last time

translation lookaside buffers

- special additional cache for last-level page table entries
- looked by virtual page number
- can practically be very small and therefore very fast

pthread API — pthread_create, pthread_join

- pthread_join — collect thread function return value + wait for thread to finish
- like waitpid: can call when thread already finished

quiz Q1-2

write 4 bytes, set index 4, tag 0x1234 — miss (W 0, R 12)

- write-allocate: read rest of block (12 bytes)
- write-back: store written data in cache only + mark dirty

read 4 bytes, set index 3, tag 0x1234 — miss (W 0, R 16)

- read 16 bytes (block)

write 4 bytes, set index 3, tag 0x1234 — hit (W 0, R 0)

- write-back: modify locally, mark dirty

write 4 bytes, set index 4, tag 0x1234 — miss (W 16, R 12)

- write-allocate: evict other block which is dirty → write 16 bytes
- write-allocate: read rest of block (12 bytes)
- write-back: store written data in cache + mark dirty)

writes to next: 0+0+0+16=16; reads: 12+16+0+12=40

quiz Q3

0x1000000-0x100000f: cache set 0, array elems 0–3

0x1000010-0x100001f: cache set 1, array elems 4–7

...

0x1000190-0x100019f: cache set 25, array elems 100–103

...

0x1000ff0-0x1000fff: cache set 255, array elems 1020–1023

0x1001000-0x100100f: cache set 0, array elems 1024–1027

0x1001010-0x100101f: cache set 1, array elems 1028–1031

...

0x1001190-0x100119f: cache set 25, array elems 1124–1127

quiz Q4

16 entries and 2 ways $\rightarrow$ 8 entries/way $\rightarrow$ 8 sets

virtual address 0xABCDEF: VPN 0xABC, page offset 0xDEF

0xABC = (TLB tag) 1010 1011 1 (TLB index) 100 (4)

quiz Q5

two address 0x1000 bytes apart same cache set?

not possible if physical addresses (different index bits)

problem: index bits depend on page table mapping

if consecutive VPNs map to similar physical page numbers

…have same index bits

quiz Q6

*p = *p + x

modifies *p (what p points to)

p points to variable z

z is local variable for main()

value is on stack

quiz Q7

pthread_create returns when new thread is setup

thread may not run until processor core available

thread might run really fast

so all but D are possible

re D: thread's retun value needs to be kept around + related bookkeeping

thread joining

pthread_join allows collecting thread return value

if you don't join joinable thread, then memory leak!

thread joining

pthread_join allows collecting thread return value

if you don't join joinable thread, then memory leak!

avoiding memory leak?

always join...or

"detach" thread to make it not joinable

pthread_detach

```
void *show_progress(void * ...) { ... }
void spawn_show_progress_thread() {
    pthread_t show_progress_thread;
    pthread_create(&show_progress_thread, NULL,
                   show_progress, NULL);

    /* instead of keeping pthread_t around to join thread later: */
    pthread_detach(show_progress_thread);
}

int main() {
    spawn_show_progress_thread();
    do_othe
    ...
}
```

detach = don't care about return value, etc.
system will deallocate when thread terminates

starting threads detached

```
void *show_progress(void * ...) { ... }
void spawn_show_progress_thread() {
    pthread_t show_progress_thread;
    pthread_attr_t attrs;
    pthread_attr_init(&attrs);
    pthread_attr_setdetachstate(&attrs, PTHREAD_CREATE_DETACHED);
    pthread_create(&show_progress_thread, attrs,
                   show_progress, NULL);
    pthread_attr_destroy(&attrs);
}
```

setting stack sizes

```
void *show_progress(void * ...) { ... }
void spawn_show_progress_thread() {
    pthread_t show_progress_thread;
    pthread_attr_t attrs;
    pthread_attr_init(&attrs);
    pthread_attr_setstacksize(&attrs, 32 * 1024 /* bytes */);
    pthread_create(&show_progress_thread, attrs,
                   show_progress, NULL);
}
```

a threading race

```
#include <pthread.h>
#include <stdio.h>
void *print_message(void *ignored_argument) {
    printf("In the thread\n");
    return NULL;
}
int main() {
    printf("About to start thread\n");
    pthread_t the_thread;
    /* assume does not fail */
    pthread_create(&the_thread, NULL, print_message, NULL);
    printf("Done starting thread\n");
    return 0;
}
```

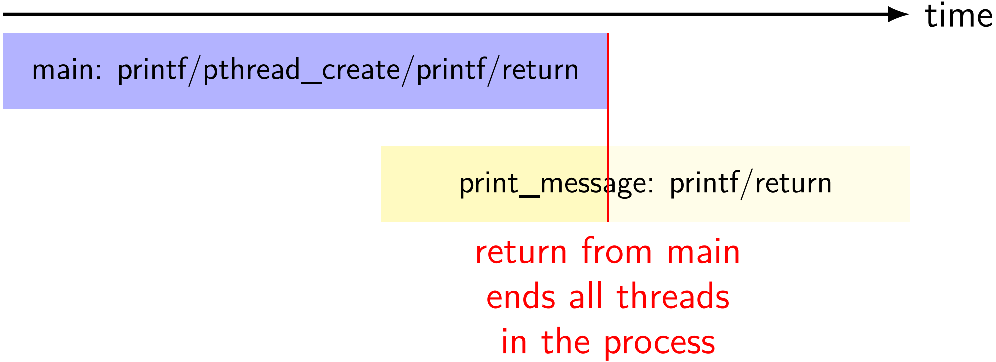
My machine: outputs `In the thread` about 4% of the time.

a race

returning from main exits the entire process (all its threads)

- same as calling exit; not like other threads

race: main's return 0 or print_message's printf first?



the correctness problem

two threads?

introduces *non-determinism*

which one runs first?

allows for "race condition" bugs

...to be avoided with synchronization constructs

example application: ATM server

commands: withdraw, deposit

one correctness goal: don't lose money

ATM server

(pseudocode)

```
ServerLoop() {
    while (true) {
        ReceiveRequest(&operation, &accountNumber, &amount);
        if (operation == DEPOSIT) {
            Deposit(accountNumber, amount);
        } else ...
    }
}
Deposit(accountNumber, amount) {
    account = GetAccount(accountNumber);
    account->balance += amount;
    SaveAccountUpdates(account);
}
```

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

→ many threads all running the server loop

multiple threads

```
main() {
    for (int i = 0; i < NumberOfThreads; ++i) {
        pthread_create(&server_loop_threads[i], NULL,
                       ServerLoop, NULL);
    }
    ...
}

ServerLoop() {
    while (true) {
        ReceiveRequest(&operation, &accountNumber, &amount);
        if (operation == DEPOSIT) {
            Deposit(accountNumber, amount);
        } else ...
    }
}
```

the lost write

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

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

the lost write

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

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

lost write to balance

"winner" of the race

the lost write

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

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

lost write to balance

“winner” of the race

lost track of thread A’s money

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)

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

...and why not 7:

B (start): $y \leftarrow 2 = 0010_{\text{TWO}}$; then y bit 3 ← 0; y bit 2 ← 1; then
A: x ← $110_{\text{TWO}} + 1 = 7$; then
B (finish): y bit 1 ← 0; y bit 0 ← 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:

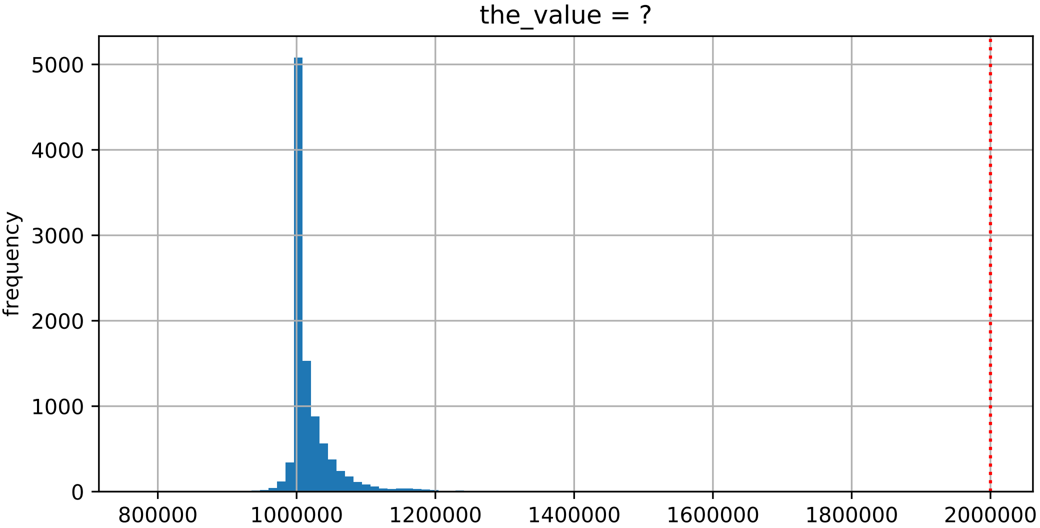
- x86: integer `add` constant to memory location
- many CPUs: loading/storing values that cross cache blocks
  - e.g. if cache blocks `0x40` bytes, load/store 4 byte from addr. `0x3E` is not atomic

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

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);
}
```

```
WaitForOther:
  movl ready, %eax  // eax <- other_ready
.L2:
  testl %eax, %eax
  je .L2                      // while (eax == 0) repeat
  ...
```

compilers move loads/stores (1)

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

```
WaitForOther:
  movl ready, %eax  // eax <- other_ready
.L2:
  testl %eax, %eax
  je .L2                  // while (eax == 0) repeat
  ...
```

compilers move loads/stores (2)

```
void WaitForOther() {
    is_waiting = 1;
    do {} while (!other_ready);
    is_waiting = 0;
}
```

```
WaitForOther:
  // compiler optimization: don't set is_waiting to 1,
  // (why? it will be set to 0 anyway)
  movl other_ready, %eax  // eax <- other_ready
.L2:
  testl %eax, %eax
  je .L2                          // while (eax == 0) repeat
  ...
  movl $0, is_waiting  // is_waiting <- 0
```

compilers move loads/stores (2)

```
void WaitForOther() {
    is_waiting = 1;
    do {} while (!other_ready);
    is_waiting = 0;
}
```

```
WaitForOther:
  // compiler optimization: don't set is_waiting to 1,
  // (why? it will be set to 0 anyway)
  movl other_ready, %eax  // eax <- other_ready
.L2:
  testl %eax, %eax
  je .L2                       // while (eax == 0) repeat
  ...
  movl $0, is_waiting  // is_waiting <- 0
```

compilers move loads/stores (2)

```
void WaitForOther() {
    is_waiting = 1;
    do {} while (!other_ready);
    is_waiting = 0;
}
```

```
WaitForOther:
  // compiler optimization: don't set is_waiting to 1,
  // (why? it will be set to 0 anyway)
  movl other_ready, %eax  // eax <- other_ready
.L2:
  testl %eax, %eax
  je .L2                     // while (eax == 0) repeat
  ...
  movl $0, is_waiting  // 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

some definitions

mutual exclusion: ensuring only one thread does a particular thing at a time

- like checking for and, if needed, buying milk

some definitions

mutual exclusion: ensuring only one thread does a particular thing at a time

- like checking for and, if needed, buying milk

critical section: code that exactly one thread can execute at a time

- result of critical section

some definitions

mutual exclusion: ensuring only one thread does a particular thing at a time

like checking for and, if needed, buying milk

critical section: code that exactly one thread can execute at a time

result of critical section

lock: object only one thread can hold at a time

interface for creating critical sections

lock analogy

agreement: only change account balances while wearing this hat

normally hat kept on table

put on hat when editing balance

hopefully, only one person (= thread) can wear hat a time

need to wait for them to remove hat to put it on

lock analogy

agreement: only change account balances while wearing this hat

normally hat kept on table

put on hat when editing balance

hopefully, only one person (= thread) can wear hat a time

need to wait for them to remove hat to put it on

“lock (or acquire) the lock” = get and put on hat

“unlock (or release) the lock” = put hat back on table

the lock primitive

locks: an object with (at least) two operations:

> *acquire* or *lock* — wait until lock is free, then "grab" it
>
> *release* or *unlock* — let others use lock, wakeup waiters

typical usage: everyone acquires lock before using shared resource

> forget to acquire lock? weird things happen

```
Lock(account_lock);
balance += ...;
Unlock(account_lock);
```

the lock primitive

locks: an object with (at least) two operations:

acquire or *lock* — wait until lock is free, then “grab” it

release or *unlock* — let others use lock, wakeup waiters

typical usage: everyone acquires lock before using shared resource

forget to acquire lock? weird things happen

```
Lock(account_lock);
balance += ...;
Unlock(account_lock);
```

waiting for lock?

when waiting — ideally:

not using processor (at least if waiting a while)

OS can context switch to other programs

pthread mutex

```
#include <pthread.h>

pthread_mutex_t account_lock;
pthread_mutex_init(&account_lock, NULL);
    // or: pthread_mutex_t account_lock =
    //                 PTHREAD_MUTEX_INITIALIZER;
...
pthread_mutex_lock(&account_lock);
balance += ...;
pthread_mutex_unlock(&account_lock);
```

exercise

```
pthread_mutex_t lock1 = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t lock2 = PTHREAD_MUTEX_INITIALIZER;
string one = "init one", two = "init two";
void ThreadA() {
    pthread_mutex_lock(&lock1);
    one = "one in ThreadA";  // (A1)
    pthread_mutex_unlock(&lock1);
    pthread_mutex_lock(&lock2);
    two = "two in ThreadA";  // (A2)
    pthread_mutex_unlock(&lock2);
}
void ThreadB() {
    pthread_mutex_lock(&lock1);
    one = "one in ThreadB";  // (B1)
    pthread_mutex_lock(&lock2);
    two = "two in ThreadB";  // (B2)
    pthread_mutex_unlock(&lock2);
    pthread_mutex_unlock(&lock1);
}
```

exercise (alternate 1)

```
pthread_mutex_t lock1 = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t lock2 = PTHREAD_MUTEX_INITIALIZER;
string one = "init one", two = "init two";
void ThreadA() {
    pthread_mutex_lock(&lock2);
    two = "two in ThreadA";  // (A2)
    pthread_mutex_unlock(&lock2);
    pthread_mutex_lock(&lock1);
    one = "one in ThreadA";  // (A1)
    pthread_mutex_unlock(&lock1);
}
void ThreadB() {
    pthread_mutex_lock(&lock1);
    one = "one in ThreadB";  // (B1)
    pthread_mutex_lock(&lock2);
    two = "two in ThreadB";  // (B2)
    pthread_mutex_unlock(&lock2);
    pthread_mutex_unlock(&lock1);
}
```

exercise (alternate 2)

```
pthread_mutex_t lock1 = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t lock2 = PTHREAD_MUTEX_INITIALIZER;
string one = "init one", two = "init two";
void ThreadA() {
    pthread_mutex_lock(&lock2);
    two = "two in ThreadA";   // (A2)
    pthread_mutex_unlock(&lock2);
    pthread_mutex_lock(&lock1);
    one = "one in ThreadA";   // (A1)
    pthread_mutex_unlock(&lock1);
}
void ThreadB() {
    pthread_mutex_lock(&lock1);
    one = "one in ThreadB";   // (B1)
    pthread_mutex_unlock(&lock1);
    pthread_mutex_lock(&lock2);
    two = "two in ThreadB";   // (B2)
    pthread_mutex_unlock(&lock2);
}
```

POSIX mutex restrictions

pthread_mutex rule: unlock from same thread you lock in

does this actually matter?

depends on how pthread_mutex is implemented

preview: general sync

lots of coordinating threads beyond locks/barriers

will talk about two general tools later:

- monitors/condition variables
- semaphores

big added feature: wait for arbitrary thing to happen

a bad idea

one bad idea to wait for an event:

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER; bool ready = false;
void WaitForReady() {
    pthread_mutex_lock(&lock);
    do {
        pthread_mutex_unlock(&lock);
        /* only time MarkReady() can run */
        pthread_mutex_lock(&lock);
    } while (!ready);
    pthread_mutex_unlock(&lock);
}
void MarkReady() {
    pthread_mutex_lock(&lock);
    ready = true;
    pthread_mutex_unlock(&lock);
}
```

DO NOT USE THIS CODE

wastes processor time; MarkReady can stall waiting for unlock

beyond locks

in practice: want more than locks for synchronization

for waiting for arbtirary events (without CPU-hogging-loop):

- monitors
- semaphores

for common synchornization patterns:

- barriers
- reader-writer locks

higher-level interface:

- transactions

barriers

compute minimum of 100M element array with 2 processors

algorithm:

compute minimum of 50M of the elements on each CPU

one thread for each CPU

wait for all computations to finish

take minimum of all the minimums

barriers

compute minimum of 100M element array with 2 processors

algorithm:

compute minimum of 50M of the elements on each CPU

- one thread for each CPU

wait for all computations to finish

take minimum of all the minimums

barriers API

barrier.Initialize(NumberOfThreads)

barrier.Wait() — return after all threads have waited

idea: multiple threads perform computations in parallel

threads wait for all other threads to call Wait()

barrier: waiting for finish

```
barrier.Initialize(2);
```

Thread 0

```
partial_mins[0] =
    /* min of first
       50M elems */;

barrier.Wait();


total_min = min(
    partial_mins[0],
    partial_mins[1]
);
```

Thread 1

```
partial_mins[1] =
    /* min of last
       50M elems */
barrier.Wait();
```

barriers: reuse

Thread 0

```
results[0][0] = getInitial(0);
barrier.Wait();

results[1][0] =
    computeFrom(
        results[0][0],
        results[0][1]
    );
barrier.Wait();

results[2][0] =
    computeFrom(
        results[1][0],
        results[1][1]
    );
```

Thread 1

```
results[0][1] = getInitial(1);
barrier.Wait();

results[1][1] =
    computeFrom(
        results[0][0],
        results[0][1]
    );
barrier.Wait();

results[2][1] =
    computeFrom(
        results[1][0],
        results[1][1]
    );
```

barriers: reuse

Thread 0

```
results[0][0] = getInitial(0);
barrier.Wait();

results[1][0] =
    computeFrom(
        results[0][0],
        results[0][1]
    );
barrier.Wait();

results[2][0] =
    computeFrom(
        results[1][0],
        results[1][1]
    );
```

Thread 1

```
results[0][1] = getInitial(1);
barrier.Wait();

results[1][1] =
    computeFrom(
        results[0][0],
        results[0][1]
    );
barrier.Wait();

results[2][1] =
    computeFrom(
        results[1][0],
        results[1][1]
    );
```

barriers: reuse

Thread 0

```
results[0][0] = getInitial(0);
barrier.Wait();

results[1][0] =
    computeFrom(
        results[0][0],
        results[0][1]
    );
barrier.Wait();

results[2][0] =
    computeFrom(
        results[1][0],
        results[1][1]
    );
```

Thread 1

```
results[0][1] = getInitial(1);
barrier.Wait();

results[1][1] =
    computeFrom(
        results[0][0],
        results[0][1]
    );
barrier.Wait();

results[2][1] =
    computeFrom(
        results[1][0],
        results[1][1]
    );
```

pthread barriers

```
pthread_barrier_t barrier;
pthread_barrier_init(
    &barrier,
    NULL /* attributes */,
    numberOfThreads
);
...
...
pthread_barrier_wait(&barrier);
```

backup slides