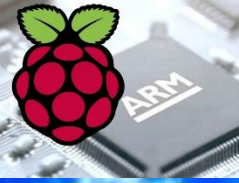


EEE3096S



Embedded Operating Systems (1)

Section 4.2 of the textbook relates

Embedded Systems II

L5

Dr Simon Winberg



Electrical Engineering
University of Cape Town

Outline of Lecture P02

- Operating system types
- What an operating system needs
- What type of OS is Linux?
- Some useful terms
- Operating system customization
- What an Operating System needs
- Characteristics of an Embedded OS
- Common Design Optimizations of an EOS

Operating System Types

Operating Systems can be classified according to

- What type of system are they design to support?
- What type of design do they follow?
- To what extent can they be customized?

Operating System Types

What type of system are they design to support?

- “General-purpose O/S” (GOS)
(technically a misnomer, but a useful term nonetheless)
- Server
- Workstation
- Distributed
- Embedded
- Real-time
- Real-time Embedded

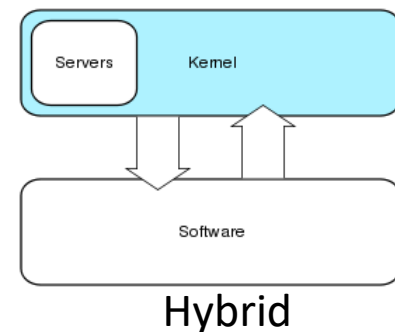
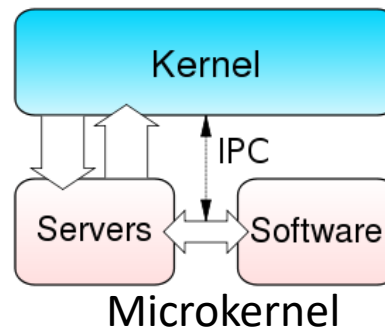
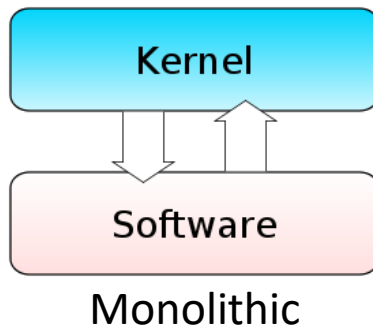
In this course we are going to focus on the embedded and real-time embedded flavours

Operating System Types

What type of design do they follow?

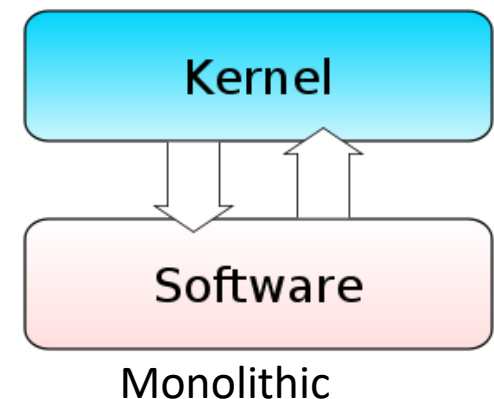
- Monolithic
- Monolithic Kernel
- Modular (usually Monolithic)
- Microkernel
- Hybrid

.. Let's explore these definitions a little more ...



Monolithic Operating System

- Monolithic Kernels
 - All kernel functions and device drivers inside one address space running in kernel mode.
 - The principle advantage to this approach is **efficiency** (especially on x86 architectures where task switches are expensive).
 - Monolithic kernels suffer the risk of the entire system crashing due to a bug in a device driver.



Monolithic Operating System

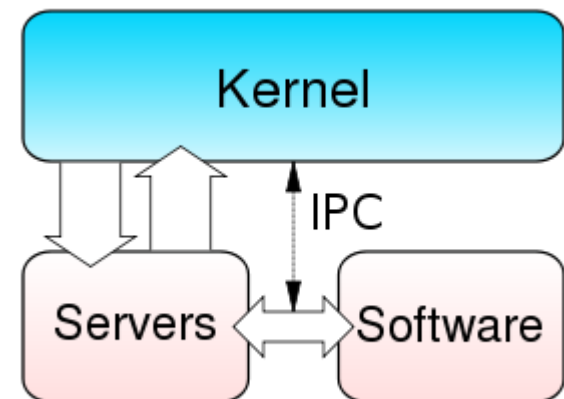
- Monolithic Kernels Benefits
 - Speed
 - Simplicity of design (sort of)
- Monolithic Kernels Drawbacks
 - Potential stability issues
 - Can become huge (Linux 4.15 has 20 million lines of code, Windows 10 has over 40 million lines!)
 - Potentially difficult to maintain
- Examples
 - Traditional Unix kernels (includes BSDs and Solaris)
 - Windows, MS-DOS
 - Mac OS versions below 8.6

Modular Monolithic Kernel

- A modular monolithic kernel essentially just supports modules (usually device drivers or services, e.g. nfs) which can be included or activated/deactivated
- But it is still a monolithic kernel, but it may be a bit more robust than a totally monolithic kernel where all the kernel code is closely collocated.

Microkernel

- A Microkernel tries to run most of its services/ drivers in userspace.
- This is a significant change from a monolithic kernel! But it doesn't mean to say the OS is a lot smaller.
- This can increase stability and possibly security.



Microkernel

- Microkernel Benefits
 - Stability and Security
(more correctly, the system is less likely to complete die, can still keep limping along if drivers are failing)
 - *Some* functions more responsive (e.g. very core functions)
- Microkernel Drawbacks
 - Additional context switches,
 - Slow Inter Process Communication (IPC) can cause poor performance
 - Potentially slow threads (lethargic non-core service access)

Microkernel Examples

QNX, Minix, MorphOS, Amoeba,
(AmigaOS, historic one still in some arcades)



Hybrid Kernel

- Hybrid kernels combines both monolithic kernels and microkernels design approaches.
- Generally implemented as a monolithic kernel but following a more microkernel design.
- Essentially the aim here is to gain the monolithic benefits, but with microkernel stability
- Examples
 - NT kernel (used in Windows NT, 2000, XP, Vista and Windows 7)
 - Darwin (MacOS and iOS kernel)
 - DragonFly BSD and FreeBSD
(FreeBSD super stable! Might say FreeBSD is the “most thoroughly developed” because it’s been around for the longest – for server operation it is the most strongly recommended)



So What type of OS design is Linux?

Standard Linux (such as RedHat, Debian, and Raspbian) follows what sort of design? ...

- ☐ Monolithic
- ☒ Monolithic Kernel
- ☒ Modular (usually Monolithic)
- ☐ Microkernel
- ☐ Hybrid

Why? Because all services (file system, device drivers, etc) as well as core functionality (e.g. memory management, scheduling, etc.) are tightly bound in the same space. This is opposite to the concept of a microkernel.

Operating System Customization

To what extent can Operating Systems be customized?

- Some give you full access. To all sources, etc. e.g. most Linux versions (i.e. open source).
- For some full access might not help because the code is so difficult (i.e. customizing may take ages to figure out).
- Some O/S allow configuration of the executables/objects but you generally can't access the source (closed source)
- For some it's *all or nothing*... or it's your risk if you choose not to use it all (essentially what one would have done in the old days with DOS, earlier versions of Windows, etc.)
- It really depends on the operating system and the availability of the sources



What an Operating System needs



What does an Operating System Need?

- An Embedded Operating System (EOS) needs most things a General-purpose Operating System (GOS) needs.
- But there are some differences. e.g. for more task-specific and resource limited embedded systems particularly.
- Let's try to map out these differences: indicate a general OS function in the left column, a short description of this concept, and then the extent it is needed by a GOS or EOS, or if they both need a complete implementation of this. As an optional extra you can also consider RTEOS aspects.
- Optional: Real-time Embedded Operating Systems (RTEOS) essentially adds additional requirements to an EOS.

Feature/Function	General Concept	EOS vs GOS Needs
Process Management	Which tasks the OS should run	GOS probably needs more. EOS could load all programs at boot, no need to load later.

SOLUTIONS – What an OS Needs

Feature/Function	General Concept	EOS vs GOS Needs
Process Management	Which tasks OS should run, incl. loading, starting, scheduling, interrupts, etc.	Both need most of these, but EOS could have less, e.g. loads at boot, doesn't need to load more progs.
Threads (special case of process management)	Abstraction for multiple tasks not necessarily physically running at once	GOS needs this. EOS might not, e.g. if it just does simple polling.
(Primary) Memory Management	Separating out memory between running programmes	Both need, but EOS might have simple mem mng., e.g. no heap.
(Secondary) Memory Management	Manage data in non-volatile storage (e.g. on disk / flash mem)	GOS needs. EOS might not have secondary mem besides boot mem.
Services	Some means to access OS services	GOS needs many. EOS might not need any, or much less.
Networking (type of service)	Support for network applications	ES might not need any networking.
Resource management	Sharing resources between tasks	Both, but ES code might be very predictable not needing resource mng
IO / Device Control	Interfacing with devices attached to computer	GOS yes. ES might do it at app level.
Bootng	It may need to load other parts of the OS	GOS less so, it generally has enough main mem. ES might need to boot in stages due to 2 nd dry or main em limits.
Data security	Help protect system's data (e.g. from illegal use)	GOS definitely. EOS depends, e.g. data may be inaccessible outside the SOC.
(Basic) user interface	Some sort of console interface	Generally both!! (a good practice)
Graphics User Interface	Graphics interface	GOS yes. EOS depends, often not.
System health management	Check system is still working properly	GOS could assume the human operator will notice something isn't working. ES might be doing something important without a human to check
Prioritization	Prioritizing tasks	GOS would be nice. EOS / RTEOS sometimes essential. This list is by no means complete(!) but it covers the main things

Some unusual terms

You sometimes see these terms used when referring to an OS...

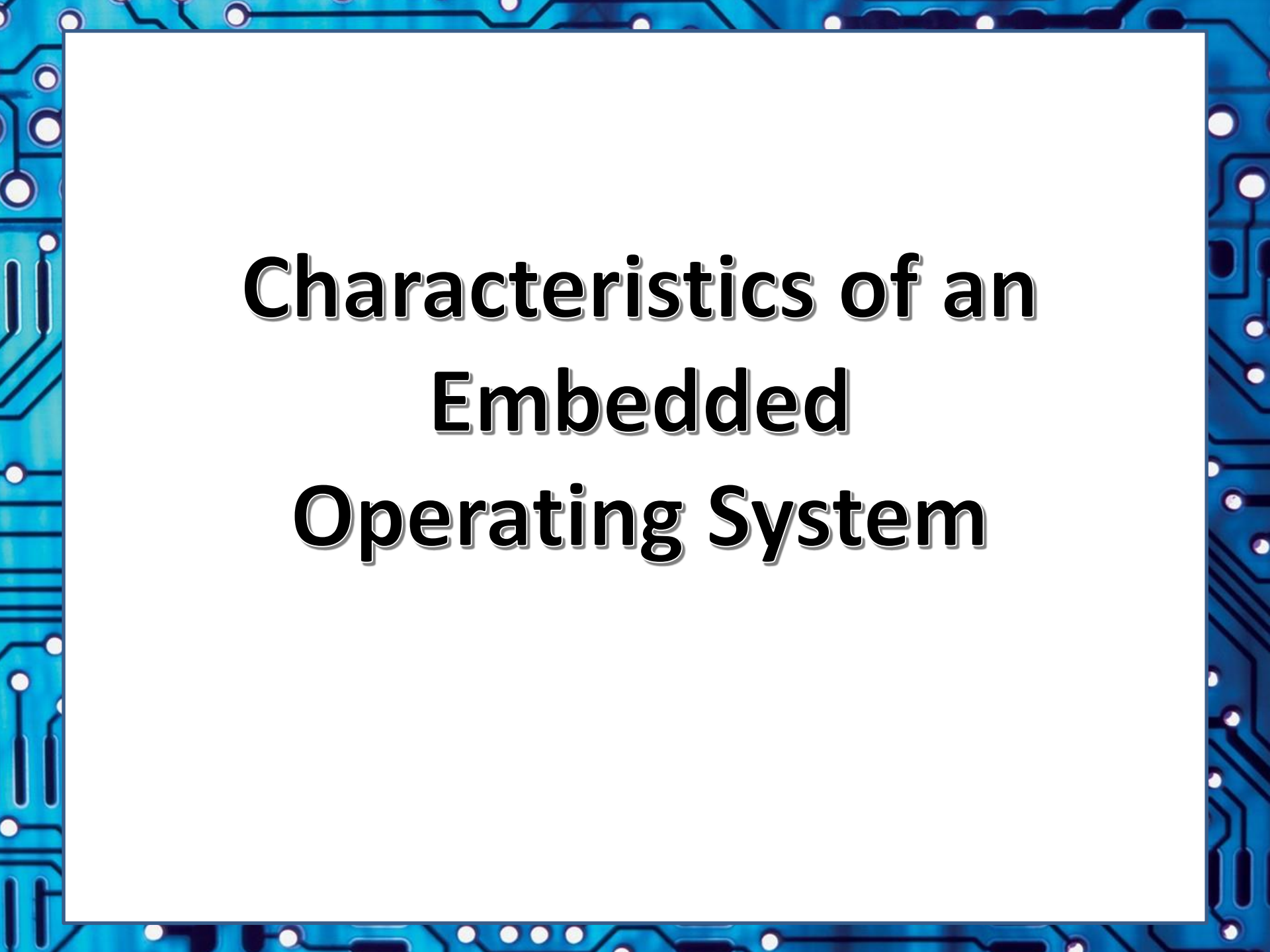
Extended Machine

The 'Extended Machine' features just means the OS provides multiple user access, possibly simultaneous use. Including sharing files between multiple users. Some provides graphical environments for various languages, features to assist disable users (e.g. for blind users) among other features.

Mastermind

Refers to a myriad of useful features, besides the basics (e.g. process management) that an OS provides. OS Mastermind includes for instance

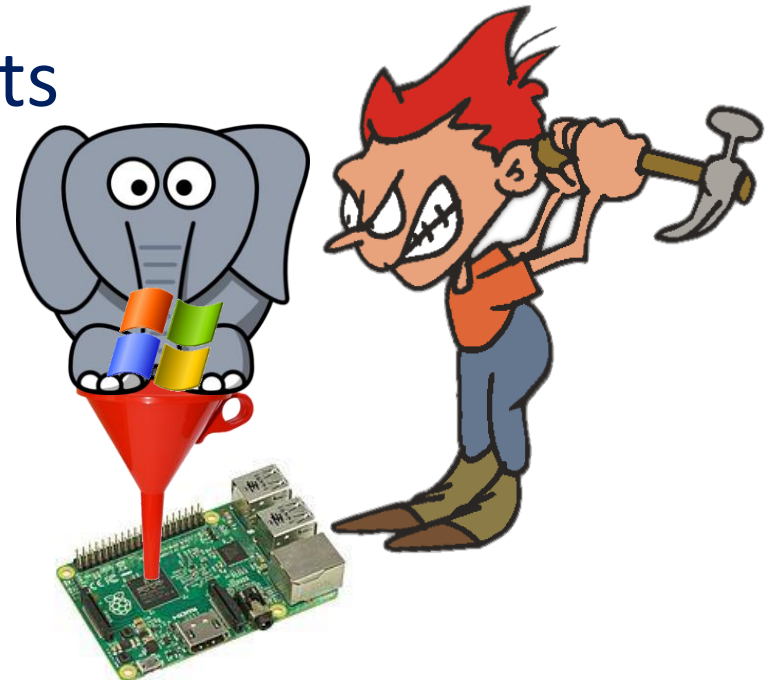
- Booting without an Operating System installed (auto self-install or temporary boot, eg. liveboot)
- Ways to increase logical memory using virtual memory (cleverly)
- Support for a broad variety of file formats, e.g. NTFS and FAT.

A decorative border with a blue and black circuit board pattern surrounds the central white text area.

Characteristics of an Embedded Operating System

Embedded Operating Systems

- Reasons why a General-purpose OS may not work for an embedded system
 - Constraint of memory space
 - Constraint of power consumption
 - Real-time requirements



Real-time Embedded OS

- **Real-time operation**
 - The correctness of computation depends partly on the time the result is delivered
- **Reactive operation**
 - Needs to consider worst-case conditions in execution so as to respond to external events that do not occur at predictable intervals

Embedded Operating Systems Design

- **Configurability**

- Supports configuration so that only the needed functionality for a specific application is provided (e.g. set it to just load the driver modules needed)

As Prof. Marwedel says: “No overhead for unused functions is tolerated and a single OS may not fit all the needs – so configurability is needed”

- **I/O device flexibility**

- Handles devices by using special tasks instead of integrating their drivers into the OS kernel – this makes it easy to add custom drivers.

Embedded Operating Systems Design

- **Streamlining protection**

- E.g. facility to cutting out usual security etc. features could be beneficial – so long as the system is such that its software is inaccessible.
- May sacrifice protection (e.g. exception handling) as tested software could be assumed to be reliable.
- I/O instructions need not be privileged instructions that need a call to the OS
 - can be convenient for tasks to access I/O directly without wasting cycles on an OS service call.

- **Possibly eliminating OS service calls together!**

- Just doing it as a function call and running in system mode... this could avoid overhead for saving/loading the task context



small but fast!

Embedded Operating Systems Design

- Direct use of interrupts
 - Allow user process to use interrupts directly
 - no need to go through OS interrupt services
 - Efficiently control a variety of devices right from one program

But Wait...

Doesn't this sound like you are taking a perfectly good design for an Operating System and tearing good things out of it?!

YES, in a way. But it should be done sensibly of course. Just taking things out you are quite sure are not necessary.



This leads to the general approaches of developing an EOS...

Developing/Configuring an Embedded OS

- There are essentially two approaches for EOS development
 1. Start with an existing OS (e.g. Linux) and adapt it for your embedded needs
 2. Start from scratch and design a purpose-built OS for your embedded system

When do you use one or the other approach?

- If it is a small but very resource constrained system, and you don't have much time then (2) is probably better, especially if you can just combine everything into one stand-alone program.
- If you are going to use compiles, standard OS calls and libraries (e.g. stdlib) then you probably want to use (1).

Common Design Optimizations of an EOS

- EOS are often designed from the ground up, designers achieving certain design optimizations
 - Some companies choose this approach, and their EOS may become a very valuable corporate asset.
- Typical design optimizations are:
 - Fast and lightweight thread switching
 - Real time scheduling policies
 - Small size
 - Small interrupt latency (e.g. $<10\ \mu\text{s}$)
 - Minimizes time interrupts are disabled
 - Fixed or variable sized memory partitions for faster or more predictable memory management
 - Logging services to accumulate data quickly

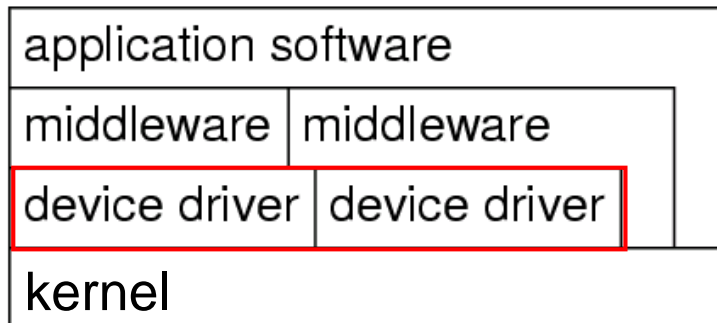
Added facilities for custom EOS

- A typical need is to deal with timing constraints:
 - provides bounded execution time for primitives
 - maintains a real-time clock
 - provides for special alarms and timeouts
 - supports real-time queuing disciplines, e.g., EDF (earliest deadline first)
 - provides primitives to delay processing and to suspend/resume execution

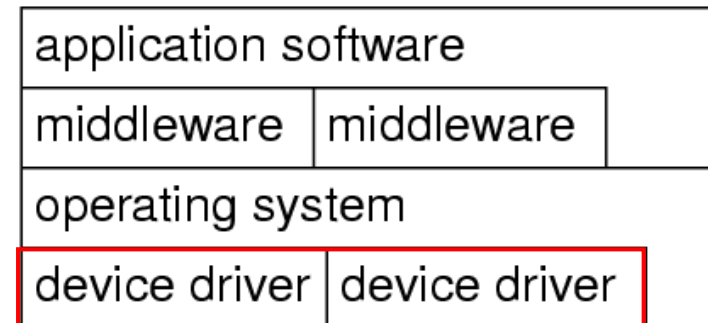
Main Design Trend for EOS

- A typically approach for designing or customizing an EOS is typically to move the device drivers – or OS services you need to access quickly or frequently – closer to the software that needs to use it ... maybe even into the application code that uses it!

Embedded OS



Standard OS



We will visit further detail on
Embedded Real-Time
Operating System

A little later in the course

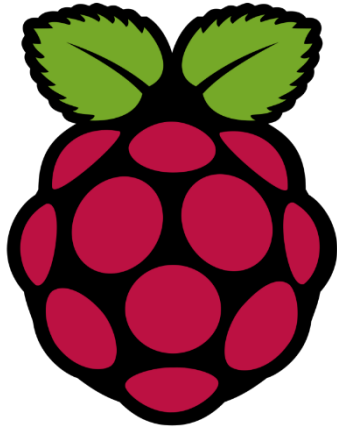
Middleware

Middleware ≠ Operating System Services

- Defn: Middleware
 - Software that provides services to applications beyond those available from the operating system. It can be described as “**software glue**”
- A common use of middleware:
 - Makes it **easier for programmers to implement communication** and input/output, so they can focus more of their time on application details.
- Middleware **gained popularity in the 1980s** as a solution to the problem of how to link newer applications to older legacy systems (although the term had been in use since 1968)*
- Middleware is mainly used for comms and data management in Distributed systems, but it is becoming increasingly more relevant to embedded systems, and there are increasingly more distributed embedded systems being developed.

* Gall, Nick (July 30, 2005). "Origin of the term middleware".

Design of Raspbian



See Lecture P2

The Next Episode...

Lecture P2

P02: Embedded Linux and Raspberian
In-depth view of the Raspberian Kernel



Reminder: test on Thursday! 12pm during class.