last time

counting semaphores

- down: decrement — wait first if would be negative
- up: increment — wake up if another thread waiting
- intuition: number that should be zero when waiting

reader/writer locks

- multiple readers share lock
- single writer at a time
- priority question: prefer readers or writers or other?

deadlocks

- circular dependencies resulting in indefinite waiting
- common with locks, but can happen with many resources
- classic example T1: Lock(A) Lock(B); T2: Lock(B) Lock(A)

deadlock

deadlock — circular waiting for resources

resource = something needed by a thread to do work

- locks
- CPU time
- disk space
- memory
- ...

often non-deterministic in practice

most common example: when acquiring multiple locks

deadlock

deadlock — circular waiting for resources

resource = something needed by a thread to do work

- locks
- CPU time
- disk space
- memory
- ...

often non-deterministic in practice

most common example: when acquiring multiple locks

deadlock versus starvation

starvation: one+ unlucky (no progress), one+ lucky (yes progress)

- example: low priority threads versus high-priority threads

deadlock: no one involved in deadlock makes progress

deadlock versus starvation

starvation: one+ unlucky (no progress), one+ lucky (yes progress)

- example: low priority threads versus high-priority threads

deadlock: no one involved in deadlock makes progress

starvation: once starvation happens, taking turns will resolve

- low priority thread just needed a chance...

deadlock: once it happens, taking turns won't fix

deadlock requirements

mutual exclusion

one thread at a time can use a resource

hold and wait

thread holding a resources waits to acquire *another* resource

no preemption of resources

resources are only released voluntarily

thread trying to acquire resources can't 'steal'

circular wait

there exists a set $\{T_1, \ldots, T_n\}$ of waiting threads such that

- T_1 is waiting for a resource held by T_2
- T_2 is waiting for a resource held by T_3
- …
- T_n is waiting for a resource held by T_1

how is deadlock possible?

Given list: A, B, C, D, E

```
RemoveNode(LinkedListNode *node) {
    pthread_mutex_lock(&node->lock);
    pthread_mutex_lock(&node->prev->lock);
    pthread_mutex_lock(&node->next->lock);
    node->next->prev = node->prev;
    node->prev->next = node->next;
    pthread_mutex_unlock(&node->next->lock);
    pthread_mutex_unlock(&node->prev->lock);
    pthread_mutex_unlock(&node->lock);
}
```

Which of these (all run in parallel) can deadlock?

A. RemoveNode(B) and RemoveNode(C)

B. RemoveNode(B) and RemoveNode(D)

C. RemoveNode(B) and RemoveNode(C) and RemoveNode(D)

D. A and C

E. B and C

F. all of the above

G. none of the above

how is deadlock — solution

Remove B	Remove C
lock B	lock C
lock A (prev)	wait to lock B (prev)
wait to lock C (next)	

With B and D — only overlap in in node C — no circular wait possible

deadlock prevention techniques

infinite resources or at least enough that never run out	no *mutual exclusion*
no shared resources	no *mutual exclusion*
no waiting "busy signal" — abort and (maybe) retry revoke/preempt resources	no *hold and wait*/ *preemption*
acquire resources in **consistent order**	no *circular wait*
request **all resources at once**	no *hold and wait*

deadlock prevention techniques

infinite resources or at least enough that never run out	no *mutual exclusion*
no shared resources	no *mutual exclusion*
no waiting "busy signal" — abort and (maybe) retry revoke/preempt resources	no *hold and wait*/ *preemption*
acquire resources in **consistent order**	no *circular wait*
request **all resources at once**	no *hold and wait*

deadlock prevention techniques

infinite resources — no *mutual exclusion*

or at least enough that never run out

no shared resources — no *mutual exclusion*

no waiting — no *hold and wait*/*preemption*

"busy signal" — abort and (maybe) retry

revoke/preempt resources

acquire resources in **consistent order** — no *circular wait*

request **all resources at once** — no *hold and wait*

deadlock prevention techniques

infinite resources — no *mutual exclusion*

or at least enough that never run out

memory allocation: malloc() fails rather than waiting (no deadlock)
locks: `pthread_mutex_trylock` fails rather than waiting
...

r — *exclusion*

no waiting — no *hold and wait*/ *preemption*

"busy signal" — abort and (maybe) retry

revoke/preempt resources

acquire resources in **consistent order** — no *circular wait*

request **all resources at once** — no *hold and wait*

deadlock prevention techniques

infinite resources — no *mutual exclusion*

or at least enough that never run out

no shared resources — no *mutual exclusion*

no waiting — no *hold and wait*/ *preemption*

"busy signal" — abort and (maybe) retry

revoke/preempt resources

acquire resources in **consistent order** — no *circular wait*

request **all resources at once** — no *hold and wait*

deadlock prevention techniques

infinite resources — no *mutual exclusion*

or at least enough that never run out

no shared resources — no *mutual exclusion*

requires some way to undo partial changes to avoid errors
common approach for databases
...

no waiti[ng] — no *hold and wait*/ *preemption*

"busy signal" — abort and (maybe) retry

revoke/preempt resources

acquire resources in **consistent order** — no *circular wait*

request **all resources at once** — no *hold and wait*

acquiring locks in consistent order (1)

```
MoveFile(Dir* from_dir, Dir* to_dir, string filename) {
  if (from_dir->path < to_dir->path) {
    lock(&from_dir->lock);
    lock(&to_dir->lock);
  } else {
    lock(&to_dir->lock);
    lock(&from_dir->lock);
  }
  ...
}
```

acquiring locks in consistent order (1)

```
MoveFile(Dir* from_dir, Dir* to_dir, string filename) {
  if (from_dir->path < to_dir->path) {
    lock(&from_dir->lock);
    lock(&to_dir->lock);
  } else {
    lock(&to_dir->lock);
    lock(&from_dir->lock);
  }
  ...
}
```

any ordering will do
e.g. compare pointers

acquiring locks in consistent order (2)

often by convention, e.g. Linux kernel comments:

```
/*
 * ...
 * Lock order:
 *       contex.ldt_usr_sem
 *         mmap_sem
 *           context.lock
 */
```

```
/*
 * ...
 * Lock order:
 *   1. slab_mutex (Global Mutex)
 *   2. node->list_lock
 *   3. slab_lock(page) (Only on some arches and for debugging)
 * ...
 */
```

deadlock prevention techniques

infinite resources or at least enough that never run out	no *mutual exclusion*
no shared resources	no *mutual exclusion*
no waiting "busy signal" — abort and (maybe) retry revoke/preempt resources	no *hold and wait*/ *preemption*
acquire resources in **consistent order**	no *circular wait*
request **all resources at once**	no *hold and wait*

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;
        ...
    }
}
```

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

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?

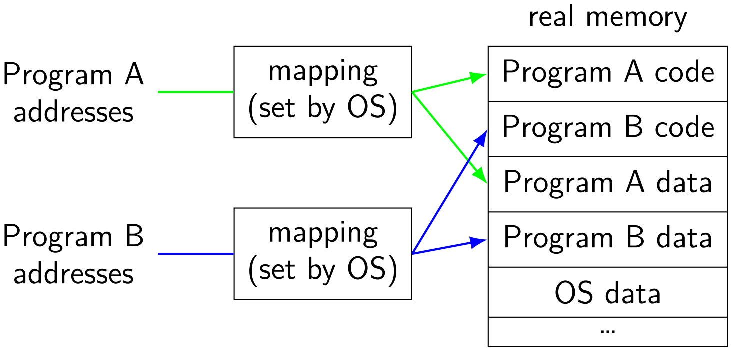
- don't have precise information about that

scheduling note on paging/protection

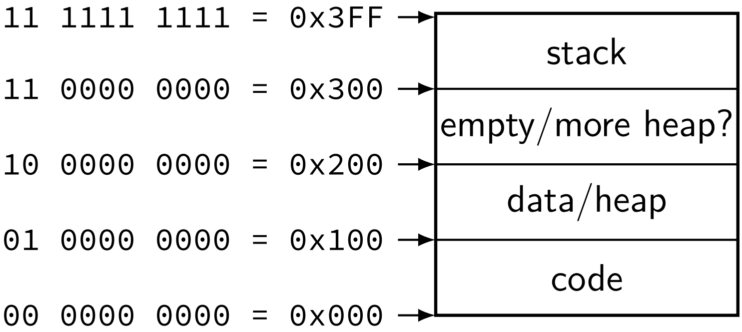
not sure if we'll get to enough for next assignment by Thursday

may adjust deadlines for that (and future assignments)

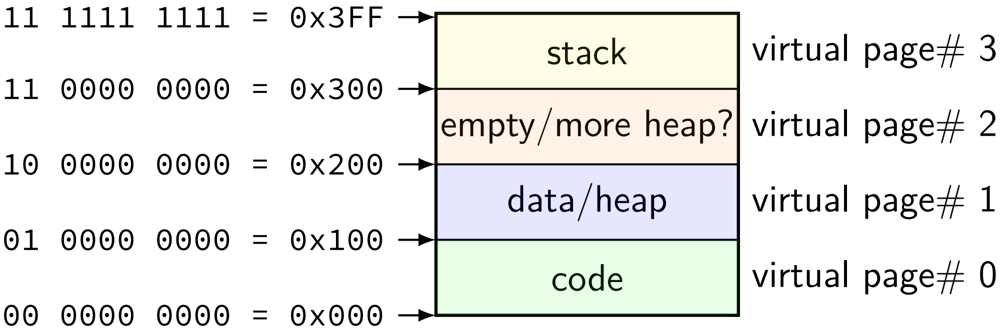
address translation



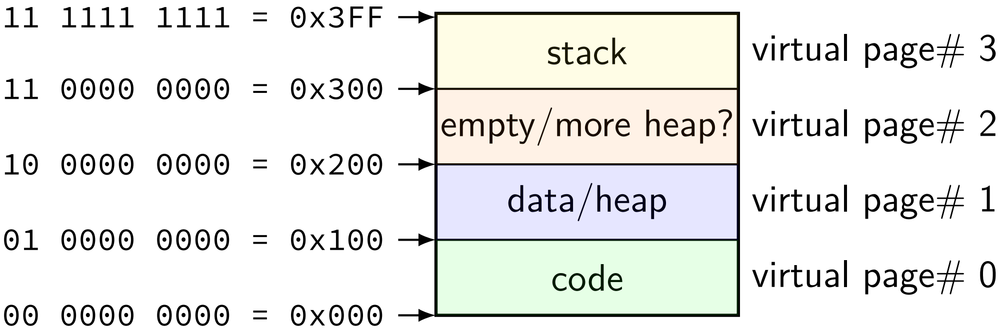
toy program memory



toy program memory

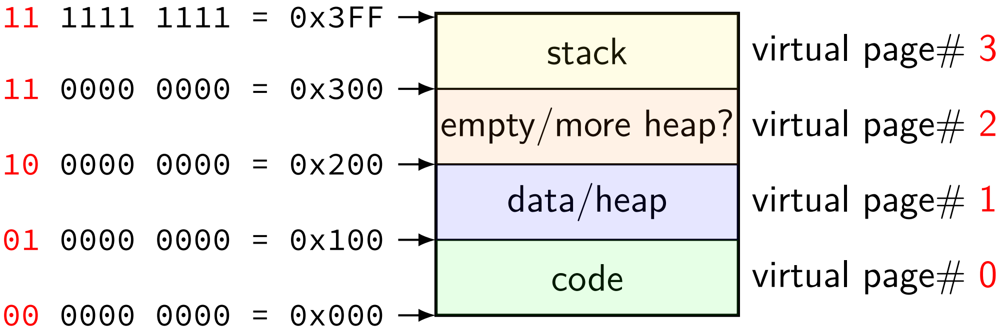


toy program memory



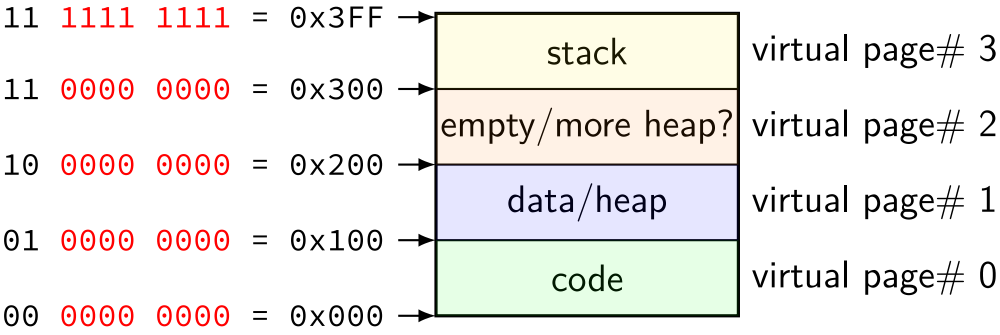
divide memory into pages (2^8 bytes in this case)
"virtual" = addresses the program sees

toy program memory



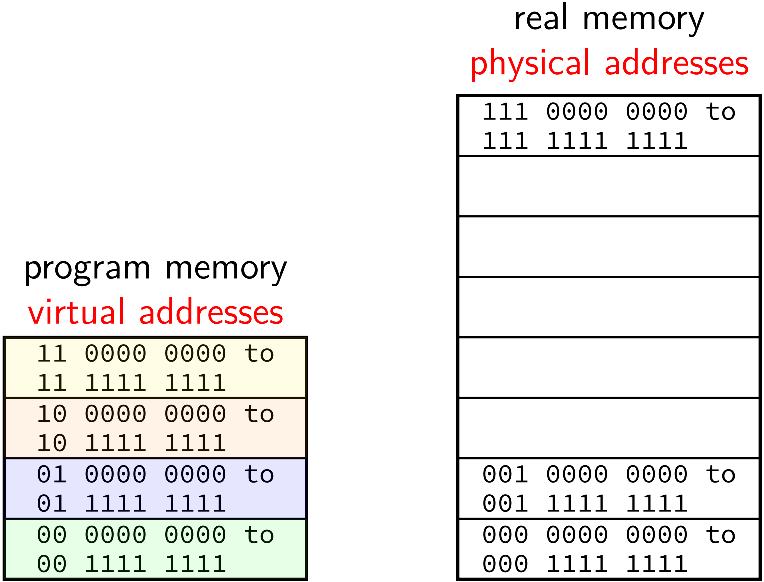
page number is upper bits of address
(because page size is power of two)

toy program memory

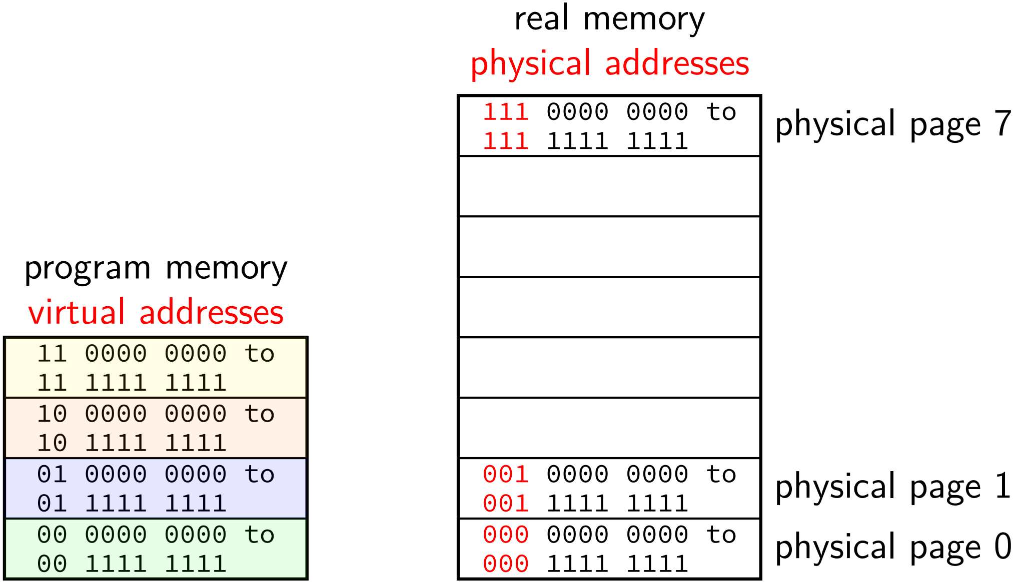


rest of address is called page offset

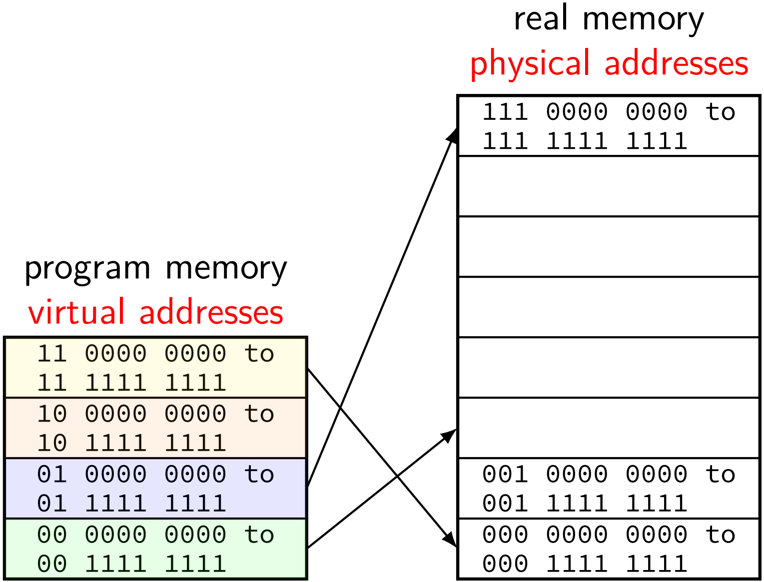
toy physical memory



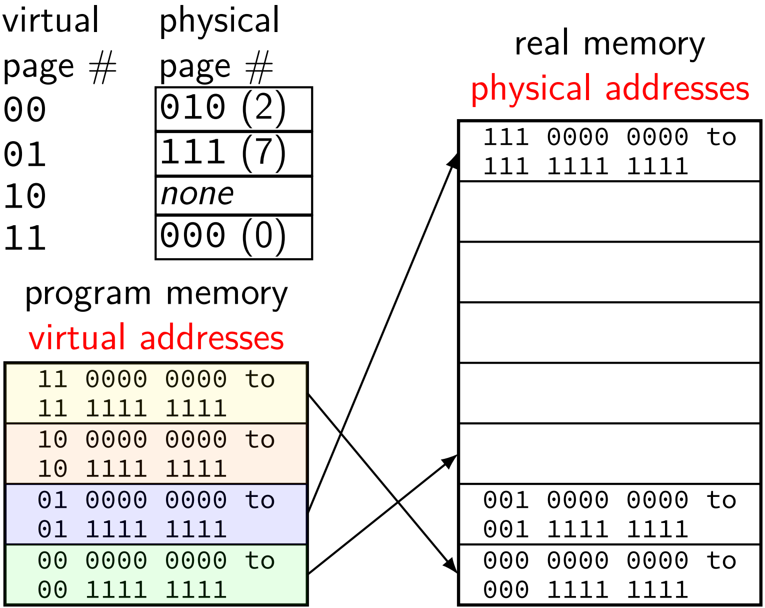
toy physical memory



toy physical memory



toy physical memory



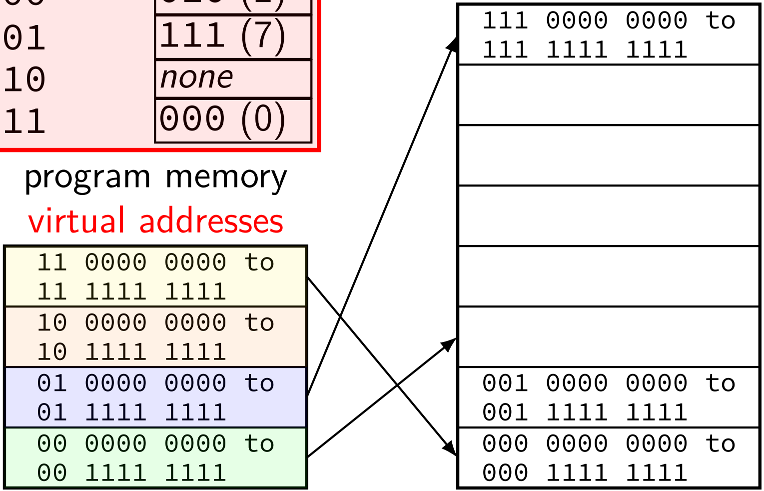
toy physical memory

page table!

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

program memory
virtual addresses

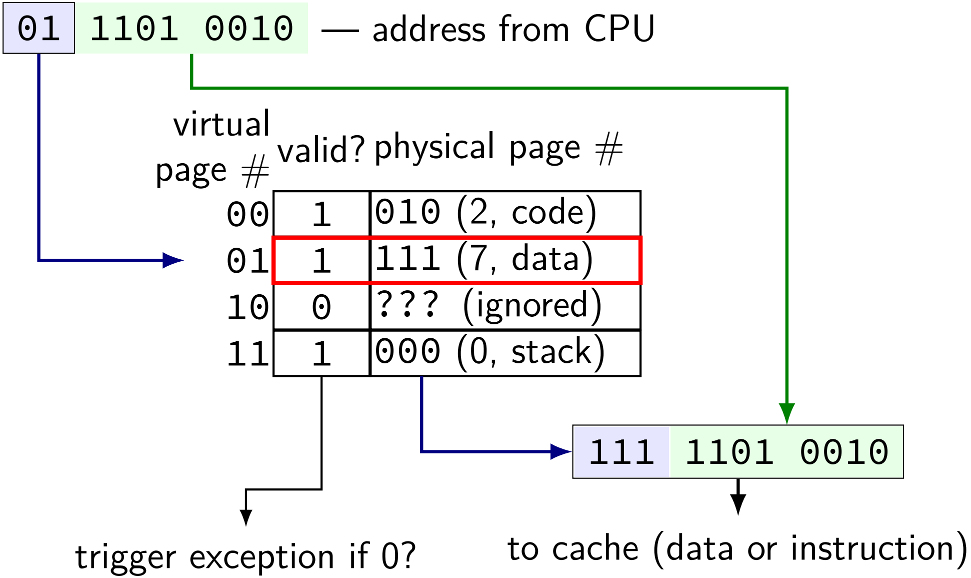
real memory
physical addresses



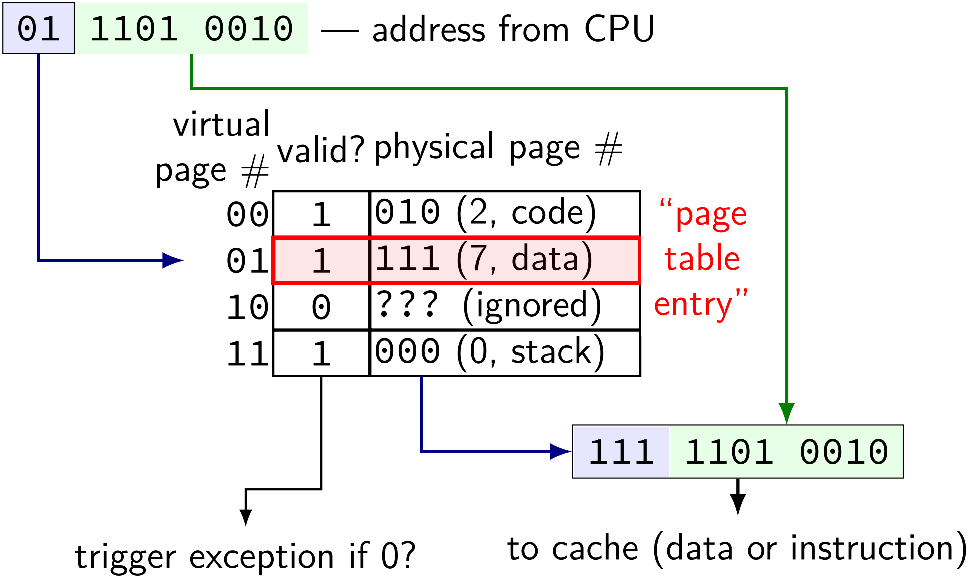
toy page table lookup

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

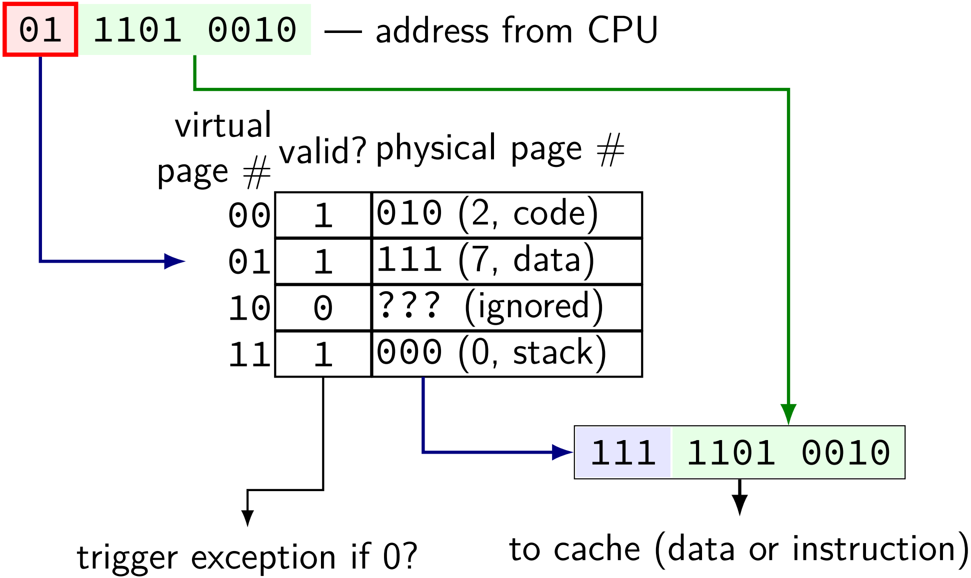
toy page table lookup



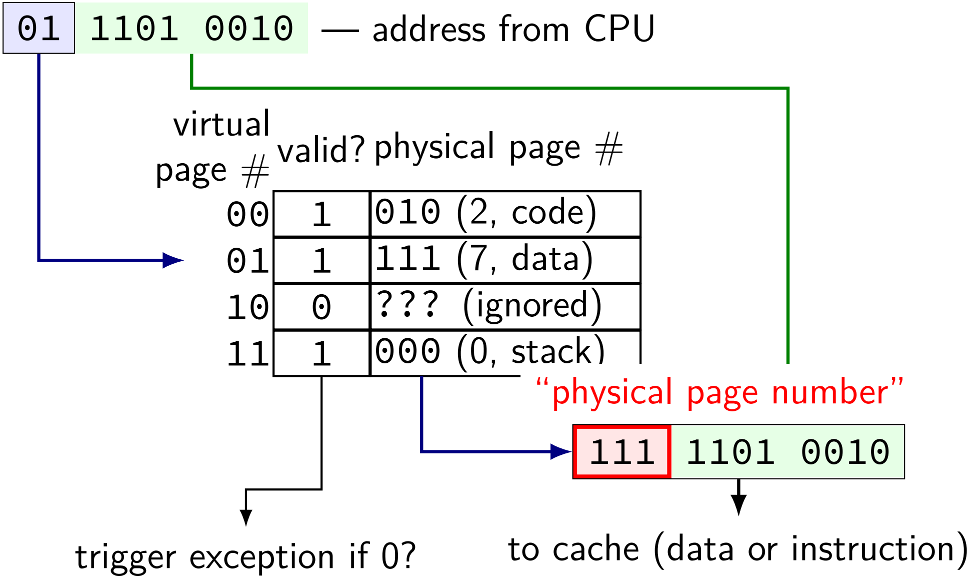
toy page table lookup



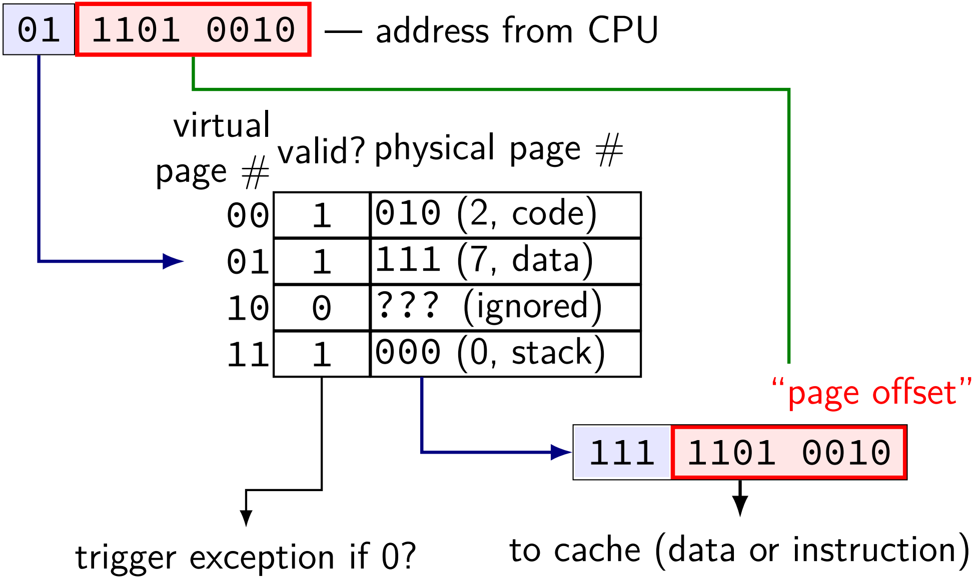
“virtual page number” lookup



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`

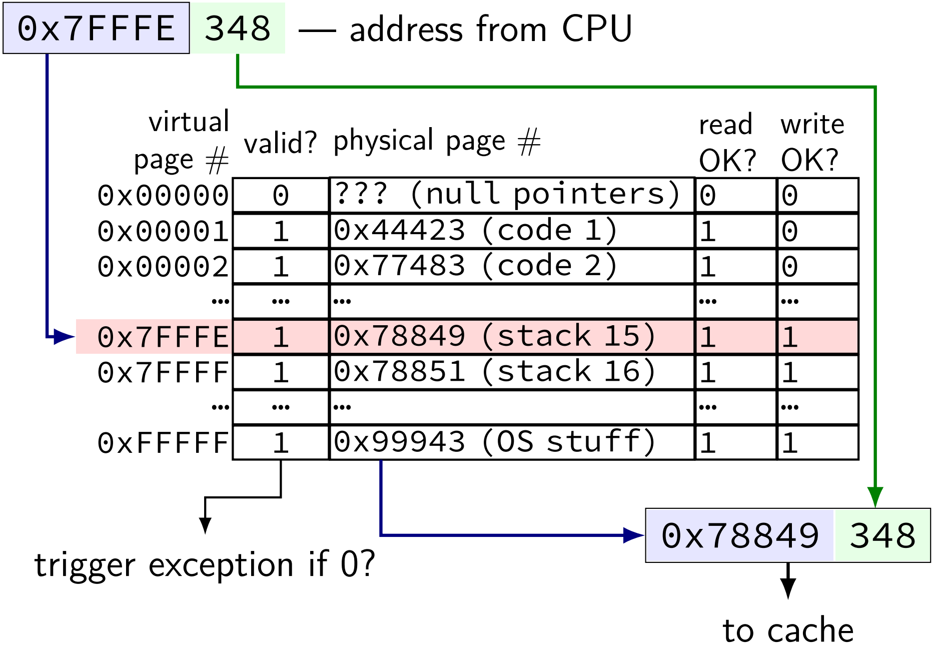
virtual page number = `0xABCD0`

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???

0x7FFFE | 348 — address from CPU

virtual page #	valid?	physical page #	read OK?	write OK?
0x00000	0	??? (null pointers)	0	0
0x00001	1	0x44423 (code 1)	1	0
0x00002	1	0x77483 (code 2)	1	0
…	…	…	…	…
0x7FFFE	1	0x78849 (stack 15)	1	1
0x7FFFF	1	0x78851 (stack 16)	1	1
…	…	…	…	…
0xFFFFF	1	0x99943 (OS stuff)	1	1

2^{20} entries???
way too big!

trigger exception if 0?

0x78849 | 348

to cache

storing huge page table?

keep it in memory

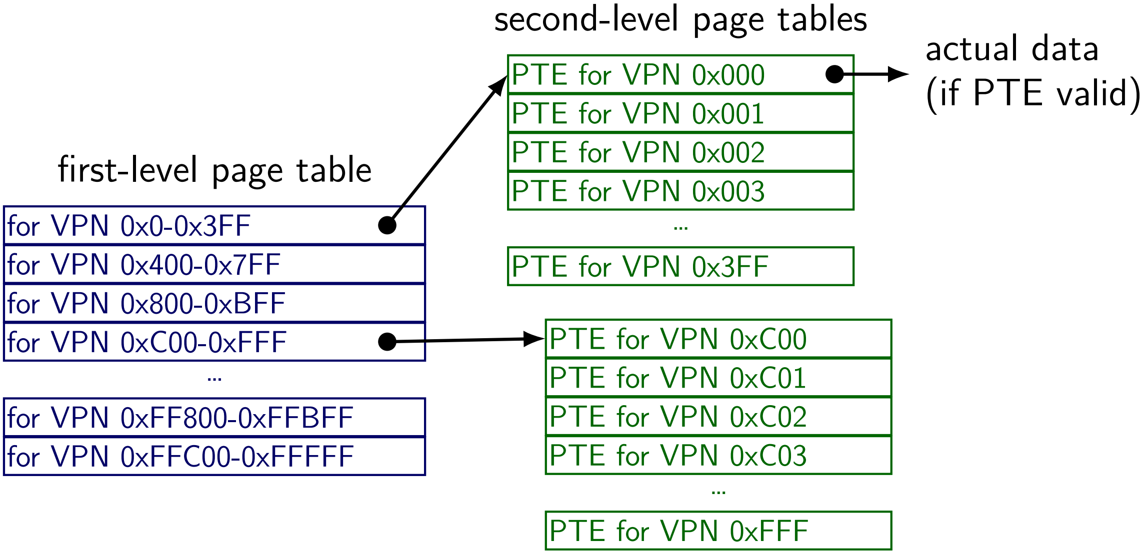
- add special cache for page table entries to handle memory being slow
- special cached called translation lookaside buffer (TLB)

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

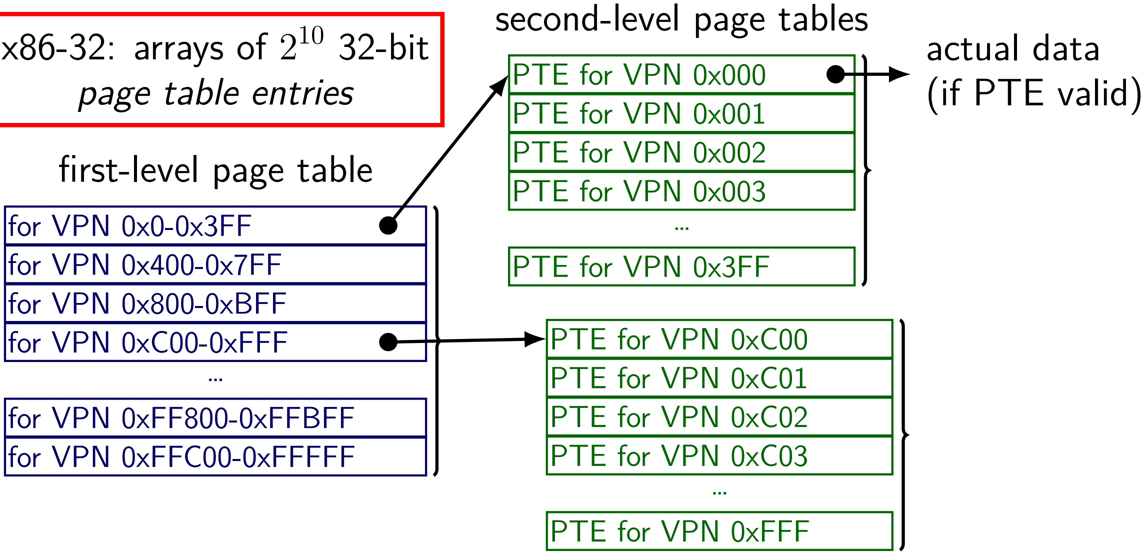
two-level page table; 2^{20} pages total; 2^{10} entries per table



two-level page tables

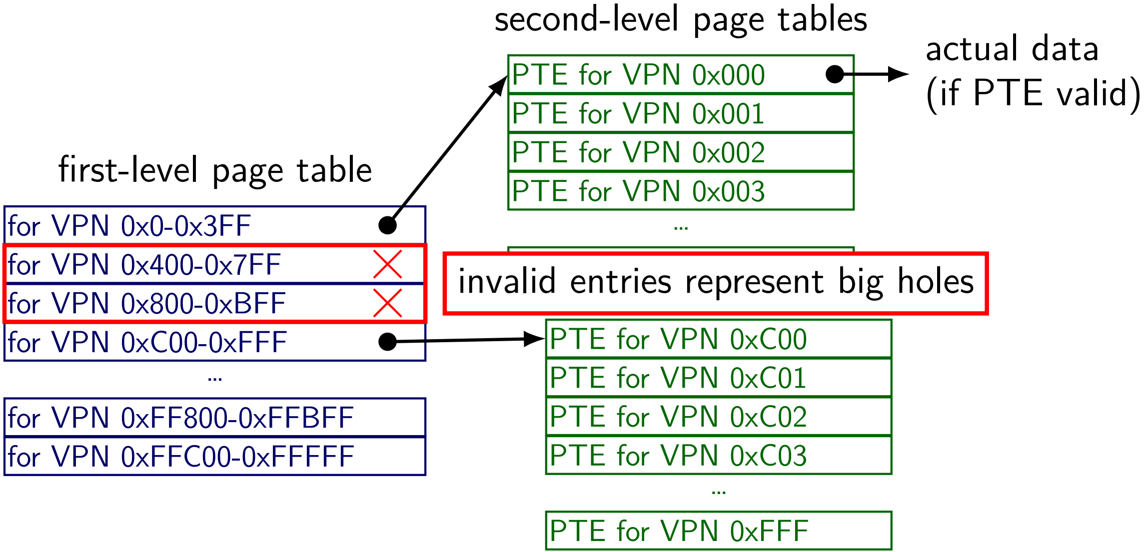
two-level page table; 2^{20} pages total; 2^{10} entries per table

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

PTE for VPN 0xC03
...
PTE for VPN 0xFFF

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

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

PTE for VPN 0xC03

...

PTE for VPN 0xFFF

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	[illegible]	[illegible]	0	0x00000
0xC00–0xFFF	[illegible]	[illegible]	0	0x33454
0x1000–0x13F	[illegible]	[illegible]	0	0xFF043
...	...	...	...	...
0xFFC00–0xFFFFF	1	1	0	0xFF045

pointers to page tables (arrays of PTEs)
but using page number (not byte number)

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

PTE for VPN 0xC03

...

PTE for VPN 0xFFF

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

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 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

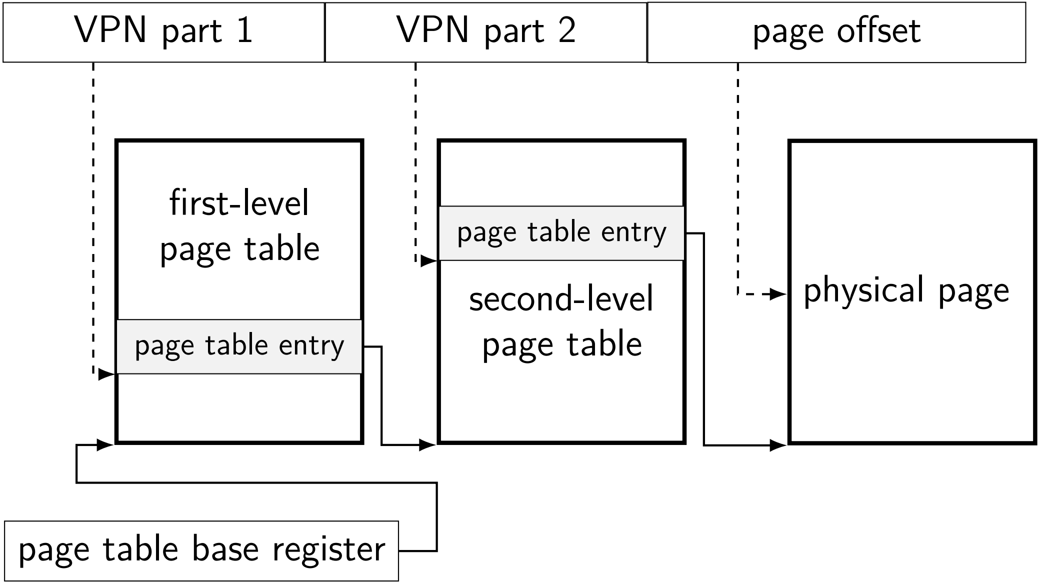
```
// A virtual address 'va' has a three-part structure as follows:
//
// +--------10------+-------10-------+---------12----------+
// | Page Directory |   Page Table   | Offset within Page  |
// |      Index     |      Index     |                     |
// +----------------+----------------+---------------------+
//  \--- 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 *contents*: 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 tables 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 tables 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 tables with all 4K pages populated? 1025 pages (1 first-level, 1024 second)

backup slides

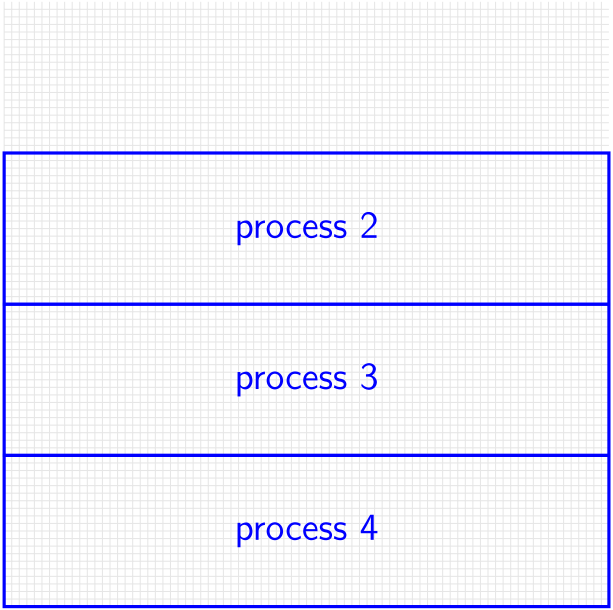
message passing API

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

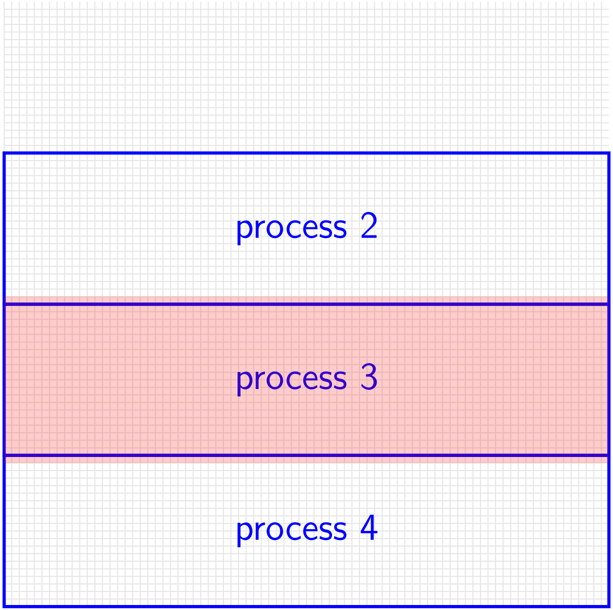


divide grid
like you would for normal threads

each process stores cells
in that part of grid

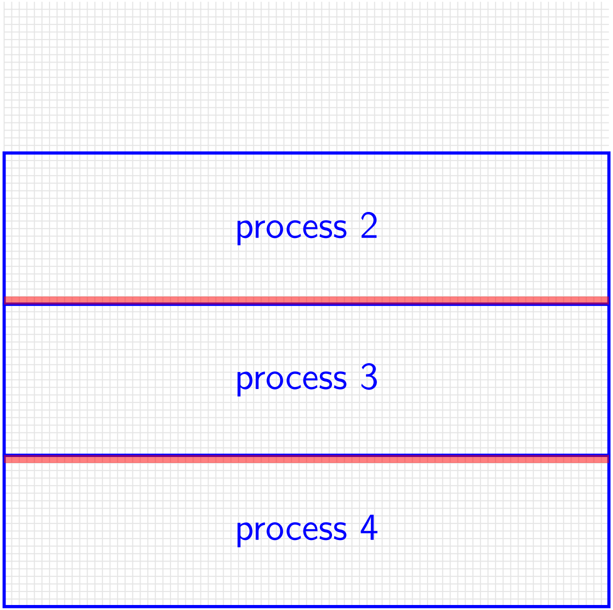
(no shared memory!)

message passing game of life



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

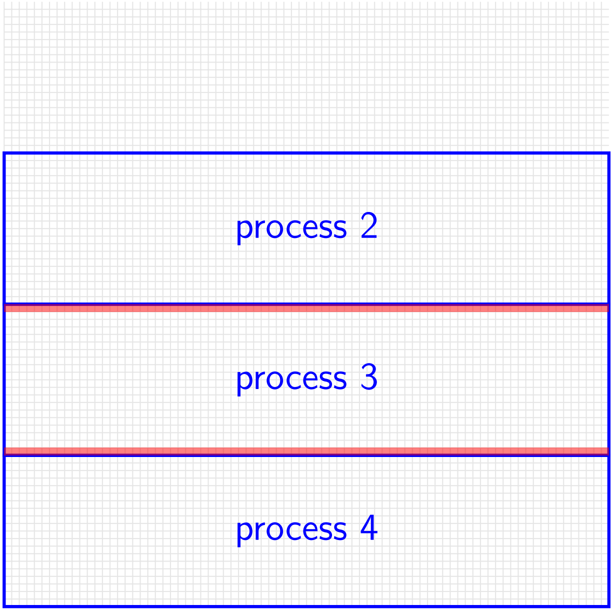
message passing game of life



small slivers of
other process's cells needed

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

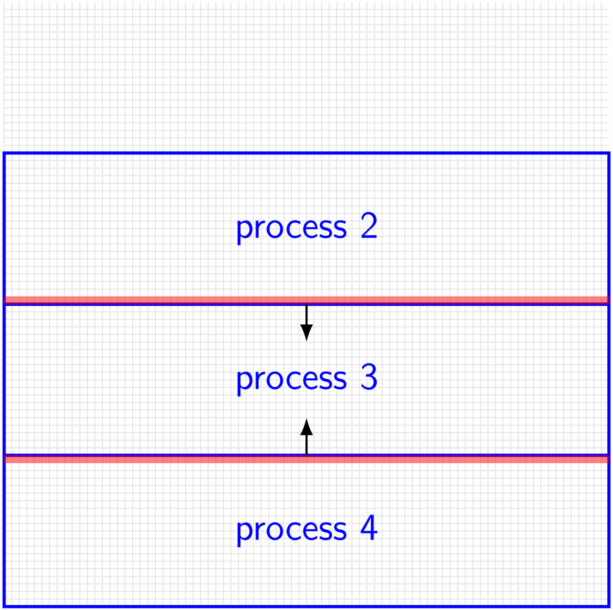
message passing game of life



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

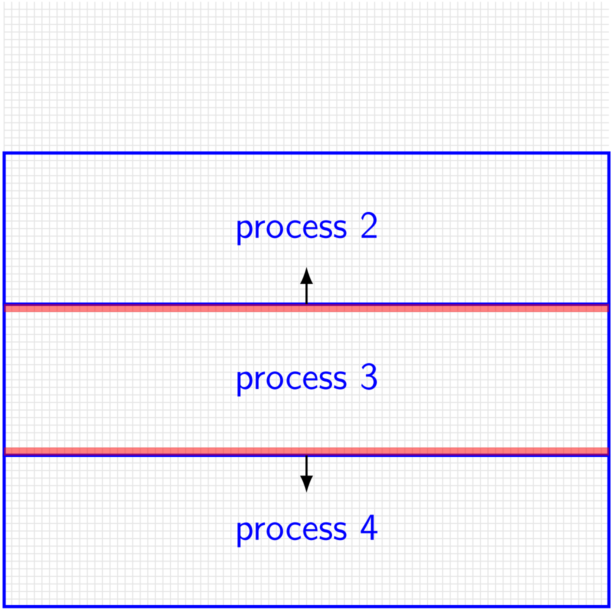
so process 3 also sends messages

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)

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
```

deadlock with free space

Thread 1	Thread 2
`AllocateOrWaitFor(1 MB)`	`AllocateOrWaitFor(1 MB)`
`AllocateOrWaitFor(1 MB)`	`AllocateOrWaitFor(1 MB)`
`(do calculation)`	`(do calculation)`
`Free(1 MB)`	`Free(1 MB)`
`Free(1 MB)`	`Free(1 MB)`

2 MB of space — deadlock possible with unlucky order

deadlock with free space (unlucky case)

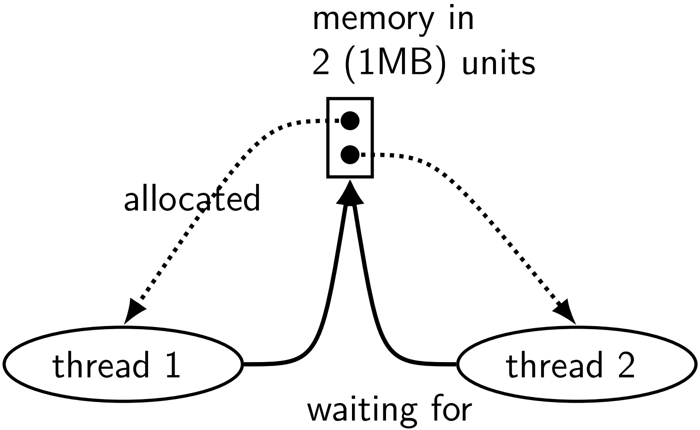
Thread 1

```
AllocateOrWaitFor(1 MB)
AllocateOrWaitFor(1 MB…
```
stalled

Thread 2

```
AllocateOrWaitFor(1 MB)
AllocateOrWaitFor(1 MB…
```
stalled

free space: dependency graph



deadlock with free space (lucky case)

Thread 1

```
AllocateOrWaitFor(1 MB)
AllocateOrWaitFor(1 MB)
(do calculation)
Free(1 MB);
Free(1 MB);
```

Thread 2

```
AllocateOrWaitFor(1 MB)
AllocateOrWaitFor(1 MB)
(do calculation)
Free(1 MB);
Free(1 MB);
```

AllocateOrFail

Thread 1	Thread 2
`AllocateOrFail(1 MB)`	
	`AllocateOrFail(1 MB)`
`AllocateOrFail(1 MB)` fails!	
	`AllocateOrFail(1 MB)` fails!
`Free(1 MB)` (cleanup after failure)	
	`Free(1 MB)` (cleanup after failure)

okay, now what?

- give up?
- both try again? — maybe this will keep happening? (called livelock)
- try one-at-a-time? — gaurenteed to work, but tricky to implement

AllocateOrSteal

Thread 1	Thread 2
`AllocateOrSteal(1 MB)`	
	`AllocateOrSteal(1 MB)`
`AllocateOrSteal(1 MB)`	`Thread killed to free 1MB`
(do work)	

problem: can one actually implement this?

problem: can one kill thread and keep system in consistent state?

fail/steal with locks

pthreads provides pthread_mutex_trylock — “lock or fail”

some databases implement *revocable locks*

- do equivalent of throwing exception in thread to ‘steal’ lock
- need to carefully arrange for operation to be cleaned up

stealing locks???

how do we make stealing locks possible

unclean: just kill the thread

- problem: inconsistent state?

clean: have code to undo partial oepration

- some databases do this

won't go into detail in this class

revokable locks?

```
try {
    AcquireLock();
    use shared data
} catch (LockRevokedException le) {
    undo operation hopefully?
} finally {
    ReleaseLock();
}
```

deadlock prevention techniques

infinite resources — no *mutual exclusion*

or at least enough that never run out

no shared resources — no *mutual exclusion*

no waiting — no *hold and wait*/*preemption*

"busy signal" — abort and (maybe) retry

revoke/preempt resources

acquire resources in **consistent order** — no *circular wait*

request **all resources at once** — no *hold and wait*

abort and retry limits?

abort-and-retry

how many times will you retry?

moving two files: abort-and-retry

```
struct Dir {
  mutex_t lock; map<string, DirEntry> entries;
};
void MoveFile(Dir *from_dir, Dir *to_dir, string filename) {
  while (true) {
    mutex_lock(&from_dir->lock);
    if (mutex_trylock(&to_dir->lock) == LOCKED) break;
    mutex_unlock(&from_dir->lock);
  }

  to_dir->entries[filename] = from_dir->entries[filename];
  from_dir->entries.erase(filename);

  mutex_unlock(&to_dir->lock);
  mutex_unlock(&from_dir->lock);
}
```

Thread 1: `MoveFile(A, B, "foo")`

Thread 2: `MoveFile(B, A, "bar")`

moving two files: lots of bad luck?

Thread 1 `MoveFile(A, B, "foo")`	Thread 2 `MoveFile(B, A, "bar")`
`lock(&A->lock)` → `LOCKED`	
	`lock(&B->lock)` → `LOCKED`
`trylock(&B->lock)` → `FAILED`	
	`trylock(&A->lock)` → `FAILED`
`unlock(&A->lock)`	
	`unlock(&B->lock)`
`lock(&A->lock)` → `LOCKED`	
	`lock(&B->lock)` → `LOCKED`
`trylock(&B->lock)` → `FAILED`	
	`trylock(&A->lock)` → `FAILED`
`unlock(&A->lock)`	
	`unlock(&B->lock)`

livelock

livelock: keep aborting and retrying without end

like deadlock — no one's making progress

potentially forever

unlike deadlock — threads are not waiting

preventing livelock

make schedule random — e.g. random waiting after abort

make threads run one-at-a-time if lots of aborting

other ideas?

deadlock detection

idea: search for cyclic dependencies

detecting deadlocks on locks

let's say I want to detect deadlocks that only involve mutexes

goal: help programmers debug deadlocks

...by modifying my threading library:

```
struct Thread {
    ... /* stuff for implementing thread */
    /* what extra fields go here? */



};

struct Mutex {
    ... /* stuff for implementing mutex */
    /* what extra fields go here? */



};
```

deadlock detection

idea: search for cyclic dependencies

need:

- list of all contended resources
- what thread is waiting for what?
- what thread ‘owns’ what?

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?

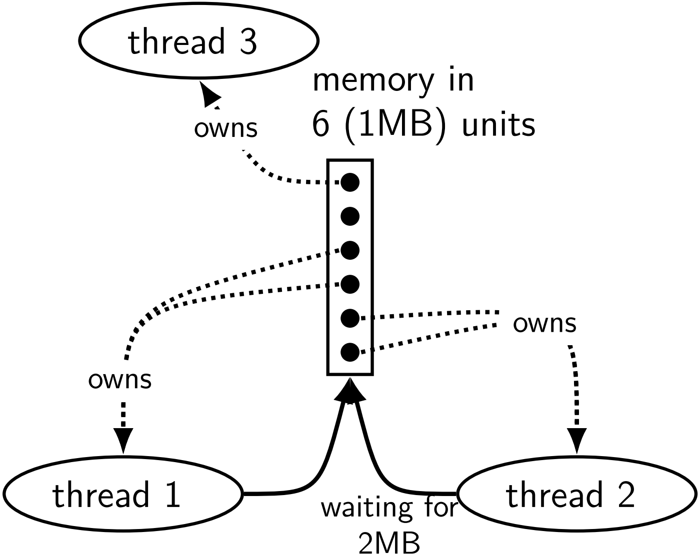
not a deadlock — thread 3 can still finish

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

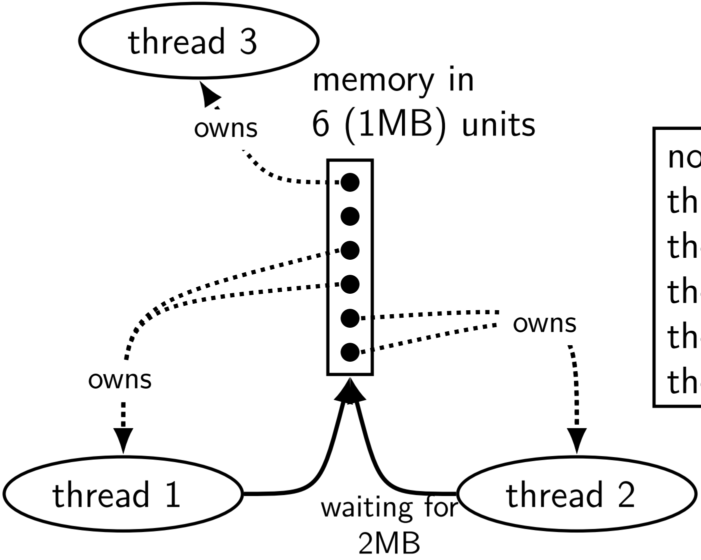
...but would be deadlock

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

divisible resources: not deadlock

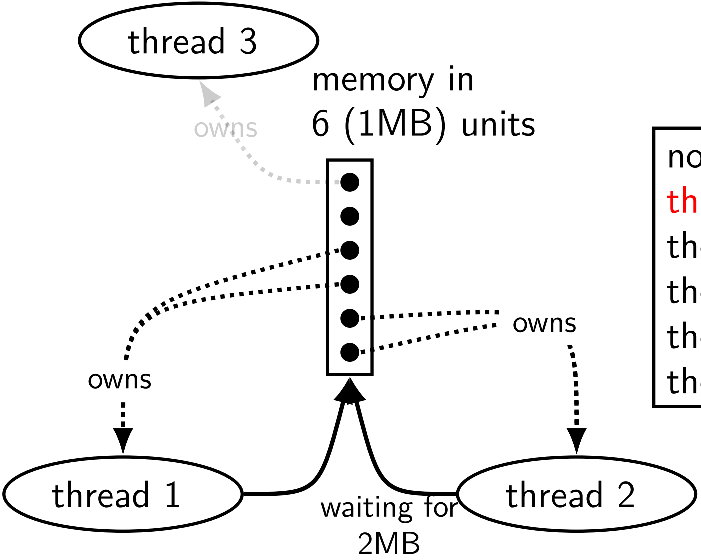


divisible resources: not deadlock



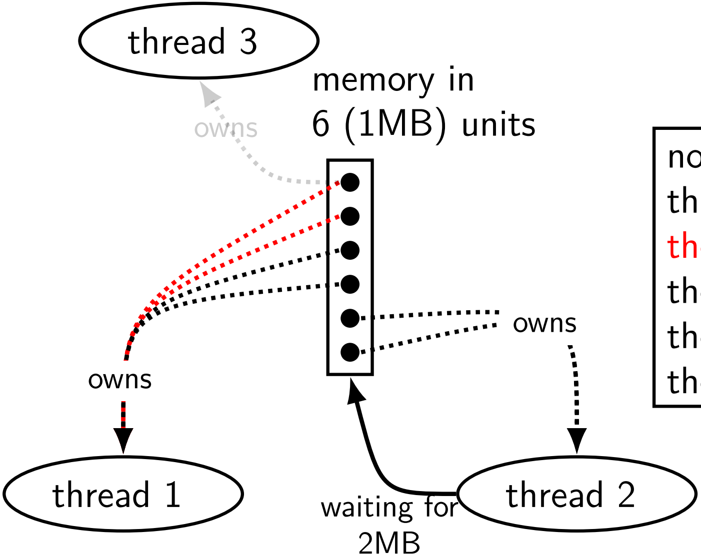
not deadlock:
thread 3 finishes
then thread 1 can get memory
then thread 1 finishes
then thread 2 can get resources
then thread 2 can finish

divisible resources: not deadlock



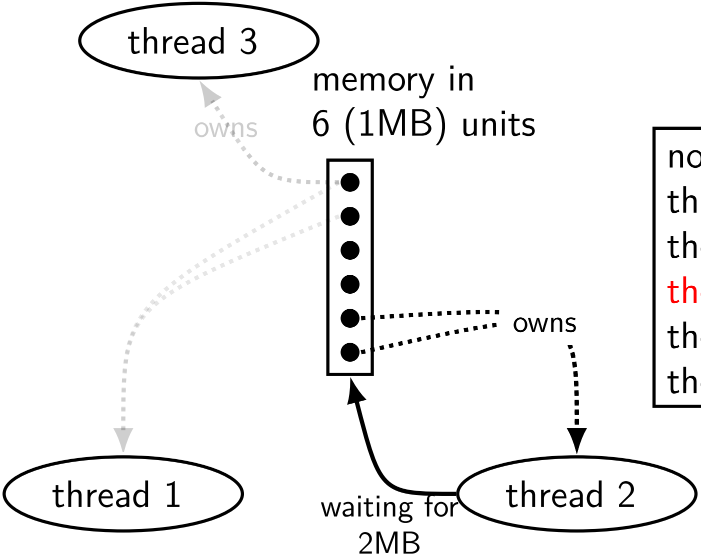
not deadlock:
thread 3 finishes
then thread 1 can get memory
then thread 1 finishes
then thread 2 can get resources
then thread 2 can finish

divisible resources: not deadlock



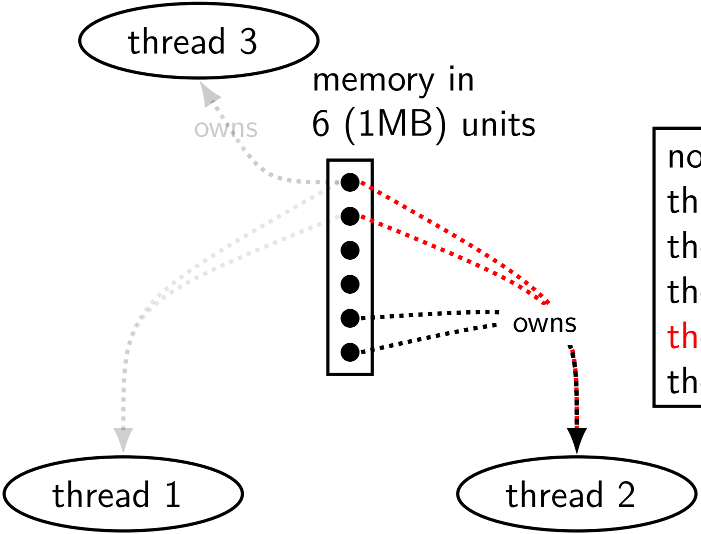
not deadlock:
thread 3 finishes
then thread 1 can get memory
then thread 1 finishes
then thread 2 can get resources
then thread 2 can finish

divisible resources: not deadlock



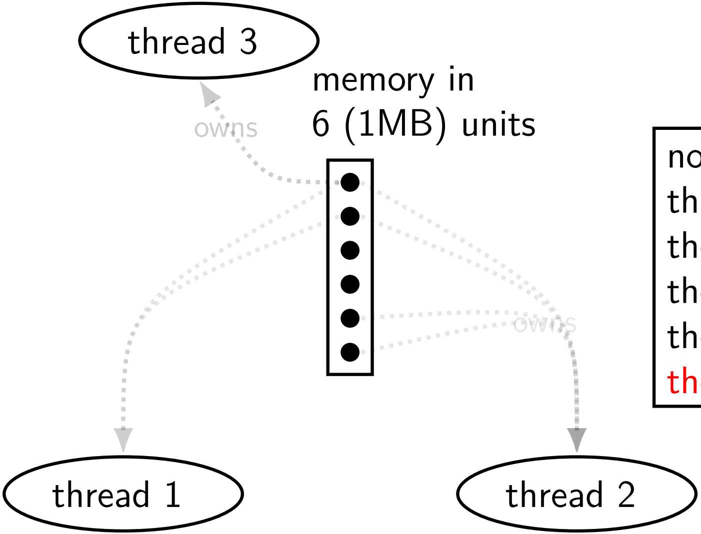
not deadlock:
thread 3 finishes
then thread 1 can get memory
then thread 1 finishes
then thread 2 can get resources
then thread 2 can finish

divisible resources: not deadlock



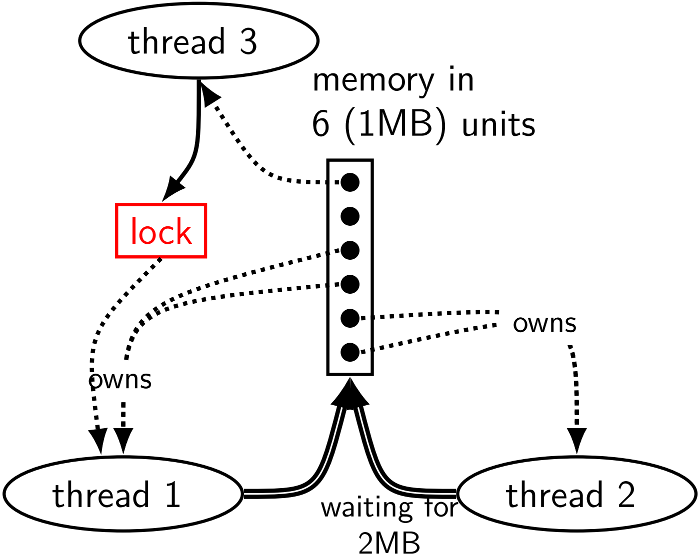
not deadlock:
thread 3 finishes
then thread 1 can get memory
then thread 1 finishes
then thread 2 can get resources
then thread 2 can finish

divisible resources: not deadlock

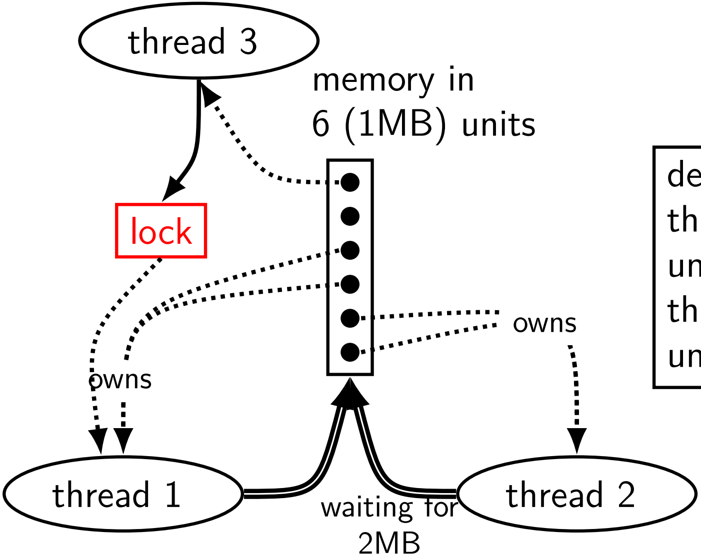


not deadlock:
thread 3 finishes
then thread 1 can get memory
then thread 1 finishes
then thread 2 can get resources
then thread 2 can finish

divisible resources: is deadlock

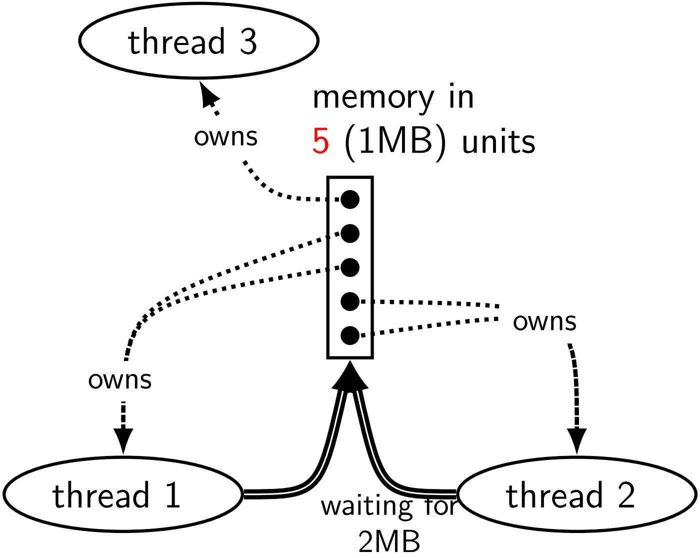


divisible resources: is deadlock

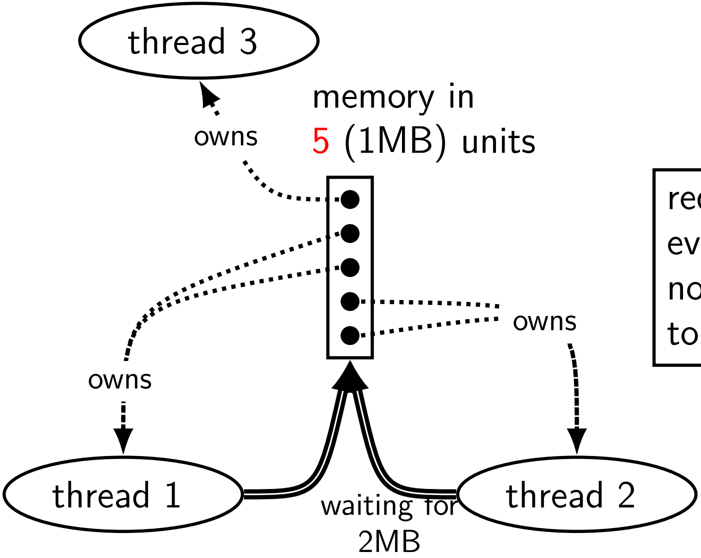


deadlock:
thread 3 can't finish
until thread 1 releases lock, but
thread 1 can't finish
until thread 3 releases memory

divisible resources: is deadlock

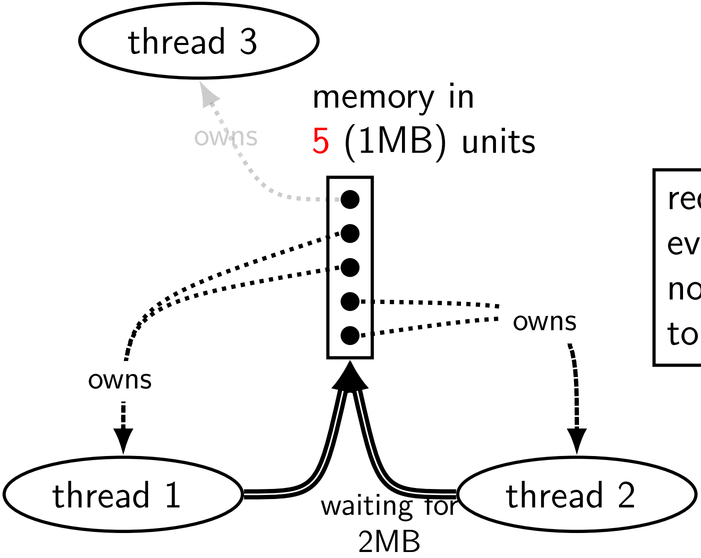


divisible resources: is deadlock



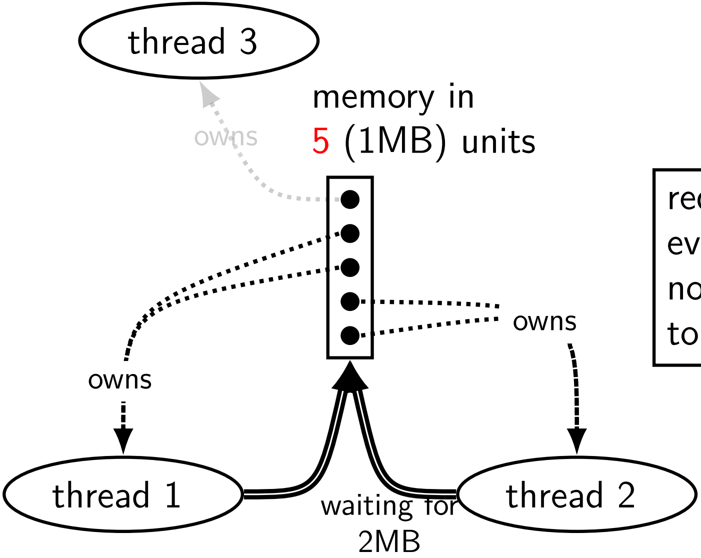
reducing memory: deadlock:
even after thread 3 finishes
no way for thread 1+2
to get what they want

divisible resources: is deadlock



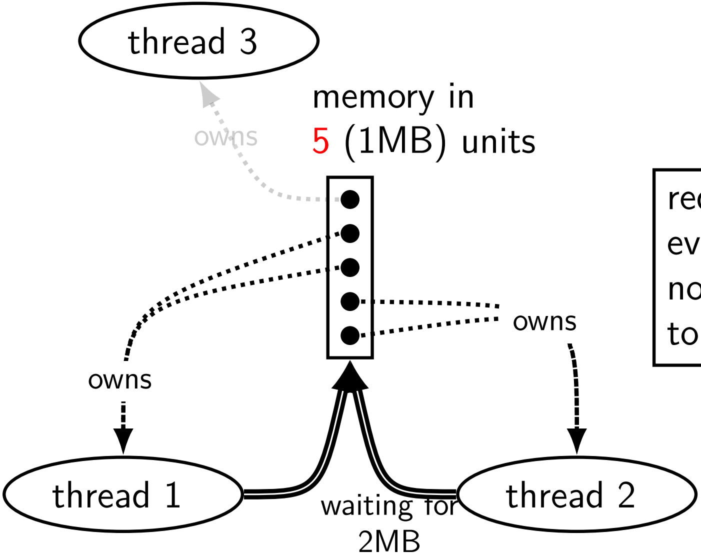
reducing memory: deadlock:
even after thread 3 finishes
no way for thread 1+2
to get what they want

divisible resources: is deadlock



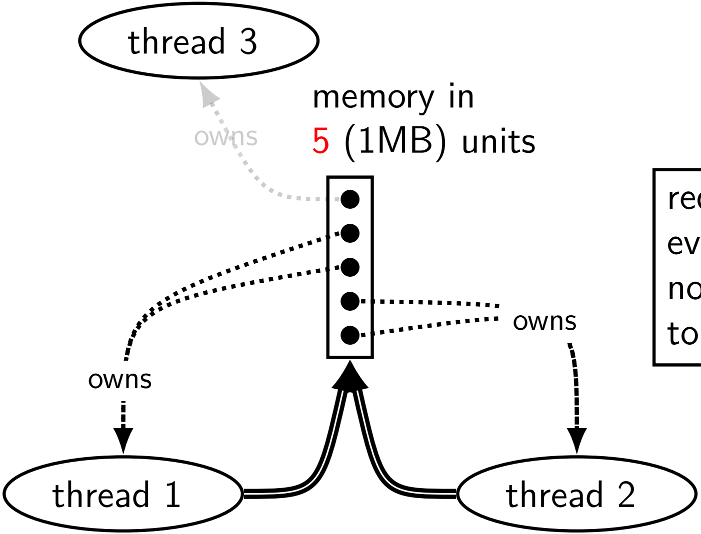
reducing memory: deadlock:
even after thread 3 finishes
no way for thread 1+2
to get what they want

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

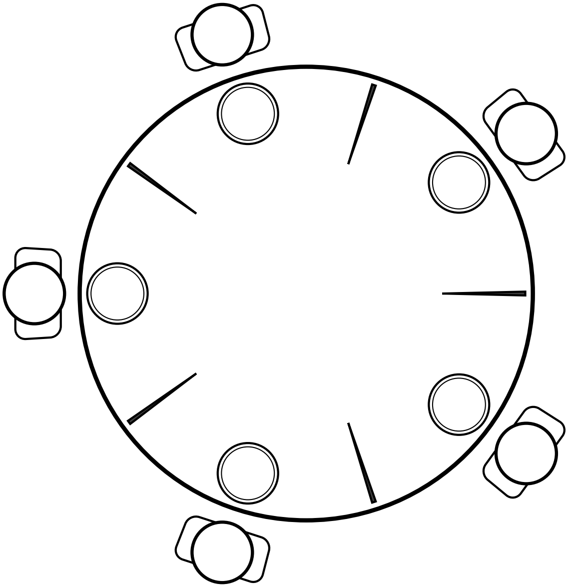
- similar technique to how we showed no deadlock

high-level intuition: simulate what could happen

- find threads that could finish based on resources available now

full details: look up Baker's algorithm

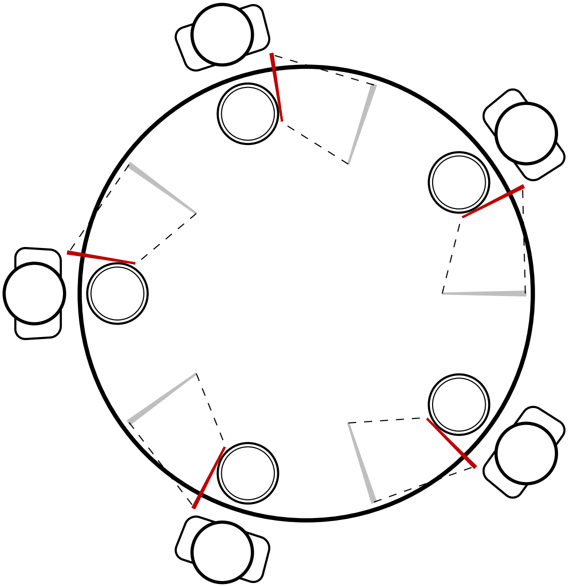
dining philosophers



five philosophers either think or eat

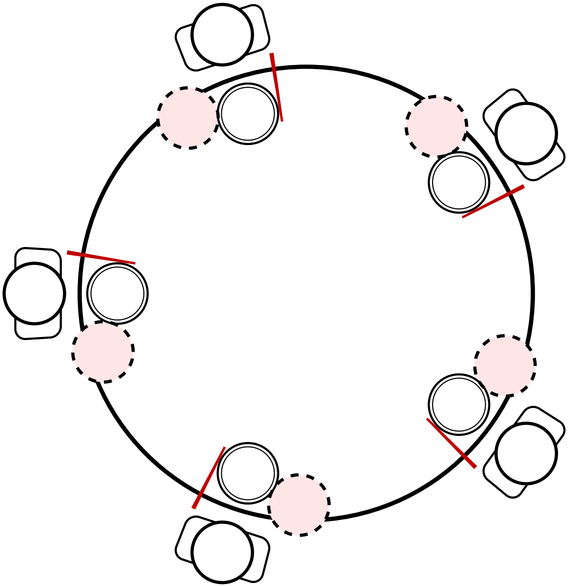
to eat, grab chopsticks on either side

dining philosophers



everyone eats at the same time?
grab left chopstick, then...

dining philosophers



everyone eats at the same time?
grab left chopstick, then
try to grab right chopstick, …
we're at an impasse

skipping the guard page

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

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

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)

```
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;
}
```

iterate through list of kernel-space mappings for everything above address `0x8000 0000` (hard-coded table including flag bits, etc. because some addresses need different flags and not all physical addresses are usable)

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;
}
```

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;
```

reading executables (headers)

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

sz — top of heap of new program
name of the field in `struct proc`

```
struct proghdr {
  uint type;                                   */
  uint off;
  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, 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, ..., 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;
}
```

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

loading user pages from executable

get physical address from page table entry

convert back to (kernel) virtual address

for read from disk

```
loaduvm(pde_t *pgdir, ch                                         , 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                                                            , uin
{
  ...
  for(i = 0; i < sz;
    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;
}
```

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;
}
```