thread 2 / synchronization 1

last time

reasoning about CFS sharing

- intuition: everyone gets equal share, if they can use it
- can't use share? divided up among remaining

multithreaded process

- same files, pid
- same address space (memory)
- newly allocated stack per thread

pthread_create ∼ fork, but run specific function

pthread_join ∼ waitpid

passing values to threads

- global variables, pointer containing something
- can have thread store value somewhere, read it from main thread

sum example (on heap)

```
struct ThreadInfo { pthread_t thread; int *values; int start; int end; int result }
void *sum_thread(void *argument) {
    ...
}

ThreadInfo *start_sum_all(int *values) {
    ThreadInfo *info = new ThreadInfo[2];
    for (int i = 0; i < 2; ++i) {
        info[i].values = values; info[i].start = i*512; info[i].end = (i+1)*512;
        pthread_create(&info[i].thread, NULL, sum_thread, (void *) &info[i]);
    }
    return info;
}

int finish_sum_all(ThreadInfo *info) {
    for (int i = 0; i < 2; ++i)
        pthread_join(info[i].thread, NULL);
    int result = info[0].result + info[1].result;
    delete[] info;
    return result;
}
```

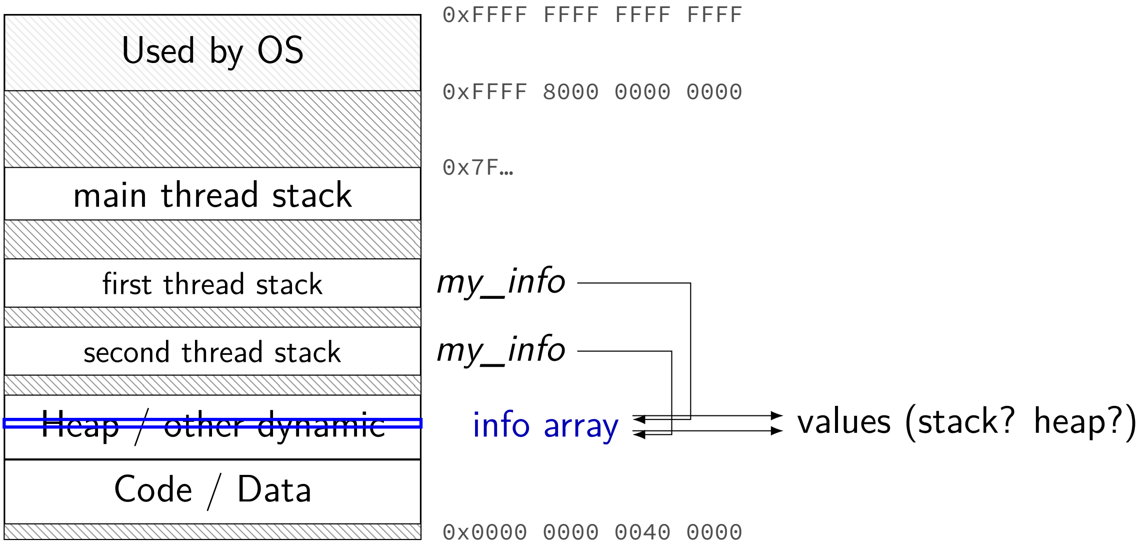
sum example (on heap)

```
struct ThreadInfo { pthread_t thread; int *values; int start; int end; int result
void *sum_thread(void *argument) {
    ...
}

ThreadInfo *start_sum_all(int *values) {
    ThreadInfo *info = new ThreadInfo[2];
    for (int i = 0; i < 2; ++i) {
        info[i].values = values; info[i].start = i*512; info[i].end = (i+1)*512;
        pthread_create(&info[i].thread, NULL, sum_thread, (void *) &info[i]);
    }
    return info;
}

int finish_sum_all(ThreadInfo *info) {
    for (int i = 0; i < 2; ++i)
        pthread_join(info[i].thread, NULL);
    int result = info[0].result + info[1].result;
    delete[] info;
    return result;
}
```

sum example (on heap)

```
struct ThreadInfo { pthread_t thread; int *values; int start; int end; int result
void *sum_thread(void *argument) {
    ...
}

ThreadInfo *start_sum_all(int *values) {
    ThreadInfo *info = new ThreadInfo[2];
    for (int i = 0; i < 2; ++i) {
        info[i].values = values; info[i].start = i*512; info[i].end = (i+1)*512;
        pthread_create(&info[i].thread, NULL, sum_thread, (void *) &info[i]);
    }
    return info;
}

int finish_sum_all(ThreadInfo *info) {
    for (int i = 0; i < 2; ++i)
        pthread_join(info[i].thread, NULL);
    int result = info[0].result + info[1].result;
    delete[] info;
    return result;
}
```

thread_sum memory (heap version)



thread resources

to create a thread, allocate:

new stack (how big???)

thread control block

deallocated when …

thread resources

to create a thread, allocate:

new stack (how big???)

thread control block

deallocated when ...

can deallocate stack when thread exits

but need to allow collecting return value

- same problem as for processes and waitpid

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_other_stuff();
    ...
}
```

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

starting threads detached

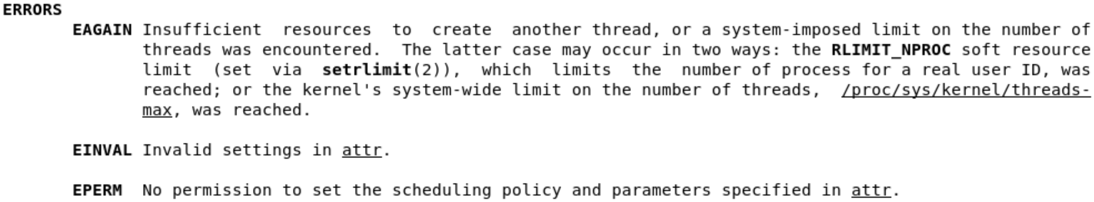
```
void *show_progress(void * ...) { ... }
void spawn_show_progress_thread() {
    pthread_t show_progress_thread;
    pthread_attr_t attrs;
    pthread_attr_init(&attrs);
    pthread_attr_setdetachstate(&attrs, PTHREAD_CREATE_DETACHED);
    pthread_create(&show_progress_thread, attrs,
                   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 note on error checking

from pthread_create manpage:



```
ERRORS
       EAGAIN Insufficient  resources  to  create  another thread, or a system-imposed limit on the number of
              threads was encountered.  The latter case may occur in two ways: the RLIMIT_NPROC soft resource
              limit  (set  via  setrlimit(2)),  which  limits  the  number of process for a real user ID, was
              reached; or the kernel's system-wide limit on the number of threads,  /proc/sys/kernel/threads-
              max, was reached.

       EINVAL Invalid settings in attr.

       EPERM  No permission to set the scheduling policy and parameters specified in attr.
```

special constants for *return value*

same pattern for many other pthreads functions

will often omit error checking in slides for brevity

error checking pthread_create

```
int error = pthread_create(...);
if (error != 0) {
    /* print some error message */
}
```

the correctness problem

schedulers introduce non-determinism

- scheduler might run threads in any order
- scheduler can switch threads at any time

worse with threads on multiple cores

- cores not precisely synchronized (stalling for caches, etc., etc.)
- different cores happen in different order each time

allows for “race condition” bugs

- outcome depends on whether one thread can ‘race’ ahead of another

…to be avoided by synchronization constructs

- what we’ll talk about for a while…

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)

what are some 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)

what are some 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)

what are some 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:

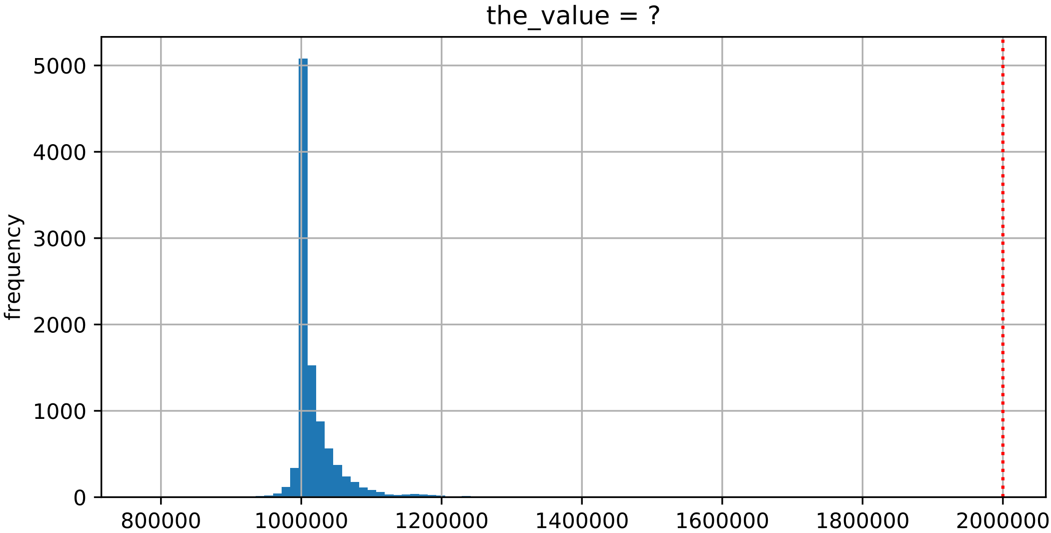
- x86: integer `add` constant to memory location
- many CPUs: loading/storing values that cross cache blocks
  - e.g. if cache blocks `0x40` bytes, load/store 4 byte from addr. `0x3E` is not atomic

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

too much milk

roommates Alice and Bob want to keep fridge stocked with milk:

time	Alice	Bob
3:00	look in fridge. no milk	
3:05	leave for store	
3:10	arrive at store	look in fridge. no milk
3:15	buy milk	leave for store
3:20	return home, put milk in fridge	arrive at store
3:25		buy milk
3:30		return home, put milk in fridge

how can Alice and Bob coordinate better?

too much milk “solution” 1 (algorithm)

leave a note: “I am buying milk”

- place before buying
- remove after buying
- don’t try buying if there’s a note

≈ setting/checking a variable (e.g. “`note = 1`”)

- with atomic load/store of variable

```
if (no milk) {
    if (no note) {
        leave note;
        buy milk;
        remove note;
    }
}
```

too much milk “solution” 1 (algorithm)

leave a note: “I am buying milk”

place before buying
remove after buying
don’t try buying if there’s a note

≈ setting/checking a variable (e.g. “`note = 1`”)

with atomic load/store of variable

```
if (no milk) {
    if (no note) {
        leave note;
        buy milk;
        remove note;
    }
}
```

exercise: why doesn’t this work?

too much milk “solution” 1 (timeline)

```
       Alice                            Bob
if (no milk) {
    if (no note) {
                           if (no milk) {
                               if (no note) {
        leave note;
        buy milk;
        remove note;
    }
}
                                   leave note;
                                   buy milk;
                                   remove note;
                               }
                           }
```

too much milk “solution” 2 (algorithm)

intuition: leave note when buying or checking if need to buy

```
leave note;
if (no milk) {
    if (no note) {
        buy milk;
    }
}
remove note;
```

too much milk: “solution” 2 (timeline)

Alice

```
leave note;
if (no milk) {
    if (no note) {
        buy milk;
    }
}
remove note;
```

too much milk: “solution” 2 (timeline)

Alice

```
leave note;
if (no milk) {
    if (no note) {
        buy milk;
    }
}
remove note;
```

but there’s always a note

too much milk: “solution” 2 (timeline)

Alice

```
leave note;
if (no milk) {
    if (no note) {
        ~~buy milk~~;
    }
}
remove note;
```

but there’s always a note

…will never buy milk (twice *or* once)

“solution” 3: algorithm

intuition: label notes so Alice knows which is hers (and vice-versa)

computer equivalent: separate noteFromAlice and noteFromBob variables

Alice

```
leave note from Alice;
if (no milk) {
    if (no note from Bob) {
        buy milk
    }
}
remove note from Alice;
```

Bob

```
leave note from Bob;
if (no milk) {
    if (no note from Alice)
        buy milk
    }
}
remove note from Bob;
```

too much milk: “solution” 3 (timeline)

```
         Alice                                 Bob
leave note from Alice
if (no milk) {
                              leave note from Bob
    if (no note from Bob) {
        ~~buy milk~~
    }
}
                              if (no milk) {
                                  if (no note from Alice) {
                                      ~~buy milk~~
                                  }
                              }
                              remove note from Bob
remove note from Alice
```

too much milk: is it possible

is there a solutions with writing/reading notes?

    ≈ loading/storing from shared memory

yes, but it's not very elegant

too much milk: solution 4 (algorithm)

Alice

```
leave note from Alice
while (note from Bob) {
    do nothing
}
if (no milk) {
    buy milk
}
remove note from Alice
```

Bob

```
leave note from Bob
if (no note from Alice) {
    if (no milk) {
        buy milk
    }
}
remove note from Bob
```

too much milk: solution 4 (algorithm)

Alice

```
leave note from Alice
while (note from Bob) {
    do nothing
}
if (no milk) {
    buy milk
}
remove note from Alice
```

Bob

```
leave note from Bob
if (no note from Alice) {
    if (no milk) {
        buy milk
    }
}
remove note from Bob
```

exercise (hard): prove (in)correctness

too much milk: solution 4 (algorithm)

Alice

```
leave note from Alice
while (note from Bob) {
    do nothing
}
if (no milk) {
    buy milk
}
remove note from Alice
```

Bob

```
leave note from Bob
if (no note from Alice) {
    if (no milk) {
        buy milk
    }
}
remove note from Bob
```

exercise (hard): prove (in)correctness

too much milk: solution 4 (algorithm)

Alice

```
leave note from Alice
while (note from Bob) {
    do nothing
}
if (no milk) {
    buy milk
}
remove note from Alice
```

Bob

```
leave note from Bob
if (no note from Alice) {
    if (no milk) {
        buy milk
    }
}
remove note from Bob
```

exercise (hard): prove (in)correctness

exercise (hard): extend to three people

Peterson’s algorithm

general version of solution

see, e.g., Wikipedia

we’ll use special hardware support instead

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

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(MilkLock);
if (no milk) {
    buy milk
}
Unlock(MilkLock);
```

pthread mutex

```
#include <pthread.h>

pthread_mutex_t MilkLock;
pthread_mutex_init(&MilkLock, NULL);
    // or: pthread_mutex_t MilkLock =
    //                  PTHREAD_MUTEX_INITIALIZER;
...
pthread_mutex_lock(&MilkLock);
if (no milk) {
    buy milk
}
pthread_mutex_unlock(&MilkLock);
```

xv6 spinlocks

```
#include "spinlock.h"
...
struct spinlock MilkLock;
initlock(&MilkLock, "name for debugging");
...
acquire(&MilkLock);
if (no milk) {
    buy milk
}
release(&MilkLock);
```

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);
}
```

possible values of one/two after A+B run?

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);
}
```

possible values of one/two after A+B run?

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);
}
```

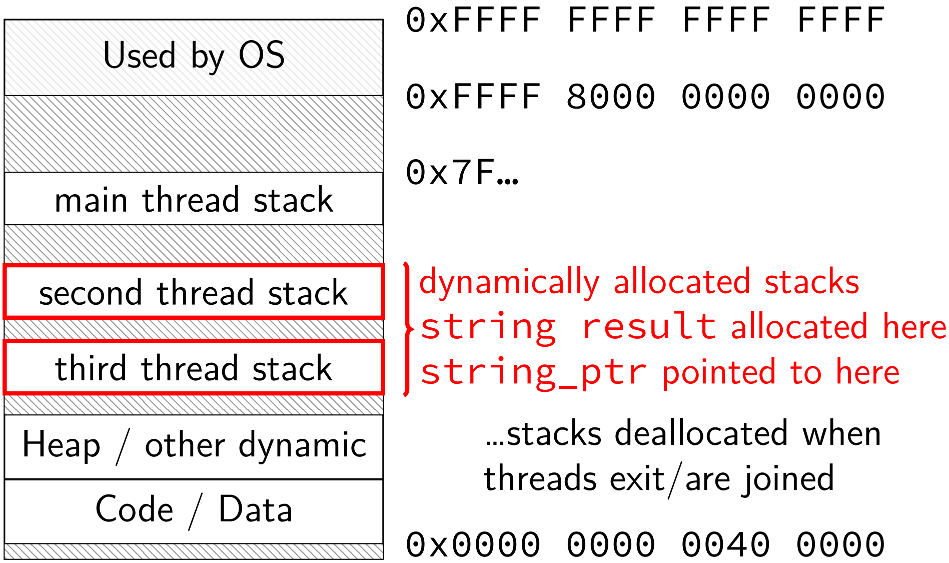
possible values of one/two after A+B run?

backup slides

what's wrong with this?

```
/* omitted: headers */
#include <string>
using std::string;
void *create_string(void *ignored_argument) {
  string result;
  result = ComputeString();
  return &result;
}
int main() {
  pthread_t the_thread;
  pthread_create(&the_thread, NULL, create_string, NULL);
  string *string_ptr;
  pthread_join(the_thread, (void*) &string_ptr);
  cout << "string is " << *string_ptr;
}
```

program memory



0xFFFF FFFF FFFF FFFF

Used by OS

0xFFFF 8000 0000 0000

0x7F...

main thread stack

second thread stack

third thread stack

dynamically allocated stacks

`string result` allocated here

`string_ptr` pointed to here

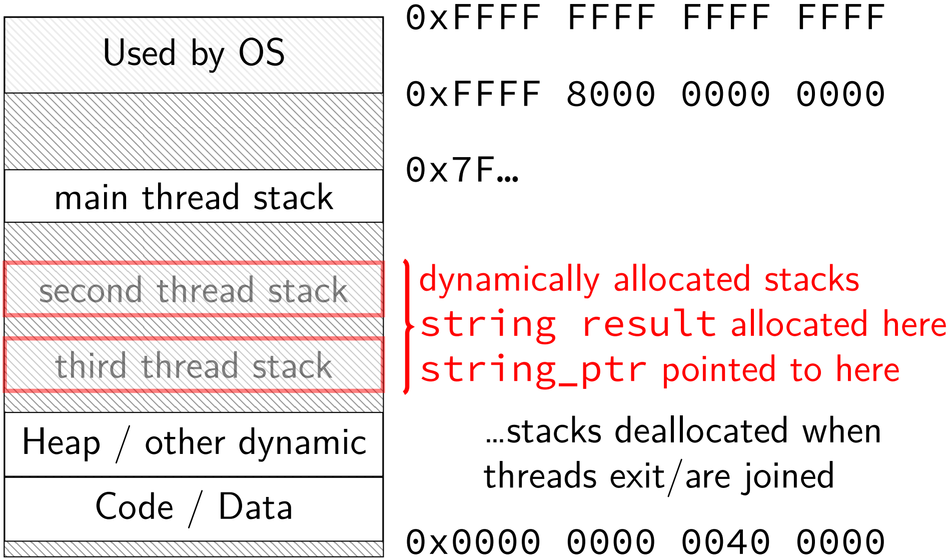
...stacks deallocated when threads exit/are joined

Heap / other dynamic

Code / Data

0x0000 0000 0040 0000

program memory



dynamically allocated stacks

`string result` allocated here

`string_ptr` pointed to here

...stacks deallocated when threads exit/are joined

sum example (to main stack)

```
struct ThreadInfo { int *values; int start; int end; int result };
void *sum_thread(void *argument) {
    ThreadInfo *my_info = (ThreadInfo *) argument;
    int sum = 0;
    for (int i = my_info->start; i < my_info->end; ++i) {
        sum += my_info->values[i];
    }
    my_info->result = sum;
    return NULL;

}
int sum_all(int *values) {
    ThreadInfo info[2]; pthread_t thread[2];
    for (int i = 0; i < 2; ++i) {
        info[i].values = values; info[i].start = i*512; info[i].end = (i+1)*512;
        pthread_create(&threads[i], NULL, sum_thread, (void *) &info[i]);
    }
    for (int i = 0; i < 2; ++i)
        pthread_join(threads[i], NULL);
    return info[0].result + info[1].result;
}
```

sum example (to main stack)

```
struct ThreadInfo { int *values; int start; int end; int result };
void *sum_thread(void *argument) {
    ThreadInfo *my_info = (ThreadInfo *) argument;
    int sum = 0;
    for (int i = my_info->start; i < my_info->end; ++i) {
        sum += my_info->values[i];
    }
    my_info->result = sum;
    return NULL;

}
int sum_all(int *values) {
    ThreadInfo info[2]; pthread_t thread[2];
    for (int i = 0; i < 2; ++i) {
        info[i].values = values; info[i].start = i*512; info[i].end = (i+1)*512;
        pthread_create(&threads[i], NULL, sum_thread, (void *) &info[i]);
    }
    for (int i = 0; i < 2; ++i)
        pthread_join(threads[i], NULL);
    return info[0].result + info[1].result;
}
```

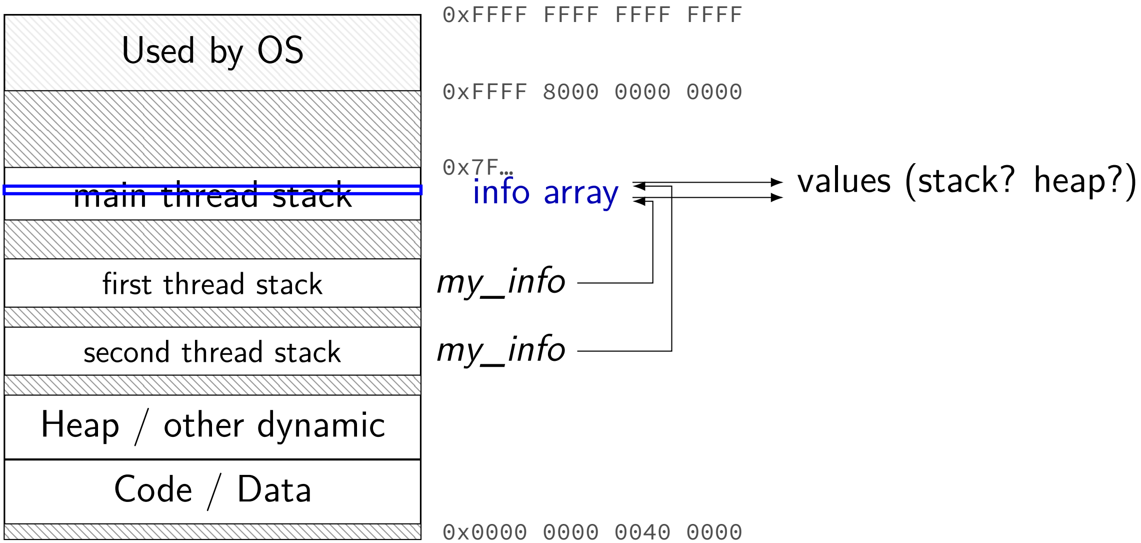
sum example (to main stack)

```
struct ThreadInfo { int *values; int start; int end; int result };
void *sum_thread(void *argument) {
    ThreadInfo *my_info = (ThreadInfo *) argument;
    int sum = 0;
    for (int i = my_info->start; i < my_info->end; ++i) {
        sum += my_info->values[i];
    }
    my_info->result = sum;
    return NULL;

}
int sum_all(int *values) {
    ThreadInfo info[2]; pthread_t thread[2];
    for (int i = 0; i < 2; ++i) {
        info[i].values = values; info[i].start = i*512; info[i].end = (i+1)*512;
        pthread_create(&threads[i], NULL, sum_thread, (void *) &info[i]);
    }
    for (int i = 0; i < 2; ++i)
        pthread_join(threads[i], NULL);
    return info[0].result + info[1].result;
}
```

sum example (to main stack)

```
struct ThreadInfo { int *values; int start; int end; int result };
void *sum_thread(void *argument) {
    ThreadInfo *my_info = (ThreadInfo *) argument;
    int sum = 0;
    for (int i = my_info->start; i < my_info->end; ++i) {
        sum += my_info->values[i];
    }
    my_info->result = sum;
    return NULL;

}
int sum_all(int *values) {
    ThreadInfo info[2]; pthread_t thread[2];
    for (int i = 0; i < 2; ++i) {
        info[i].values = values; info[i].start = i*512; info[i].end = (i+1)*512;
        pthread_create(&threads[i], NULL, sum_thread, (void *) &info[i]);
    }
    for (int i = 0; i < 2; ++i)
        pthread_join(threads[i], NULL);
    return info[0].result + info[1].result;
}
```

program memory (to main stack)


Used by OS
main thread stack
first thread stack
second thread stack
Heap / other dynamic
Code / Data
0xFFFF FFFF FFFF FFFF
0xFFFF 8000 0000 0000
0x7F…
0x0000 0000 0040 0000
info array
values (stack? heap?)
my_info
my_info