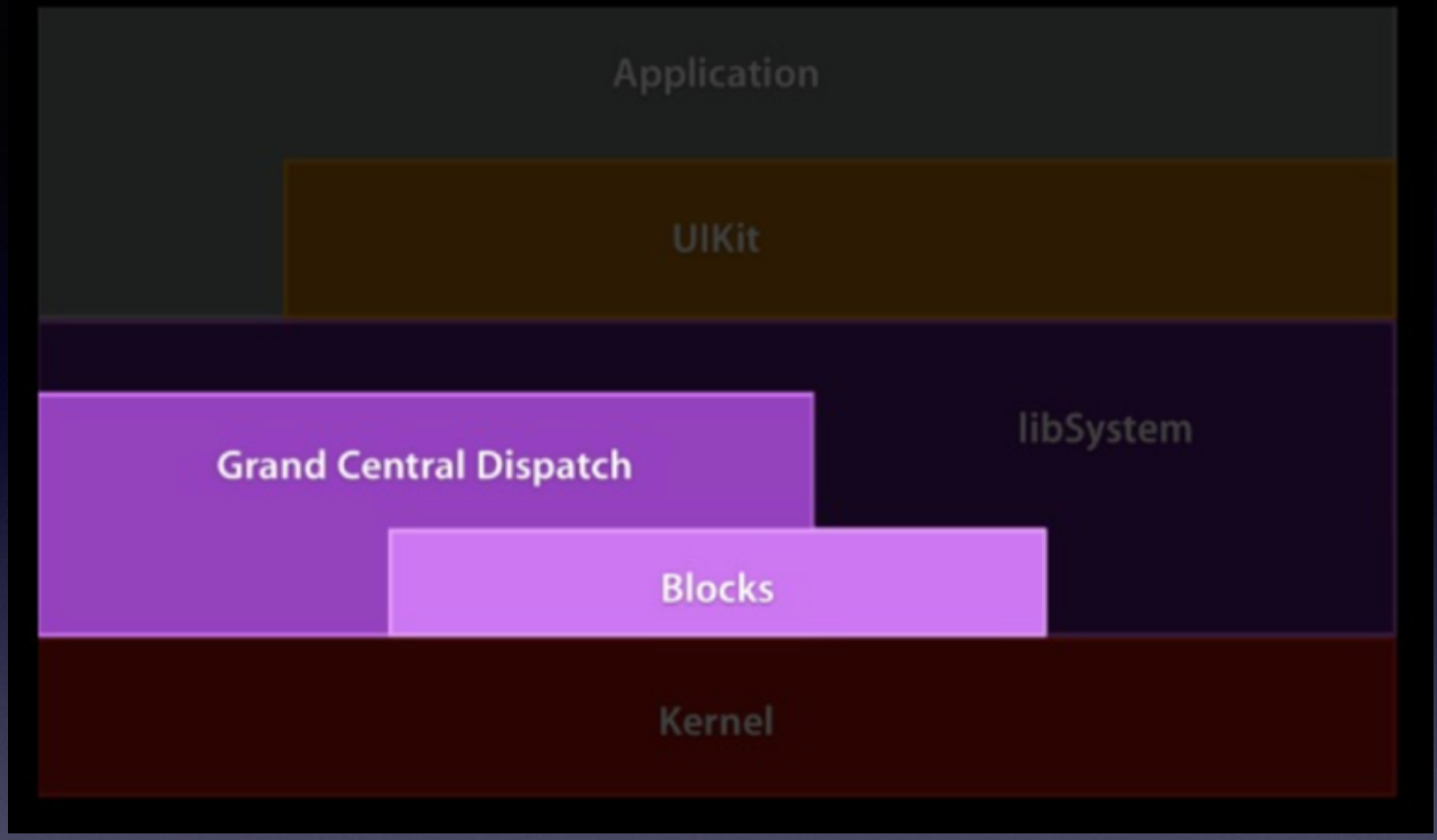
Programmation concurrente

Programmation concurrente

- La **programmation concurrente** est un paradigme de programmation tenant compte, dans un programme, de l'existence de plusieurs piles sémantiques qui peuvent être appelées threads, processus ou tâches. Elles sont matérialisées en machine par une pile d'exécution et un ensemble de données privées.

- lien wiki : https://fr.wikipedia.org/wiki/Programmation_concurrente

Grand central dispatch (GCD) en SWIFT



Framework

API bas niveau pour iOS et OSX

Présentation Générale

- Tous les framework haut niveau utilisent nativement GCD (ex: NSOperation, etc…)
- Diviser le travail d'un processus en plusieurs taches individuelles
- Gérer l'exécution en parallèle ou en séquentielle (Queues)
- Abstraction des Threads, processeurs, coeurs, etc…

Les Queues

- FIFO

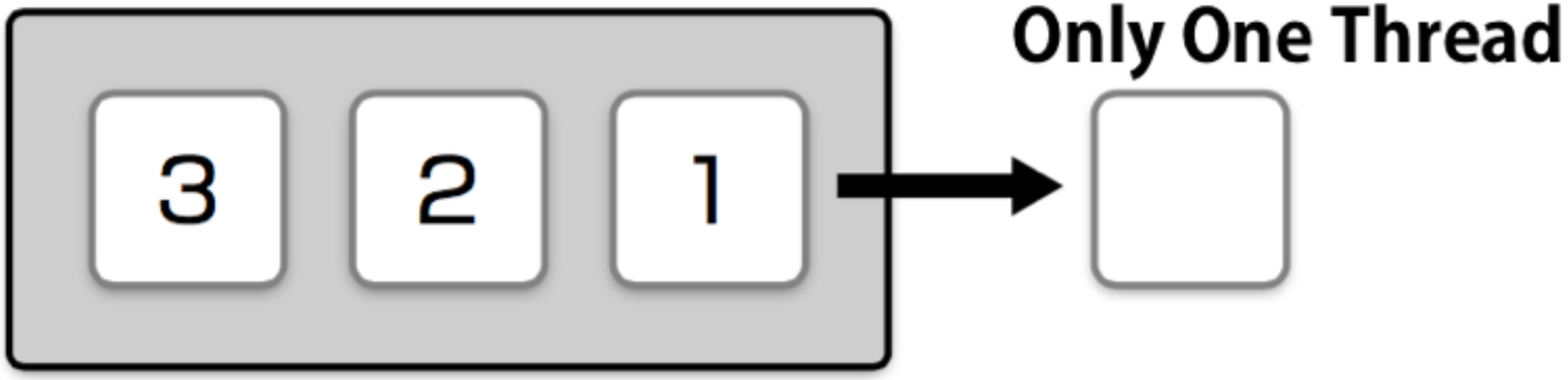
- 2 types de queue :

    1. Concurrent Queue (concurrente)

    2. Serial Queue (séquentielle)

- `Main Queue :` équivalent au Thread principal qui gère principalement l'UI (séquentielle)
- `Global Queue :` concurrente
- `Custom queue :` séquentielle ou concurrente

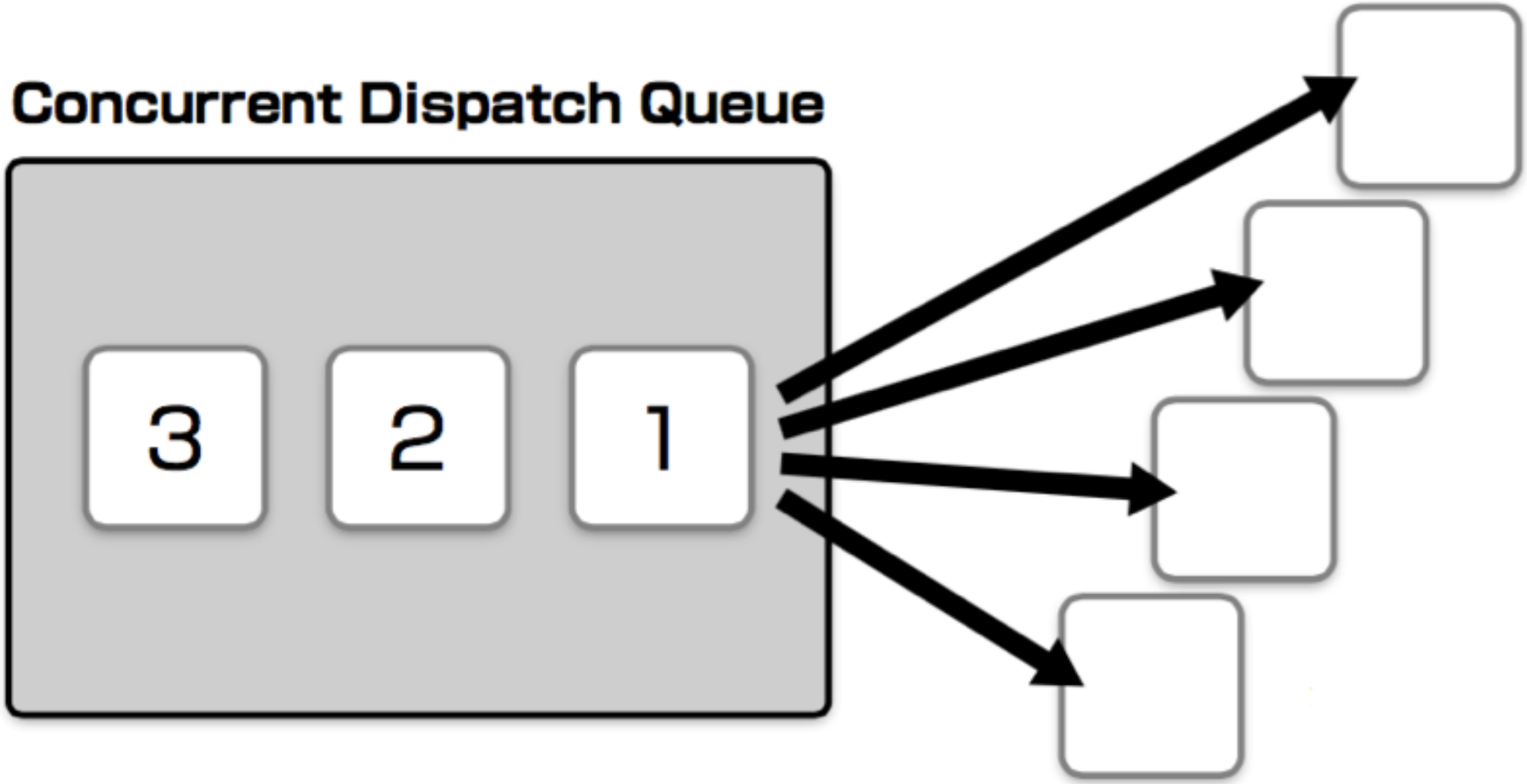
Serial Dispatch Queue



Only One Thread

Multiple Threads

Concurrent Dispatch Queue



En pratique

Récupération / Création

```
func creation() {

        // 1 Step : Creation

        // Main queue
        let mainQueue = dispatch_get_main_queue()

        // Global queue: HIGH, DEFAULT ou LOW
        let globalDefaultQueue = dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0)

        // Custom queue: SERIAL ou CONCURRENT
        let customSerialQueue = dispatch_queue_create("com.loyaltytechnology.queue",
DISPATCH_QUEUE_SERIAL)


    }
```

Utilisation et dispatcher

```
let queue = dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_HIGH, 0)
let group = dispatch_group_create()

for _ in 0 ..< 10 {

      dispatch_group_enter(group)

      dispatch_async(queue) {
         // Long job process
         dispatch_group_leave(group)
      }
}

dispatch_group_notify(group, queue) {
       print("All jobs done!")
}
```

Cas d’usage : 1

- Plusieurs tâches à executer
- Lancer une action une fois que l’ensemble des taches ont été effectuées

dispatch_group_t

Cas d’usage : 1

```
let queue = dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_HIGH, 0)
let group = dispatch_group_create()

for _ in 0 ..< 10 {
        dispatch_group_enter(group)

    dispatch_async(queue) {
            // Long job process

            dispatch_group_leave(group)
        }
}

dispatch_group_notify(group, queue) {
    print("All jobs done!")
}
```

Cas d’usage : 2

- exécuter du code après un certain temps

`dispatch_after`

```
let twoSeconds = dispatch_time(DISPATCH_TIME_NOW, Int64(2 * NSEC_PER_SEC))
let queue      = dispatch_get_main_queue()

print("before")
dispatch_after(twoSeconds, queue) {
    print("2 seconds after")
}
print("after")
```

Et pour finir !

```
 * @param flags
 * Reserved for future use. Passing any value other than zero may result in
 * a NULL return value.
 *
 * @result
 * Returns the requested global queue or NULL if the requested global queue
 * does not exist.
 */
@available(iOS 4.0, *)
@warn_unused_result
public func dispatch_get_global_queue(identifier: Int, _ flags: UInt) -> dispatch_queue_t!
```

Conclusion

- Framework très puissant
- Réelle abstraction des contraintes bas niveaux

Questions ?