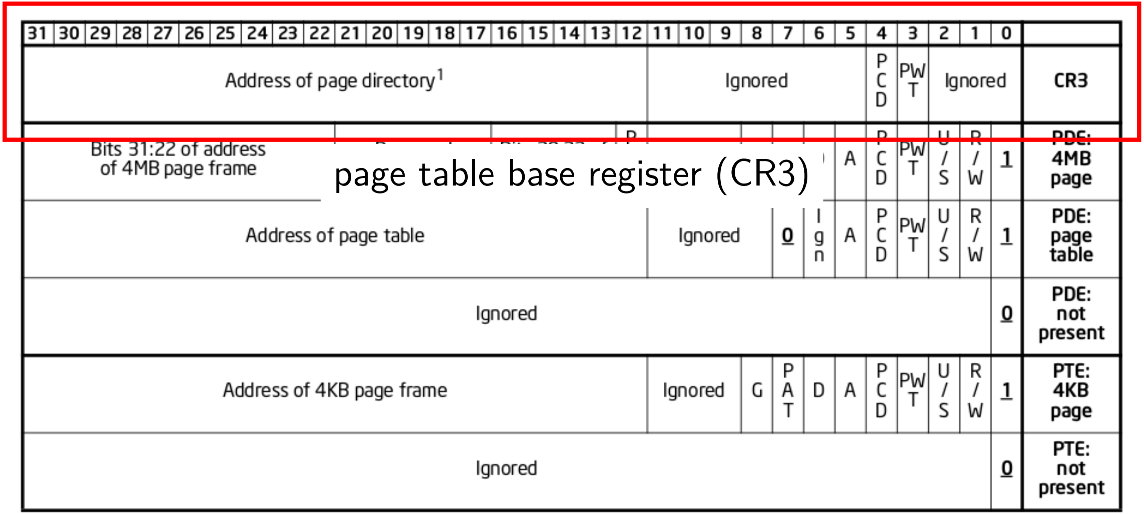


Figure 4-4. Formats of CR3 and Paging-Structure Entries with 32-Bit Paging

x86-32 page table entries

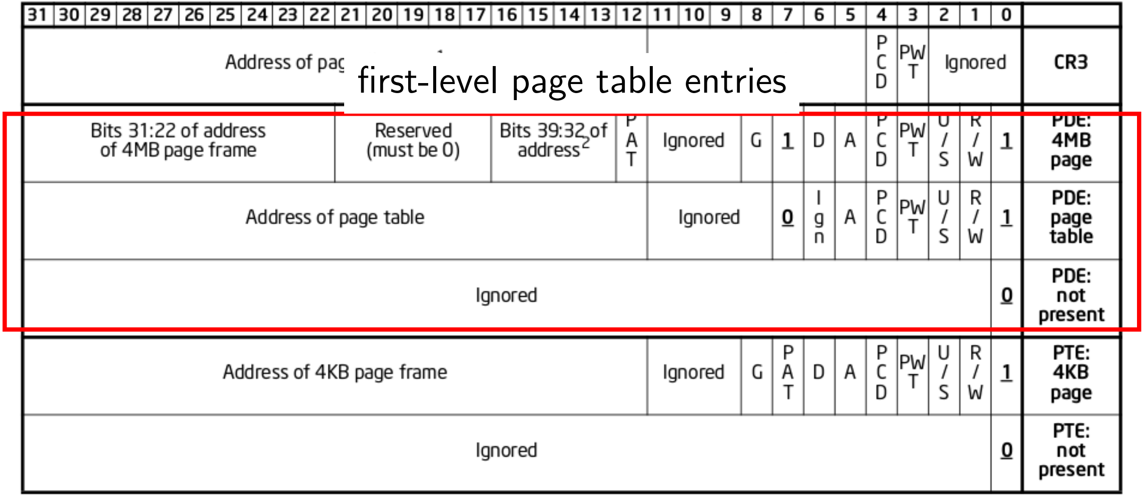


Figure 4-4. Formats of CR3 and Paging-Structure Entries with 32-Bit Paging

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 entries

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

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

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

makes constructing page table entries simpler:

```
physicalAddress | flagsBits
```

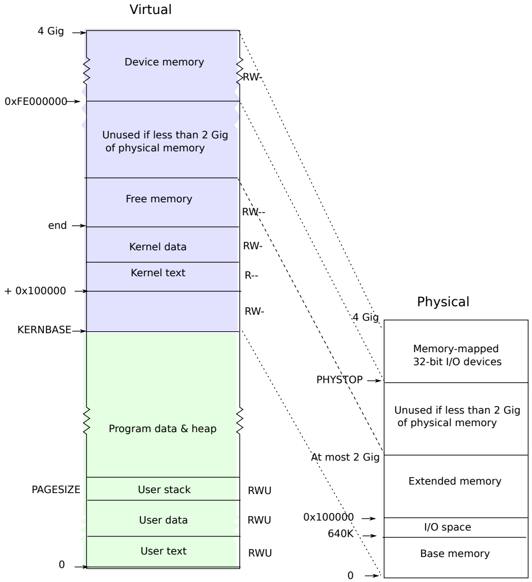
x86-32 pagetables: page table entries

xv6 header: mmu.h

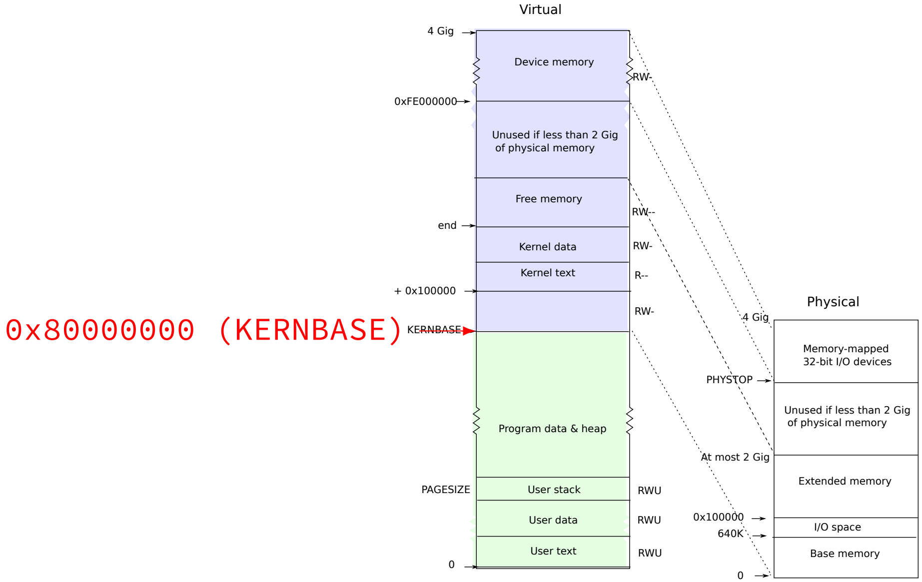
```
// Page table/directory entry flags.
#define PTE_P           0x001   // Present
#define PTE_W           0x002   // Writeable
#define PTE_U           0x004   // User
#define PTE_PWT         0x008   // Write-Through
#define PTE_PCD         0x010   // Cache-Disable
#define PTE_A           0x020   // 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)
```

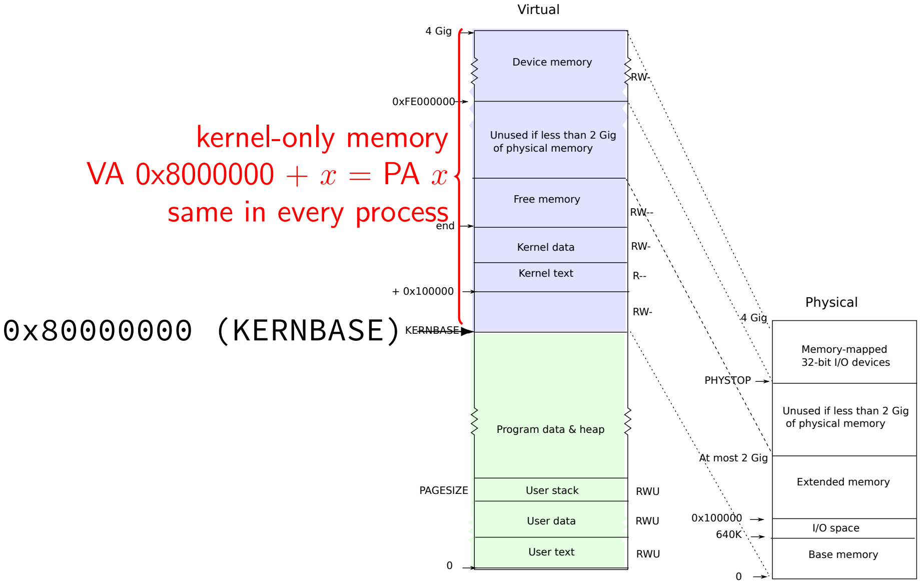
xv6 memory layout



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

P2V/V2P

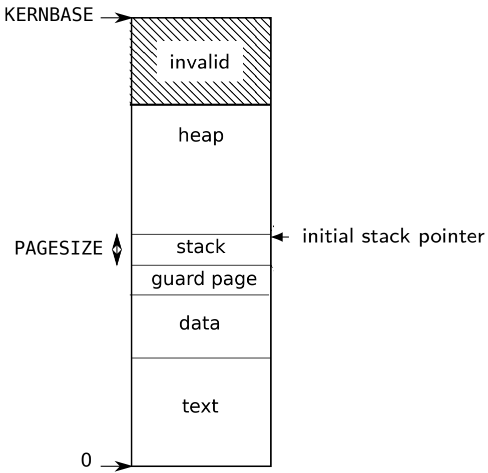
V2P(x) (virtual to physical)
convert *kernel* address x to physical address

subtract KERNBASE (0x8000 0000)

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

add KERNBASE (0x8000 0000)

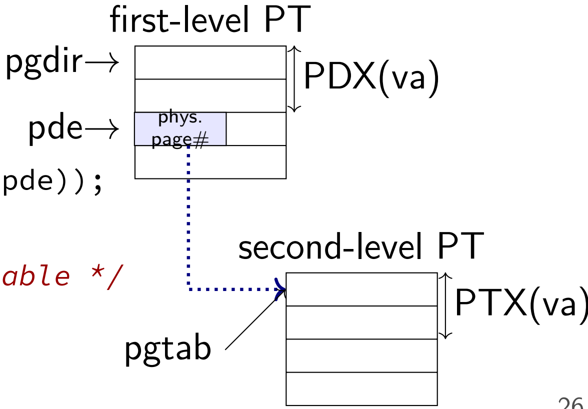
xv6 program memory



xv6: finding page table entries

```
// Return the address of the PTE in page table pgdir
// that corresponds to virtual address va.  If alloc!=0,
// create any required page table pages.
static pte_t *
walkpgdir(pde_t *pgdir, const void *va, int alloc)
{
  pde_t *pde;
  pte_t *pgtab;

  pde = &pgdir[PDX(va)];
  if(*pde & PTE_P){
    pgtab = (pte_t*)P2V(PTE_ADDR(*pde));
  } else {
    ... /* create new
              second-level page table */
  }
  return &pgtab[PTX(va)];
}
```

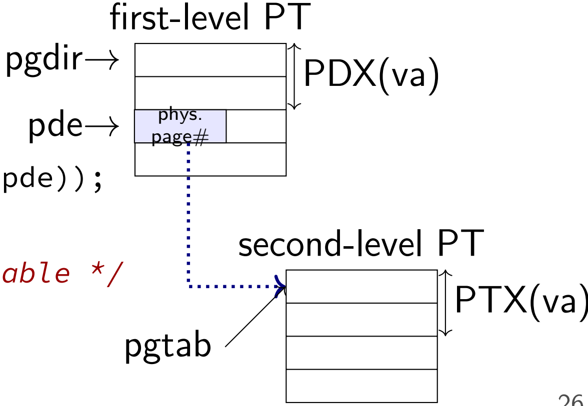


xv6: finding page table entries

pde_t — page directory entry
pte_t — page table entry
both aliases for uint (32-bit unsigned int)

```
// Return the address of
// that corresponds to vi
// create any required page table pages.
static pte_t *
walkpgdir(pde_t *pgdir, const void *va, int alloc)
{
  pde_t *pde;
  pte_t *pgtab;

  pde = &pgdir[PDX(va)];
  if(*pde & PTE_P){
    pgtab = (pte_t*)P2V(PTE_ADDR(*pde));
  } else {
    ... /* create new
              second-level page table */
  }
  return &pgtab[PTX(va)];
}
```

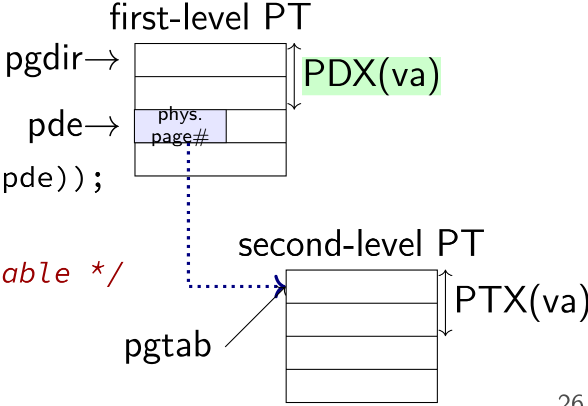


xv6: finding page table entries

PDX(va) — extract top 10 bits of va
used to index into first-level page table

```
// Return the address of th
// that corresponds to virtual address va.  If alloc!=0,
// create any required page table pages.
static pte_t *
walkpgdir(pde_t *pgdir, const void *va, int alloc)
{
  pde_t *pde;
  pte_t *pgtab;

  pde = &pgdir[PDX(va)];
  if(*pde & PTE_P){
    pgtab = (pte_t*)P2V(PTE_ADDR(*pde));
  } else {
    ... /* create new
              second-level page table */
  }
  return &pgtab[PTX(va)];
}
```



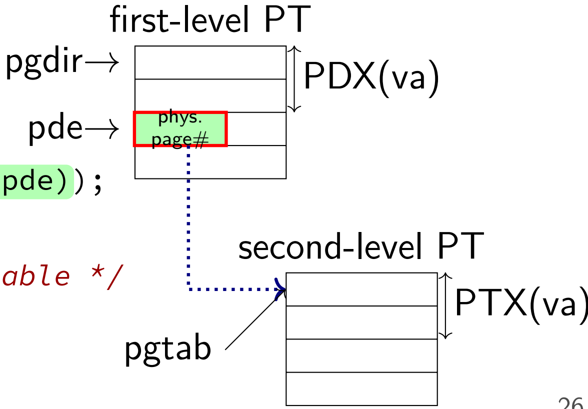
xv6: finding page table entries

PTE_ADDR(*pde) — return second-level page table address from first-level page table entry *pde

returns *physical address*

```
// Retu
// that
// create any required page table pages.
static pte_t *
walkpgdir(pde_t *pgdir, const void *va, int alloc)
{
  pde_t *pde;
  pte_t *pgtab;

  pde = &pgdir[PDX(va)];
  if(*pde & PTE_P){
    pgtab = (pte_t*)P2V(PTE_ADDR(*pde));
  } else {
    ... /* create new
             second-level page table */
  }
  return &pgtab[PTX(va)];
}
```



xv6: finding page table entries

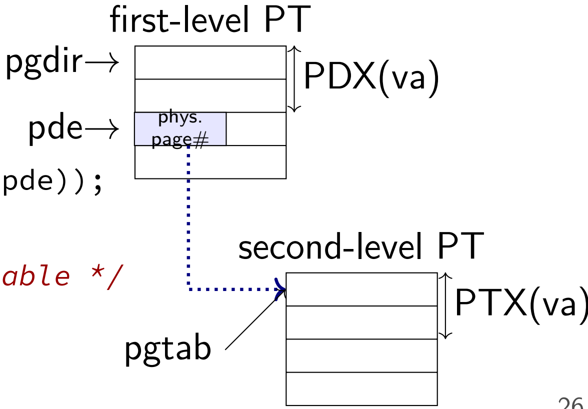
P2V — physical address to virtual addresss

by convention, kernel maps physical memory at address KERNBASE (will show setup later)

result is address that can access second-level page table

```
// Return th
// that corr
// create an
static pte_t
walkpgdir(pd
{
  pde_t *pde;
  pte_t *pgtab;

  pde = &pgdir[PDX(va)];
  if(*pde & PTE_P){
    pgtab = (pte_t*)P2V(PTE_ADDR(*pde));
  } else {
    ... /* create new
              second-level page table */
  }
  return &pgtab[PTX(va)];
}
```

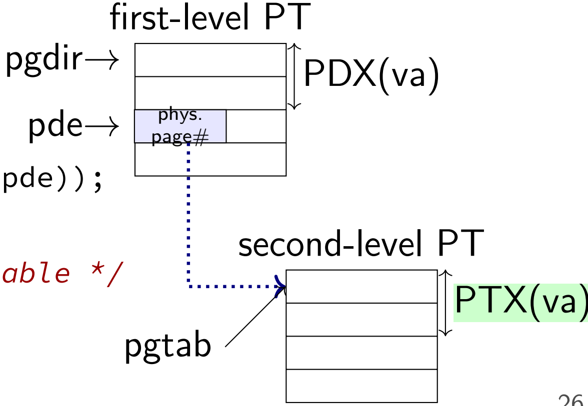


xv6: finding page table entries

lookup in second-level page table
PTX retrieves second-level page table index
(= bits 10-20 of va)

```
// Return the address o
// that corresponds to
// create any required page table pages.
static pte_t *
walkpgdir(pde_t *pgdir, const void *va, int alloc)
{
  pde_t *pde;
  pte_t *pgtab;

  pde = &pgdir[PDX(va)];
  if(*pde & PTE_P){
    pgtab = (pte_t*)P2V(PTE_ADDR(*pde));
  } else {
    ... /* create new
               second-level page table */
  }
  return &pgtab[PTX(va)];
}
```

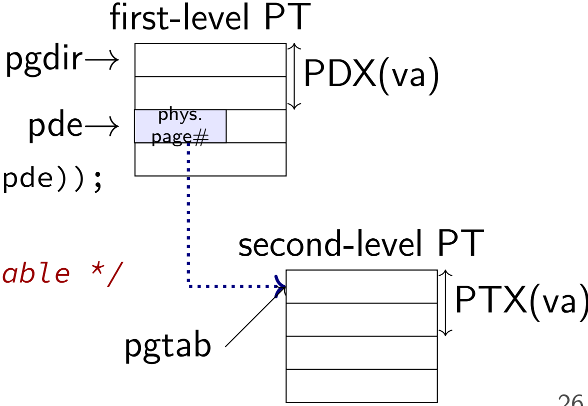


xv6: finding page table entries

```
// Return the address of the PTE in
// that corresponds to virtual addr
// create any required page table pa
static pte_t *
walkpgdir(pde_t *pgdir, const void *va, int alloc)
{
  pde_t *pde;
  pte_t *pgtab;

  pde = &pgdir[PDX(va)];
  if(*pde & PTE_P){
    pgtab = (pte_t*)P2V(PTE_ADDR(*pde));
  } else {
    ... /* create new
               second-level page table */
  }
  return &pgtab[PTX(va)];
}
```

if no second-level page table (present bit in first-level = 0) create one (if `alloc=1`) or return null (if `alloc=0`)



xv6: creating second-level page tables

```
...
if(*pde & PTE_P){
  pgtab = (pte_t*)P2V(PTE_ADDR(*pde));
} else {
  if(!alloc || (pgtab = (pte_t*)kalloc()) == 0)
    return 0;
  // Make sure all those PTE_P bits are zero.
  memset(pgtab, 0, PGSIZE);
  // The permissions here are overly generous, but they can
  // be further restricted by the permissions in the page table
  // entries, if necessary.
  *pde = V2P(pgtab) | PTE_P | PTE_W | PTE_U;
}
```

xv6: creating second-level page tables

return NULL if not trying to make new page table
otherwise use kalloc to allocate it

```
...
if(*pde & PTE_P){
  pgtab = (pte_t*)P2V(PTE_ADDR(*pde));
} else {
  if(!alloc || (pgtab = (pte_t*)kalloc()) == 0)
    return 0;
  // Make sure all those PTE_P bits are zero.
  memset(pgtab, 0, PGSIZE);
  // The permissions here are overly generous, but they can
  // be further restricted by the permissions in the page table
  // entries, if necessary.
  *pde = V2P(pgtab) | PTE_P | PTE_W | PTE_U;
}
```

xv6: creating second-level page tables

clear the page table
PTE = 0 → present = 0

```
...
if(*pde & PTE_P){
  pgtab = (pte_t*)P2V(PTE_ADDR(*pde));
} else {
  if(!alloc || (pgtab = (pte_t*)kalloc()) == 0)
    return 0;
  // Make sure all those PTE_P bits are zero.
  memset(pgtab, 0, PGSIZE);
  // The permissions here are overly generous, but they can
  // be further restricted by the permissions in the page table
  // entries, if necessary.
  *pde = V2P(pgtab) | PTE_P | PTE_W | PTE_U;
}
```

xv6: creating second-level page tables

```
...
if(*pde & PTE
  pgtab = (pt
} else {
  if(!alloc |
    return 0;
  // Make sur
  memset(pgta
  // The permissions here are overly generous, but they can
  // be further restricted by the permissions in the page table
  // entries, if necessary.
  *pde = V2P(pgtab) | PTE_P | PTE_W | PTE_U;
}
```

create a first-level page entry
with physical address of second-level page table
P for "present" (valid)
W for "writable" (pages access via may be writable)
U for "user-mode" (in addition to kernel)

second-level permission bits can restrict further

xv6: creating second-level page tables

```
...
if(*pde & PTE_P){
  pgtab = (pte_t*)P2V(PTE_ADDR(*pde));
} else {
  if(!alloc || (pgtab = (pte_t*)kalloc()) == 0)
    return 0;
  // Make sure all those PTE_P bits are zero.
  memset(pgtab, 0, PGSIZE);
  // The permissions here are overly generous, but they can
  // be further restricted by the permissions in the page table
  // entries, if necessary.
  *pde = V2P(pgtab) | PTE_P | PTE_W | PTE_U;
}
```

xv6: setting last-level page entries

```
static int
mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm)
{
  char *a, *last; pte_t *pte;

  a = (char*)PGROUNDDOWN((uint)va);
  last = (char*)PGROUNDDOWN(((uint)va) + size - 1);
  for(;;){
    if((pte = walkpgdir(pgdir, a, 1)) == 0)
      return -1;
    if(*pte & PTE_P)
      panic("remap");
    *pte = pa | perm | PTE_P;
    if(a == last)
      break;
    a += PGSIZE;
    pa += PGSIZE;
  }
  return 0;
```

xv6: setting last-level page entries

for each virtual page in range (va to va + size)
get its page table entry
(or fail if out of memory)

```
static int
mappages(pde_t *pgdi
{
  char *a, *last; pte_t *pte;

  a = (char*)PGROUNDDOWN((uint)va);
  last = (char*)PGROUNDDOWN(((uint)va) + size - 1);
  for(;;){
    if((pte = walkpgdir(pgdir, a, 1)) == 0)
      return -1;
    if(*pte & PTE_P)
      panic("remap");
    *pte = pa | perm | PTE_P;
    if(a == last)
      break;
    a += PGSIZE;
    pa += PGSIZE;
  }
  return 0;
```

xv6: setting last-level page entries

make sure it's not already set

```
static int
mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm)
{
  char *a, *last; pte_t *pte;

  a = (char*)PGROUNDDOWN((uint)va);
  last = (char*)PGROUNDDOWN(((uint)va) + size - 1);
  for(;;){
    if((pte = walkpgdir(pgdir, a, 1)) == 0)
      return -1;
    if(*pte & PTE_P)
      panic("remap");
    *pte = pa | perm | PTE_P;
    if(a == last)
      break;
    a += PGSIZE;
    pa += PGSIZE;
  }
  return 0;
```

xv6: setting last-level page entries

create page table entry
pointing to physical page at pa
with specified permission bits (write and/or user-mode)
and P for present

```
static int
mappages(pde
{
  char *a, *

  a = (char*)PGROUNDDOWN((uint)va);
  last = (char*)PGROUNDDOWN(((uint)va) + size - 1);
  for(;;){
    if((pte = walkpgdir(pgdir, a, 1)) == 0)
      return -1;
    if(*pte & PTE_P)
      panic("remap");
    *pte = pa | perm | PTE_P;
    if(a == last)
      break;
    a += PGSIZE;
    pa += PGSIZE;
  }
  return 0;
```

xv6: setting last-level page entries

advance to next physical page (pa) and next virtual page (va)

```
static int
mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm)
{
  char *a, *last; pte_t *pte;

  a = (char*)PGROUNDDOWN((uint)va);
  last = (char*)PGROUNDDOWN(((uint)va) + size - 1);
  for(;;){
    if((pte = walkpgdir(pgdir, a, 1)) == 0)
      return -1;
    if(*pte & PTE_P)
      panic("remap");
    *pte = pa | perm | PTE_P;
    if(a == last)
      break;
    a += PGSIZE;
    pa += PGSIZE;
  }
  return 0;
```

xv6: setting process page tables

step 1: create new page table with kernel mappings

- kernel code runs unchanged in every process's address space
- mappings unaccessible in user mode

step 2: load executable pages from executable file

- executable contains list of parts of file to load
- allocate new pages (`kalloc`)

step 3: allocate pages for heap, stack

xv6: setting process page tables

step 1: create new page table with kernel mappings

- kernel code runs unchanged in every process's address space
- mappings unaccessible in user mode

step 2: load executable pages from executable file

- executable contains list of parts of file to load
- allocate new pages (`kalloc`)

step 3: allocate pages for heap, stack

create new page table (kernel mappings)

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

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

create new page table (kernel mappings)

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

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

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

create new page table (kernel mappings)

initialize to 0 — every page invalid

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

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

create new page table (kernel mappings)

iterate through list of kernel-space mappings
everything above address 0x8000 0000

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

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

create new page table (kernel mappings)

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

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

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

xv6: setting process page tables

step 1: create new page table with kernel mappings

- kernel code runs unchanged in every process's address space
- mappings unaccessible in user mode

step 2: load executable pages from executable file

- executable contains list of parts of file to load
- allocate new pages (`kalloc`)

step 3: allocate pages for heap, stack

reading executables (headers)

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

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

reading executables (headers)

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

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

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

allocating user pages

```
allocuvm(pde_t *pgdir, uint oldsz, uint newsz)
{
  ...
  a = PGROUNDUP(oldsz);
  for(; a < newsz; a += PGSIZE){
    mem = kalloc();
    if(mem == 0){
      cprintf("allocuvm␣out␣of␣memory\n");
      deallocuvm(pgdir, newsz, oldsz);
      return 0;
    }
    memset(mem, 0, PGSIZE);
    if(mappages(pgdir, (char*)a, PGSIZE, V2P(mem), PTE_W|PTE_U) < 0)
      cprintf("allocuvm␣out␣of␣memory␣(2)\n");
      deallocuvm(pgdir, newsz, oldsz);
      kfree(mem);
      return 0;
    }
  }
```

allocating user pages

allocate a new, zero page

```
allocuvm(pde_t *pgdir, uint oldsz, uint newsz)
{
  ...
  a = PGROUNDUP(oldsz);
  for(; a < newsz; a += PGSIZE){
    mem = kalloc();
    if(mem == 0){
      cprintf("allocuvm out of memory\n");
      deallocuvm(pgdir, newsz, oldsz);
      return 0;
    }
    memset(mem, 0, PGSIZE);
    if(mappages(pgdir, (char*)a, PGSIZE, V2P(mem), PTE_W|PTE_U) < 0)
      cprintf("allocuvm out of memory (2)\n");
      deallocuvm(pgdir, newsz, oldsz);
      kfree(mem);
      return 0;
    }
  }
```

allocating user pages

add page to second-level page table

```
allocuvm(pde_t *pgdir, uint oldsz, uint newsz)
{
  ...
  a = PGROUNDUP(oldsz);
  for(; a < newsz; a += PGSIZE){
    mem = kalloc();
    if(mem == 0){
      cprintf("allocuvm out of memory\n");
      deallocuvm(pgdir, newsz, oldsz);
      return 0;
    }
    memset(mem, 0, PGSIZE);
    if(mappages(pgdir, (char*)a, PGSIZE, V2P(mem), PTE_W|PTE_U) < 0){
      cprintf("allocuvm out of memory (2)\n");
      deallocuvm(pgdir, newsz, oldsz);
      kfree(mem);
      return 0;
    }
  }
```

allocating user pages

same function used to allocate memory for heap

```
allocuvm(pde_t *pgdir, uint oldsz, uint newsz)
{
   ...
   a = PGROUNDUP(oldsz);
   for(; a < newsz; a += PGSIZE){
     mem = kalloc();
     if(mem == 0){
       cprintf("allocuvm␣out␣of␣memory\n");
       deallocuvm(pgdir, newsz, oldsz);
       return 0;
     }
     memset(mem, 0, PGSIZE);
     if(mappages(pgdir, (char*)a, PGSIZE, V2P(mem), PTE_W|PTE_U) < 0)
       cprintf("allocuvm␣out␣of␣memory␣(2)\n");
       deallocuvm(pgdir, newsz, oldsz);
       kfree(mem);
       return 0;
     }
   }
```

reading executables (headers)

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

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

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

loading user pages from executable

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

loading user pages from executable

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

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

loading user pages from executable

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

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

loading user pages from executable

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

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

kalloc/kfree

kalloc/kfree — xv6's physical memory allocator

allocates/deallocates whole pages only

keep linked list of free pages

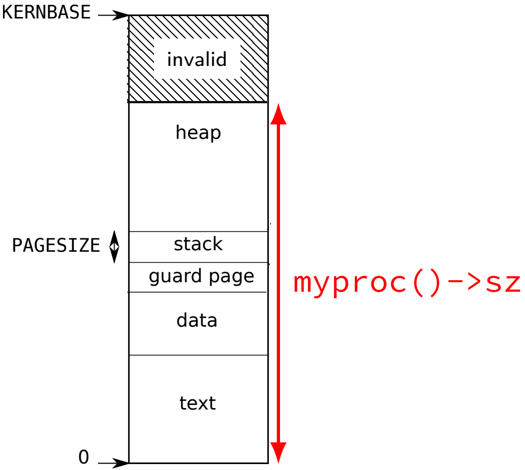
- list nodes — stored in corresponding free page itself
- kalloc — return first page in list
- kfree — add page to list

linked list created at boot

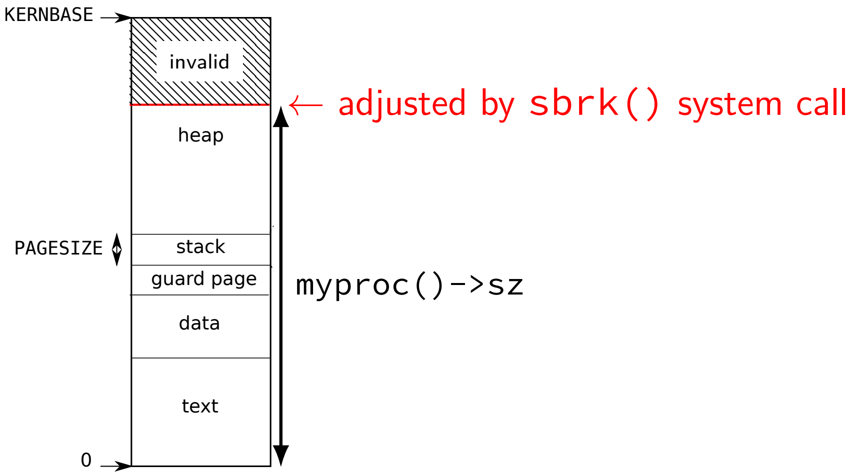
usuable memory fixed size (224MB)

- determined by PHYSTOP in `memlayout.h`

xv6 program memory



xv6 program memory



xv6 heap allocation

xv6: every process has a heap at the top of its address space

- yes, this is unlike Linux where heap is below stack

tracked in `struct proc` with `sz`

- = last valid address in process

position changed via `sbrk(amount)` system call

- sets `sz += amount`
- same call exists in Linux, etc. — but also others

sbrk

```
sys_sbrk()
{
  if(argint(0, &n) < 0)
    return -1;
  addr = myproc()->sz;
  if(growproc(n) < 0)
    return -1;
  return addr;
}
```

sbrk

`sz`: current top of heap

```
sys_sbrk()
{
  if(argint(0, &n) < 0)
    return -1;
  addr = myproc()->sz;
  if(growproc(n) < 0)
    return -1;
  return addr;
}
```

sbrk

sbrk(N): grow heap by N (shrink if negative)

```
sys_sbrk()
{
  if(argint(0, &n) < 0)
    return -1;
  addr = myproc()->sz;
  if(growproc(n) < 0)
    return -1;
  return addr;
}
```

sbrk

returns old top of heap (or -1 on out-of-memory)

```
sys_sbrk()
{
  if(argint(0, &n) < 0)
    return -1;
  addr = myproc()->sz;
  if(growproc(n) < 0)
    return -1;
  return addr;
}
```

growproc

```
growproc(int n)
{
  uint sz;
  struct proc *curproc = myproc();

  sz = curproc->sz;
  if(n > 0){
    if((sz = allocuvm(curproc->pgdir, sz, sz + n)) == 0)
      return -1;
  } else if(n < 0){
    if((sz = deallocuvm(curproc->pgdir, sz, sz + n)) == 0)
      return -1;
  }
  curproc->sz = sz;
  switchuvm(curproc);
  return 0;
}
```

growproc

allocuvm — same function used to allocate initial space
maps pages for addresses `sz` to `sz + n`
calls `kalloc` to get each page

```
growproc(int
{
  uint sz;
  struct proc *curproc = myproc();

  sz = curproc->sz;
  if(n > 0){
    if((sz = allocuvm(curproc->pgdir, sz, sz + n)) == 0)
      return -1;
  } else if(n < 0){
    if((sz = deallocuvm(curproc->pgdir, sz, sz + n)) == 0)
      return -1;
  }
  curproc->sz = sz;
  switchuvm(curproc);
  return 0;
}
```

xv6 page faults (now)

fault from accessing page table entry marked 'not-present'

xv6: prints an error and kills process:

```
*((int*) 0x800444) = 1;
...
/* in trap.c: */
    cprintf("pid␣%d␣%s:␣trap␣%d␣err␣%d␣on␣cpu␣%d␣"
            "eip␣0x%x␣addr␣0x%x--kill␣proc\n",
            myproc()->pid, myproc()->name, tf->trapno,
            tf->err, cpuid(), tf->eip, rcr2());
    myproc()->killed = 1;
```

```
pid 4 processname: trap 14 err 6 on cpu 0 eip 0x1a addr 0x800444--k
```

14 = T_PGFLT

special register CR2 contains faulting address

xv6 page faults (now)

fault from accessing page table entry marked 'not-present'

xv6: prints an error and kills process:

```
*((int*) 0x800444) = 1;
...
/* in trap.c: */
    cprintf("pid␣%d␣%s:␣trap␣%d␣err␣%d␣on␣cpu␣%d␣"
            "eip␣0x%x␣addr␣0x%x--kill␣proc\n",
            myproc()->pid, myproc()->name, tf->trapno,
            tf->err, cpuid(), tf->eip, rcr2());
    myproc()->killed = 1;
```

```
pid 4 processname: trap 14 err 6 on cpu 0 eip 0x1a addr 0x800444--k
```

14 = T_PGFLT

special register CR2 contains faulting address

xv6 page faults (now)

fault from accessing page table entry marked 'not-present'

xv6: prints an error and kills process:

```
*((int*) 0x800444) = 1;
...
/* in trap.c: */
    cprintf("pid␣%d␣%s:␣trap␣%d␣err␣%d␣on␣cpu␣%d␣"
            "eip␣0x%x␣addr␣0x%x--kill␣proc\n",
            myproc()->pid, myproc()->name, tf->trapno,
            tf->err, cpuid(), tf->eip, rcr2());
    myproc()->killed = 1;
```

```
pid 4 processname: trap 14 err 6 on cpu 0 eip 0x1a addr 0x800444--k
```

14 = T_PGFLT

special register CR2 contains faulting address

xv6: if one handled page faults

returning from page fault handler without killing process

…retries the failing instruction

can use to update the page table — “just in time”

```
if (tf->trapno == T_PGFLT) {
  void *address = (void *) rcr2();
  if (is_address_okay(myproc(), address)) {
      setup_page_table_entry_for(myproc(), address);
      // return from fault, retry access
  } else {
      // actual segfault, kill process
      cprintf("...");
      myproc()->killed = 1;
  }
}
```

xv6: if one handled page faults

check *process control block* to see if access okay

returning from page fault handler without killing process

…retries the failing instruction

can use to update the page table — "just in time"

```
if (tf->trapno == T_PGFLT) {
  void *address = (void *) rcr2();
  if (is_address_okay(myproc(), address)) {
      setup_page_table_entry_for(myproc(), address);
      // return from fault, retry access
  } else {
      // actual segfault, kill process
      cprintf("...");
      myproc()->killed = 1;
  }
}
```

xv6: if one handled page faults

if so, setup the page table so it works next time
i.e. immediately after returning from fault

returning from page

…retries the failing instruction

can use to update the page table — “just in time”

```
if (tf->trapno == T_PGFLT) {
  void *address = (void *) rcr2();
  if (is_address_okay(myproc(), address)) {
      setup_page_table_entry_for(myproc(), address);
      // return from fault, retry access
  } else {
      // actual segfault, kill process
      cprintf("...");
      myproc()->killed = 1;
  }
}
```

extra data structures needed

OSs can do all sorts of tricks with page tables

...but more bookkeeping is required

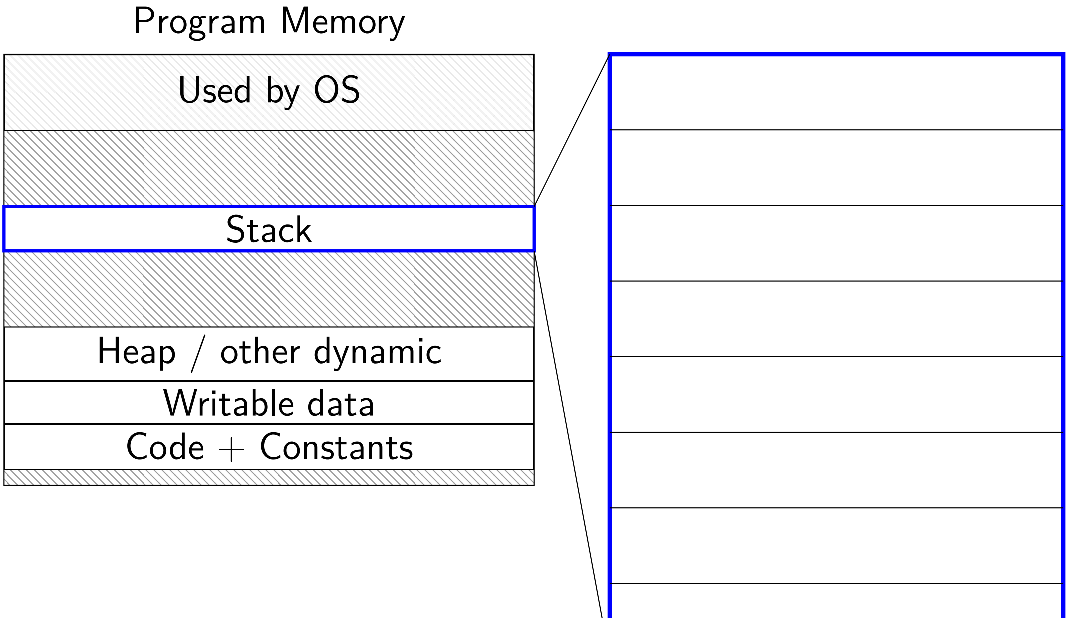
tracking what processes think they have in memory

- since page table won't tell the whole story
- OS will change page table

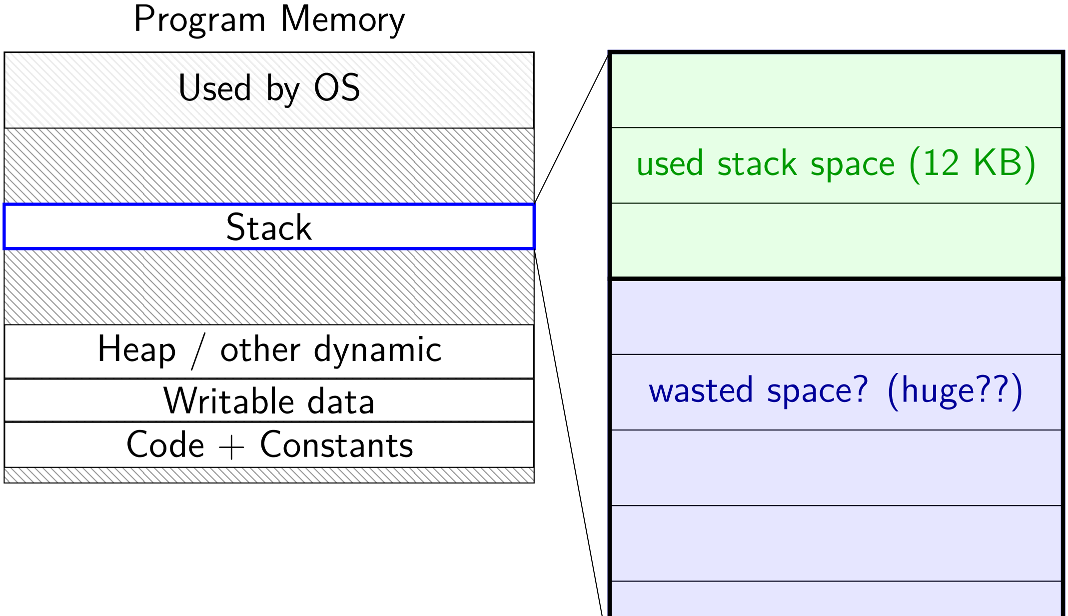
tracking how physical pages are used in page tables

- multiple processes might want same data = same page

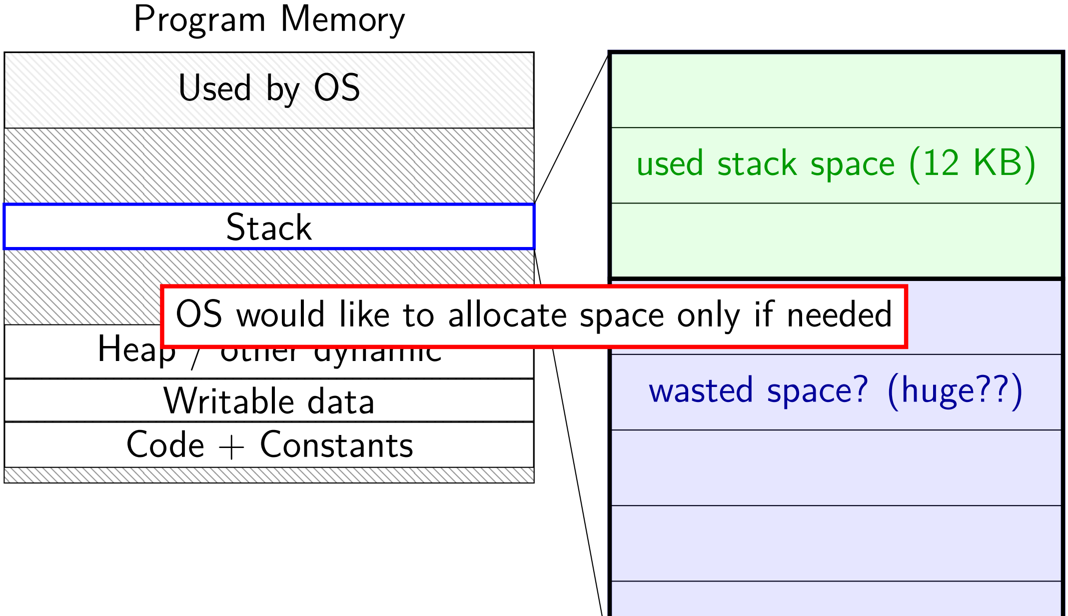
space on demand



space on demand



space on demand



OS would like to allocate space only if needed

allocating space on demand

`%rsp = 0x7FFFC000`

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

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

VPN	valid?	physical page
…	…	…
0x7FFFB	0	---
0x7FFFC	1	0x200DF
0x7FFFD	1	0x12340
0x7FFFE	1	0x12347
0x7FFFF	1	0x12345
…	…	…

allocating space on demand

%rsp = 0x7FFFC000

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

page fault!

VPN	valid?	physical page
…	…	…
0x7FFFB	0	---
0x7FFFC	1	0x200DF
0x7FFFD	1	0x12340
0x7FFFE	1	0x12347
0x7FFFF	1	0x12345
…	…	…

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

allocating space on demand

`%rsp = 0x7FFFC000`

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

restarted

VPN	valid?	physical page
…	…	…
0x7FFFB	1	0x200D8
0x7FFFC	1	0x200DF
0x7FFFD	1	0x12340
0x7FFFE	1	0x12347
0x7FFFF	1	0x12345
…	…	…

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

xv6: adding space on demand

```
struct proc {
  uint sz;    // Size of process memory (bytes)
  ...
};
```

adding allocate on demand logic:

on page fault: if address $\geq$ sz
- kill process — out of bounds

on page fault: if address $<$ sz
- find virtual page number of address
- allocate page of memory, add to page table
- return from interrupt

versus more complicated OSes

range of valid addresses is not just 0 to maximum

need some more complicated data structure to represent

will get to that later

fast copies

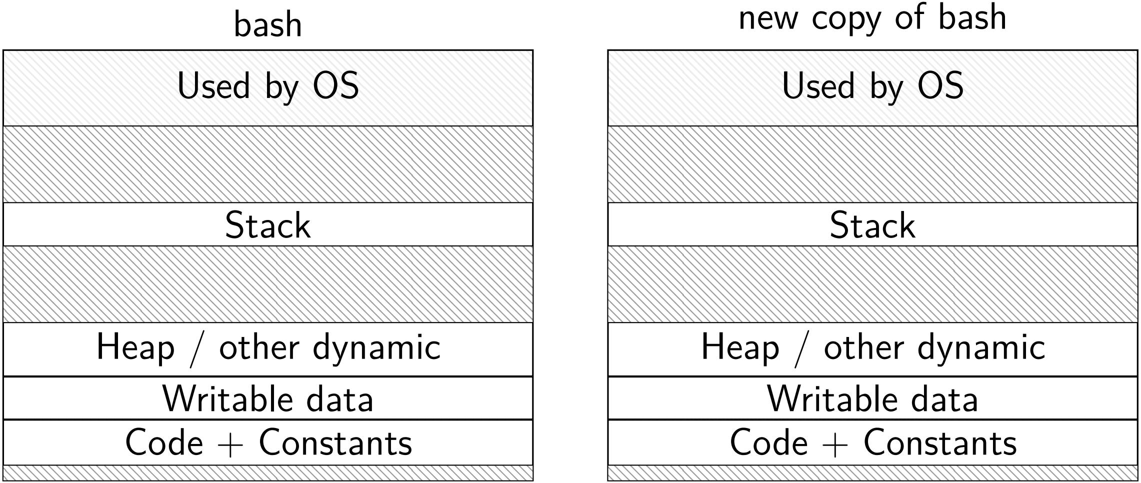
recall : `fork()`

creates a copy of an entire program!

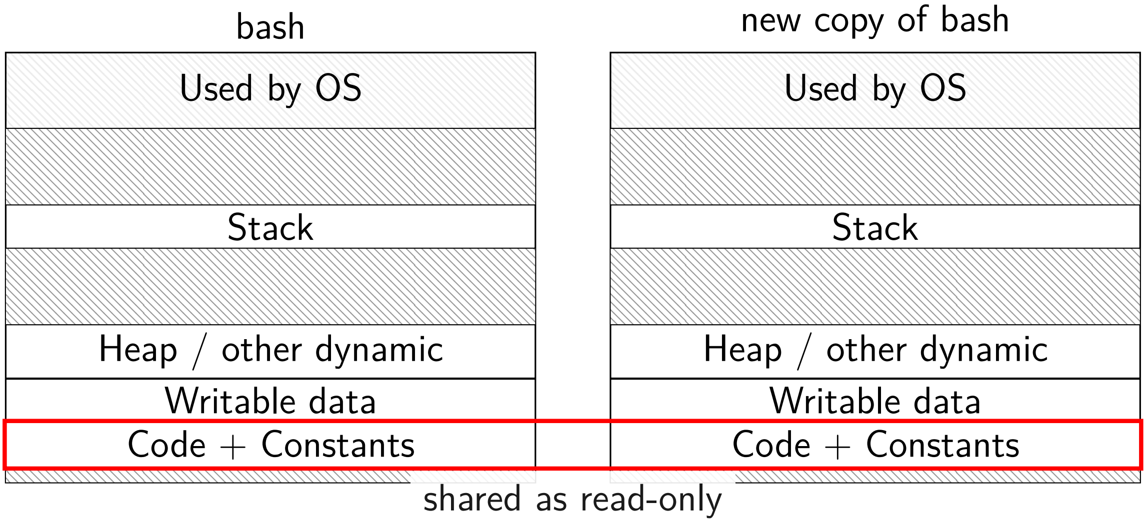
(usually, the copy then calls `execve` — replaces itself with another program)

how isn't this really slow?

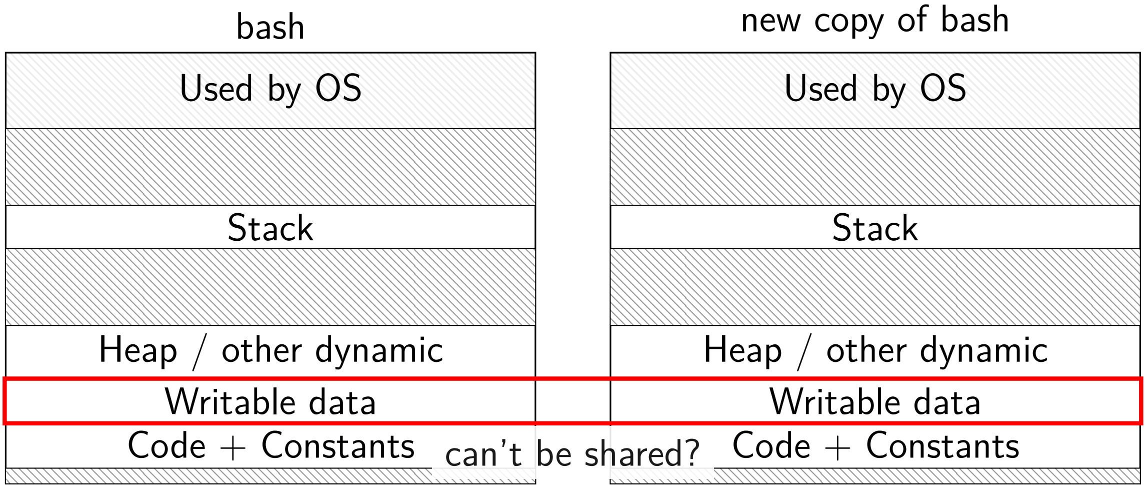
do we really need a complete copy?



do we really need a complete copy?



do we really need a complete copy?



trick for extra sharing

sharing writeable data is fine — until either process modifies the copy

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

copy-on write cases

trying to write forbidden page (e.g. kernel memory)
- kill program instead of making it writable

trying to write read-only page and…

only one page table entry refers to it
- make it writeable
- return from fault

multiple process's page table entries refer to it
- copy the page
- replace read-only page table entry to point to copy
- return from fault

page tables in memory

page table entry layout

valid (bit 15)	kernel (bit 14)	physical page # (bits 4–13)	unused (bit 0-3)

page tables in memory

page table entry layout

valid (bit 15)	kernel (bit 14)	physical page # (bits 4–13)	unused (bit 0-3)

page table base register

`0x00010000`

page tables in memory

page table entry layout

valid (bit 15)	kernel (bit 14)	physical page # (bits 4–13)	unused (bit 0-3)

page table base register

0x00010000

physical memory

addresses	bytes
0x00000000-1	00000000 00000000
…	
0x00010000-1	00000000 00000000
0x00010002-3	10100010 01100000
0x00010004-5	10000010 11000000
0x00010006-7	10110000 00110000
…	
0x000101FE-F	10001110 10000000
0x00010200-1	10100010 00111010

page tables in memory

page table entry layout

valid (bit 15)	kernel (bit 14)	physical page # (bits 4–13)	unused (bit 0-3)

page table base register

0x00010000

physical memory

addresses	bytes
0x00000000-1	00000000 00000000
...	
0x00010000-1 (← page table base register)	00000000 00000000
0x00010002-3	10100010 01100000
0x00010004-5	10000010 11000000
0x00010006-7	10110000 00110000
...	
0x000101FE-F	10001110 10000000
0x00010200-1	10100010 00111010

page tables in memory

page table entry layout

valid (bit 15)	kernel (bit 14)	physical page # (bits 4–13)	unused (bit 0-3)

page table base register

0x00010000

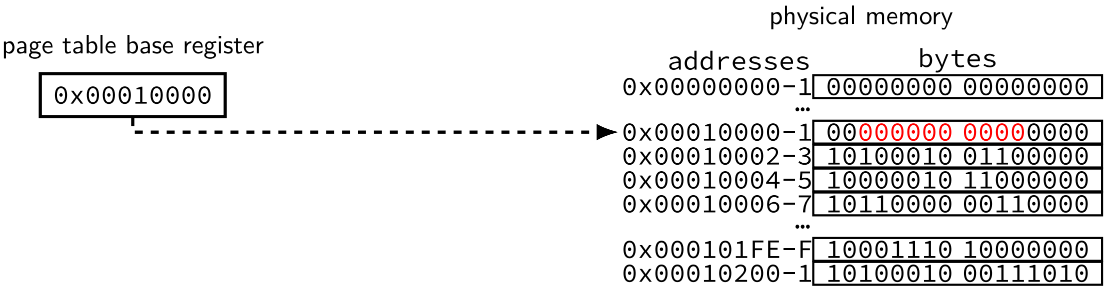
physical memory

addresses	bytes
0x00000000−1	00000000 00000000
...	
0x00010000−1	00000000 00000000
0x00010002−3	10100010 01100000
0x00010004−5	10000010 11000000
0x00010006−7	10110000 00110000
...	
0x000101FE−F	10001110 10000000
0x00010200−1	10100010 00111010

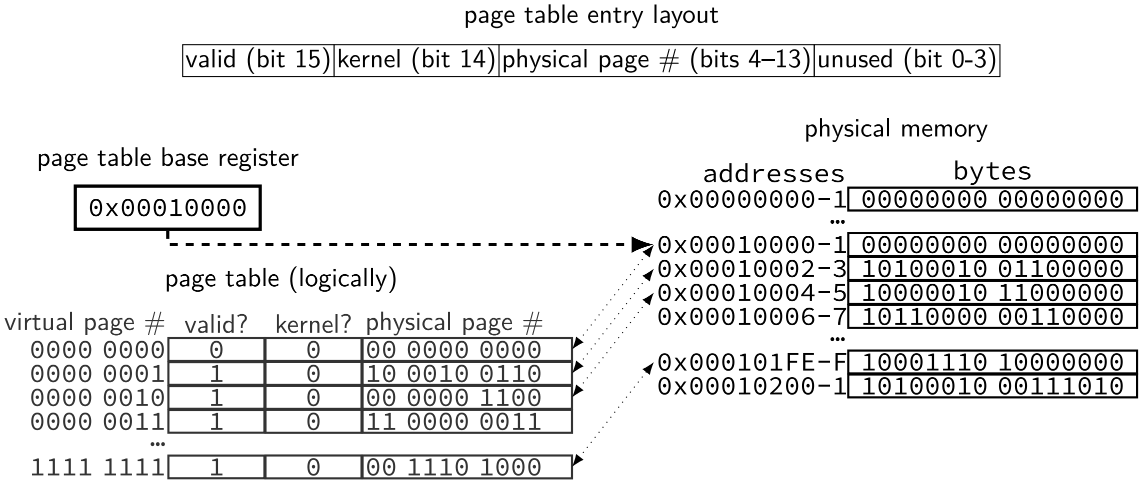
page tables in memory

page table entry layout

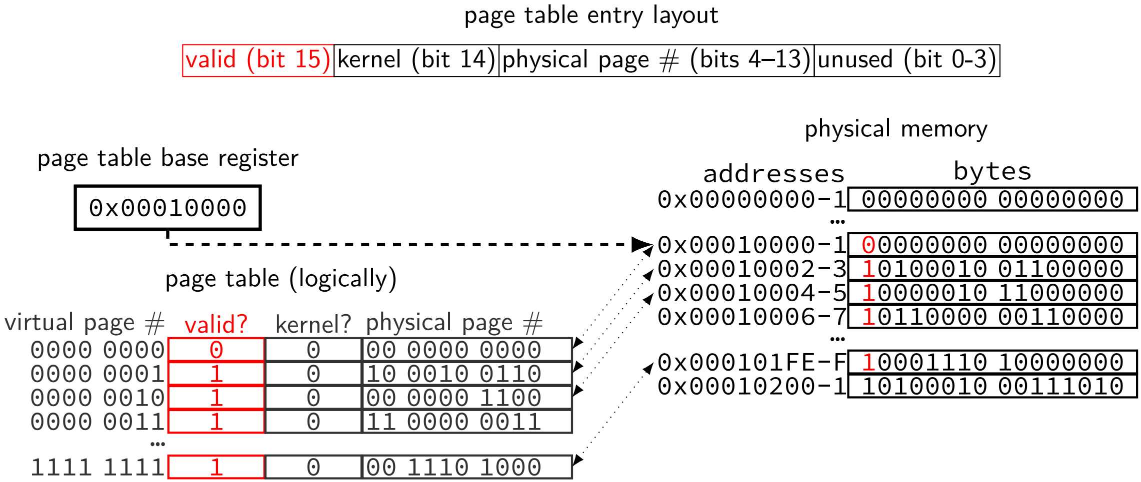
valid (bit 15)	kernel (bit 14)	physical page # (bits 4–13)	unused (bit 0-3)



page tables in memory



page tables in memory



page tables in memory

page table entry layout

valid (bit 15)	kernel (bit 14)	physical page # (bits 4–13)	unused (bit 0-3)

page table base register

0x00010000

page table (logically)

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

physical memory

addresses	bytes
0x00000000-1	00000000 00000000
…	
0x00010000-1	00000000 00000000
0x00010002-3	10100010 01100000
0x00010004-5	10000010 11000000
0x00010006-7	10110000 00110000
…	
0x000101FE-F	10001110 10000000
0x00010200-1	10100010 00111010

page tables in memory

page table entry layout

valid (bit 15)	kernel (bit 14)	physical page # (bits 4–13)	unused (bit 0-3)

page table base register

0x00010000

page table (logically)

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

physical memory

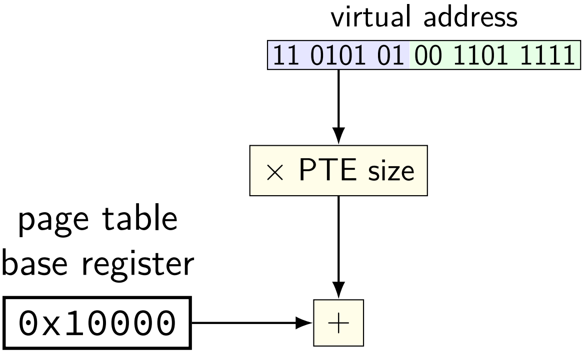
addresses	bytes
0x00000000-1	00000000 00000000
…	
0x00010000-1	00000000 00000000
0x00010002-3	10100010 01100000
0x00010004-5	10000010 11000000
0x00010006-7	10110000 00110000
…	
0x000101FE-F	10001110 10000000
0x00010200-1	10100010 00111010

memory access with page table

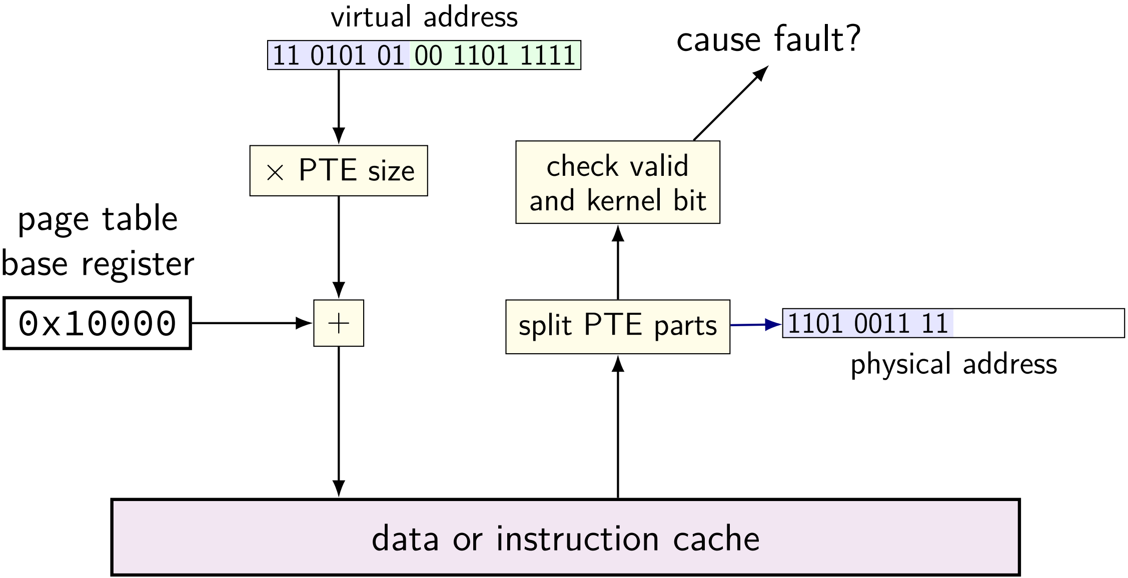
virtual address

11 0101 01	00 1101 1111

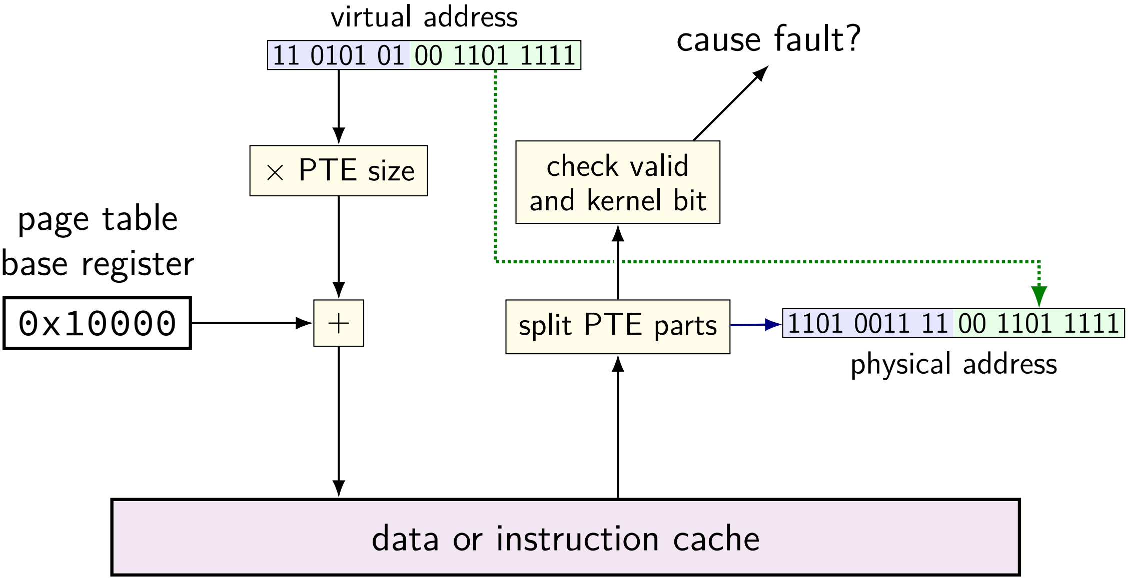
memory access with page table



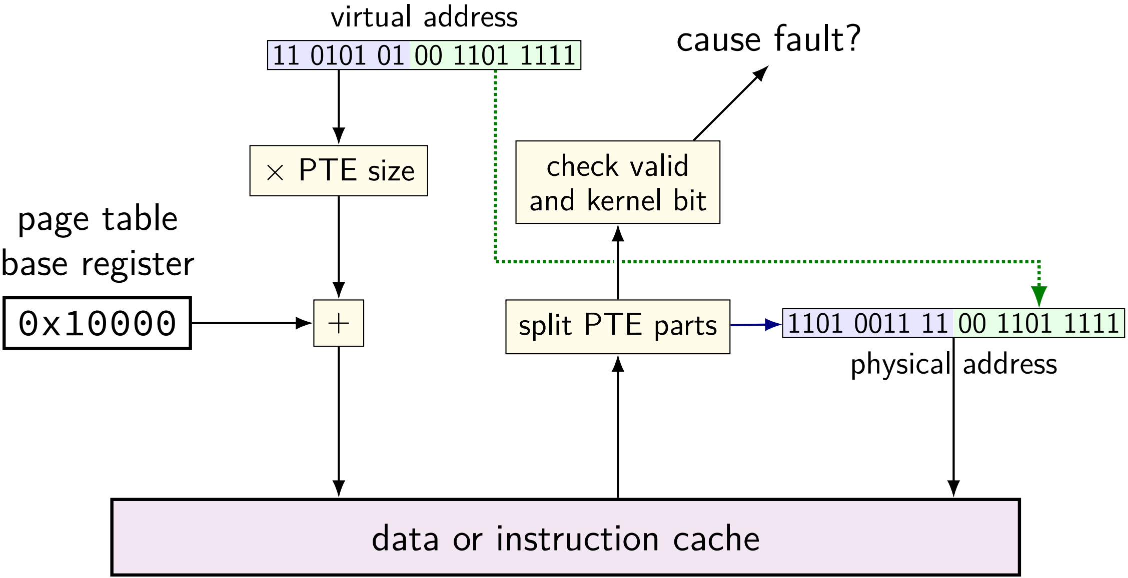
memory access with page table



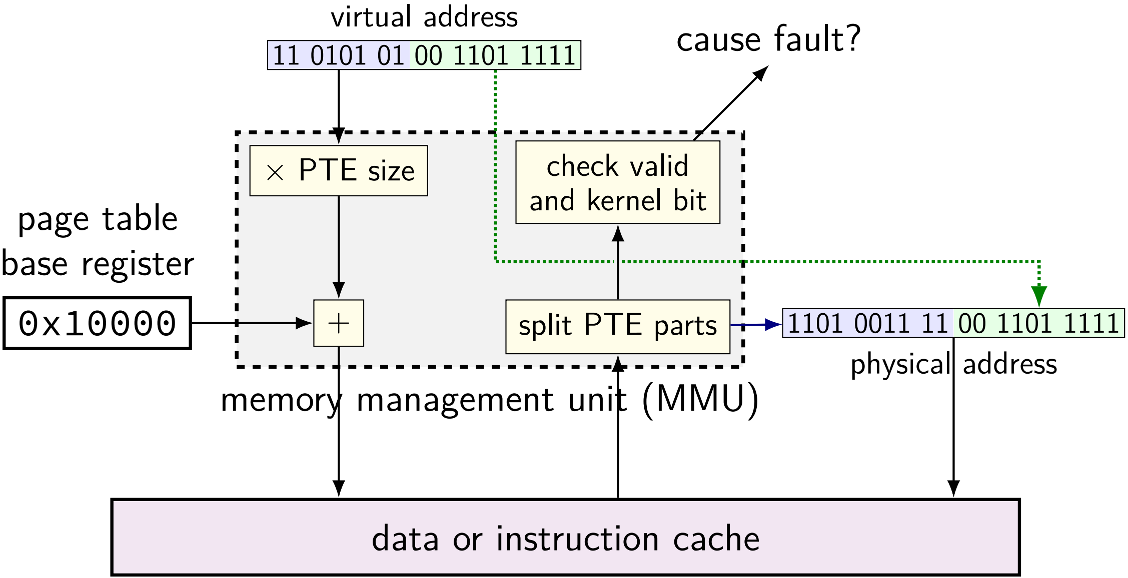
memory access with page table



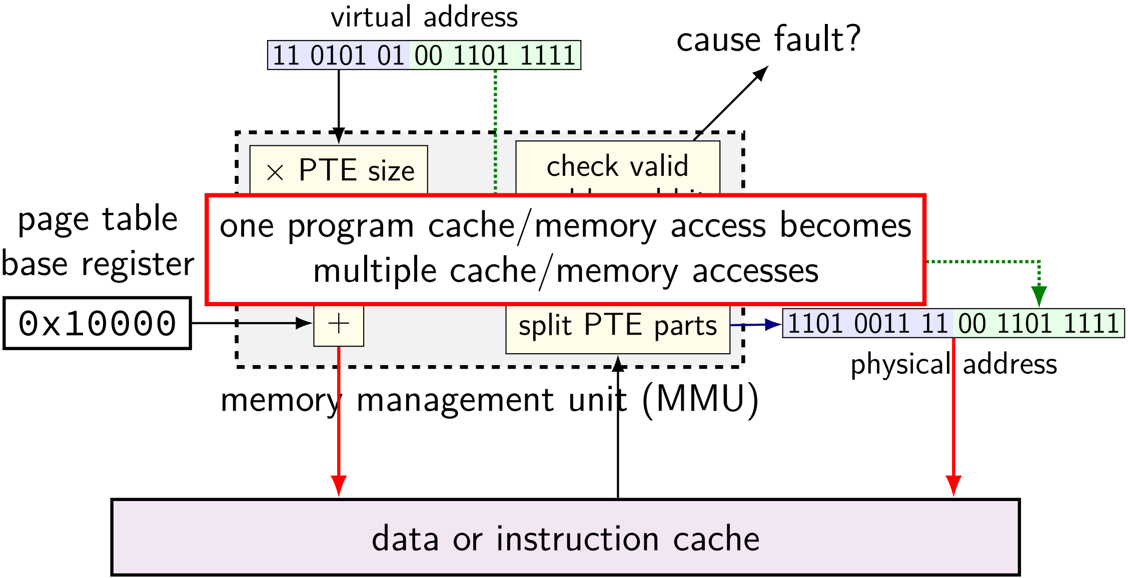
memory access with page table



memory access with page table

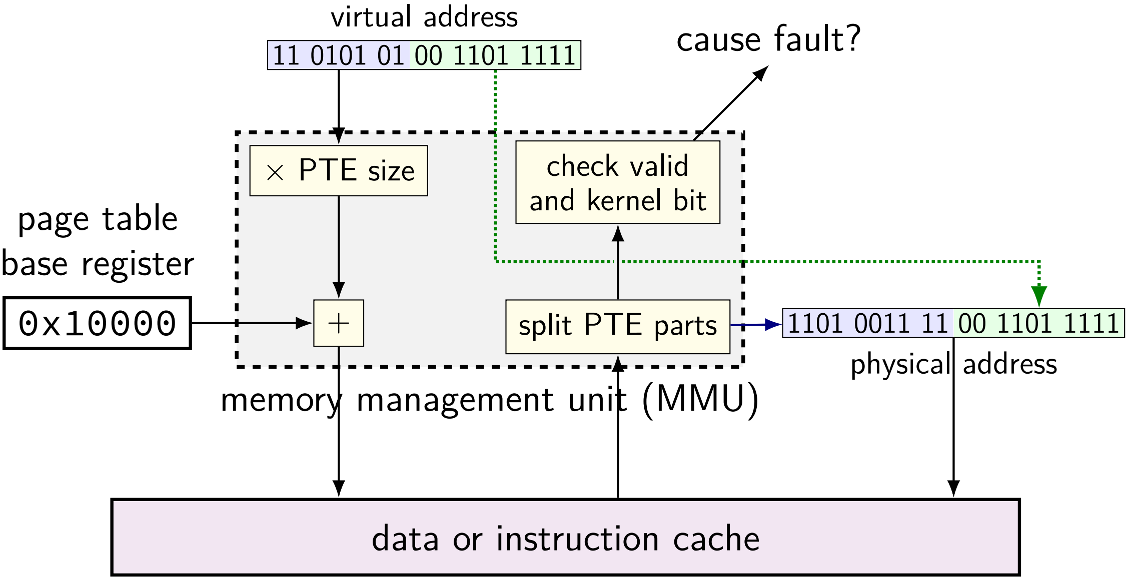


memory access with page table

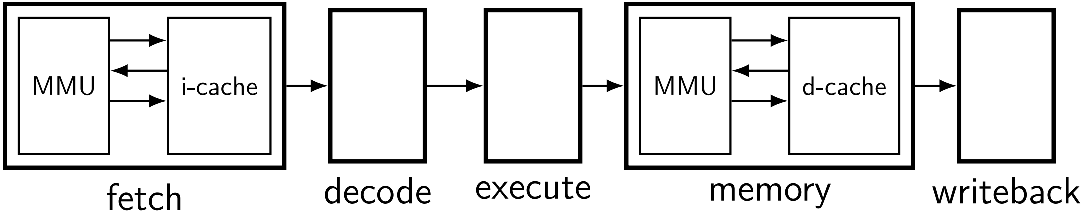


one program cache/memory access becomes multiple cache/memory accesses

memory access with page table



MMUs in the pipeline



up to four memory accesses per instruction

cache for page-table entries to make fast

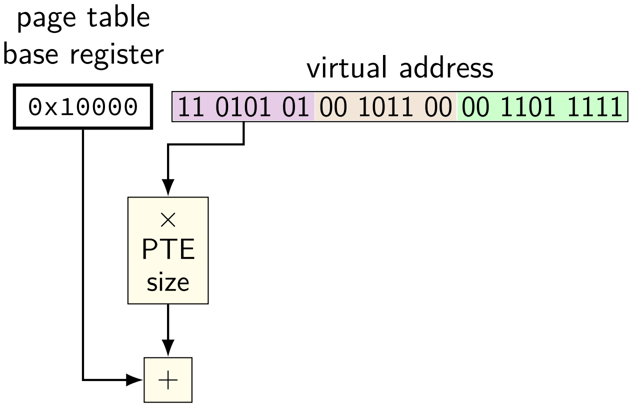
two-level page table lookup

virtual address

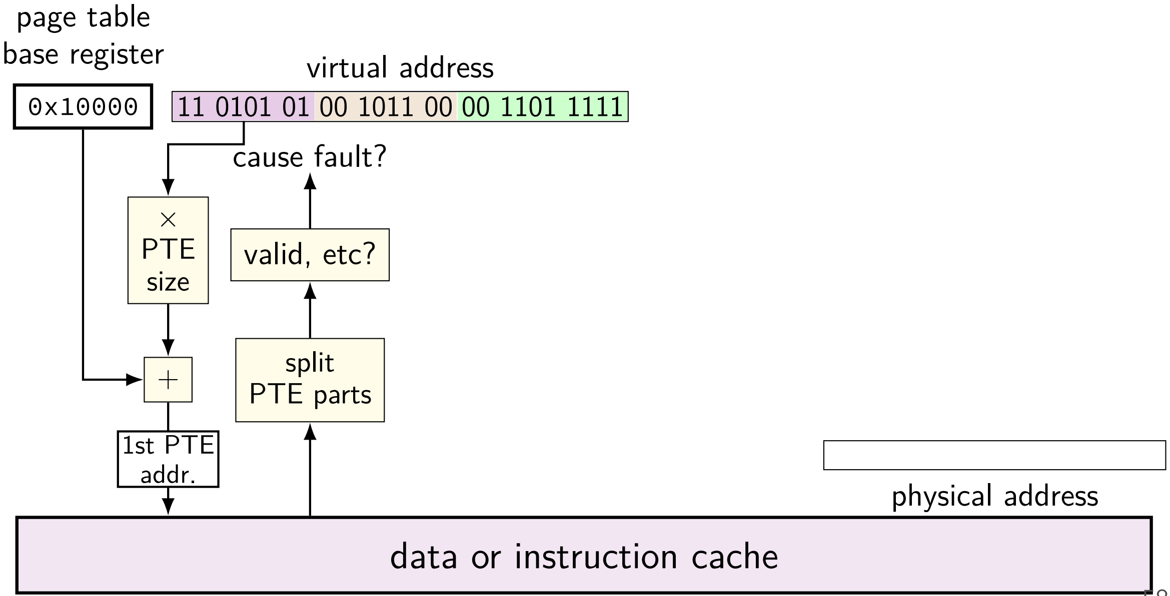
11 0101 01	00 1011 00	00 1101 1111

VPN — split into two parts (one per level)

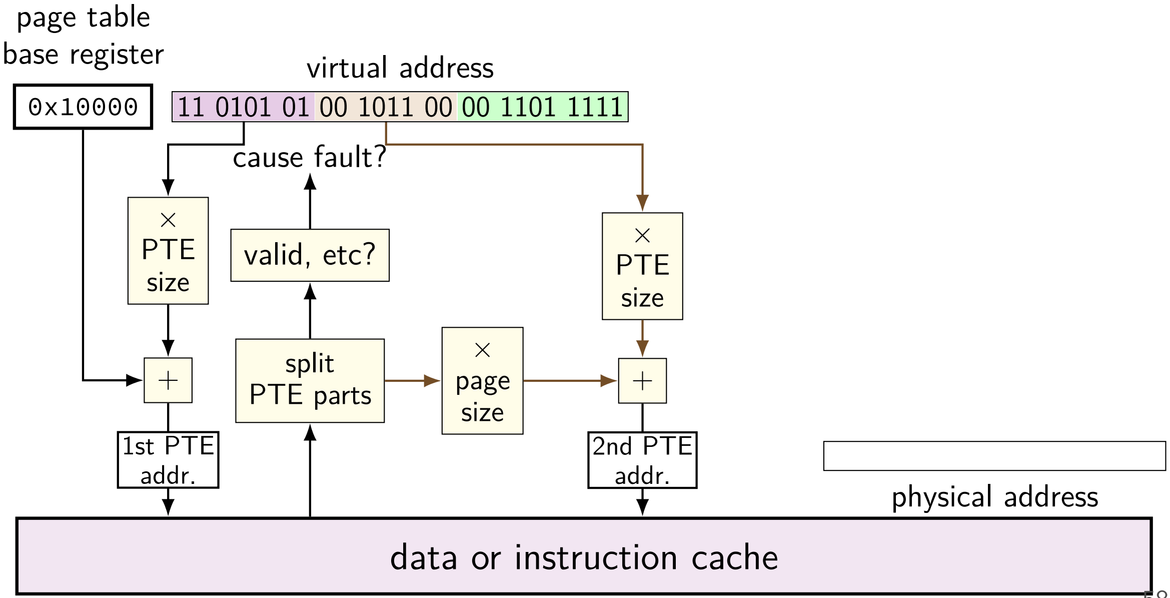
two-level page table lookup



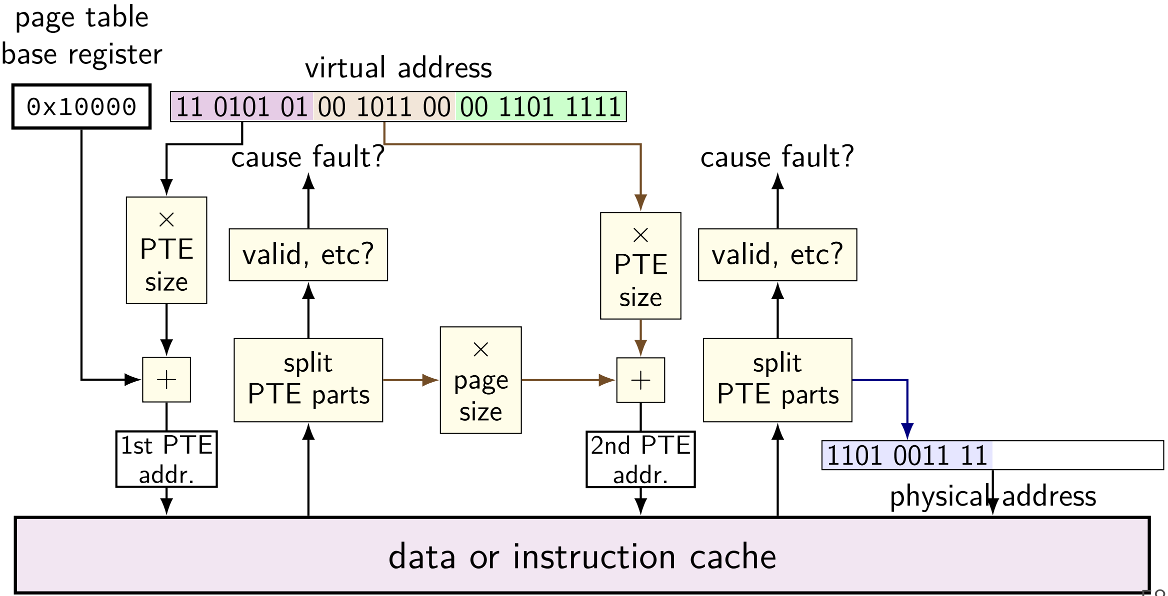
two-level page table lookup



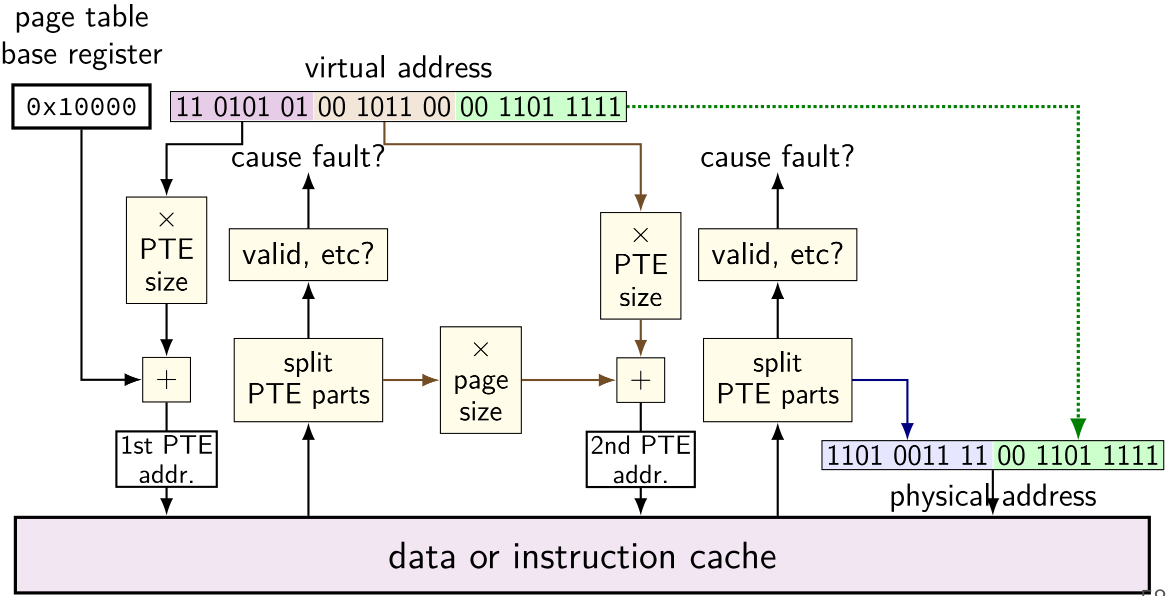
two-level page table lookup



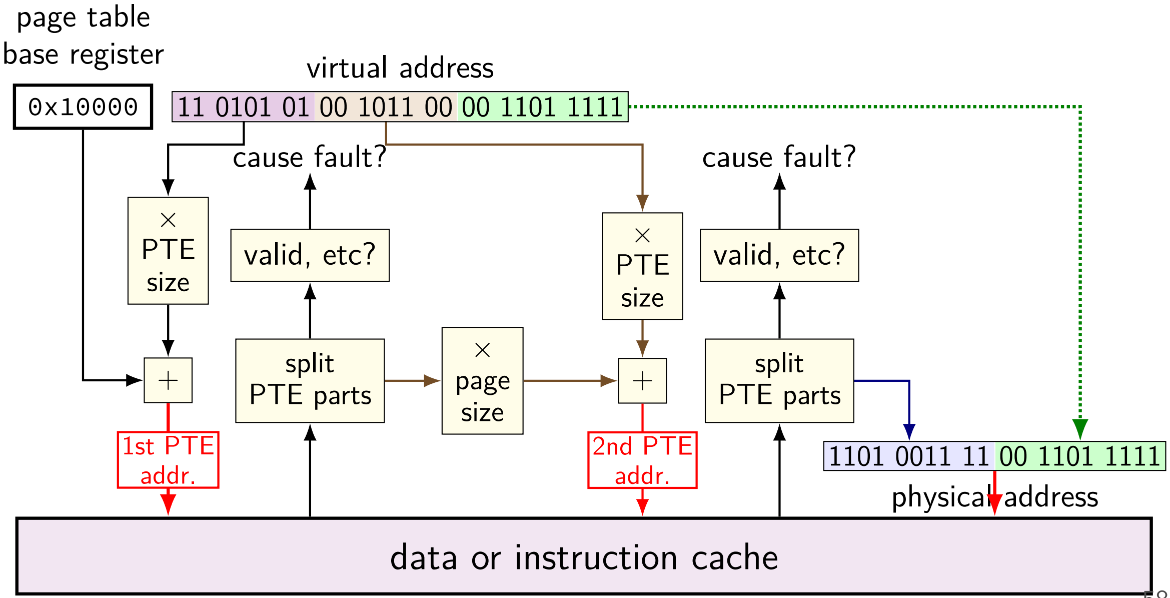
two-level page table lookup



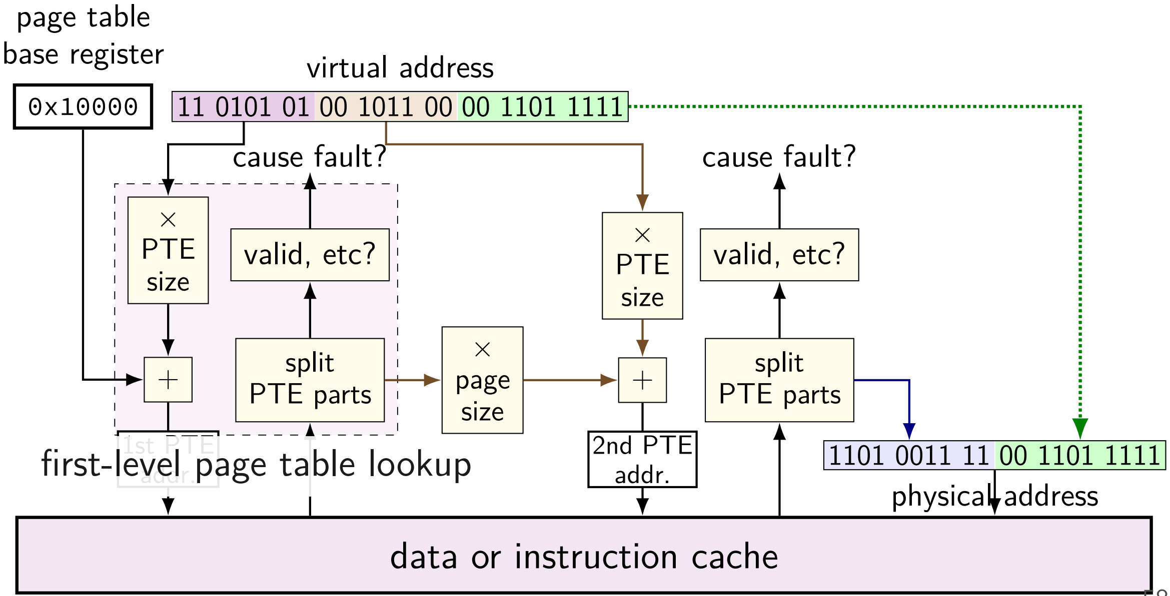
two-level page table lookup



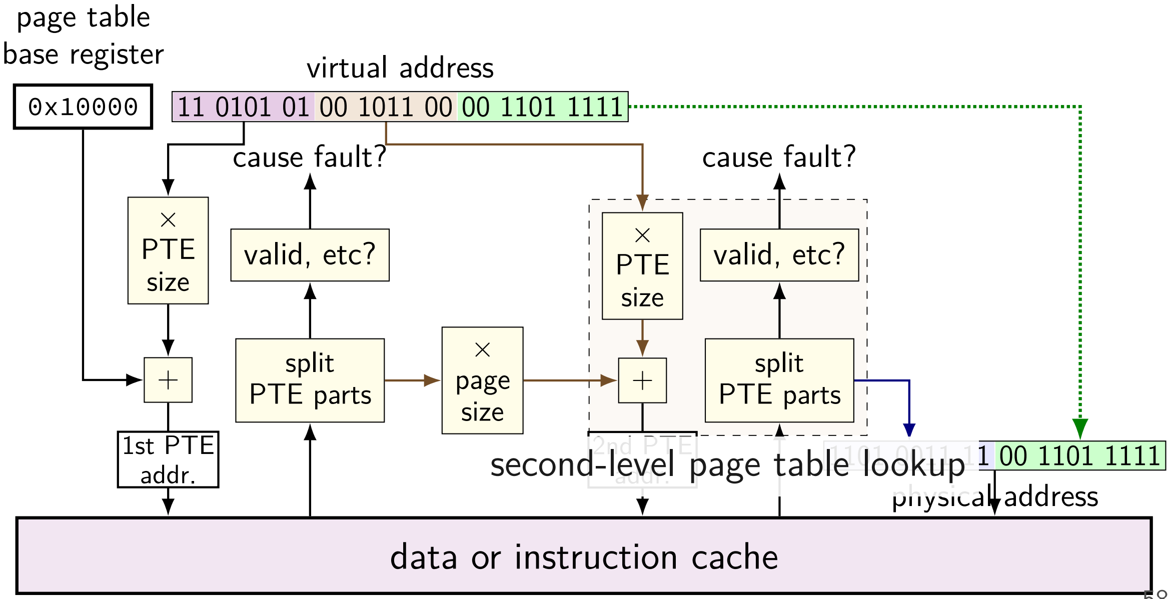
two-level page table lookup



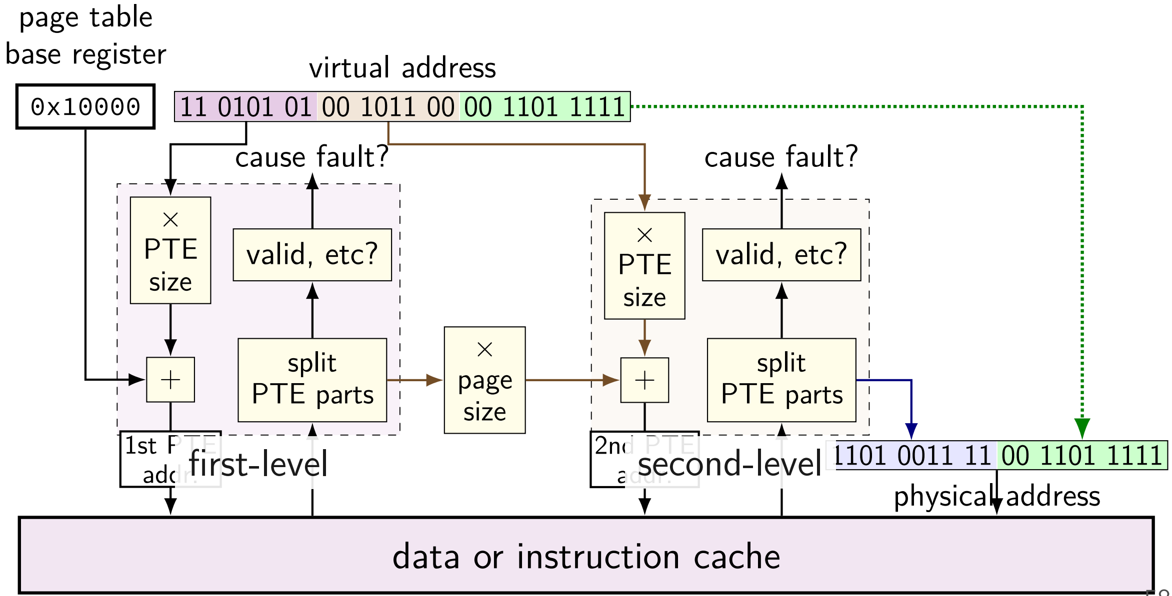
two-level page table lookup



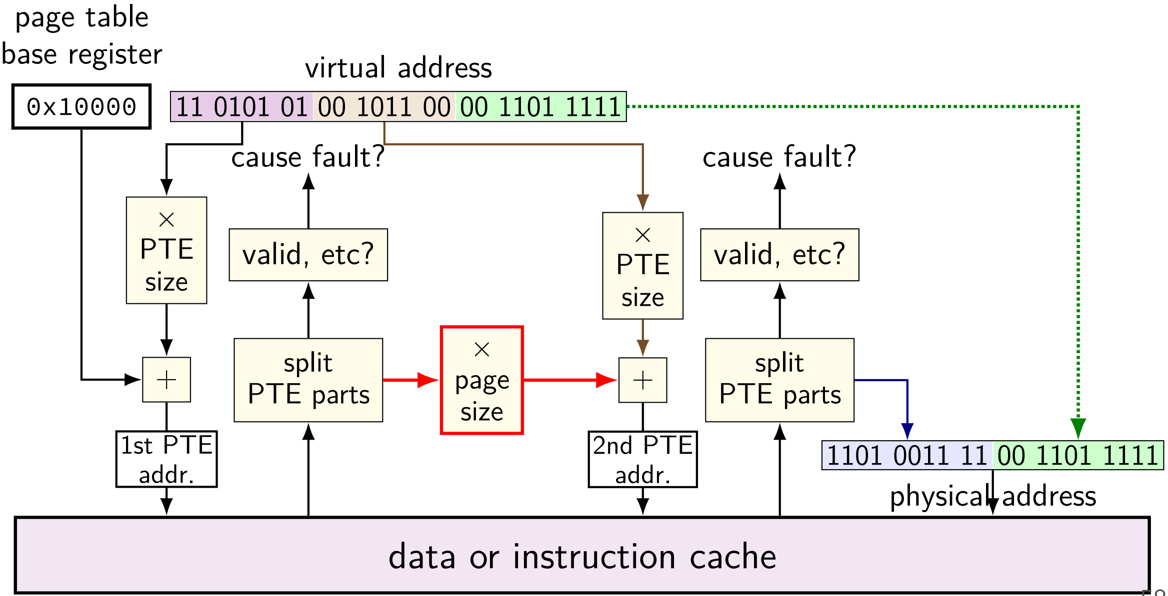
two-level page table lookup



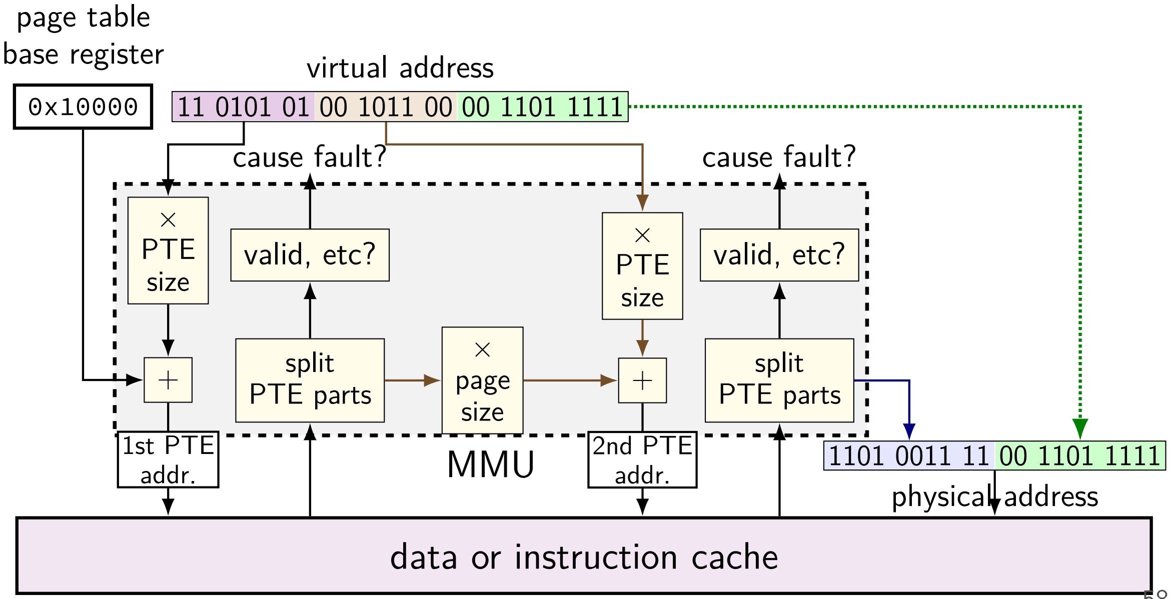
two-level page table lookup



two-level page table lookup



two-level page table lookup



xv6 kernel space mapings

```
// This table defines the kernel's mappings, which are present in
// every process's page table.
static struct kmap {
  void *virt;
  uint phys_start;
  uint phys_end;
  int perm;
} kmap[] = {
 // I/O space
 { (void*)KERNBASE, 0,             EXTMEM,    PTE_W},

 // kern text+rodata
 { (void*)KERNLINK, V2P(KERNLINK), V2P(data), 0},

 // kern data+memory
 { (void*)data,     V2P(data),     PHYSTOP,   PTE_W},

 // more devices
 { (void*)DEVSPACE, DEVSPACE,      0,         PTE_W},
```