

## Performance (finish) / Exceptions

1

## Changelog

Changes made in this version not seen in first lecture:

9 November 2017: an infinite loop: correct infinite loop code

9 November 2017: move sync versus async slide earlier

1

## 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"

2

## other vector instructions

multiple extensions to the X86 instruction set for vector instructions

this class: SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2  
supported on lab machines  
128-bit vectors

latest X86 processors: AVX, AVX2, AVX-512  
256-bit and 512-bit vectors

3

## other vector instructions features

AVX2/AVX/SSE pretty limiting

other vector instruction sets often more featureful:  
(and require more sophisticated HW support)

better conditional handling

better variable-length vectors

ability to load/store non-contiguous values

4

## addressing efficiency

```
for (int i = 0; i < N; ++i) {
  for (int j = 0; j < N; ++j) {
    float Bij = B[i * N + j];
    for (int k = kk; k < kk + 2; ++k) {
      Bij += A[i * N + k] * A[k * N + j];
    }
    B[i * N + j] = Bij;
  }
}
```

tons of multiplies by N??

isn't that slow?

5

## addressing transformation

```
for (int kk = 0; k < N; kk += 2 )
  for (int i = 0; i < N; ++i) {
    for (int j = 0; j < N; ++j) {
      float Bij = B[i * N + j];
      float *Akj_pointer = &A[kk * N + j];
      for (int k = kk; k < kk + 2; ++k) {
        // Bij += A[i * N + k] * A[k * N + j];
        Bij += A[i * N + k] * Akj_pointer;
        Akj_pointer += N;
      }
      B[i * N + j] = Bij;
    }
  }
```

transforms loop to **iterate with pointer**

**compiler** will usually do this!

increment/decrement by N ( $\times$  sizeof(float))

6

## addressing transformation

```
for (int kk = 0; k < N; kk += 2 )
  for (int i = 0; i < N; ++i) {
    for (int j = 0; j < N; ++j) {
      float Bij = B[i * N + j];
      float *Akj_pointer = &A[kk * N + j];
      for (int k = kk; k < kk + 2; ++k) {
        // Bij += A[i * N + k] * A[k * N + j];
        Bij += A[i * N + k] * Akj_pointer;
        Akj_pointer += N;
      }
      B[i * N + j] = Bij;
    }
  }
```

transforms loop to **iterate with pointer**

**compiler** will usually do this!

increment/decrement by N ( $\times$  sizeof(float))

6

## addressing efficiency

compiler will **usually** eliminate slow multiplies  
doing transformation yourself often slower if so

```
i * N; ++i into i_times_N; i_times_N += N
```

way to check: see if assembly uses lots multiplies in loop

if it doesn't — do it yourself

7

## optimizing real programs

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?

9

## profilers

first step — tool to determine where you spend time

tools exist to do this for programs

example on Linux: perf

8

10

## perf usage

*sampling* profiler

stops periodically, takes a look at what's running

perf record OPTIONS program

example OPTIONS:

-F 200 — record 200/second

--call-graph=dwarf — record stack traces

perf report or perf annotate

11

## children/self

“children” — samples in function or things it called

“self” — samples in function alone

12

## demo

13

## other profiling techniques

count number of times each function is called

not sampling — exact counts, but higher overhead  
might give less insight into amount of time

14

## tuning optimizations

biggest factor: how fast is it actually

setup a benchmark

make sure it's realistic (right size? uses answer? etc.)

compare the alternatives

15

16

## an infinite loop

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

If I run this on a lab machine, can you still use it?  
...if the machine only has one core?

17

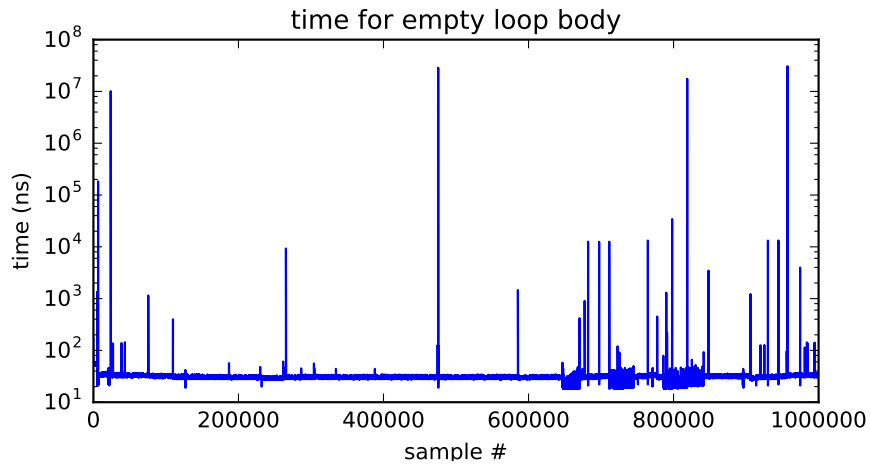
## timing nothing

```
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?

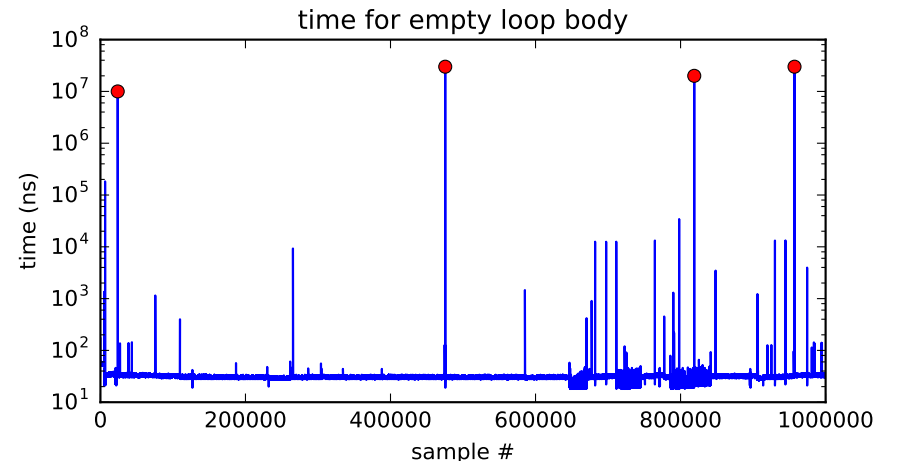
18

## doing nothing on a busy system



19

## doing nothing on a busy system



20

## time multiplexing



21

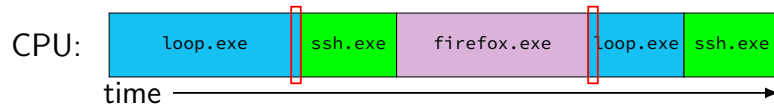
## time multiplexing



```
...  
call get_time  
    // whatever get_time does  
movq %rax, %rbp  
—— million cycle delay ——  
call get_time  
    // whatever get_time does  
subq %rbp, %rax  
...
```

21

## 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  
...
```

21

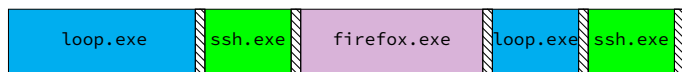
## time multiplexing really



 = operating system


22

## time multiplexing really



exception happens

return from exception

 = operating system

22

## 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**

23

## context

all registers values

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

condition codes

program counter

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

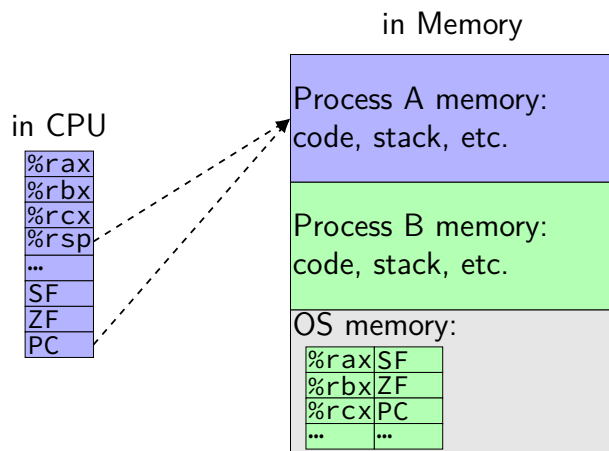
24

## 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
```

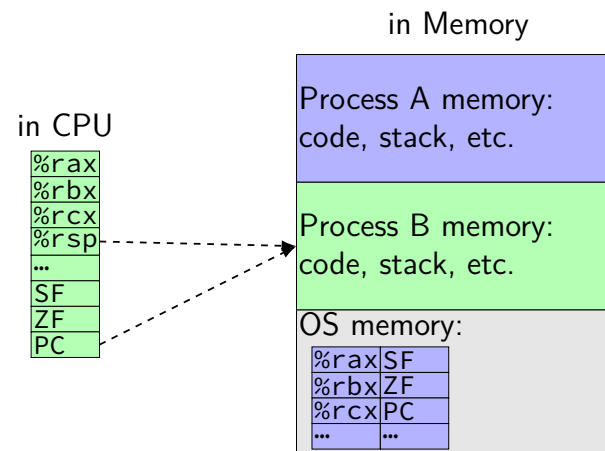
25

## contexts (A running)



26

## contexts (B running)



27



## memory protection

reading from another program's memory?

Program A	Program B
<pre>0x10000: .word 42 // ... // do work // ... movq 0x10000, %rax</pre>	<pre><i>// while A is working:</i> movq \$99, %rax movq %rax, 0x10000 ...</pre>

28

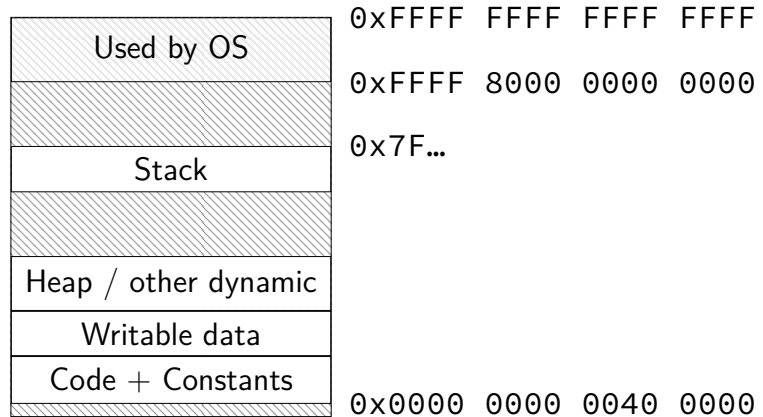
## memory protection

reading from another program's memory?

Program A	Program B
<pre>0x10000: .word 42 // ... // do work // ... movq 0x10000, %rax</pre>	<pre><i>// while A is working:</i> movq \$99, %rax movq %rax, 0x10000 ...</pre>
result: %rax is 42 (always)	result: <b>might crash</b>

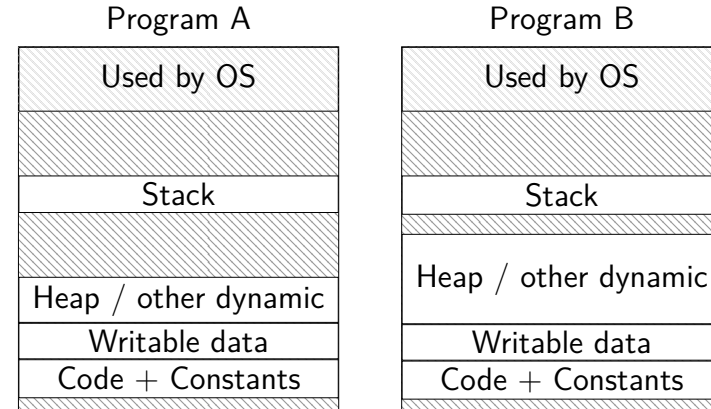
28

## program memory



29

## program memory (two programs)

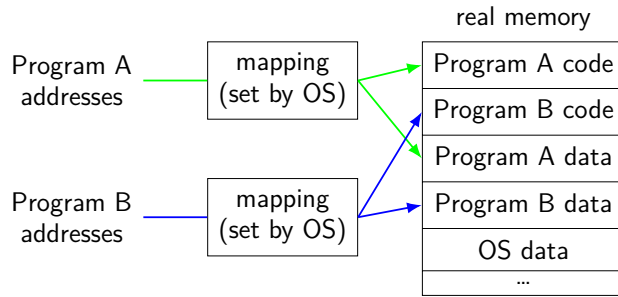


30

## address space

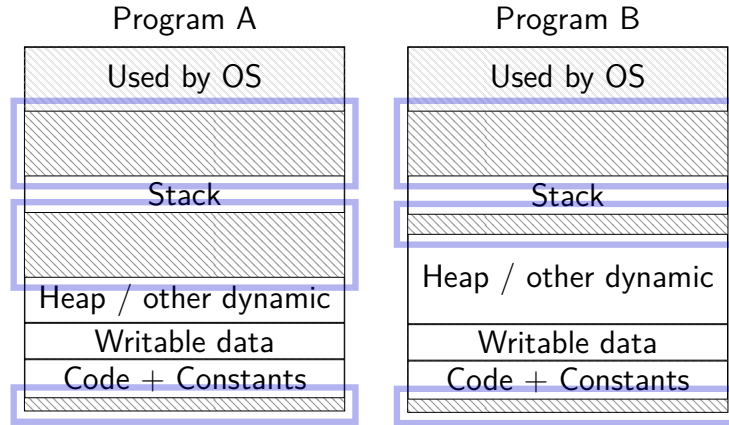
programs have **illusion of own memory**

called a program's **address space**



31

## program memory (two programs)

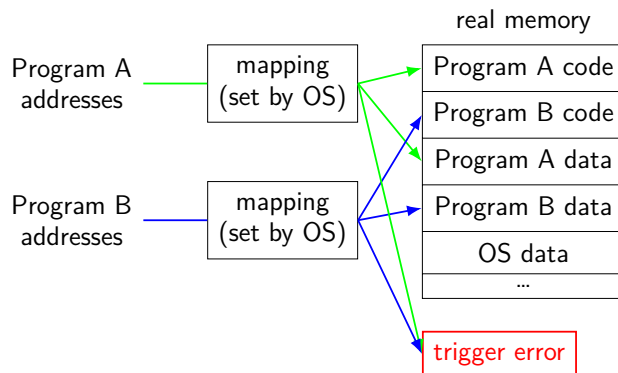


32

## address space

programs have **illusion of own memory**

called a program's **address space**



33

## address space mechanisms

next week's topic

called **virtual memory**

mapping called **page tables**

mapping part of what is changed in context switch

34

## 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

35

## The Process

**process** = thread(s) + address space

illusion of **dedicated machine**:

thread = illusion of own CPU

address space = illusion of own memory

36

## synchronous versus asynchronous

**synchronous** — triggered by a particular instruction  
traps and faults

**asynchronous** — comes from outside the program

interrupts and aborts

timer event

keypress, other input event

37

## types of exceptions

interrupts — externally-triggered

timer — keep program from hogging CPU

I/O devices — key presses, hard drives, networks, ...

faults — errors/events in programs

memory not in address space (“Segmentation fault”)

divide by zero

invalid instruction

traps — intentionally triggered exceptions

system calls — ask OS to do something

aborts

38

## types of exceptions

**interrupts** — externally-triggered

**timer** — keep program from hogging CPU

I/O devices — key presses, hard drives, networks, ...

**faults** — errors/events in programs

memory not in address space (“Segmentation fault”)

divide by zero

invalid instruction

**traps** — intentionally triggered exceptions

system calls — ask OS to do something

aborts

39

## 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

40

## types of exceptions

**interrupts** — externally-triggered

**timer** — keep program from hogging CPU

I/O devices — key presses, hard drives, networks, ...

**faults** — errors/events in programs

memory not in address space (“Segmentation fault”)

divide by zero

invalid instruction

**traps** — intentionally triggered exceptions

system calls — ask OS to do something

aborts

41

## types of exceptions

**interrupts** — externally-triggered

**timer** — keep program from hogging CPU

I/O devices — key presses, hard drives, networks, ...

**faults** — errors/events in programs

memory not in address space (“Segmentation fault”)

divide by zero

invalid instruction

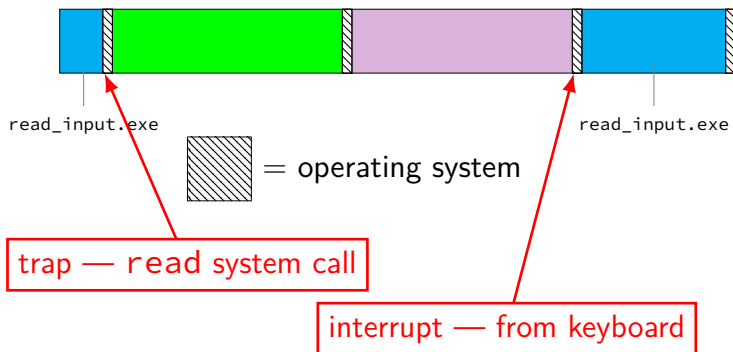
**traps** — intentionally triggered exceptions

**system calls** — ask OS to do something

aborts

42

## keyboard input timeline



43

## types of exceptions

**interrupts** — externally-triggered

timer — keep program from hogging CPU

**I/O devices** — key presses, hard drives, networks, ...

**faults** — errors/events in programs

memory not in address space (“Segmentation fault”)

divide by zero

invalid instruction

**traps** — intentionally triggered exceptions

system calls — ask OS to do something

**aborts**

44

## types of exceptions

**interrupts** — externally-triggered

timer — keep program from hogging CPU

**I/O devices** — key presses, hard drives, networks, ...

**faults** — errors/events in programs

memory not in address space (“Segmentation fault”)

divide by zero

invalid instruction

**traps** — intentionally triggered exceptions

system calls — ask OS to do something

**aborts**

45

## 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

46

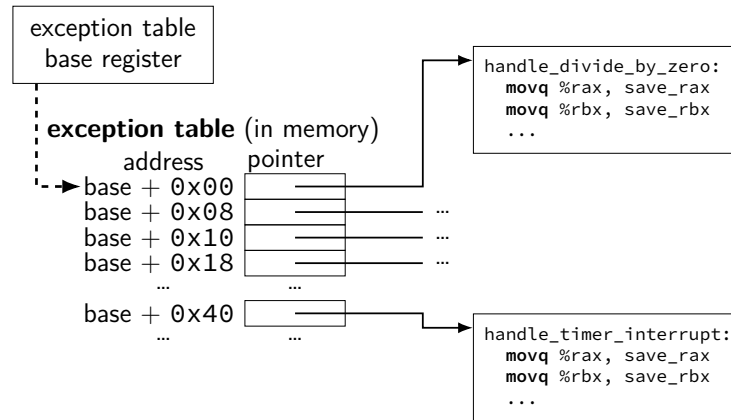
## exception implementation: notes

I/textbook describe a **simplified** version

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

47

## locating exception handlers



48

## 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.

49

## added to CPU for exceptions

new instruction: set exception table base

new logic: jump based on exception table

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

50

## added to CPU for exceptions

new instruction: set **exception table base**

new logic: **jump based on exception table**

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

50

## added to CPU for exceptions

new instruction: set exception table base

new logic: jump based on exception table

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

50

## added to CPU for exceptions

new instruction: set exception table base

new logic: jump based on exception table

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

50

## why return from exception?

reasons related to protection (later)

not just ret — can't modify process's stack  
would break the **illusion of dedicated CPU/memory**  
program could use stack in weird way

```
movq $100, -8(%rsp)
```

```
...
```

```
movq -8(%rsp), %rax
```

(even though this wouldn't be following calling conventions)

need to restart program **undetectably!**

51

## 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
```

52

## 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
```

53

## 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
```

54

## 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!)

55



## 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

56

## kernel mode

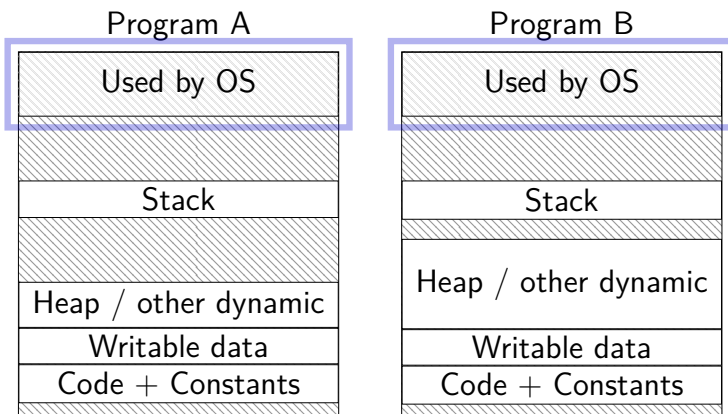
extra one-bit register: “are we in kernel mode”

exceptions **enter kernel mode**

return from exception instruction **leaves kernel mode**

57

## program memory (two programs)

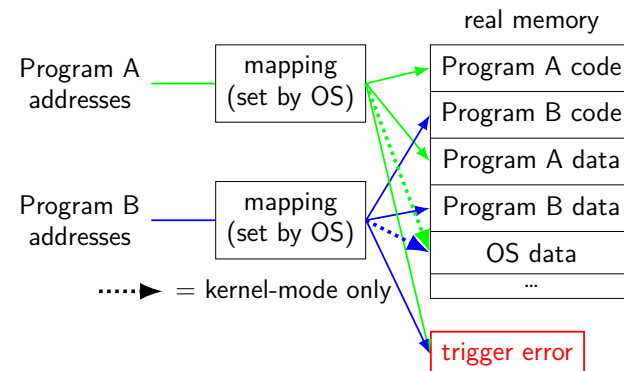


58

## address space

programs have **illusion of own memory**

called a program's **address space**



59

## types of exceptions

interrupts — externally-triggered

timer — keep program from hogging CPU

I/O devices — key presses, hard drives, networks, ...

**faults** — errors/events in programs

memory not in address space (“Segmentation fault”)

divide by zero

invalid instruction

**traps** — intentionally triggered exceptions

system calls — ask OS to do something

aborts

60

## 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

61

## types of exceptions

interrupts — externally-triggered

timer — keep program from hogging CPU

I/O devices — key presses, hard drives, networks, ...

**faults** — errors/events in programs

memory not in address space (“Segmentation fault”)

divide by zero

invalid instruction

**traps** — intentionally triggered exceptions

**system calls** — ask OS to do something

aborts

62

## kernel services

allocating memory? (change address space)

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

read input? (communicate with keyboard)

all need privileged instructions!

need to **run code in kernel mode**

63

## Linux x86-64 system calls

special instruction: `syscall`

triggers **trap** (deliberate exception)

64

## 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

65

## 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
```

66

## 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
```

67

## 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

68

## system calls and protection

exceptions are **only way** to access kernel mode

operating system controls what processes can do

... by writing exception handlers **very carefully**

69

## careful exception handlers

```
movq $important_os_address, %rsp
```

can't trust user's **stack pointer!**

need to have own stack in kernel-mode-only memory

need to check all inputs really carefully

70

## protection and sudo

programs **always** run in user mode

extra permissions from OS **do not change this**  
sudo, superuser, root, SYSTEM, ...

**operating system** may remember extra privileges

71

## system call wrappers

library functions to not write assembly:

```
open:
    movq $2, %rax // 2 = sys_open
    // 2 arguments happen to use same registers
    syscall
    // return value in %eax
    cmp $0, %rax
    jl has_error
    ret
has_error:
    neg %rax
    movq %rax, errno
    movq $-1, %rax
    ret
```

72

## system call wrappers

library functions to not write assembly:

```
open:
    movq $2, %rax // 2 = sys_open
    // 2 arguments happen to use same registers
    syscall
    // return value in %eax
    cmp $0, %rax
    jl has_error
    ret
has_error:
    neg %rax
    movq %rax, errno
    movq $-1, %rax
    ret
```

72

## system call wrapper: usage

```
/* unistd.h contains definitions of:
   O_RDONLY (integer constant), open() */
#include <unistd.h>
int main(void) {
    int file_descriptor;
    file_descriptor = open("input.txt", O_RDONLY);
    if (file_descriptor < 0) {
        printf("error:_%s\n", strerror(errno));
        exit(1);
    }
    ...
    result = read(file_descriptor, ...);
    ...
}
```

73

## system call wrapper: usage

```
/* unistd.h contains definitions of:
   O_RDONLY (integer constant), open() */
#include <unistd.h>
int main(void) {
    int file_descriptor;
    file_descriptor = open("input.txt", O_RDONLY);
    if (file_descriptor < 0) {
        printf("error:_%s\n", strerror(errno));
        exit(1);
    }
    ...
    result = read(file_descriptor, ...);
    ...
}
```

73

## 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

74

## 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 privileged operations work

75

## recall: square

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

76

77

## square unrolled

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

78

## handy intrinsic functions for square

`_mm_set1_epi32` — load four copies of a 32-bit value into a 128-bit value

instructions generated vary; one example: `movq + pshufd`

`_mm_mullo_epi32` — multiply four pairs of 32-bit values, give lowest 32-bits of results

generates `pmulld`

79

## vectorizing square

```
/* goal: vectorize this */
B[i * N + j + 0] += A[i * N + k] * A[k * N + j + 0];
B[i * N + j + 1] += A[i * N + k] * A[k * N + j + 1];
B[i * N + j + 2] += A[i * N + k] * A[k * N + j + 2];
B[i * N + j + 3] += A[i * N + k] * A[k * N + j + 3];
```

80

## vectorizing square

```
/* goal: vectorize this */
B[i * N + j + 0] += A[i * N + k] * A[k * N + j + 0];
B[i * N + j + 1] += A[i * N + k] * A[k * N + j + 1];
B[i * N + j + 2] += A[i * N + k] * A[k * N + j + 2];
B[i * N + j + 3] += A[i * N + k] * A[k * N + j + 3];
```

```
// load four elements from B
Bij = _mm_loadu_si128(&B[i * N + j + 0]);
... // manipulate vector here
// store four elements into B
_mm_storeu_si128((__m128i*) &B[i * N + j + 0], Bij);
```

80

## vectorizing square

```
/* goal: vectorize this */
B[i * N + j + 0] += A[i * N + k] * A[k * N + j + 0];
B[i * N + j + 1] += A[i * N + k] * A[k * N + j + 1];
B[i * N + j + 2] += A[i * N + k] * A[k * N + j + 2];
B[i * N + j + 3] += A[i * N + k] * A[k * N + j + 3];
```

---

```
// load four elements from A
AkJ = _mm_loadu_si128(&A[k * N + j + 0]);
... // multiply each by A[i * N + k] here
```

80

## vectorizing square

```
/* goal: vectorize this */
B[i * N + j + 0] += A[i * N + k] * A[k * N + j + 0];
B[i * N + j + 1] += A[i * N + k] * A[k * N + j + 1];
B[i * N + j + 2] += A[i * N + k] * A[k * N + j + 2];
B[i * N + j + 3] += A[i * N + k] * A[k * N + j + 3];
```

---

```
// load four elements starting with A[k * n + j]
AkJ = _mm_loadu_si128(&A[k * N + j + 0]);
// load four copies of A[i * N + k]
Aik = _mm_set1_epi32(A[i * N + k]);
// multiply each pair
multiply_results = _mm_mullo_epi32(Aik, AkJ);
```

80

## vectorizing square

```
/* goal: vectorize this */
B[i * N + j + 0] += A[i * N + k] * A[k * N + j + 0];
B[i * N + j + 1] += A[i * N + k] * A[k * N + j + 1];
B[i * N + j + 2] += A[i * N + k] * A[k * N + j + 2];
B[i * N + j + 3] += A[i * N + k] * A[k * N + j + 3];
```

---

```
Bij = _mm_add_epi32(Bij, multiply_results);
// store back results
_mm_storeu_si128(..., Bij);
```

80

## square vectorized

```
_mm128i Bij, AkJ, Aik, Aik_times_AkJ;

// Bij = {Bi,j, Bi,j+1, Bi,j+2, Bi,j+3}
Bij = _mm_loadu_si128((__m128i*) &B[i * N + j]);
// AkJ = {Ak,j, Ak,j+1, Ak,j+2, Ak,j+3}
AkJ = _mm_loadu_si128((__m128i*) &A[k * N + j]);

// Aik = {Ai,k, Ai,k, Ai,k, Ai,k}
Aik = _mm_set1_epi32(A[i * N + k]);

// Aik_times_AkJ = {Ai,k × Ak,j, Ai,k × Ak,j+1, Ai,k × Ak,j+2, Ai,k × Ak,j+3}
Aik_times_AkJ = _mm_mullo_epi32(Aik, AkJ);

// Bij = {Bi,j + Ai,k × Ak,j, Bi,j+1 + Ai,k × Ak,j+1, ...}
Bij = _mm_add_epi32(Bij, Aik_times_AkJ);

// store Bij into B
_mm_storeu_si128((__m128i*) &B[i * N + j], Bij);
```

81



## constant multiplies/divides (1)

```
unsigned int fiveEights(unsigned int x) {  
    return x * 5 / 8;  
}
```

---

```
fiveEights:  
    leal    (%rdi,%rdi,4), %eax  
    shr    $3, %eax  
    ret
```

82

## constant multiplies/divides (2)

```
int oneHundredth(int x) { return x / 100; }
```

---

```
oneHundredth:  
    movl    %edi, %eax  
    movl    $1374389535, %edx  
    sarl    $31, %edi  
    imull   %edx  
    sarl    $5, %edx  
    movl    %edx, %eax  
    subl    %edi, %eax  
    ret
```

$$\frac{1374389535}{2^{37}} \approx \frac{1}{100}$$

83

## constant multiplies/divides

compiler is very good at handling

...but need to actually use constants

84