last time

things compilers sometimes don't do well

- space/time tradeoffs
- predicting values
- cross-file/method opitmizations

redundant operations in loops

function inlining

- copy function body to where it's used
- avoid running instructions to call/ret/move arguments
- function used a lot? lots of extra code

vector instructions

- AKA SIMD (single instruction, multiple data)
- registers holding vector (fixed-size array)
- instructions that act on all pairs between two vectors
- hardware support: basically duplicated ALUs

128-bit version, too

history: 256-bit vectors added in extension called AVX (c. 2011)

before: 128-bit vectors added in extension called SSE (c. 1999)

128-bit intrinsics exist, too:

`__m256i` becomes `__m128i`
`_mm256_add_epi32` becomes `_mm_add_epi32`
`_mm256_loadu_si256` becomes `_mm_loadu_si128`

matrix multiply

```
void matmul(unsigned int *A, unsigned int *B, unsigned int *C)
    for (int k = 0; k < N; ++k)
        for (int i = 0; i < N; ++i)
            for (int j = 0; j < N; ++j)
                C[i * N + j] += A[i * N + k] * B[k * N + j];
}
```

(simple version, no cache blocking, no avoiding aliasing beteeen C, B, A,...)

matmul unrolled

```
void matmul(unsigned int *A, unsigned int *B, unsigned int *C) {
  for (int k = 0; k < N; ++k) {
    for (int i = 0; i < N; ++i)
      for (int j = 0; j < N; j += 8) {
        /* goal: vectorize this */
        C[i * N + j + 0] += A[i * N + k] * B[k * N + j + 0];
        C[i * N + j + 1] += A[i * N + k] * B[k * N + j + 1];
        C[i * N + j + 2] += A[i * N + k] * B[k * N + j + 2];
        C[i * N + j + 3] += A[i * N + k] * B[k * N + j + 3];
        C[i * N + j + 4] += A[i * N + k] * B[k * N + j + 4];
        C[i * N + j + 5] += A[i * N + k] * B[k * N + j + 5];
        C[i * N + j + 6] += A[i * N + k] * B[k * N + j + 6];
        C[i * N + j + 7] += A[i * N + k] * B[k * N + j + 7];
      }
}
```

(NB: would probably also want to do cache blocking...)

handy intrinsic functions for matmul

`_mm256_set1_epi32` — load eight copies of a 32-bit value into a 256-bit value

instructions generated vary; one example: `vmovd` + `vpbroadcastd`

`_mm256_mullo_epi32` — multiply eight pairs of 32-bit values, give lowest 32-bits of results

generates `vpmulld`

vectorizing matmul

```
/* goal: vectorize this */
C[i * N + j + 0] += A[i * N + k] * B[k * N + j + 0];
C[i * N + j + 1] += A[i * N + k] * B[k * N + j + 1];
...
C[i * N + j + 6] += A[i * N + k] * B[k * N + j + 6];
C[i * N + j + 7] += A[i * N + k] * B[k * N + j + 7];
```

---

vectorizing matmul

```
/* goal: vectorize this */
C[i * N + j + 0] += A[i * N + k] * B[k * N + j + 0];
C[i * N + j + 1] += A[i * N + k] * B[k * N + j + 1];
...
C[i * N + j + 6] += A[i * N + k] * B[k * N + j + 6];
C[i * N + j + 7] += A[i * N + k] * B[k * N + j + 7];
```

```
// load eight elements from C
Cij = _mm256_loadu_si256((__m256i*) &C[i * N + j + 0]);
... // manipulate vector here
// store eight elements into C
_mm_storeu_si256((__m256i*) &C[i * N + j + 0], Cij);
```

vectorizing matmul

```
/* goal: vectorize this */
C[i * N + j + 0] += A[i * N + k] * B[k * N + j + 0];
C[i * N + j + 1] += A[i * N + k] * B[k * N + j + 1];
...
C[i * N + j + 6] += A[i * N + k] * B[k * N + j + 6];
C[i * N + j + 7] += A[i * N + k] * B[k * N + j + 7];
```

```
// load eight elements from B
Bkj = _mm256_loadu_si256((__m256i*) &B[k * N + j + 0]);
... // multiply each by B[i * N + k] here
```

vectorizing matmul

```
/* goal: vectorize this */
C[i * N + j + 0] += A[i * N + k] * B[k * N + j + 0];
C[i * N + j + 1] += A[i * N + k] * B[k * N + j + 1];
...
C[i * N + j + 6] += A[i * N + k] * B[k * N + j + 6];
C[i * N + j + 7] += A[i * N + k] * B[k * N + j + 7];
```

```
// load eight elements starting with B[k * n + j]
Bkj = _mm256_loadu_si256((__m256i*) &B[k * N + j + 0]);
// load eight copies of A[i * N + k]
Aik = _mm256_set1_epi32(A[i * N + k]);
// multiply each pair
multiply_results = _mm256_mullo_epi32(Aik, Bkj);
```

vectorizing matmul

```
/* goal: vectorize this */
C[i * N + j + 0] += A[i * N + k] * B[k * N + j + 0];
C[i * N + j + 1] += A[i * N + k] * B[k * N + j + 1];
...
C[i * N + j + 6] += A[i * N + k] * B[k * N + j + 6];
C[i * N + j + 7] += A[i * N + k] * B[k * N + j + 7];
```

```
Cij = _mm256_add_epi32(Cij, multiply_results);
// store back results
_mm256_storeu_si256(..., Cij);
```

matmul vectorized

```
__m256i Cij, Bkj, Aik, multiply_results;

// Cij = {C_{i,j}, C_{i,j+1}, C_{i,j+2}, ..., C_{i,j+7}}
Cij = _mm256_loadu_si256((__m256i*) &C[i * N + j]);
// Bkj = {B_{k,j}, B_{k,j+1}, B_{k,j+2}, ..., B_{k,j+7}}
Bkj = _mm256_loadu_si256((__m256i*) &B[k * N + j]);

// Aik = {A_{i,k}, A_{i,k}, ..., A_{i,k}}
Aik = _mm256_set1_epi32(A[i * N + k]);

// Aik_times_Bkj = {A_{i,k} × B_{k,j}, A_{i,k} × B_{k,j+1}, A_{i,k} × B_{k,j+2}, ..., A_{i,k} × B_{k,j+7}}
multiply_results = _mm256_mullo_epi32(Aij, Bkj);

// Cij= {C_{i,j} + A_{i,k} × B_{k,j}, C_{i,j+1} + A_{i,k} × B_{k,j+1}, ...}
Cij = _mm256_add_epi32(Cij, multiply_results);

// store Cij into C
_mm256_storeu_si256((__m256i*) &C[i * N + j], Cij);
```

vector exercise (2a)

```
long A[1024], B[1024];
...
for (int i = 0; i < N; i += 1)
    for (int j = 0; j < N; j += 1)
        A[i] += B[i] * B[j];
```

(casts omitted below to reduce clutter:)

```
for (int i = 0; i < 1024; i += 4) {
    A_part = _mm256_loadu_si256(&A[i]);
    Bi_part = _mm256_loadu_si256(&B[i]);
    for (int j = 0; j < 1024; /* BLANK 1 */) {
        Bj_part = _mm256_/* BLANK 2 */;
        A_part = _mm256_add_epi64(A_part, _mm256_mullo_epi64(Bi_par
    }
    _mm256_storeu_si256(&A[i], A_part);
}
```

What goes in BLANK 1 and BLANK 2?

A. `loadu_si256(&B[j])`, j += 1 B. `loadu_si256(&B[j])`, j += 4

C. `set1_epi64(B[j])`, j += 1 D. `set1_epi64(B[j])`, j += 4

moving values in vectors?

sometimes values aren't in the right place in vector

example:

have: [1, 2, 3, 4]

want: [3, 4, 1, 2]

there are instructions/intrinsics for doing this

called shuffling/swizzling/permute/...

sometimes might need combination of them

worst-case: could rearrange on stack..., I guess

example shuffling operation (1)

goal: [1, 2, 3, 4] to [3, 4, 1, 2] (64-bit values)

```
/* x = {1, 2, 3, 4} */
__m256i x = _mm256_setr_epi64x(1, 2, 3, 4);
__m256i result = _mm256_permute4x64_epi64(
        x,
        /* index 2, then 3, then 0, then 1 */
        2 | (3 << 2) | (0 << 4) | (1 << 6)
        /* could also write _MM_SHUFFLE(1, 0, 3, 2) */
    );
/* result = {3, 4, 1, 2} */
```

other vector instructions

multiple extensions to the X86 instruction set for vector instructions

early versions (128-bit vectors): SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2

- 128-bit vectors

this class (256-bit): AVX, AVX2

not this class (512+-bit): AVX-512

- 512-bit vectors

also other ISAs have these: e.g. NEON on ARM, MSA on MIPS, AltiVec/VMX on POWER, …

GPUs are essentially vector-instruction-specialized CPUs

other vector interfaces

intrinsics (our assignments) one way

some alternate programming interfaces

- have compiler do more work than intrinsics

e.g. CUDA, OpenCL, GCC's vector instructions

other vector instructions features

more flexible vector instruction features:

- invented in the 1990s
- often present in GPUs and being rediscovered by modern ISAs

reasonable conditional handling

better variable-length vectors

ability to load/store non-contiguous values

some of these features in AVX2/AVX512

alternate vector interfaces

intrinsics functions/assembly aren't the only way to write vector code

e.g. GCC vector extensions: more like normal C code

- types for each kind of vector
- write + instead of `_mm_add_epi32`

e.g. CUDA (GPUs): looks like writing multithreaded code, but each thread is vector "lane"

optimizing real programs

ask your compiler to try first

spend effort where it matters

e.g. 90% of program time spent reading files, but optimize computation?

e.g. 90% of program time spent in routine A, but optimize B?

profilers

first step — tool to determine where you spend time

tools exist to do this for programs

example on Linux: `perf`

example

```
Samples: 37K of event 'cycles', Event count (approx.): 37367555513
  Children      Self  Command          Shared Object          Symbol
+  100.00%     0.00%  hclrs-with-debu  hclrs-with-debuginfo  [.] _start
+  100.00%     0.00%  hclrs-with-debu  libc-2.31.so          [.] __libc_start_main
+  100.00%     0.00%  hclrs-with-debu  hclrs-with-debuginfo  [.] main
+  100.00%     0.00%  hclrs-with-debu  hclrs-with-debuginfo  [.] std::sys_common::backtrace::__rust_begin_short_backt
+  100.00%     0.00%  hclrs-with-debu  hclrs-with-debuginfo  [.] hclrs::main
+   99.99%     9.75%  hclrs-with-debu  hclrs-with-debuginfo  [.] hclrs::program::RunningProgram::run
+   60.37%    31.67%  hclrs-with-debu  hclrs-with-debuginfo  [.] hclrs::ast::SpannedExpr::evaluate
+   41.34%    23.29%  hclrs-with-debu  hclrs-with-debuginfo  [.] hashbrown::map::make_hash
+   18.08%    18.07%  hclrs-with-debu  hclrs-with-debuginfo  [.] <std::collections::hash::map::DefaultHasher as core:
+   16.33%     0.68%  hclrs-with-debu  hclrs-with-debuginfo  [.] hclrs::program::Program::process_register_banks
+    9.54%     3.15%  hclrs-with-debu  hclrs-with-debuginfo  [.] std::collections::hash::map::HashMap<K,V,S>::get
+    9.10%     9.09%  hclrs-with-debu  libc-2.31.so          [.] __memcmp_avx2_movbe
+    6.11%     2.10%  hclrs-with-debu  hclrs-with-debuginfo  [.] hashbrown::map::HashMap<K,V,S>::get_mut
+    2.32%     0.88%  hclrs-with-debu  hclrs-with-debuginfo  [.] std::collections::hash::map::HashMap<K,V,S>::get
+    1.45%     0.52%  hclrs-with-debu  hclrs-with-debuginfo  [.] hashbrown::map::HashMap<K,V,S>::insert
     0.37%     0.11%  hclrs-with-debu  hclrs-with-debuginfo  [.] <alloc::string::String as core::clone::Clone>::clone
     0.19%     0.19%  hclrs-with-debu  libc-2.31.so          [.] malloc
```

an infinite loop

```
int main(void) {
    while (1) {
        /* waste CPU time */
    }
}
```

If I run this on a shared department machine, can you still use it?

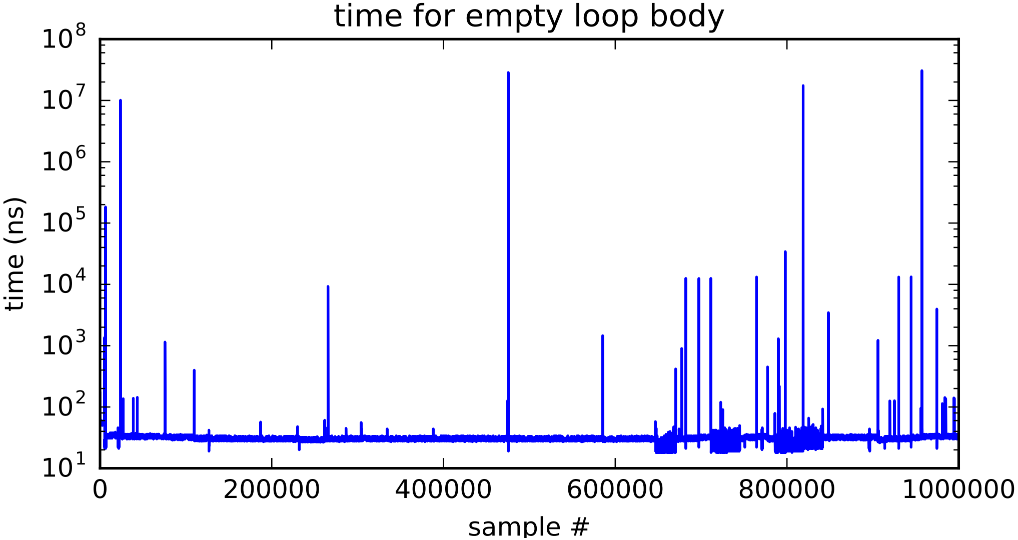
...if the machine only has one core?

timing nothing

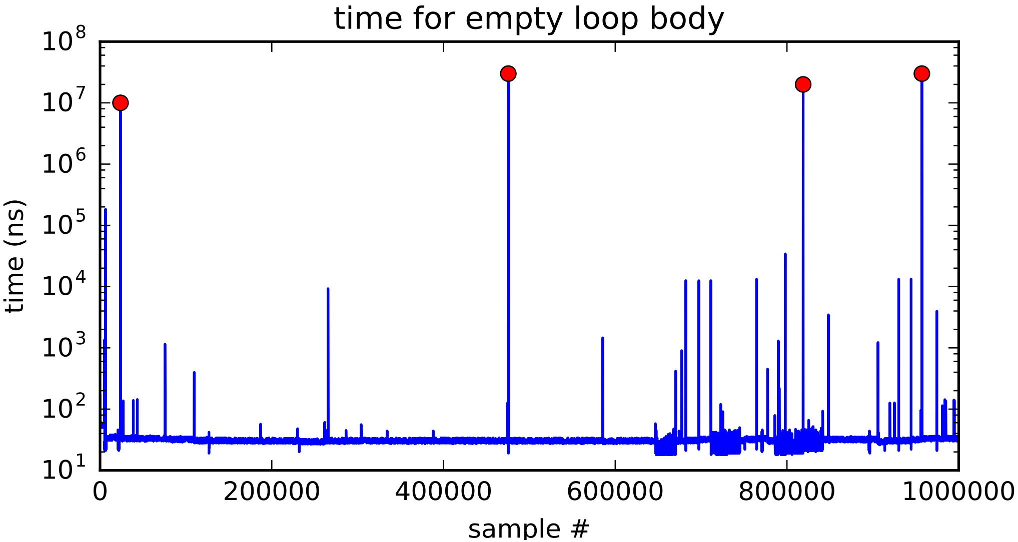
```
long times[NUM_TIMINGS];
int main(void) {
    for (int i = 0; i < N; ++i) {
        long start, end;
        start = get_time();
        /* do nothing */
        end = get_time();
        times[i] = end - start;
    }
    output_timings(times);
}
```

same instructions — same difference each time?

doing nothing on a busy system



doing nothing on a busy system



time multiplexing

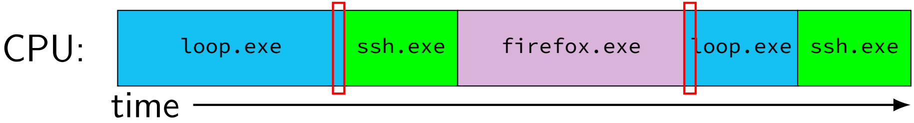
time multiplexing


```
...
call get_time
    // whatever get_time does
movq %rax, %rbp
```

million cycle delay

```
call get_time
    // whatever get_time does
subq %rbp, %rax
...
```

time multiplexing

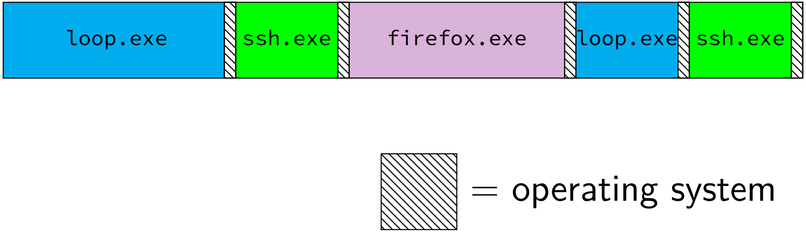



```
...
call get_time
    // whatever get_time does
movq %rax, %rbp
```

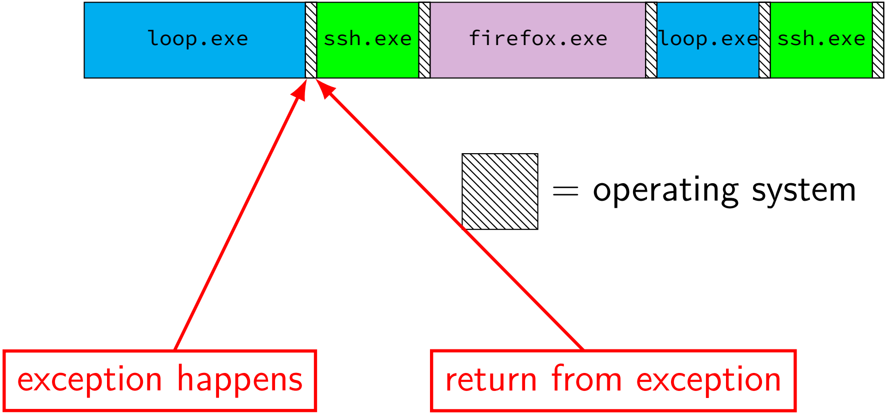
million cycle delay

```
call get_time
    // whatever get_time does
subq %rbp, %rax
...
```

time multiplexing really



time multiplexing really



OS and time multiplexing

starts running instead of normal program

- mechanism for this: exceptions (later)

saves old program counter, registers somewhere

sets new registers, jumps to new program counter

called context switch

- saved information called context

context

all registers values

`%rax` `%rbx`, ..., `%rsp`, ...

condition codes

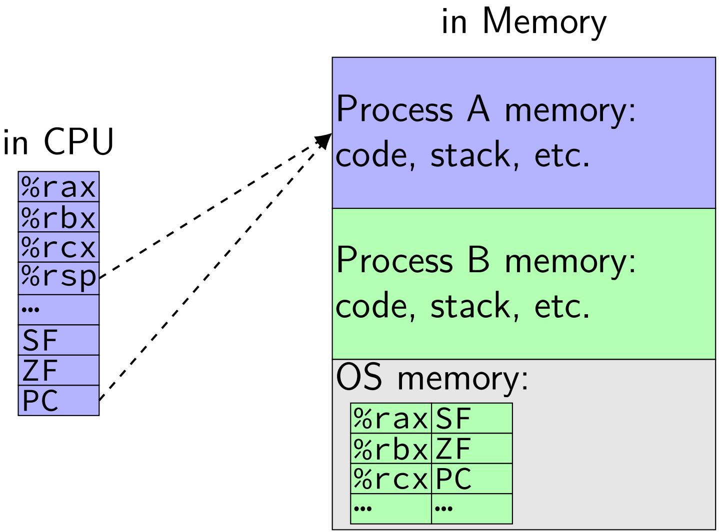
program counter

i.e. all visible state in your CPU except memory

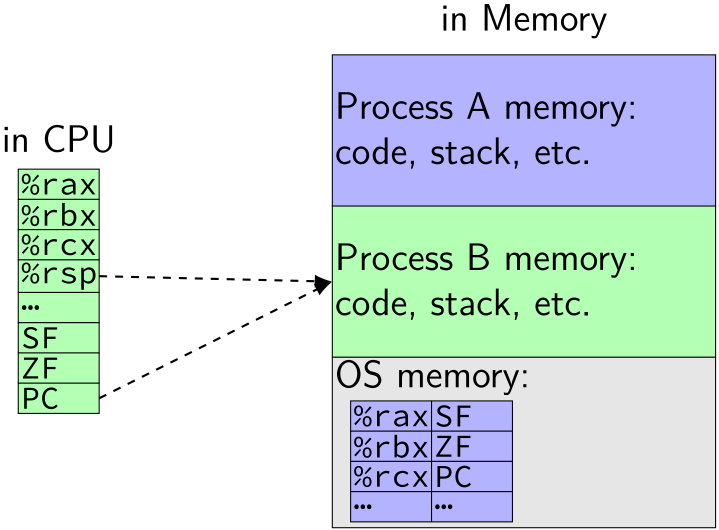
context switch pseudocode

```
context_switch(last, next):
  copy_preexception_pc last->pc
  mov rax,last->rax
  mov rcx, last->rcx
  mov rdx, last->rdx
  ...
  mov next->rdx, rdx
  mov next->rcx, rcx
  mov next->rax, rax
  jmp next->pc
```

contexts (A running)



contexts (B running)



memory protection

reading from another program's memory?

Program A	Program B

`0x10000: .word 42`
`    // ...`
`    // do work`
`    // ...`
`    movq 0x10000, %rax`
	`// while A is working:`
	`movq $99, %rax`
	`movq %rax, 0x10000`
	`...`

memory protection

reading from another program's memory?

Program A	Program B

`0x10000: .word 42`
`    // ...`
`    // do work`
`    // ...`
`    movq 0x10000, %rax`
	`// while A is working:`
	`movq $99, %rax`
	`movq %rax, 0x10000`
	`...`

result: `%rax` is ...	result: `%rax` is ...

A. 42 B. 99 C. 0x10000

D. 42 or 99 (depending on timing/program layout/etc)

E. 42 or program might crash (depending on ...)

F. 99 or program might crash (depending on ...)

G. 42 or 99 or program might crash (depending on ...)

H. something else

memory protection

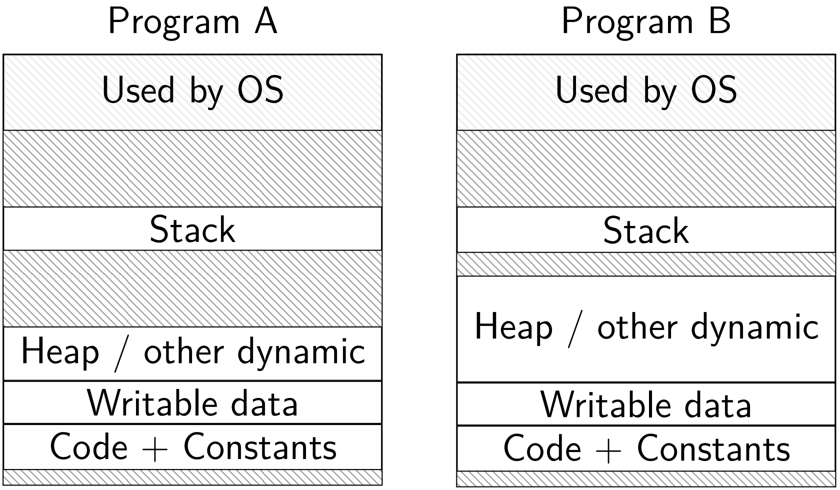
reading from another program's memory?

Program A	Program B

`0x10000: .word 42`
`    // ...`
`    // do work`
`    // ...`
`    movq 0x10000, %rax`
	`// while A is working:`
	`movq $99, %rax`
	`movq %rax, 0x10000`
	`...`

result: `%rax` is 42 (always)	result: might crash

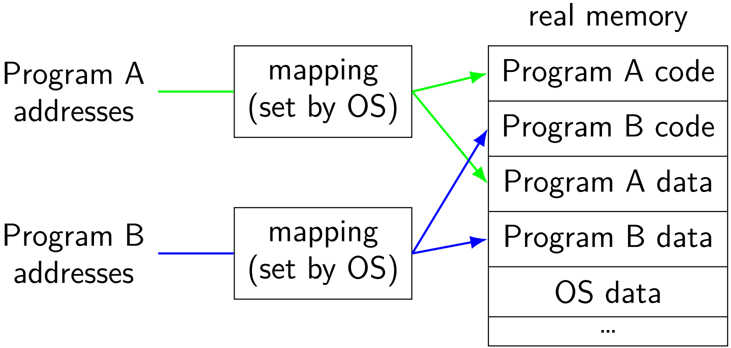
program memory (two programs)



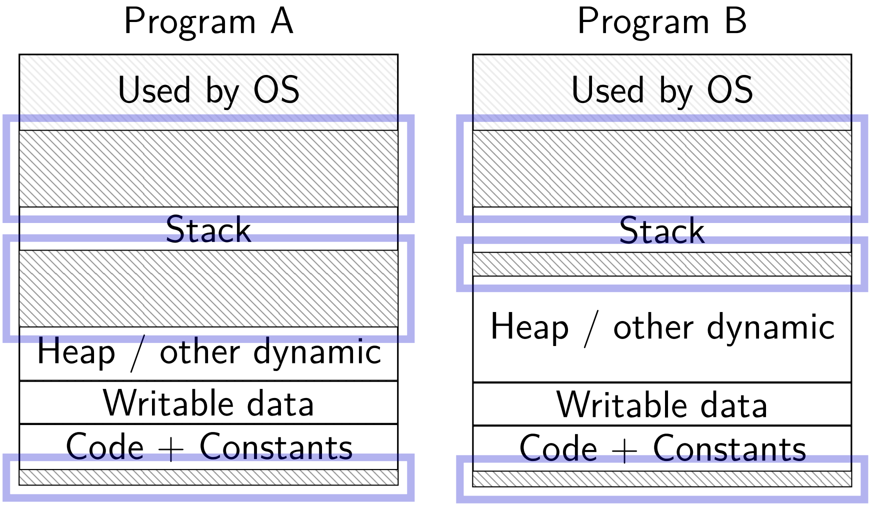
address space

programs have illusion of own memory

called a program's address space



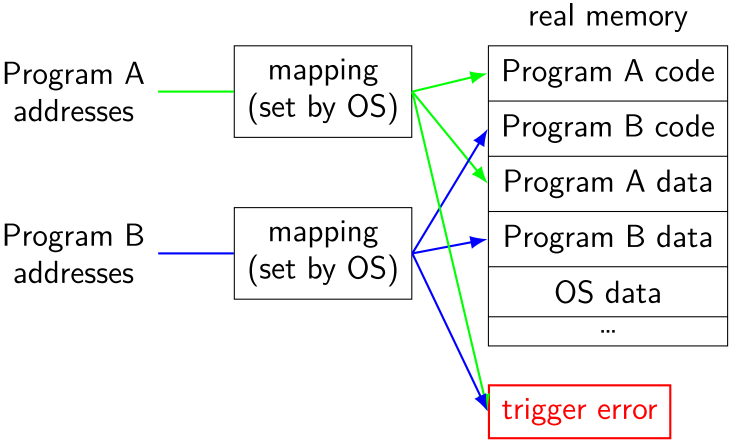
program memory (two programs)



address space

programs have illusion of own memory

called a program's address space



address space mechanisms

topic after exceptions

called virtual memory

mapping called page tables

mapping part of what is changed in context switch

context

all registers values

`%rax` `%rbx`, ..., `%rsp`, ...

condition codes

program counter

~~i.e. all visible state in your CPU except memory~~

address space: map from program to real addresses

The Process

process = thread(s) + address space

illusion of dedicated machine:

- thread = illusion of own CPU
- address space = illusion of own memory

types of exceptions

interrupts — externally-triggered

- timer — keep program from hogging CPU
- I/O devices — key presses, hard drives, networks, …

aborts — hardware is broken

asynchronous — not triggered by running program

traps — intentionally triggered exceptions

- system calls — ask OS to do something

faults — errors/events in programs

- memory not in address space ("Segmentation fault")
- privileged instruction
- divide by zero
- invalid instruction

synchronous — triggered by current program

types of exceptions

interrupts — externally-triggered

- timer — keep program from hogging CPU
- I/O devices — key presses, hard drives, networks, …

aborts — hardware is broken

asynchronous: not triggered by running program

traps — intentionally triggered exceptions

- system calls — ask OS to do something

faults — errors/events in programs

- memory not in address space ("Segmentation fault")
- privileged instruction
- divide by zero
- invalid instruction

synchronous: triggered by current program

types of exceptions

interrupts — externally-triggered

- timer — keep program from hogging CPU
- I/O devices — key presses, hard drives, networks, …

aborts — hardware is broken

} **asynchronous** — not triggered by running program

traps — intentionally triggered exceptions

- system calls — ask OS to do something

faults — errors/events in programs

- memory not in address space ("Segmentation fault")
- privileged instruction
- divide by zero
- invalid instruction

} **synchronous** — triggered by current program

timer interrupt

(conceptually) external timer device

(usually on same chip as processor)

OS configures before starting program

sends signal to CPU after a fixed interval

types of exceptions

interrupts — externally-triggered

- timer — keep program from hogging CPU
- I/O devices — key presses, hard drives, networks, ...

aborts — hardware is broken

asynchronous: not triggered by running program

traps — intentionally triggered exceptions

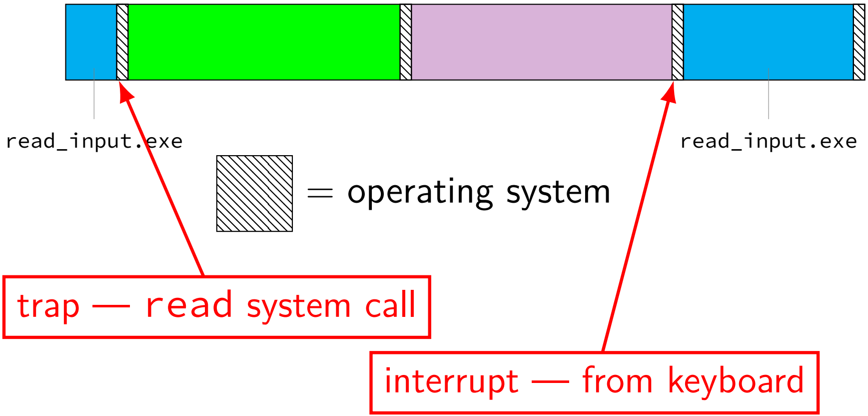
- system calls — ask OS to do something

faults — errors/events in programs

- memory not in address space ("Segmentation fault")
- privileged instruction
- divide by zero
- invalid instruction

synchronous: triggered by current program

keyboard input timeline



types of exceptions

interrupts — externally-triggered

- timer — keep program from hogging CPU
- I/O devices — key presses, hard drives, networks, ...

aborts — hardware is broken

} **asynchronous** not triggered by running program

traps — intentionally triggered exceptions

- system calls — ask OS to do something

faults — errors/events in programs

- memory not in address space ("Segmentation fault")
- privileged instruction
- divide by zero
- invalid instruction

} **synchronous** triggered by current program

exception implementation

detect condition (program error or external event)

save current value of PC somewhere

jump to exception handler (part of OS)

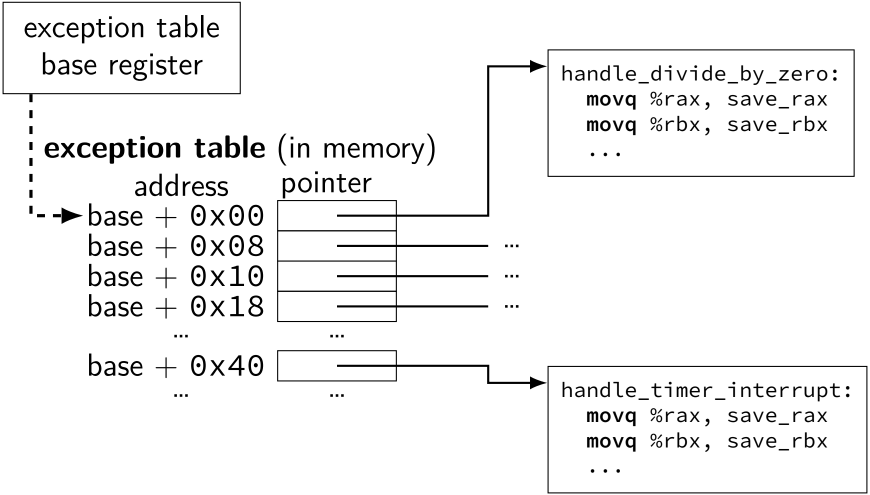
- jump done without program instruction to do so

exception implementation: notes

I/textbook describe a simplified version

real x86/x86-64 is a bit more complicated
(mostly for historical reasons)

locating exception handlers



running the exception handler

hardware saves the old program counter (and maybe more)

identifies location of exception handler via table

then jumps to that location

OS code can save anything else it wants to , etc.

added to CPU for exceptions

new instruction: set exception table base

new logic: jump based on exception table

- may need to cancel partially completed instructions before jumping

new logic: save the old PC (and maybe more)

- to special register or to memory

new instruction: return from exception

- i.e. jump to saved PC

added to CPU for exceptions

new instruction: set exception table base

new logic: jump based on exception table

- may need to cancel partially completed instructions before jumping

new logic: save the old PC (and maybe more)

- to special register or to memory

new instruction: return from exception

- i.e. jump to saved PC

added to CPU for exceptions

new instruction: set exception table base

new logic: jump based on exception table

- may need to cancel partially completed instructions before jumping

new logic: save the old PC (and maybe more)

- to special register or to memory

new instruction: return from exception

- i.e. jump to saved PC

added to CPU for exceptions

new instruction: set exception table base

new logic: jump based on exception table

- may need to cancel partially completed instructions before jumping

new logic: save the old PC (and maybe more)

- to special register or to memory

new instruction: return from exception

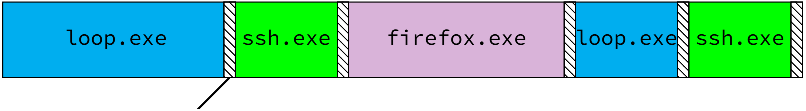
- i.e. jump to saved PC

exception handler structure

1. save process's state somewhere
2. do work to handle exception
3. restore a process's state (maybe a different one)
4. jump back to program

```
handle_timer_interrupt:
  mov_from_saved_pc save_pc_loc
  movq %rax, save_rax_loc
  ... // choose new process to run here
  movq new_rax_loc, %rax
  mov_to_saved_pc new_pc
  return_from_exception
```

exceptions and time slicing



timer interrupt

exception table lookup

```
handle_timer_interrupt:
    ...
    ...
    set_address_space ssh_address_space
    mov_to_saved_pc saved_ssh_pc
    return_from_exception
```

defeating time slices?

```
my_exception_table:
    ...
my_handle_timer_interrupt:
    // HA! Keep running me!
    return_from_exception

main:
    set_exception_table_base my_exception_table
loop:
    jmp loop
```

defeating time slices?

wrote a program that tries to set the exception table:

```
my_exception_table:
    ...

main:
    // "Load Interrupt
    //  Descriptor Table"
    // x86 instruction to set exception table
    lidt my_exception_table
    ret
```

result: `Segmentation fault` (exception!)

types of exceptions

interrupts — externally-triggered

- timer — keep program from hogging CPU
- I/O devices — key presses, hard drives, networks, …

aborts — hardware is broken

} **asynchronous** — not triggered by running program

traps — intentionally triggered exceptions

- system calls — ask OS to do something

faults — errors/events in programs

- memory not in address space ("Segmentation fault")
- privileged instruction
- divide by zero
- invalid instruction

} **synchronous** — triggered by current program

privileged instructions

can't let any program run some instructions

allows machines to be shared between users (e.g. lab servers)

examples:

- set exception table
- set address space
- talk to I/O device (hard drive, keyboard, display, …)
- …

processor has two modes:

- kernel mode — privileged instructions work
- user mode — privileged instructions cause exception instead

kernel mode

extra one-bit register: “are we in kernel mode”

exceptions enter kernel mode

return from exception instruction leaves kernel mode

types of exceptions

interrupts — externally-triggered

- timer — keep program from hogging CPU
- I/O devices — key presses, hard drives, networks, ...

aborts — hardware is broken

asynchronous — not triggered by running program

traps — intentionally triggered exceptions

- system calls — ask OS to do something

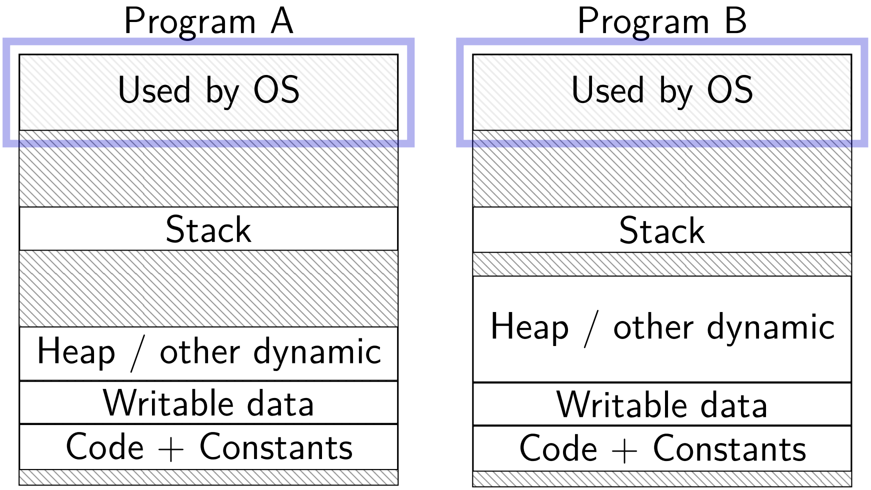
faults — errors/events in programs

- memory not in address space ("Segmentation fault")
- privileged instruction
- divide by zero
- invalid instruction

synchronous — triggered by current program

what about editing exception table?

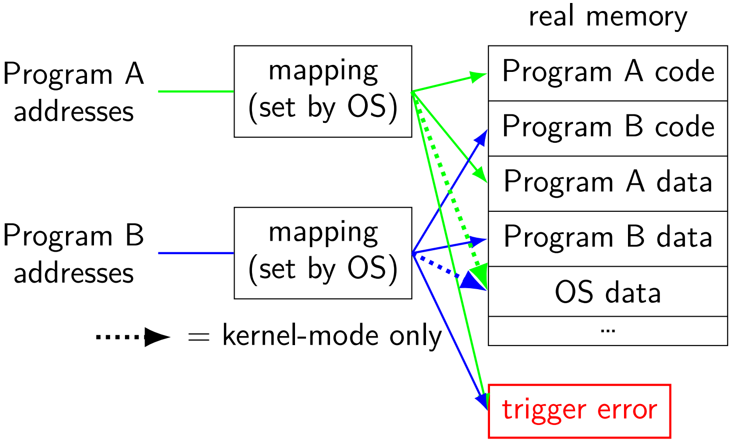
program memory (two programs)



address space

programs have illusion of own memory

called a program's address space



protection fault

when program tries to access memory it doesn't own

e.g. trying to write to bad address

when program tries to do other things that are not allowed

e.g. accessing I/O devices directly

e.g. changing exception table base register

OS gets control — can crash the program

or more interesting things

types of exceptions

interrupts — externally-triggered

- timer — keep program from hogging CPU
- I/O devices — key presses, hard drives, networks, ...

aborts — hardware is broken

asynchronous — not triggered by running program

traps — intentionally triggered exceptions

- system calls — ask OS to do something

faults — errors/events in programs

- memory not in address space ("Segmentation fault")
- privileged instruction
- divide by zero
- invalid instruction

synchronous — triggered by current program

which requires kernel mode?

which operations are likely to fail (trigger an exception to run the OS instead) if attempted in user mode?

A. reading data on disk by running special instructions that communicate with the hard disk device

B. changing a program's address space to allocate it more memory

C. returning from a standard library function

D. incrementing the stack pointer

kernel services

allocating memory? (change address space)

reading/writing to file? (communicate with hard drive)

read input? (communicate with keyborad)

all need privileged instructions!

need to run code in kernel mode

Linux x86-64 system calls

special instruction: `syscall`

triggers trap (deliberate exception)

Linux syscall calling convention

before `syscall`:

`%rax` — system call number

`%rdi`, `%rsi`, `%rdx`, `%r10`, `%r8`, `%r9` — args

after `syscall`:

`%rax` — return value

on error: `%rax` contains -1 times "error number"

almost the same as normal function calls

Linux x86-64 hello world

```
.globl _start
.data
hello_str: .asciz "Hello, World!\n"
.text
_start:
  movq $1, %rax # 1 = "write"
  movq $1, %rdi # file descriptor 1 = stdout
  movq $hello_str, %rsi
  movq $15, %rdx # 15 = strlen("Hello, World!\n")
  syscall

  movq $60, %rax # 60 = exit
  movq $0, %rdi
  syscall
```

approx. system call handler

```
sys_call_table:
    .quad handle_read_syscall
    .quad handle_write_syscall
    // ...

handle_syscall:
    ... // save old PC, etc.
    pushq %rcx // save registers
    pushq %rdi
    ...
    call *sys_call_table(,%rax,8)
    ...
    popq %rdi
    popq %rcx
    return_from_exception
```

Linux system call examples

`mmap`, `brk` — allocate memory

`fork` — create new process

`execve` — run a program in the current process

`_exit` — terminate a process

`open`, `read`, `write` — access files

terminals, etc. count as files, too

system call wrappers

can't write C code to generate syscall instruction

solution: call "wrapper" function written in assembly

which of these require exceptions? context switches?

A. program calls a function in the standard library

B. program writes a file to disk

C. program A goes to sleep, letting program B run

D. program exits

E. program returns from one function to another function

F. program pops a value from the stack

a note on terminology (1)

real world: inconsistent terms for exceptions

we will follow textbook's terms in this course

the real world won't

you might see:

- 'interrupt' meaning what we call 'exception' (x86)
- 'exception' meaning what we call 'fault'
- 'hard fault' meaning what we call 'abort'
- 'trap' meaning what we call 'fault'
- ... and more

a note on terminology (2)

we use the term "kernel mode"

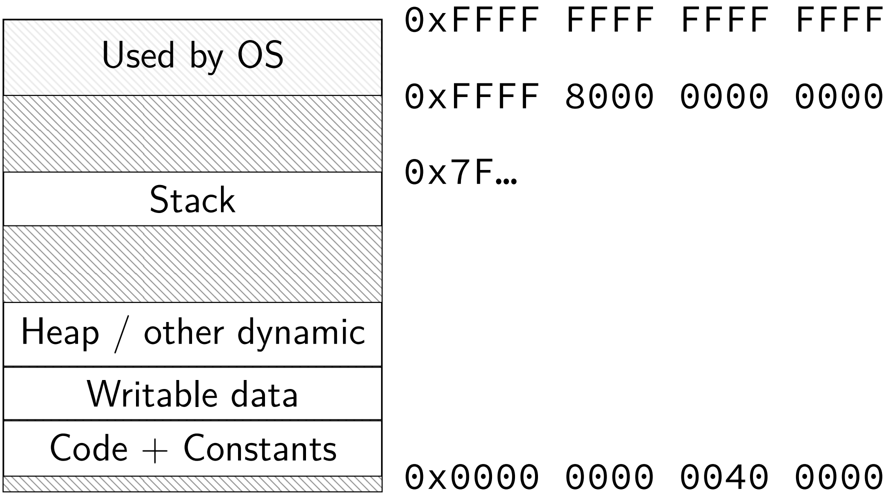
some additional terms:

- supervisor mode
- privileged mode
- ring `0`

some systems have multiple levels of privilege

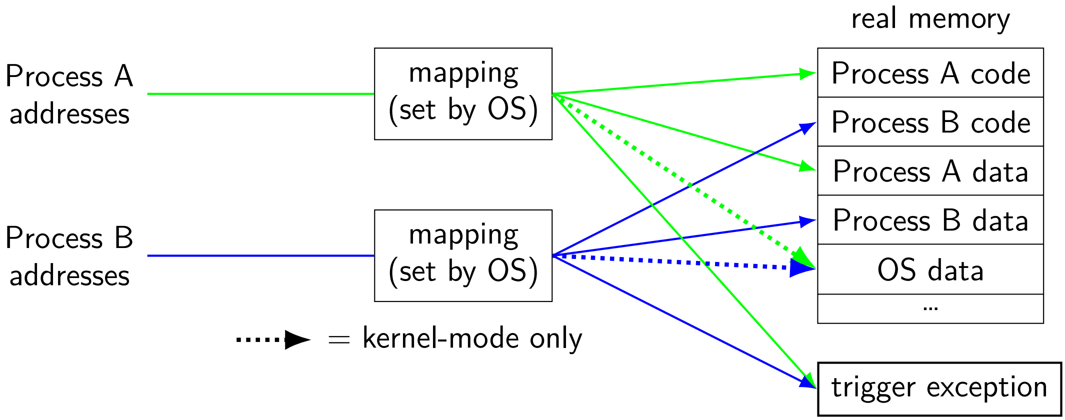
- different sets of priviliged operations work

program memory



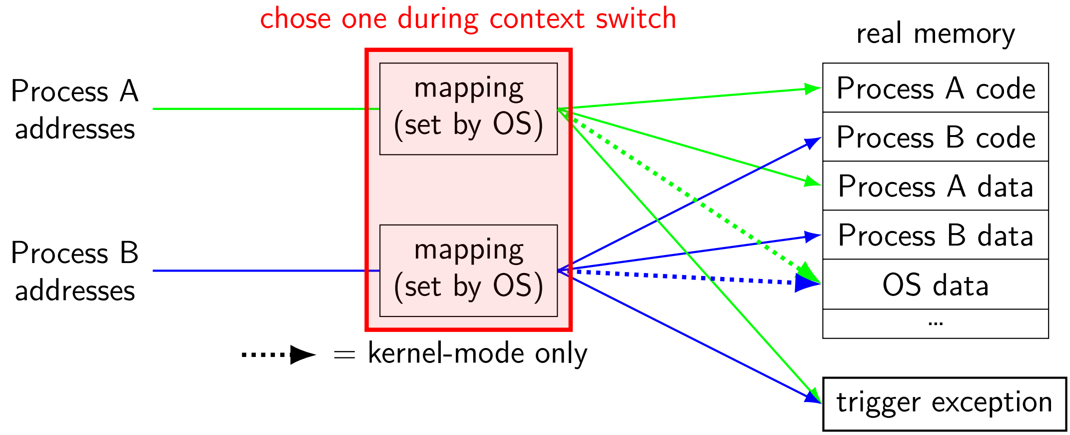
address spaces

illuision of dedicated memory

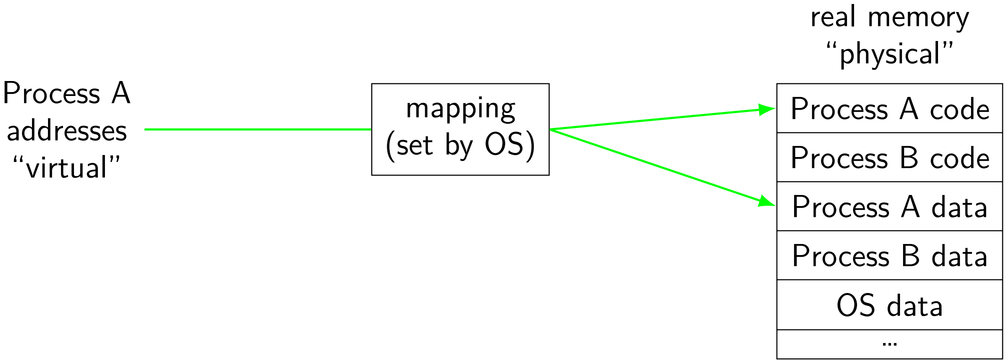


address spaces

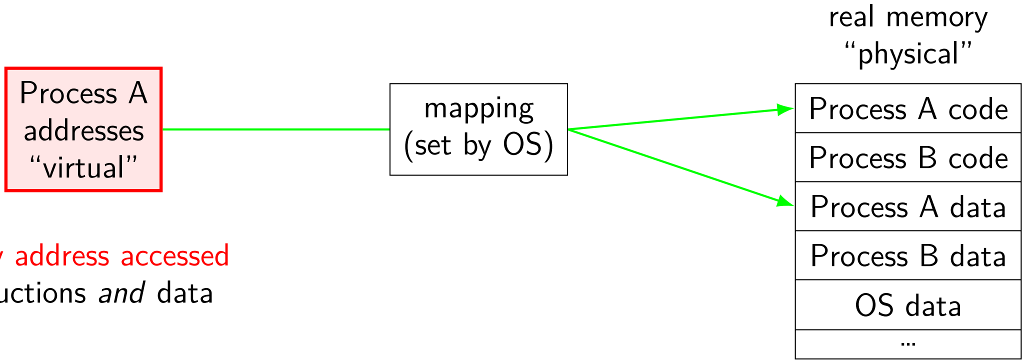
illuision of dedicated memory



address translation

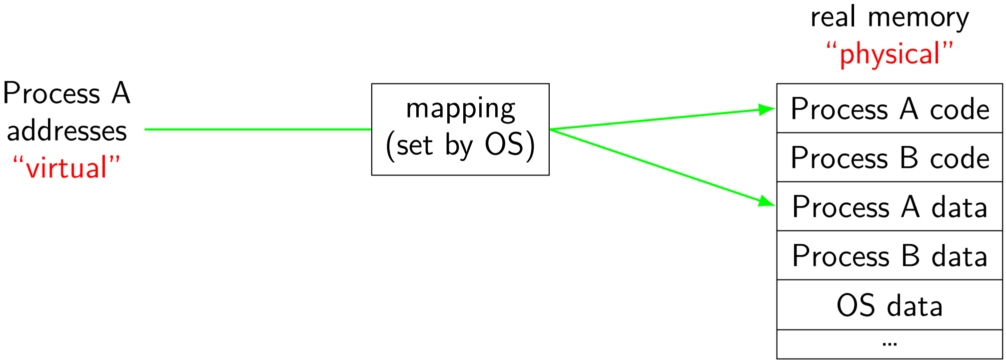


address translation



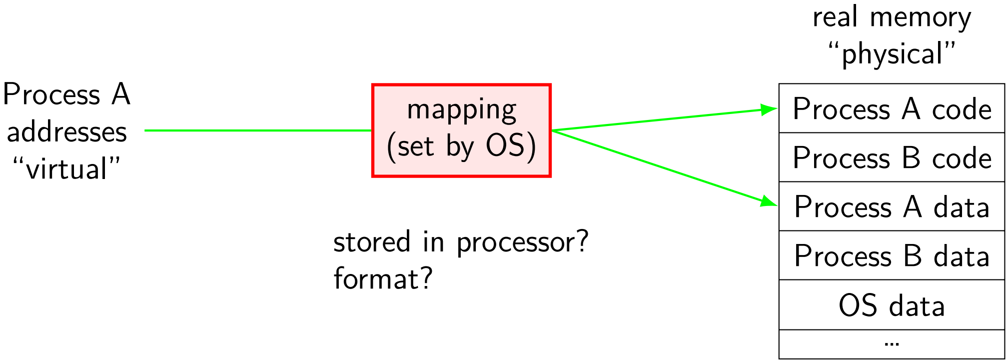
every address accessed
instructions *and* data

address translation



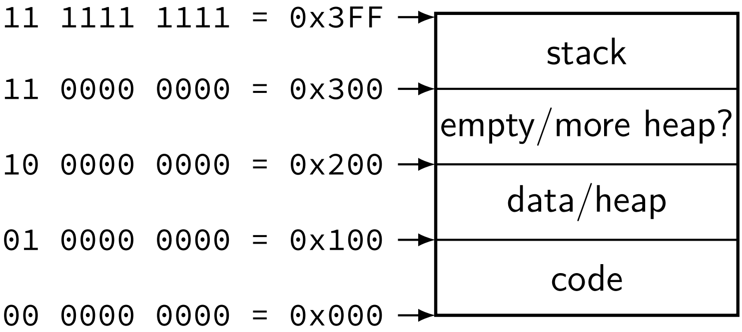
program addresses are ‘virtual’
real addresses are ‘physical’
can be different sizes!

address translation

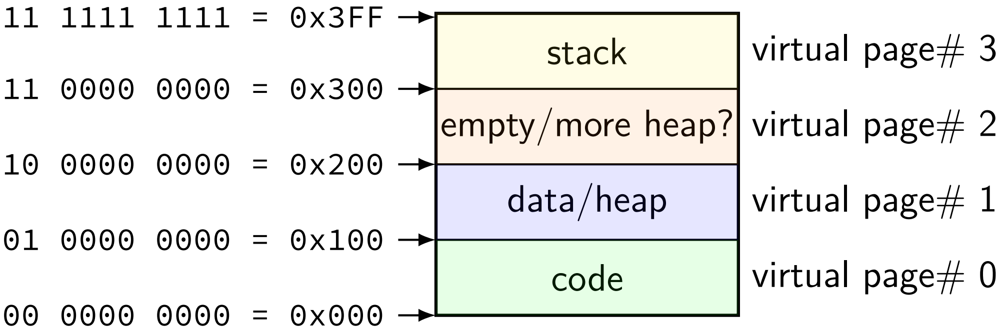


stored in processor?
format?

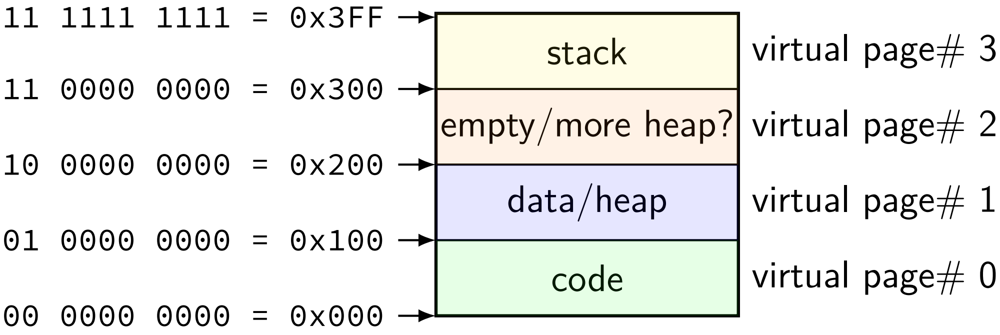
toy program memory



toy program memory

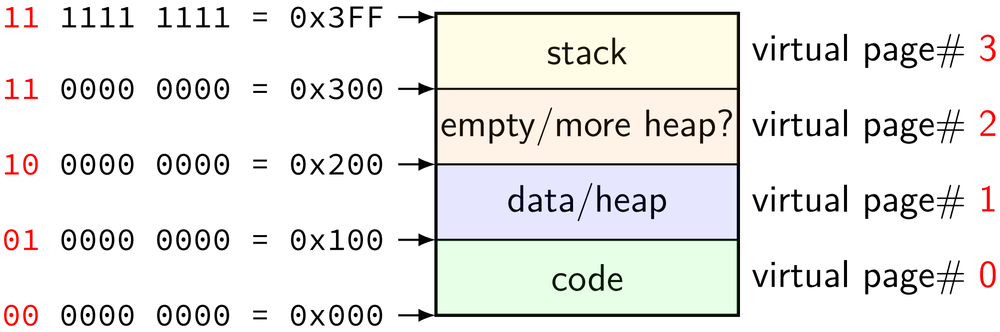


toy program memory



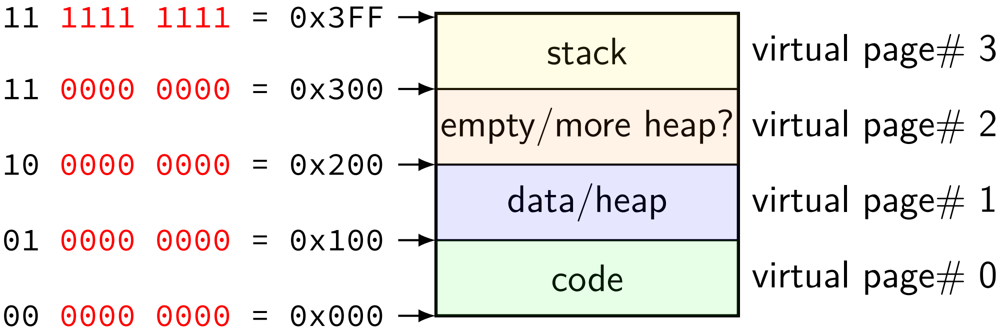
divide memory into pages (2^8 bytes in this case)
"virtual" = addresses the program sees

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

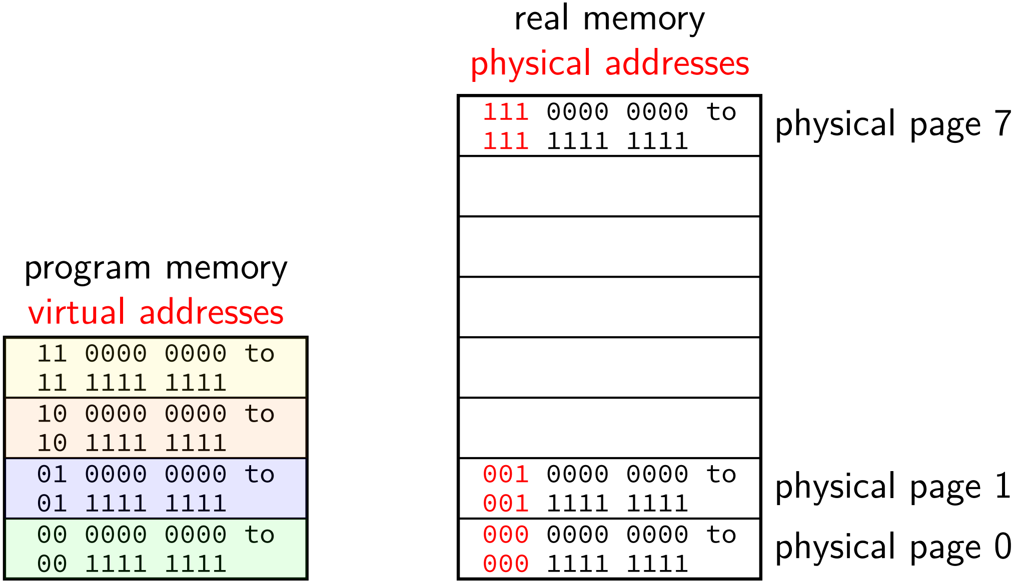
real memory
physical addresses

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

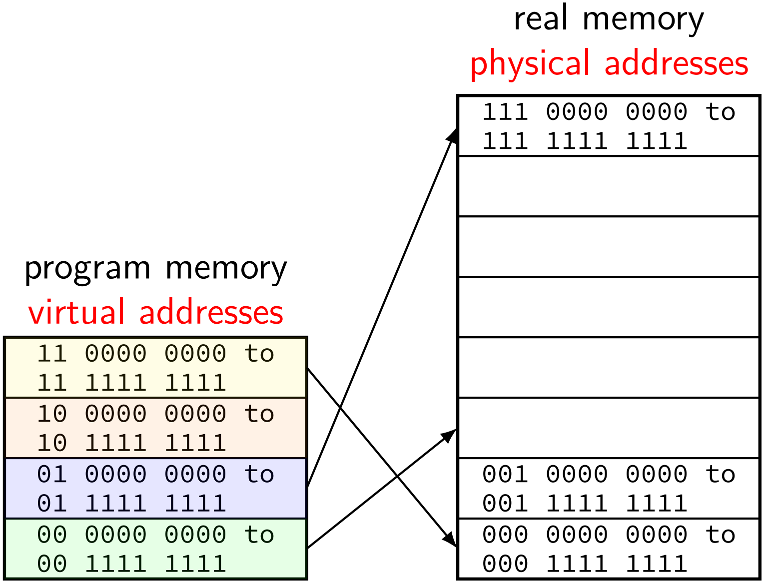
program memory
virtual addresses

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

toy physical memory

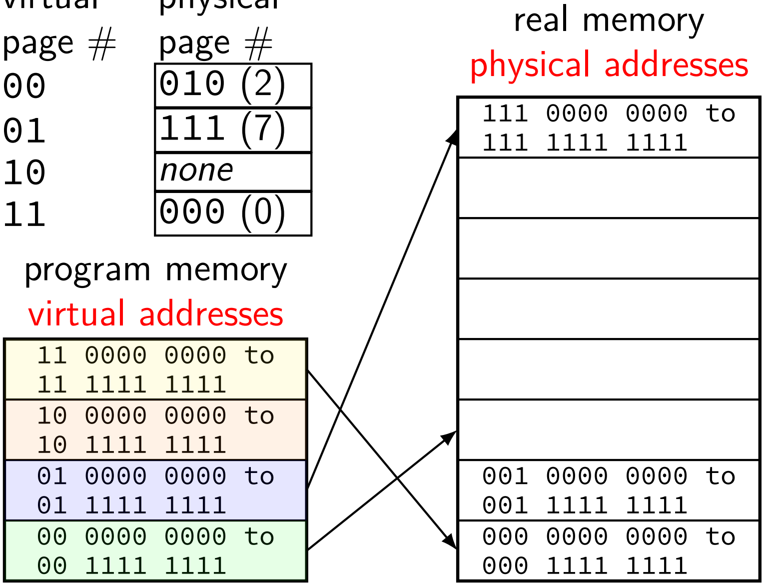


toy physical memory



toy physical memory

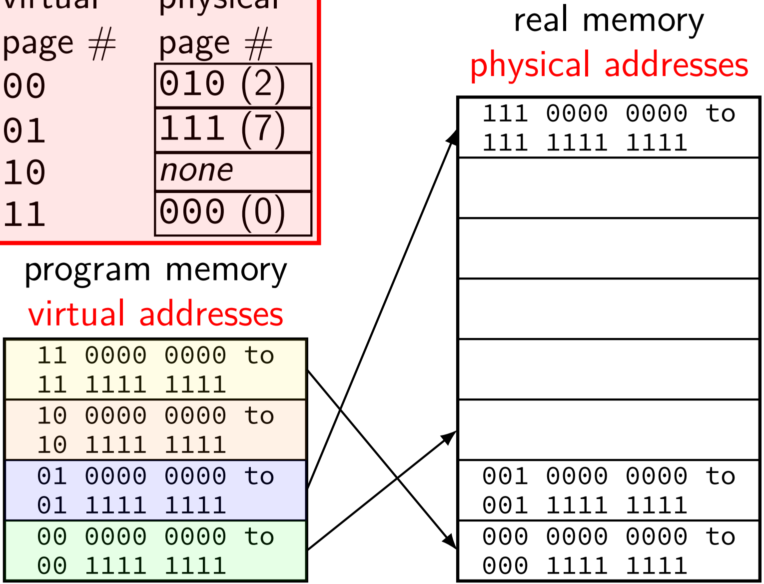
virtual page #	physical page #
00	010 (2)
01	111 (7)
10	*none*
11	000 (0)



toy physical memory

page table!

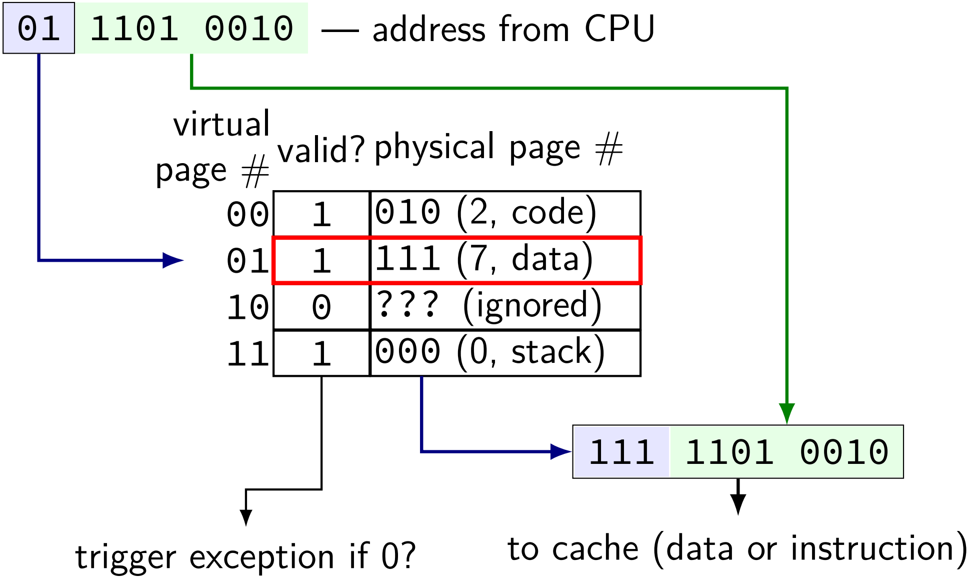
virtual page #	physical page #
00	010 (2)
01	111 (7)
10	*none*
11	000 (0)



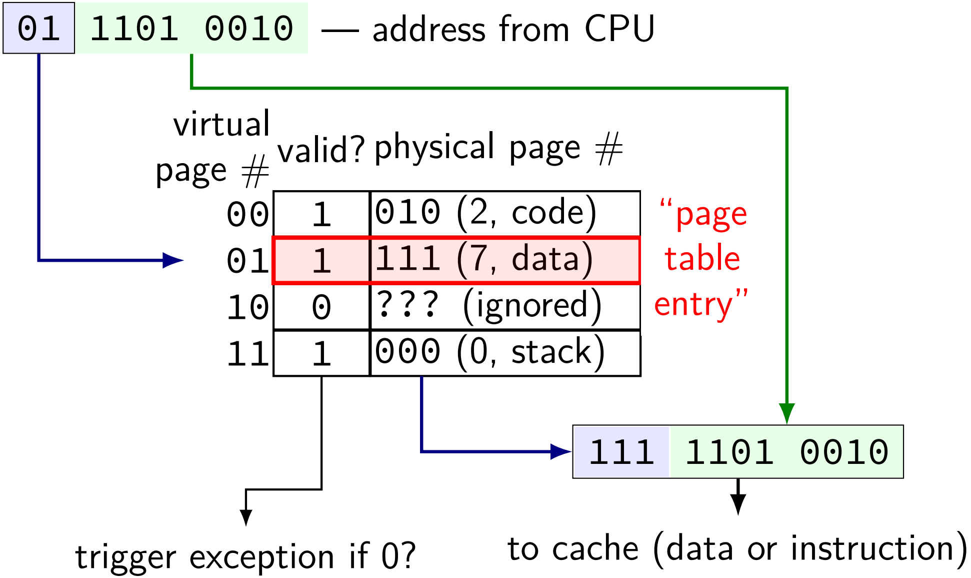
toy page table lookup

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

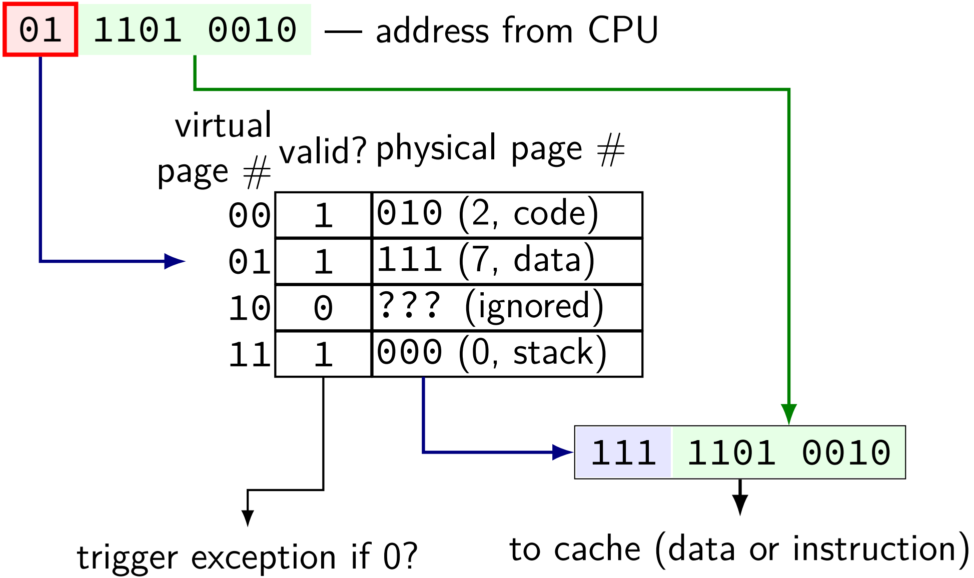
toy page table lookup



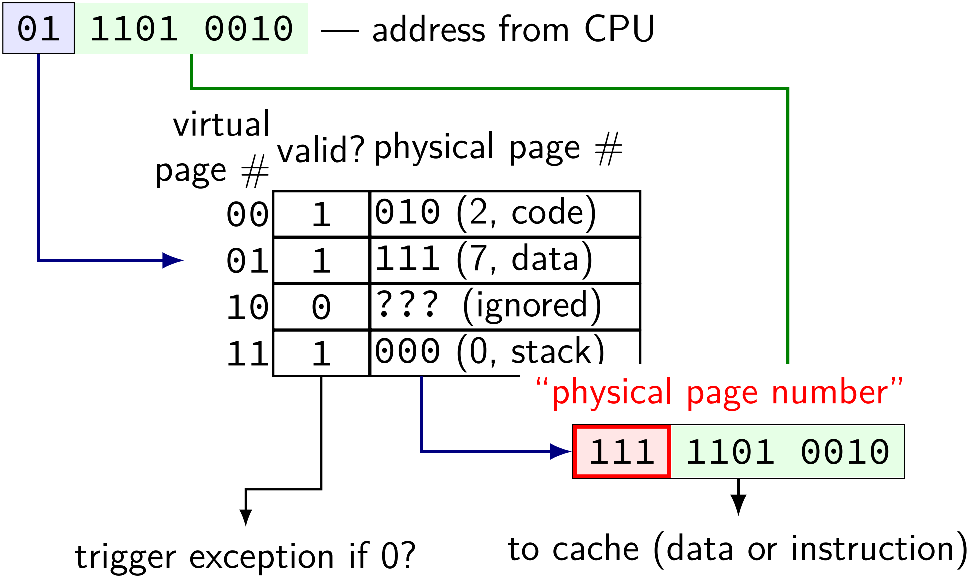
toy page table lookup



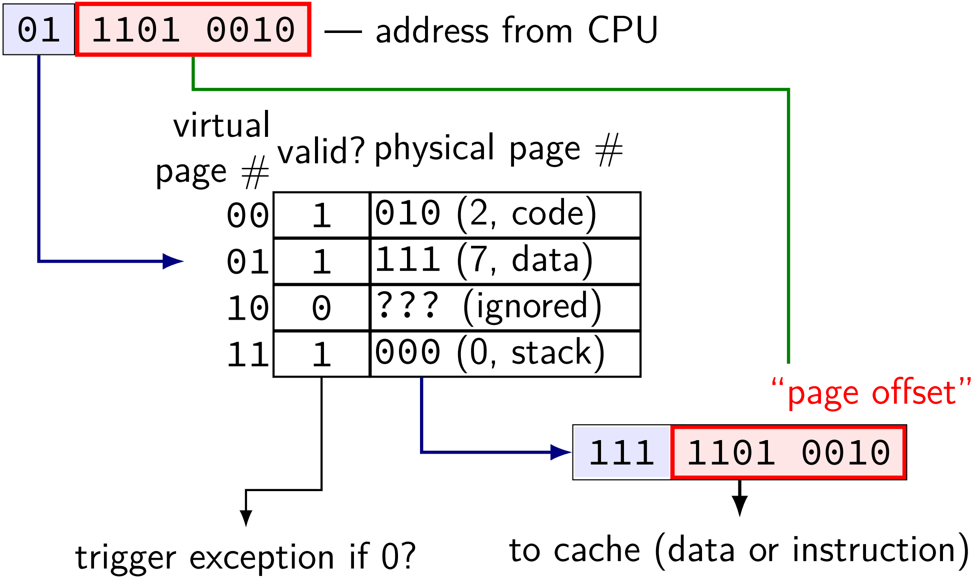
“virtual page number” lookup



toy page table lookup



toy pa “page offset” ookup



switching page tables

part of context switch is changing the page table

extra privileged instructions

switching page tables

part of context switch is changing the page table

extra privileged instructions

where in memory is the code that does this switching?

switching page tables

part of context switch is changing the page table

extra privileged instructions

where in memory is the code that does this switching?

- probably have a page table entry pointing to it
- hopefully marked kernel-mode-only

switching page tables

part of context switch is changing the page table

extra privileged instructions

where in memory is the code that does this switching?

- probably have a page table entry pointing to it
- hopefully marked kernel-mode-only

code better not be modified by user program

- otherwise: uncontrolled way to "escape" user mode

vector intrinsics: add example

```
int A[512], B[512];

for (int i = 0; i < 512; i += 8) {
  // "si256" --> 256 bit integer
  // a_values = {A[i], A[i+1], ..., A[i+7]} (8 x 32 bits)
  __m256i a_values = _mm256_loadu_si256((__m256i*) &A[i]);
  // b_values = {B[i], B[i+1] ..., A[i+7]} (8 x 32 bits)
  __m256i b_values = _mm256_loadu_si256((__m256i*) &B[i]);

  // add eight 32-bit integers
  // sums = {A[i] + B[i], A[i+1] + B[i+1], ...., A[i+7] + B[i+
  __m256i sums = _mm256_add_epi32(a_values, b_values);

  // {A[i], A[i+1], A[i+2], A[i+3], ..., A[i+7]} = sums
  _mm256_storeu_si256((__m256i*) &A[i], sums);
}
```

vector intrinsics: add example

special type `__m256i` — “256 bits of integers”
other types: `__m256` (floats), `__m128d` (doubles)

```
int A[51

for (int i = 0; i < 512; i += 8) {
  // "si256" --> 256 bit integer
  // a_values = {A[i], A[i+1], ..., A[i+7]} (8 x 32 bits)
  __m256i a_values = _mm256_loadu_si256((__m256i*) &A[i]);
  // b_values = {B[i], B[i+1] ..., A[i+7]} (8 x 32 bits)
  __m256i b_values = _mm256_loadu_si256((__m256i*) &B[i]);

  // add eight 32-bit integers
  // sums = {A[i] + B[i], A[i+1] + B[i+1], ...., A[i+7] + B[i+
  __m256i sums = _mm256_add_epi32(a_values, b_values);

  // {A[i], A[i+1], A[i+2], A[i+3], ..., A[i+7]} = sums
  _mm256_storeu_si256((__m256i*) &A[i], sums);
}
```

vector intrinsics: add example

functions to store/load

si256 means "256-bit integer value"

u for "unaligned" (otherwise, pointer address must be multiple of 32)

```
i
f
  // "si256" --> 256 bit integer
  // a_values = {A[i], A[i+1], ..., A[i+7]} (8 x 32 bits)
  __m256i a_values = _mm256_loadu_si256((__m256i*) &A[i]);
  // b_values = {B[i], B[i+1] ..., A[i+7]} (8 x 32 bits)
  __m256i b_values = _mm256_loadu_si256((__m256i*) &B[i]);

  // add eight 32-bit integers
  // sums = {A[i] + B[i], A[i+1] + B[i+1], ...., A[i+7] + B[i+
  __m256i sums = _mm256_add_epi32(a_values, b_values);

  // {A[i], A[i+1], A[i+2], A[i+3], ..., A[i+7]} = sums
  _mm256_storeu_si256((__m256i*) &A[i], sums);
}
```

vector intrinsics: add example

```
int A[512], B[512];

for (int i = 0; i < 512; i += 8) {
  // "si256" -->
  // a_values =                                    (8 x 32 bits)
  __m256i a_valu                                   56i*) &A[i]);
  // b_values = {B[i], B[i+1] ..., A[i+7]} (8 x 32 bits)
  __m256i b_values = _mm256_loadu_si256((__m256i*) &B[i]);

  // add eight 32-bit integers
  // sums = {A[i] + B[i], A[i+1] + B[i+1], ...., A[i+7] + B[i+
  __m256i sums = _mm256_add_epi32(a_values, b_values);

  // {A[i], A[i+1], A[i+2], A[i+3], ..., A[i+7]} = sums
  _mm256_storeu_si256((__m256i*) &A[i], sums);
}
```

function to add

epi32 means “8 32-bit integers”

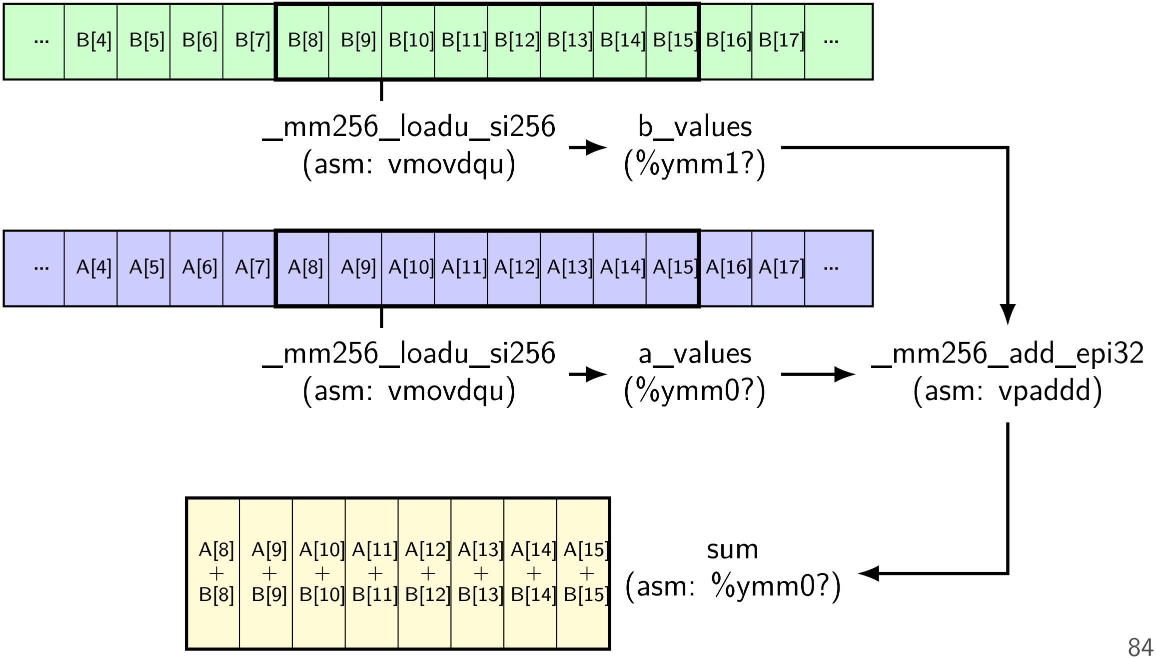
vector intrinsics: different size

```
long A[512], B[512]; /* instead of int */
...
for (int i = 0; i < 512; i += 4) {
  // a_values = {A[i], A[i+1], A[i+2], A[i+3]} (4 x 64 bits)
   __m256i a_values = _mm256_loadu_si256((__m256i*) &A[i]);
  // b_values = {B[i], B[i+1], B[i+2], B[i+3]} (4 x 64 bits)
   __m256i b_values = _mm256_loadu_si256((__m256i*) &B[i]);
   // add four 64-bit integers: vpaddq %ymm0, %ymm1
   // sums = {A[i] + B[i], A[i+1] + B[i+1], ...}
   __m256i sums = _mm256_add_epi64(a_values, b_values);
   // {A[i], A[i+1], A[i+2], A[i+3]} = sums
   _mm256_storeu_si256((__m256i*) &A[i], sums);
}
```

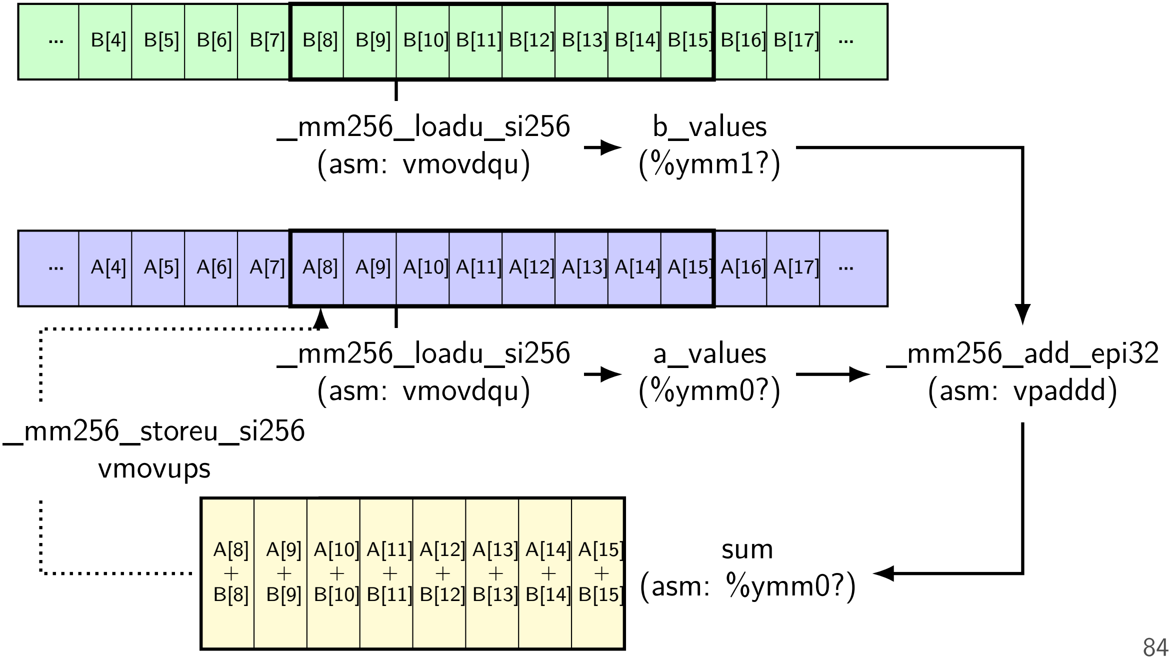
vector intrinsics: different size

```
long A[512], B[512]; /* instead of int */
...
for (int i = 0; i < 512; i += 4) {
  // a_values = {A[i], A[i+1], A[i+2], A[i+3]} (4 x 64 bits)
   __m256i a_values = _mm256_loadu_si256((__m256i*) &A[i]);
  // b_values = {B[i], B[i+1], B[i+2], B[i+3]} (4 x 64 bits)
   __m256i b_values = _mm256_loadu_si256((__m256i*) &B[i]);
   // add four 64-bit integers: vpaddq %ymm0, %ymm1
   // sums = {A[i] + B[i], A[i+1] + B[i+1], ...}
   __m256i sums = _mm256_add_epi64(a_values, b_values);
   // {A[i], A[i+1], A[i+2], A[i+3]} = sums
   _mm256_storeu_si256((__m256i*) &A[i], sums);
}
```

vector add picture (intrinsics)



vector add picture (intrinsics)



exercise

```
long foo[8] = {1,1,2,2,3,3,4,4};
long bar[8] = {2,2,2,3,3,3,4,4};
__mm256i foo0_as_vector = _mm256_loadu_si256((__m256i*)&foo[0])
__mm256i foo4_as_vector = _mm256_loadu_si256((__m256i*)&foo[4])
__mm256i bar0_as_vector = _mm256_loadu_si256((__m256i*)&bar[0])

__mm256i result = _mm256_add_epi64(foo0_as_vector, foo4_as_vector);
result = _mm256_mullo_epi64(result, bar0_as_vector);
_mm256_storeu_si256((__mm256i*) &bar[4], result);
```

Final value of bar array?

```
A. {2,2,2,3,12,12,24,24}  B. {2,2,2,3,15,15,28,28}
C. {2,2,2,3,10,10,20,20}  D. {12,12,24,24,3,3,4,4}
E. {14,14,26,27,3,3,4,4}  F. {14,14,26,27,12,12,24,24}
G. something else
```