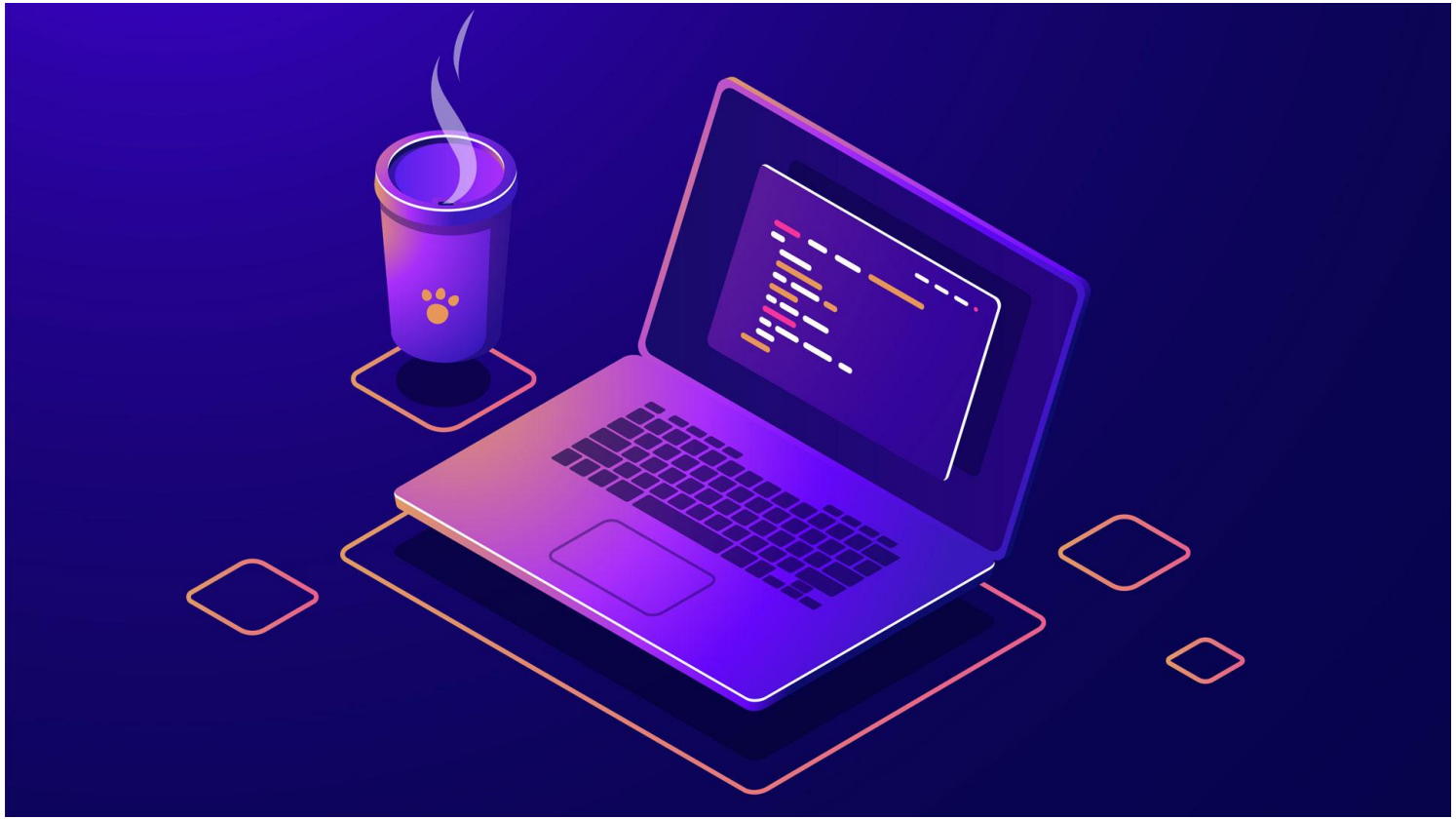
Locks



Locks

Locks: special variables that live in memory with two basic operations:

- Acquire: while a thread has the lock, nobody else gets it; this allows the thread to do its work in private, not bothered by other threads
- Release: allow other threads to acquire the lock and do their work (one at a time) in private

Type definition and instrinsics:

```
void omp_init_lock(omp_lock_t *lock)
void omp_destroy_lock(omp_lock_t *lock)

void omp_set_lock(omp_lock_t *lock)
void omp_unset_lock(omp_lock_t *lock)

int omp_test_lock(omp_lock_t *lock)
```

Locks

OpenMP provides lock primitives for low-level synchronization

`omp_init_lock`	Initialize the lock
`omp_set_lock`	Acquires the lock
`omp_unset_lock`	Releases the lock
`omp_test_lock`	Tries to acquire the lock (won't block)
`omp_destroy_lock`	Frees lock resources

Example

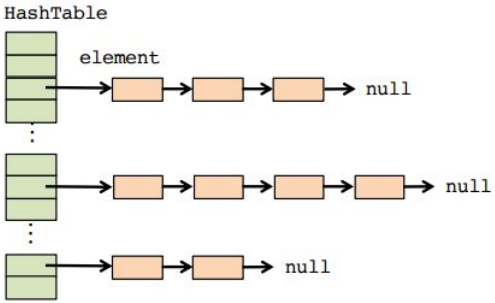
```
#include <omp.h>
void foo ()
{
    omp_lock_t lock;

    omp_init_lock(&lock);
    #pragma omp parallel
    {
        omp_set_lock(&lock);
        // mutual exclusion region
        omp_unset_lock(&lock);
    }
    omp_destroy_lock(&lock);
}
```

Lock must be initialized before being used

Only one thread at a time here

Reducing task interactions: serialization

Example: inserting elements in hash table defined as a collection of linked lists




```
typedef struct {
    int data;
    element *next;
} element;

int dataTable[SIZE_TABLE];
element * HashTable[SIZE_HASH];

for (i = 0; i < SIZE_TABLE; i++) {
    int index = hash_function (dataTable[i], SIZE_HASH);
    insert_element (dataTable[i], index, HashTable);
}
```

Reducing task interactions: serialization

Easily parallelizable using an iterative task decomposition using `taskloop`. However ...

- ... updates to the list in any particular slot must be protected to prevent a race condition

```
typedef struct {
    int data;
    element *next;
} element;

int dataTable[SIZE_TABLE];
element * HashTable[SIZE_HASH];

#pragma omp taskloop
for (i = 0; i < elements; i++) {
    int index = hash_function (dataTable[i], SIZE_HASH);
    #pragma omp critical  // atomic not possible here
    insert_element (dataTable[i], index, HashTable);
}
```

- Serialization in the insertion of elements

Reducing task interactions: serialization

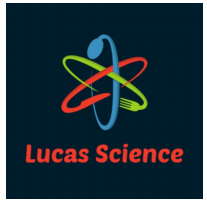
Associate a lock variable with each slot in the hash table, protecting the chain of elements in an slot

```
omp_lock_t hash_lock[SIZE_HASH];

#pragma omp parallel
#pragma omp single
{
for (i = 0; i < SIZE_HASH; i++) omp_init_lock(&hash_lock[i]);

#pragma omp taskloop
for (i = 0; i < SIZE_TABLE; i++) {
    int index = hash_function (dataTable[i], SIZE_HASH);
    omp_set_lock (&hash_lock[index]);
    insert_element (dataTable[i], index, HashTable);
    omp_unset_lock (&hash_lock[index]);
}

for (i = 0; i < SIZE_HASH; i++) omp_destroy_lock(&hash_lock[i]);
}
```

Threads may be inserting elements into the hash table in parallel, as long as these elements hash to different slots

Instructor Social Media

Youtube: Lucas Science



Instagram: lucaasbazilio

Twitter: lucasebazilio