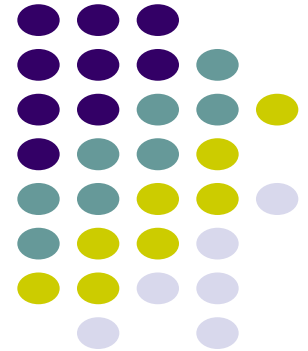


OS Designs

ECE 469, April 21

Aravind Machiry



How to design a OS?

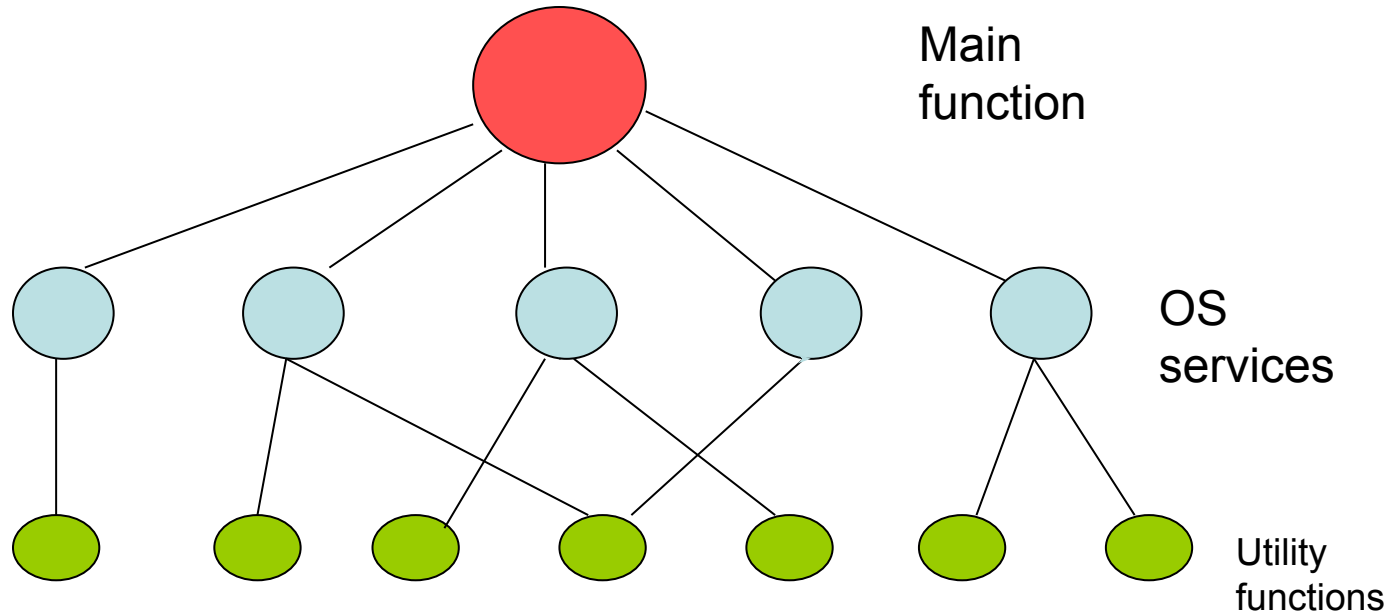


- Design: Components, organization of these components and interaction between the components.
- Also, called OS architectures.

Design 1: Monolithic



- All components bundled as a single entity.





Monolithic Kernel

- All OS functionality is included in a single program (address space)
 - E.g., UNIX, Linux, most commercial systems
- Advantages:
 - Easy to design and reason about.
 - Good performance.
- Disadvantages:
 - Poor separation: kernel components aren't protected from each other.
 - Cannot be easily extended.

Linux Kernel



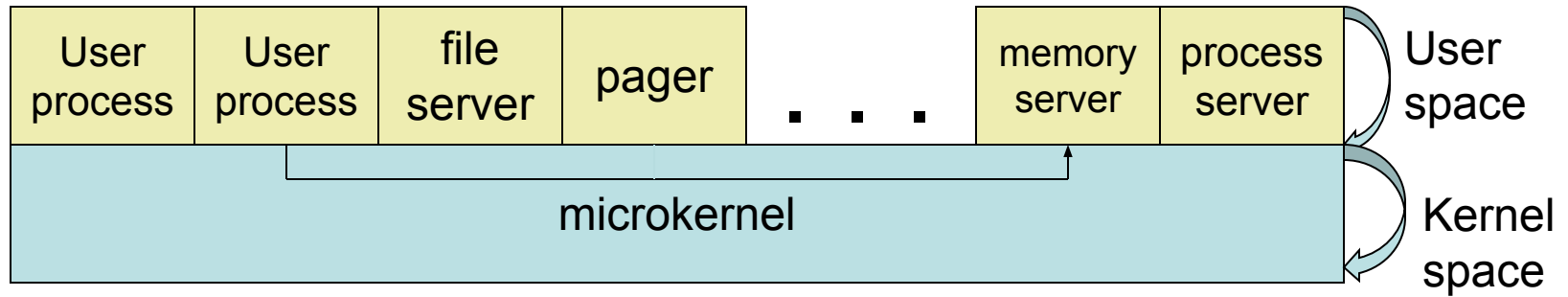
- Mainly monolithic.
- Extensible: Kernel modules.
- Fairly modular.



Microkernels - outline

- OS kernel is very small – minimal functionality.
- Other OS functions provided at user level by **trusted** servers.
 - User-level process, but trusted by kernel
- Advantage: Design reflects good software engineering practices
- Problem: performance

Microkernels



Microkernels - approach



- The microkernel layer provides a set of minimal core services and is the interface to the hardware layer.
- Other services (drivers, memory managers, etc.) are implemented as separate modules with clearly defined interfaces.



Microkernels - Advantages

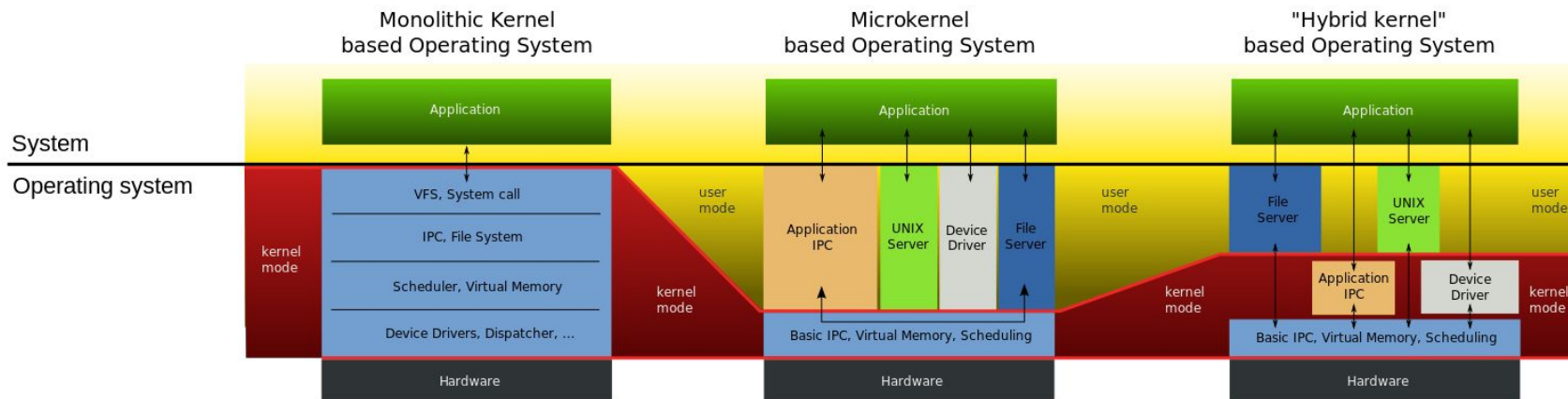
- Modularity
- Flexibility and extensibility
 - Easier to replace modules – fewer dependencies
 - Different servers can implement the same service in different ways
- Safety (each server is protected by the OS from other servers) using standard OS memory protection techniques
- Servers are largely hardware independent
- Correctness
 - Easier to verify a small kernel
 - Servers are isolated; errors in one don't affect others



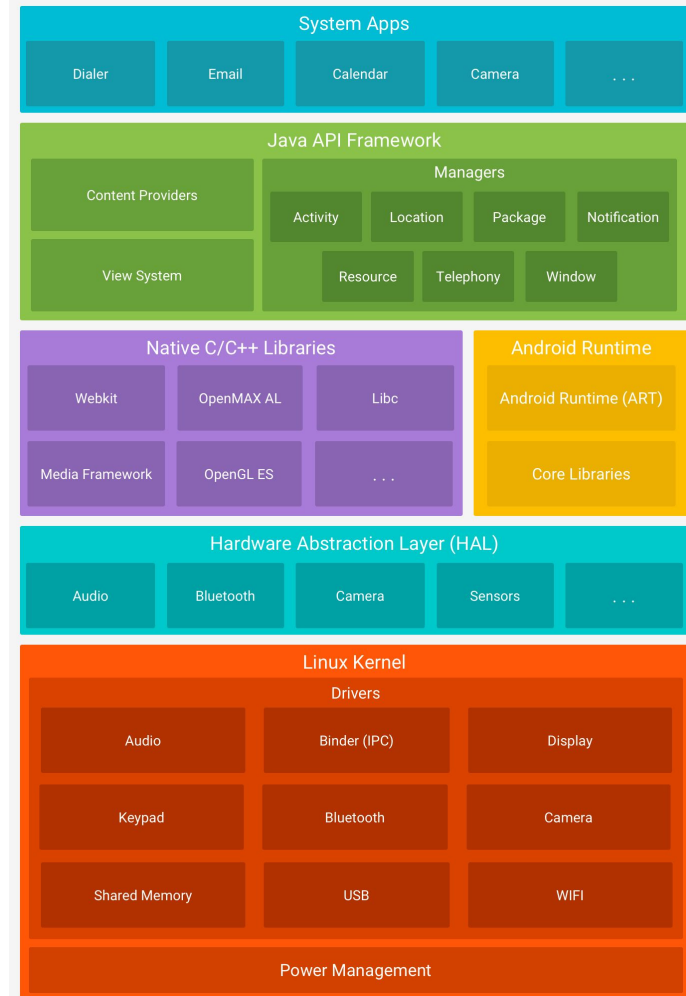
Microkernels - Disadvantages

- Slow – is this due to “cross-domain” information transfers? Maybe
 - Server-to-OS, OS-to-server IPC is thought to be a major source of inefficiency
 - Mode switches/context switches
- Generally it's faster to communicate between two modules that are both in OS – no mode switching involved

Monolithic/Micro/Hybrid kernel



Case Study: Android



From the eyes of an app

- Android is based on Linux
- Each app has its own Linux user ID*
- Each app lives in its own security *sandbox*
 - Standard Linux process isolation
 - Restricted file system permissions

* There are ways to setup apps so that they share the user ID. See "sharedUserId".

App Installation

- The Android framework creates a new Linux user
- Each app is given a private directory
 - Also called "Internal Storage"
 - No other app can access it*

* There are ways to setup apps so that they share the user ID.
See "sharedUserId".

App Isolation

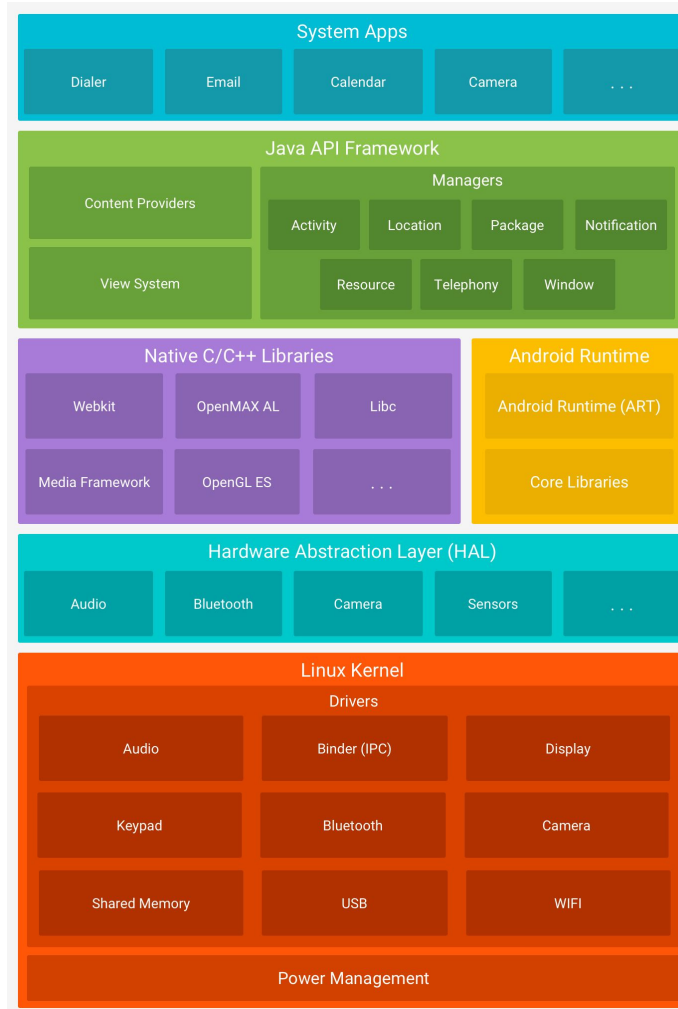
- Apps are run in separate processes
- Apps being in sandbox means that they can't
 - talk to each other
 - do anything security-sensitive
- Q: how can apps do anything interesting?
- This is when architecture & security get mixed up

Example: Saving a file

- Going down: Java ~> libc ~> syscalls
- `fd = open(const char *filename, int flags, umode_t mode)`
- `n = write(unsigned int fd, char *buf, size_t count)`
- `close(unsigned int fd);`

Not all requests are as easy as opening a file...

- Get current location?
- Send an SMS?
- Display something to the UI?
- Play a sound?
- Talk to other apps!?



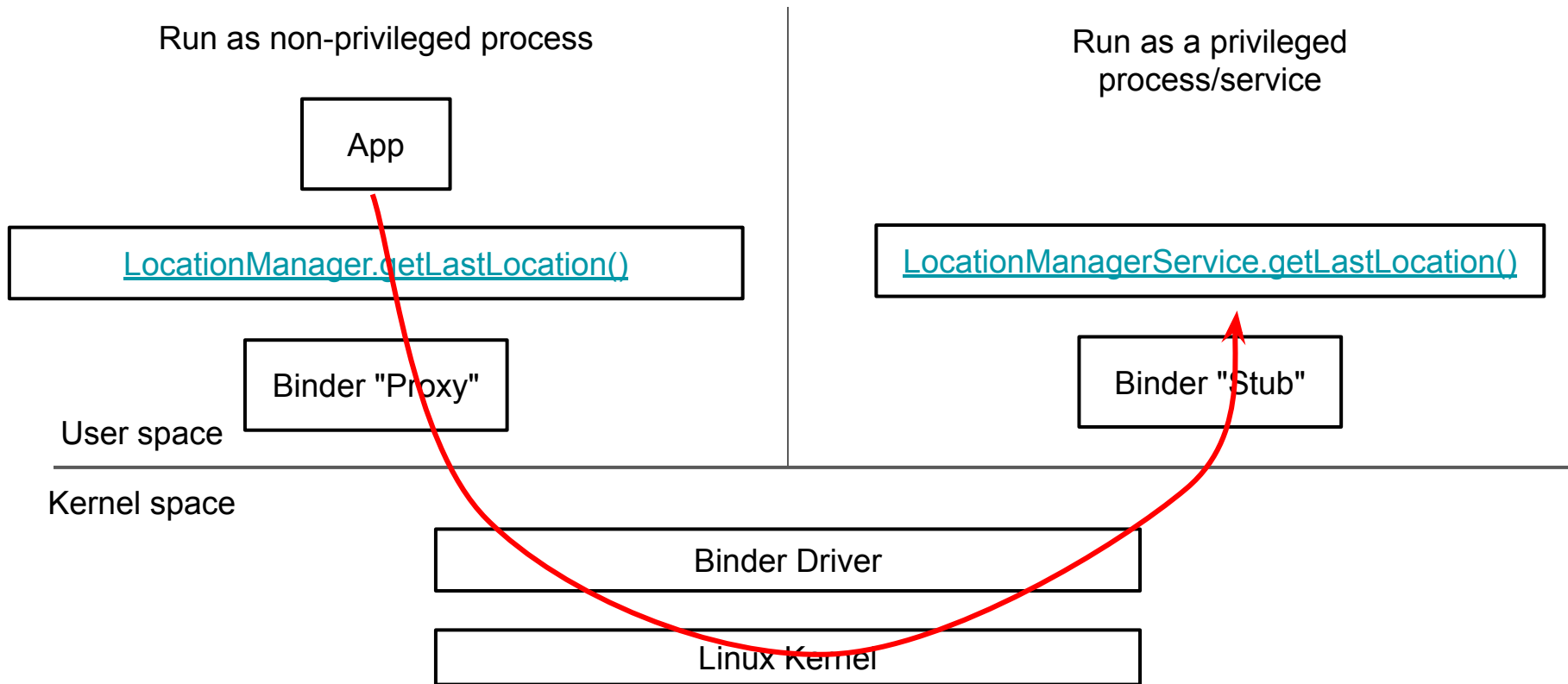
Example: getLastLocation()

- App invokes Android API
 - `LocationManager.getLastLocation()` ([ref](#))
 - We are still within the app's sandbox!
- Actual implementation of the privileged API
 - `LocationManagerService.getLastLocation()` ([ref](#))
 - We are in a "privileged" service
- How do we go from one side to the other one?

Crossing the bridge

- Binder!
- Binder: one of the main Android's "extensions" over Linux
- It allows for
 - Remote Procedure Call (RPC)
 - Inter-Process Communication (IPC)

Binder RPC



Binder details

- Proxy and Stub are automatically generated starting from [AIDL](#)
- Binder internals
 - /dev/binder
 - ioctl syscall
 - Multi-purpose syscall, to talk to drivers
 - The Binder kernel driver takes care of it, dispatches messages and returns replies

Many "Managers"

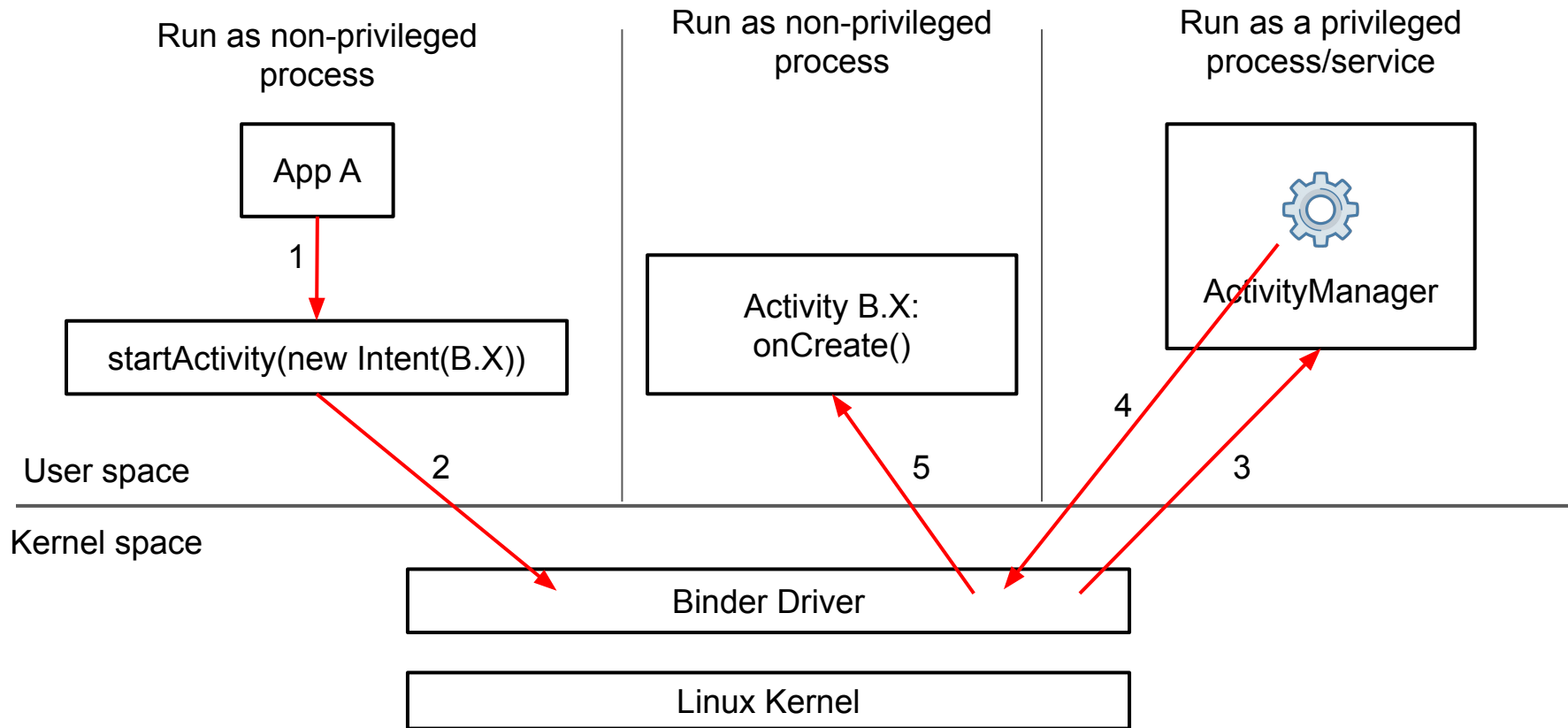
- Activity Manager
- Package Manager
- Telephony Manager
- Resource Manager
- Location Manager
- Notification Manager
- Resource Manager

```
$ adb shell service list
```

Binder as IPC mechanism

- How do apps talk to each other?
- High-level API: Intents
- Under the hood: Binder calls!

Binder IPC: $A \rightarrow B.X$



What about security?

- Can an app always do all these things? Nope.
- It has a private folder... that's it?
 - It can start other apps (the main activity is always "exported")
 - It can show things on the screen (when the app is in foreground)
- It can't
 - Open internet connection
 - Get current location
 - Write on the external storage
 - ...

Android Permission System ([overview](#), [ref](#))

- Android framework defines a long list of permissions
- Each of these "protects" security-sensitive capabilities
 - The ability to "do" something sensitive
 - Open Internet connection, send SMS
 - The ability to "access" sensitive information
 - Location, user contacts, ...

Examples of Permissions

- INTERNET (string: "android.permission.INTERNET")

Examples of Permissions

- [INTERNET](#) (string: "android.permission.INTERNET")
- ACCESS_NETWORK_STATE, ACCESS_WIFI_STATE, CHANGE_NETWORK_STATE, READ_PHONE_STATE
- ACCESS_COARSE_LOCATION, ACCESS_FINE_LOCATION
- READ_SMS, RECEIVE_SMS, SEND_SMS
- ANSWER_PHONE_CALLS, CALL_PHONE, READ_CALL_LOG, WRITE_CALL_LOG
- READ_CONTACTS, WRITE_CONTACTS
- READ_CALENDAR, WRITE_CALENDAR
- READ_EXTERNAL_STORAGE, WRITE_EXTERNAL_STORAGE
- RECORD_AUDIO, CAMERA
- BLUETOOTH, NFC
- RECEIVE_BOOT_COMPLETED
- SYSTEM_ALERT_WINDOW
- SET_WALLPAPER

Permissions from an app's perspective

Permission Request

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.awesomeapp">

    <uses-permission android:name="android.permission.SEND_SMS"/>
    <application ...>
        ...
    </application>
</manifest>
```

Custom Permissions ([doc](#))

- Apps can define "custom" permissions!

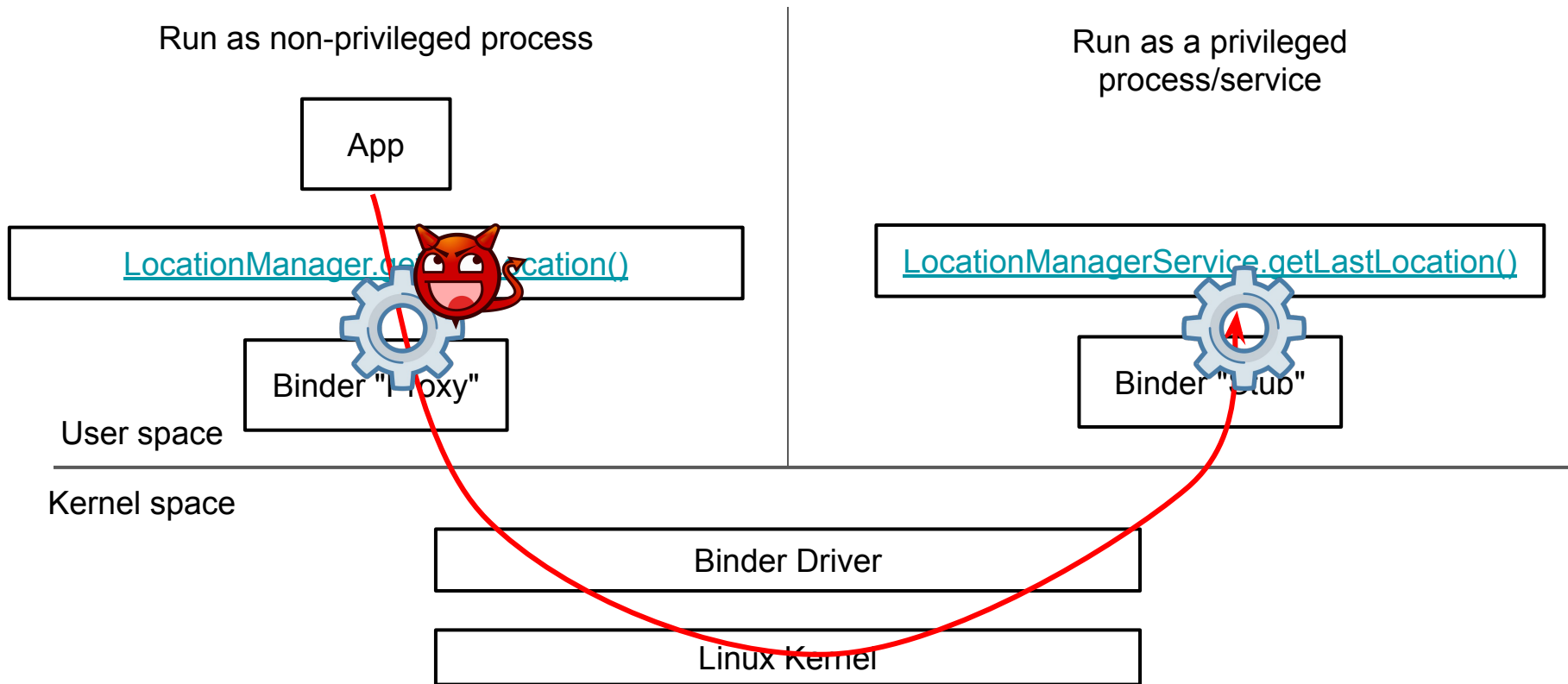
```
<permission
    android:name="com.example.myapp.permission.DEADLY_STUFF"
    android:label="@string/permlab_deadlyStuff"
    android:description="@string/permdesc_deadlyStuff"
    android:permissionGroup="android.permission-group.DEADLY"
    android:protectionLevel="signature" />
```

- The "system" permissions are defined in the same way
 - [AndroidManifest.xml](#)

Permission Enforcement Implementation

- Two technical ways: Linux groups vs. explicit checks
- Linux groups
 - INTERNET permission ~> app's user is added to "inet" Linux group
 - BLUETOOTH permission ~> app's user is added to "bt_net" Linux group
 - Declaration in AOSP: [code](#)
- Explicit check

Binder RPC



Explicit Checks

- The service's code actually does a check
- LocationService
 - [LocationManagerService.getLastLocation\(\)](#)
 - [LocationManagerService.checkResolutionLevellsSufficientForProviderUse\(\)](#)
- AudioFlinger
 - [ServiceUtilities.recordingAllowed\(\)](#)
 - [ContextImpl.checkPermission\(\)](#)
 - [ActivityManagerService.checkPermission\(\)](#)
 - [ActivityManagerService.checkComponentPermission\(\)](#)
 - [ActivityManager.checkComponentPermission\(\)](#)
 - [PackageManagerService.checkUidPermission\(\)](#)



Case Study: FreeRTOS

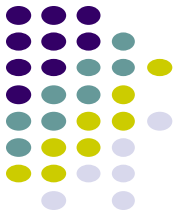
- A Real Time Operating System
- Written by Richard Barry & FreeRTOS Team
- Owned by Real Time Engineers Ltd but free to use
- Huge number of users all over the world
 - 6000 Download per month
- Simple but very powerful





Real Time Operating System?

- A type of an operating system
- It's all about scheduler :
 - multi user operating system(UNIX) — fair amount of the processing time
 - desktop operating system(Windows) — remain responsive to its user
 -
- RTOS scheduler focuses on **predictable** execution pattern



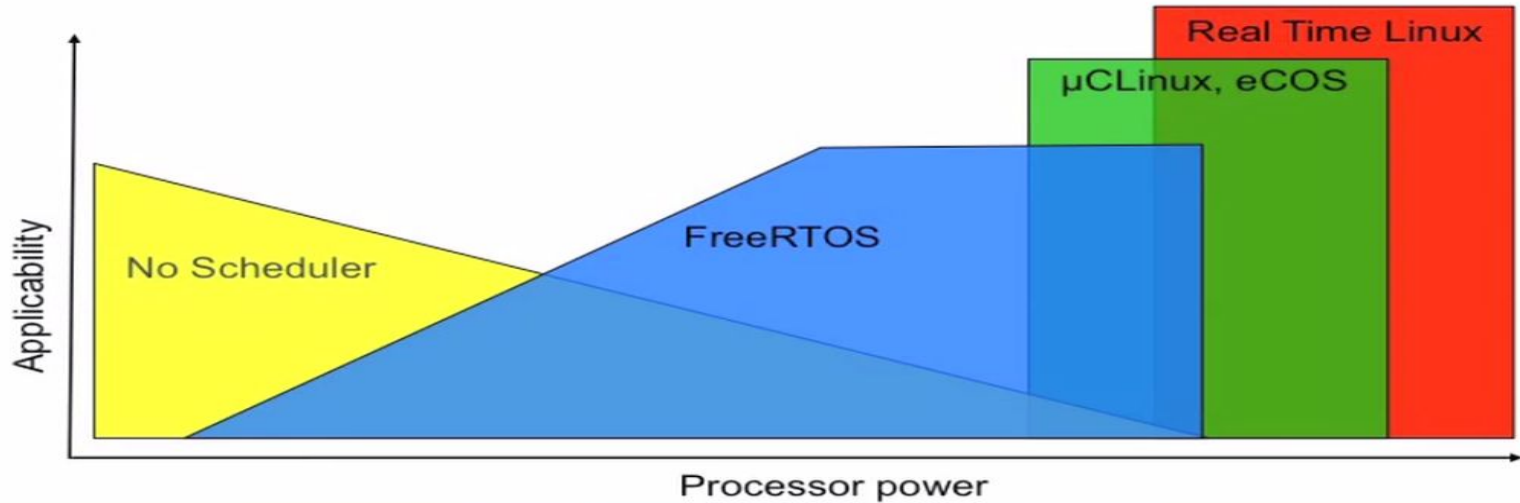
Why FreeRTOS?

- Abstract out timing information
- Maintainability/Extensibility
- Modularity
- Cleaner interfaces
- Easier testing (in some cases)
- Code reuse
- Improved efficiency?
- Idle time

.....

.....

When to use FreeRTOS?





FreeRTOS Architecture?

- Tasks (50%)
 - `task.c` and `task.h` do all the heavy lifting for creating, scheduling, and maintaining tasks.
- Communication (40%)
 - `queue.c` and `queue.h` handle FreeRTOS communication. Tasks and interrupts use queues to send data to each other and to signal the use of critical resources using semaphores and mutexes.
- Hardware Interfacing (6%)



User space/Kernel space

- No MMU => No Separation!
- Large Program:
 - Each task => Thread
 - Scheduling between tasks (i.e., threads) based on priority.



Ideal OS architecture?

- Hard.
- Different use cases / Hardware capabilities/ Requirements.
- Even the same OS is customized for different use cases:
 - Ubuntu => Desktop v/s Server.