G53MDP
Mobile Device Programming

Services

Services

- An Application **Component** that
  - Has no UI
  - Represents a desire to perform a longer-running operation
    - I.e. longer than a single-activity element of the task
    - Threads are associated with the activity that started them
      - i.e. could be orphaned
- Activities are loaded/unloaded as users move around app
  - Services remain for as long as they are needed
- Expose functionality for other apps
  - One service may be used by many applications
  - Avoid duplication of resources

Service Lifecycle

- By nature, services are singleton objects
  - “There can be only one”
- The Service sub-class object is instantiated if necessary
  - onCreate() is called
  - either onStartCommand or onBind will be called depending on how the service has been “called”
- onCreate / onStart / onBind are called in the context of the main UI thread
  - Must spawn a worker thread to do any significant work
- *Something* calls stopService()
  - Could be the OS again
  - How do we ensure we don’t lose work?
- onDestroy

Terminating Services

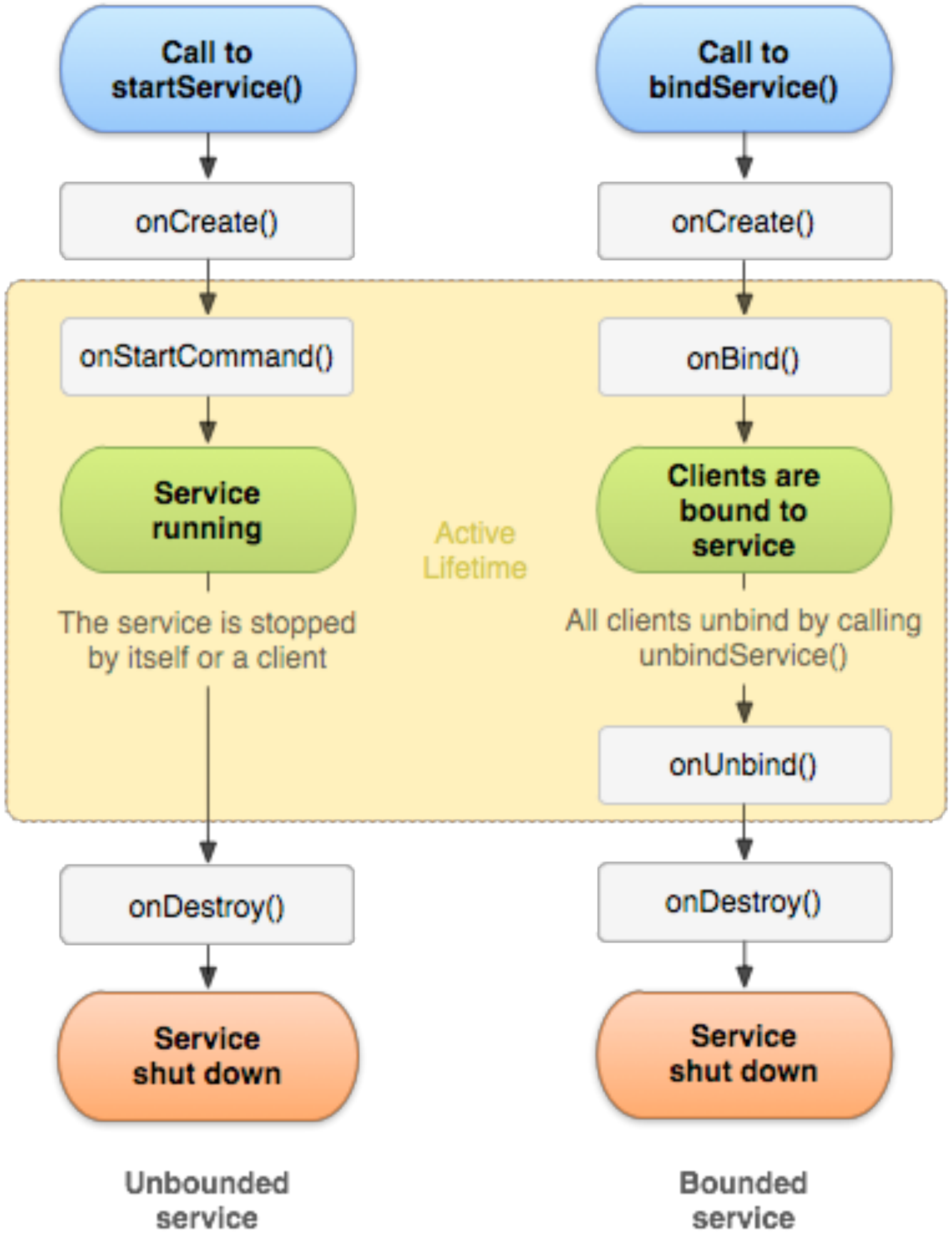
- A Service runs in the background indefinitely
  - Even if the component that started it is destroyed
- Termination of a service
  - Self-termination (calling stopSelf())
  - stopService() via an Intent
  - System termination
    - i.e. memory shortage – Last recently used again
- Avoiding termination
  - Foregrounding a Service
    - This is something the user should really know about
    - Active in the Status Bar / shows a Notification
    - Is treated as important as a foregrounded Activity
    - startForeground(…)
  - Background services are vulnerable
    - Android 8.0
      - Stopped by the system - why?

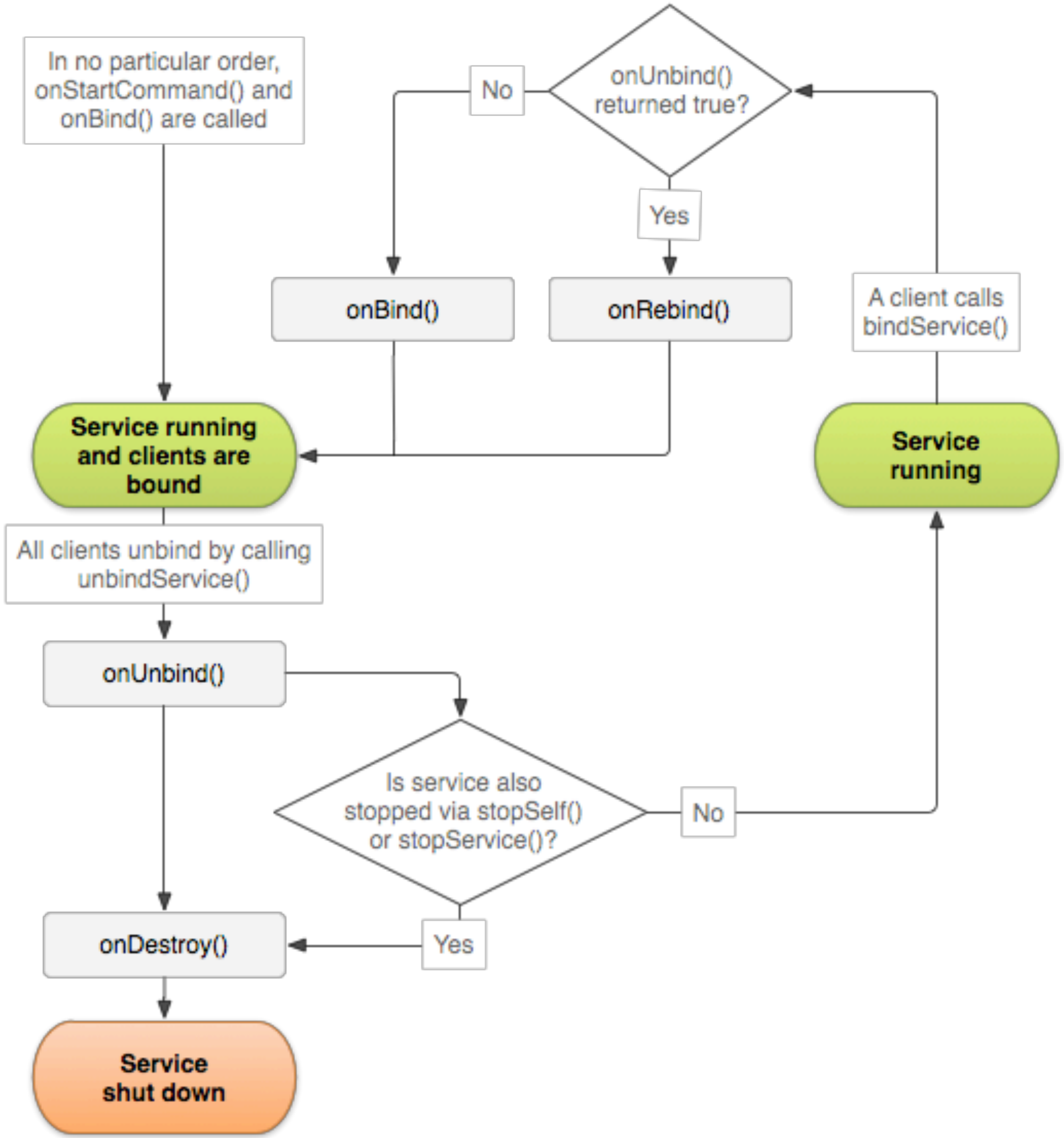
Terminating Services

- A Service runs in the background indefinitely
  - Even if the component that started it is destroyed
  - onStartCommand return value determines how the service should be continued if it is destroyed
- START_NOT_STICKY
  - After onStartCommand returns, do not recreate the service unless there are intents to deliver
- START_STICKY
  - Recreate the service and call onStartCommand again, but do not redeliver the last intent
- START_REDELIVER_INTENT
  - Recreate the service and call onStartCommand again, redeliver the last intent
    - Immediately resume the previous job, i.e. downloading a file

Notifications

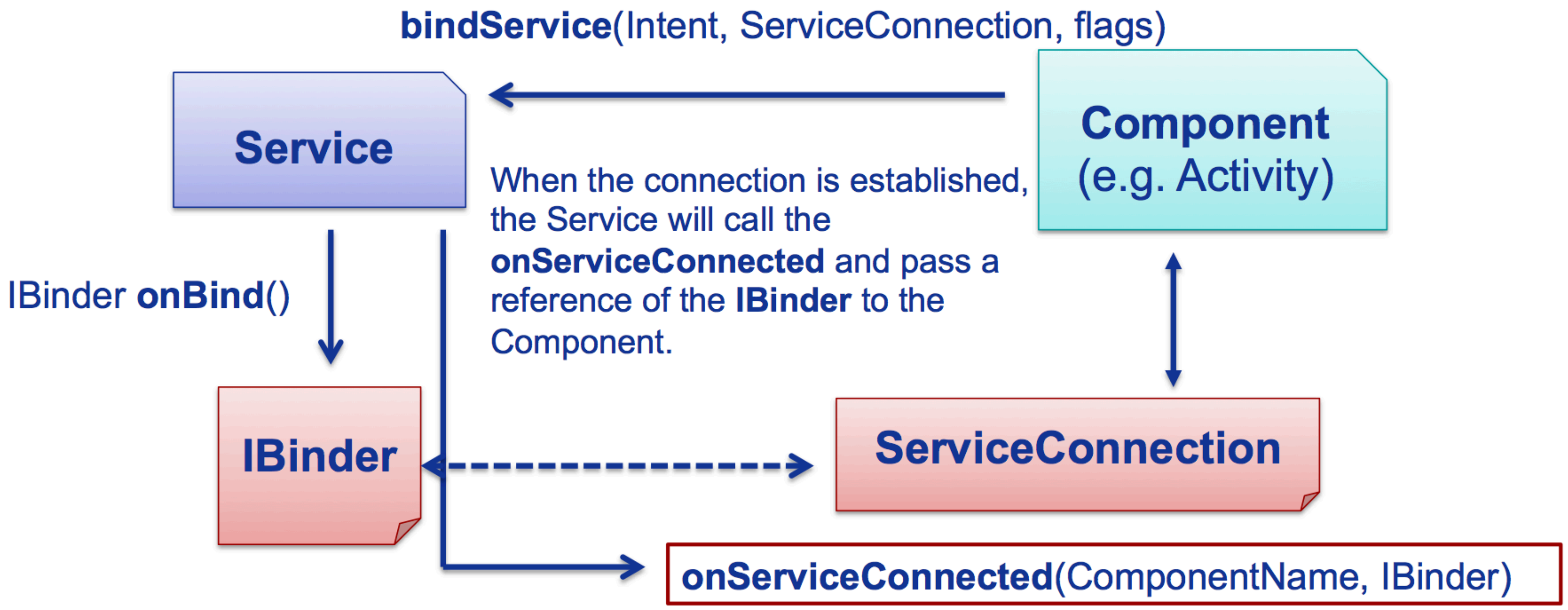
- But how do we notify the user that the Service is operating / has done something?
  - The original Activity may no longer exist
- Status bar notification
  - Maintained by the service
  - Can specify an Intent / Activity to launch if the user clicks on it
    - Return to the Activity that spawned the Service
    - Via a *Pending* Intent


Call to startService()
onCreate()
onStartCommand()
Service running
The service is stopped by itself or a client
onDestroy()
Service shut down
Unbounded service
Call to bindService()
onCreate()
onBind()
Clients are bound to service
All clients unbind by calling unbindService()
onUnbind()
onDestroy()
Service shut down
Bounded service
Active Lifetime


In no particular order, onStartCommand() and onBind() are called
No
onUnbind() returned true?
Yes
onBind()
onRebind()
A client calls bindService()
Service running and clients are bound
Service running
All clients unbind by calling unbindService()
onUnbind()
Is service also stopped via stopSelf() or stopService()?
No
Yes
onDestroy()
Service shut down

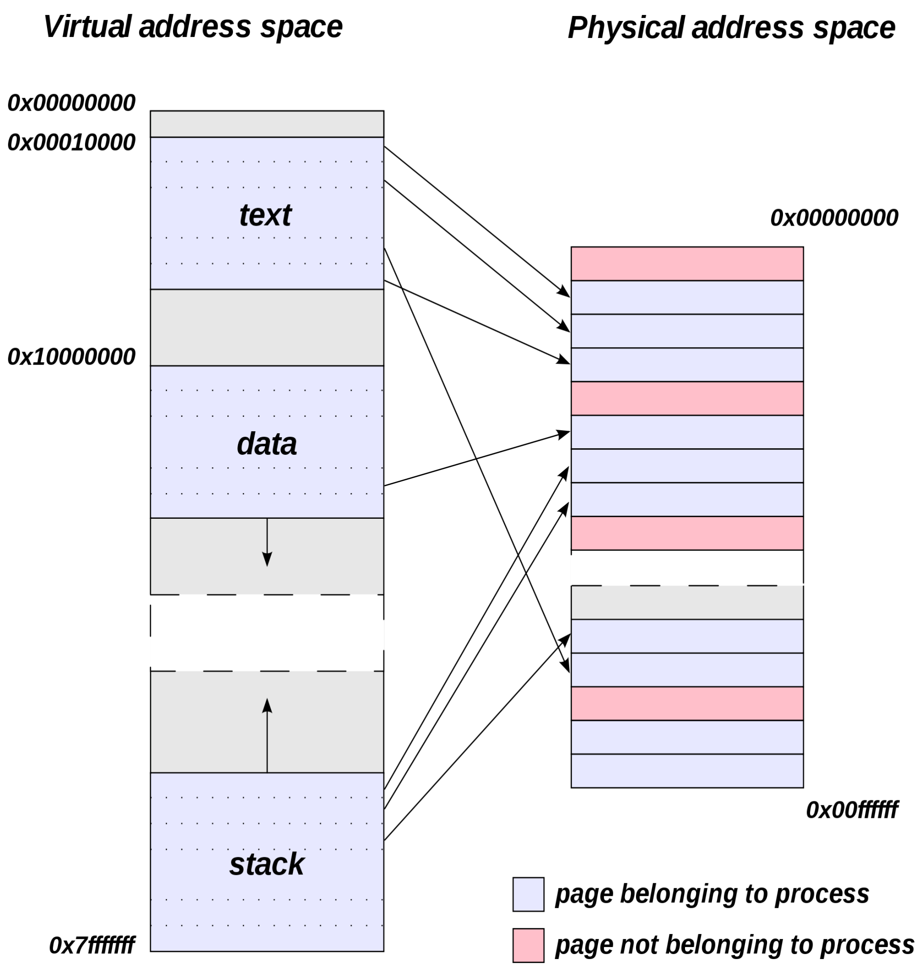
Bound Services

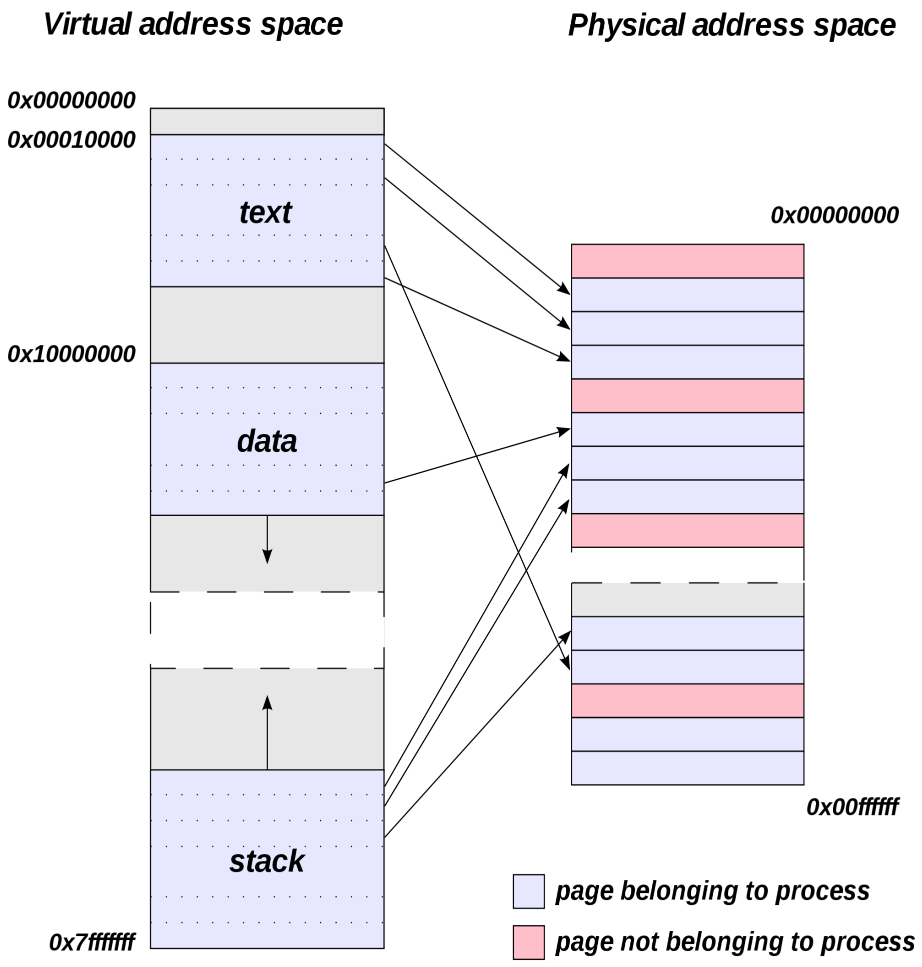
- If not explicitly started, will be started by the o/s
  - …when something binds to it
  - Then stopped if everything unbinds from it
  - What is it **is** explicitly started?
- Provide an interface for clients (Activities) to interact with a Service
  - Provide a programmatic interface for clients
  - Fast *and* stable?
- **Extending** the Binder class
  - Return an interface via the onBind method
  - Only for a Service used by the same application
    - Local Services only
      - i.e. the same process
    - Make method calls within the same JVM
- Binder object asynchronously provides a reference to the service that we can call methods on
  - Via *ServiceConnection*
  - Why asynchronous?


bindService(Intent, ServiceConnection, flags)
Service
Component
(e.g. Activity)
IBinder onBind()
IBinder
ServiceConnection
onServiceConnected(ComponentName, IBinder)

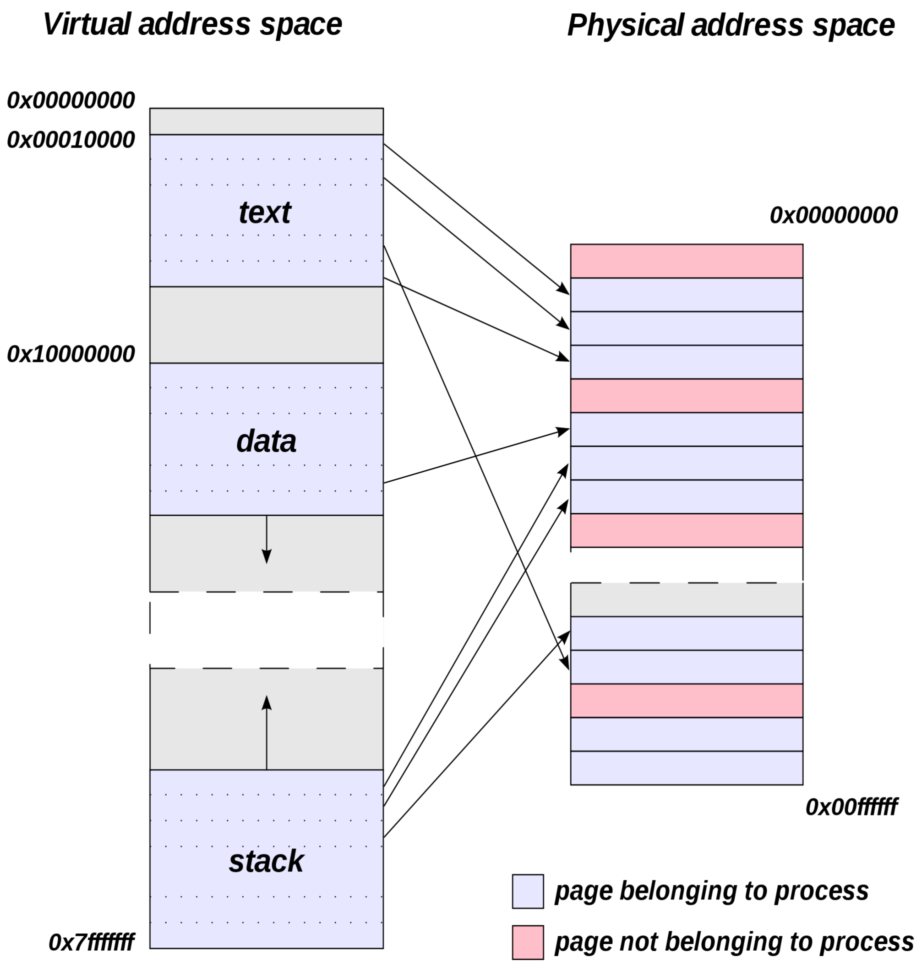
When the connection is established, the Service will call the **onServiceConnected** and pass a reference of the **IBinder** to the Component.

Let’s have a look…


Virtual address space
Physical address space
0x00000000
0x00010000
text
0x10000000
data
stack
0x7fffffff
0x00000000
0x00ffffff
page belonging to process
page not belonging to process



Virtual address space
Physical address space
0x00000000
0x00010000
text
0x10000000
data
stack
0x7fffffff
0x00000000
0x00ffffff
page belonging to process
page not belonging to process



Virtual address space
Physical address space
0x00000000
0x00010000
text
0x10000000
data
stack
0x7fffffff
0x00000000
0x00ffffff
page belonging to process
page not belonging to process

Remote Services

- For communicating across process boundaries
  - i.e. using a Service belonging to a different application / process
  - Likely to be used by multiple processes at once
- Starting the service
  - Declare the service as *exported* in the Manifest
    - Explicit rather than implicit
    - More sophisticated permissions system later on
  - Must not use *implicit* Intents
    - Added in later Android SDK versions
    - Why?
- Communicating
  - Using a Messenger
    - Simplest implementation
    - C.f. using a Handler to talk between Threads
      - Queues Messages into a single Thread, handled sequentially
        - » Bundles of data instead of method calls
    - Messages must be Parcelable
    - Bi-directional communication
  - Defining an interface
    - System services

Parcelable

- Locally (same process) bound Services share the same process memory space
  - Easy to call methods, transfer objects / references between classes
- How should different processes talk to each other?
  - java.io.Serializable
    - Short-term persistence
    - Write object ID, field via reflection
    - Change the class / variable name, what happens?
    - Slow
  - Parcelable
    - Define a simple wire-protocol for writing primitives
      - Re-create an object by passing salient data (c.f. deep copy)
    - Immune to minor changes to class definitions
      - Same interface, different class
    - Supported by Android kernel driver
    - Fast!

Remotely Bound Services

- Using the Android Interface Definition Language (AIDL)
  - Provide a standard interface to access the Service from different applications
    - Specify an interface and protocol to cross process boundaries
    - Trigger method calls to a different JVM, return results
- Define remote interface in the Android Interface Definition Language (AIDL)
  - Providing OS wide services for all applications
    - i.e download management
  - Multithreading with complex client / server bi-directional communication
    - A thread pool handles concurrent method calls
- Implement remote interface
  - Stub and application specific methods
- Implement Service methods
- Implement Client methods

AIDL

- Similar to Java interface definition syntax
  - Can declare methods
  - Cannot declare static fields
- Label method parameters
  - in: transferred to the remote method
  - out: returned to the caller
  - inout: both in and out
- Types
  - Java **primitive** types
  - StringList
    - List elements must be valid AIDL data types
  - Map
    - Map elements must be valid AIDL data types
  - CharSequence
  - Other AIDL-generated interfaces
  - Classes implementing the Parcelable protocol

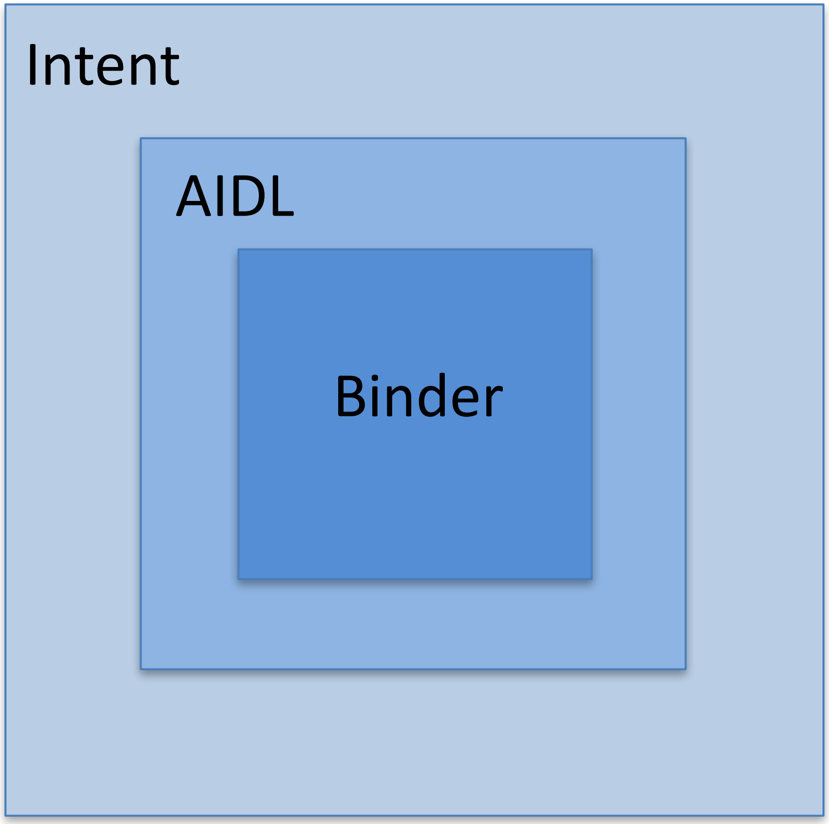
Let’s have a look…

IPC

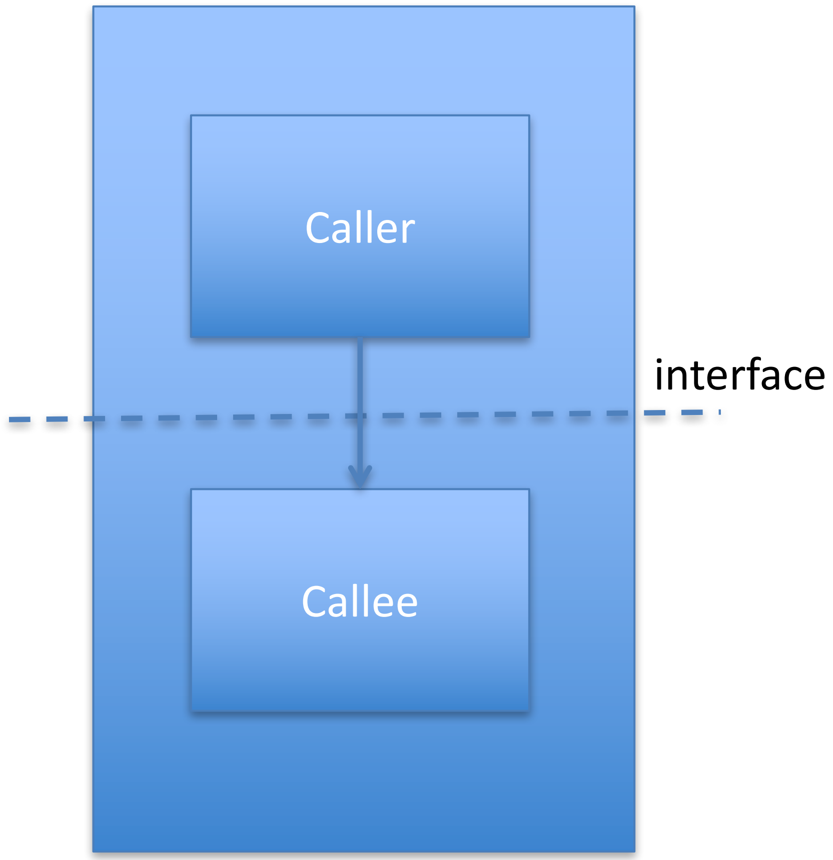
- Each process has its own address space
  - Provides data isolation
  - Prevents direct interaction between different processes
    - However, often required for modularisation
- What **actually** happens when we start a Service, or send an Intent?
- Binder
  - Underpins most Android communication
    - i.e. when we use the NotificationManager
  - Provides lightweight RPC (remote procedure communication)
    - C.f. Linux/Unix signals / pipes / sockets etc
  - Kernel driver
  - High performance via shared memory
    - Reading and writing *Parcels* between processes
  - Per-process thread pool for handling requests
  - Synchronous calls between processes

IPC Abstraction

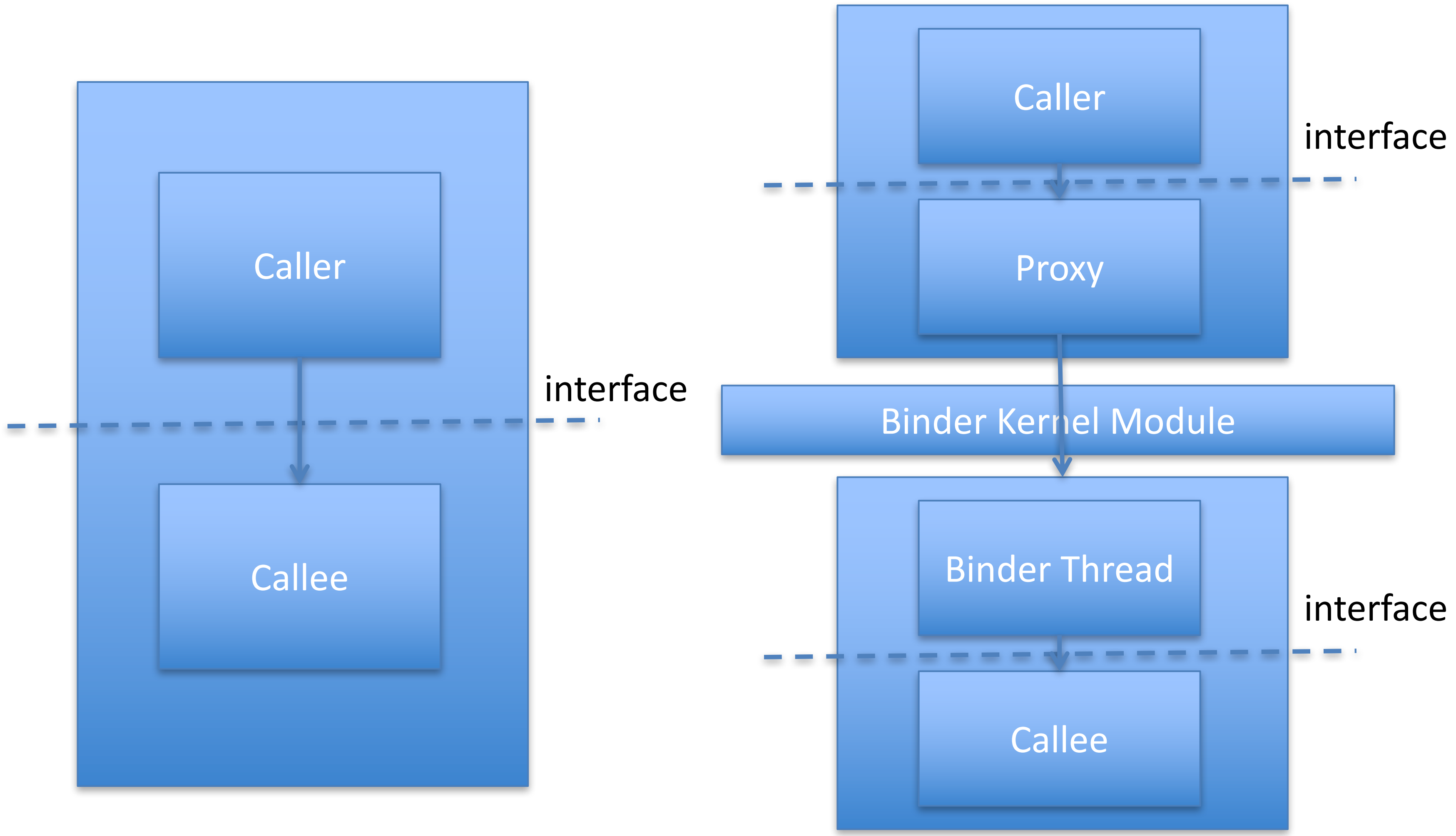
- Intent
  - Highest level abstraction
- Inter process method invocation
  - AIDL
- binder: kernel driver
- ashmem: shared memory



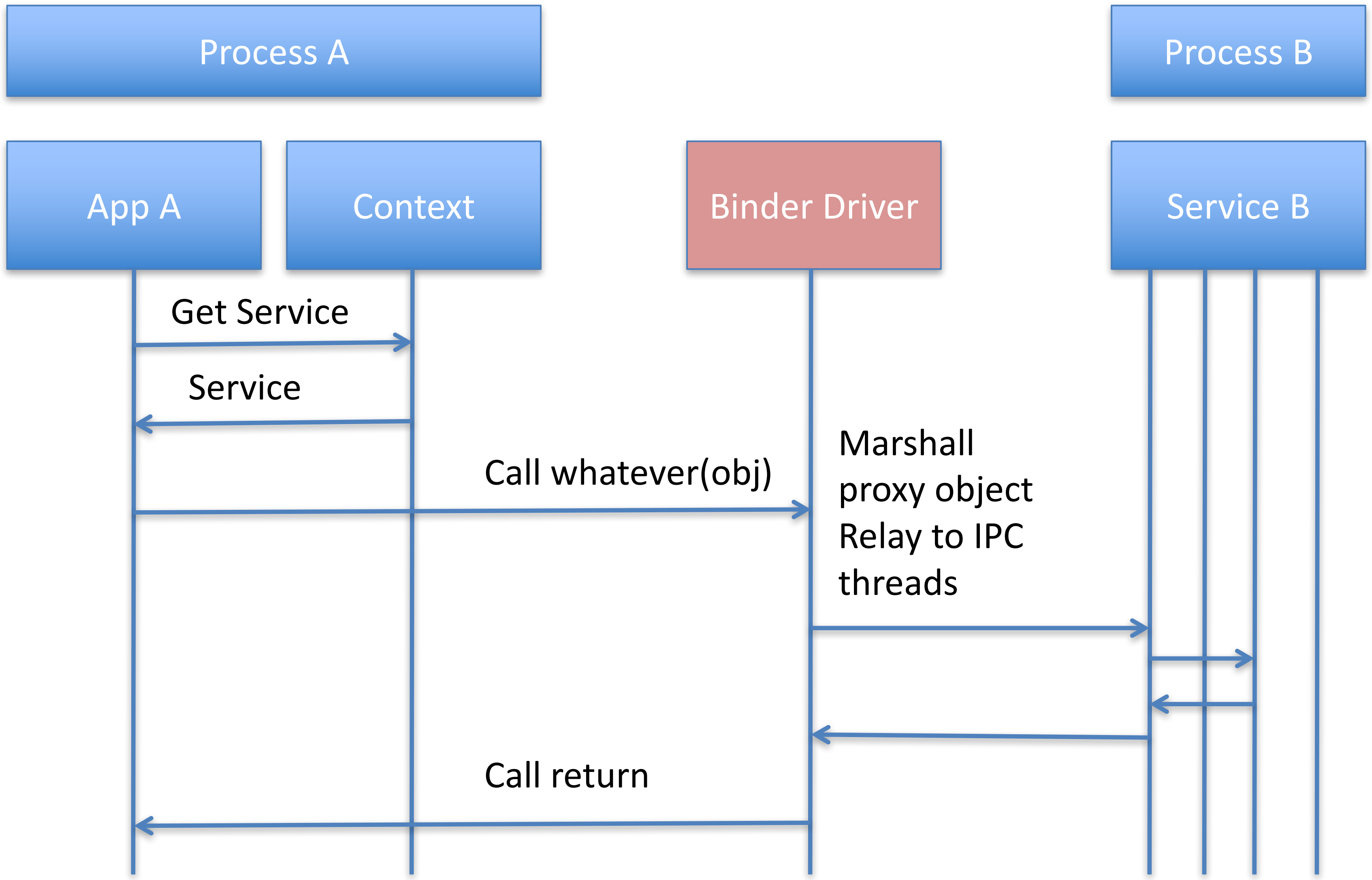
Inter-process method invocation



Inter-process method invocation



Binder in action



References

- http://developer.android.com/guide/components/processes-and-threads.html
- http://developer.android.com/guide/components/services.html
- http://elinux.org/Android_Binder