Synchronization 3: Barriers (con't) / Semaphores / Monitors

Changelog

Changes made in this version not seen in first lecture:

- 27 September: counting resources: use consistent colors for up/down
- 27 September: counting semaphores with binary semaphores: add missing mutex unlock, correct function names

last time

compiler reordering — optimizer assumes no threads

- unless you use something special to tell it

cache coherency — keeping caches in sync

- communicate via shared bus
- ensure at most one writing cache at a time

hardware support for locking

- atomic read-modify-write
- and instructions to prevent reordering

mutexes: locks with wait queues

- give up CPU instead of “spinning”

quiz question: code snippet

```
node *head = NULL;
void prepend(int new_value) {
    node *new_head = new node;
    new_head->value = new_value;
    new_head->next = head;
    head = new_head;
}
```

`new_head`: on stack (local to thread)

`new_head->next`: on heap (unique for each call)

`head`: in global data area

`head = new_head`: ***pointer*** assignment (doesn't dereference pointers)

quiz question: interleaving

```
void prepend(int new_value) {
    node *new_head = new node;
    new_head->value = new_value;
    new_head->next = head;
    ...
    ...
    ...
    ...
    ...
    head = new_head;
}
```

```
void prepend(int new_value) {
    ...
    ...
    ...
    node *new_head = new node;
    new_head->value = new_value;
    new_head->next = head;
    head = new_head;
}
```

example 2: parallel processing

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

example 2: parallel processing

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

barriers are reusable:

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

barriers are reusable:

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

barriers are reusable:

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);
```

life homework (pseudocode)

```
for (int time = 0; time < MAX_ITERATIONS; ++time) {
    for (int y = 0; y < size; ++y) {
        for (int x = 0; x < size; ++x) {
            to_grid(x, y) = computeValue(from_grid, x, y);
        }
    }
    swap(from_grid, to_grid);
}
```

life homework

compute grid of values for time t from grid for time $t-1$

- compute new value at i, j based on surrounding values

parallel version: produce parts of grid in different threads

use barriers to finish time t before going to time $t+1$

- avoid trying to read things that aren't computed

life homework even/odd

naive way has an operation that needs locking:

```
for (int time = 0; time < MAX_ITERATIONS; ++time) {
    ... compute to_grid ...
    swap(from_grid, to_grid);
}
```

but this alternative needs less locking:

```
Grid grids[2];
for (int time = 0; time < MAX_ITERATIONS; ++time) {
    from_grid = &grids[time % 2];
    to_grid = &grids[time % 2 + 1];
    ... compute to_grid ...
}
```

life homework even/odd

naive way has an operation that needs locking:

```
for (int time = 0; time < MAX_ITERATIONS; ++time) {
    ... compute to_grid ...
    swap(from_grid, to_grid);
}
```

but this alternative needs less locking:

```
Grid grids[2];
for (int time = 0; time < MAX_ITERATIONS; ++time) {
    from_grid = &grids[time % 2];
    to_grid = &grids[time % 2 + 1];
    ... compute to_grid ...
}
```

generalizing locks

barriers are very useful

do things locks can't do

but can't do things locks can do

semaphores and condition variables are more general

can implement locks *and* barriers *and* ...

generalizing locks: semaphores

semaphore has a non-negative integer **value** and two operations:

P() or **down** or **wait**:
wait for semaphore to become positive (> 0),
then decerement by 1

V() or **up** or **signal** or **post**:
increment semaphore by 1 (waking up thread if needed)

P, V from Dutch: *proberen* (test), *verhogen* (increment)

semaphores are kinda integers

semaphore like an integer, but...

cannot read/write directly

- down/up operaion only way to access (typically)
- exception: initialization

never negative — wait instead

- down operation wants to make negative? thread waits

reserving books

suppose tracking copies of library book...

```
Semaphore free_copies = Semaphore(3);
void ReserveBook() {
    // wait for copy to be free
    free_copies.down();
    ... // ... then take reserved copy
}

void ReturnBook() {
    ... // return reserved copy
    free_copies.up();
    // ... then wakeup waiting thread
}
```

counting resources: reserving books

suppose tracking copies of same library book
non-negative integer count = # how many books used?
up = give back book; down = take book

Copy 1
Copy 2
Copy 3

free copies 3

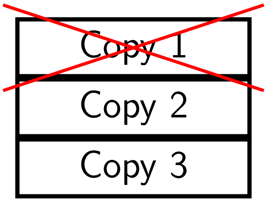
counting resources: reserving books

suppose tracking copies of same library book

non-negative integer count = # how many books used?

up = give back book; down = take book

taken out



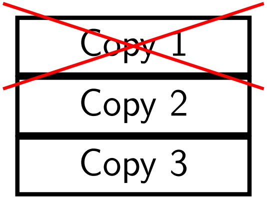
free copies ~~3~~ 2

after calling down to reserve

counting resources: reserving books

suppose tracking copies of same library book
non-negative integer count = # how many books used?
up = give back book; down = take book

taken out



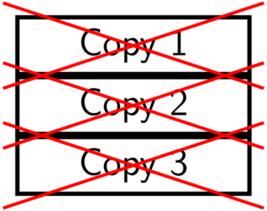
free copies 2

after calling down to reserve

counting resources: reserving books

suppose tracking copies of same library book
non-negative integer count = # how many books used?
up = give back book; down = take book

taken out
taken out
taken out

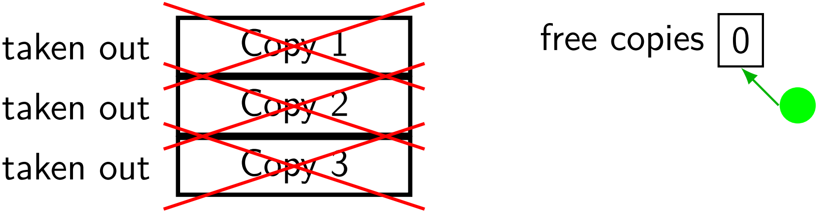


free copies 0

after calling down three times
to reserve all copies

counting resources: reserving books

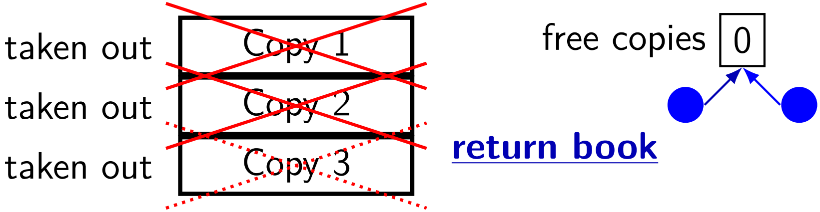
suppose tracking copies of same library book
non-negative integer count = # how many books used?
up = give back book; down = take book



reserve book
call *down* again
start waiting...

counting resources: reserving books

suppose tracking copies of same library book
non-negative integer count = # how many books used?
up = give back book; down = take book



return book

call *up*
release waiter

reserve book

call *down*
~~waiting~~
done waiting

implementing mutexes with semaphores

```
struct Mutex {
    Semaphore s; /* with inital value 1 */
    /* value = 1 --> mutex if free */
    /* value = 0 --> mutex is busy */
}

MutexLock(Mutex *m) {
    m->s.down();
}
MutexUnlock(Mutex *m) {
    m->s.up();
}
```

implementing join with semaphores

```
struct Thread {
    ...
    Semaphore finish_semaphore; /* with initial value 0 */
    /* value = 0: either thread not finished OR already joined */
    /* value = 1: thread finished AND not joined */
};
thread_join(Thread *t) {
    t->finish_semaphore->down();
}

/* assume called when thread finishes */
thread_exit(Thread *t) {
    t->finish_semaphore->up();
    /* tricky part: deallocating struct Thread safely? */
}
```

POSIX semaphores

```
#include <semaphore.h>
...
sem_t my_semaphore;
int process_shared = /* 1 if sharing between processes */;
sem_init(&my_semaphore, process_shared, initial_value);
...
sem_wait(&my_semaphore);  /* down */
sem_post(&my_semaphore);  /* up */
...
sem_destroy(&my_semaphore);
```

semaphore intuition

What do you need to wait for?

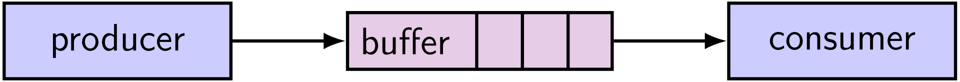
- critical section to be finished
- queue to be non-empty
- array to have space for new items

what can you count that will be 0 when you need to wait?

- # of threads that can start critical section now
- # of threads that can join another thread without waiting
- # of items in queue
- # of empty spaces in array

use up/down operations to maintain count

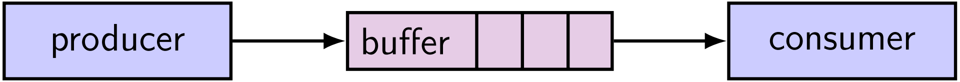
example: producer/consumer



shared buffer (queue) of fixed size

- one or more producers inserts into queue
- one or more consumers removes from queue

example: producer/consumer



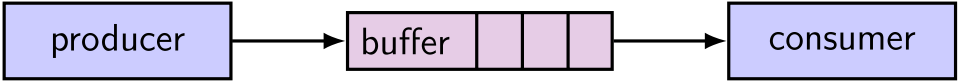
shared buffer (queue) of fixed size

- one or more producers inserts into queue
- one or more consumers removes from queue

producer(s) and consumer(s) don't work in lockstep

- (might need to wait for each other to catch up)

example: producer/consumer



shared buffer (queue) of fixed size

- one or more producers inserts into queue
- one or more consumers removes from queue

producer(s) and consumer(s) don't work in lockstep

- (might need to wait for each other to catch up)

example: C compiler

- preprocessor → compiler → assembler → linker

producer/consumer constraints

consumer waits for producer(s) if buffer is empty

producer waits for consumer(s) if buffer is full

any thread waits while a thread is manipulating the buffer

producer/consumer constraints

consumer waits for producer(s) if buffer is empty

producer waits for consumer(s) if buffer is full

any thread waits while a thread is manipulating the buffer

one semaphore per constraint:

```
sem_t full_slots;   // consumer waits if empty
sem_t empty_slots;  // producer waits if full
sem_t mutex;        // either waits if anyone changing buffer
FixedSizedQueue buffer;
```

producer/consumer pseudocode

```
sem_init(&full_slots, ..., 0 /* # buffer slots initially used */);
sem_init(&empty_slots, ..., BUFFER_CAPACITY);
sem_init(&mutex, ..., 1 /* # thread that can use buffer at once */);
buffer.set_size(BUFFER_CAPACITY);
...

Produce(item) {
    sem_wait(&empty_slots);   // wait until free slot, reserve it
    sem_wait(&mutex);
    buffer.enqueue(item);
    sem_post(&mutex);
    sem_post(&full_slots);  // tell consumers there is more data
}

Consume() {
    sem_wait(&full_slots);   // wait until queued item, reserve it
    sem_wait(&mutex);
    item = buffer.dequeue();
    sem_post(&mutex);
    sem_post(&empty_slots);  // let producer reuse item slot
    return item;
}
```

producer/consumer pseudocode

```
sem_init(&full_slots, ..., 0 /* # buffer slots initially used */);
sem_init(&empty_slots, ..., BUFFER_CAPACITY);
sem_init(&mutex, ..., 1 /* # thread that can use buffer at once */);
buffer.set_size(BUFFER_CAPACITY);
...

Produce(item) {
    sem_wait(&empty_slots);   // wait until free slot, reserve it
    sem_wait(&mutex);
    buffer.enqueue(item);
    sem_post(&mutex);
    sem_post(&full_slots);  // tell consumers there is more data
}

Consume() {
    sem_wait(&full_slots);   // wait until queued item, reserve it
    sem_wait(&mutex);
    item = buffer.dequeue();
    sem_post(&mutex);
    sem_post(&empty_slots);  // let producer reuse item slot
    return item;
}
```

producer/consumer pseudocode

```
sem_init(&full_slots, ..., 0 /* # buffer slots initially used */);
sem_init(&empty_slots, ..., BUFFER_CAPACITY);
sem_init(&mutex, ..., 1 /* # thread that can use buffer at once */);
buffer.set_size(BUFFER_CAPACITY);
...

Produce(item) {
    sem_wait(&empty_slots);   // wait until free slot, reserve it
    sem_wait(&mutex);
    buffer.enqueue(item);
    sem_post(&mutex);
    sem_post(&full_slots);  // tell consumers there is more data
}

Consume() {
    sem_wait(&full_slots);   // wait until queued item, reserve it
    sem_wait(&mutex);
    item = buffer.dequeue();
    sem_post(&mutex);
    sem_post(&empty_slots);  // let producer reuse item slot
    return item;
}
```

producer/consumer pseudocode

```
sem_init(&full_slots, ..., 0 /* # buffer slots initially used */);
sem_init(&empty_slots, ..., BUFFER_CAPACITY);
sem_init(&mutex, ..., 1 /* # thread that can use buffer at once */);
buffer.set_size(BUFFER_CAPACITY);
...

Produce(item) {
    sem_wait(&empty_slots);    // wait until free slot, reserve it
    sem_wait(&mutex);
    buffer.enqueue(item);
    sem_post(&mutex);
    sem_post(&full_slots);                                   re data
}

Consume() {
    sem_wait(&full_slots);    // wait until queued item, reserve it
    sem_wait(&mutex);
    item = buffer.dequeue();
    sem_post(&mutex);
    sem_post(&empty_slots);   // let producer reuse item slot
    return item;
}
```

Can we do

```
sem_wait(&mutex);
sem_wait(&empty_slots);
```

instead?

producer/consumer pseudocode

```
sem_init(&full_slots, ..., 0 /* # buffer slots initially used */);
sem_init(&empty_slots, ..., BUFFER_CAPACITY);
sem_init(&mutex, ..., 1 /* # thread that can use buffer at once */);
buffer.set_size(BUFFER_CAPACITY);
...

Produce(item) {
    sem_wait(&empty_slots);    // wait until free slot, reserve it
    sem_wait(&mutex);
    buffer.enqueue(item);
    sem_post(&mutex);
    sem_post(&full_slots);                                re data
}

Consume() {
    sem_wait(&full_slots);
    sem_wait(&mutex);
    item = buffer.dequeue()
    sem_post(&mutex);
    sem_post(&empty_slots);
    return item;
}
```

Can we do
```
sem_wait(&mutex);
sem_wait(&empty_slots);
```
instead?

No. Consumer waits on `sem_wait(&mutex)` so can't `sem_post(&empty_slots)` (result: producer waits forever problem called *deadlock*)

producer/consumer: cannot reorder mutex/empty

```
ProducerReordered() {
  // BROKEN: WRONG ORDER
  sem_wait(&mutex);
  sem_wait(&empty_slots);

  ...

  sem_post(&mutex);
```

```
Consumer() {
  sem_wait(&full_slots);

  // can't finish until
  // Producer's sem_post(&mutex):
  sem_wait(&mutex);

  ...

  // so this is not reached
  sem_post(&full_slots);
```

producer/consumer pseudocode

```
sem_init(&full_slots, ..., 0 /* # buffer slots initially used */);
sem_init(&empty_slots, ..., BUFFER_CAPACITY);
sem_init(&mutex, ..., 1 /* # thread that can use buffer at once */);
buffer.set_size(BUFFER_CAPACITY);
...

Produce(item) {
    sem_wait(&empty_slots);   // wait until free slot, reserve it
    sem_wait(&mutex);
    buffer.enqueue(item);
    sem_post(&mutex);
    sem_post(&full_slots                             s more data
}

Consume() {
    sem_wait(&full_slots                             m, reserve it
    sem_wait(&mutex);
    item = buffer.dequeu
    sem_post(&mutex);
    sem_post(&empty_slots);  // let producer reuse item slot
    return item;
}
```

Can we do
```
sem_post(&full_slots);
sem_post(&mutex);
```
instead?

Yes — post never waits

producer/consumer summary

producer: wait (down) empty_slots, post (up) full_slots

consumer: wait (down) full_slots, post (up) empty_slots

two producers or consumers?

- still works!

binary semaphores

binary semaphores — semaphores that are only zero or one

as powerful as normal semaphores

- exercise: simulate counting semaphores with binary semaphores (more than one) and an integer

counting semaphores with binary semaphores

```
// assuming initialValue > 0
BinarySemaphore mutex(1);
int value = initialValue;
BinarySemaphore gate(1);

void Down() {
  mutex.Down();
  value -= 1;
  if (val == 0) {
    mutex.Up();
    // decremented to 0, wait
    gate.Down();
    mutex.Down();
  }
  mutex.Up();
}

void Up() {
  mutex.Down();
  value += 1;
  if (value == 1) {
    // value was 0, start a waiter
    gate.Up();
  }
  mutex.Up();
}
```

Anderson-Dahlin and semaphores

Anderson/Dahlin complains about semaphores

> "Our view is that programming with locks and condition variables is superior to programming with semaphores."

argument 1: clearer to have separate constructs for

- waiting for condition to be come true, and
- allowing only one thread to manipulate a thing at a time

arugment 2: tricky to verify thread calls up exactly once for every down

- alternatives allow one to be sloppier (in a sense)

monitors/condition variables

locks for mutual exclusion

condition variables for waiting for event

- operations: wait (for event); signal/broadcast (that event happened)

related data structures

monitor = lock + 0 or more condition variables + shared data

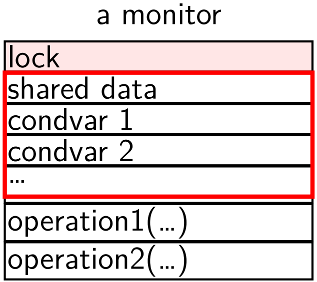
- Java: every object is a monitor (has instance variables, built-in lock, cond. var)
- pthreads: build your own: provides you locks + condition variables

monitor idea

a monitor

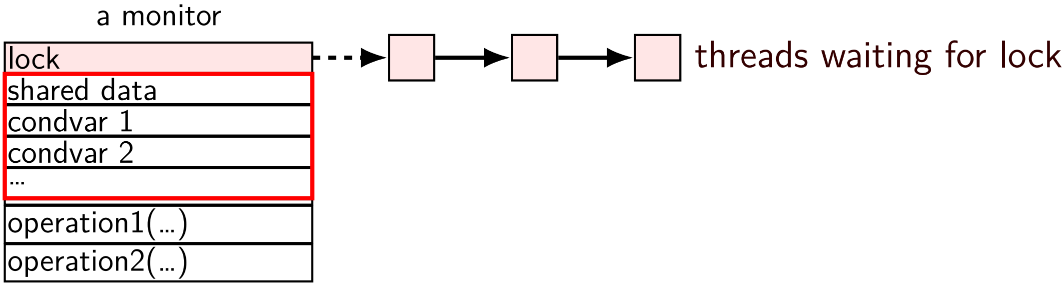
lock
shared data
condvar 1
condvar 2
...
operation1(...)
operation2(...)

monitor idea

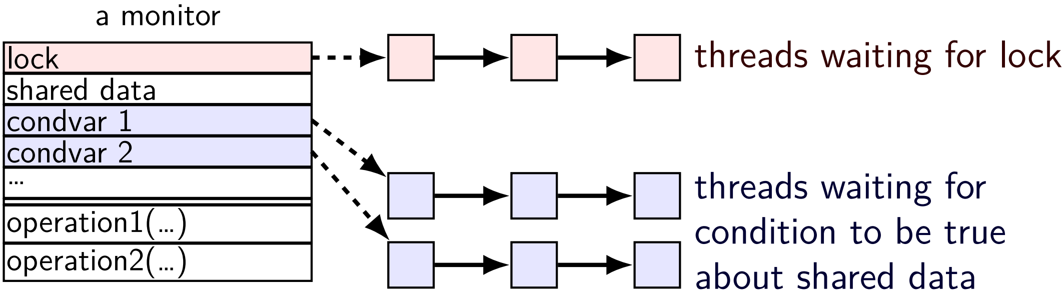


lock must be acquired
before accessing
any part of monitor's stuff

monitor idea

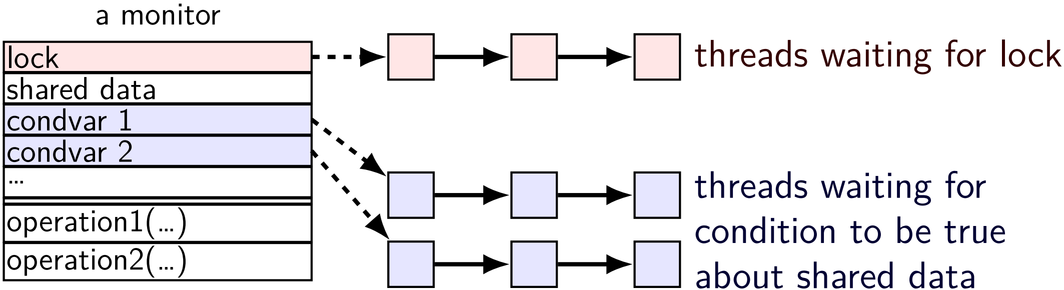


monitor idea



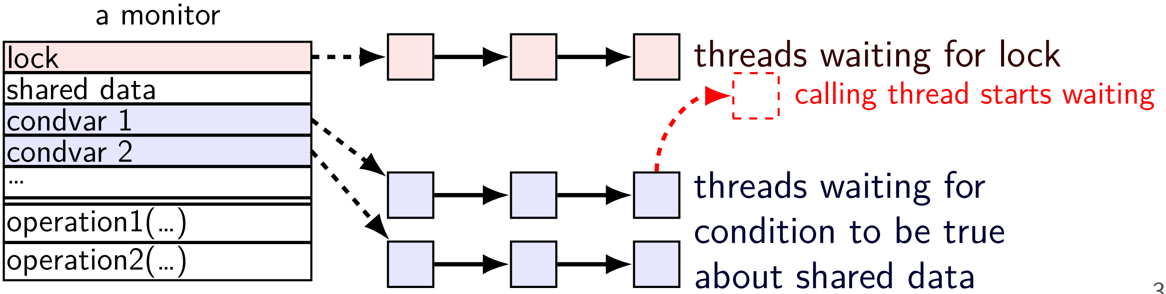
condvar operations

condvar operations:
Wait(cv, lock) — unlock lock, add current thread to cv queue
…and reacquire lock before returning
Broadcast(cv) — remove all from condvar queue
Signal(cv) — remove one from condvar queue



condvar operations

condvar operations:
Wait(cv, lock) — unlock lock, add current thread to cv queue
...and reacquire lock before returning
Broadcast(cv) — remove all from condvar queue
Signal(cv) — remove one from condvar queue



condvar operations

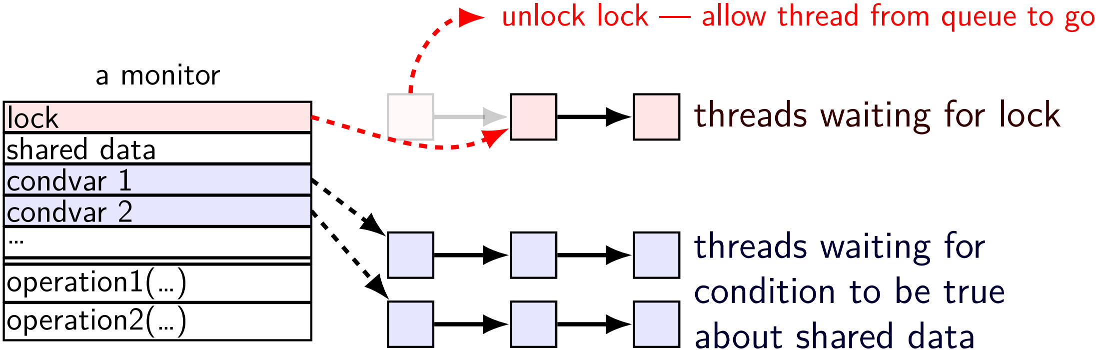
condvar operations:

Wait(cv, lock) — unlock lock, add current thread to cv queue

...and reacquire lock before returning

Broadcast(cv) — remove all from condvar queue

Signal(cv) — remove one from condvar queue



condvar operations

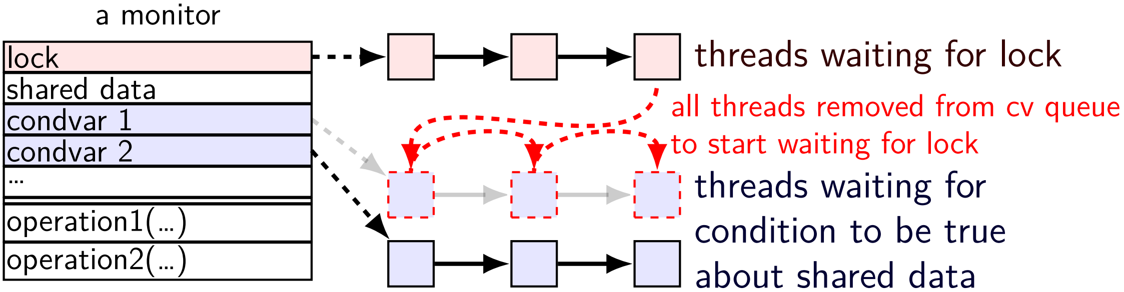
condvar operations:

Wait(cv, lock) — unlock lock, add current thread to cv queue

…and reacquire lock before returning

Broadcast(cv) — remove all from condvar queue

Signal(cv) — remove one from condvar queue



condvar operations

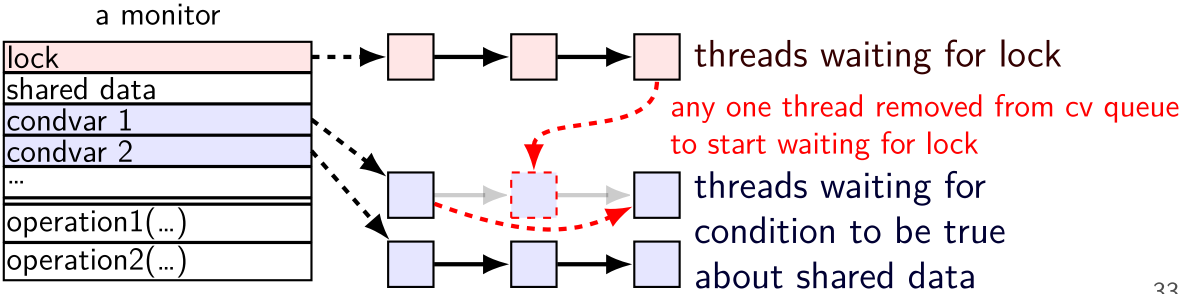
condvar operations:

Wait(cv, lock) — unlock lock, add current thread to cv queue

...and reacquire lock before returning

Broadcast(cv) — remove all from condvar queue

Signal(cv) — remove one from condvar queue



pthread cv usage

```
// MISSING: init calls, etc.
pthread_mutex_t lock;
bool finished;   // data, only accessed with after acquiring lock
pthread_cond_t finished_cv;  // to wait for 'finished' to be true

void WaitForFinished() {
  pthread_mutex_lock(&lock);
  while (!finished) {
    pthread_cond_wait(&finished_cv, &lock);
  }
  pthread_mutex_unlock(&lock);
}

void Finish() {
  pthread_mutex_lock(&lock);
  finished = true;
  pthread_cond_broadcast(&finished_cv);
  pthread_mutex_unlock(&lock);
}
```

pthread cv usage

```
// MISSING: init calls, etc.
pthread_mutex_t lock;
bool finished;    // data, only accessed with after acquiring lock
pthread_cond_t finished_cv;  // to wait for 'finished' to be true

void WaitForFinished() {
  pthread_mutex_lock(&lock);
  while (!finished) {
    pthread_cond_wait(&finished_cv, &lock);
  }
  pthread_mutex_unlock(&lock);
}

void Finish() {
  pthread_mutex_lock(&lock);
  finished = true;
  pthread_cond_broadcast(&finished_cv);
  pthread_mutex_unlock(&lock);
}
```

acquire lock before reading or writing `finished`

pthread cv usage

```
// MISSING: init calls, etc.
pthread_mutex_t lock;
bool finished;   // data, only accessed with after acquiring lock
pthread_cond_t finished_cv;  // to wait for 'finished' to be true

void WaitForFinished() {
  pthread_mutex_lock(&lock);
  while (!finished) {
    pthread_cond_wait(&finished_cv, &lock);
  }
  pthread_mutex_unlock(&lock);
}

void Finish() {
  pthread_mutex_lock(&lock);
  finished = true;
  pthread_cond_broadcast(&finished_cv);
  pthread_mutex_unlock(&lock);
}
```

check whether we need to wait at all

(why a loop? we'll explain later)

pthread cv usage

```
// MISSING: init calls, etc.
pthread_mutex_t lock;
bool finished;   // data, only accessed with after acquiring lock
pthread_cond_t finished_cv;  // to wait for 'finished' to be true

void WaitForFinished() {
  pthread_mutex_lock(&lock);
  while (!finished) {
    pthread_cond_wait(&finished_cv, &lock);
  }
  pthread_mutex_unlock(&lock);
}

void Finish() {
  pthread_mutex_lock(&lock);
  finished = true;
  pthread_cond_broadcast(&finished_cv);
  pthread_mutex_unlock(&lock);
}
```

know we need to wait
(finished can't change while we have lock)
so wait, releasing lock...

pthread cv usage

```
// MISSING: init calls, etc.
pthread_mutex_t lock;
bool finished;   // data, only accessed with after acquiring lock
pthread_cond_t finished_cv;  // to wait for 'finished' to be true

void WaitForFinished() {
  pthread_mutex_lock(&lock);
  while (!finished) {
    pthread_cond_wait(&finished_cv, &lock);
  }
  pthread_mutex_unlock(&lock);
}

void Finish() {
  pthread_mutex_lock(&lock);
  finished = true;
  pthread_cond_broadcast(&finished_cv);
  pthread_mutex_unlock(&lock);
}
```

allow all waiters to proceed (once we unlock the lock)

WaitForFinish timeline 1

WaitForFinish thread	Finish thread
`mutex_lock(&lock)` (thread has lock)	
	`mutex_lock(&lock)` (start waiting for lock)

`while (!finished) ...`
`cond_wait(&finished_cv, &lock);`
(start waiting for cv)
	(done waiting for lock)

	`finished = true` `cond_broadcast(&finished_cv)`
(done waiting for cv) (start waiting for lock)	
	`mutex_unlock(&lock)`

(done waiting for lock)
`while (!finished) ...`
(finished now true, so return)
`mutex_unlock(&lock)`	

WaitForFinish timeline 2

WaitForFinish thread	Finish thread

	`mutex_lock(&lock)`
	`finished = true`
	`cond_broadcast(&finished_cv)`
	`mutex_unlock(&lock)`

`mutex_lock(&lock)`
`while (!finished) ...`
(finished now true, so return)
`mutex_unlock(&lock)`	

why the loop

```
while (!finished) {
  pthread_cond_wait(&finished_cv, &lock);
}
```

we only `broadcast` if `finished` is true

so why check `finished` afterwards?

why the loop

```
while (!finished) {
  pthread_cond_wait(&finished_cv, &lock);
}
```

we only `broadcast` if `finished` is true

so why check `finished` afterwards?

pthread_cond_wait manual page:

> "Spurious wakeups ... may occur."

spurious wakeup = `wait` returns even though nothing happened

why spurious wakeups?

makes implementing condition variables simpler

why spurious wakeups?

makes implementing condition variables simpler

can be hard to avoid loop in more complicated scenarios

e.g. signal() saying okay to remove item from queue
what if another thread sneaks in and does it first?

- maybe signal() could be redesigned to prevent this somehow?
- …but that's harder to implement

unbounded buffer producer/consumer

```
pthread_mutex_t lock;
pthread_cond_t data_ready;
UnboundedQueue buffer;

Produce(item) {
    pthread_mutex_lock(&lock);
    buffer.enqueue(item);
    pthread_cond_signal(&data_ready);
    pthread_mutex_unlock(&lock);
}

Consume() {
    pthread_mutex_lock(&lock);
    while (buffer.empty()) {
        pthread_cond_wait(&data_ready, &lock);
    }
    item = buffer.dequeue();
    pthread_mutex_unlock(&lock);
    return item;
}
```

unbounded buffer producer/consumer

```
pthread_mutex_t lock;
pthread_cond_t data_ready;
UnboundedQueue buffer;

Produce(item) {
    pthread_mutex_lock(&lock);
    buffer.enqueue(item);
    pthread_cond_signal(&data_ready);
    pthread_mutex_unlock(&lock);
}

Consume() {
    pthread_mutex_lock(&lock);
    while (buffer.empty()) {
        pthread_cond_wait(&data_ready, &lock);
    }
    item = buffer.dequeue();
    pthread_mutex_unlock(&lock);
    return item;
}
```

rule: never touch `buffer` without acquiring lock

otherwise: what if two threads simulatenously en/dequeue?

(both use same array/linked list entry?)

(both reallocate array?)

unbounded buffer producer/consumer

```
pthread_mutex_t lock;
pthread_cond_t data_ready;
UnboundedQueue buffer;

Produce(item) {
    pthread_mutex_lock(&lock);
    buffer.enqueue(item);
    pthread_cond_signal(&data_ready);
    pthread_mutex_unlock(&lock);
}

Consume() {
    pthread_mutex_lock(&lock);
    while (buffer.empty()) {
        pthread_cond_wait(&data_ready, &lock);
    }
    item = buffer.dequeue();
    pthread_mutex_unlock(&lock);
    return item;
}
```

check if empty
if so, dequeue

okay because have lock

other threads can**not** dequeue here

unbounded buffer producer/consumer

```
pthread_mutex_t lock;
pthread_cond_t data_ready;
UnboundedQueue buffer;

Produce(item) {
    pthread_mutex_lock(&lock);
    buffer.enqueue(item);
    pthread_cond_signal(&data_ready);
    pthread_mutex_unlock(&lock);
}

Consume() {
    pthread_mutex_lock(&lock);
    while (buffer.empty()) {
        pthread_cond_wait(&data_ready, &lock);
    }
    item = buffer.dequeue();
    pthread_mutex_unlock(&lock);
    return item;
}
```

wake one Consume thread *if any are waiting*

unbounded buffer producer/consumer

```
pthread_mutex_t lock;
pthread_cond_t data_ready;
UnboundedQueue buffer;

Produce(item) {
    pthread_mutex_lock(&lock);
    buffer.enqueue(item);
    pthread_cond_signal(&data_ready
    pthread_mutex_unlock(&lock);
}

Consume() {
    pthread_mutex_lock(&lock);
    while (buffer.empty()) {
        pthread_cond_wait(&data_ready, &lock);
    }
    item = buffer.dequeue();
    pthread_mutex_unlock(&lock);
    return item;
}
```

Thread 1	Thread 2
Produce()	
…lock	
…enqueue	
…signal	
…unlock	
	Consume()
	…lock
	…empty? no
	…dequeue
	…unlock
	return

0 iterations: Produce() called before Consume()
1 iteration: Produce() signalled, probably
2+ iterations: spurious wakeup or …?

unbounded buffer producer/consumer

```
pthread_mutex_t lock;
pthread_cond_t data_ready;
UnboundedQueue buffer;

Produce(item) {
    pthread_mutex_lock(&lock);
    buffer.enqueue(item);
    pthread_cond_signal(&data_ready);
    pthread_mutex_unlock(&lock);
}

Consume() {
    pthread_mutex_lock(&lock);
    while (buffer.empty()) {
        pthread_cond_wait(&data_ready, &lock);
    }
    item = buffer.dequeue();
    pthread_mutex_unlock(&lock);
    return item;
}
```

Thread 1	Thread 2
	Consume()
	...lock
	...empty? yes
	...unlock/start wait
Produce()	waiting for data_ready
...lock	
...enqueue	
...signal	stop wait
...unlock	lock
	...empty? no
	...dequeue
	...unlock
	return

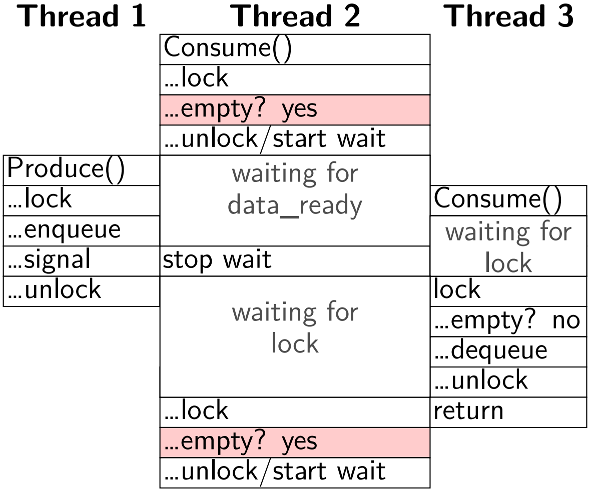
0 iterations: Produce() called before Consume()
1 iteration: Produce() signalled, probably
2+ iterations: spurious wakeup or ...?

unbounded buffer producer/consumer

```
pthread_mutex_t lock;
pthread_cond_t data_ready;
UnboundedQueue buffer;

Produce(item) {
    pthread_mutex_lock(&lock);
    buffer.enqueue(item);
    pthread_cond_signal(&data_rea
    pthread_mutex_unlock(&lock);
}

Consume() {
    pthread_mutex_lock(&lock);
    while (buffer.empty()) {
        pthread_cond_wait(&data_r
    }
    item = buffer.dequeue();
    pthread_mutex_unlock(&lock);
    return item;
}
```



0 iterations: Produce() called before Consume()
1 iteration: Produce() signalled, probably
2+ iterations: spurious wakeup or ...?

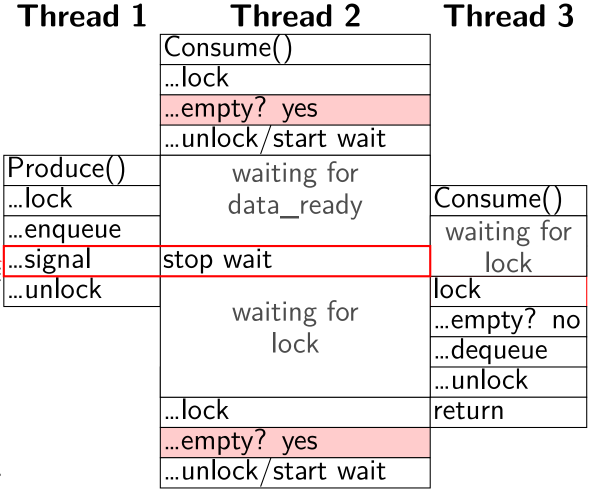
unbounded buffer producer/consumer

```
pthread_mutex_t lock;
pthread_cond_t data_ready;
```

in pthreads: signalled thread not gaurenteed to hold lock next

alternate design: signalled thread gets lock next called "Hoare scheduling" not done by pthreads, Java, …

```
    pthread_mutex_lock(&lock);
    while (buffer.empty()) {
        pthread_cond_wait(&data_r
    }
    item = buffer.dequeue();
    pthread_mutex_unlock(&lock);
    return item;
}
```



0 iterations: Produce() called before Consume()
1 iteration: Produce() signalled, probably
2+ iterations: spurious wakeup or …?

Hoare versus Mesa monitors

Hoare-style monitors

signal 'hands off' lock to awoken thread

Mesa-style monitors

any eligible thread gets lock next
(maybe some other idea of priority?)

every current threading library I know of does Mesa-style

threads: sum example (to global)

```
int array[1024];
int results[2];
void *sum_thread(void *argument) {
    int id = (int) argument;
    int sum = 0;
    for (int i = id * 512; i < (id + 1) * 512; ++i) {
        sum += array[i];
    }
    results[id] = sum;
    return NULL;
}
int sum_all() {
    pthread_t thread[2];
    for (int i = 0; i < 2; ++i) {
        pthread_create(&threads[i], NULL, sum_thread, (void *) i);
    }
    for (int i = 0; i < 2; ++i)
        pthread_join(threads[i], NULL);
    return results[0] + results[1];
}
```

threads: sum example (to global)

array, results: global variables — shared

```
int array[1024];
int results[2];
void *sum_thread(void *argument) {
    int id = (int) argument;
    int sum = 0;
    for (int i = id * 512; i < (id + 1) * 512; ++i) {
        sum += array[i];
    }
    results[id] = sum;
    return NULL;
}
int sum_all() {
    pthread_t thread[2];
    for (int i = 0; i < 2; ++i) {
        pthread_create(&threads[i], NULL, sum_thread, (void *) i);
    }
    for (int i = 0; i < 2; ++i)
        pthread_join(threads[i], NULL);
    return results[0] + results[1];
}
```

threads: sum example (to main stack)

```
int array[1024];
struct ThreadInfo {
    int start, end, 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 += array[i];
    }
    my_info->result = sum;
    return NULL;
}
int sum_all() {
    pthread_t thread[2]; ThreadInfo info[2];
    for (int i = 0; i < 2; ++i) {
        info[i].start = i*512; info[i].end = (i+1)*512;
        pthread_create(&threads[i], NULL, sum_thread, &info[i]);
    }
    for (int i = 0; i < 2; ++i)
        pthread_join(threads[i], NULL);
    return info[0].result + info[1].result;
}
```

threads: sum example (to main stack)

array: global variable — shared

```
int array[1024];
struct ThreadInfo
    int start, end, 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 += array[i];
    }
    my_info->result = sum;
    return NULL;
}
int sum_all() {
    pthread_t thread[2]; ThreadInfo info[2];
    for (int i = 0; i < 2; ++i) {
        info[i].start = i*512; info[i].end = (i+1)*512;
        pthread_create(&threads[i], NULL, sum_thread, &info[i]);
    }
    for (int i = 0; i < 2; ++i)
        pthread_join(threads[i], NULL);
    return info[0].result + info[1].result;
}
```

threads: sum example (to main stack)

```
int array[1024];
struct ThreadInfo {
    int start, end, result;
};
void *sum_thread(void *argument) {
    ThreadInfo *my_info = (ThreadInfo *) argument;
    int sum = 0;
    for (int i = my_info->
        sum += array[i];
    }
    my_info->result = sum;
    return NULL;
}
int sum_all() {
    pthread_t thread[2]; ThreadInfo info[2];
    for (int i = 0; i < 2; ++i) {
        info[i].start = i*512; info[i].end = (i+1)*512;
        pthread_create(&threads[i], NULL, sum_thread, &info[i]);
    }
    for (int i = 0; i < 2; ++i)
        pthread_join(threads[i], NULL);
    return info[0].result + info[1].result;
}
```

my_info: pointer to sum_all's stack
only okay because sum_all waits!

threads: sum example (to main stack)

```
int array[1024];
struct ThreadInfo {
    int start, end, 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 += array[i];
    }
    my_info->result = sum;
    return NULL;
}
int sum_all() {
    pthread_t thread[2]; ThreadInfo info[2];
    for (int i = 0; i < 2; ++i) {
        info[i].start = i*512; info[i].end = (i+1)*512;
        pthread_create(&threads[i], NULL, sum_thread, &info[i]);
    }
    for (int i = 0; i < 2; ++i)
        pthread_join(threads[i], NULL);
    return info[0].result + info[1].result;
}
```

threads: sum example (no globals)

```
struct ThreadInfo { int *array; 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->array[i];
    }
    my_info->result = sum;
    return NULL;

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

threads: sum example (no globals)

```
struct ThreadInfo { int *array; 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->array[i];
    }
    my_info->result = sum;
    return NULL;

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

threads: sum example (no globals)

```
struct ThreadInfo { int *array; 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->array[i];
    }
    my_info->result = sum;
    return NULL;

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

threads: sum example (no globals)

```
struct ThreadInfo { int *array; 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->array[i];
    }
    my_info->result = sum;
    return NULL;

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

threads: sum example (on heap)

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

threads: sum example (on heap)

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

threads: sum example (on heap)

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