sync-reorder

the correctness problem

two threads?

introduces *non-determinism*

which one runs first?

allows for “race condition” bugs

… to be avoided with synchronization constructs

a threaded server?

```
Deposit(accountNumber, amount) {
    account = GetAccount(accountId);
    account->balance += amount;
    SaveAccountUpdates(account);
}
```

maybe GetAccount/SaveAccountUpdates can be slow?

- read/write disk sometimes? contact another server sometimes?

maybe lots of requests to process?

- maybe real logic has more checks than Deposit()
- ...

all reasons to handle multiple requests at once

$\longrightarrow$ many threads all running the server loop

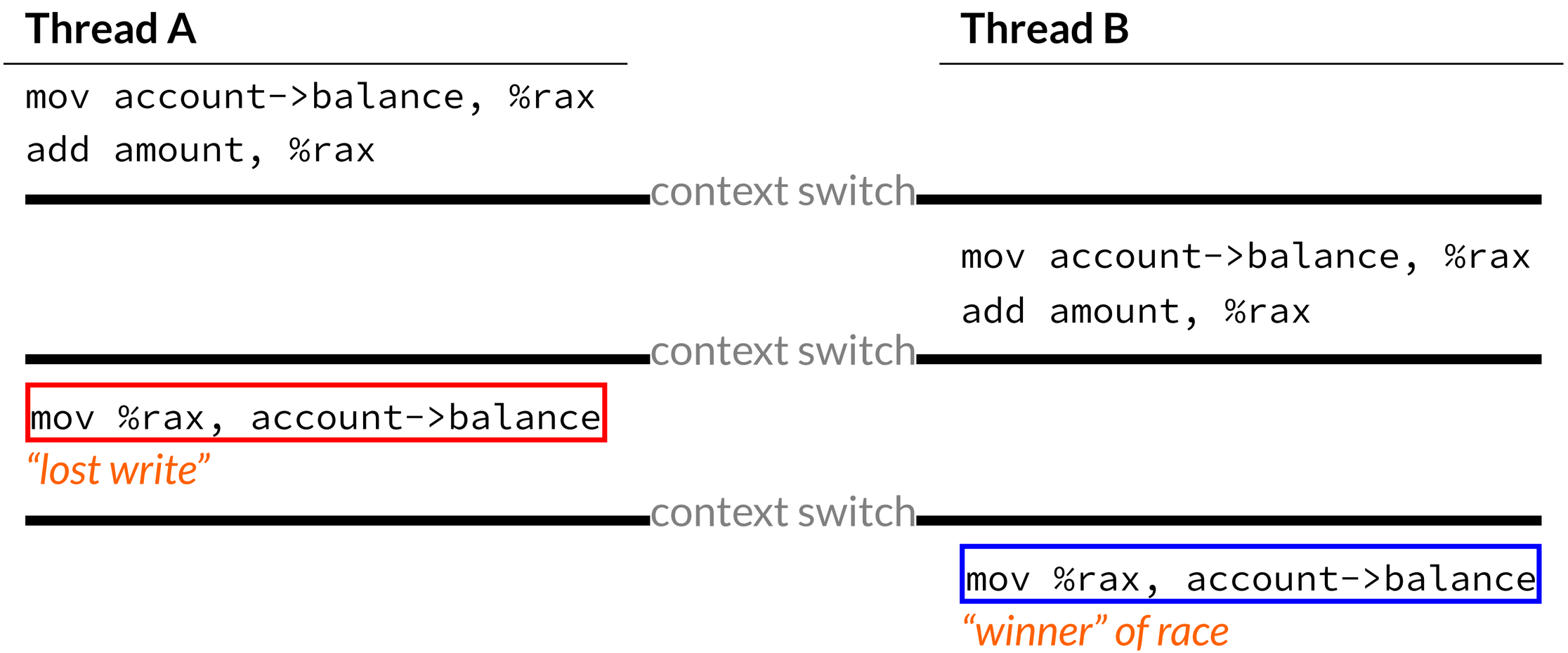
the lost write

`account->balance += amount;` (in two threads; same account)

Thread A	Thread B
`mov account->balance, %rax`	
`add amount, %rax`	
context switch	context switch
	`mov account->balance, %rax`
	`add amount, %rax`
context switch	context switch
`mov %rax, account->balance`	
context switch	context switch
	`mov %rax, account->balance`

the lost write

`account->balance += amount;` (in two threads; same account)



thinking about race conditions (1)

what are the possible values of x? (initially $x = y = 0$)

Thread A	*Thread B*
$x \leftarrow 1$	$y \leftarrow 2$

thinking about race conditions (1)

what are the possible values of x? (initially $x = y = 0$)

Thread A	*Thread B*
$x \leftarrow 1$	$y \leftarrow 2$

must be 1. Thread B can't do anything

thinking about race conditions (2)

possible values of x? (initially $x = y = 0$)

Thread A	*Thread B*
$x \leftarrow y + 1$	$y \leftarrow 2$
	$y \leftarrow y \times 2$

thinking about race conditions (2)

possible values of x? (initially $x = y = 0$)

Thread A	*Thread B*
$x \leftarrow y + 1$	$y \leftarrow 2$
	$y \leftarrow y \times 2$

if A goes first, then B: 1

if B goes first, then A: 5

if B line one, then A, then B line two: 3

thinking about race conditions (3)

what are the possible values of x?

(initially $x = y = 0$)

Thread A	*Thread B*
$x \leftarrow 1$	$x \leftarrow 2$

thinking about race conditions (3)

what are the possible values of x?

(initially $x = y = 0$)

Thread A	*Thread B*
$x \leftarrow 1$	$x \leftarrow 2$

1 or 2

thinking about race conditions (3)

what are the possible values of x?

(initially $x = y = 0$)

Thread A	*Thread B*
$x \leftarrow 1$	$x \leftarrow 2$

1 or 2

… but why not 3?

B: x bit 0 $\leftarrow 0$

A: x bit 0 $\leftarrow 1$

A: x bit 1 $\leftarrow 0$

B: x bit 1 $\leftarrow 1$

thinking about race conditions (2, reprise)

possible values of x? (initially $x = y = 0$)

Thread A	*Thread B*
$x \leftarrow y + 1$	$y \leftarrow 2$
	$y \leftarrow y \times 2$

why not 7?

B (start): $y \leftarrow 2 = 0010_{\mathrm{TWO}}$; then y bit 3 $\leftarrow$ 0; y bit 2 $\leftarrow$ 1; then

A: x $\leftarrow 110_{\mathrm{TWO}} + 1 = 7$; then

B (finish): y bit 1 $\leftarrow$ 0; y bit 0 $\leftarrow$ 0

atomic operation

atomic operation = operation that runs to completion or not at all

we will use these to let threads work together

most machines: loading/storing (aligned) words is atomic

- so can't get 3 from $x \leftarrow 1$ and $x \leftarrow 2$ running in parallel
- aligned $\approx$ address of word is multiple of word size (typically done by compilers)

but some instructions are not atomic; examples:

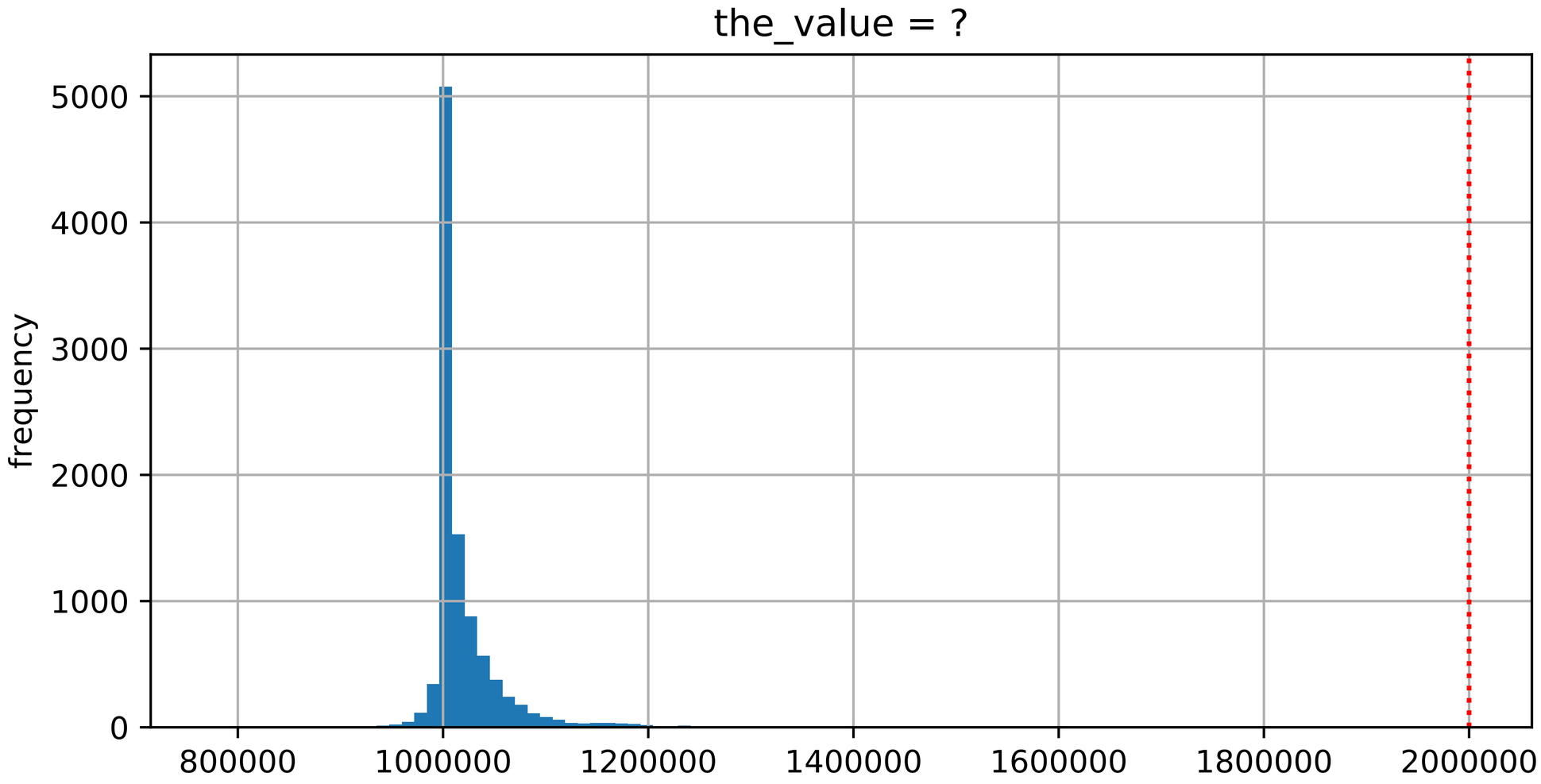
- x86: integer `add` constant to memory location
- many CPUs: loading/storing values that cross cache blocks
  - e.g. if cache blocks `0x40` bytes, load/store 4 byte from addr. `0x3E` is not atomic

lost adds (program)

```
.global update_loop
update_loop:
    addl $1, the_value // the_value (global variable) += 1
    dec %rdi           // argument 1 -= 1
    jg update_loop     // if argument 1 >= 0 repeat
    ret
```

```
int the_value;
extern void *update_loop(void *);
int main(void) {
    the_value = 0;
    pthread_t A, B;
    pthread_create(&A, NULL, update_loop, (void*) 1000000);
    pthread_create(&B, NULL, update_loop, (void*) 1000000);
    pthread_join(A, NULL); pthread_join(B, NULL);
    // expected result: 1000000 + 1000000 = 2000000
    printf("the_value = %d\n", the_value);
}
```

lost adds (results)



but how?

probably not possible on single core

- exceptions can't occur in the middle of `add` instruction

… but 'add to memory' implemented with multiple steps

- still needs to load, add, store internally
- can be interleaved with what other cores do

(and actually it's more complicated than that — we'll talk later)

so, what is actually atomic

for now we'll assume: load/stores of 'words'

(64-bit machine = 64-bits words)

in general: *processor designer will tell you*

their job to design caches, etc. to work as documented

compilers move loads/stores (1)

```
void WaitForReady() {
  do {} while (!ready);
}
```

```
WaitForReady:
  movl ready, %eax          // eax <- other_ready

  // value only loaded before loop
.L2:
  testl %eax, %eax
  je .L2                    // while (eax == 0) repeat
  ...
```

C standard says: can assume no other thread
So, it's fine to load `ready` in advance and reuse that

compilers move loads/stores (2)

```
void WaitForReady() {
  is_waiting = 1;
  do {} while (!ready);
  is_waiting = 0;
}
```

```
WaitForOther:
  movl ready, %eax
.L2:
  testl %eax, %eax
  je .L2                    // while (eax == 0) repeat
  movl $0, is_waiting
  ret
```

C standard says: can assume no other thread

So, can do assignments `is_waiting` anytime before return

And `is_waiting = 1; is_waiting = 0` same as `is_waiting = 0`

fixing compiler reordering?

isn't there a way to tell compiler not to do these optimizations?

yes, but that is *still not enough*!

processors sometimes do this kind of reordering too (between cores)

pthreads and reordering

many pthreads functions *prevent reordering*

- everything before function call actually happens before

includes *preventing some optimizations*

- e.g. keeping global variable in register for too long

pthread_create, pthread_join, other tools we'll talk about …

- basically: if pthreads is waiting for/starting something, no weird ordering

implementation part 1: prevent compiler reordering

implementation part 2: use special instructions

- example: x86 `mfence` instruction

Backup slides

a simple race

```
thread_A:
    movl $1, x   /* x <- 1 */
    movl y, %eax /* return y */
    ret
```

```
thread_B:
    movl $1, y   /* y <- 1 */
    movl x, %eax /* return x */
    ret
```

```
x = y = 0;
pthread_create(&A, NULL, thread_A, NULL);
pthread_create(&B, NULL, thread_B, NULL);
pthread_join(A, &A_result); pthread_join(B, &B_result);
printf("A:%d B:%d\n", (int) A_result, (int) B_result);
```

if loads/stores atomic, then possible results:

- A:1 B:1 — both moves into x and y, then both moves into eax execute
- A:0 B:1 — thread A executes before thread B
- A:1 B:0 — thread B executes before thread A

a simple race: results

```
thread_A:
    movl $1, x   /* x <- 1 */
    movl y, %eax /* return y */
    ret
```

```
thread_B:
    movl $1, y   /* y <- 1 */
    movl x, %eax /* return x */
    ret
```

```
x = y = 0;
pthread_create(&A, NULL, thread_A, NULL);
pthread_create(&B, NULL, thread_B, NULL);
pthread_join(A, &A_result); pthread_join(B, &B_result);
printf("A:%d B:%d\n", (int) A_result, (int) B_result);
```

my desktop, 100M trials:

frequency	result	
99 823 739	A:0 B:1	(‘A executes before B’)
171 161	A:1 B:0	(‘B executes before A’)
4 706	A:1 B:1	(‘execute moves into x+y first’)
394	A:0 B:0	???

a simple race: results

```
thread_A:
    movl $1, x   /* x <- 1 */
    movl y, %eax /* return y */
    ret
```

```
thread_B:
    movl $1, y   /* y <- 1 */
    movl x, %eax /* return x */
    ret
```

```
x = y = 0;
pthread_create(&A, NULL, thread_A, NULL);
pthread_create(&B, NULL, thread_B, NULL);
pthread_join(A, &A_result); pthread_join(B, &B_result);
printf("A:%d B:%d\n", (int) A_result, (int) B_result);
```

my desktop, 100M trials:

frequency	result	
99 823 739	A:0 B:1	(‘A executes before B’)
171 161	A:1 B:0	(‘B executes before A’)
4 706	A:1 B:1	(‘execute moves into x+y first’)
394	*A:0 B:0*	*???*

why reorder here?

```
thread_A:
    movl $1, x    /* x <- 1 */
    movl y, %eax  /* return y */
    ret
```

```
thread_B:
    movl $1, y    /* y <- 1 */
    movl x, %eax  /* return x */
    ret
```

thread A: faster to load `y` right now!

… rather than wait for write of `x` to finish

why load/store reordering?

fast processor designs can execute instructions out of order

goal: do something instead of waiting for slow memory accesses, etc.

more on this later in the semester

GCC: preventing reordering example (1)

```
void WaitForReady() {
    int one = 1;
    __atomic_store(&is_waiting, &one, __ATOMIC_SEQ_CST);
    do {
    } while (!__atomic_load_n(&ready, __ATOMIC_SEQ_CST));
    ...
}
```

```
WaitForReady:
  movl $1, is_waiting
  mfence
.L2:
  movl ready, %eax
  testl %eax, %eax
  jz .L2
  ...
```

GCC: preventing reordering example (2)

```
void WaitForReady() {
    is_waiting = 1;
    do {
        __atomic_thread_fence(__ATOMIC_SEQ_CST);
    } while (!ready);
    ...
}
```

```
WaitForReady:
  movl $1, is_waiting // is_waiting <- 1
.L3:
  mfence  // make sure store is visible to other cores before loading
          // on x86: not needed on second+ iteration of loop
  cmpl $0, ready // if (ready == 0) repeat fence
  jz .L3
  ...
```

C++: preventing reordering

to help *implementing things like pthread_mutex_lock*

C++ 2011 standard: *atomic* header, *std::atomic* class

prevent CPU reordering *and* prevent compiler reordering

also provide other tools for implementing locks (more later)

could also hand-write assembly code

- compiler can't know what assembly code is doing

C++: preventing reordering example

```
#include <atomic>
void WaitForReady() {
    is_waiting = 1;
    do {
        std::atomic_thread_fence(std::memory_order_seq_cst);
    } while (!ready);
}
```

```
WaitForReady:
  movl $1, is_waiting // is_waiting <- 1
.L2:
  mfence  // make sure store visible on/from other cores
  cmpl $0, ready // if (ready == 0) repeat fence
  jz .L2
  ...
```

C++ atomics: no reordering

```
std::atomic<int> is_waiting, ready;
void WaitForReaady() {
    is_waiting.store(1);
    do {
    } while (ready.load());
}
```

```
WaitForReady:
  movl $1, is_waiting
  mfence
.L2:
  movl ready, %eax
  testl %eax, %eax
  jz .L2
  ...
```

GCC: built-in atomic functions

used to implement std::atomic, etc.

predate std::atomic

builtin functions starting with `__sync` and `__atomic`

aside: some x86 reordering rules

each core sees its own loads/stores in order

(if a core stores something, it can always load it back)

stores *from other cores* appear in a consistent order

(but a core might observe its own stores too early)

causality:

if a core reads X=a and (after reading X=a) writes Y=b,

then a core that reads Y=b cannot later read X=older value than a

Source: Intel 64 and IA-32 Software Developer's Manual, Volume 3A, Chapter 8

how do you do anything with this?

difficult to reason about what modern CPU's reordering rules do

typically: don't depend on details, instead:

special instructions with stronger (and simpler) ordering rules

- often same instructions that help with implementing locks in other ways

special instructions that restrict ordering of instructions around them ("fences")

- loads/stores can't cross the fence

mfence

x86 instruction `mfence`

make sure all loads/stores in progress finish

… and make sure no loads/stores were started early

fairly expensive

- Intel ‘Skylake’: order 33 cycles + time waiting for pending stores/loads

aside: this instruction did not exist in the original x86
so xv6 uses something older that’s equivalent