CS 329E – Elements of Mobile Computing

Dr. William C. Bulko

Concepts of Multithreading

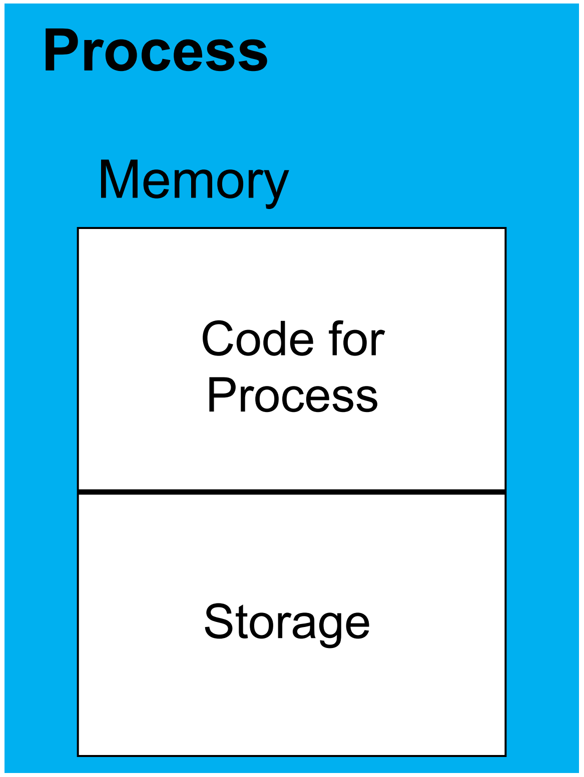
Processes

A *process* is the context in which your program executes.

It includes things like:

- The main memory allocated to your program for data (i.e., variables) and code
- An instruction pointer, which indicates the point in your code you are executing at any given time
- Other resources such as a handle to a database, a network connection, etc.

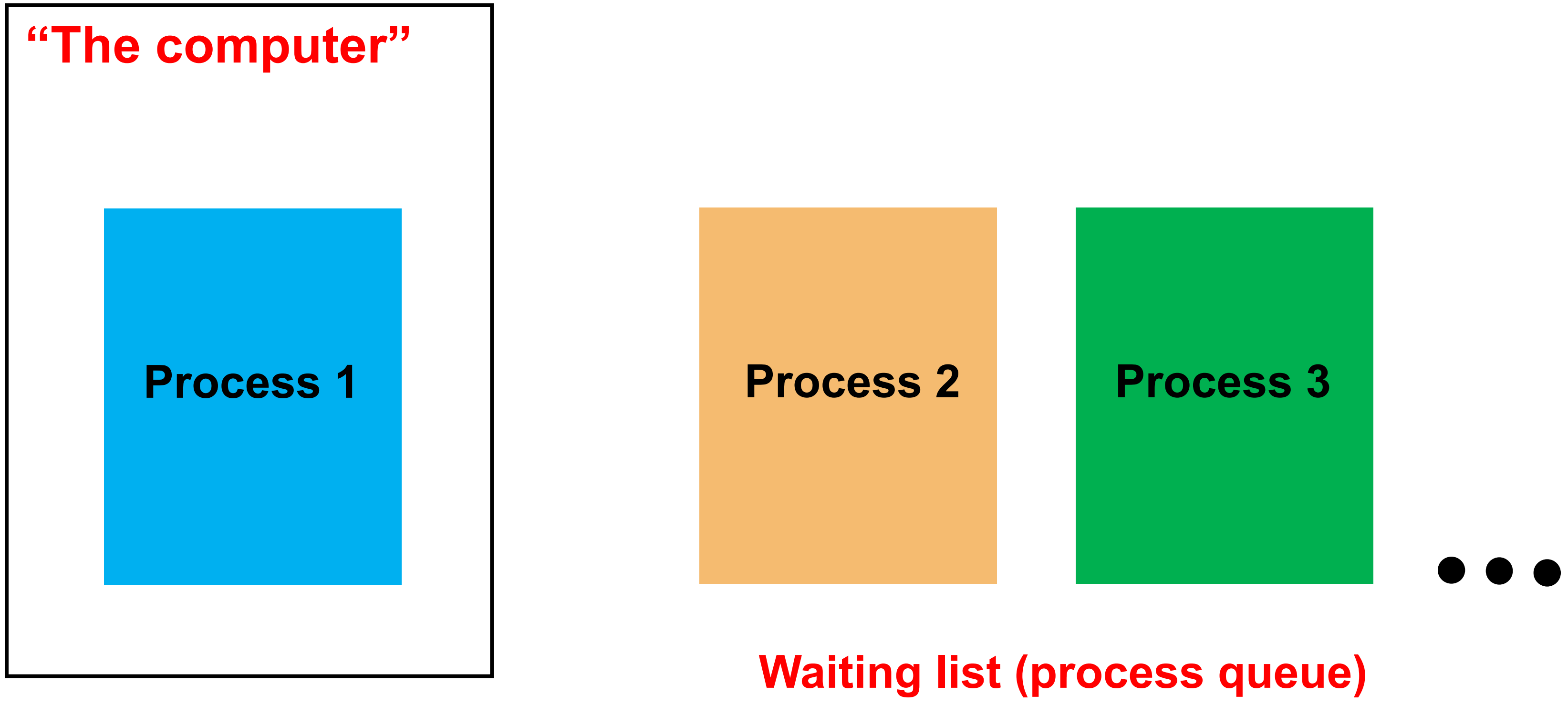
Each program executes in its own process.



Processes (cont.)

Originally, computers could only run one program at a time.

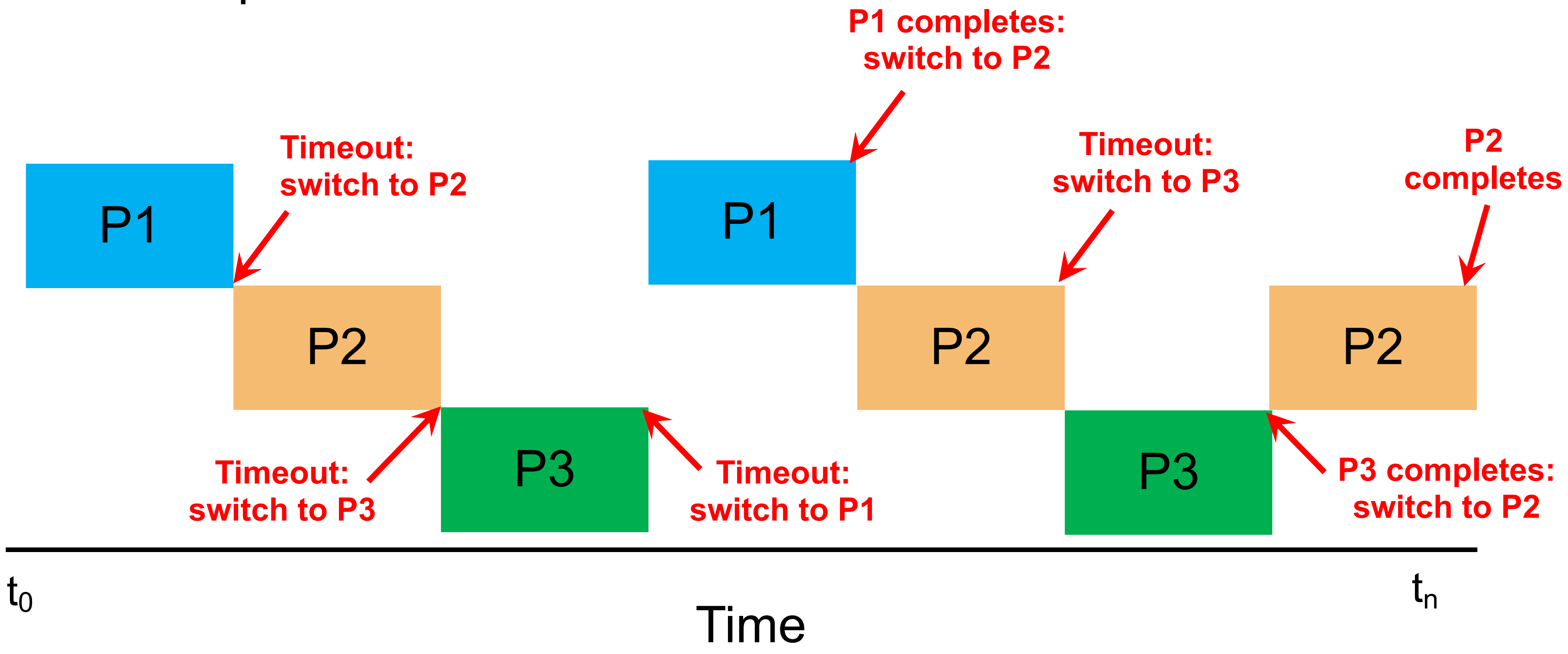
That program would have control of all resources (CPU, memory, I/O, etc.) until it completed or was terminated.



Processes (cont.)

A more efficient way is to allow one process to run only for a specified amount of time.

If it doesn't complete execution when its time is up, a snapshot is taken of the process and what it was doing, and is stored while another process takes a turn.



Processes (cont.)

This is called *context switching.*

- It gives the appearance that more than one thing is happening on your computer at one time because the operating system is very efficient at switching between all of the processes running.
- It can be more efficient, because if an I/O-bound process is really not using the CPU, the scheduler can allow another process to use it.
- It is still not perfect, because the CPU may still be sitting idle most of the time.
- Switching contexts isn't free – it does take time and resources.

Processes (cont.)

Pretty much all CPUs these days have multiple cores (4+ processors), which enables your computer to execute multiple processes simultaneously without context switching. This is called *multiprocessing*.

In addition to running multiple programs, there are some problems that lend themselves really well to be broken down into subproblems.

Processes (cont.)

In a multiprocessing system, the main process can spawn off subprocesses to be run on the other processors. Each subproblem is assigned to a different processor.

- Each subproblem can be solved independently and in parallel to the other subproblems
- The results of the subproblems can be combined to produce the solution to the original problem
- Example: calculating graphics transformations, where the same calculation is applied to all points in the coordinate space.
- Another example is sorting a very large list using Merge Sort: split the list into 4 sublists, sort each sublist in a separate processor, and then merge the results.

Threads

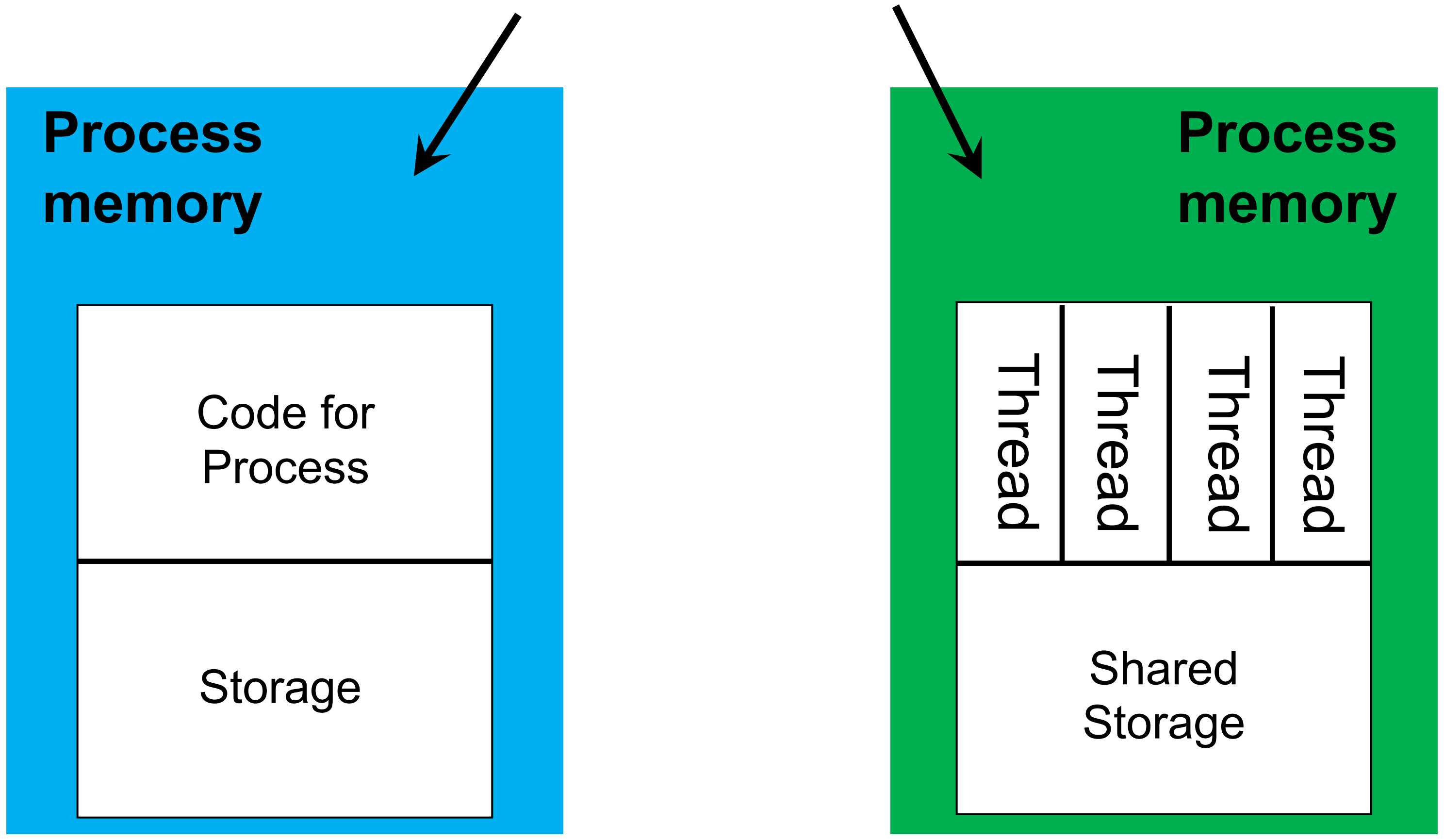
A thread is like a subprocess.

- A process may spawn multiple threads.
- Every process has at least one thread (the *main* thread) but it can have many more than one.
- The big difference between multiprocessing and *multithreading* is that a thread shares data space with other threads associated with the same process.
- For iOS applications, the most important thread is the main thread, which is where all of the UI stuff is done.

Processes vs. Threads

Other process-level stuff

- Pointers to open files
- Child processes
- Signal handlers
- Accounting information



Multithreading

Why do we care about multithreading?

- It helps us make better use of system resources – in particular, the CPU
- It creates more responsive interactive applications
  - Downloading a file from the network while we're updating the user interface
  - Having one thread doing heavy-duty calculations while another thread is interacting with the user

Multithreading (cont.)

Threads have the same context-switching concept as processes do.

- However, with multi-core CPUs, it doesn’t just “feel” like we’re doing multiple things at once: we can really run more than one process and/or thread at the same time
- We have processes being switched in and out, and threads within processes being switched in and out – all managed by the operating system.

Multithreading

The down side:

The programmer must carefully design the program in such a way that all of the threads can run at the same time without interfering with each other: you may need to *synchronize* access to data.

The bright side:

It’s a lot easier to do than it used to be.

Tasks

From our perspective, a task is essentially a *closure*. Recall that:

- Closures are self-contained, callable blocks of code that can be stored and passed around.
- When called, they behave like functions, and can have parameters and return values.
- In addition to having its own variables with local scope, a closure can reference variables it uses from outside its own scope: that is, it sees the variables from the enclosing scope and remembers their value

In Objective-C, unnamed chunks of code are called “blocks”.

Serial vs. Concurrent Tasks

Serial and *Concurrent* describe when tasks are executed with respect to each other.

- Tasks executed *serially* are always executed one at a time
- Tasks executed *concurrently* might be executed at the same time. The system determines if and when they are actually executed concurrently.

Concurrency vs. Parallelism

Concurrency means that an application is making progress on more than one task at the same time (concurrently).

- If the computer only has one CPU, the application may not make progress on more than one task at *exactly the same time*.

Parallelism means that an application splits its tasks up into smaller subtasks which can be processed in parallel, for instance on multiple CPUs at the exact same time.

- Parallelism requires concurrency, but concurrency does not guarantee parallelism
- You code for concurrency, with the expectation that stuff happens in parallel

Synchronous vs. Asynchronous

These terms describe when a function call will return control to the caller, and how much work will have been done by that point.

- A *synchronous* function call returns only after the task it initiates has completed, similar to a regular function call.
- An *asynchronous* function call returns immediately: it starts a task executing, but it returns without waiting for the task to complete.

Thus, an asynchronous function call does not block the current thread of execution from proceeding.

Critical Section

A *critical section* is a piece of code that must NOT be executed concurrently (i.e., from two threads at once).

This is usually because the code manipulates a shared resource, such as a variable, that can become corrupt if it's accessed by concurrent processes.

```
func func1() {
   print("line 1")
   print("line 2")
   print("line 3")
}

func func2() {
   print("line one")
   print("line two")
   print("line three")
}

func1()
func2()
```

Might print:

```
line 1
line one
line 2
line two
line three
line 3
```

Race Condition

A *race condition* is a situation where the behavior of a software system depends on a specific sequence or timing of events that execute in an uncontrolled manner, such as the exact order of execution of the program’s concurrent tasks.

- Race conditions can produce unpredictable behavior that isn’t immediately evident through code inspection.

Deadlock

Two or more threads are said to be *deadlocked* if they all get stuck waiting for each other to complete or perform another action.

- The first can’t finish because it’s waiting for the second to finish.
- The second can’t finish because it’s waiting for the first to finish.

Thread Safe

Thread safe code is code that can be safely called from multiple threads or concurrent tasks without causing any problems (data corruption, crashing, etc.)

- Code that is NOT thread safe must only be run in one context at a time.

Example: allowing access in multiple threads to a mutable object (such as `var a = [""]`) would be unsafe, because without care, multiple threads could change it at any time.

CS 329E – Elements of Mobile Computing

Dr. William C. Bulko

Multithreading in iOS

GCD

Grand Central Dispatch (GCD) is Apple’s library for concurrent code execution on multi-core hardware.

- It is embedded in the OS and, as a result, is very efficient.

```
import Foundation
```

- You add tasks (blocks of code) to queues, and GCD manages a *thread pool* behind the scenes. Consequently, as a developer, you think about tasks in a queue rather than threads.
- GCD decides on which thread in a given queue your code executes on
- GCD manages the threads according to available system resource: it selects the optimal number of threads.

GCD Queues

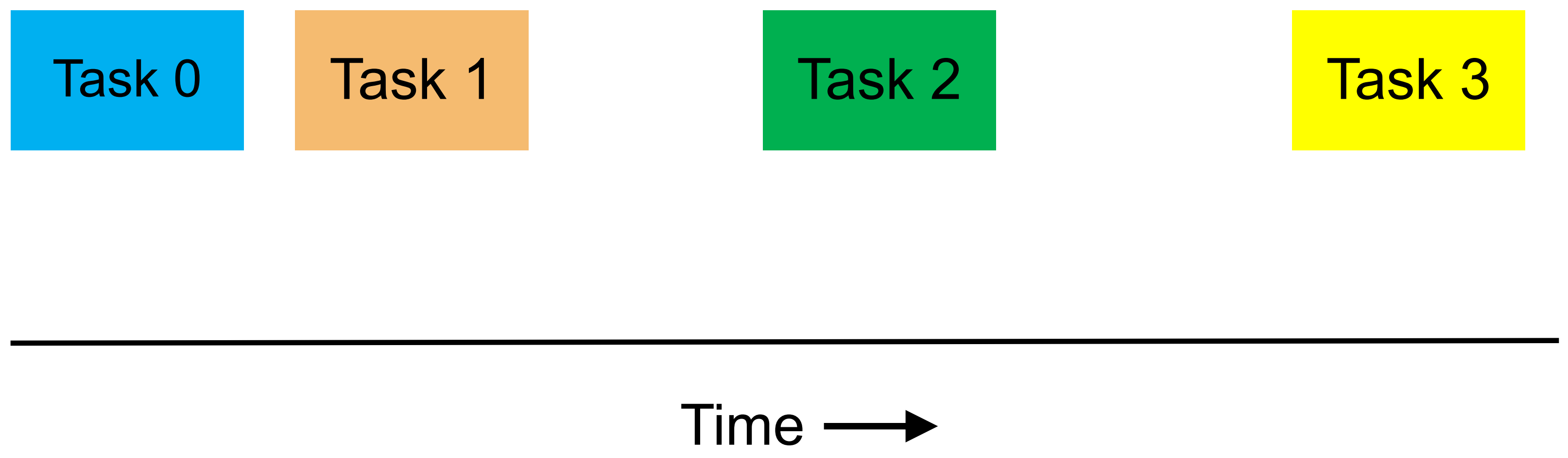
GCD provides *dispatch queues* to handle submitted tasks.

- Foundation defines `class DispatchQueue` for you.
- These queues manage the tasks you provide to GCD, and execute those tasks in FIFO order.
- All dispatch queues are themselves thread-safe in that you can access them from multiple threads simultaneously without issue
- The key to using GCD well is to choose the right kind of dispatch queue and the right dispatching function to submit your work to the queue.

Serial Queues

Tasks in *serial queues* execute one at a time, each task starting only after the previous task has finished.

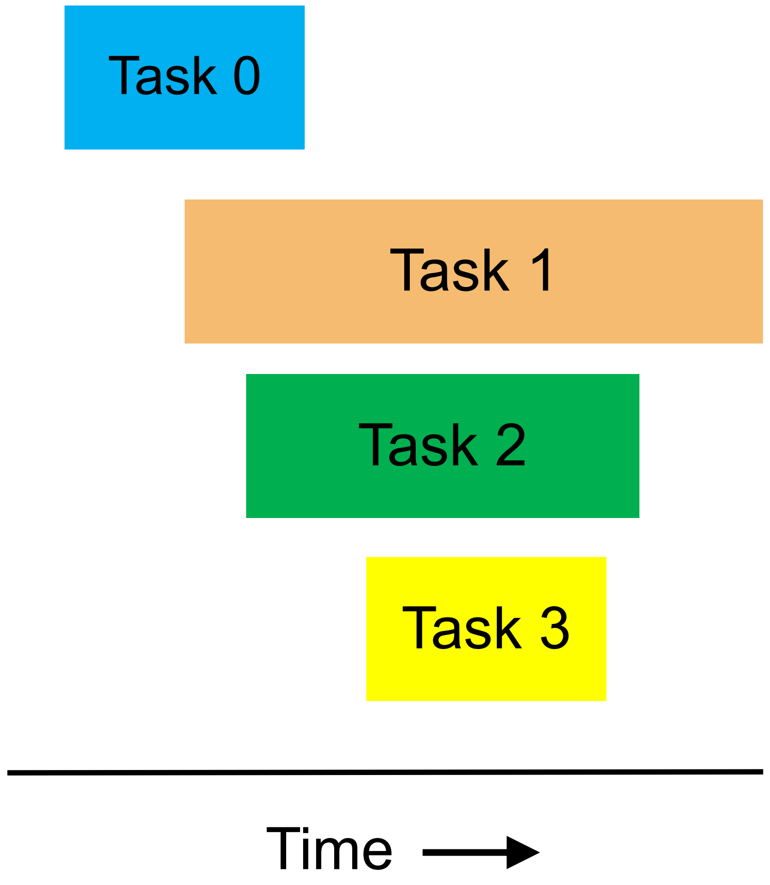
- You won't know the amount of time between one task ending and the next one beginning.
- Excellent for managing a shared resource.



Concurrent Queues

Tasks in *concurrent queues* are guaranteed to start in the order they were added, but that’s all.

- Items can finish in any order and you have no knowledge of the time it will take for the next task to start, nor the number of tasks that are running at any given time: that’s up to GCD.



Main Dispatch Queue

GCD provides you with several major queues to choose from.

`main` is the globally available serial queue that executes tasks on the application's main thread.

- When you don't use any multiprocessing, everything is run serially on the `main` queue.
- It's used to update the app UI and perform all tasks related to the update of UIViews.
- Since it's serial, you can only execute one task at a time. Consequently, the UI can be blocked when you run a heavy task in the `main` queue.

Global Dispatch Queues

GCD also provides four concurrent queues called “global queues”.

- You can divide your app’s work across parallel processes on the global queues. . .but be careful. This can backfire and really slow things down.
- In general, what you probably *really* want is to use asynchronous tasks instead of parallel processes. Understand the difference and use the global queues judiciously.

Apple’s APIs also use the global dispatch queues, so any tasks you add won’t be the only ones on these queues.

Quality of Service (QoS)

Although `main` has the highest priority, we can also specify how to prioritize the queues we create. This specification is referred to as **Quality of Service** (QoS).

QoS is an enum. We can assign the values below to our queues listed in order from highest priority to lowest:

```
.userInteractive     ← highest priority
.userInitiated
.default
.utility
.background
.unspecified         ← lowest priority
```

Quality of Service (QoS)

`userInteractive`

- For tasks that need to be done immediately in order to provide a good user experience.
- Use it for UI updates, event handling, and small workloads that require low latency.
- The total amount of work done in this class during the execution of your app should be small.

`userInitiated`

- For tasks that are initiated from the UI and can be performed asynchronously.
- Use it when the user is waiting for immediate results, and for tasks required in order to continue user interaction.

GCD Global Dispatch Queues

`default` self-explanatory

`utility`

- For long-running tasks, typically with a user-visible progress indicator.
- Use it for computations, I/O, networking, continuous data feeds, and similar tasks.

`background`

- For tasks that the user is not directly aware of.
- Use it for prefetching, maintenance, and other tasks that don't require user interaction and aren't time sensitive.

`Unspecified` self-explanatory

GCD Queues

You can also create your own custom serial or concurrent queues.

- This means you have *at least* five queues at your disposal: the main queue, four global dispatch queues, plus any custom queues that you add.

Methods to Create Queues

Create a global queue:

```
let queue = DispatchQueue.global()
```

Create a custom queue named `<name>`:

```
let queue = DispatchQueue(label: <name>)
let queue = DispatchQueue(label: <name>,
        qos: .userInitiated)
```

Use the main queue:

Already created. Just reference `DispatchQueue.main` as your queue.

Methods to Use to Initiate Tasks

```
queueName.sync {
  <task>
  }
```

Queues the task and returns only after the block of code is done executing

```
queueName.async {
  <task>
  }
```

Queues the task and returns immediately