# CPS 310
# Taming Concurrency: Design Patterns

Jeff Chase

Duke University

## How to use threads?

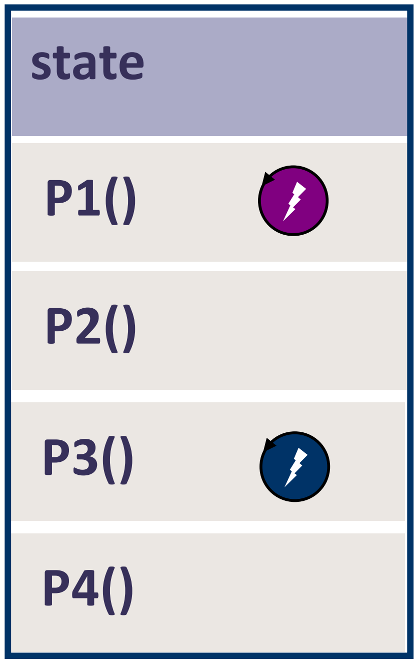
- Preview some common abstract **design patterns.**
- **Abstract** → they strip out lots of details about the system context in which they (re)appear.
  - Unix programming, servers, GUI apps, streaming data processing, concurrent PLs, scalable everything.
- These patterns illustrate ways to:
  - Spread the processing work to parallelize and scale up.
  - Structure any needed blocking to maintain responsiveness.
  - Limit shared data to reduce synchronization.
  - Decompose complex programs —or— compose simple components into complex programs.

## Software architecture

What are the “right” building blocks for large systems?
How to structure concurrency in large systems?

# Modular atomic objects

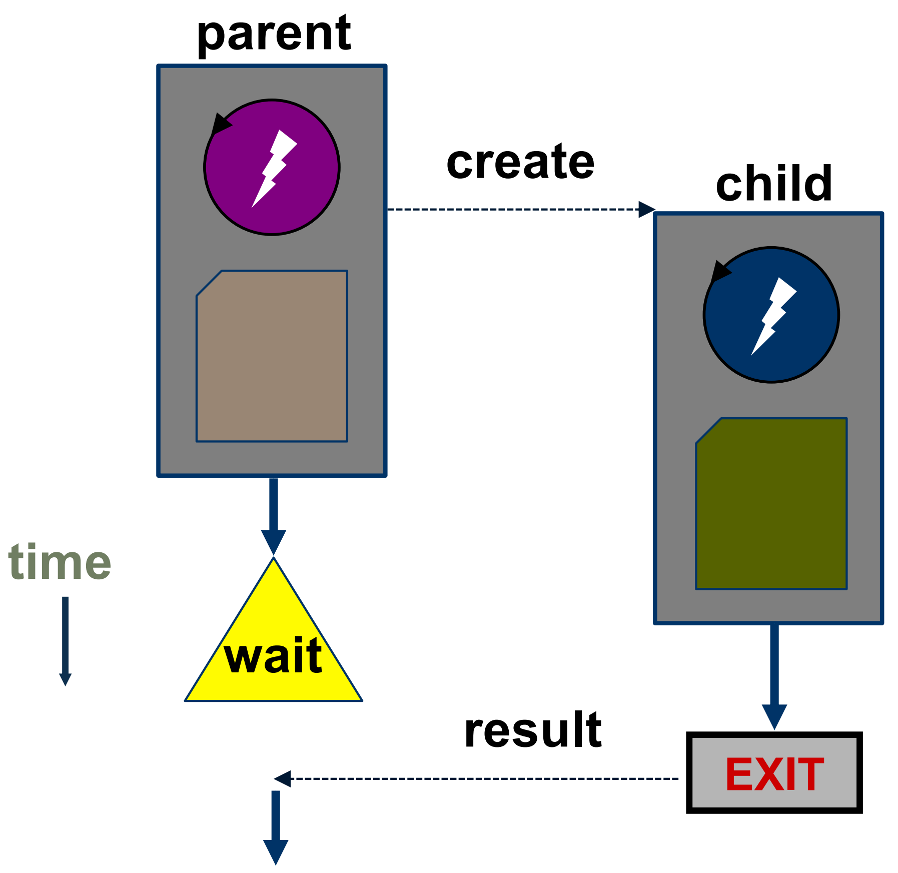
- A set of procedures/methods and API that defines how threads call/invoke them.
- **Private state**: data accessed (only) by the methods, **shared** by threads inside.
- E.g., objects instantiated from classes
- Threads invoke API→ concurrency inside, but we don't care where threads came from.
- **Atomic**→hides internal concurrency: behavior equivalent to some serial order.
- No locks across API→ "freely composable".



E.g., classical **monitors**

**Abstract Data Type (ADT):** the module is encapsulated: its **state** is manipulated only through its **API** (Application Programming Interface).

# Parent-child (fork/join)



**Unix**: **P** launches **C** as an isolated process, which may run a different program and return an exit status.

- Parent thread **creates** Child.
- **P** passes input/arguments.
- **C** runs, outputs **result**.

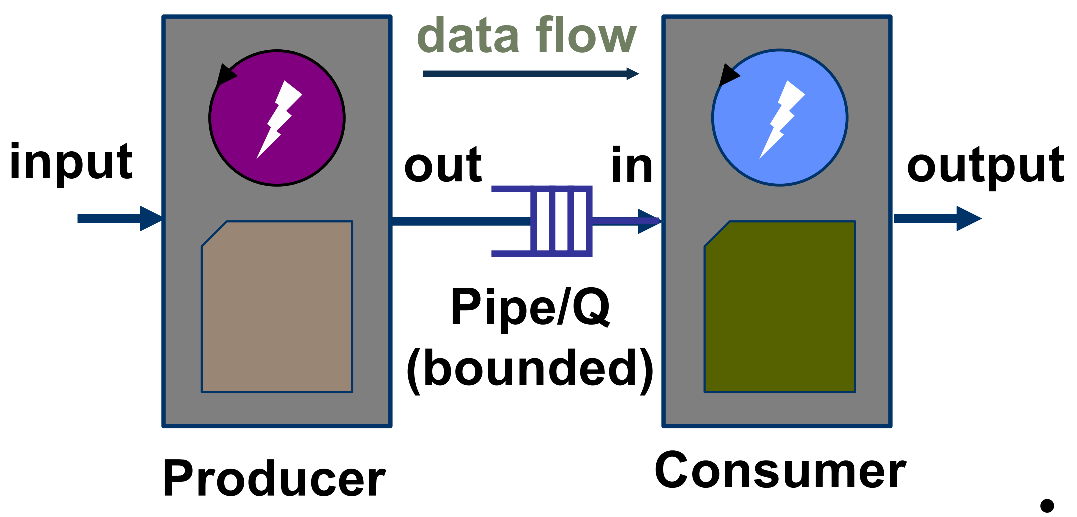
Optional: **P joins** on **C**.

- P's join "reaps" result from C.
- Join blocks until **C** terminates.

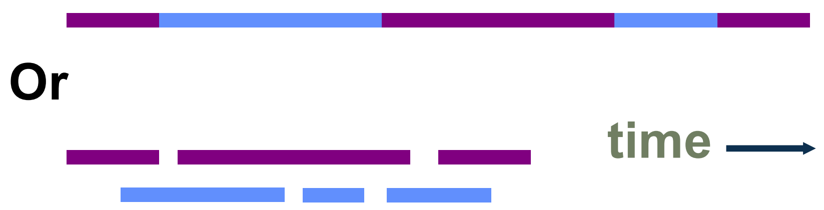
Examples:

- Pthreads in shared memory
- Unix shell: run command in **C**
- Unix process tree: fork/wait

# Pipeline



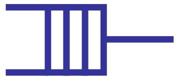
Thread design pattern: **pipeline.**
To process a stream of data.

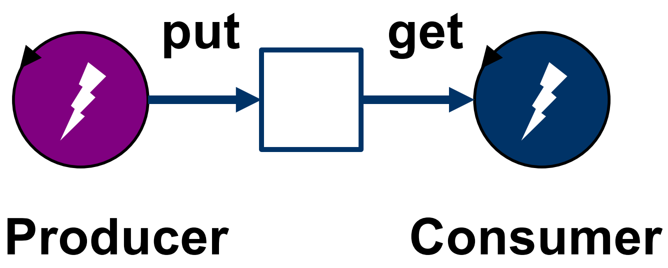


- Sequence of processing **stages**
- Run stage when input is ready **and** space for output.
- Block if no input **or** no space.

- Run stages in parallel (on multi-core): faster completion.
- Unix uses pipes to compose (string together) independent programs.
- Auto-match speeds by wait/notify on full/empty conditions (soda).

# Mailbox

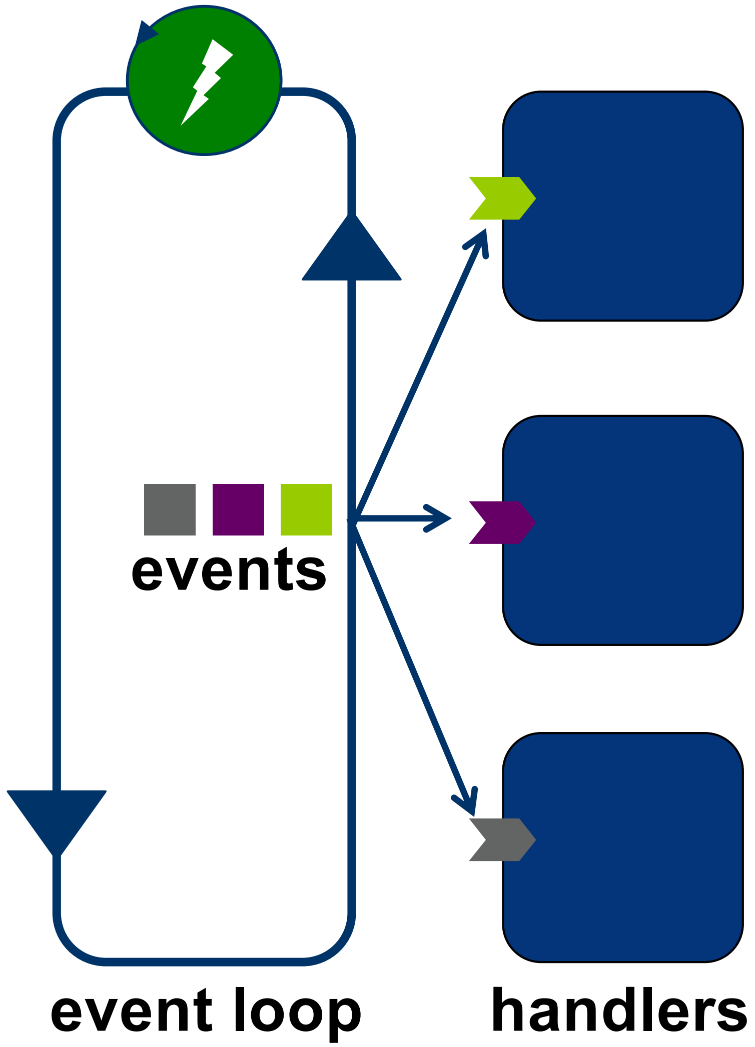
- Suppose only one slot to pass data.
- Then the producer and consumer must alternate strictly: **ping pong**.
    - Ping pong is the exception to prove the “loop before you leap” rule.
- If bounded buffer has multiple slots, then it becomes **soda machine**.
- Bounded queue: 
- Example: **pipe** between pipeline stages.
- These schemes provide **flow control** to speed-match producer and consumer.



```
lock(m);
while(true) {
      put/get() etc.;
      signal(m,c);
      wait(m,c);
}
unlock(m);
```

# Event-driven programming

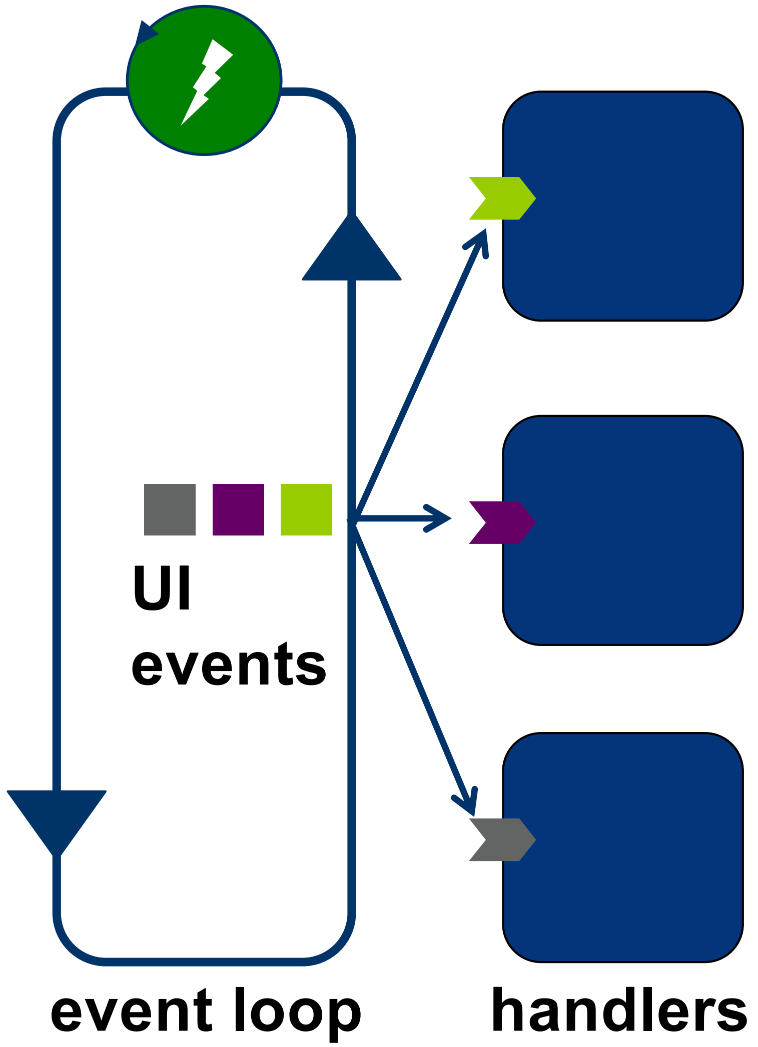
- Event-driven programming is a design pattern for a thread's program.
- The thread receives and handles a sequence of typed messages or events.
- The program is a set of handler routines, one for each event type.
  - The thread handles each event in sequence by calling the event's handler.
  - **Reactive**: thread "reacts" to events.
- In its pure form handlers do not block.
  - Thread blocks **only** if no event (idle).
  - → **responsive**, no way to "get stuck".



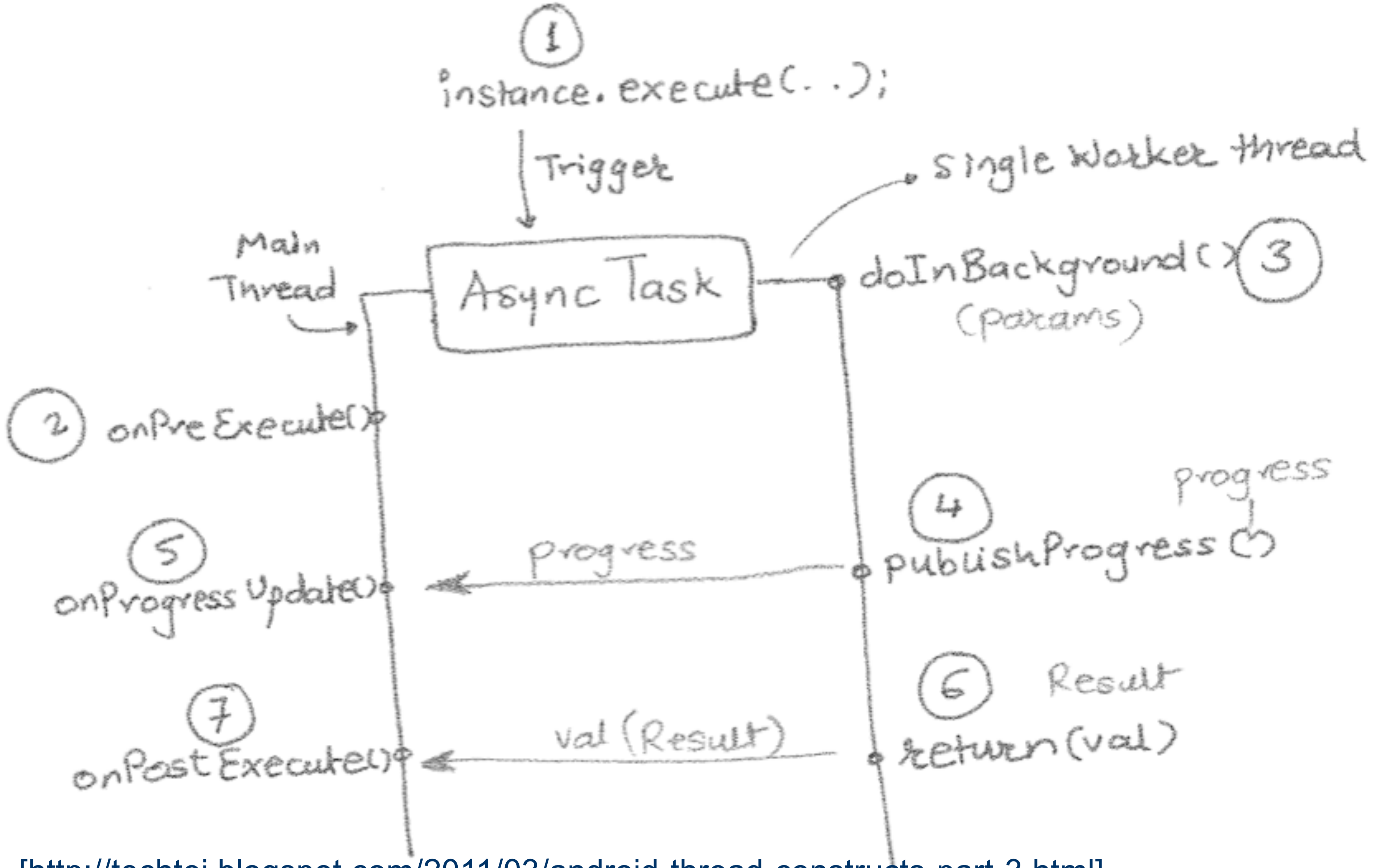
# Event-driven example: UI thread

In GUI apps, e.g.: Android, Windows:

- App's main thread is event-driven, e.g., by User Interface (UI) events.
- Also called the "UI thread".
- UI toolkit code defines a handler for each kind of UI event (e.g., a click).
- UI handlers called only by UI thread: no need for locks.  (Not thread-safe!)
- UI handlers should not block, or app becomes unresponsive.
- What if some task needs to block?



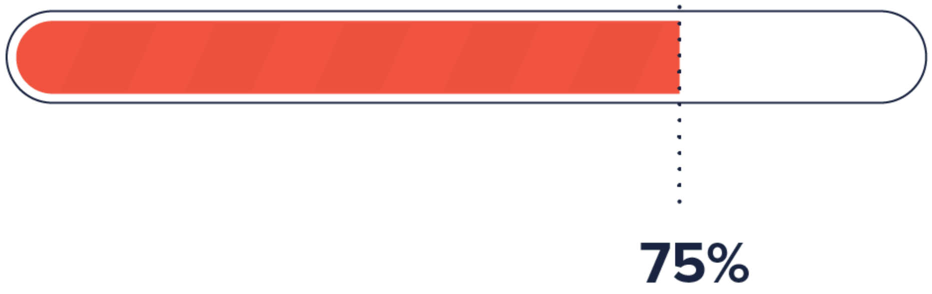
# AsyncTask (Android)



[http://techtej.blogspot.com/2011/03/android-thread-constructs-part-3.html]

## AsyncTask ← Parent/child + event-driven

- UI thread **P** spawns AsyncTask **C** for background ops.
  - **Worker** for e.g., downloads, media, data processing
- **C** may block, but **P** remains responsive UI events.
- **P** never blocks to receive result from **C**.
- Instead, **C** sends status as events on **P**'s event queue.
- **P** handles **C**'s events mixed with other (e.g., UI) events.
- In essence, **P** "calls" **C**'s function but never waits for it.

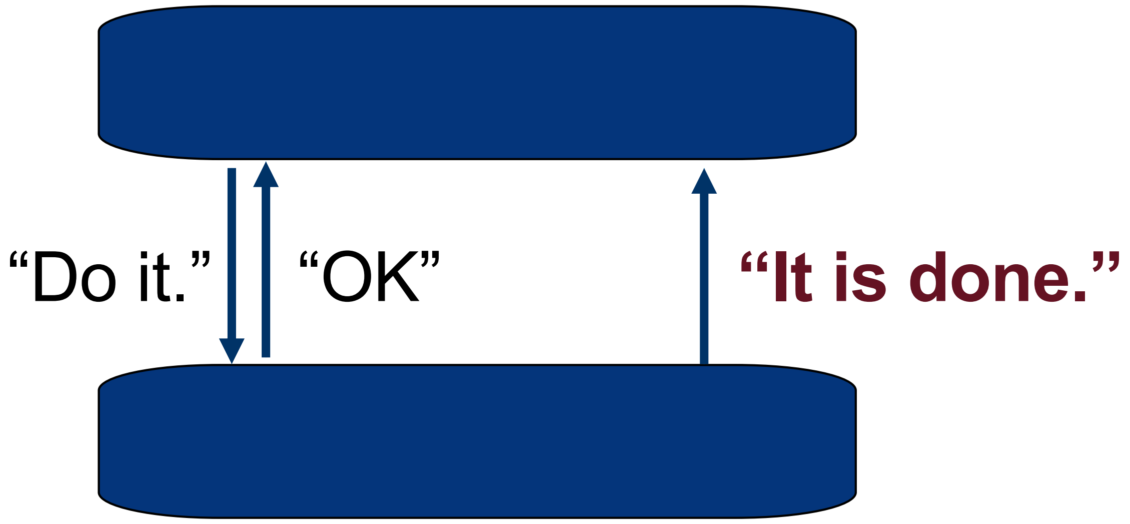


E.g., UI thread updates progress bar on status from **C**.

# Asynchronous API

- AsyncTask illustrates an **asynchronous API**.
  1. Call method.
  2. Method returns immediately.
  3. Caller receives completion event (notification upcall).
- Enables a thread to initiate multiple concurrent activities and oversee them without blocking.
  - Languages: **Future** in Java, Scala, etc.
- Also used in modern OS APIs for **asynchronous I/O**
  - E.g., Windows; BSD/MacOS **kqueue;** Linux **epoll**
  - Similar to low-level I/O: initiate DMA + interrupt

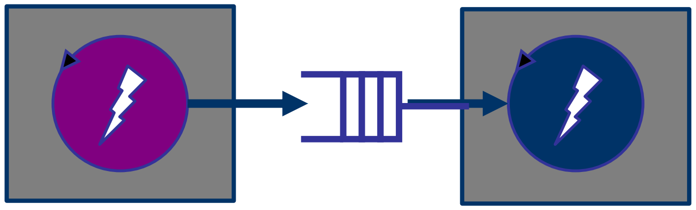
# Upcall



- **Upcall**: cause to invoke a procedure in a client module.
- E.g., by sending an event message to its event queue.
- Notifies client of an asynchronous event or completion.
- "Like an interrupt." *See also*: Unix signal handler.

## Perils of shared state→mitigate

- Shared state→races. Synchronize! Do it right—but can we do less of it?
  - Less is more!
  - More concurrency?
  - Fewer bugs?
  - Less/safer blocking?

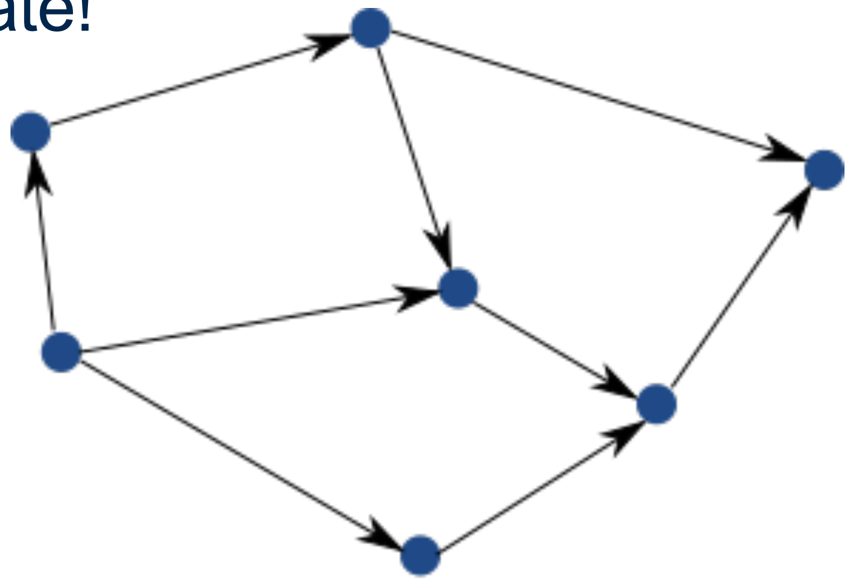


- The thread design patterns create opportunities to factor state among threads and keep state private.
  1. Decouple state into objects/processes.
  2. Place each under control of e.g. a single thread.
  3. Pass events/data via mailboxes, queues, buffers.

# Actor model (“Reactive”)

- An **actor** has **private state** and **one thread** that reacts to messages on its **one queue** (event-driven).
- Single-threaded→no internal synchronization!
- Parallelism—with no shared state!
- Actors can live anywhere!
- All **asynchronous**

  → actors don’t “get stuck”.
- Sends are “fire and forget”.
- No synchronous request-reply!
- **Languages**: Erlang, Scala/Akka.



Actors interacting with each other by sending messages to each other

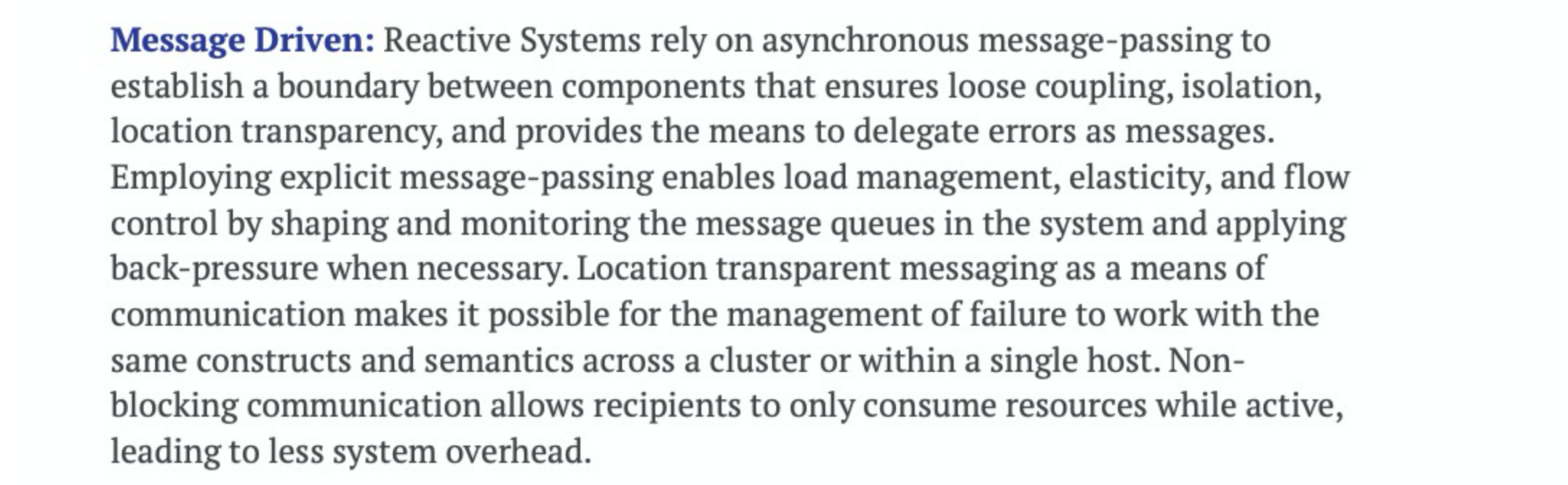
Graphic from: https://doc.akka.io/docs/akka/current/typed/guide/actors-intro.html

# The Reactive Manifesto

We Are Reactive

*Published on September 16 2014. (v2.0)*

We believe that a coherent approach to systems architecture is needed, and we believe that all necessary aspects are already recognised individually: we want systems that are Responsive, Resilient, Elastic and Message Driven. We call these Reactive Systems.

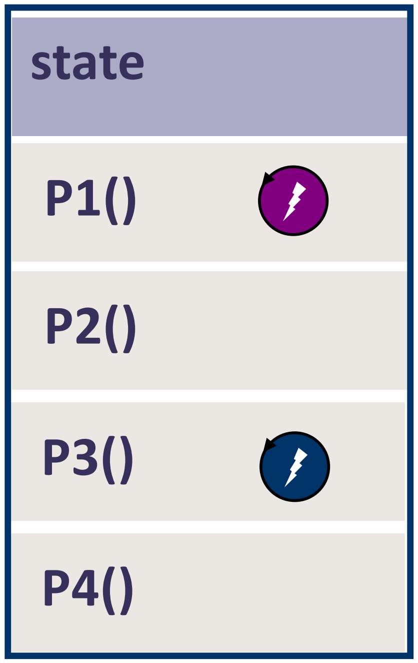


**Message Driven:** Reactive Systems rely on asynchronous message-passing to establish a boundary between components that ensures loose coupling, isolation, location transparency, and provides the means to delegate errors as messages. Employing explicit message-passing enables load management, elasticity, and flow control by shaping and monitoring the message queues in the system and applying back-pressure when necessary. Location transparent messaging as a means of communication makes it possible for the management of failure to work with the same constructs and semantics across a cluster or within a single host. Non-blocking communication allows recipients to only consume resources while active, leading to less system overhead.

https://www.reactivemanifesto.org/

# Be wary of extremist ideologies

- Which brings us back where we started.
- “Reactivisionism” is appealing.
- It is a valuable approach.
- But sometimes you need shared state.
- Reactive models are built on classical threads with shared state (e.g., queues).
- It’s what we teach; it’s what we do.
- There is no One True Way to tame concurrency.
- Or to solve **any** problem!



Threads in Android

## EXTRA

## Threads in Android

Three examples/models for use of threads in Android.

1. Main thread (UI thread): receives UI events and other upcall events on a single incoming message queue. Illustrates event-driven pattern: thread blocks only to wait for next event.
2. ThreadPool: an elastic pool of threads that handle incoming calls from clients: Android supports “binder” request/response calls from one application to another. When a request arrives, a thread from the pool receives it, handles it, responds to it, and returns to the pool to wait for the next request.
3. AsyncTask: the main thread can create an AsyncTask thread to perform some long-running activity without blocking the UI thread. The AsyncTask thread sends progress updates to the main thread on its message queue.

These patterns are common in many other systems as well.

**Adapted from**
**http://developer.android.com/guide/components/processes-and-threads.html**

**Summary**: By default, all components of the same application run in the same process and thread (called the "main" thread). The main thread controls the UI, so it is also called the UI thread. If the UI thread blocks then the application stops responding to the user. You can create additional background threads for operations that block, e.g., I/O, to avoid doing those operations on the UI thread. The background threads can interact with the UI by posting messages/tasks/events to the UI thread.

**Details**: When an application component starts and the application does not have any other components running, the Android system starts a new Linux process for the application with a single thread of execution called the **main thread**.

All components that run in the same process are initialized by its main thread, and system upcalls to those components (onCreate, onBind, onStart,…) run on the main thread.

The main thread is also called the **UI thread**. It is in charge of dispatching events to user interface widgets and interacting with elements of the Android UI toolkit. For instance, when the user touches a button on the screen, your app's UI thread dispatches the touch event to the widget, to set its pressed state and redraw itself.

If you have operations that might require blocking, e.g., to perform I/O like network communication or database access, you should run them in separate threads. A thread that is not the UI thread is called a **background thread** or "worker" thread.

**Adapted from http://developer.android.com/guide/components/processes-and-threads.html**

Your app should never block the UI thread. When the UI thread is blocked, no events can be dispatched, including drawing events. From the user's perspective, the application appears to “hang” or “freeze”.

Even worse, if the app blocks the UI thread for more than a few seconds, Android presents the user with the infamous "**application not responding**" (ANR) dialog. The user might then decide to quit your application and uninstall it.

In a correct Android program the UI thread blocks only to wait for the next event, when it has nothing else to do (it is idle). If you have an operation to perform that might block for any other reason, then you should arrange for a background/worker thread to do it.

Additionally, the Android **UI toolkit is not thread-safe**: if multiple threads call a module that is not thread-safe, then the process might crash. A correct app manipulates the user interface only from a single thread, the UI thread. So: your app must not call UI widgets from a worker thread.

So how can a worker thread interact with the UI, e.g., to post status updates? Android offers several ways for a worker to post operations to run on the UI thread.

**Note**: this concept of a single event-driven main/UI thread appears in other systems too.