Concurrency Primitives

Locks

Locks are definitely needed when we update (read then modify) the value of a variable

Concurrency Primitives

Locks

Locks are definitely needed when we update (read then modify) the value of a variable

The question arises regarding whether we need a lock around a simple read of a multi-byte value, such as a 32-bit (4 byte) integer

Concurrency Primitives

Locks

Locks are definitely needed when we update (read then modify) the value of a variable

The question arises regarding whether we need a lock around a simple read of a multi-byte value, such as a 32-bit (4 byte) integer

It is easy to believe some bytes of a value might be written while half-way through being read, resulting in a mix of the bits of the old and new values

Concurrency Primitives
Locks

Locks are definitely needed when we update (read then modify) the value of a variable

The question arises regarding whether we need a lock around a simple read of a multi-byte value, such as a 32-bit (4 byte) integer

It is easy to believe some bytes of a value might be written while half-way through being read, resulting in a mix of the bits of the old and new values

Called read (or write) *tearing*

Concurrency Primitives

Locks

However, for most (non-embedded) machine architectures these days it is likely (not certain!) to be safe to read simple values like integers or doubles that fit in a register: the hardware read is atomic (another side effect of the caching mechanism)

Concurrency Primitives

Locks

However, for most (non-embedded) machine architectures these days it is likely (not certain!) to be safe to read simple values like integers or doubles that fit in a register: the hardware read is atomic (another side effect of the caching mechanism)

Though you do need to be careful on strange machine architectures, or with compilers that try to be too clever (For hackers: think about non-aligned accesses)

Concurrency Primitives

Locks

However, for most (non-embedded) machine architectures these days it is likely (not certain!) to be safe to read simple values like integers or doubles that fit in a register: the hardware read is atomic (another side effect of the caching mechanism)

Though you do need to be careful on strange machine architectures, or with compilers that try to be too clever (For hackers: think about non-aligned accesses)

Certainly, though, for reading all of a larger object or structure, a lock will be necessary to ensure consistency across the multiple machine reads it takes to read in the whole structure

Concurrency Primitives

Locks

```
int x, y;
...
y = x;
```

Usually safe as reads of `ints` are generally atomic

Concurrency Primitives

Locks

```
// Also OO classes or objects
struct rational {
  int num, den;
};
struct rational r, s;
...
r = s;
```

Possibly unsafe, as it could take two machine reads to get all of `s`, which might be modified halfway through by another thread

Concurrency Primitives

Locks

```
// Also OO classes or objects
struct rational {
  int num, den;
};
struct rational r, s;
...
r = s;
```

Possibly unsafe, as it could take two machine reads to get all of `s`, which might be modified halfway through by another thread

Unlikely, but you can't rely on that

Concurrency Primitives

Locks

```
// Also OO classes or objects
struct rational {
  int num, den;
};
struct rational r, s;
...
r = s;
```

Possibly unsafe, as it could take two machine reads to get all of `s`, which might be modified halfway through by another thread

Unlikely, but you can't rely on that

Analogously for the write of `r`

Concurrency Primitives

Locks

Exercise For C geeks. There is an aliasing problem with bit fields in a `struct`

```
struct {
   int a: 5;
   int b: 3;
}
```

where an update to field `a` might be implemented as a read of a byte, modifying the bits of `a`, then writing a byte. Investigate

Exercise What about a 128-bit `long long int` on a 64-bit machine?

Concurrency Primitives

Locks

What about when we need to use more than one lock?

What about when we need to use more than one lock?

Of course, we can and should have separate locks in order to protect separate resources: we *could* use `countlock` to protect updates to another shared variable `sum`, but that would prevent one thread updating `count` while another is updating `sum`, which is perfectly safe to do

Concurrency Primitives

Locks

What about when we need to use more than one lock?

Of course, we can and should have separate locks in order to protect separate resources: we *could* use `countlock` to protect updates to another shared variable `sum`, but that would prevent one thread updating `count` while another is updating `sum`, which is perfectly safe to do

The only real reason to share a lock like this would be in when there are severe memory limitations: but lock implementations tend to use only a little memory per lock

Concurrency Primitives

Locks

But we do need to be careful about what we protect from what as it all has a cost

Concurrency Primitives

Locks

But we do need to be careful about what we protect from what as it all has a cost

Getting and releasing a lock can be relatively cheap (in some architectures and operating systems; expensive in others) but it is not free: it is an overhead to be taken into account and avoided if you can

Concurrency Primitives

Locks

But we do need to be careful about what we protect from what as it all has a cost

Getting and releasing a lock can be relatively cheap (in some architectures and operating systems; expensive in others) but it is not free: it is an overhead to be taken into account and avoided if you can

In many implementations these days the cost of getting an uncontended lock (not already locked) is cheap, while the cost of getting a lock that is already held is expensive

Concurrency Primitives

Locks

But we do need to be careful about what we protect from what as it all has a cost

Getting and releasing a lock can be relatively cheap (in some architectures and operating systems; expensive in others) but it is not free: it is an overhead to be taken into account and avoided if you can

In many implementations these days the cost of getting an uncontended lock (not already locked) is cheap, while the cost of getting a lock that is already held is expensive

So the common (you hope) case is cheap

Concurrency Primitives

Locks

Also note, locks can be used to protect anything, not just variables, e.g., whole function calls or whole loops. But we should try too keep the regions small

```
get_lock(mux);
someone_elses_dodgy_code();
free_lock(mux);
```

Also note, locks can be used to protect anything, not just variables, e.g., whole function calls or whole loops. But we should try too keep the regions small

```
get_lock(mux);
someone_elses_dodgy_code();
free_lock(mux);
```

Another reason to use a single lock could be that the code you want to protect is so complicated you are not clear on how to proceed!

Concurrency Primitives

Locks

Locks are a simple, low level mechanism

Concurrency Primitives

Locks

Locks are a simple, low level mechanism

Too low level: they are easy to use incorrectly

Concurrency Primitives

Locks

Locks are a simple, low level mechanism

Too low level: they are easy to use incorrectly

Suppose we have a couple of variables `x` and `y` we are protecting with mutexes `mx` and `my` respectively. We want to swap their values; elsewhere replace them both by their average

Concurrency Primitives

Locks

Locks are a simple, low level mechanism

Too low level: they are easy to use incorrectly

Suppose we have a couple of variables `x` and `y` we are protecting with mutexes `mx` and `my` respectively. We want to swap their values; elsewhere replace them both by their average

```
tmp = x;                  av = (x+y)/2;
x = y;                    x = av;
y = tmp;                  y = av;
```

To make this safe we have to use both locks

```
get_lock(mx);
get_lock(my);
tmp = x;
x = y;
y = tmp;
free_lock(my);
free_lock(mx);
```

Concurrency Primitives

Locks

Some pages of code later

```
get_lock(my);
get_lock(mx);
av = (x+y)/2;
x = av;
y = av;
free_lock(mx);
free_lock(my);
```

Spot the bug!

Concurrency Primitives

Locks

This will probably work most of the time, but occasionally just hangs doing nothing

Concurrency Primitives

Locks

This will probably work most of the time, but occasionally just hangs doing nothing

Sometimes we will get

This will probably work most of the time, but occasionally just hangs doing nothing

Sometimes we will get

1	**2**
`get_lock(mx);`	`get_lock(my);`

Concurrency Primitives

Locks

This will probably work most of the time, but occasionally just hangs doing nothing

Sometimes we will get

1	2
`get_lock(mx);`	`get_lock(my);`
`get_lock(my); (waits)`	`get_lock(mx); (waits)`

Concurrency Primitives

Locks

This will probably work most of the time, but occasionally just hangs doing nothing

Sometimes we will get

1	**2**
`get_lock(mx);`	`get_lock(my);`
`get_lock(my); (waits)`	`get_lock(mx); (waits)`

This is simple deadlock, another race condition

Concurrency Primitives

Locks

A very easy error to make, but often very difficult to find, particularly as the locks of `mx` and `my` may be widely separated in the code, or in someone else's code

A very easy error to make, but often very difficult to find, particularly as the locks of `mx` and `my` may be widely separated in the code, or in someone else's code

The use of locks requires a great deal of careful management when the code gets large

Concurrency Primitives

Locks

A very easy error to make, but often very difficult to find, particularly as the locks of `mx` and `my` may be widely separated in the code, or in someone else's code

The use of locks requires a great deal of careful management when the code gets large

Exercise Why wouldn't having another mutex `mxy` to protect both `x` and `y` solve things?

Concurrency Primitives

Locks

If we want to use a lock in portable code, we can use a library specification like *POSIX*

If we want to use a lock in portable code, we can use a library specification like *POSIX*

This is a standard that covers a large number of functions, specifying their use and behaviour

Concurrency Primitives

POSIX `pthread`

The `pthread` section on the POSIX specification contains several functions that we shall soon be looking at:

Concurrency Primitives

POSIX `pthread`

The `pthread` section on the POSIX specification contains several functions that we shall soon be looking at:

- Locks: `pthread_mutex_` `init`, `lock`, `unlock`, `destroy`

Concurrency Primitives

POSIX `pthread`

The `pthread` section on the POSIX specification contains several functions that we shall soon be looking at:

- Locks: `pthread_mutex_` `init`, `lock`, `unlock`, `destroy`
- Barriers: `pthread_barrier_` `init`, `wait`, `destroy`

Concurrency Primitives

POSIX `pthread`

The `pthread` section on the POSIX specification contains several functions that we shall soon be looking at:

- Locks: `pthread_mutex_` `init`, `lock`, `unlock`, `destroy`
- Barriers: `pthread_barrier_` `init`, `wait`, `destroy`
- Condition Variables: `pthread_cond_` `init`, `wait`, `signal`, `broadcast`, `destroy`

Concurrency Primitives

POSIX pthread

The pthread section on the POSIX specification contains several functions that we shall soon be looking at:

- Locks: pthread_mutex_ init, lock, unlock, destroy
- Barriers: pthread_barrier_ init, wait, destroy
- Condition Variables: pthread_cond_ init, wait, signal, broadcast, destroy
- Semaphores: sem_ init, post, wait, destroy

Concurrency Primitives

POSIX `pthread`

The `pthread` section on the POSIX specification contains several functions that we shall soon be looking at:

- Locks: `pthread_mutex_` `init`, `lock`, `unlock`, `destroy`
- Barriers: `pthread_barrier_` `init`, `wait`, `destroy`
- Condition Variables: `pthread_cond_` `init`, `wait`, `signal`, `broadcast`, `destroy`
- Semaphores: `sem_` `init`, `post`, `wait`, `destroy`
- Management: `pthread_` `create`, `join`

Concurrency Primitives

POSIX `pthread`

The `pthread` section on the POSIX specification contains several functions that we shall soon be looking at:

- Locks: `pthread_mutex_` `init`, `lock`, `unlock`, `destroy`
- Barriers: `pthread_barrier_` `init`, `wait`, `destroy`
- Condition Variables: `pthread_cond_` `init`, `wait`, `signal`, `broadcast`, `destroy`
- Semaphores: `sem_` `init`, `post`, `wait`, `destroy`
- Management: `pthread_` `create`, `join`

And many others

Concurrency Primitives

POSIX `pthread`

For example, `pthread_create` (we shall come back to this later)

Concurrency Primitives

POSIX pthread

For example, pthread_create (we shall come back to this later)

```
#include <pthread.h>
int pthread_create(pthread_t *thread,
                   const pthread_attr_t *attr,
                   void *(*start_routine) (void *),
                   void *arg);
```

Concurrency Primitives

POSIX pthread

For example, pthread_create (we shall come back to this later)

```
#include <pthread.h>
int pthread_create(pthread_t *thread,
                   const pthread_attr_t *attr,
                   void *(*start_routine) (void *),
                   void *arg);
```

is how to create a new thread: it takes an *attribute* (always NULL for our purposes), a function of one argument to start executing, and a value to pass as the argument to that function

Concurrency Primitives

POSIX pthread

For example, `pthread_create` (we shall come back to this later)

```
#include <pthread.h>
int pthread_create(pthread_t *thread,
                   const pthread_attr_t *attr,
                   void *(*start_routine) (void *),
                   void *arg);
```

is how to create a new thread: it takes an *attribute* (always `NULL` for our purposes), a function of one argument to start executing, and a value to pass as the argument to that function

It returns a *thread identifier* in the first argument

Concurrency Primitives

POSIX `pthread`

Documentation for POSIX pthread functions is available everywhere, online and possibly on your own computer

Concurrency Primitives

POSIX `pthread`

Documentation for POSIX pthread functions is available everywhere, online and possibly on your own computer

For example, on Linux you can use manual pages, e.g.,
`man pthread_create`
to get detailed information

Concurrency Primitives

POSIX Locks

A real example of locks, as defined by the POSIX standard, where they are called mutexes

```
#include <pthread.h>
pthread_mutex_t mutex;
```

An (uninitialised) mutex

Concurrency Primitives

POSIX Locks

```
int pthread_mutex_init(pthread_mutex_t *restrict mutex,
                       const pthread_mutexattr_t
                             *restrict attr)
```

Initialises the mutex pointed at by the first argument, returns a 0 that indicates success or non-0 to indicate failure

Concurrency Primitives

POSIX Locks

```
int pthread_mutex_init(pthread_mutex_t *restrict mutex,
                       const pthread_mutexattr_t
                             *restrict attr)
```

Initialises the mutex pointed at by the first argument, returns a 0 that indicates success or non-0 to indicate failure

POSIX locks come with various attributes: the default (NULL) is normally what you want

Concurrency Primitives

POSIX Locks

```
int pthread_mutex_init(pthread_mutex_t *restrict mutex,
                       const pthread_mutexattr_t
                             *restrict attr)
```

Initialises the mutex pointed at by the first argument, returns a 0 that indicates success or non-0 to indicate failure

POSIX locks come with various attributes: the default (NULL) is normally what you want

```
pthread_mutex_t mut;
if (pthread_mutex_init(&mut, NULL) != 0) { ...error... }
```

Concurrency Primitives

POSIX Locks

There is a alternative static way to initialise mutexes if all you need is a basic lock:

```
// declare and initialise
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

Concurrency Primitives

POSIX Locks

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

The main grab and free functions

Concurrency Primitives

POSIX Locks

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

The main grab and free functions

It is an error to try and unlock a mutex that is held by another thread: the thread that locks must be the thread that unlocks

Concurrency Primitives

POSIX Locks

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

The main grab and free functions

It is an error to try and unlock a mutex that is held by another thread: the thread that locks must be the thread that unlocks

This is a POSIX specification designed to make locks widely implementable of a variety of architectures

Concurrency Primitives

POSIX Locks

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

The main grab and free functions

It is an error to try and unlock a mutex that is held by another thread: the thread that locks must be the thread that unlocks

This is a POSIX specification designed to make locks widely implementable of a variety of architectures

And this is not a limitation: it is a desired behaviour. If you allowed another thread to unlock a mutex you can bet this would be misused by some programmers thus opening a new opportunity to write buggy code

Concurrency Primitives

POSIX Locks

“It is an error”: some implementations return an error value, while others (depending on the OS) have undefined behaviour

Concurrency Primitives

POSIX Locks

"It is an error": some implementations return an error value, while others (depending on the OS) have undefined behaviour

Some versions of mutexes also allow *recursive* (or *reentrant*) locking, where a thread that already owns a lock can lock it again; it needs to do the same number of unlocks to free the lock

Concurrency Primitives

POSIX Locks

“It is an error”: some implementations return an error value, while others (depending on the OS) have undefined behaviour

Some versions of mutexes also allow *recursive* (or *reentrant*) locking, where a thread that already owns a lock can lock it again; it needs to do the same number of unlocks to free the lock

Non-recursive versions just self-deadlock, or have undefined behaviour

On fairness of POSIX mutexes:

Concurrency Primitives
POSIX Locks

On fairness of POSIX mutexes:

Posix says “the scheduling policy shall determine which thread shall acquire the mutex” if more than one is waiting

Concurrency Primitives

POSIX Locks

On fairness of POSIX mutexes:

Posix says “the scheduling policy shall determine which thread shall acquire the mutex” if more than one is waiting

This allows implementations to take `pthread_attr_setschedpolicy` and thread priorities into account: we shall not talk about that here!

Concurrency Primitives

POSIX Locks

```
int pthread_mutex_trylock(pthread_mutex_t *mutex);
```

Like `pthread_mutex_lock` but return immediately (without getting the lock) if the lock was already held. It returns a value of 0 if it got the lock, a non-zero otherwise

Concurrency Primitives

POSIX Locks

```
int pthread_mutex_trylock(pthread_mutex_t *mutex);
```

Like `pthread_mutex_lock` but return immediately (without getting the lock) if the lock was already held. It returns a value of 0 if it got the lock, a non-zero otherwise

This function is occasionally useful, but less than you might believe, as the result doesn't quite mean what people think it means (sequential assumptions. . .)

Concurrency Primitives

POSIX Locks

It doesn't say "the mutex *is* locked", but really says "the mutex *was* locked"

Concurrency Primitives

POSIX Locks

It doesn’t say “the mutex *is* locked”, but really says “the mutex *was* locked”

It gives the instantaneous state of the lock at the time of the `trylock` function call: it is possible that by the time the calling thread looks at the value that was returned by `trylock` the lock is already free

Concurrency Primitives

POSIX Locks

```
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

Concurrency Primitives

POSIX Locks

```
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

It's good to clear up when you no longer need the mutex as this may free up some system resources

Concurrency Primitives

POSIX Locks

Example code:

```
#include <pthread.h>
...
pthread_mutex_t m;
/* ought to check values returned by these calls */
pthread_mutex_init(&m, NULL);
...
pthread_mutex_lock(&m);
... <CR> ...
pthread_mutex_unlock(&m);
...
pthread_mutex_destroy(&m);
```

We can lock and unlock a mutex as often as we wish: we would typically create it once and use it many times before tidying up

Concurrency Primitives

POSIX Locks

The properties of POSIX locks are specified just to the point to make them useful: in a portable program you can't rely on any feature not explicitly mentioned

Concurrency Primitives

POSIX Locks

The properties of POSIX locks are specified just to the point to make them useful: in a portable program you can't rely on any feature not explicitly mentioned

For example, calling `destroy` on an uninitialised lock; or calling `init` on an already-initialised lock; or destroying a lock while another thread holds it; or using a bitwise copy of a lock structure; and so on

Concurrency Primitives

POSIX Locks

The properties of POSIX locks are specified just to the point to make them useful: in a portable program you can't rely on any feature not explicitly mentioned

For example, calling `destroy` on an uninitialised lock; or calling `init` on an already-initialised lock; or destroying a lock while another thread holds it; or using a bitwise copy of a lock structure; and so on

Remember that a lot of machines don't have the nice predictable architecture of a PC

Concurrency Primitives

POSIX Locks

The properties of POSIX locks are specified just to the point to make them useful: in a portable program you can't rely on any feature not explicitly mentioned

For example, calling `destroy` on an uninitialised lock; or calling `init` on an already-initialised lock; or destroying a lock while another thread holds it; or using a bitwise copy of a lock structure; and so on

Remember that a lot of machines don't have the nice predictable architecture of a PC

And even PC architectures are very complicated these days

Concurrency Primitives
POSIX `pthread`

Exercise Read about `pthread_spin_lock` and `pthread_rwlock`

Advanced Exercise Think about mutexes in the context of async programming, where we have concurrency (but not necessarily parallelism) and we require threads never to block

How to make threads

Now we have been introduced to POSIX, we need to take a little diversion from talking about primitives to cover something essential to parallelism

How to make threads

Now we have been introduced to POSIX, we need to take a little diversion from talking about primitives to cover something essential to parallelism

Namely, how do we create new threads to run?

How to make threads

Now we have been introduced to POSIX, we need to take a little diversion from talking about primitives to cover something essential to parallelism

Namely, how do we create new threads to run?

As always, a simple idea that can have unexpected consequences

How to make threads

Now we have been introduced to POSIX, we need to take a little diversion from talking about primitives to cover something essential to parallelism

Namely, how do we create new threads to run?

As always, a simple idea that can have unexpected consequences

We shall look at the POSIX mechanism

Concurrency Control

POSIX

Creating threads:

```
#include <pthread.h>
int pthread_create(pthread_t *thread,
                   const pthread_attr_t *attr,
                   void *(*start_routine) (void *),
                   void *arg);
```

Link with `-lpthread`

Concurrency Control
POSIX

Creating threads:

```
#include <pthread.h>
int pthread_create(pthread_t *thread,
                   const pthread_attr_t *attr,
                   void *(*start_routine) (void *),
                   void *arg);
```

Link with `-lpthread`

This looks ugly, but is quite simple in practice: it creates a new thread running the function `start_routine` on the argument `arg`

It returns a thread identifier in argument `thread`. This can be used to do things to the thread

Concurrency Control

It returns a thread identifier in argument `thread`. This can be used to do things to the thread

`attr` is a thread attribute: you probably will never need more than the default (NULL), but occasionally you might (stack size; detached thread)

Concurrency Control

POSIX

It returns a thread identifier in argument `thread`. This can be used to do things to the thread

`attr` is a thread attribute: you probably will never need more than the default (NULL), but occasionally you might (stack size; detached thread)

The `start_routine` names a function of one argument that the thread will start executing when it begins running

Concurrency Control

POSIX

It returns a thread identifier in argument `thread`. This can be used to do things to the thread

`attr` is a thread attribute: you probably will never need more than the default (NULL), but occasionally you might (stack size; detached thread)

The `start_routine` names a function of one argument that the thread will start executing when it begins running

The `arg` is the argument passed to the function (a pointer)

Concurrency Control

POSIX

Roughly:

```
void *hello(void *n)
{
  printf("hello %d\n", *(int*)n);
  return n;
}

int main(void)
{
  int m;
  pthread_t thr;

  m = 1;
  // should check return value from create ...
  pthread_create(&thr, NULL, hello, (void*)&m);
  ...
}
```

Concurrency Control
POSIX

`pthread_create` returns (pretty much) immediately with an error code, 0 indicating success `pthread_create` returns (pretty much) immediately with an error code, 0 indicating success

It makes a new thread that runs separately from the main thread

Concurrency Control

POSIX

`pthread_create` returns (pretty much) immediately with an error code, 0 indicating success

It makes a new thread that runs separately from the main thread

Possibly simultaneously with the main thread, depending on the number of cores and the OS's scheduling

It runs the function `hello` with argument a pointer to `m`

Concurrency Control

POSIX

It runs the function `hello` with argument a pointer to `m`

It does this concurrently with the `main` function, which continues to run

Concurrency Control

POSIX

It runs the function `hello` with argument a pointer to `m`

It does this concurrently with the `main` function, which continues to run

The `start_function` will generally call lots of other functions to perform whatever the thread needs to do

Concurrency Control
POSIX

It runs the function `hello` with argument a pointer to `m`

It does this concurrently with the `main` function, which continues to run

The `start_function` will generally call lots of other functions to perform whatever the thread needs to do

Ugly type casting is common in C

Threads

Aside

This also works on uniprocessor systems: the threads are scheduled in a similar way to processes

Threads

Aside

This also works on uniprocessor systems: the threads are scheduled in a similar way to processes

You can debug a concurrent program on a sequential machine, but it may not exhibit some of the more subtle race conditions or deadlocks as the threads won't truly be running in parallel

Threads

Aside

You can make more threads than there are cores: for example, run 10 (or 1000) threads on a 4 core machine

Threads

Aside

You can make more threads than there are cores: for example, run 10 (or 1000) threads on a 4 core machine

And the OS will schedule between the threads

Threads

Aside

You can make more threads than there are cores: for example, run 10 (or 1000) threads on a 4 core machine

And the OS will schedule between the threads

A thread that is blocked (e.g., waiting on a lock) typically would not be scheduled, so it uses no CPU cycles

Threads

Aside

You can make more threads than there are cores: for example, run 10 (or 1000) threads on a 4 core machine

And the OS will schedule between the threads

A thread that is blocked (e.g., waiting on a lock) typically would not be scheduled, so it uses no CPU cycles

The question remains whether that is worth it or not to have more threads than cores, as both creating threads and OS scheduling eats up CPU time

Threads

Aside

You can make more threads than there are cores: for example, run 10 (or 1000) threads on a 4 core machine

And the OS will schedule between the threads

A thread that is blocked (e.g., waiting on a lock) typically would not be scheduled, so it uses no CPU cycles

The question remains whether that is worth it or not to have more threads than cores, as both creating threads and OS scheduling eats up CPU time

A common error is to create hundreds of threads and then wonder why everything is running slowly

Threads

Aside

You can make more threads than there are cores: for example, run 10 (or 1000) threads on a 4 core machine

And the OS will schedule between the threads

A thread that is blocked (e.g., waiting on a lock) typically would not be scheduled, so it uses no CPU cycles

The question remains whether that is worth it or not to have more threads than cores, as both creating threads and OS scheduling eats up CPU time

A common error is to create hundreds of threads and then wonder why everything is running slowly

Threads create concurrency, not parallelism