beyond threads / virtual memory 1

last time

reader/writer-priority with rwlocks

- choosing priority by tracking who's waiting

deadlock definition/conditions

- mutual exclusion
- hold and wait
- no preemption
- circular dependency

fixing deadlock

detecting deadlocks

- construct graph, look for cycles if (finitely) divisible resources
- something else if divisible resources

quiz note

I made mistake in the code on the semaphore output question

$x > 1$ should have been $x > 0$

accepting results as written (none correct)
or with correction above

on office hours waits

I know they're not great

only so much I can do given availability/number of TAs

maybe some options with queues, but...

- makes it harder to get TAs to group students with same Q together
- ...which should be happening more to make OH go quicker

but note: not all office hours used evenly

some times (especially earlier in day, or mine??) less crowded

last time

reader/writer-priority with rwlocks

- choosing priority by tracking who's waiting

deadlock definition/conditions

- mutual exclusion
- hold and wait
- no preemption
- circular dependency

fixing deadlock

detecting deadlocks

- construct graph, look for cycles if (finitely) divisible resources
- something else if divisible resources

aside: divisible resources

deadlock is possible with divislbe resources like memory,...

example: suppose 6MB of RAM for threads total:

- thread 1 has 2MB allocated, waiting for 2MB
- thread 2 has 2MB allocated, waiting for 2MB
- thread 3 has 1MB allocated, waiting for keypress

cycle: thread 1 waiting on memory owned by thread 2?

not a deadlock — thread 3 can still finish

- and after it does, thread 1 or 2 can finish

aside: divisible resources

deadlock is possible with divislbe resources like memory,...

example: suppose 6MB of RAM for threads total:

- thread 1 has 2MB allocated, waiting for 2MB
- thread 2 has 2MB allocated, waiting for 2MB
- thread 3 has 1MB allocated, waiting for keypress

cycle: thread 1 waiting on memory owned by thread 2?

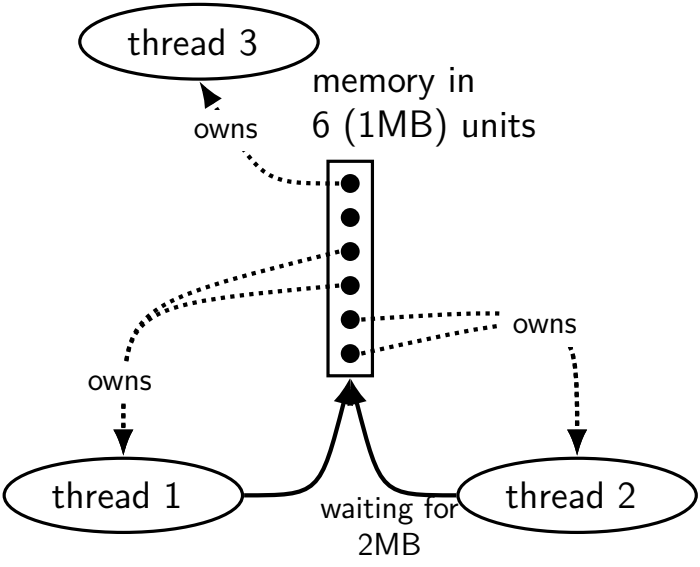
not a deadlock — thread 3 can still finish

- and after it does, thread 1 or 2 can finish

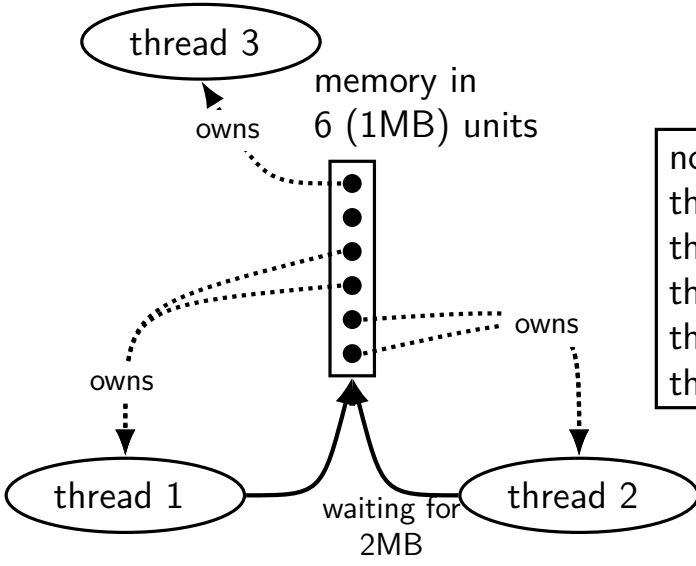
...but would be deadlock

- ...if thread 3 waiting lock held by thread 1
- ...with 5MB of RAM

divisible resources: not deadlock

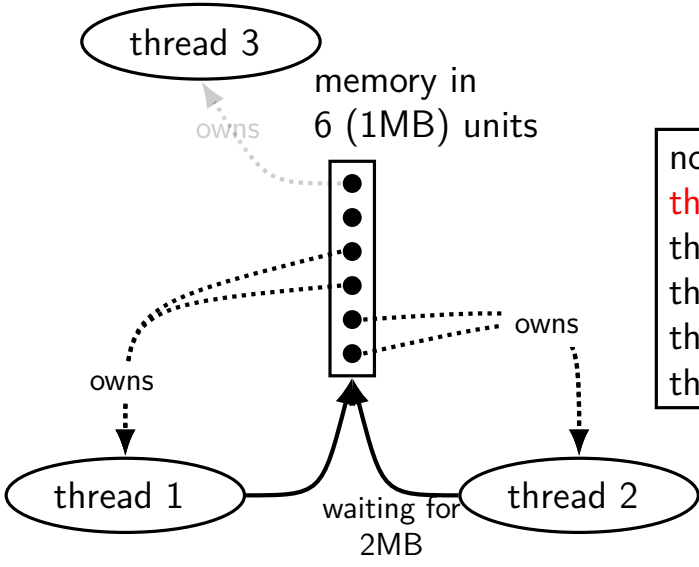


divisible resources: not deadlock



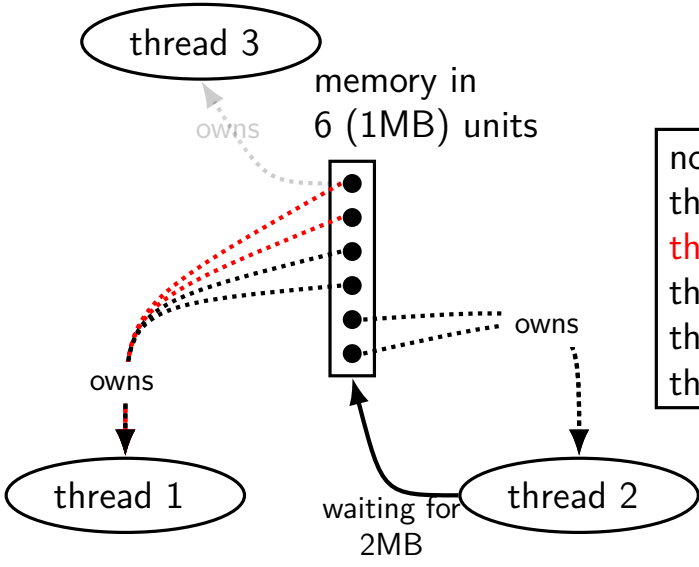
not deadlock:
thread 3 finishes
then thread 1 can get memory
then thread 1 finishes
then thread 2 can get resources
then thread 2 can finish

divisible resources: not deadlock



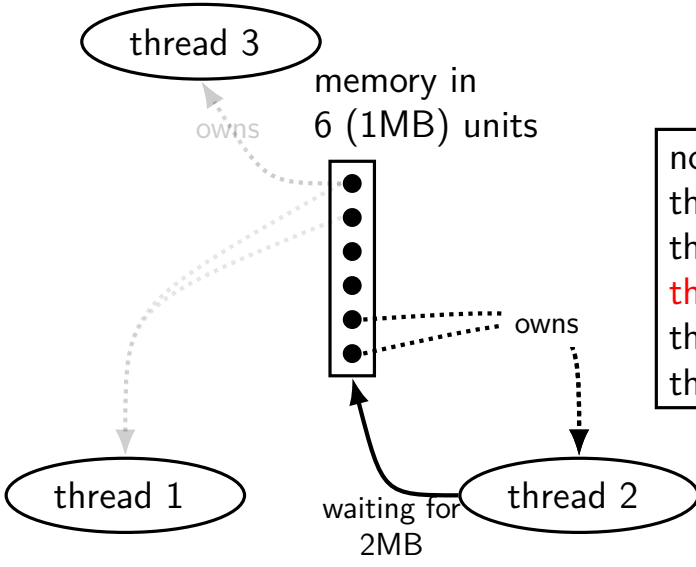
not deadlock:
thread 3 finishes
then thread 1 can get memory
then thread 1 finishes
then thread 2 can get resources
then thread 2 can finish

divisible resources: not deadlock



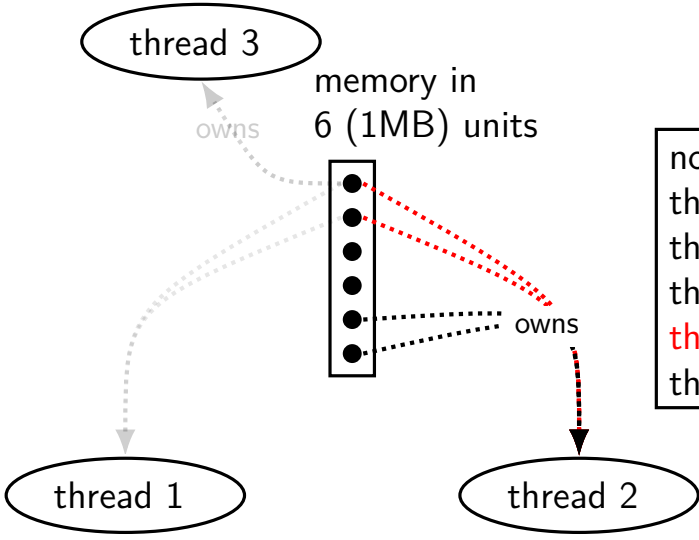
not deadlock:
thread 3 finishes
then thread 1 can get memory
then thread 1 finishes
then thread 2 can get resources
then thread 2 can finish

divisible resources: not deadlock



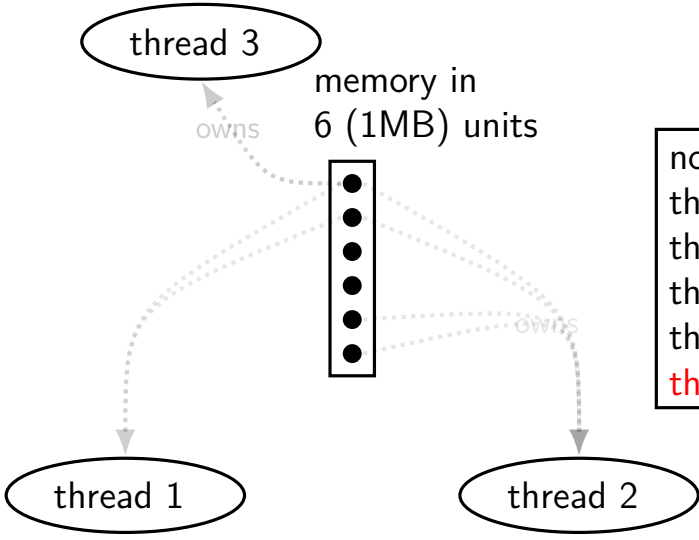
not deadlock:
thread 3 finishes
then thread 1 can get memory
then thread 1 finishes
then thread 2 can get resources
then thread 2 can finish

divisible resources: not deadlock



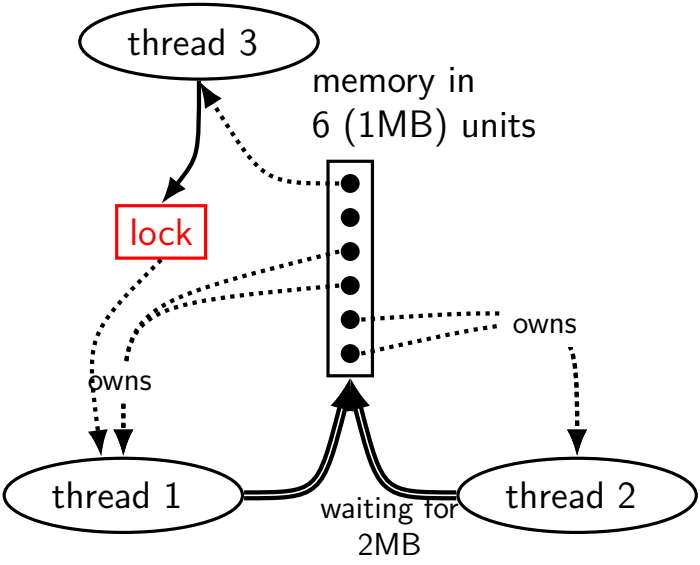
not deadlock:
thread 3 finishes
then thread 1 can get memory
then thread 1 finishes
then thread 2 can get resources
then thread 2 can finish

divisible resources: not deadlock

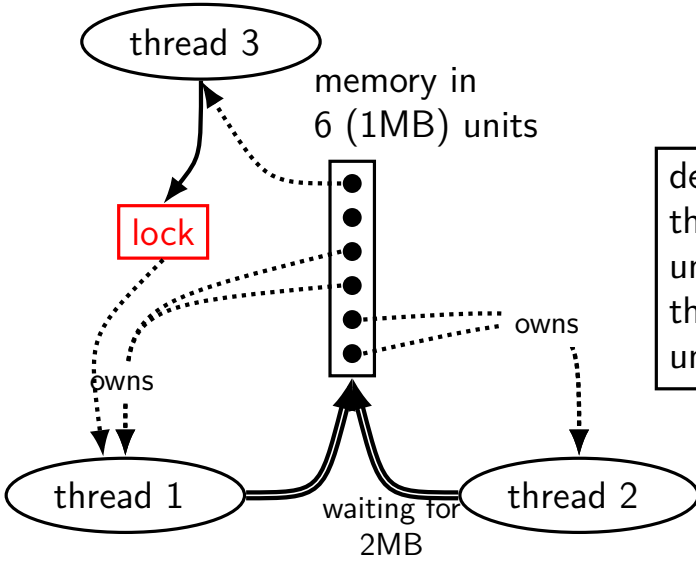


not deadlock:
thread 3 finishes
then thread 1 can get memory
then thread 1 finishes
then thread 2 can get resources
then thread 2 can finish

divisible resources: is deadlock

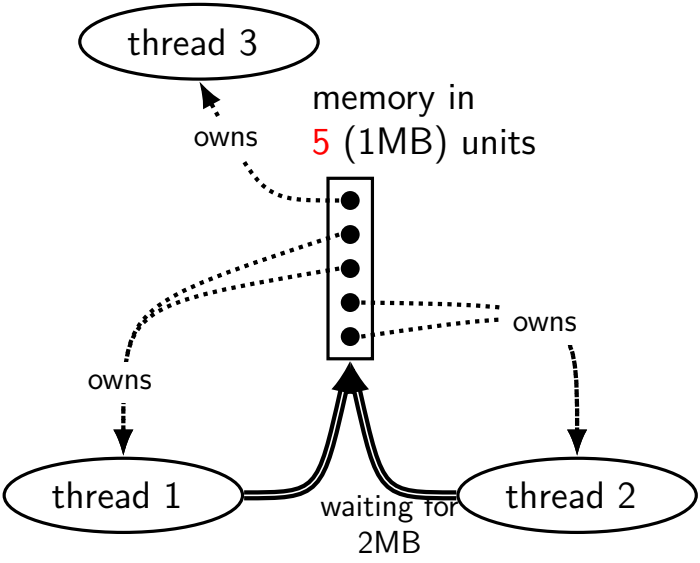


divisible resources: is deadlock

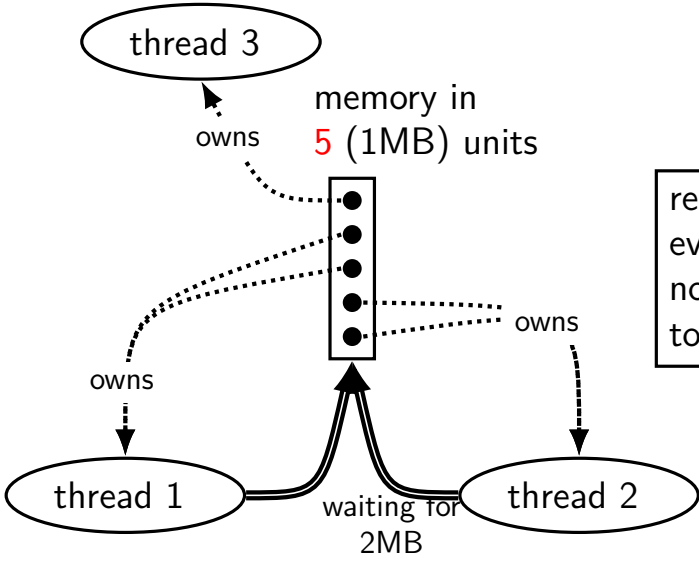


deadlock:
thread 3 can't finish
until thread 1 releases lock, but
thread 1 can't finish
until thread 3 releases memory

divisible resources: is deadlock

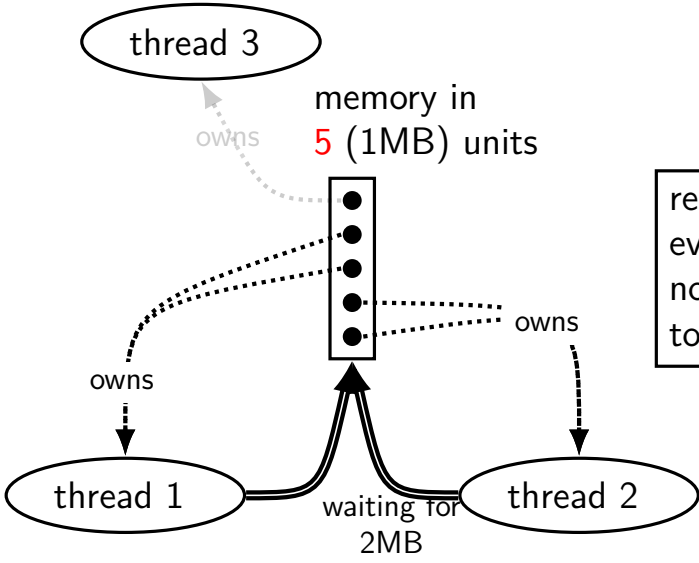


divisible resources: is deadlock



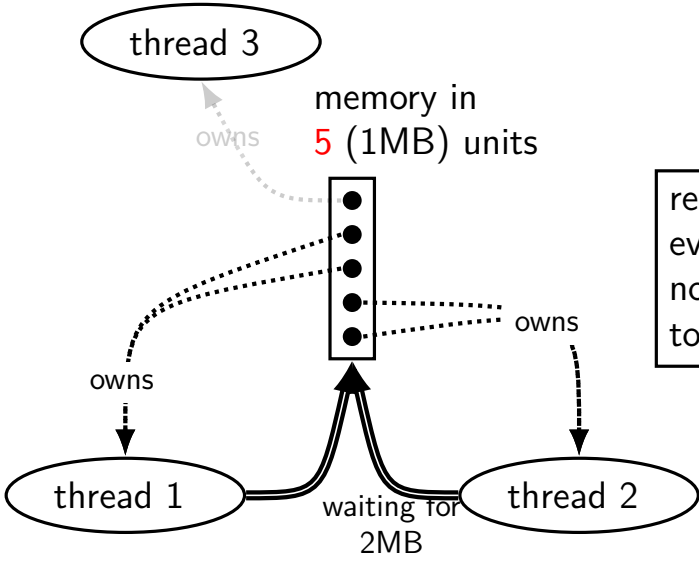
reducing memory: deadlock:
even after thread 3 finishes
no way for thread 1+2
to get what they want

divisible resources: is deadlock



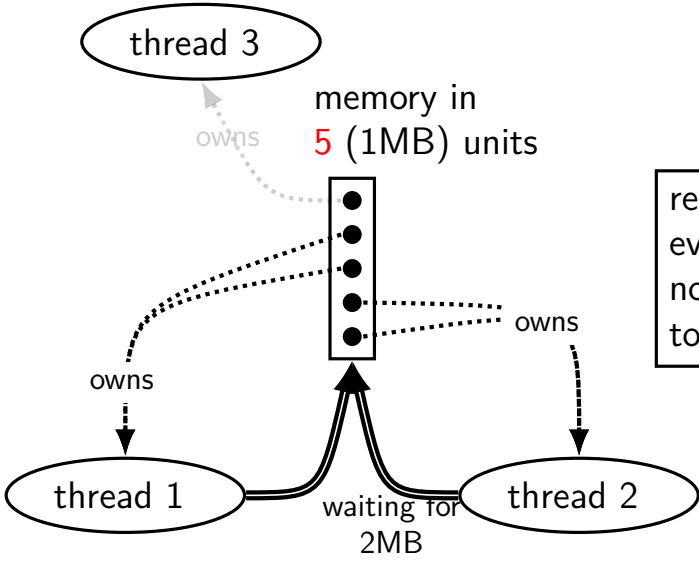
reducing memory: deadlock:
even after thread 3 finishes
no way for thread 1+2
to get what they want

divisible resources: is deadlock



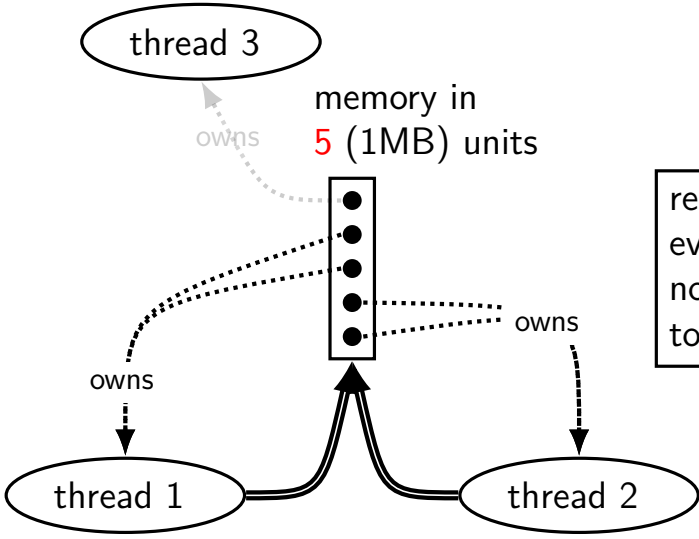
reducing memory: deadlock:
even after thread 3 finishes
no way for thread 1+2
to get what they want

divisible resources: is deadlock



reducing memory: deadlock:
even after thread 3 finishes
no way for thread 1+2
to get what they want

divisible resources: is deadlock



reducing memory: deadlock:
even after thread 3 finishes
no way for thread 1+2
to get what they want

deadlock detection with divisibe resources

can't rely on cycles in graphs in this case

alternate algorithm exists

- similar technique to how we showed no deadlock

high-level intuition: simulate what could happen

- find threads that could finish based on resources available now

full details: look up Baker's algorithm

aside: deadlock detection in reality

instrument all contended resources?

- add tracking of who locked what
- modify every lock implementation — no simple spinlocks?
- some tricky cases: e.g. what about counting semaphores?

doing something useful on deadlock?

- want way to "undo" partially done operations

...but done for some applications

common example: for locks in a database

- database typically has customized locking code
- "undo" exists as side-effect of code for handling power/disk failures

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') {
        StopWaitingToRead(c->fd);
        if (IsExitCommand(command)) { CleanupConnection(c); return;
        computeResponse(c->response, c->command);
        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') {
        StopWaitingToRead(c->fd);
        if (IsExitCommand(command)) { CleanupConnection(c); return;
        computeResponse(c->response, c->command);
        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') {
        StopWaitingToRead(c->fd);
        if (IsExitCommand(command)) { CleanupConnection(c); return;
        computeResponse(c->response, c->command);
        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') {
        StopWaitingToRead(c->fd);
        if (IsExitCommand(command)) { CleanupConnection(c); return;
        computeResponse(c->response, c->command);
        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') {
    StopWaitingToRead(c->fd);
    if (IsExitCommand(command)) { CleanupConnection(c); return; }
    computeResponse(c->response, c->command);
    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') {
    StopWaitingToRead(c->fd);
    if (IsExitCommand(command)) { CleanupConnection(c); return; }
    computeResponse(c->response, c->command);
    StartWaitingToWrite(c->fd);
  }
}
```

```
...
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; }
computeResponse(response, commmand);
... // write 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') {
    StopWaitingToRead(c->fd);
    if (IsExitCommand(command)) { CleanupConnection(c); return; }
    computeResponse(c->response, c->command);
    StartWaitingToWrite(c->fd);
  }
}
```

```
...
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; }
computeResponse(response, commmand);
... // write 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') {
    StopWaitingToRead(c->fd);
    if (IsExitCommand(command)) { CleanupConnection(c); return; }
    computeResponse(c->response, c->command);
    StartWaitingToWrite(c->fd);
  }
}
```

```
...
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; }
computeResponse(response, commmand);
... // write 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') {
    StopWaitingToRead(c->fd);
    if (IsExitCommand(command)) { CleanupConnection(c); return; }
    computeResponse(c->response, c->command);
    StartWaitingToWrite(c->fd);
  }
}
```

```
...
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; }
computeResponse(response, commmand);
... // write response
```

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

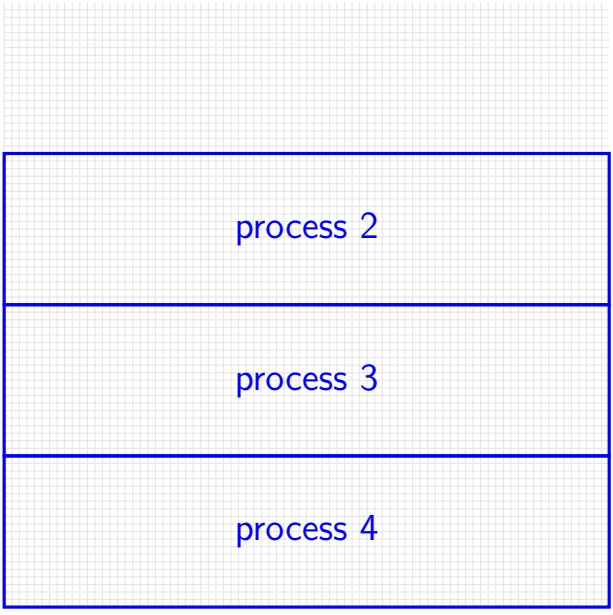
message passing API

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

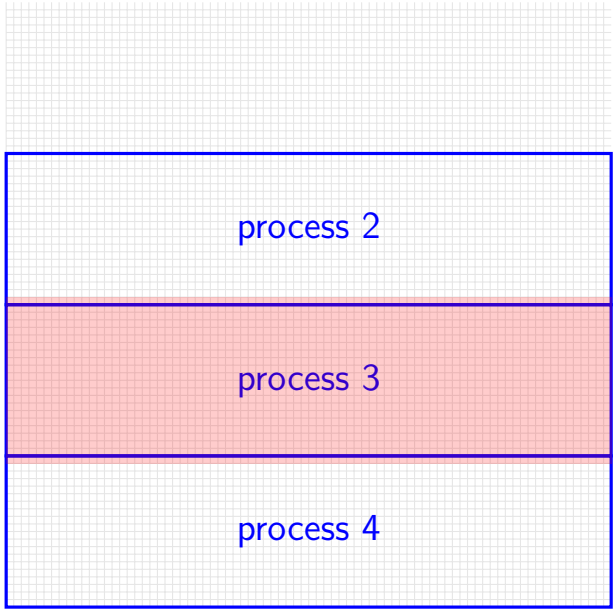


divide grid
like you would for normal threads

each process stores cells
in that part of grid

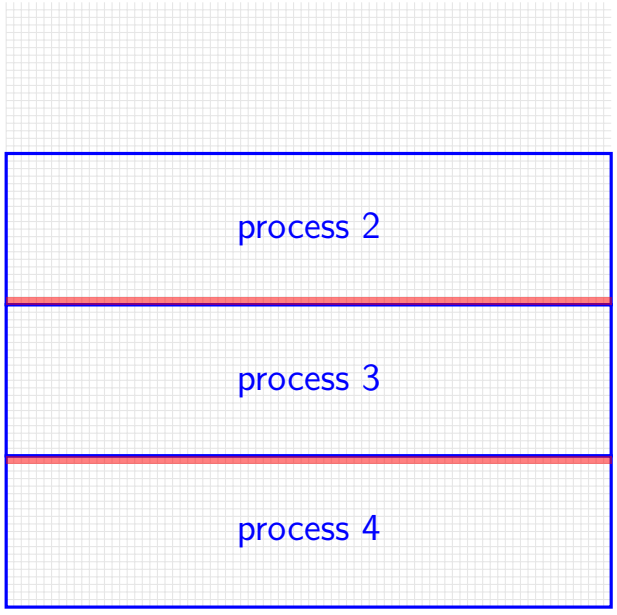
(no shared memory!)

message passing game of life



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

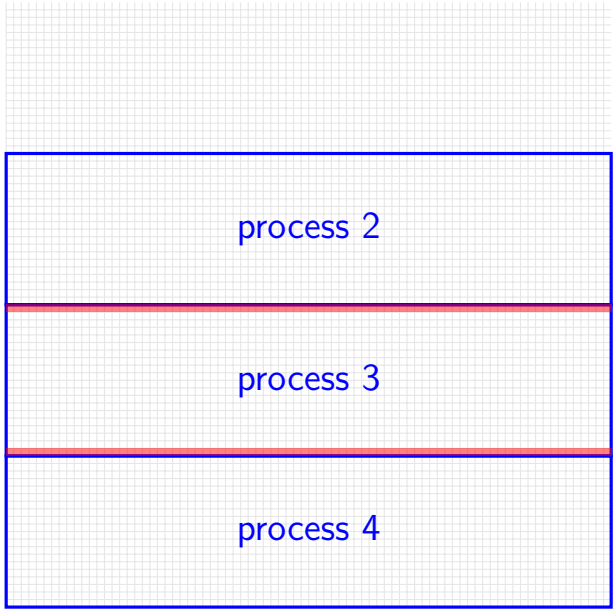
message passing game of life



small slivers of
other process's cells needed

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

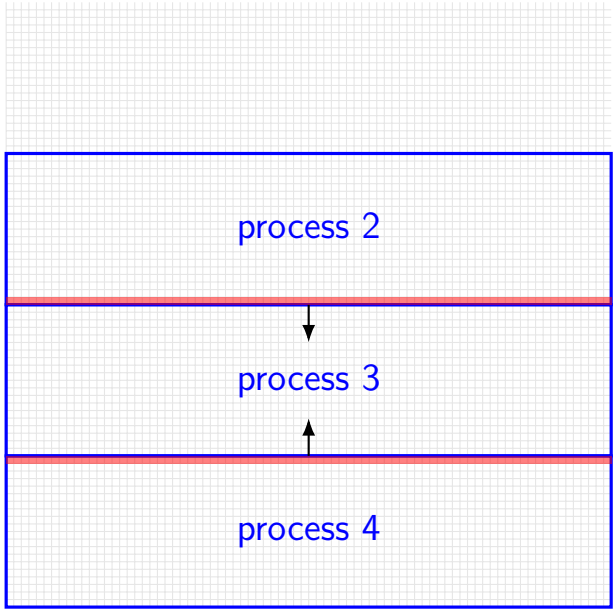
message passing game of life



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

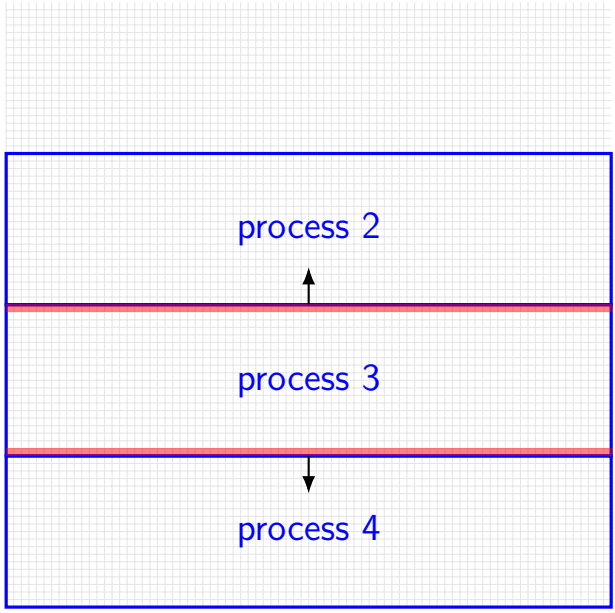
so process 3 also sends messages

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

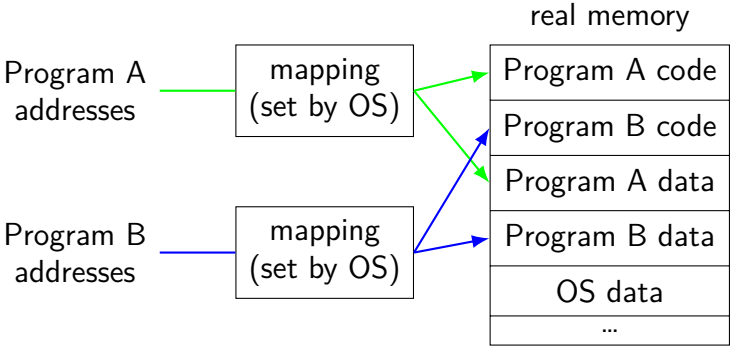
for CpEs: the prereq for this class is ECE's *embedded* class

(and *not the CpE architecture class*)

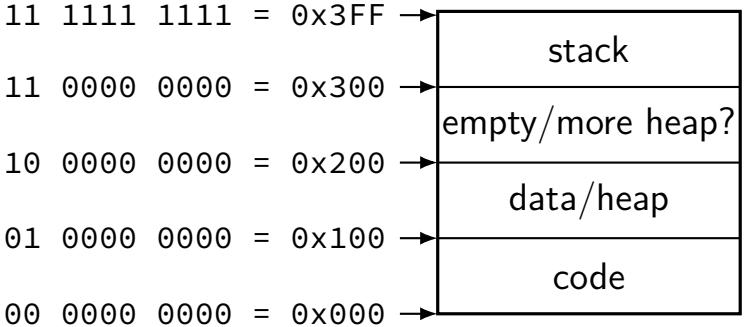
I think little virtual memory coverage in CpE embedded *or* architecture?

- don't have precise information about that

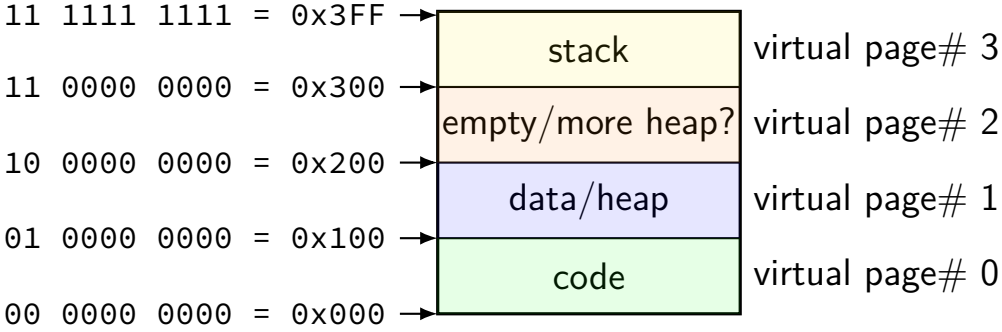
address translation



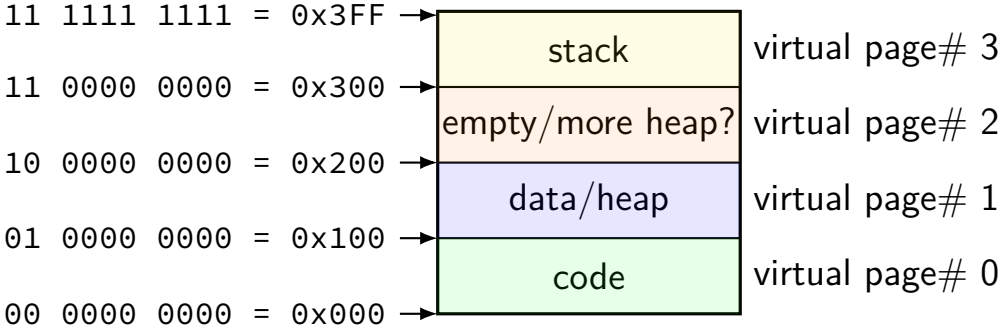
toy program memory



toy program memory

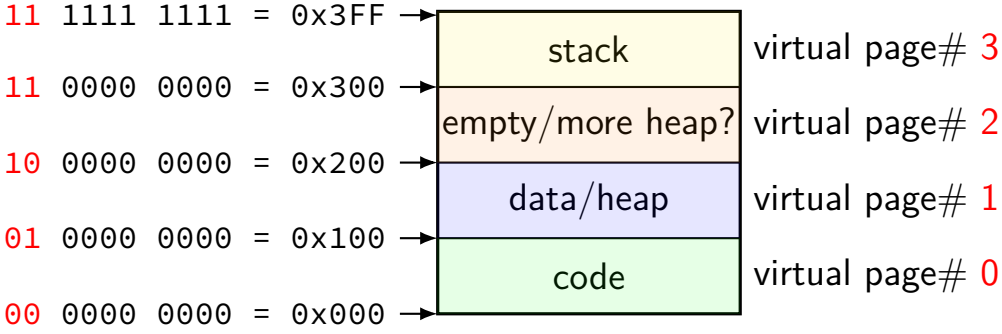


toy program memory



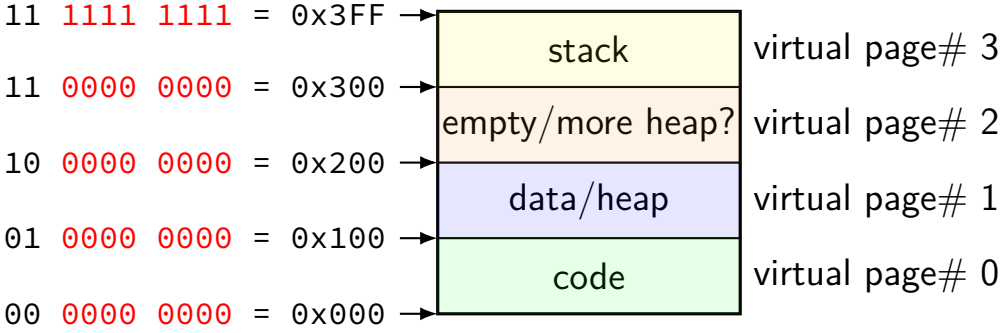
divide memory into pages (2^8 bytes in this case)
"virtual" = addresses the program sees

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

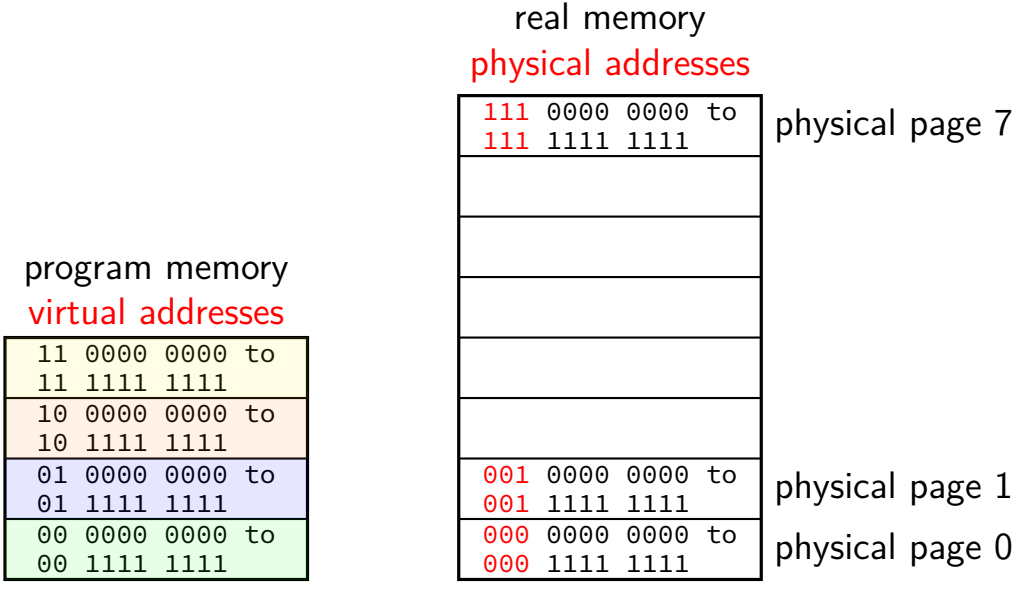
program memory
virtual addresses

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

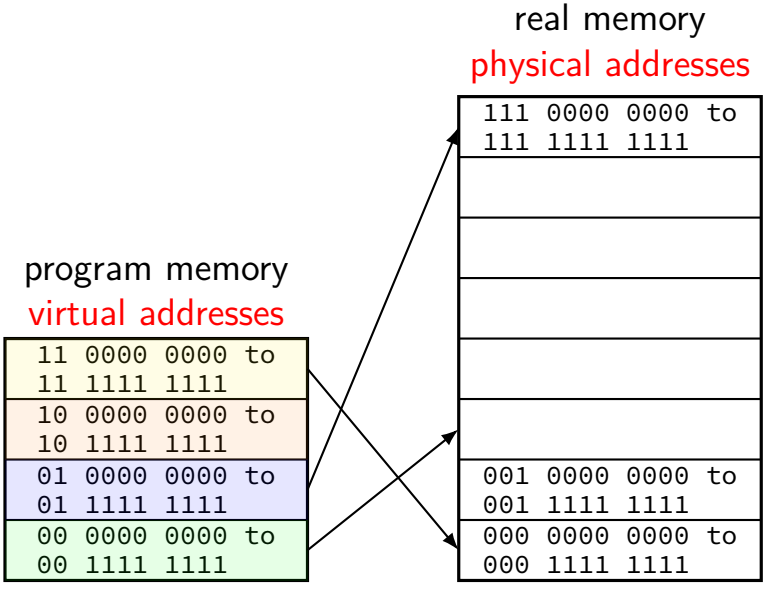
real memory
physical addresses

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

toy physical memory

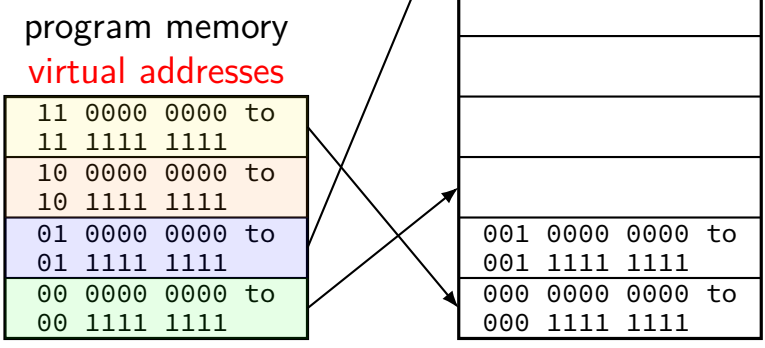


toy physical memory



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

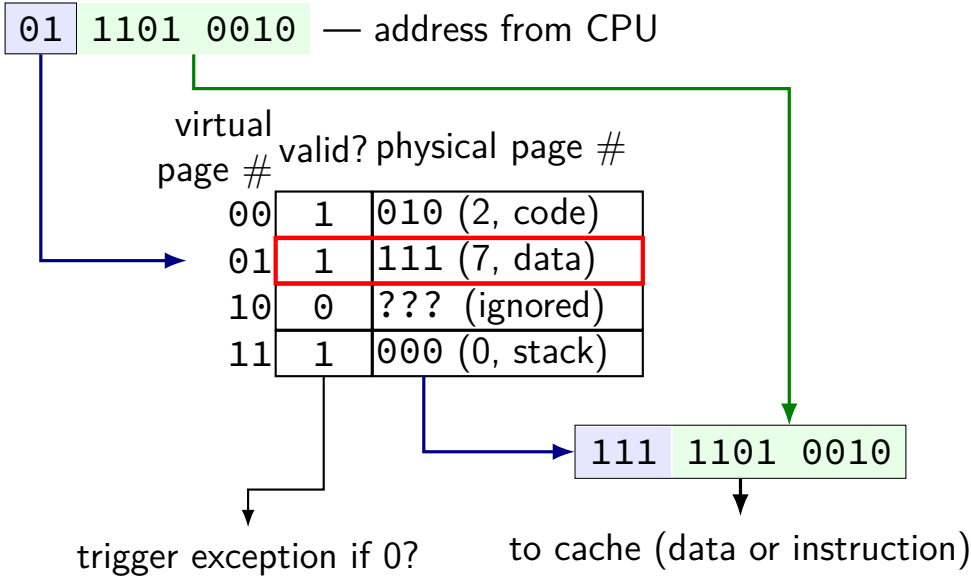
real memory
physical addresses

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

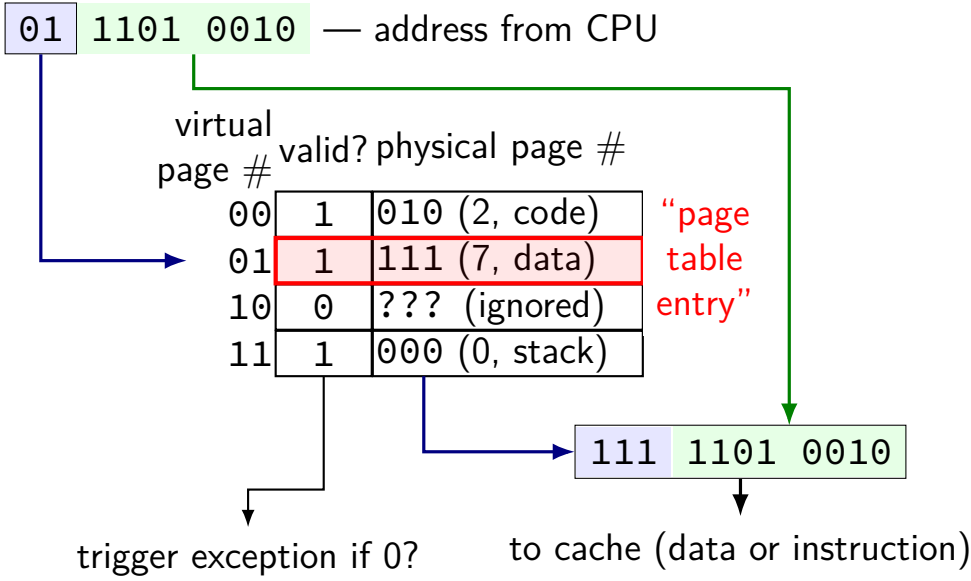
toy page table lookup

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

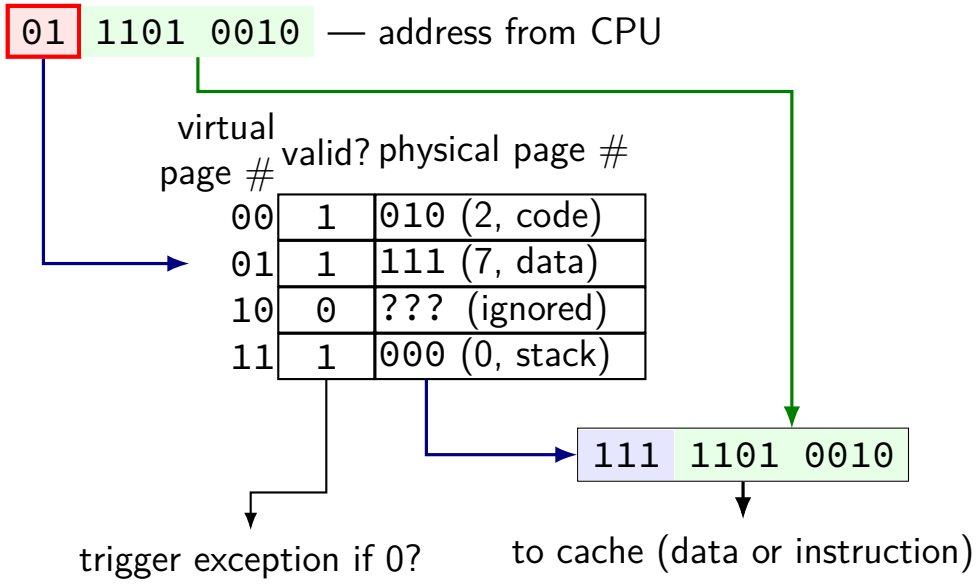
toy page table lookup



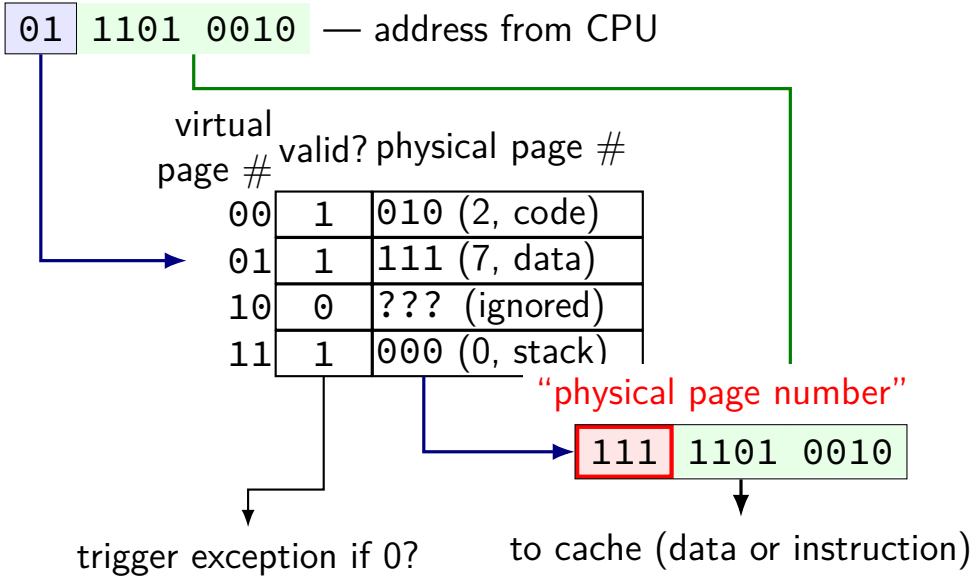
toy page table lookup



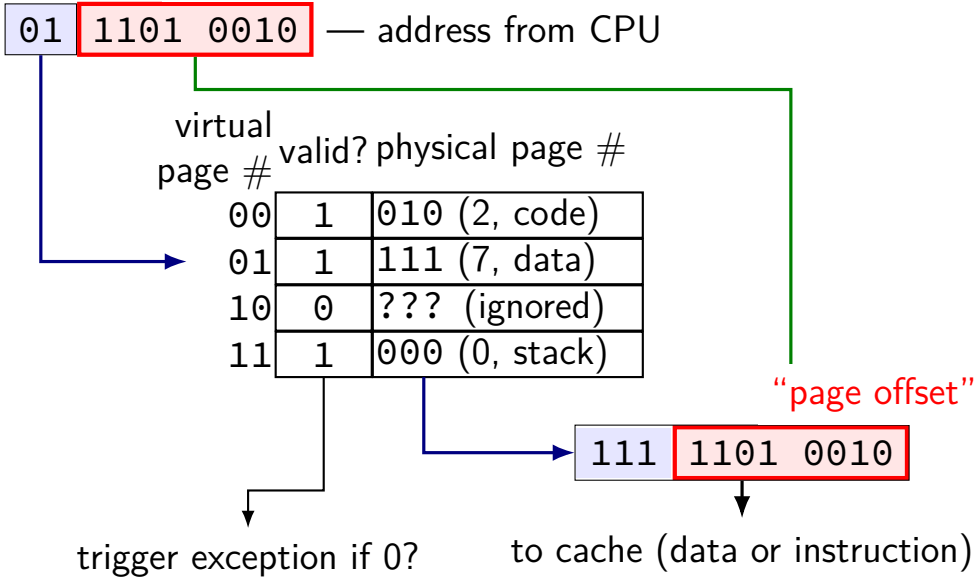
"virtual page number" lookup



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 = _____

x86-32: VPN and PO (solution)

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

given virtual address `0xABCD0123`

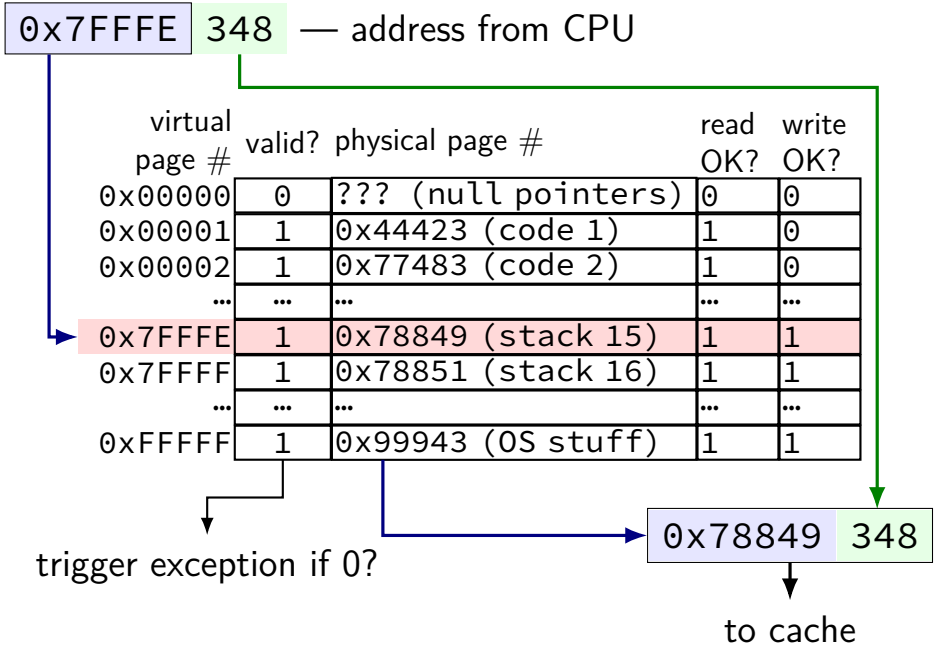
virtual page number = `0xABCD0`

page offset = `0x123`

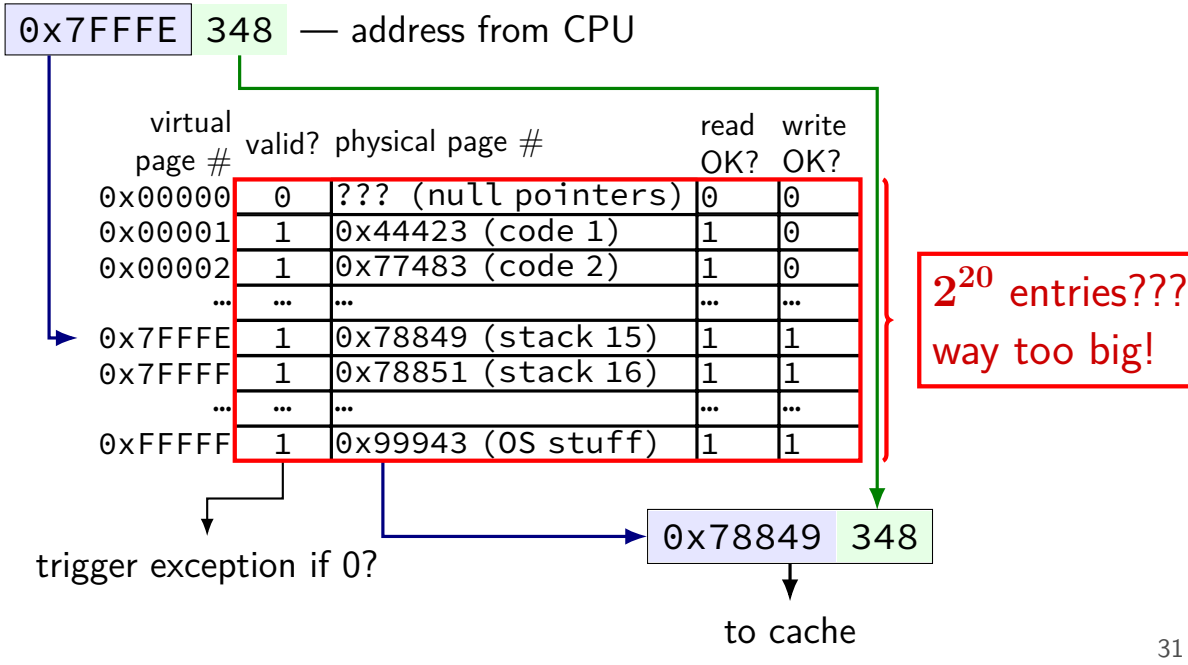
if that virtual page maps to physical page `0x998`

physical address = `0x998123`

32-bit x86 flat page table???



32-bit x86 flat page table???



storing huge page table?

keep it in memory

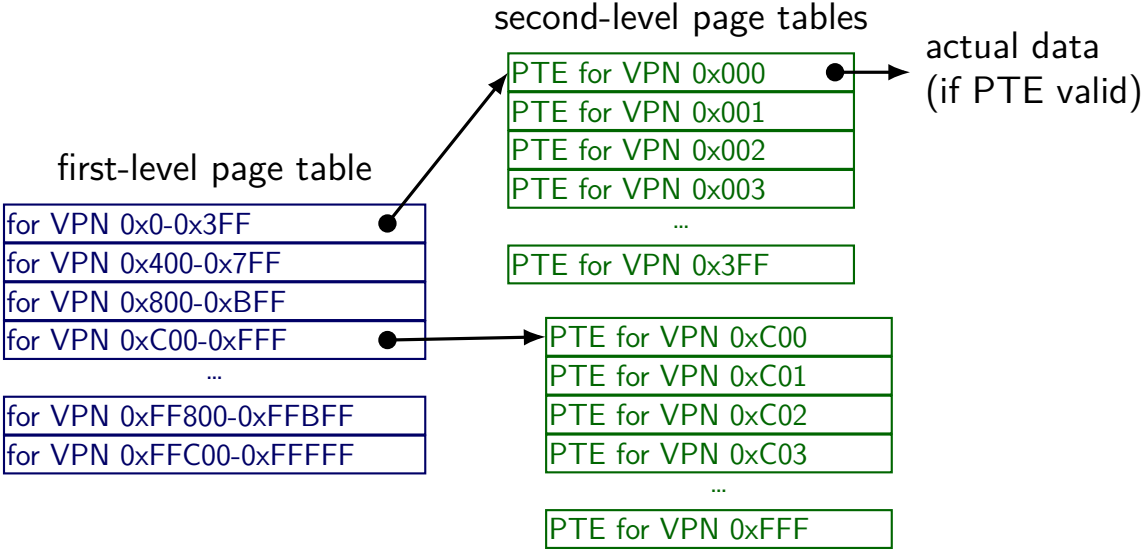
- add special cache for page table entries to handle memory being slow
- special cached called translation lookaside buffer (TLB)

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

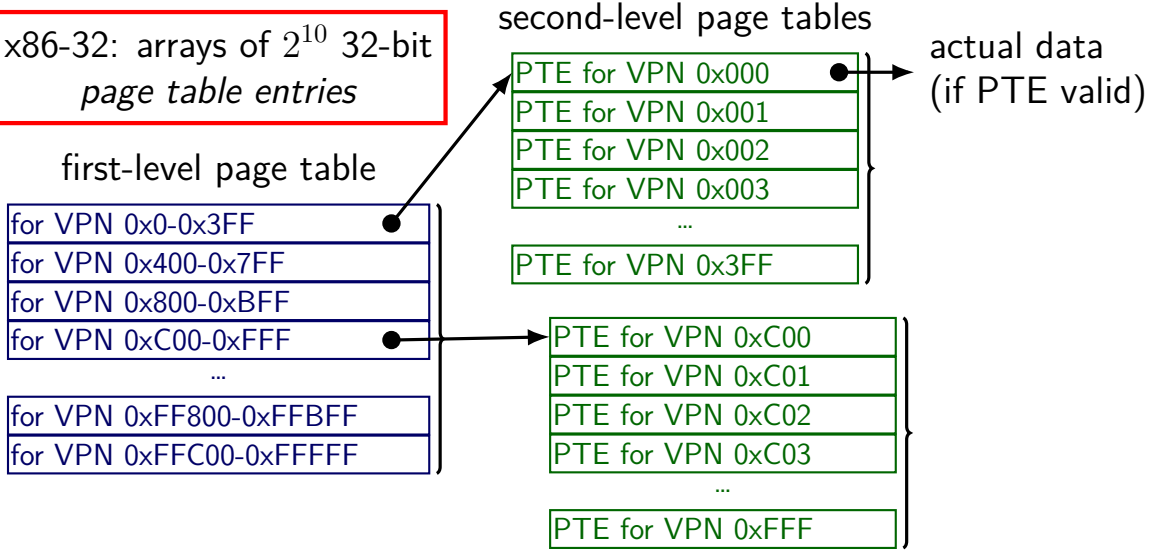
two-level page table; 2^{20} pages total; 2^{10} entries per table



two-level page tables

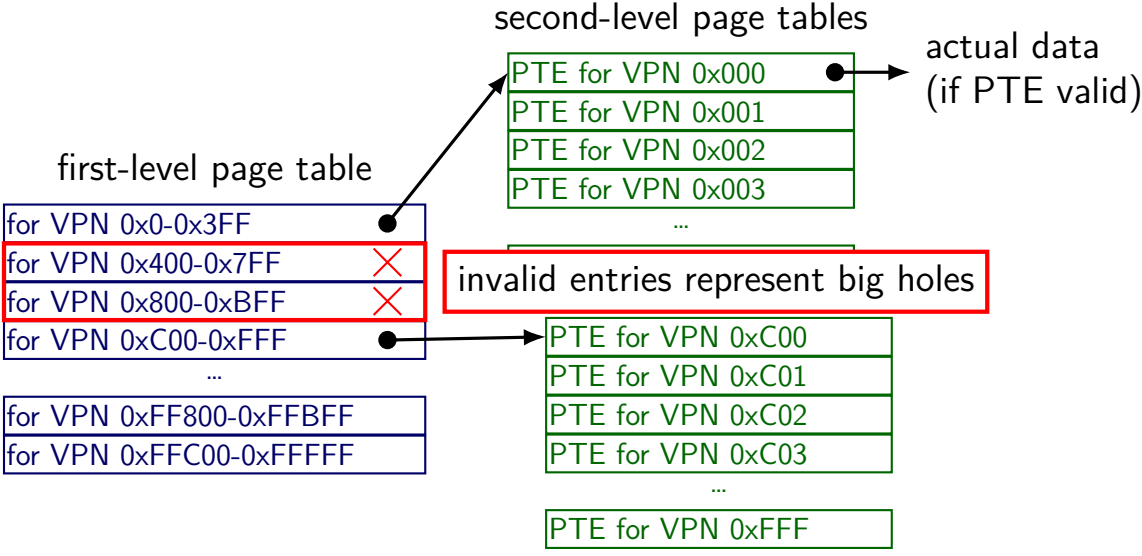
two-level page table; 2^{20} pages total; 2^{10} entries per table

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

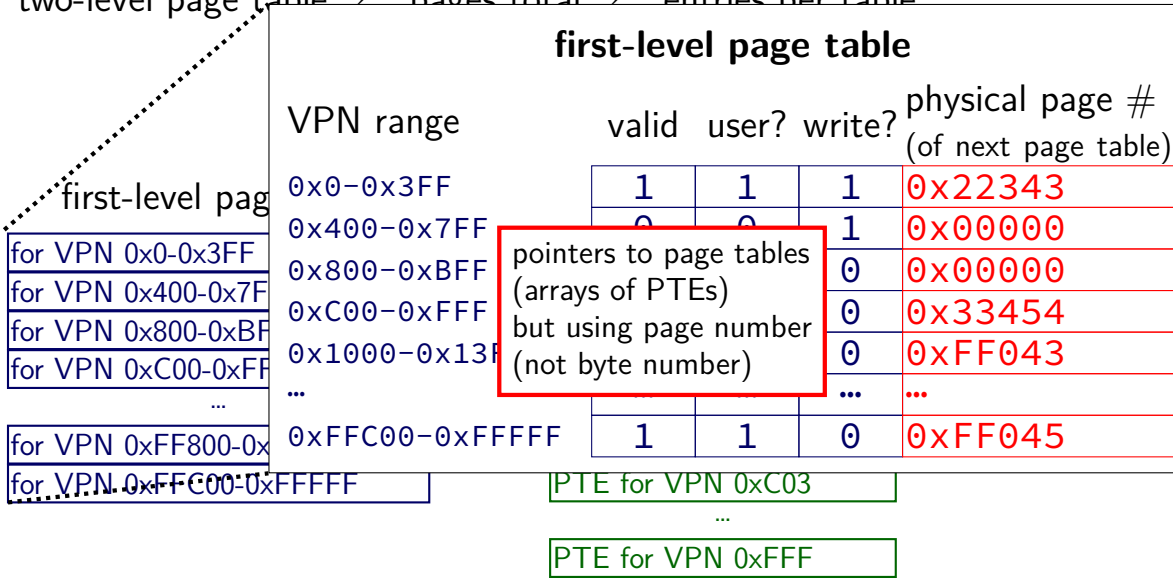
PTE for VPN 0xC03

...

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

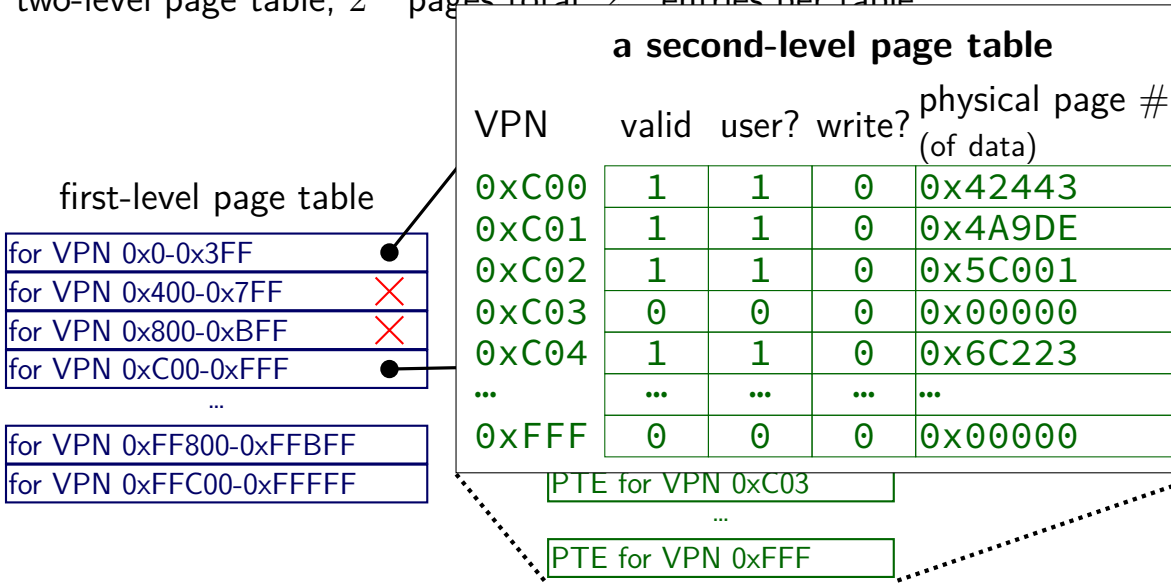
PTE for VPN 0xC03

...

PTE for VPN 0xFFF

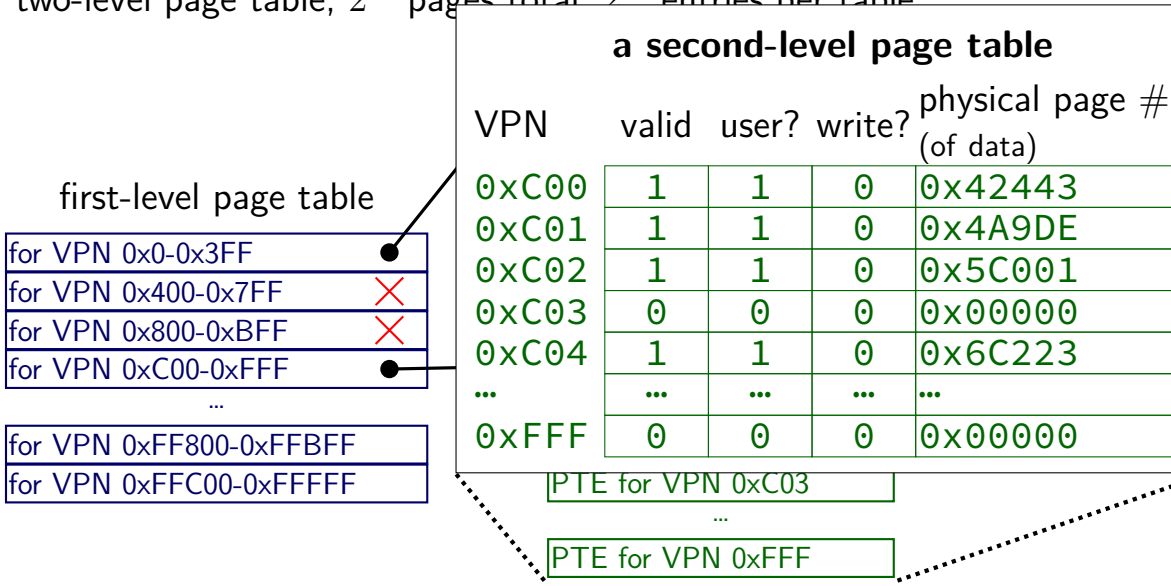
two-level page tables

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

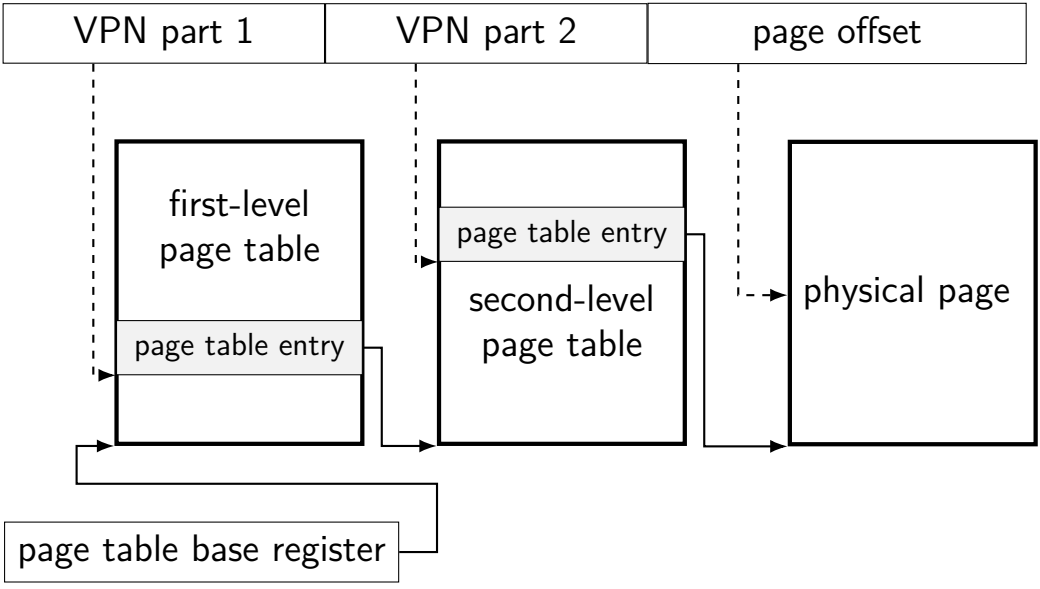
```
// A virtual address 'va' has a three-part structure as follows:
//
// +--------10------+-------10-------+---------12----------+
// | Page Directory |   Page Table    | Offset within Page  |
// |      Index     |      Index      |                     |
// +----------------+-----------------+---------------------+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

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? 2 pages (1 first-level, 1 second)
- a process's x86-32 page table with all 4K pages populated?

exercise (2)

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

4-byte page table entries — stored in memory

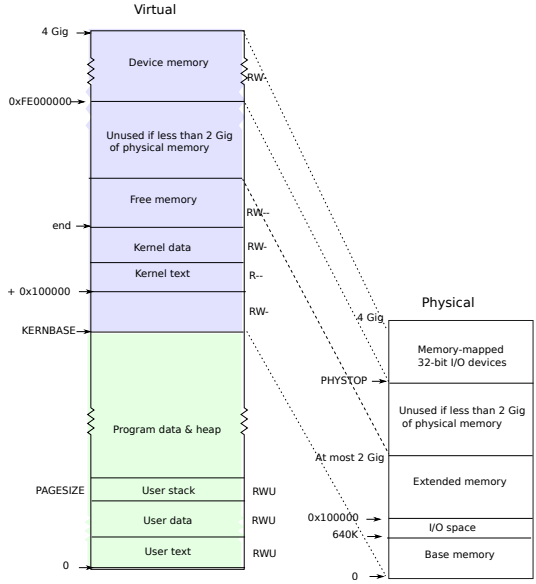
two-level table:

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

exercise: how big is...

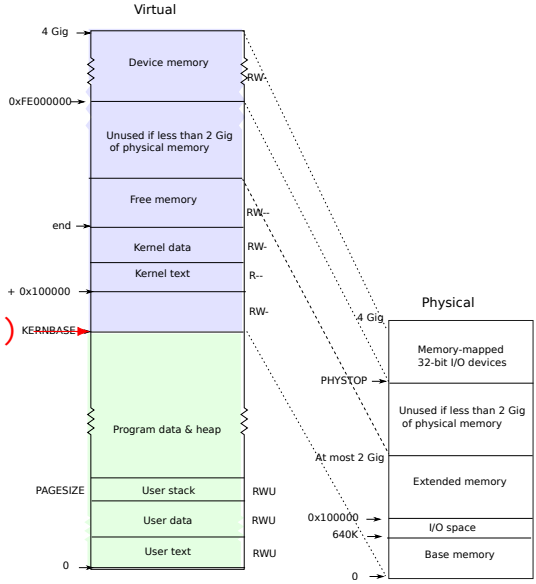
a process's x86-32 page tables with 1 valid 4K page? 2 pages (1 first-level, 1 second)
a process's x86-32 page table with all 4K pages populated? 1025 pages (1 first-level, 1024 second)

xv6 memory layout

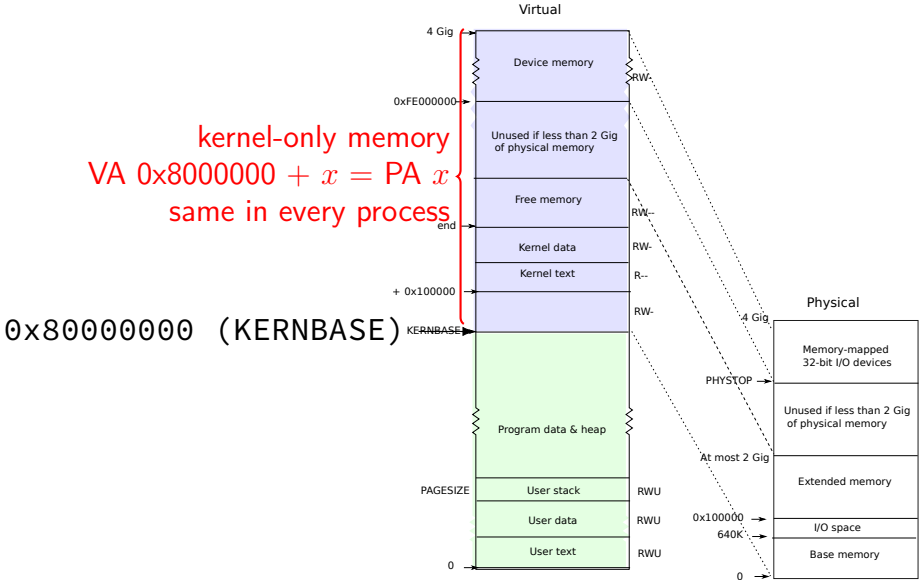


xv6 memory layout

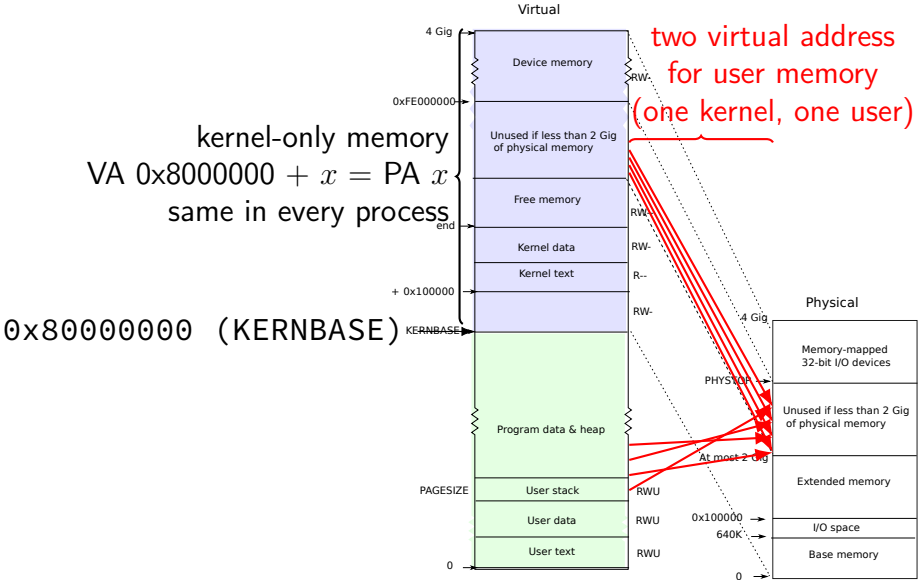
0x80000000 (KERNBASE)



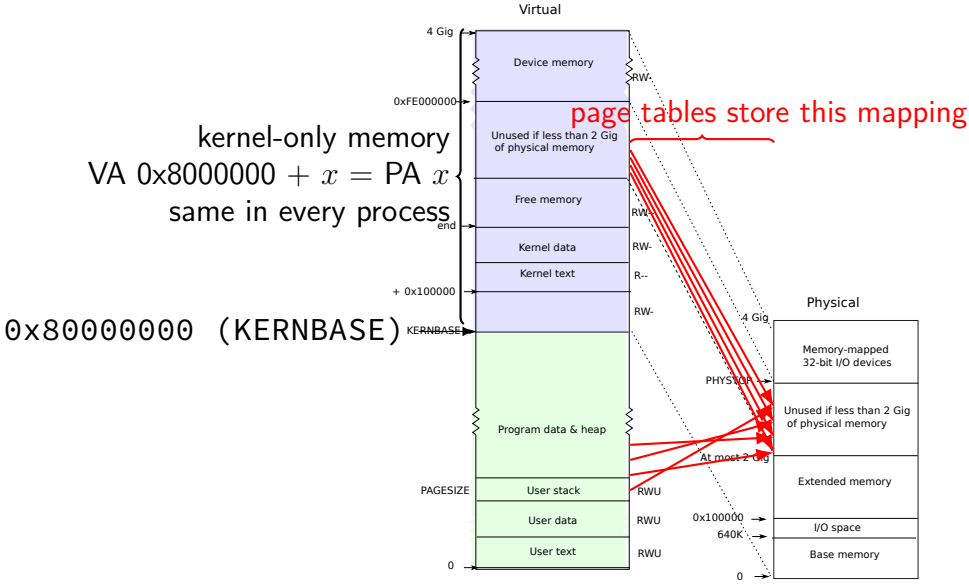
xv6 memory layout



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

why two mappings?

program memory: layout programs expect

- sized based on executable, heap allocations
- uses any available memory

kernel code: access to all memory

kernel code: easy translation of physical to virtual addresses

- e.g. page table setup: want to use particular physical addresses
- no x86 instruction to read/write value using physical address only