MANCHESTER 1824
The University of Manchester

COMP35112 Chip Multiprocessors

Programming with Locks and Barriers

Pierre Olivier

Synchronisation Mechanisms

- In a multithreaded program, threads rarely execute completely independently:
  - In many scenarios they need to wait for each other and to communicate by accessing shared data

Synchronisation Mechanisms

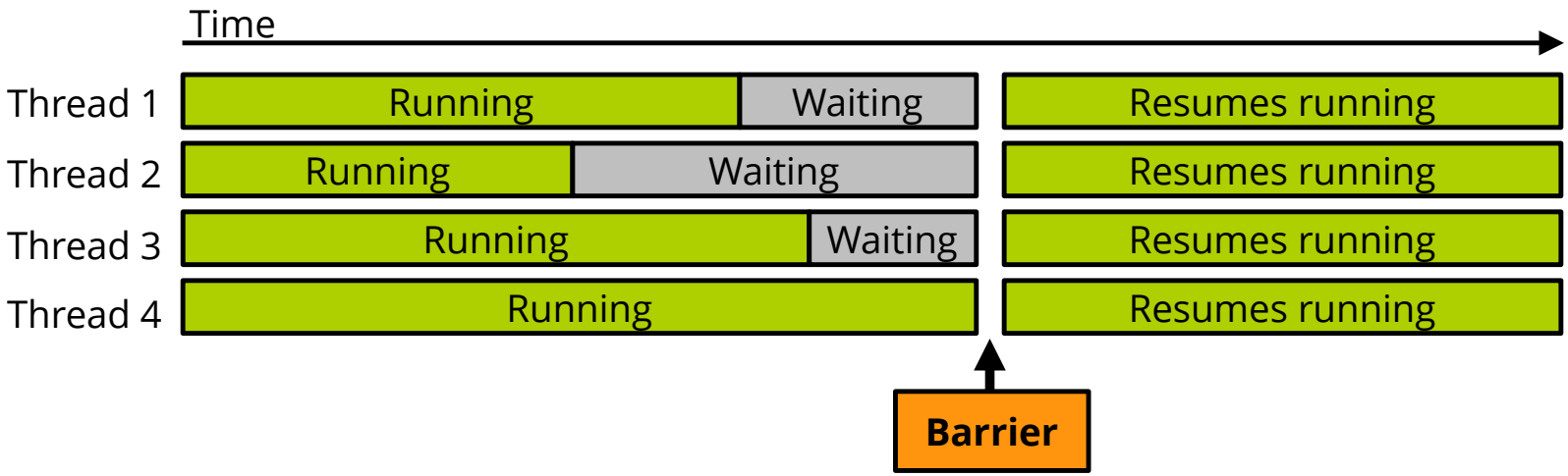
- In a multithreaded program, threads rarely execute completely independently:
  - In many scenarios they need to wait for each other and to communicate by accessing shared data
  - Brings the need for **synchronisation mechanisms**

Synchronisation Mechanisms

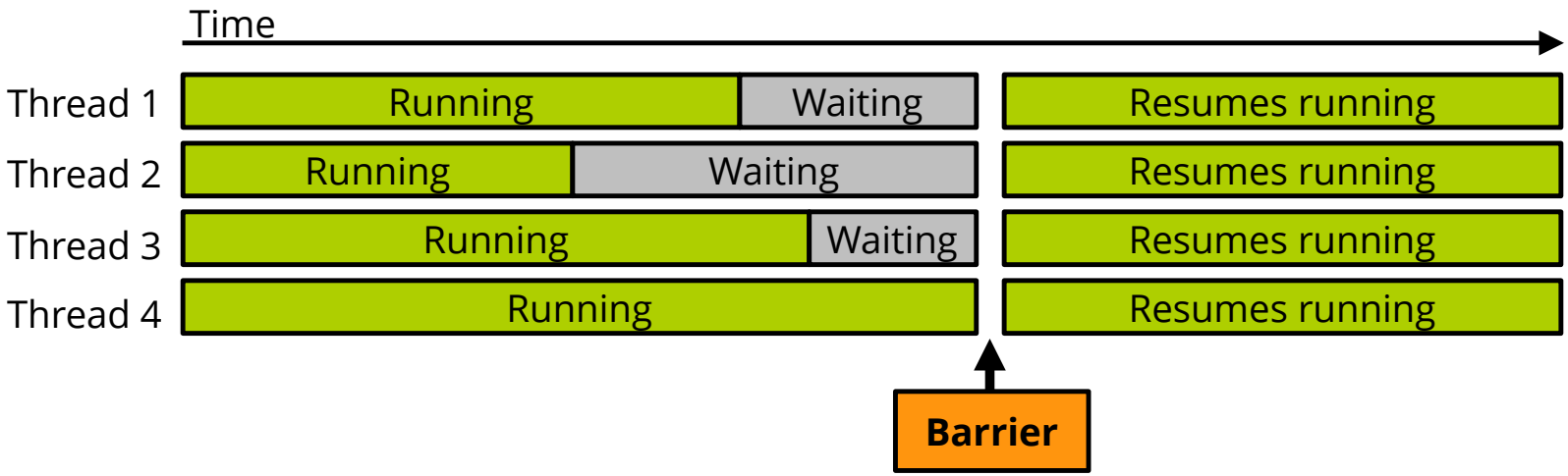
- In a multithreaded program, threads rarely execute completely independently:
  - In many scenarios they need to wait for each other and to communicate by accessing shared data
  - Brings the need for **synchronisation mechanisms**
- In this lecture we'll cover:
  - **Barriers**: simple mechanism letting threads wait for each other
  - **Locks** allowing threads to safely access shared data
  - **Condition variables** for event signalling between threads

Barriers

Barriers

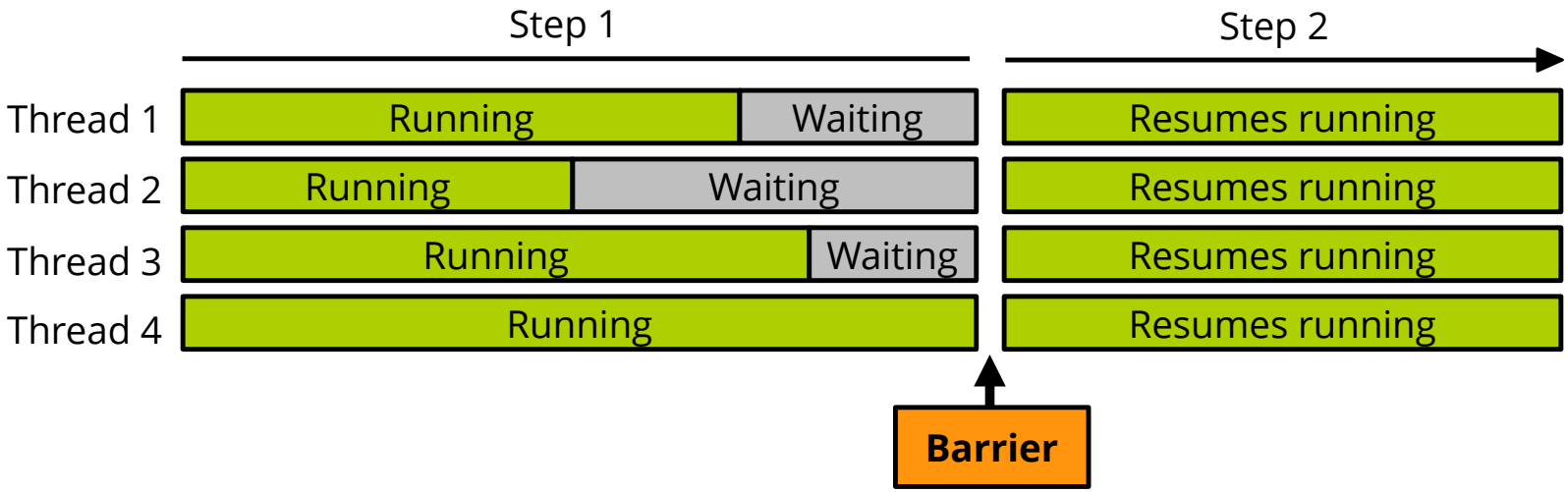

Time
Thread 1
Running
Waiting
Resumes running
Thread 2
Running
Waiting
Resumes running
Thread 3
Running
Waiting
Resumes running
Thread 4
Running
Resumes running
Barrier

Barriers



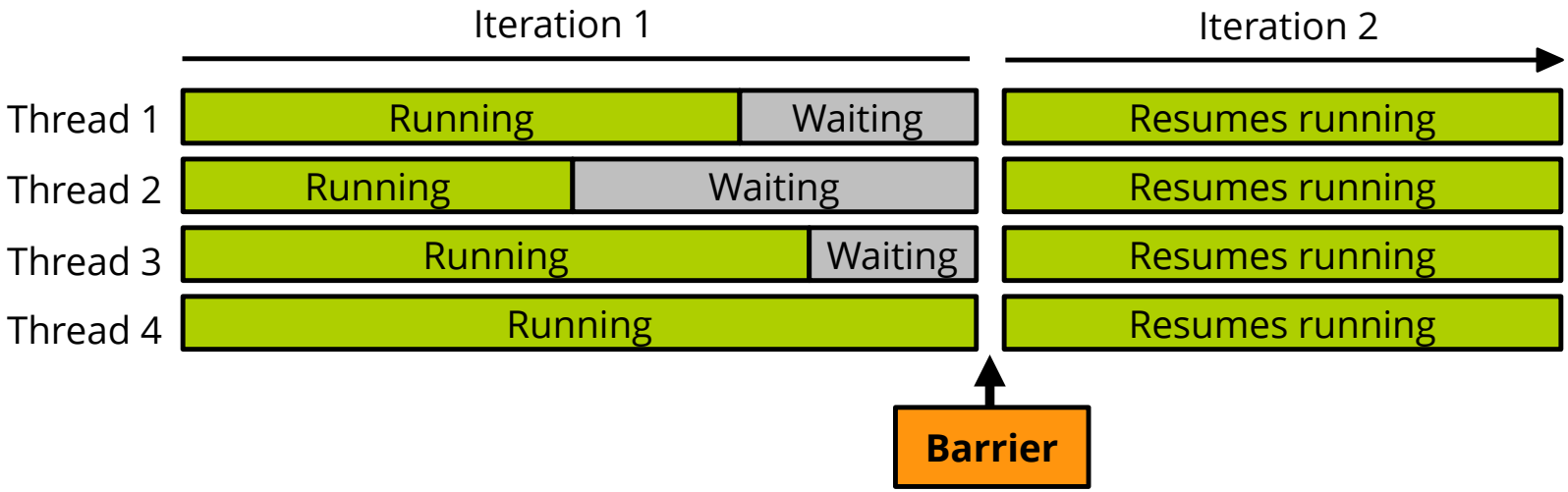
- **A barrier is where a number of threads "meet up"**
  - When all threads have reached it, they can all proceed

Barriers



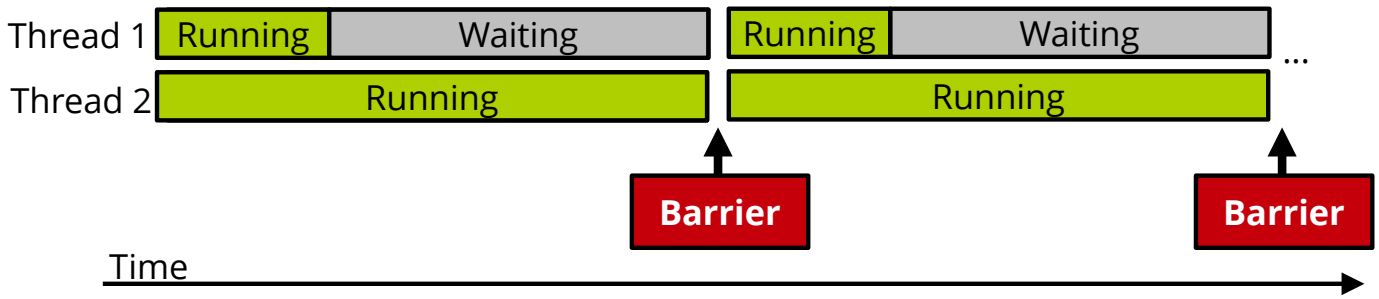
- Very natural when threads are used to implement data parallelism
  - Want the whole answer from this step before proceeding to the next step

Barriers



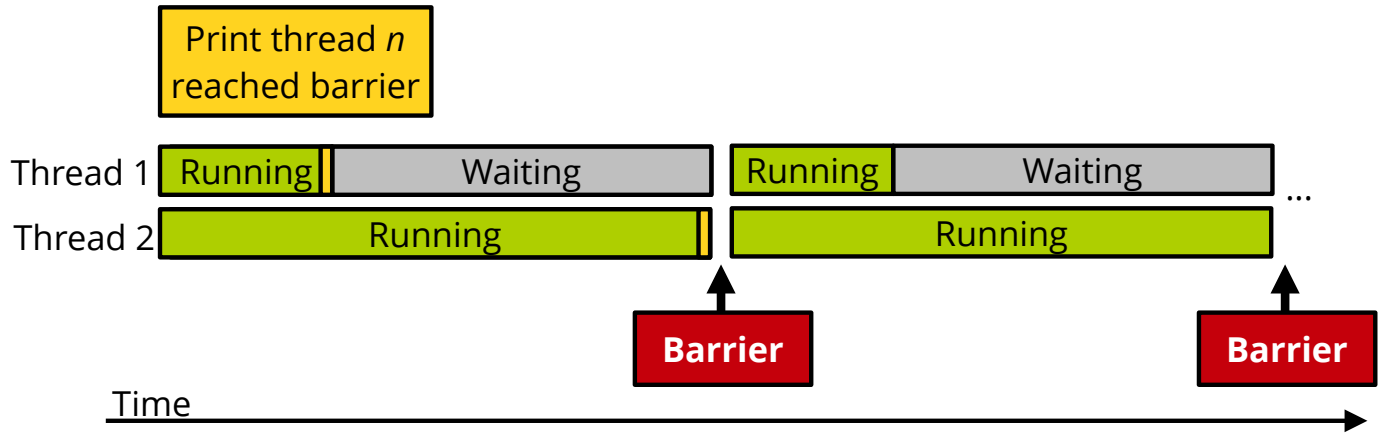
- Very natural when threads are used to implement data parallelism
  - Want the whole answer from this step before proceeding to the next step
- Would also use when data dependence limits loop parallelisation
  - `pthread_barrier_t` allows multiple use!

Barrier Example



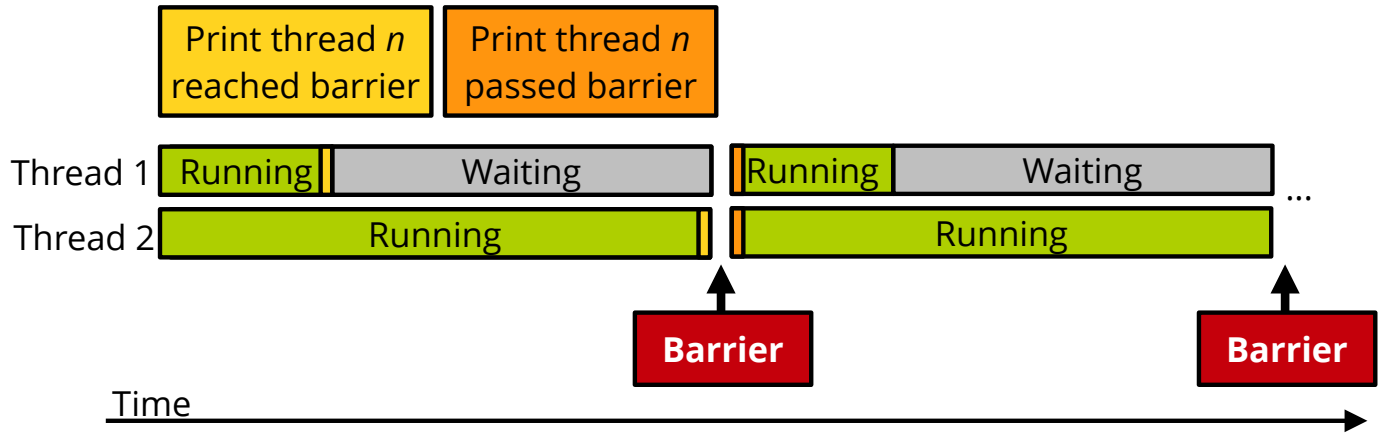
- Force thread 2 to execute for a longer time than thread 1 at each iteration

Barrier Example



- Force thread 2 to execute for a longer time than thread 1 at each iteration

Barrier Example



- Force thread 2 to execute for a longer time than thread 1 at each iteration

Barrier Example

```c
#define ITERATIONS      10

typedef struct {
    int id;
    int spin_amount;
    pthread_barrier_t *barrier;
} worker;
```

```c
void *thread_fn(void *data) {
  worker *arg = (worker *)data;
  int id = arg->id;
  int iteration = 0;

  while(iteration != ITERATIONS) {

    /* busy loop to simulate activity */
    for(int i=0; i<arg->spin_amount; i++);

    printf("Thread %d reached barrier\n", id);

    int r = pthread_barrier_wait(arg->barrier);
    if(r!=PTHREAD_BARRIER_SERIAL_THREAD && r) {
      perror("pthread_barrier_wait");
      exit(-1);
    }

    printf("Thread %d passed barrier\n", id);
    iteration++;
  }
  pthread_exit(NULL);
}
```

Code Sandbox

Barrier Example

```c
#include <err.h> // for errx
/* ... */
#define T1_SPIN_AMOUNT  200000000
#define T2_SPIN_AMOUNT  (10 * T1_SPIN_AMOUNT)

int main(int argc, char **argv) {
    pthread_t t1, t2;
    pthread_barrier_t barrier;

    worker w1 = {1, T1_SPIN_AMOUNT, &barrier};
    worker w2 = {2, T2_SPIN_AMOUNT, &barrier};

    if(pthread_barrier_init(&barrier, NULL, 2))
        errx(-1, ("pthread_barrier_init"));      // equivalent of perror(msg); exit(-1);

    if(pthread_create(&t1, NULL, thread_fn, (void *)&w1) ||
            pthread_create(&t2, NULL, thread_fn, (void *)&w2))
        errx(-1, "pthread_create");

    if(pthread_join(t1, NULL) || pthread_join(t2, NULL))
        errx(-1, "phread_join");

    return 0;
}
```

Code Sandbox

Locks

Locks: Motivational Example 1

- Locks protect shared data
  - Why do so?
- Cash machine example: cash withdrawal

Locks: Motivational Example 1

- Locks protect shared data
  - Why do so?
- Cash machine example: cash withdrawal

```
int total = get_total_from_account();     /* total funds in user account */
int withdrawal = get_withdrawal_amount(); /* amount user asking to withdraw */

/* check whether the user has enough funds in her account */
if(total < withdrawal)
    abort("Not enough money!");

/* The user has enough money, deduct the withdrawal amount from here total */
total -= withdrawal;
update_total_funds(total);

/* give the money to the user */
spit_out_money(withdrawal);
```

Locks: Motivational Example 1

```
int total = get_total_from_account();
int withdrawal = get_withdrawal_amount();

if(total < withdrawal)
    abort("Not enough money!");

total -= withdrawal;
update_total_funds(total);

spit_out_money(withdrawal);
```

Locks: Motivational Example 1

```
int total = get_total_from_account();
int withdrawal = get_withdrawal_amount();

if(total < withdrawal)
    abort("Not enough money!");

total -= withdrawal;
update_total_funds(total);

spit_out_money(withdrawal);
```

- Assume 2 transactions are hapenning nearly at the same time
  - E.g. shared credit card account
- Assume: `total == 105`, `withdrawal1 == 100`, `withdrawal2 == 10`
  - Should fail as `(100+10)>105`

Locks: Motivational Example 1

```
int total = get_total_from_account();
int withdrawal = get_withdrawal_amount();

if(total < withdrawal)
    abort("Not enough money!");

total -= withdrawal;
update_total_funds(total);

spit_out_money(withdrawal);
```

- `total == 105`, `withdrawal1 == 100`, `withdrawal2 == 10`
- A possible scenario:
  1. Threads get total in local variable, both get `105`
  2. Threads check that `100 < 105` and `10 < 105`
     - All good
  3. Thread 1 updates:
     - `total = 105 - 100 = 5`
     - `update_total_funds(5)`
  4. Slightly later thread 2 updates:
     - `total = 105 - 10 = 95`
     - `update_total_funds(95)`

Locks: Motivational Example 1

```
int total = get_total_from_account();
int withdrawal = get_withdrawal_amount();

if(total < withdrawal)
    abort("Not enough money!");

total -= withdrawal;
update_total_funds(total);

spit_out_money(withdrawal);
```

Total withdrawal: 110, and there is 95 left on the account!
Free money 🤑

- **Race condition:** operations on shared state should not happen at the same time

- `total == 105`, `withdrawal1 == 100`, `withdrawal2 == 10`
- A possible scenario:
  1. Threads get total in local variable, both get `105`
  2. Threads check that `100 < 105` and `10 < 105`
     - All good
  3. Thread 1 updates:
     - `total = 105 - 100 = 5`
     - `update_total_funds(5)`
  4. Slightly later thread 2 updates:
     - `total = 105 - 10 = 95`
     - `update_total_funds(95)`

Locks: Motivational Example 2

- Consider the `i++` statement
  - Could translate into machine code as:

```
1. load the current value of i from memory and copy it into a register
2. add one to the value stored into the register
3. store from the register to memory the new value of i
```

Locks: Motivational Example 2

- Consider the `i++` statement
  - Could translate into machine code as:

```
1. load the current value of i from memory and copy it into a register
2. add one to the value stored into the register
3. store from the register to memory the new value of i
```

A possible scenario when 2 threads execute `i++`:

Thread 1	Thread 2
load i (7)	-
increment i (7→8)	-
store i (8)	-
-	load i (8)
-	increment i (8→9)
-	store i (9)

Locks: Motivational Example 2

- Consider the `i++` statement
  - Could translate into machine code as:

```
1. load the current value of i from memory and copy it into a register
2. add one to the value stored into the register
3. store from the register to memory the new value of i
```

Another possible scenario: **Race condition** 🔥

- Need **locks** to **serialise** access to shared data, i.e. to ensure that **critical sections** are executed atomically

Thread 1	**Thread 2**
load i (7)	load i (7)
increment i (7→8)	-
-	increment i (7→8)
store i (8)	-
-	store i (8)

Locks

- Bits of code in our program where shared data is accessed/updated are called **critical sections**

Locks

- Bits of code in our program where shared data is accessed/updated are called **critical sections**
- **Lock: synchronisation primitive enforcing limits on the execution of a critical section:**
  - Amount of threads that can concurrently execute it (generally 1, **serialisation**)
  - **Atomicity**: when a thread starts to run a critical section, it must finish it before another thread can enter the critical section

Locks

- Critical sections can be protected with **locks**:
  - **Only one thread can hold a given lock at a time**
  - Holding the lock lets the thread enter and execute the corresponding critical section

Locks

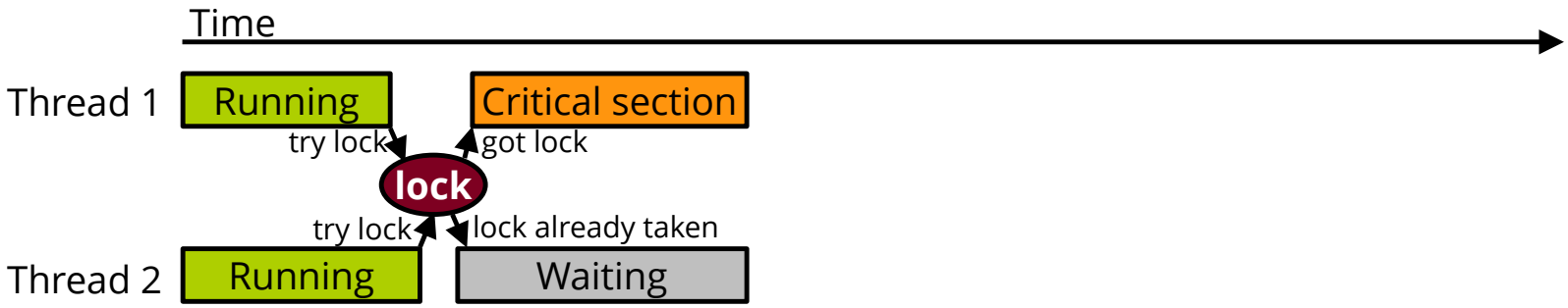
- Critical sections can be protected with **locks**:
  - **Only one thread can hold a given lock at a time**
  - Holding the lock lets the thread enter and execute the corresponding critical section
- Threads wishing to enter the critical section **try** to take the lock:
  - A thread attempting to take a free lock will get it
  - Other threads requesting the lock wait until the lock is released by its holder

Locks

- Critical sections can be protected with **locks**:
  - **Only one thread can hold a given lock at a time**
  - Holding the lock lets the thread enter and execute the corresponding critical section

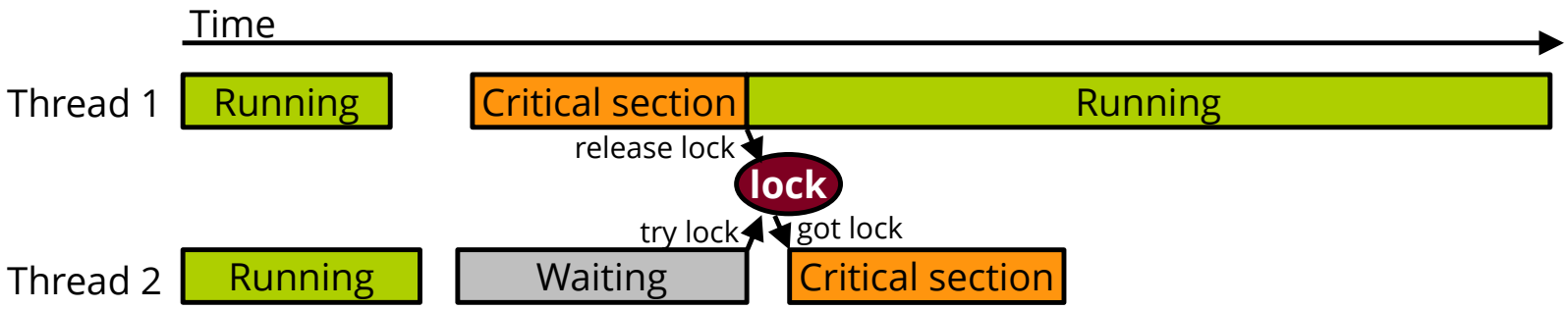
Locks

- Critical sections can be protected with **locks**:
  - **Only one thread can hold a given lock at a time**
  - Holding the lock lets the thread enter and execute the corresponding critical section



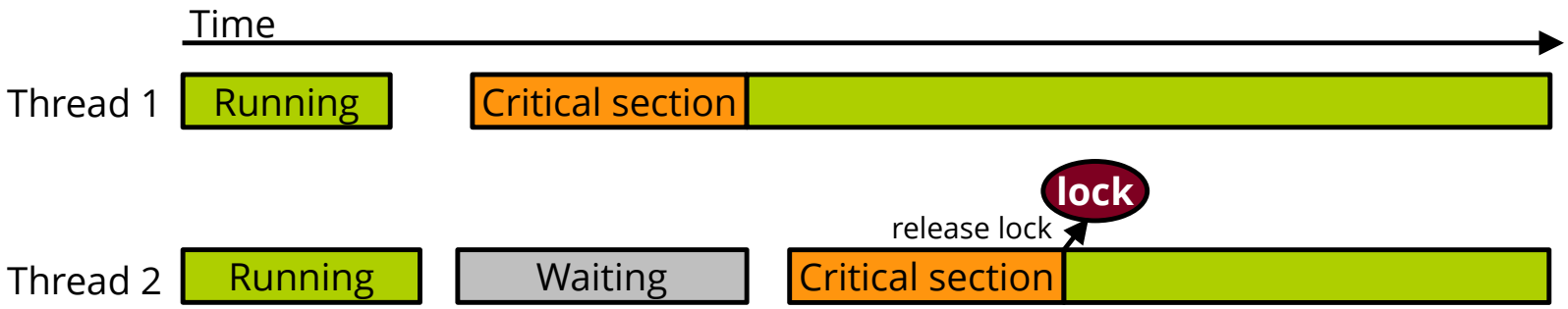
Locks

- Critical sections can be protected with **locks**:
  - **Only one thread can hold a given lock at a time**
  - Holding the lock lets the thread enter and execute the corresponding critical section



Locks

- Critical sections can be protected with **locks**:
  - **Only one thread can hold a given lock at a time**
  - Holding the lock lets the thread enter and execute the corresponding critical section



Pthreads Mutexes

- **Mutexes: mutual exclusion locks**

```c
#include <pthread.h>

pthread_mutex_t mutex;

void my_thread_function() {

    pthread_mutex_lock(&mutex);

    /* critical section here */

    pthread_mutex_unlock(&mutex);

}
```

Pthreads Mutexes

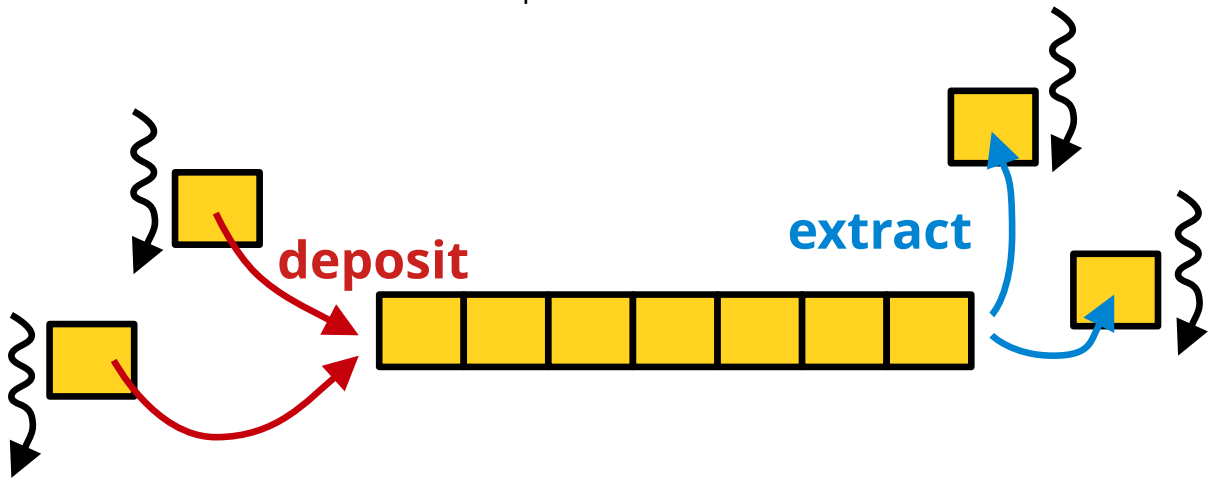
- Mutexes: mutual exclusion locks

```c
#include <pthread.h>

pthread_mutex_t mutex;

void my_thread_function() {

    pthread_mutex_lock(&mutex);

    /* critical section here */

    pthread_mutex_unlock(&mutex);

}
```

Important:
in the next examples we omit pthread functions return code checking for the sake of brevity, but note that almost all of them can fail!

Lock Usage Example

- **Bounded buffer**: circular FIFO producer-consumer buffer



Lock Usage Example

```c
typedef struct {
    int *buffer;           // the buffer
    int max_elements;      // size of the buffer
    int in_index;          // index of the next free slot
    int out_index;         // index of the next message to extract
    int count;             // number of used slots
    pthread_mutex_t lock;  // lock protecting the buffer
} bounded_buffer;

int init_bounded_buffer(bounded_buffer *b, int size) {
    b->buffer = malloc(size * sizeof(int));
    if(!b->buffer)
        return -1;

    b->max_elements = size;
    b->in_index = 0;
    b->out_index = 0;
    b->count = 0;
    pthread_mutex_init(&b->lock, NULL);
    return 0;
}

void destroy_bounded_buffer(bounded_buffer *b) {
    free(b->buffer);
}
```

Code Sandbox

Lock Usage Example (continued)

```
void deposit(bounded_buffer *b, int message) {
    pthread_mutex_lock(&b->lock);

    int full = (b->count == b->max_elements);

    while(full) {
        // buffer is full, can't deposit! Release the lock and wait a bit
        // to give another thread a chance to extract an element
        pthread_mutex_unlock(&b->lock);
        usleep(100);
        pthread_mutex_lock(&b->lock);

        // is the buffer still full?
        full = (b->count == b->max_elements);
    }

    // perform deposit
    b->buffer[b->in_index] = message;
    b->in_index = (b->in_index + 1) % b->max_elements;
    b->count++;

    pthread_mutex_unlock(&b->lock);
}
```

Code Sandbox

Lock Usage Example (continued)

```
int extract(bounded_buffer *b) {
    pthread_mutex_lock(&b->lock);

    int empty = !(b->count);

    while(empty) {
        // buffer is empty, nothing to extract! Release the lock and wait a bit
        // to give another thread a chance to deposit an element
        pthread_mutex_unlock(&b->lock);
        usleep(100);
        pthread_mutex_lock(&b->lock);

        // is the buffer still empty?
        empty = !(b->count);
    }

    // perform extract
    int message = b->buffer[b->out_index];
    b->out_index = (b->out_index + 1) %
        b->max_elements;
    b->count--;

    pthread_mutex_unlock(&b->lock);
    return message;
}
```

Code Sandbox

Lock Usage Example (continued)

```c
typedef struct {
    int iterations;
    bounded_buffer *bb;
} worker;

void *deposit_thread_fn(void *data) {
  worker *w = (worker *)data;
  for(int i=0; i<w->iterations; i++) {
    deposit(w->bb, i);
    printf("[deposit thread] put %d\n", i);
  }
  pthread_exit(NULL);
}

void *extract_thread_fn(void *data) {
  worker *w = (worker *)data;
  for(int i=0; i<w->iterations; i++) {
    int x = extract(w->bb);
    printf("[extract thread] got %d\n", x);
  }
  pthread_exit(NULL);
}
```

```c
#define BUFFER_SIZE 100
int main(int argc, char **argv) {
  bounded_buffer b;
  pthread_t t1, t2;

  if(init_bounded_buffer(&b, BUFFER_SIZE)) {
    /* error */
  }

  worker w1 = {BUFFER_SIZE*2, &b};
  worker w2 = {BUFFER_SIZE*2, &b};

  pthread_create(&t1, NULL, deposit_thread_fn,
    (void *)&w1);
  pthread_create(&t2, NULL, extract_thread_fn,
    (void *)&w2);

  pthread_join(t1, NULL);
  pthread_join(t2, NULL);
  destroy_bounded_buffer(&b);
  return 0;
}
```

Code Sandbox

Omit the Locks and ...

Omit the Locks and ...

- Madness ensues 🙃

Omit the Locks and ...

- Madness ensues 🙃
- Could have two threads in `deposit()` both writing to the same element of buffer
  - One value is lost

Omit the Locks and ...

- Madness ensues 🙃
- Could have two threads in `deposit()` both writing to the same element of buffer
  - One value is lost
- Could then either increment `in_index`
  - Once: whole call of `deposit()` lost
  - Twice: spurious (old) value apparently deposited

Omit the Locks and ...

- Madness ensues 🙃
- Could have two threads in `deposit()` both writing to the same element of buffer
  - One value is lost
- Could then either increment `in_index`
  - Once: whole call of `deposit()` lost
  - Twice: spurious (old) value apparently deposited
- Similarly for two calls of `extract()`
- Even problems between a call of `deposit()` and one of `extract()`, e.g. both change count

Omit the Locks and ...

- Madness ensues 🙃
- Could have two threads in `deposit()` both writing to the same element of buffer
  - One value is lost
- Could then either increment `in_index`
  - Once: whole call of `deposit()` lost
  - Twice: spurious (old) value apparently deposited
- Similarly for two calls of `extract()`
- Even problems between a call of `deposit()` and one of `extract()`, e.g. both change count

Concurrency issues can be *extremely* hard to debug in medium/large scale programs

Omit the Locks and ...

```
/* BUGGY version of deposit, without locks */
void deposit(bounded_buffer *b, int message) {

    while (b->count == b->max_elements);

    b->buffer[b->in_index] = message;
    b->in_index = (b->in_index + 1) % b->max_elements;
    b->count++;
}

/* BUGGY version of extract, without locks */
int extract(bounded_buffer *b) {

    while (!(b->count));

    int message = b->buffer[b->out_index];
    b->out_index = (b->out_index + 1) % b->max_elements;
    b->count--;

    return message;
}
```

Code Sandbox

Condition Variables

Event Signalling

- Sleeping or busy-waiting for the buffer to be non-full/non-empty is suboptimal

```
while(full) {
    pthread_mutex_unlock(&b->lock);
    usleep(100);
    pthread_mutex_lock(&b->lock);
    full = (b->count == b->max_elements);
}
```

```
while(full) {
    pthread_mutex_unlock(&b->lock);
    /* busy wait */
    pthread_mutex_lock(&b->lock);
    full = (b->count == b->max_elements);
}
```

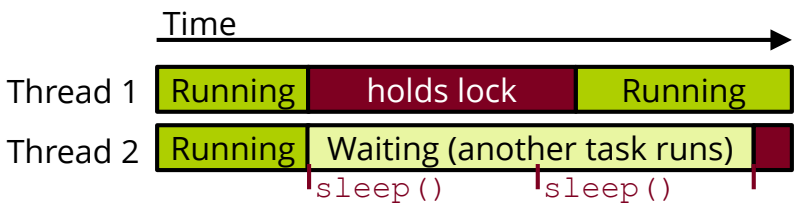
- With the `usleep` set to an arbitrary time, may sleep for a much longer time than needed
- Without usleep, keep trying non-stop, monopolising the CPU

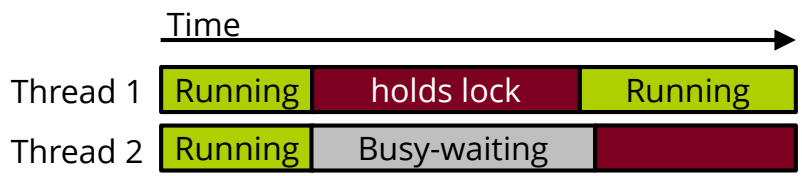
Event Signalling

- Sleeping or busy-waiting for the buffer to be non-full/non-empty is suboptimal

```c
while(full) {
    pthread_mutex_unlock(&b->lock);
    usleep(100);
    pthread_mutex_lock(&b->lock);
    full = (b->count == b->max_elements);
}
```

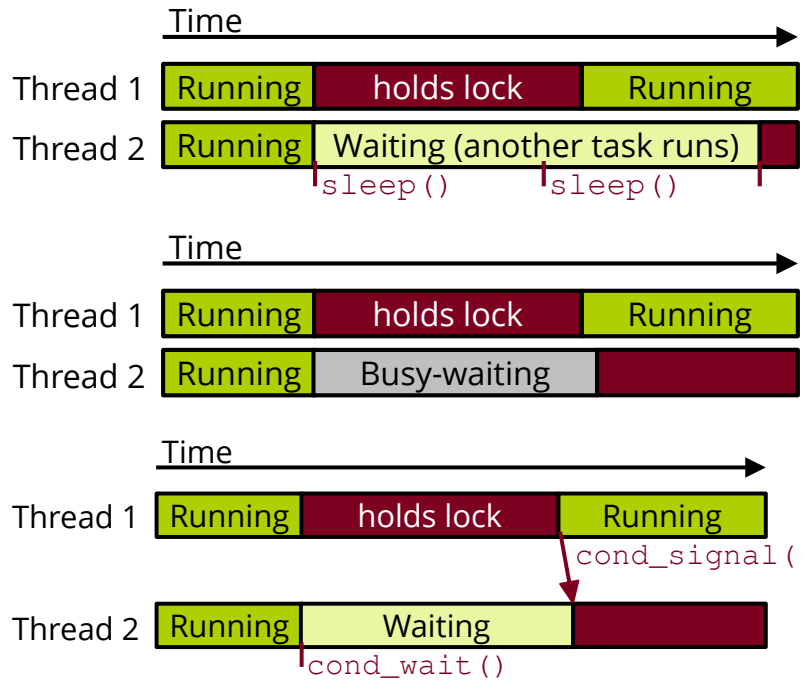
```c
while(full) {
    pthread_mutex_unlock(&b->lock);
    /* busy wait */
    pthread_mutex_lock(&b->lock);
    full = (b->count == b->max_elements);
}
```





Condition Variables

- Best of both worlds: thread wakes up right when needed, without monopolising the CPU by spinning



Condition Variables

```
void deposit(bounded_buffer *b, int message) {
  pthread_mutex_lock(&b->lock);

  int full = (b->count == b->max_elements);
  while(full) {
    // wait on condfull to be signalled
    // when the buffer becomes non-full
    pthread_cond_wait(&b->condfull, &b->lock);
    full = (b->count == b->max_elements);
  }

  b->buffer[b->in_index] = message;
  b->in_index = (b->in_index + 1) %
    b->max_elements;

  // signal condempty if buffer becomes
  // non-empty
  if(b->count++ == 0)
    pthread_cond_signal(&b->condempty);

  pthread_mutex_unlock(&b->lock);
}
```

```
int extract(bounded_buffer *b) {
  pthread_mutex_lock(&b->lock);

  int empty = !(b->count);
  while(empty) {
    // wait on condempty to be signalled
    // when the buffer becomes non-empty
    pthread_cond_wait(&b->condempty, &b->lock);
    empty = !(b->count);
  }

  int message = b->buffer[b->out_index];
  b->out_index = (b->out_index + 1) %
    b->max_elements;

  // signal condfull if buffer becomes
  // non-full
  if(b->count-- == b->max_elements)
    pthread_cond_signal(&b->condfull);

  pthread_mutex_unlock(&b->lock);
  return message;
}
```

Code Sandbox

Summary

- Thread need to sync up and access shared data
  - Need for **synchronisation mechanisms such as barriers, locks, condition variables**
- Shared memory multithreading opens up the risk of **race conditions**
  - When a shared resource is accessed by multiple threads at the same time, including at least a write access
- Next lecture: more about locks