UNIMORE
UNIVERSITÀ DEGLI STUDI DI
MODENA E REGGIO EMILIA

Paralellism and POSIX Threads

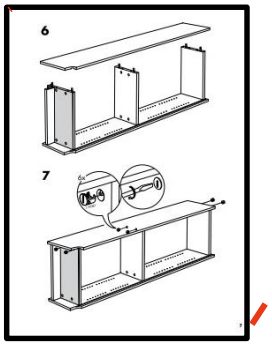
Paolo Burgio
paolo.burgio@unimore.it

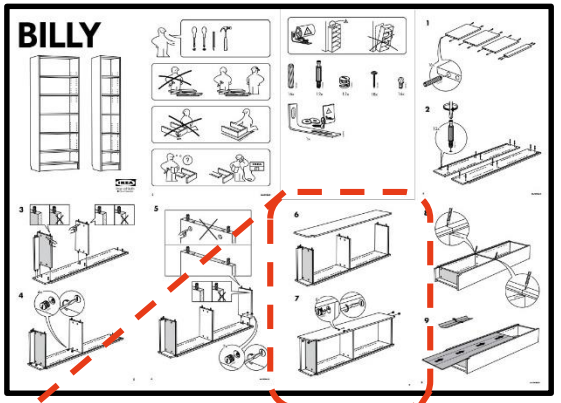
What is…

- ✓ ..a core?
- ✓ …a program?
- ✓ …a process?
- ✓ …a thread?
- ✓ ..a task?

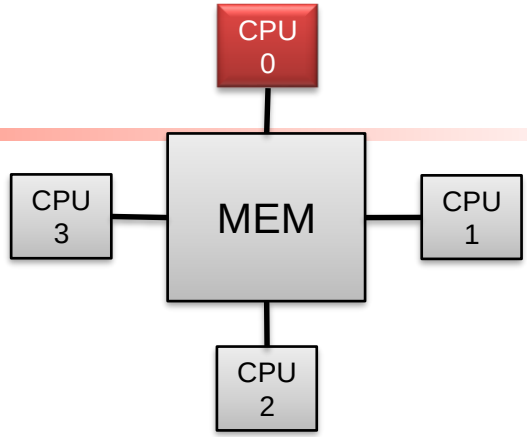
What is…

- ✓ ..a core?
  - An electronic circuit to execute instruction (=> programs)
- ✓ …a program?
  - The implementation of an algorithm
- ✓ …a process?
  - A program that is executing
- ✓ …a thread?
  - A unit of execution (of a process)
- ✓ ..a task?
  - A unit of work (of a program)

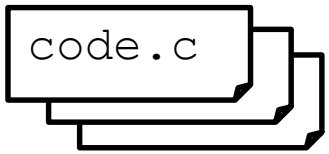




What is…



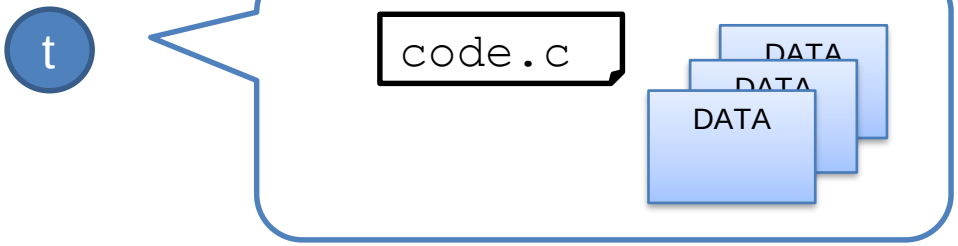
- ✓ ..a core?
- ✓ …a program?



- ✓ …a process?

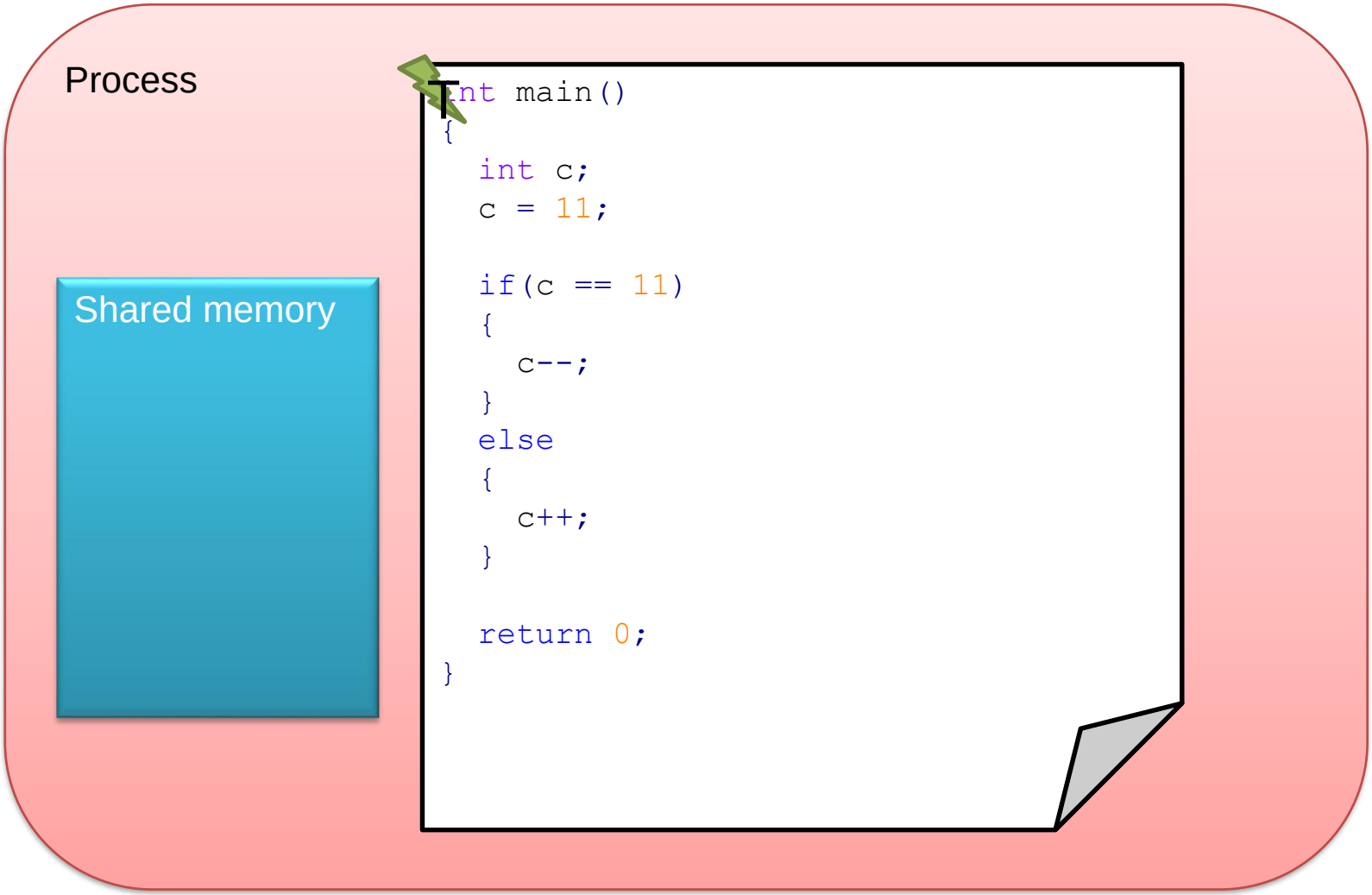
- ✓ …a thread?

- ✓ ..a task?



Process

- ✓ "A program that executes"



Process

- ✓ "A program that executes"

Process

Shared memory

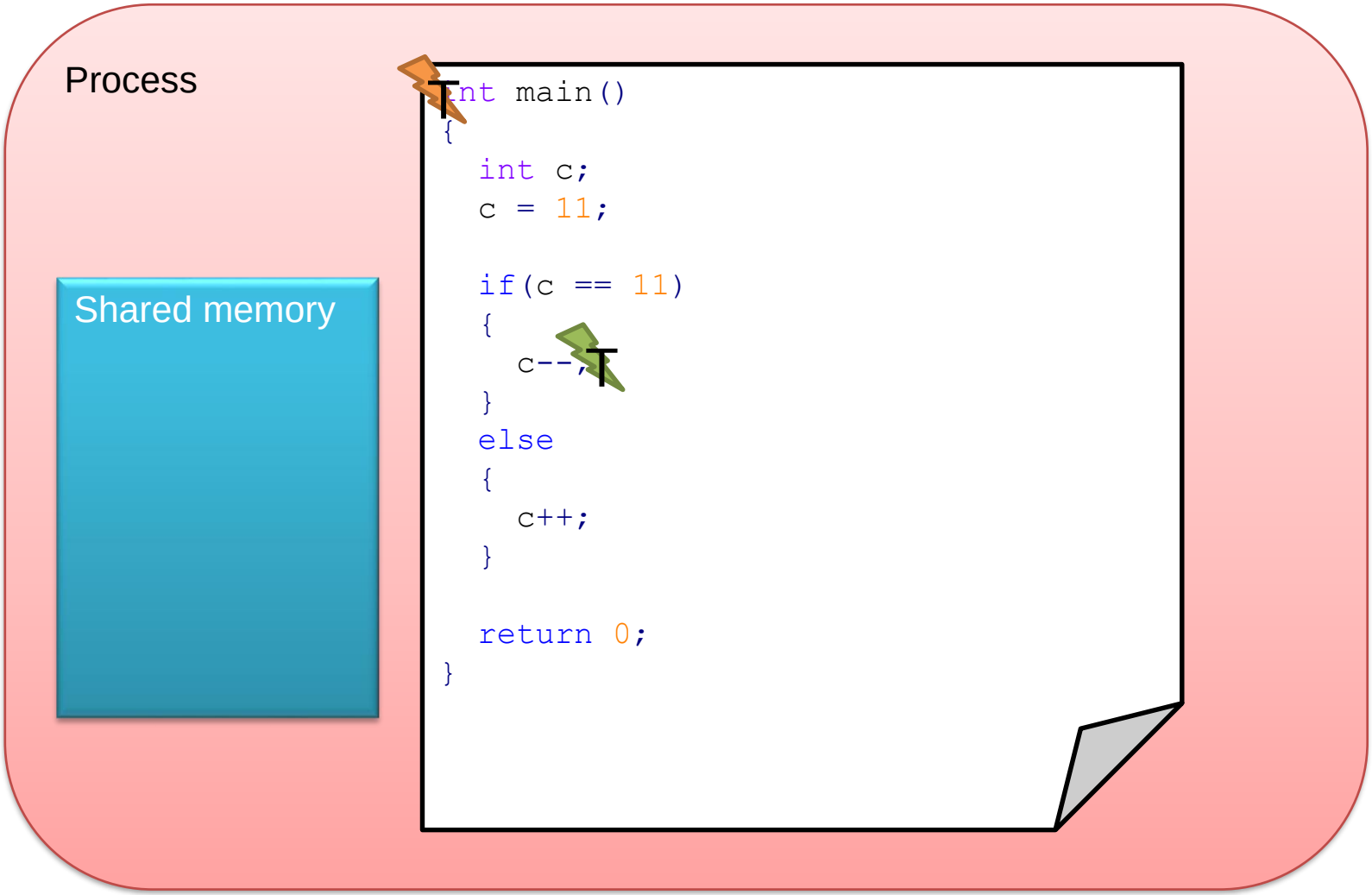
```
int main()
{
  int c;
  c = 11;

  if(c == 11)
  {
    c--;
  }
  else
  {
    c++;
  }

  return 0;
}
```

Process

- ✓ "A program that executes"



Process

- ✓ "A program that executes"

Process

Shared memory

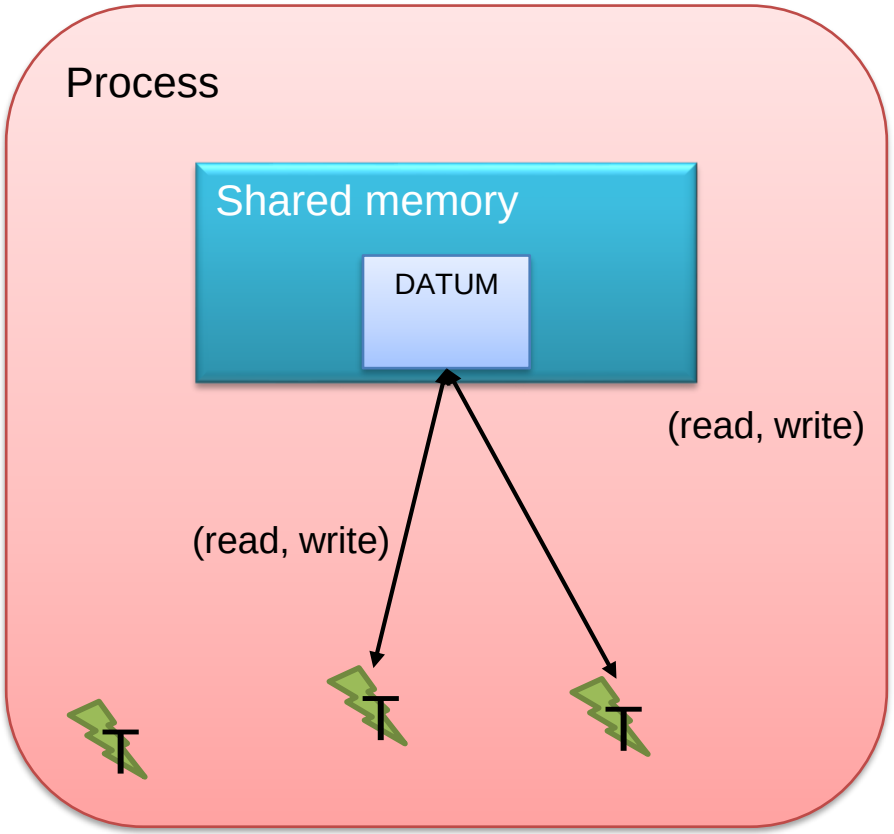
```
int main()
{
  int c;
  c = 11;

  if(c == 11)
  {
    c--;
  }
  else
  {
    c++;
  }

  return 0;
}
```

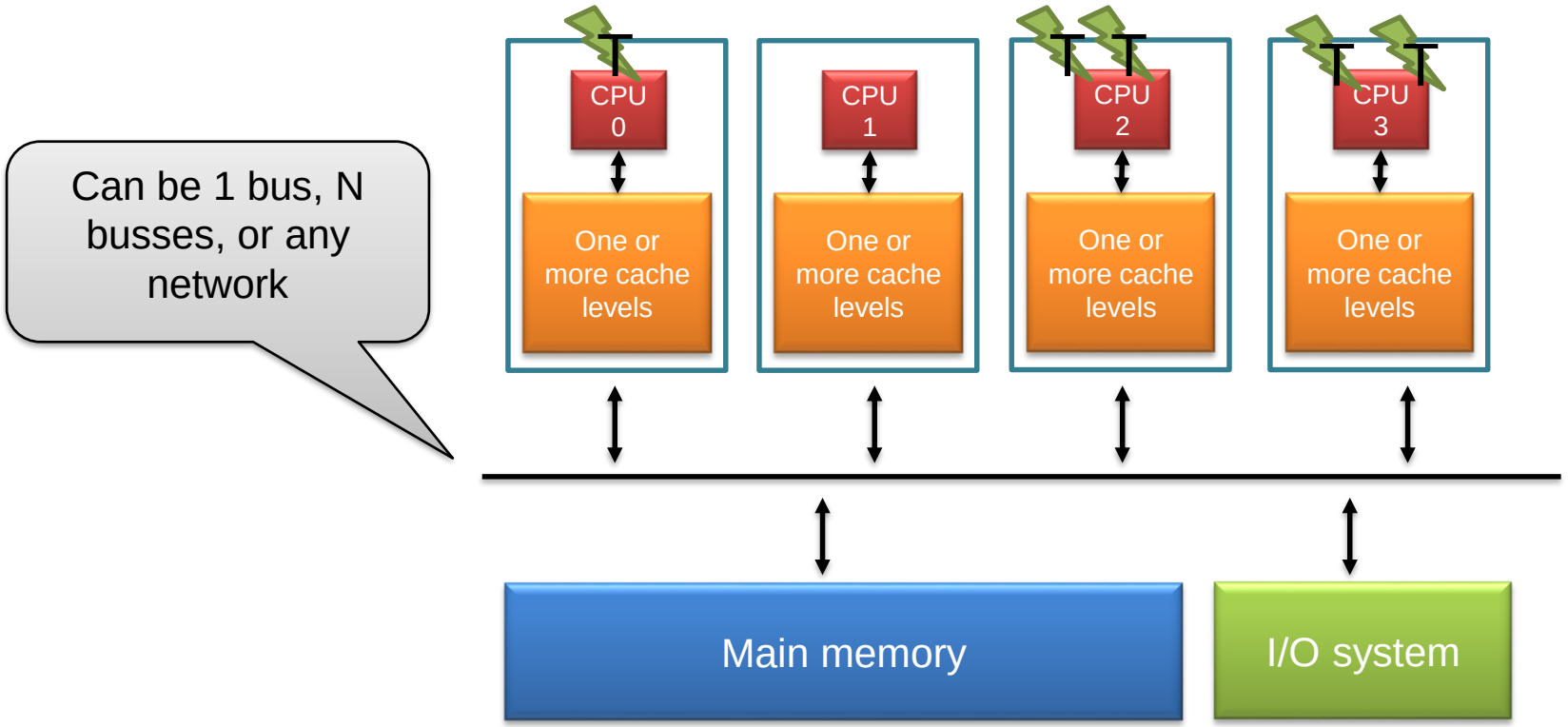
Shared memory

- ✓ Processes have a given amount of RAM memory that can be shared among concurrent threads



Symmetric multi-processing

- ✓ Memory: centralized with uniform access time (UMA) and bus interconnect, I/O
  - Typically, multi-core (sub)systems (8-16 cores)
  - Examples: Sun Enterprise 6000, SGI Challenge, Intel



UNIMORE
UNIVERSITÀ DEGLI STUDI DI
MODENA E REGGIO EMILIA

Ok, let's go

The POSIX IEEE standard

- ✓ Specifies an operating system interface similar to most UNIX systems
  - *It extends the C language with primitives that allows the specification of the concurrency*

- ✓ POSIX distinguishes between the terms process and thread
  - "A process is an address space with one or more threads executing in that address space"
  - "A thread is a single flow of control within a process (a unit of execution)"

- ✓ Every process has at least one thread
  - the "`main()`" (aka "master") thread; its termination ends the process
  - All the threads share the same address space, and have a private stack

The PThread library

- ✓ The pthread primitives are usually implemented into a pthread library
- ✓ All the declarations of the primitives cited in these slides can be found into `sched.h`, `pthread.h` and `semaphore.h`
  - Use `man` to get on-line documentation
- ✓ When compiling under gcc & GNU/Linux, remember the `-lpthread` option!

UNIMORE
UNIVERSITÀ DEGLI STUDI DI MODENA E REGGIO EMILIA

PThread creation, join, end

PThread body

- ✓ A (P)thread is identified by a C function, called body:

```
void *my_pthread_fn(void *arg)
{
  …
}
```

- ✓ A thread starts with the first instruction of its body

- ✓ The threads ends when the body function ends
  - – it's not the only way a thread can die

Thread creation

- ✓ Thread can be created using the primitive

```
int pthread_create ( pthread_t *ID,
                     pthread_attr_t *attr,
                     void *(*body)(void *),
                     void * arg
                    );
```

- ✓ `pthread_t` is the type that contains the thread ID
- ✓ `pthread_attr_t` is the type that contains the parameters of the thread
- ✓ `arg` is the argument passed to the thread `body` when it starts

Thread attributes

- ✓ Thread attributes specifies the characteristics of a thread
  - Stack size and address
  - Detach state (joinable or detached)
  - Scheduling parameters (priority, …)

- ✓ Attributes must be initialized and destroyed
  - `int pthread_attr_init(pthread_attr_t *attr);`
  - `int pthread_attr_destroy(pthread_attr_t *attr);`

Thread termination

- ✓ A thread can terminate itself calling

```
void pthread_exit(void *retval);
```

- ✓ When the thread body ends after the last "`}`", `pthread_exit()` is called implicitly
- ✓ Exception: when `main()` terminates, `exit()` is called implicitly

Thread IDs

- ✓ Each thread has a unique ID
- ✓ The thread ID of the current thread can be obtained using

```
pthread_t pthread_self(void);
```

- ✓ Two thread IDs can be compared using

```
int pthread_equal(   pthread_t thread1,
                     pthread_t thread2  );
```

Joining a thread

- ✓ A thread can wait the termination of another thread using

```
int pthread_join ( pthread_t th,
                   void **thread_return);
```

- ✓ It gets the return value of the thread or `PTHREAD_CANCELED` if the thread has been killed

- ✓ By default, every thread must be joined
  - The join frees all the internal resources
  - Stack, registers, and so on

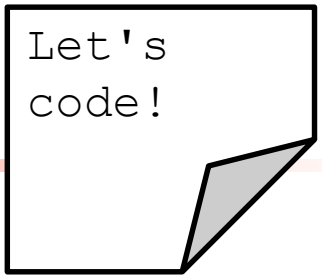
Joining a thread (2)

- ✓ A thread that does not need to be joined has to be declared as detached

- ✓ 2 ways to have it:
  - While creating (in father thread) using `pthread_attr_setdetachstate()`
  - The thread itself can become detached calling in its body `pthread_detach()`

- ✓ Joining a detached thread returns an error

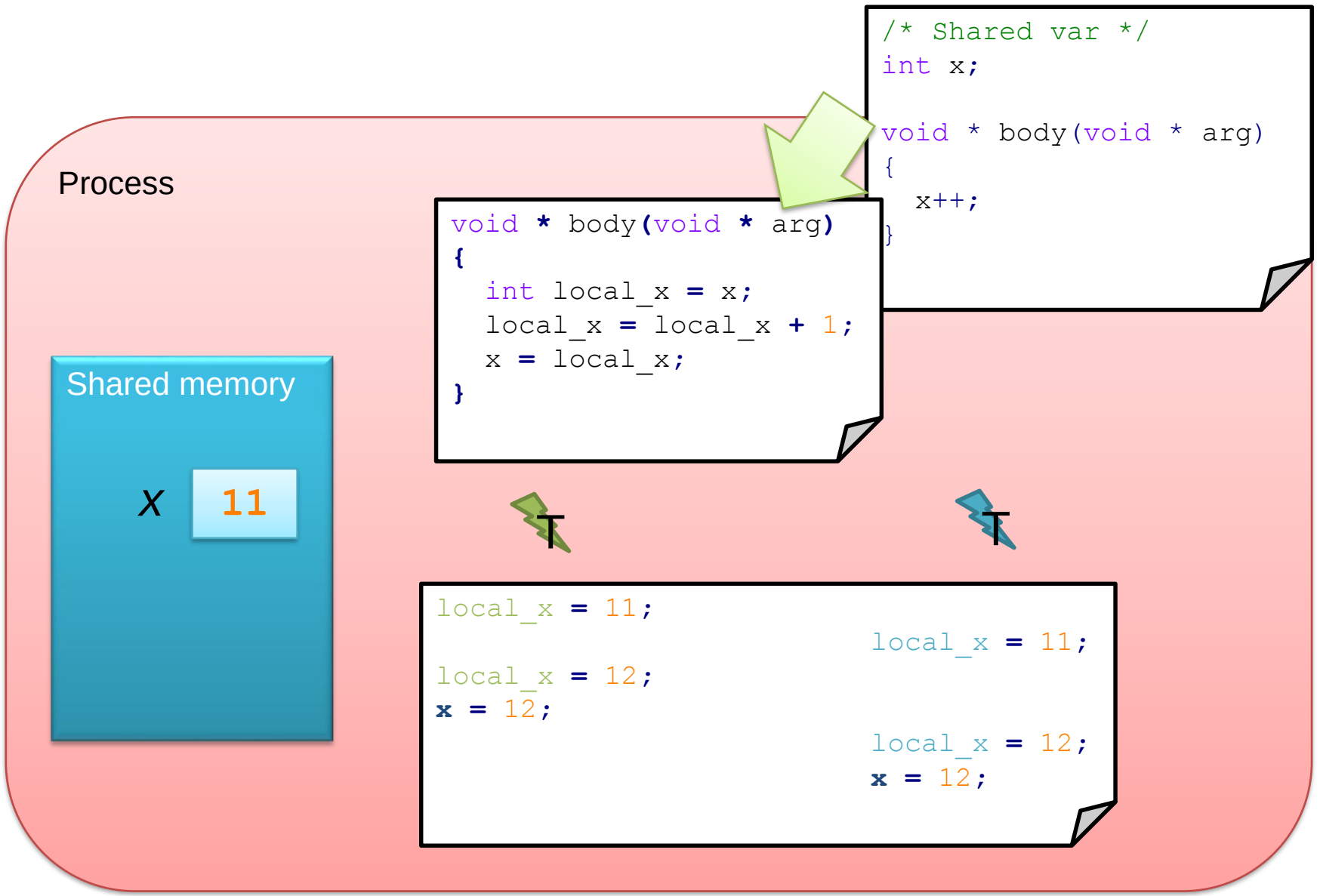
Example



- ✓ Filename: `ex_create.c`
- ✓ The demo explains how to create a thread
  - the `main()` thread creates another thread (called `body()`)
  - the `body()` thread checks the thread Ids using `pthread_equal()` and then ends
  - the `main()` thread joins the `body()` thread

- ✓ *Credits to PJ*

The "variable increment" problem



UNIMORE
UNIVERSITÀ DEGLI STUDI DI
MODENA E REGGIO EMILIA

POSIX semaphores

Semaphores

- ✓ A semaphore is a counter managed with a set of primitives
- ✓ It is used for
  - Synchronization
  - Mutual exclusion
- ✓ POSIX Semaphores can be
  - Unnamed (local to a process)
  - Named (shared between processed through a file descriptor)

Unnamed semaphores

- ✓ Mainly used with multithread applications
- ✓ Operations permitted:
  - initialization /destruction
  - blocking wait / nonblocking wait
    - counter decrement
  - post
    - counter increment
  - counter reading
    - simply returns the counter

Initializing a semaphore

- ✓ The `sem_t` type contains all the semaphore data structures

```
int sem_init(sem_t *sem, int pshared,
             unsigned int value);
```

  - `pshared` is `0` if `sem` is not shared between processes

```
int sem_destroy(sem_t *sem)
```

  - It destroys the `sem` semaphore

Semaphore waits

```
int sem_wait(sem_t *sem);
int sem_trywait(sem_t *sem);
```

- ✓ Under the hood..
- ✓ Blocks the calling thread
- ✓ If the counter is greater than 0 the thread does not block
  - `sem_trywait` never blocks

Other semaphore primitives

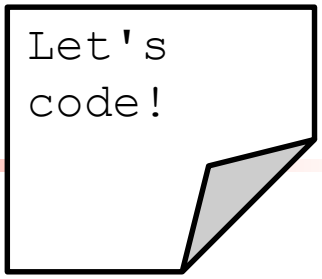
```
int sem_post(sem_t *sem);
```

- It increments the semaphore counter
- It unblocks a waiting thread

```
int sem_getvalue(sem_t *sem,int *val);
```

- It simply returns the semaphore counter

Example



- ✓ Filename: `ex_sem.c`
- ✓ In this example, semaphores are used to implement mutual exclusion in the output of a character in the console.