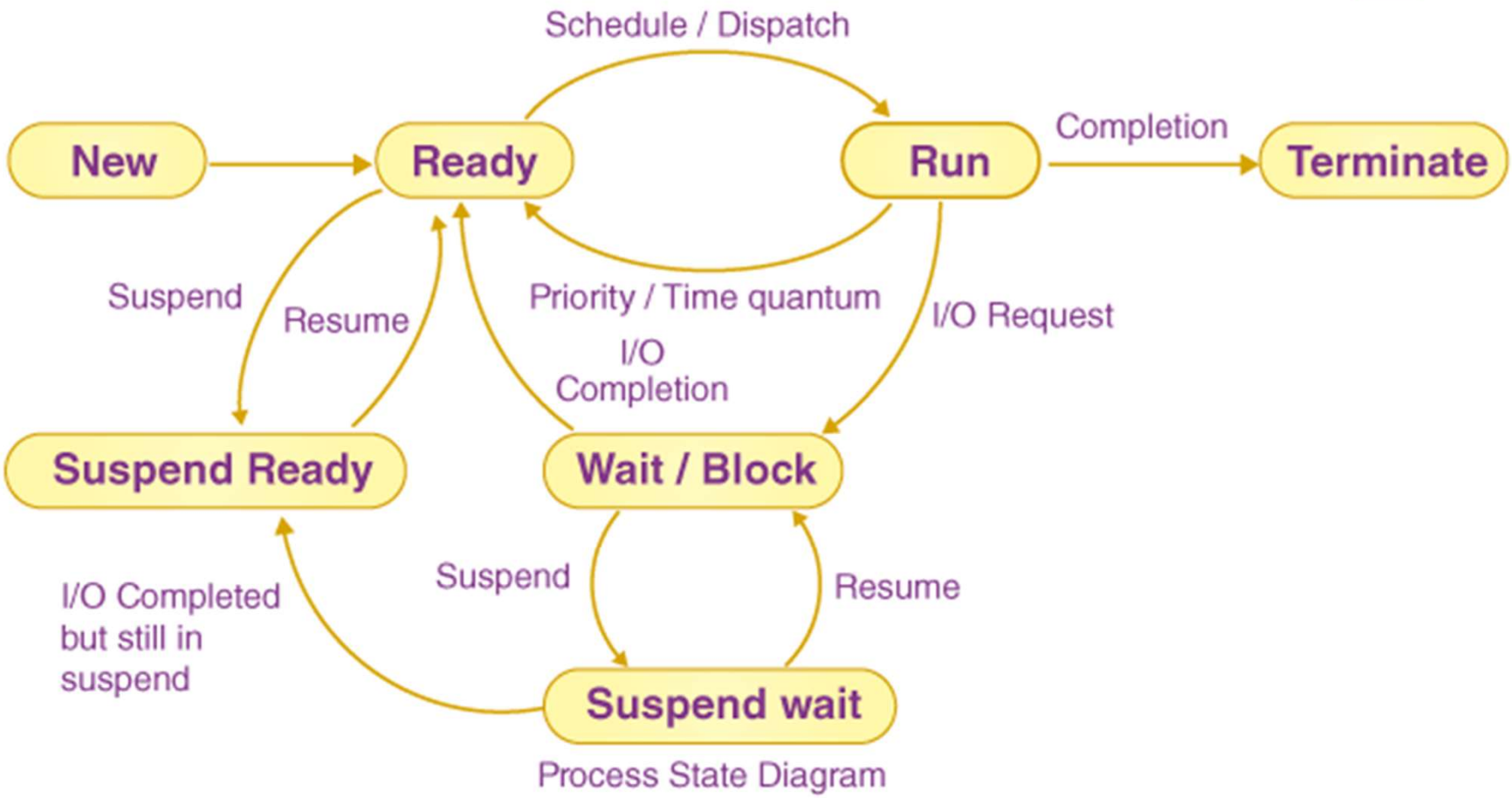
Realtime Mastering

C++ Threads, Concurrency, Free RTOS, Port to STM, BSP Layers, GFx

ABOUT - Mr. Kyaw Swar Tun

- Graduated Bachelor Degree Of Computer Science
- Red Hat Certified System Administrator
- Embedded Linux System Engineer
- Embedded Developer (MCU, MPU)
- Student at Bachelor Engineering
- System Engineer at
- ROM Dynamics – Robotics Company Limited.
- Experience From Age 16 to now.



Process State Diagram

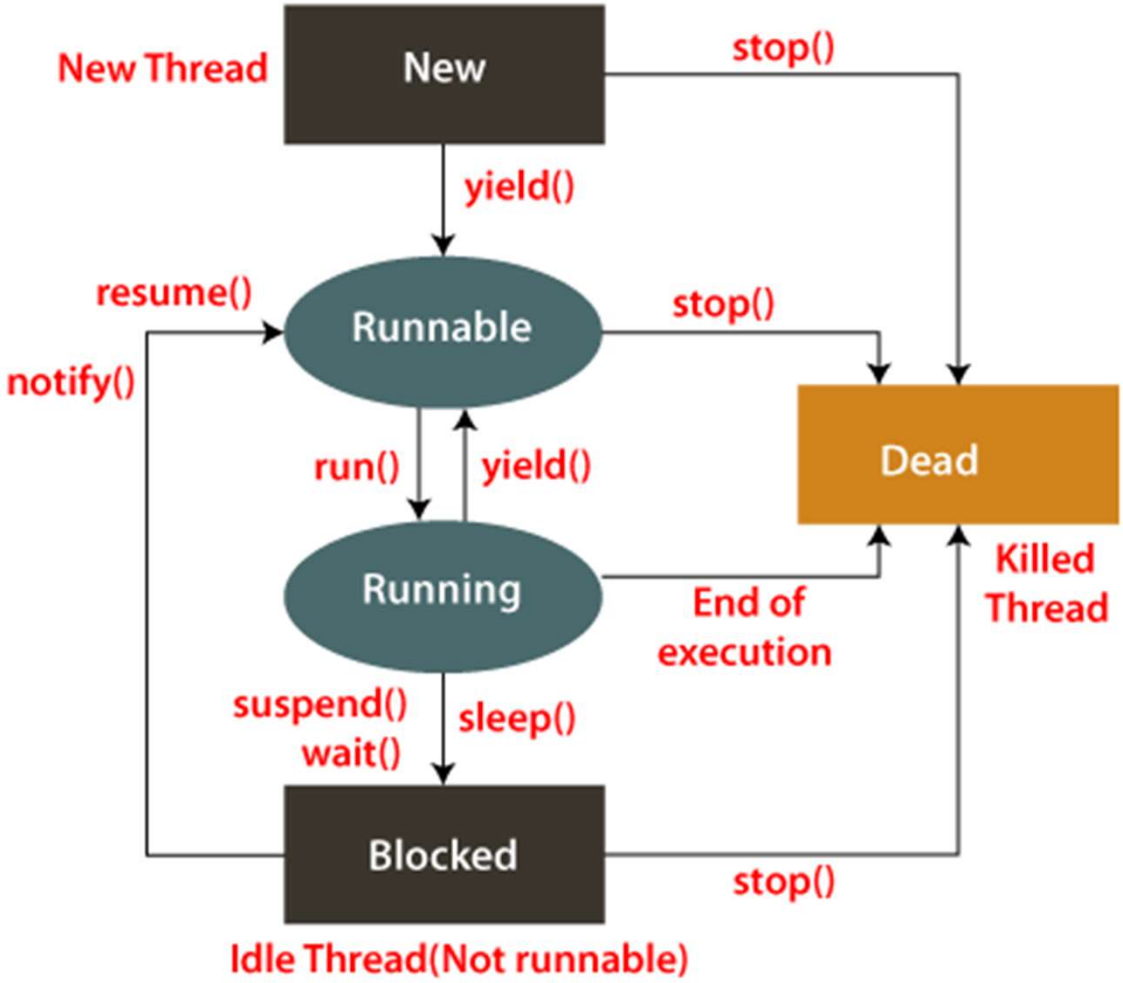
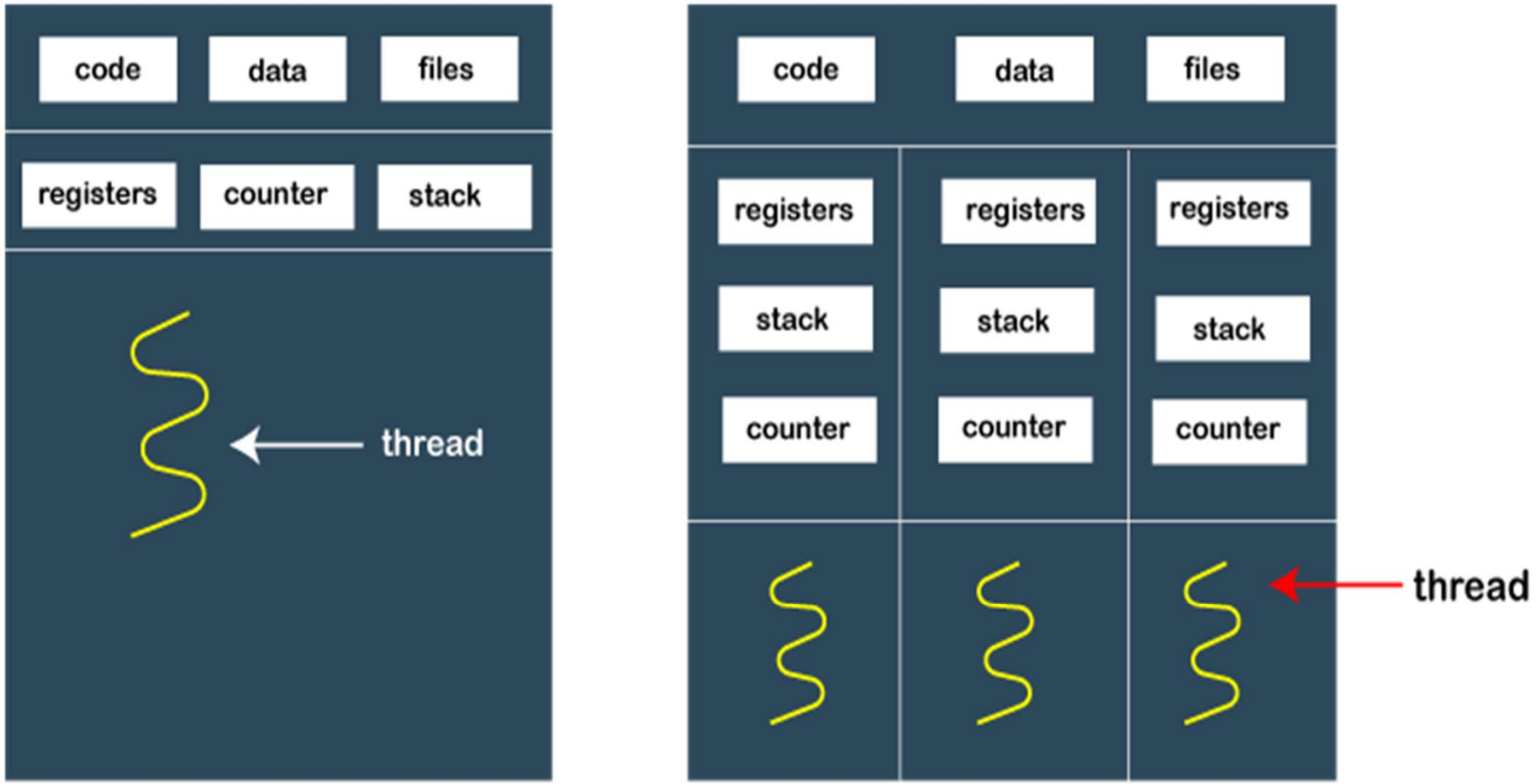


Fig: State Transition Diagram of a Thread



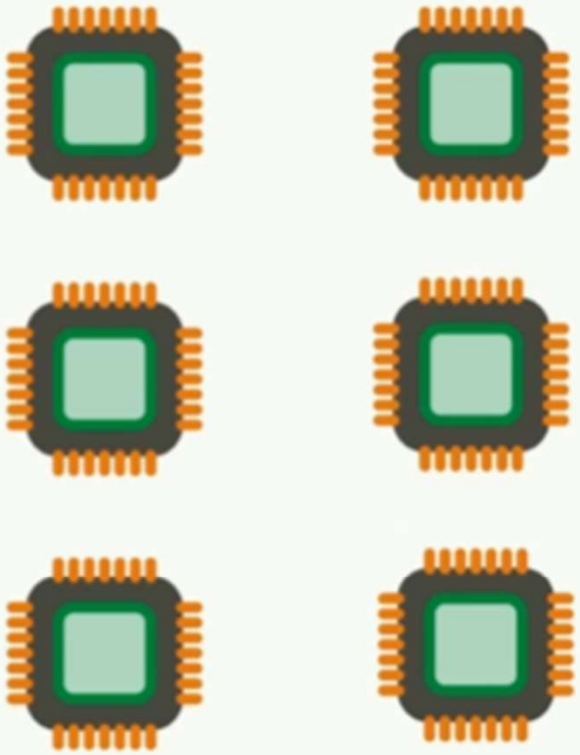
Single-threaded process

Multi-threaded process

- ***Thread*** of execution is the smallest sequence of programmed instructions that can be managed independently by a scheduler.

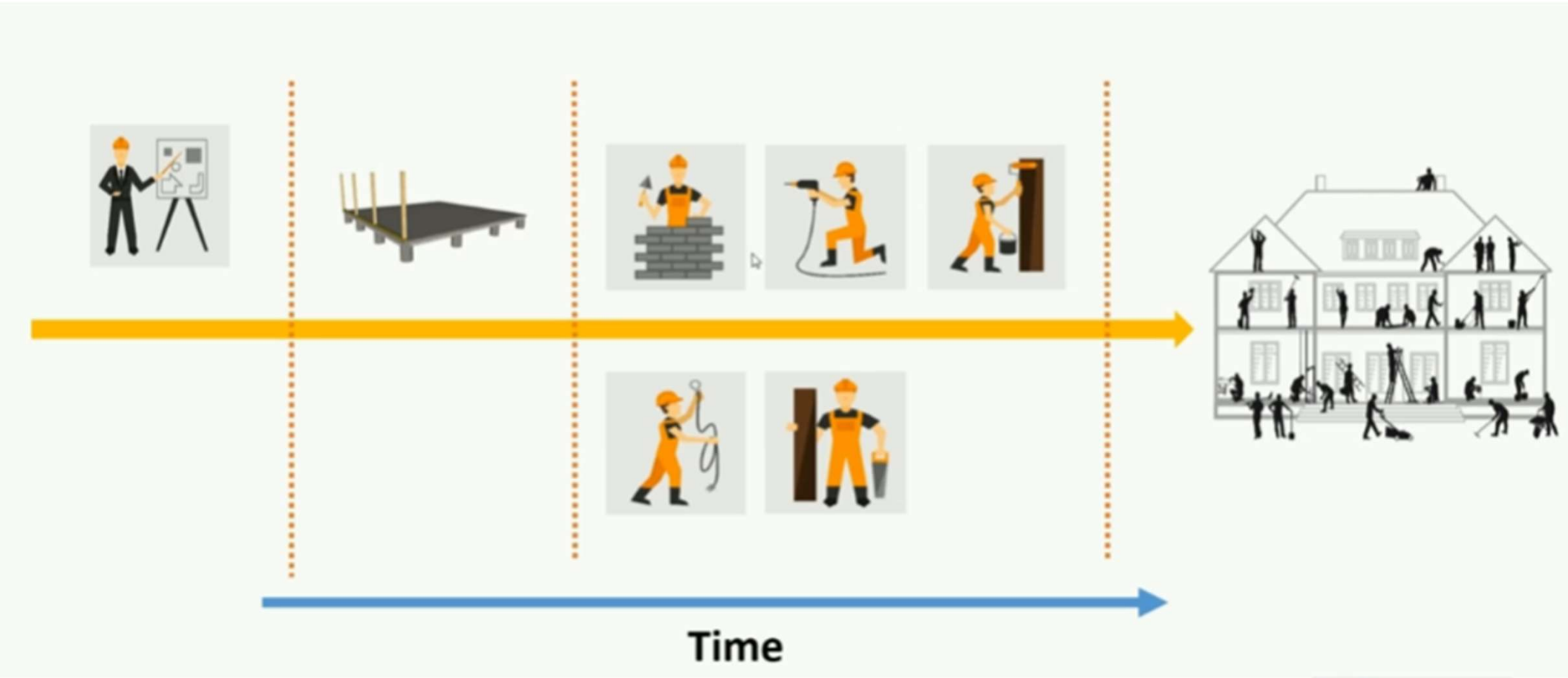
- Thread is component of a process. Every process have at least on thread called ***main thread*** which is the entry point for the program.

Multiple threads



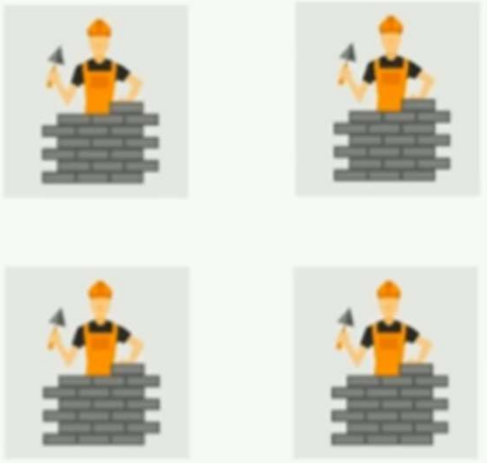
Multiple cores

Parallel world


Time

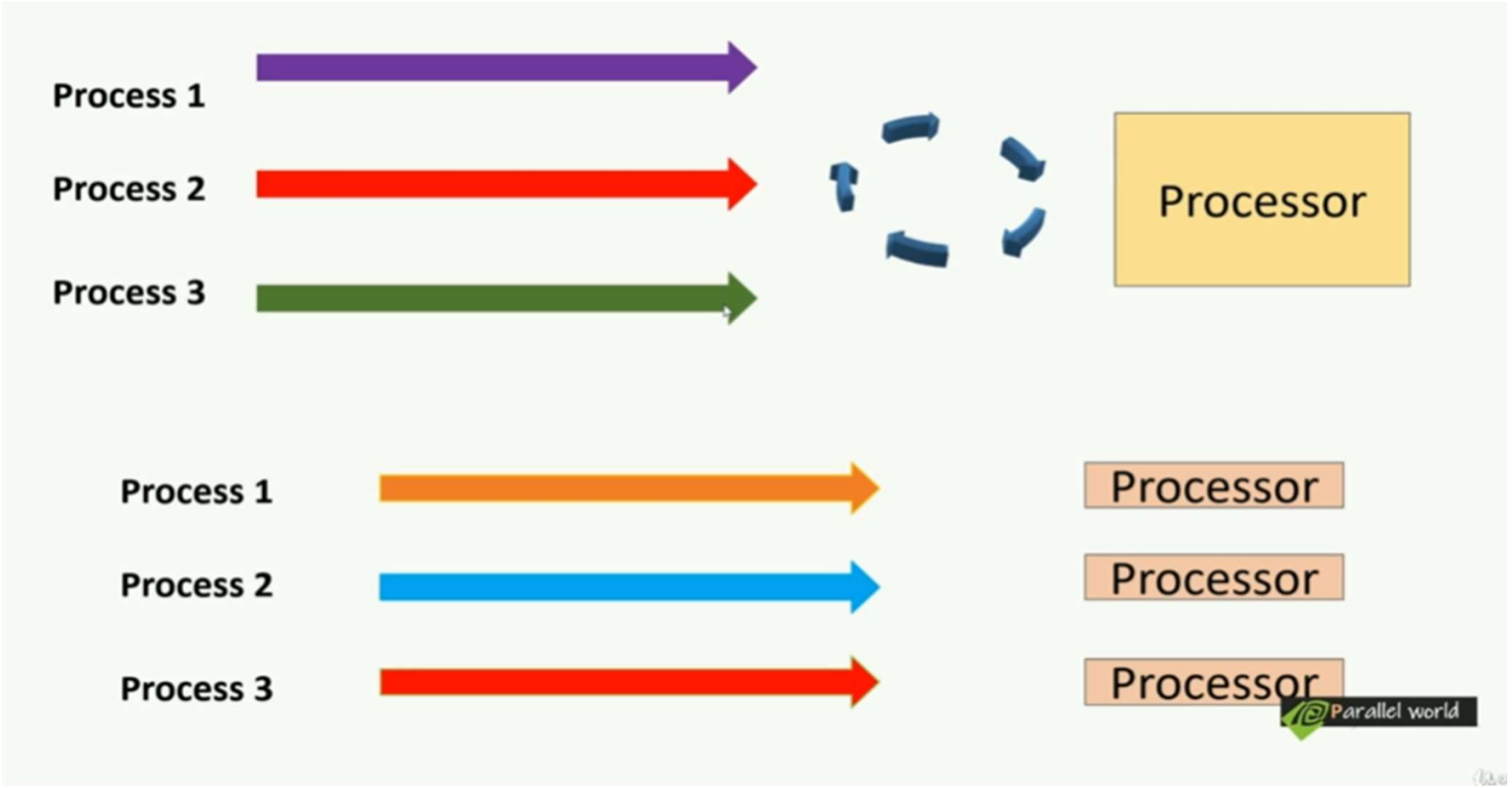
Task level and data level parallelism

Different task on same or different data

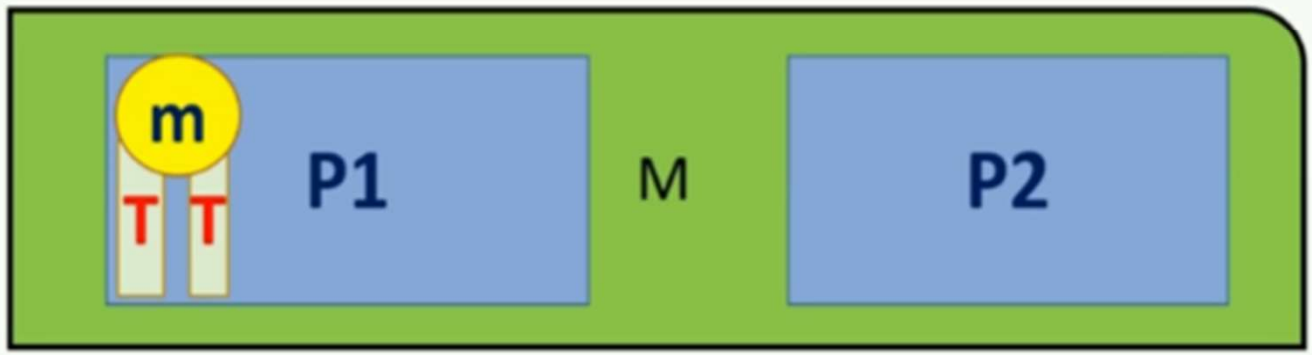


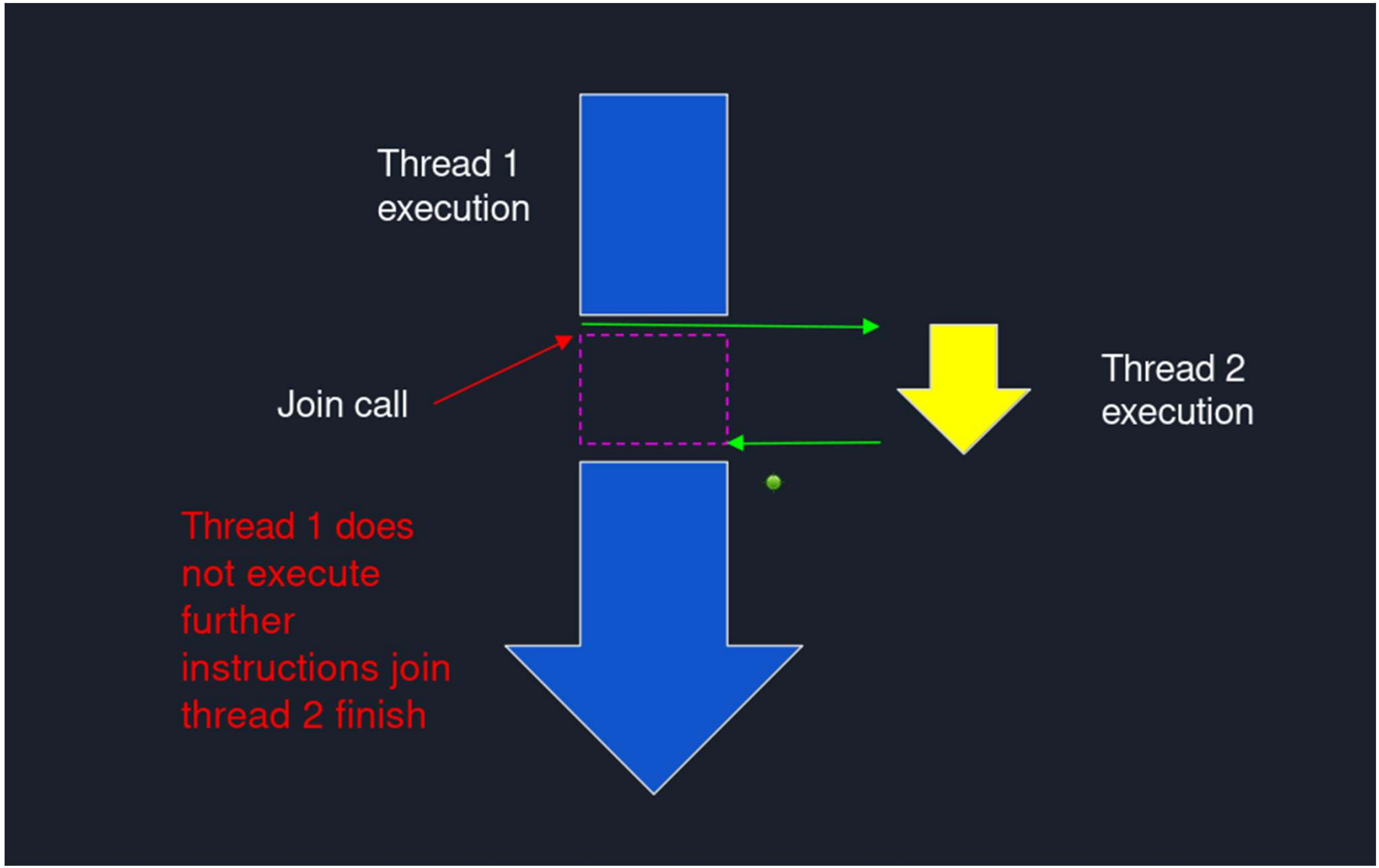
Same task on different data

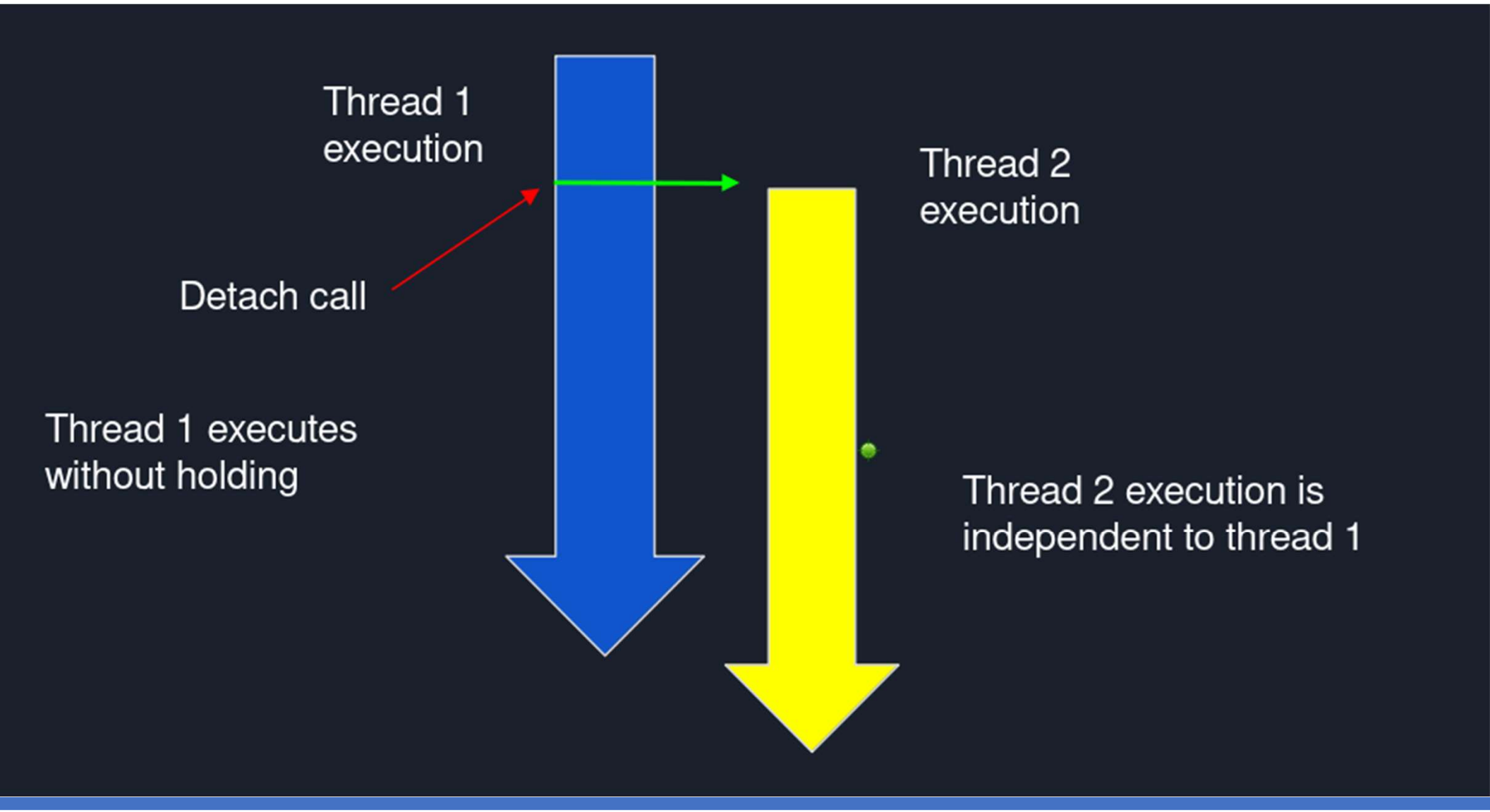
Concurrency Vs Parallelism

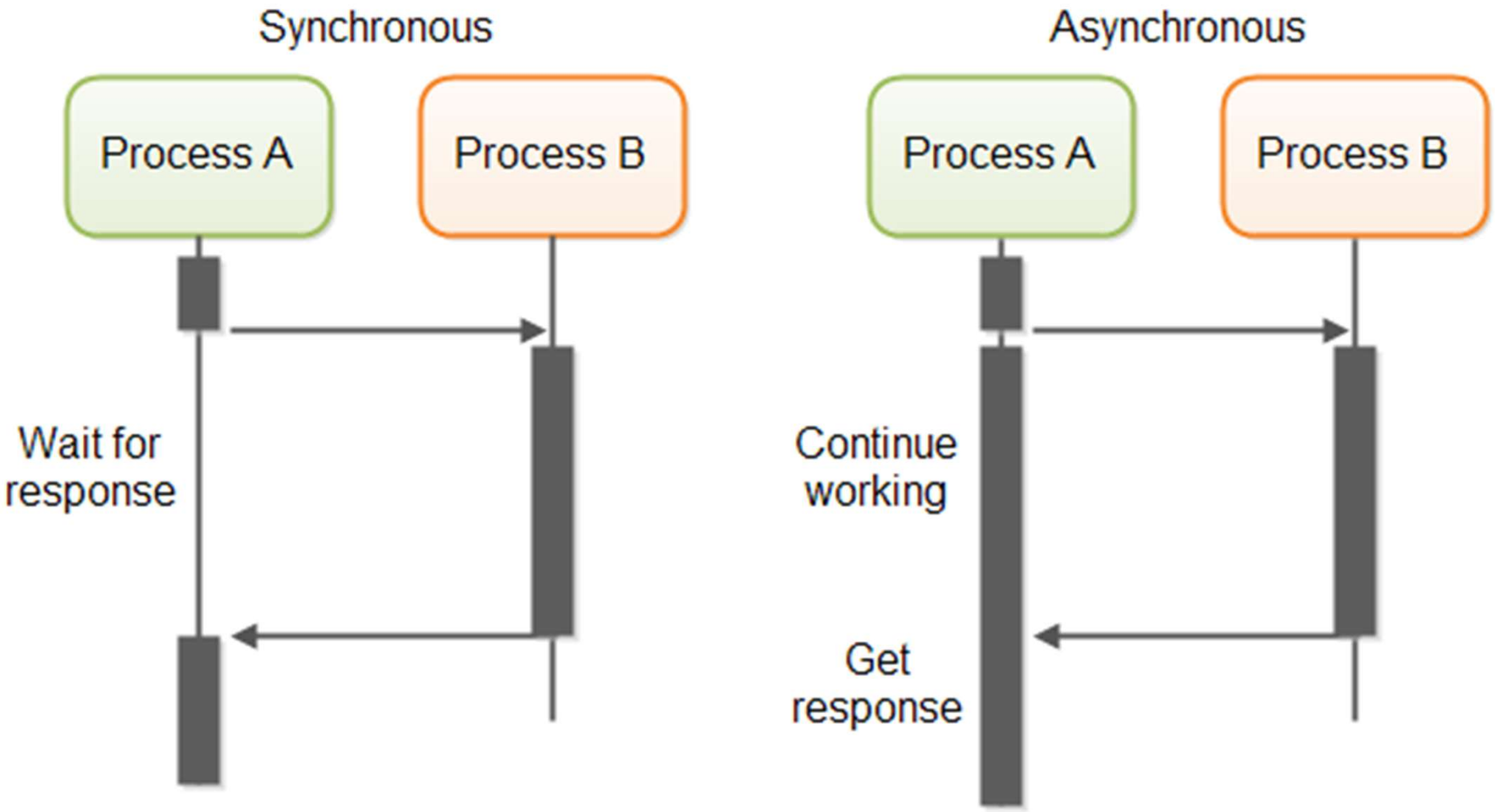


The typical difference between the threads and process is that threads (of the same process) run in a ***shared memory space***, while processes run in ***separate memory spaces***




Thread 1
execution
Join call
Thread 2
execution
Thread 1 does
not execute
further
instructions join
thread 2 finish


Thread 1
execution
Thread 2
execution
Detach call
Thread 1 executes
without holding
Thread 2 execution is
independent to thread 1


Synchronous
Process A
Process B
Wait for response
Asynchronous
Process A
Process B
Continue working
Get response

Concurrency support library (since C++11)

C++ includes built-in support for threads, atomic operations, mutual exclusion, condition variables, and futures.

Threads

Threads enable programs to execute across several processor cores.

Defined in header <thread>

thread (C++11)	manages a separate thread (class)
jthread (C++20)	std::thread with support for auto-joining and cancellation (class)

Functions managing the current thread

Defined in namespace this_thread

yield (C++11)	suggests that the implementation reschedule execution of threads (function)
get_id (C++11)	returns the thread id of the current thread (function)
sleep_for (C++11)	stops the execution of the current thread for a specified time duration (function)
sleep_until (C++11)	stops the execution of the current thread until a specified time point (function)

Member functions

(constructor)	constructs new `thread` object (public member function)
(destructor)	destructs the thread object, underlying thread must be joined or detached (public member function)
`operator=`	moves the thread object (public member function)

Observers

`joinable`	checks whether the thread is joinable, i.e. potentially running in parallel context (public member function)
`get_id`	returns the *id* of the thread (public member function)
`native_handle`	returns the underlying implementation-defined thread handle (public member function)
`hardware_concurrency` [static]	returns the number of concurrent threads supported by the implementation (public static member function)

Operations

`join`	waits for the thread to finish its execution (public member function)
`detach`	permits the thread to execute independently from the thread handle (public member function)
`swap`	swaps two thread objects (public member function)

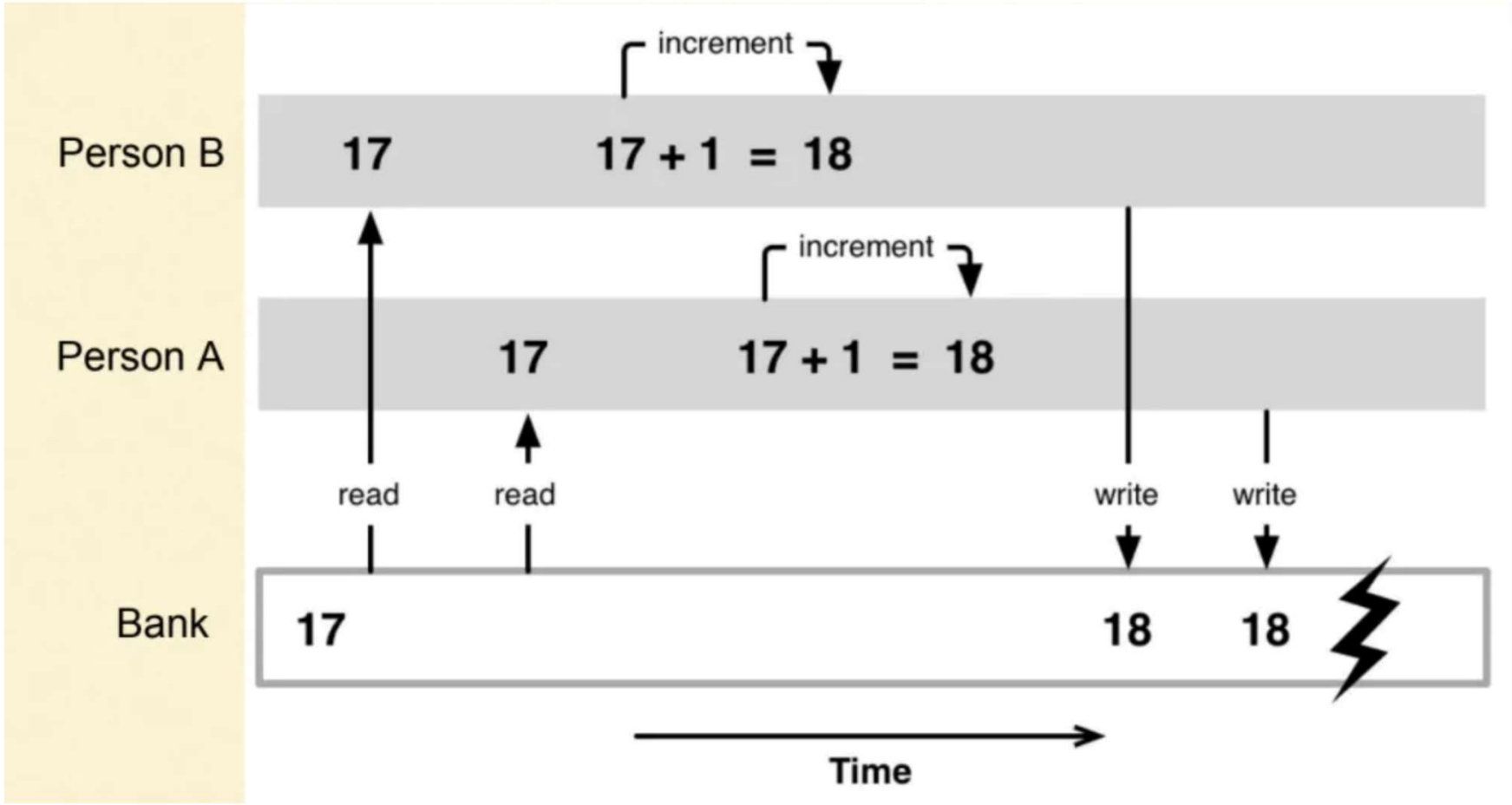
Thread cancellation

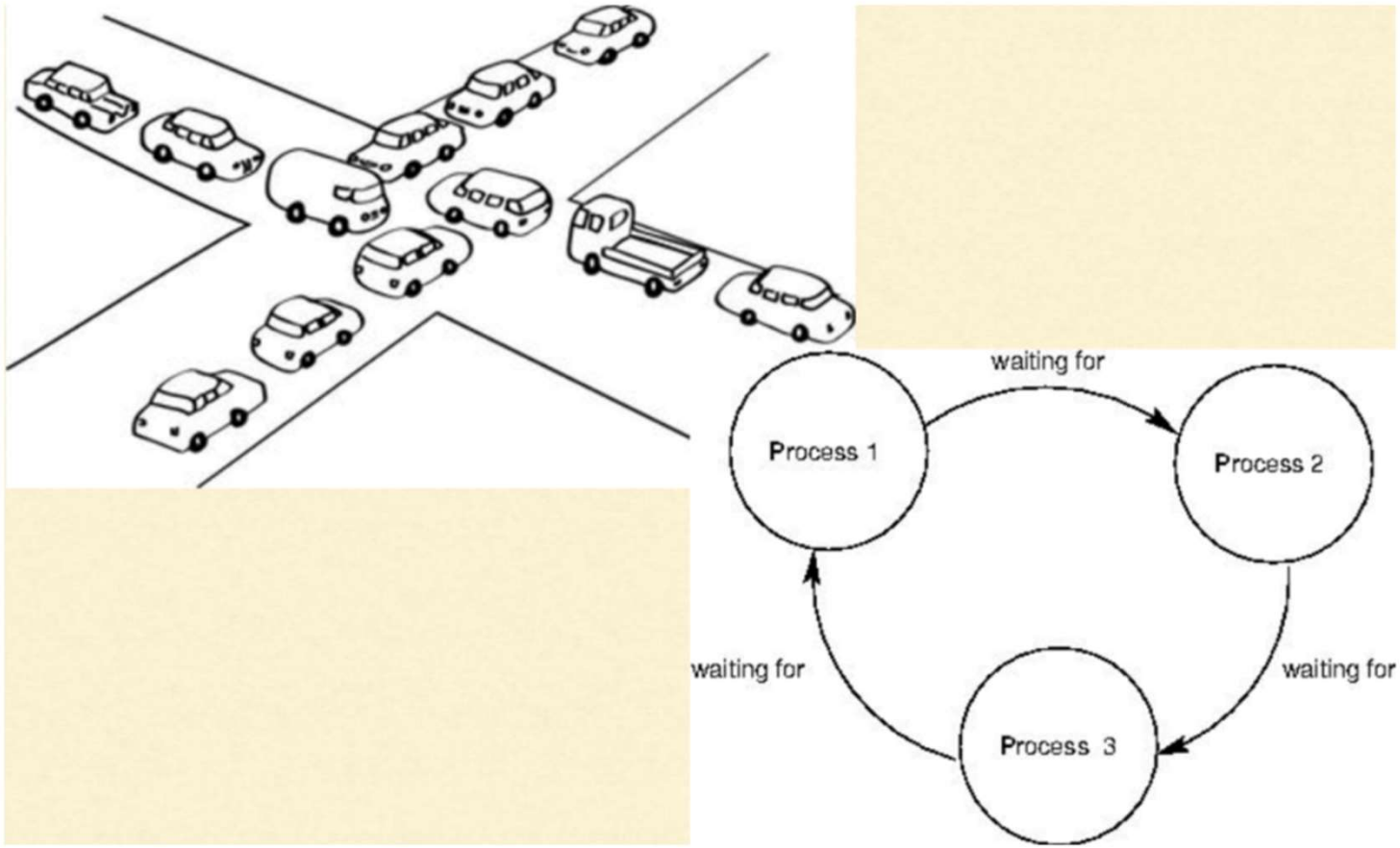
The `stop_XXX` types are designed to enable thread cancellation for `std::jthread`, although they can also be used independently of `std::jthread` - for example to interrupt `std::condition_variable_any` waiting functions, or for a custom thread management implementation. In fact they do not even need to be used to "stop" anything, but can instead be used for a thread-safe one-time function(s) invocation trigger, for example. (since C++20)

Defined in header `<stop_token>`

`stop_token` (C++20)	an interface for querying if a `std::jthread` cancellation request has been made (class)
`stop_source` (C++20)	class representing a request to stop one or more `std::jthread`s (class)
`stop_callback` (C++20)	an interface for registering callbacks on `std::jthread` cancellation (class template)

1–1 Race condition




waiting for
Process 1
Process 2
waiting for
waiting for
Process 3

Deadlock

Mutual Exclusion (Mutex)

- algorithms prevent multiple threads from simultaneously accessing shared resources.
- prevents data racesprevents data races
- synchronization between threads.

Mutual Exclusion (CLASS)

- mutex
- time_mutex
- recursive_mutex
- recursive_time_mutex
- shared_mutex
- shared_time_mutex

Member functions

(constructor)	constructs the mutex (public member function)
(destructor)	destroys the mutex (public member function)
operator= [deleted]	not copy-assignable (public member function)

Locking

lock	locks the mutex, blocks if the mutex is not available (public member function)
try_lock	tries to lock the mutex, returns if the mutex is not available (public member function)
unlock	unlocks the mutex (public member function)

Mutual Exclusion (Generic Lock)

- lock_guard
- scope_lock
- unique_lock

Mutual Exclusion (Locking Strategy)

- defer_lock
- try_to_lock
- adopt_lock

Communication between threads

- Condition Variable

- Semaphore

Condition Variable

- multiple threads to communicate with each other.

- allows some number of threads to wait (possibly with a timeout) for notification from another thread that they may proceed.

condition_variable (C++11)	provides a condition variable associated with a `std::unique_lock` (class)
condition_variable_any (C++11)	provides a condition variable associated with any lock type (class)
notify_all_at_thread_exit (C++11)	schedules a call to `notify_all` to be invoked when this thread is completely finished (function)
cv_status (C++11)	lists the possible results of timed waits on condition variables (enum)

Notification

notify_one	notifies one waiting thread (public member function)
notify_all	notifies all waiting threads (public member function)

waiting

wait

blocks the current thread until the condition variable is awakened

(public member function)

wait_for

blocks the current thread until the condition variable is awakened or after the specified timeout duration

(public member function)

wait_until

blocks the current thread until the condition variable is awakened or until specified time point has been reached

(public member function)

Asynchronous Operations

- Future
- Promise
- Package tasks
- Async

promise(C++11)

stores a value for asynchronous retrieval

(class template)

packaged_task(C++11)

packages a function to store its return value for asynchronous retrieval

(class template)

future(C++11)

waits for a value that is set asynchronously

(class template)

shared_future(C++11)

waits for a value (possibly referenced by other futures) that is set asynchronously

(class template)

async(C++11)

runs a function asynchronously (potentially in a new thread) and returns

a std::future that will hold the result

(function template)

launch(C++11)

specifies the launch policy for std::async

(enum)

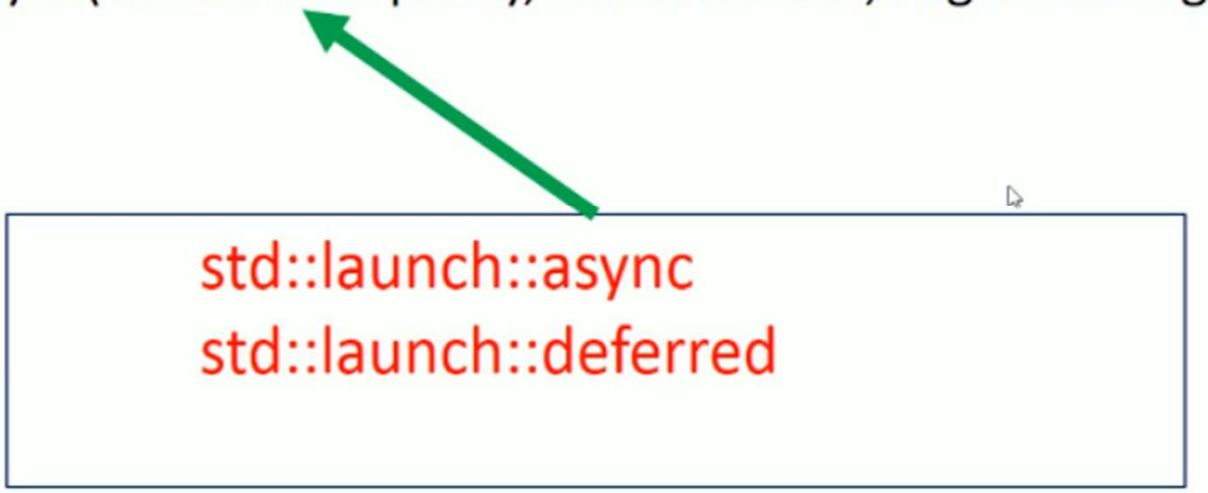
future_status(C++11)

specifies the results of timed waits performed on std::future and std::shared_future

(enum)

asynchronous operation in c++

- In c++ asynchronous operation can be created via
  - std::async
    - async(std::launch policy, Function&& f, Args&&... args);



Package_task

- The class template std::packaged_task wraps any Callable target so that it can be invoked asynchronously.

- It's return value or exception thrown, is stored in a shared state which can be accessed through std::future objects.

- class packaged_task<R(Args...)>;

  std::packaged_task<int(int,int)> task(callable object)

- Each std::promise object is paired with a std::future object.

- A thread with access to the std::future object can wait for the result to be set, while another thread that has access to the corresponding std::promise object can call set_value() to store the value and make the future ready.

Future Errors

Future errors

future_error (C++11)	reports an error related to futures or promises (class)
future_category (C++11)	identifies the future error category (function)
future_errc (C++11)	identifies the future error codes (enum)

Semaphore (A Phoe Gyi)

a lightweight synchronization primitive used to constrain concurrent access to a shared resource.

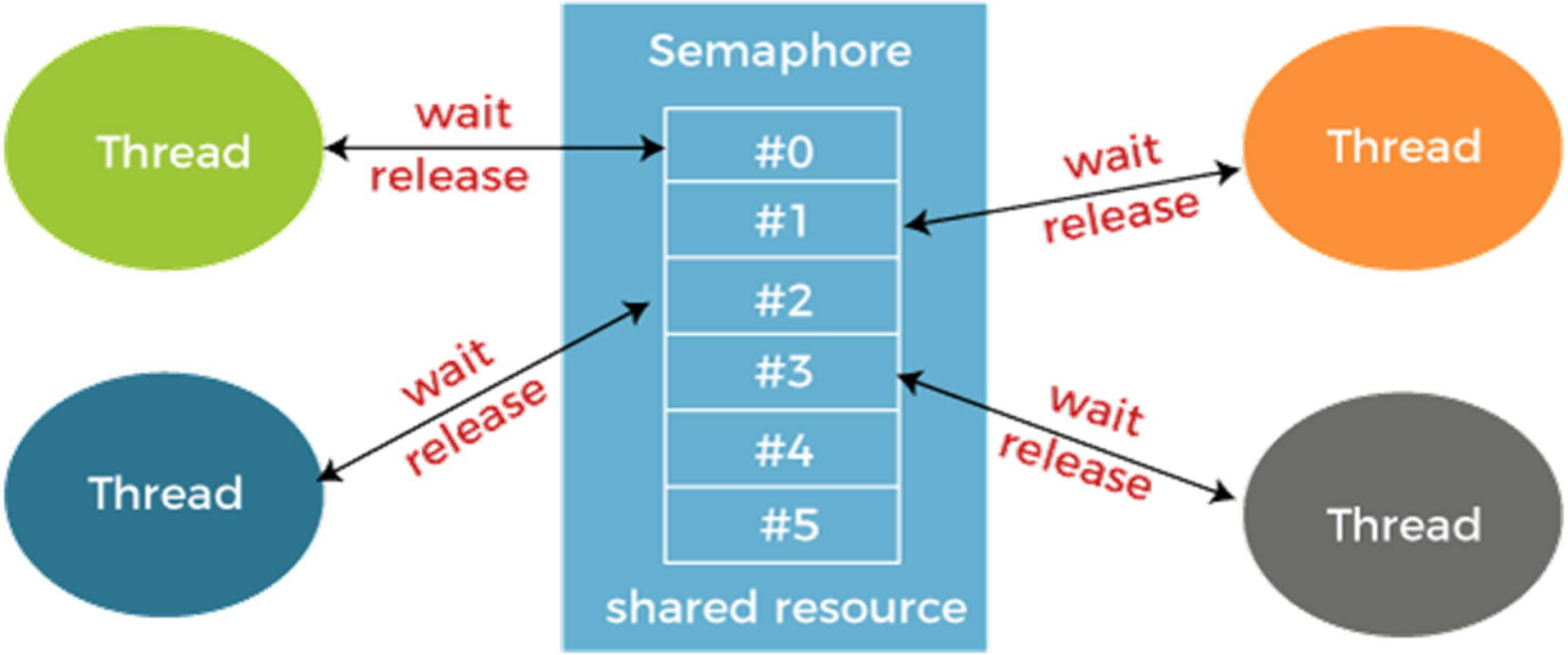
counting_semaphore(C++20)

semaphore that models a non-negative resource count

binary_semaphore(C++20)

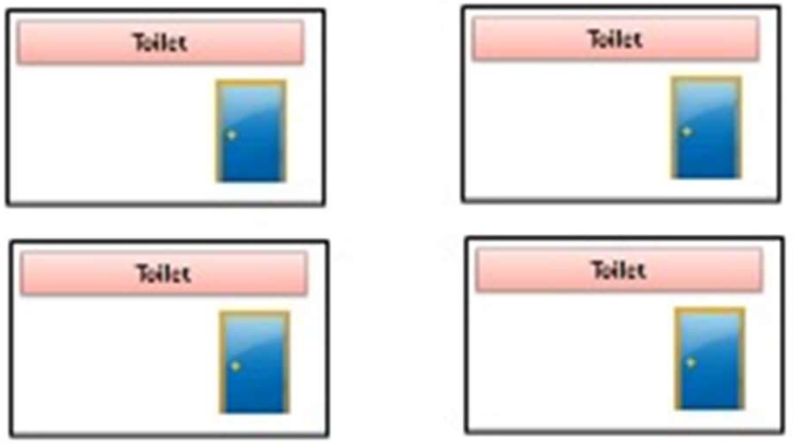
semaphore that has only two states

Counting Semaphore



MUTEX

SEMAPHORE



four identical keys

Counting & Binary Semaphores

- There are two types of semaphores:
  - Counting Semaphore
  - Binary Semaphore.
- Counting Semaphore can be used for mutual exclusion and conditional synchronization.
- Binary Semaphore is specially designed for mutual exclusion.

Latche & Barriers

are thread coordination mechanisms. block until an expected number of threads arrive

latch(C++20)

single-use thread barrier (class)

barrier(C++20)

reusable thread barrier (class template)

Atomic Operation (Indivisible)

Atomic Objects are free of Data Raced.

atomic(C++11)

atomic class template and specializations for bool, integral, floating-point,(since C++20) and pointer types (class template)

atomic_ref(C++20)

provides atomic operations on non-atomic objects

(class template)

Operations on atomic types

`atomic_is_lock_free`	(C++11)
`atomic_store`	(C++11)
`atomic_store_explicit`	(C++11)
`atomic_load`	(C++11)
`atomic_load_explicit`	(C++11)
`atomic_exchange`	(C++11)
`atomic_exchange_explicit`	(C++11)

Operations on atomic types

atomic_compare_exchange_weak	(C++11)
atomic_compare_exchange_weak_explicit	(C++11)
atomic_compare_exchange_strong	(C++11)
atomic_compare_exchange_strong_explicit	(C++11)
atomic_fetch_add	(C++11)
atomic_fetch_add_explicit	(C++11)
atomic_fetch_sub	(C++11)
atomic_fetch_sub_explicit	(C++11)

Flag Type and Safe Operation

atomic_flag

the lock-free boolean atomic type

(class)

atomic_flag_test_and_set

atomically sets the flag to true and returns its previous value

(function)

Flag Type and Safe Operation

atomic_flag_clear

atomically returns the value of the flag

.....

atomic_flag_test

atomic_flag_wait

atomic_flag_notify_one

atomic_flag_notify_all

Thread Cancellation (Coroutines)

- Coroutines VS Subroutines
- Co-await
- Yield (std::this_thread::yield) reschedule the execution of threads, allowing other threads to run.
- co_return
- co-yield

So far our concerns were,

- Invariants were upheld at all times
- Avoiding race conditions inherit from interface
- Handling exception scenarios
- Avoiding deadlocks at all cost.