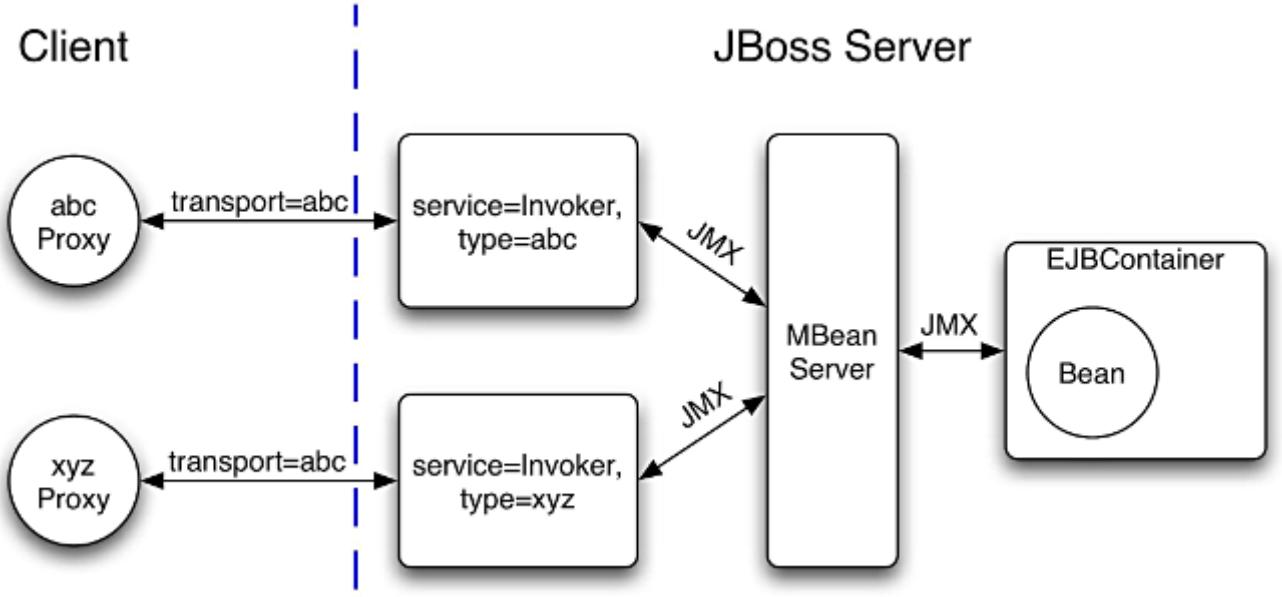
OSCAR

淺談 Microkernel 設計和真實世界中的應用

((本系列講座全名為 Operating System Concepts And Renaissance [作業系統概念和文藝復興]，簡稱 OSCAR，資料彙整於 hackfoldr))

破題

標題的 "Microkernel" 恐怕是最有爭議的部份，因為不單單是作業系統有此詞彙，在許多軟體工程或架構規劃中，不乏有此用法，比方說 Java EE 與 Enterprise Computing 領域中赫赫有名的 JBoss，即有 "JBoss Microkernel" (JMX) 的術語，用以描述在 JBossMX 上運作一系列 MBeans 的設計。



不過本講座則是專注於作業系統領域，同時，"Microkernel" 也不全然指其 "micro" 微小之意，而且是探討相對於傳統 Monolithic kernel 的 Microkernel，後面的段落會有更詳細的解釋。

自幹「靠北 Kernel」

- **microkernel.info** 列出這個世界中開放原始碼的微核心專案，其中包含由成功大學師生合作開發的 F9 microkernel

F9

An experimental microkernel used to construct flexible real-time and embedded systems for ARM Cortex-M series microprocessors with power efficiency and security in mind. (*github.com/f9micro*)

- Construct an Efficient and Secure Microkernel for IoT

無所不在的 Microkernel

2015 年，洪士灝 教授補充了 microkernel 和 network security enhanced hypervisor 的設計考量 [source]，不過我們要特別留意到，iPhone 裡面不是只有一個 iOS (其核心名為 XNU，而由 XNU 核心所構成的作業系統家族稱為 Darwin，部份原始程式碼在 opensource.apple.com 網站可取得)，從 iPhone 5S 以後，採用 Apple A7 處理器，就伴隨著 Secure Enclave coprocessor，後者裡面運作 L4 microkernel (!)

以 2015 年 9 月 Apple Inc. 發布的 iOS Security 白皮書 第 7 頁提到:
"""
The Secure Enclave uses encrypted memory and includes a hardware random number generator. Its microkernel is based on the L4 family, with modifications by Apple. Communication between the Secure Enclave and the application processor is isolated to an interrupt-driven mailbox and shared memory data buffers.
"""
這個 Secure Enclave 就是搭配 Touch ID 這個數位資料保護機制，其設計非常複雜，由硬體亂數產生器和特製的 L4 microkernel 構成，又根據 NICTA/UNSW 表示，Apple 特製的 L4 就是以 NICTA 的 Darbat 專案 (又稱 L4/Darwin) 為基礎，也就是將 Darwin 作業系統移植於 L4 microkermel 之上。
(延伸閱讀: A More Secure Architecture for Mobile)

無獨有偶，Samsung Galaxy 手機則搭配稱為 Knox mobile security and management 解決方案，後者

採用 ARM TrustZone 為基礎的技術，來提昇安全性。

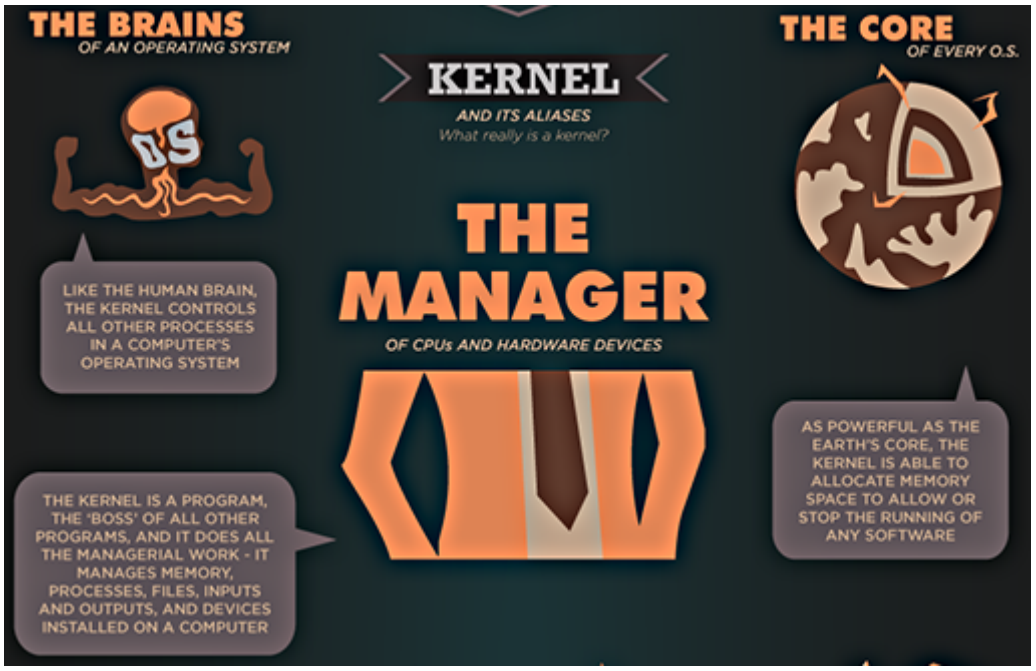
澳洲新南威爾斯大學教授、ACM Fellow、前 OK Labs 技術長 Gernot Heiser 博士在 Linux.conf.au 2015 以 "seL4 Is Free – What Does This Mean For You?" 為題進行演說，提及 seL4 不只是個「一行原始程式碼價值美金一千元」的作業系統核心，事實上，seL4 作為 L4 microkernel 家族最前瞻、充分商業驗證的研究計畫，早已在全世界大量的裝置內現身，全球有超過 15 億的裝置運作 L4 microkernel (!)

不管是物聯網，還是雲端運算，我們都可以看出一個顯然的事實，唯有更高附加價值並透過作業系統提昇規模、深度、廣度，還有彈性，才有機會創造正向循環。

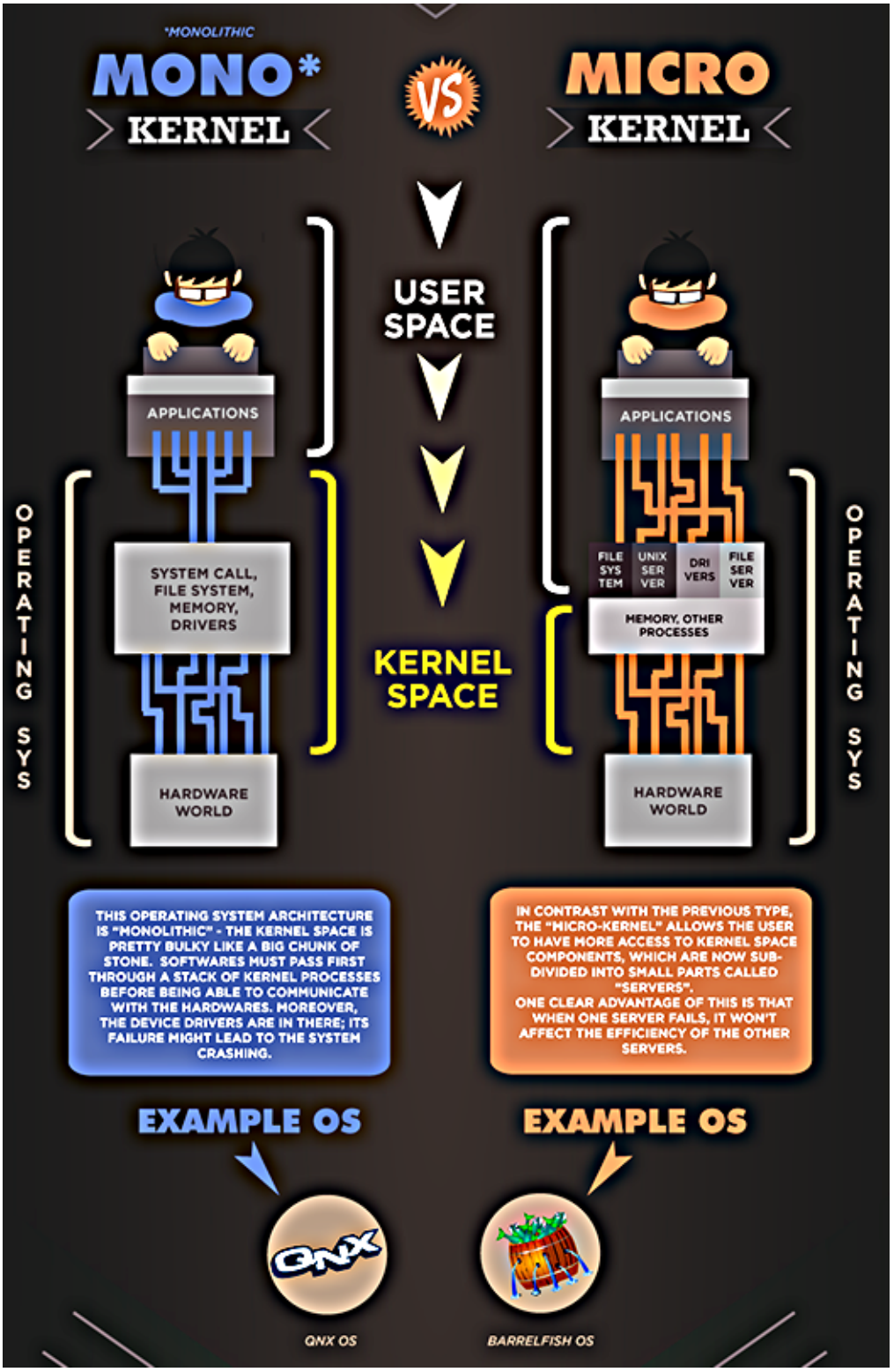
圖解作業系統核心設計

[source: Kernel vs Microkernel vs Multikernel – Explained Visually]

- [ ] 作業系統核心扮演的角色



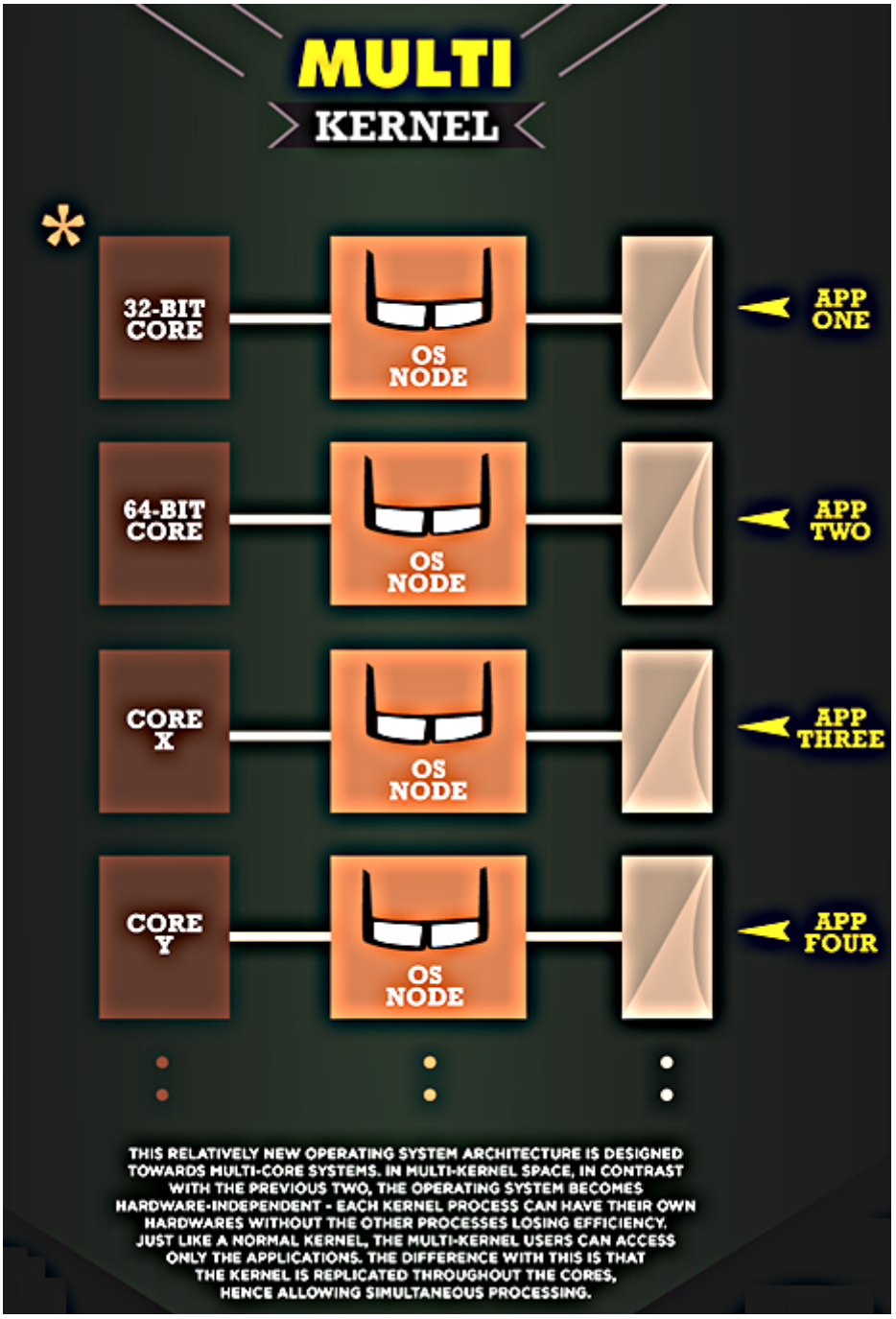
- [ ] monolithic kernel 和 microkernel



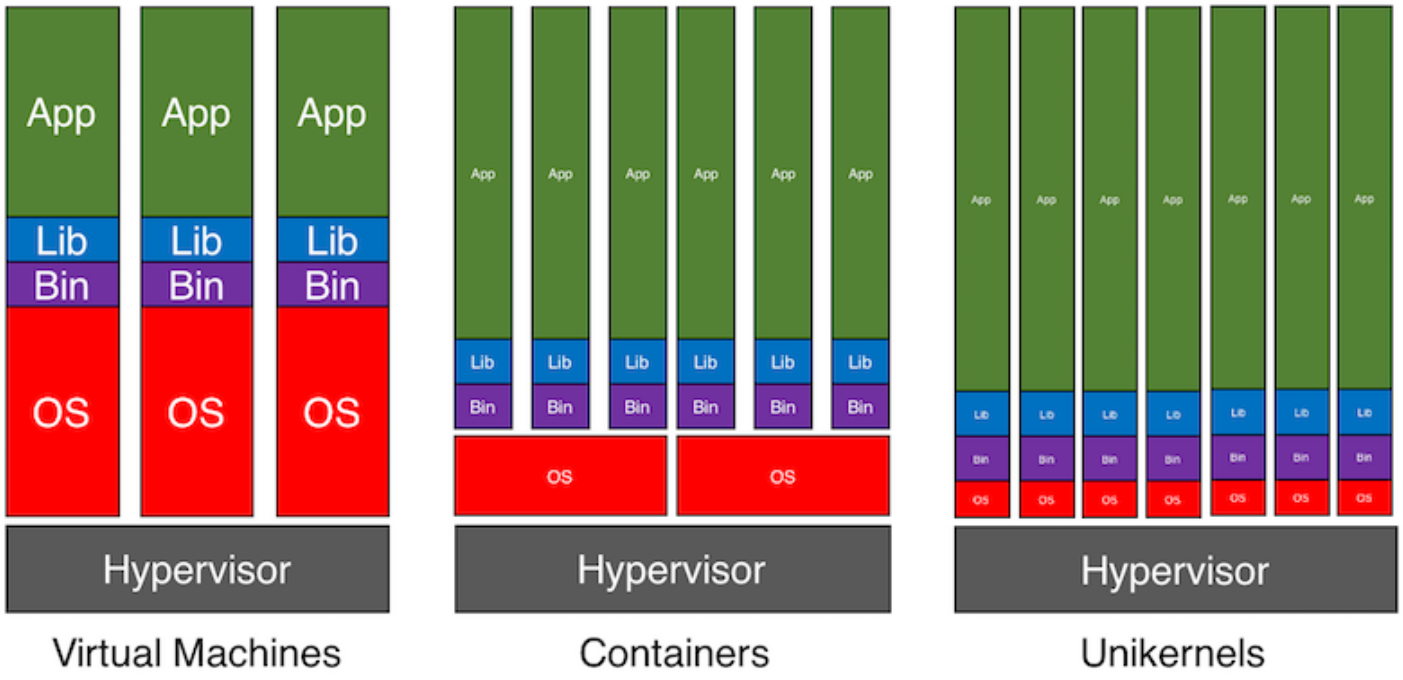
- QNX
  - QNX 2016 Technology Concept Vehicle and Reference Vehicle: Paving Way to Autonomous (YouTube)
  - 黑莓手機 (BlackBerry) 內建的作業系統是 QNX。扣除智慧型手機領域，QNX 廣泛地執行在許多領域，舉凡車載系統、核電站、軍用無人駕駛機、軍用無人駕駛坦克等等，包含台北捷運車廂廣播系統。 [video]
  - 1980 年，Gordon Bell 與 Dan Dodge 這兩位加拿大滑鐵盧大學 (University Ave W, Waterloo) 的學生，以電腦科學的概念 (當時缺乏實做訓練)，再加上一些設想而撰寫出一個即時作業系統核心，他們認為這個作業系統核心能滿足商業需求，就成立 Quantum Software Systems，並於 1982 年發表了第一個版本的 QNX，最早的版本稱為 Quick UNIX，一直到 AT&T 公司的律師來函，認為 UNIX 一詞侵犯到 AT&T 的商標權，才將更名為 QNX。當時主要支援的硬體平台是 16

位元的 Intel 8088 處理器。
    - 1990 年代末，QNX 開始進行全新的改版，新版能支援 SMP 架構，同時支援所有現有的 POSIX API，但 QNX 依然保有其 microkernel 特性。此一改版於 2001 年正式完成，稱為 QNX Neutrino。
    - QNX 的核心只包含以下：
        - 處理器排程
        - 行程間通訊
        - 中斷轉向 (redirection): IRQ handler 執行於 userspace
        - 計時器
    - 除這些外其餘的程式一律以 user process 的型態來執行，包括一個專門用來產生 process 的特別 proces：proc 也是如此，另外記憶管理的機制運作也是以 user process 的方式與微核心接通。
    - QNX 在車用市場佔有率達到 75%，目前全球有超過 230 種車型使用QNX 系統，全球有超過兩千萬輛汽車已裝配了 QNX 的授權軟體，其中包括全數位儀表板、藍牙免持系統、多媒體娛樂中控系統、車載聯網模塊和 3D 導航系統。QNX 還提供業界最大、最廣泛且通過量產驗證的汽車資訊娛樂生態系統。
- 到了 21 世紀，純正的 microkernel 架構的作業系統已經很罕見，Microsoft Windows 和 Mac OS X 就實做層面來說，都不算是 microkernel，頂多歸類於 "hybrid kernel"，依舊堅持 microkernel 路線的作業系統剩下 QNX, L4 家族，以及 GNU Hurd。VxWorks , ThreadX, Zephyr RTOS, ...

- [ ] Multi-kernel


MULTI
KERNEL
32-BIT CORE
64-BIT CORE
CORE X
CORE Y
OS NODE
OS NODE
OS NODE
OS NODE
APP ONE
APP TWO
APP THREE
APP FOUR
THIS RELATIVELY NEW OPERATING SYSTEM ARCHITECTURE IS DESIGNED TOWARDS MULTI-CORE SYSTEMS. IN MULTI-KERNEL SPACE, IN CONTRAST WITH THE PREVIOUS TWO, THE OPERATING SYSTEM BECOMES HARDWARE-INDEPENDENT - EACH KERNEL PROCESS CAN HAVE THEIR OWN HARDWARES WITHOUT THE OTHER PROCESSES LOSING EFFICIENCY. JUST LIKE A NORMAL KERNEL, THE MULTI-KERNEL USERS CAN ACCESS ONLY THE APPLICATIONS. THE DIFFERENCE WITH THIS IS THAT THE KERNEL IS REPLICATED THROUGHOUT THE CORES, HENCE ALLOWING SIMULTANEOUS PROCESSING.

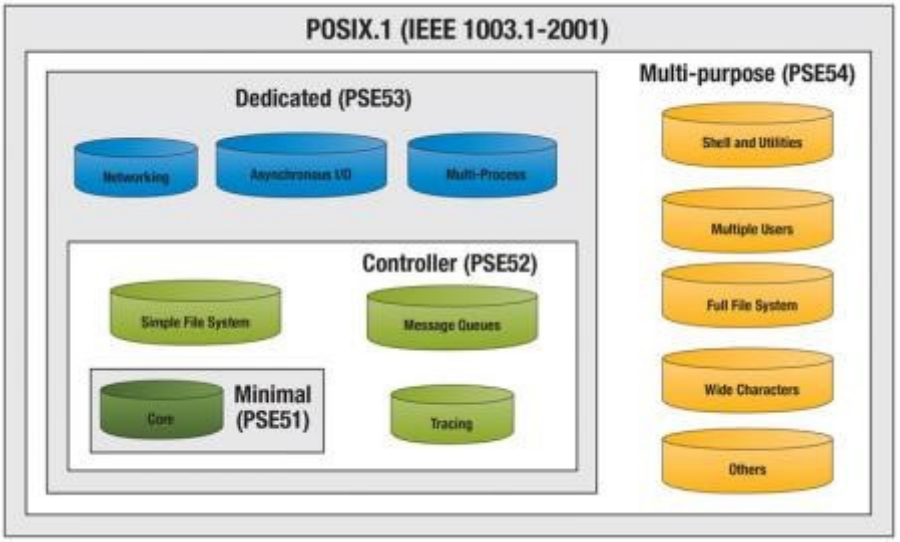
- Unikernel



- library OS
  - Unikernel (part of Docker Inc.)
  - MIT Exokernel

POSIX

- 時代背景
  - 交大資工學生製作的短片〈USL v BSDi 官司〉
  - 從 Revolution OS 看作業系統生態變化
- POSIX – 25 Years of Open Standard APIs
  - 用圖表說明了 PSE51, PSE52, PSE53, PSE54，以及用於即時處理應用的 IEEE 1003.13-2003 (POSIX.13) Profiles



1986 年，IEEE 指定了一個委員會制定了一個開放作業系統的標準，稱為 POSIX (Portable Operating Systems Interface)，最後加上個 "X" 是因為本質上是 UNIX 的標準。

源自 UNIX 的諸多「變種」

雖然 UNIX 不是最早的作業系統，但絕對是作業系統設計的典範，特別是其多種演化版本與重新實做的計畫，影響絕大部分通用作業系統的設計。

1960 年代末期，Bell Telephone Laboratories (Bell Labs), General Electric 和 Massachusetts Institude of Technology (MIT) 合作研發一個 multi-user, multi-tasking 的作業系統：Multics (CTSS 之後的「第二系統」，在《人月神話》提及)，Bell Labs 隨後因為 Multics 進度嚴重落後，於 1969 年三月退出該計畫。

- Ken Thompson 在 DEC PDP-7 上寫了名為 Space Travel 的電玩遊戲
- PDP-7 欠缺程式發展環境，所以 Ken Thompson 與 Dennis Ritchie 開發 UNIX
- Ken Thompson (ken) 寫了 B 語言(由 BCPL 演化而來的直譯語言)
- Dennis Ritchie (dmr) 把 B 改成了著名的 C 語言.
- 1973 年 11 月 Unix version 4，使用 C 語言改寫
- Unix 的第一篇論文 "The UNIX Time Sharing System"，由 Ken Thompson 和 Dennis Ritchie 在 1973 年 10 月 the ACM Symposium on OS (SOSP) 中提出
- 論文在 1974 年 7 月的 the Communications of the ACM 發表，這是 UNIX 與外界的首次曝光

UNIX 得以廣泛流通的時代背景

- 1956 年，AT&T 受到反托拉斯法調查，調查期間 AT&T 與聯邦政府簽訂了一個協議：不得經營與電話電報無關之業務，而 Bell Labs 隸屬於 AT&T，自然同樣受到約束
- UNIX 在 SOSP 發表後，學術界對 UNIX 及其原始碼索求不斷，所以 AT&T 便免費提供原始碼給學術界，此舉造成了 UNIX 的廣泛流傳

加州大學 Berkeley 分校的 Computer Science Research Group (CSRG) 對 UNIX 的發展做了很多的貢獻，Berkeley 版本的 UNIX 稱為 BSD UNIX，主要貢獻包含：

- virtual memory
- TCP/IP
- Fast File System (FFS)
- reliable signals
- socket介面

4.4BSD 把原來的 virtual machine 換成 Mach microkernel 的實做，並引進了 Logged File System. (LFS)。但 CSRG 做完 BSD4.4 後，就關門大吉。

反拖拉司法調查結束後，AT&T 拆成數個子公司，Bell Labs 更名為 AT&T Bell Laboratories，並且 AT&T 允許進入電腦市場。AT&T 發表的商業版 UNIX 計有

- System III
- System V
- System V Release 2 (SVR2)
- System V Release 3
- System V Release 4/4.2

System V 引進了許多新的特色 (相較舊的 UNIX)，如 regions 架構的虛擬記憶體 (和 BSD 的作法不同), IPC, remote file sharing, shared libraries, STREAMS 架構等等。

1980 年代，AT&T 廣泛授權 UNIX 給不同的廠商，包含開發大同電鍋的大同公司也有自己的 UNIX，稱為 MITUX。商業化的 UNIX 為 UNIX 爭添不少特色，像是：

- SunOS 的 Network File System (NFS), vnode/vfs interface 支援多重檔案系統、新的 VM 架構 (為 SVR4 所採用)
- AIX是第一個支援 journaling file system 的商業UNIX.
- Ultrix (DEC 的 UNIX) 是支援multiprocessor UNIX的先趨之一

- The BSD family of operating systems
  - slides
- Berkeley CSRG (1977 - 1995)
- 4.3BSD UNIX Operating System
- NetBSD
  - IoT visionaries by targeting a Toaster
  - VAX
  - ATF
  - Unprivileged builds: `build.sh`
- FreeBSD
  - RISC-V
  - ARMv8
- OpenBSD
  - libc
  - LibreSSL
  - PF
  - CARP
- DragonFly BSD
  - HAMMER
  - CPU Scaling
  - 將 microkernel 概念實做於 monolithic kernel (BSD) 中！
  - hybrid kernel

回顧 Microkernel 發展

- Microkernels: The veterans of OS design
  - 1985 年至今 30 年間，microkernel 的演化和對產業的衝擊
- Microkernel Evolution (2013)

microkernel 不是新的概念，這個名詞至少在 1970 年代初期就有。一般認為，microkernel 源自 Brinch Hansen 在 1969 年提出的 RC 4000 multiprogramming system，而更早之前就在電腦系統應用此概念

了。

Mach microkernel

Mach (發音 [mʌk]) 是美國 Carnegie-Mellon 大學 (CMU) 的 microkernel 作業系統，發展於 1980 年代，著眼點是，隨著功能越來越多，UNIX 也日漸龐大複雜而難以掌握，Mach 的設計概念就是「去蕪存菁」，僅留下真正關鍵的部份，其餘的功能都用使用者層級 (user-level) 的程式 (特徵 server) 來實做，藉此減低核心的複雜度。

Mach 設計目標：

- 與 UNIX 相容
- 物件導向設計
- 跨平台：在單處理器、多處理器上都能執行
- 適合分散式運算環境

值得一提的是，儘管 Mach 已式微，但 Mach 的眾多技術突破陸續被 BSD 和 Linux kernel 所吸收。Mach 2.5 是許多商業 UNIX，像是 DEC OSF/1, NeXTSTEP (後來移轉到 Apple Computer) 的基礎，Mach 3.0 才是真正純粹的完全 microkernel 化的實做，而 Mach 4.0 則由 Utah (猶他) 大學維護。

Mach 的主要開發者 Richard Rashid 自 1991 年就在 Microsoft 服務，領導 Microsoft Research 若干技術突破，另一位主要 Mach 開發者 Avie Tevanian 曾在 NeXT 擔任軟體主管，並在 Apple 收購 NeXT 後，成為 Apple Inc. 的技術長。

Mach 被視為以下這些元件所組成:

- ports (埠)
- messages (訊息)
- tasks (工作)
- threads (執行緒)
- virtual memory (虛擬記憶體)

如同一個設計成熟的物件導向系統，這些物件的介面已經定義明確，因此物件內的改變不會影響到使用這些物件的行程 (process)。

可將 Mach 中的 task 看待為 UNIX 環境中的 process。 對執行中的行程而言，它是一個可執行的環境，如同虛擬記憶體或是處理器的執行週期。不過，相對於傳統 UNIX 的 process，Mach 的 task 並不表示包含一個正在執行的 thread，對 Mach 而言，thread 也是一個獨立的物件 (!)

=> Android: task, thread, activity, service (microkernel terms)

=> Android: binder IPC/RPC

因此，一個有用的 task 必須包含至少一個 thread。Mach 的 thread 與其它常見作業系統的 thread 相仿，在同一個 task 中的 thread 相互分享記憶體與其它資源。Mach 在設計時，就希望成為一個 multi-threaded，而可有效執行於多顆處理器 (SMP) 上。

相較之下，Linux 發展之初，只是一個以 single thread 為導向的作業系統，multi-threaded 與 SMP 也在發展 10 年後才納入，早期甚至得用彆腳的 LinuxThread 套件來實現 multi-threading，而 Mach 與

Hurd 在設計初期，就已經考慮這些需求。在 NPTL 出現之前，Linux 的 multi-threaded 實做非常奇怪，仍然把 process 當作最基本的abstraction，也就是說 scheduling, context switch 等基本操作對象仍是 process，而thread / LWP 只是和別人分享定址空間和資源的 process。因此：

- 用 clone() 產生的 thread 本質上仍是 process，可以說 Linux 核心中定義的 "thread" 只做到了資源共享，而不構成執行工作的最基本單位
- 嚴格來說，Linux 只實做了一半的 thread，但這並不是壞事，因為許多的應用程式不見得用到 thread，而且簡化 thread 實做的結果，使得 process 管理變得更有效率，副作用是產生出來的 "thread" 比其它作業系統的實做，顯得更 heavy-weight，可以說，過去 Linux 犧牲 thread 的效率，以換取 process 的效率
- 以 abstraction 的角度來看 Linux 過去並非在本質上支援 thread，但以 programming model 來看，Linux 的確是有 thread 可用，儘管效率較差

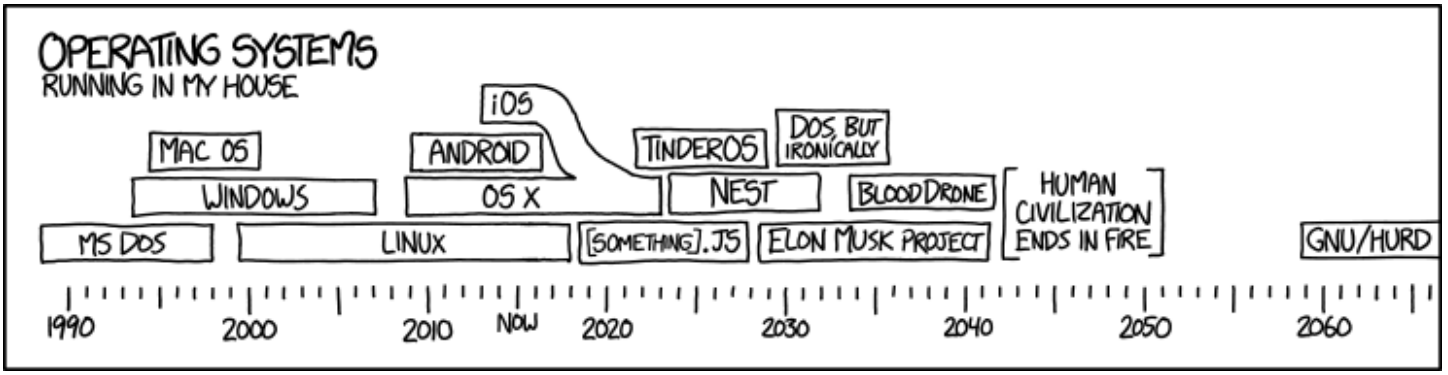
早期 Linux 的 process 和 thread 的效能和其他作業系統的客觀數據比較，可參照
論文 "An Overview of the Singularity Project" (Microsoft Research, 2005 年) 的第 31 頁

Task 在 Mach 中最主要的溝通方式是使用 port。Mach 的 port 有點像是 socket 中的 port，兩個 task 間的一條溝通管道。如同 socket 環境下的 port，Mach 的 port 被設計在兩個不同的 thread 執行於同一個處理器、不同的處理器，或是網絡上不同的電腦時的通訊動作。

因為 port 主要需求在於兩個 Mach 應用程式間內部的通訊，所以這個設計可以輕易地擴展到多顆處理器。同樣的，這些資料的內容在傳送前必須形成 message (訊息)，message 是 Mach 內的一種物件。最後，Mach 提供 memory- management servers (記憶體管理服務)，其目地在各 task 間分享記憶體。

GNU Hurd

你想過一個作業系統發展了整整 30 年，才開始支援音效裝置嗎？有，這樣的作業系統就是 GNU Hurd。由於 Hurd 的長期延宕，幾乎成為人類歷史的傳奇作業系統，xkcd 甚至有則漫畫來諷刺這件事。



1984 年 1 月，Richard Stallman (以下簡稱 RMS) 從 MIT 人工智慧實驗室辭職，開始寫 GNU 軟體。離開 MIT 是必要的，這樣 MIT 就不能干涉 GNU 的軟體發布方式。因為如果 RMS 還屬於那裡的職員，MIT 就可要求這些軟體的擁有權、或是施加額外的發行條款。隔年，RMS 成立 Free Software Foundation (FSF，中譯為「自由軟體基金會」) 以便全力投入 Project GNU 的工作，並成為 FSF 的終身義工。

RMS 不單自己一個人在叫著免費的軟體的行動，他已掀出一陣風，讓許多人都支持他、為他撰寫免費

軟體並提供幫助。GNU 是 Gnu's Not Unix 的縮寫，在 GNU Manifesto 中，RMS 提到 UNIX 雖不是最好的作業系統，但是至少不會太差，而他自信有能力把 UNIX 不足之處補全。於是，RMS 著手寫一套優秀並能與 UNIX 相容的作業系統。此外，這套作業系統開放給每個人免費傳播使用，這就是名為 GNU 的作業系統。

因為確定 GNU 會是一個與 UNIX 相容的系統，而 UNIX 下的標準介面都已經算完備，也就是前述的 POSIX，所以，只要照著標準完成的程式，將來 GNU 的 Kernel 完成後，立刻可相容並執行所有 UNIX 的公用程式，因此會先有 emacs、gcc 等工具程式出來。其中 gcc 更使得 Project GNU 能不依靠 non-Free Software 來產生 / 編譯自己。

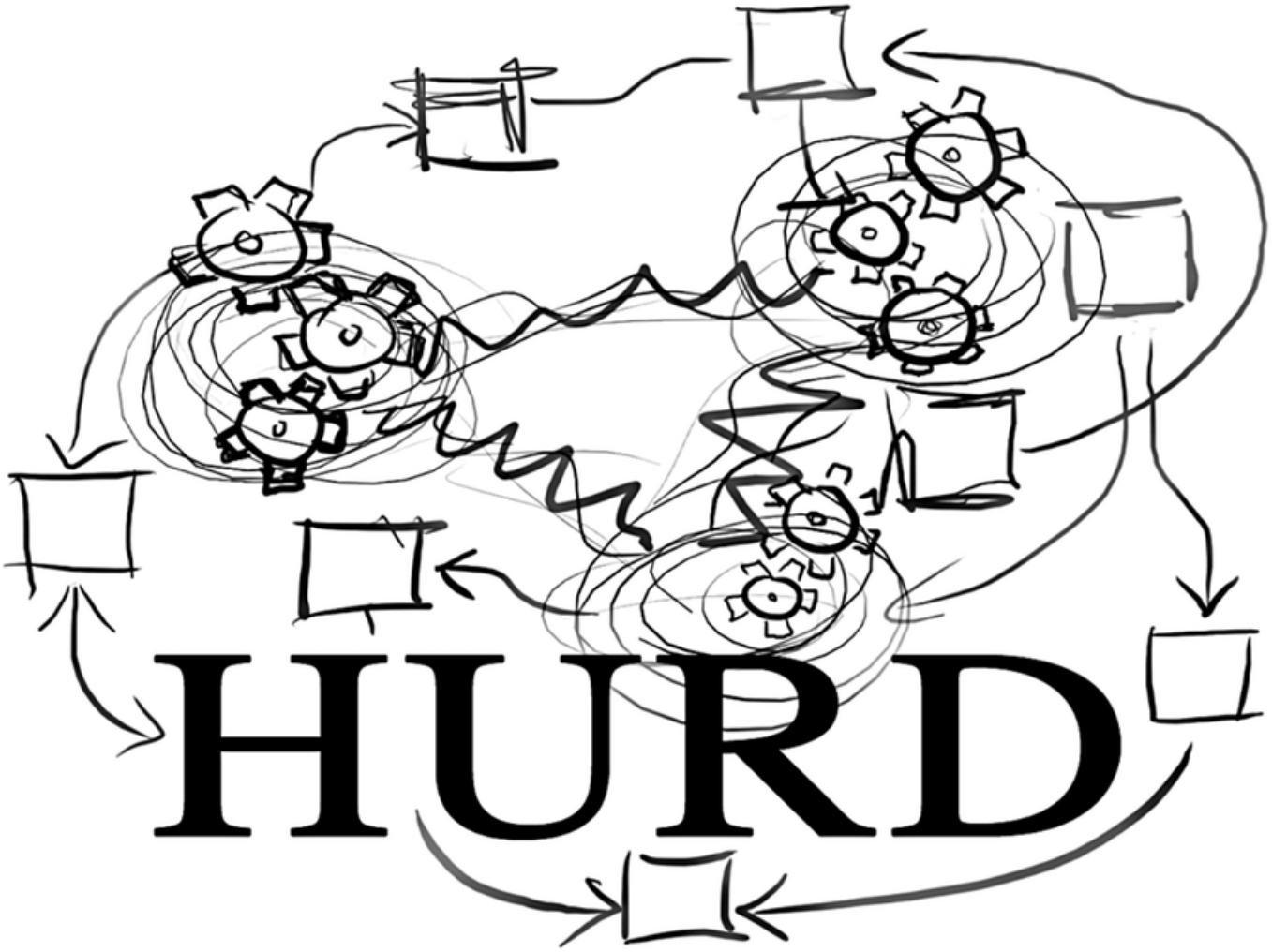
如今程式發展工具成熟了，而屏息以待的，剩下 GNU 作業系統的核心，也就是 Hurd。一旦 Hurd 完成，GNU 就是一套能夠自我開機、真正自給自足、完整的作業系統。就在 Hurd 著手進行的幾個月後，當時仍是芬蘭赫爾辛基大學生的 Linus Torvalds 在 comp.os.minix 新聞群組上發布得以運作於 Intel 386/486 電腦的小型作業系統 Linux，並開宗明義說「只是業餘作品，不像 GNU 那樣龐大且專業」。

從使用者角度來說，Hurd 需要漫長的等待，Linux 開發者於是採用 Linux 填補 GNU 作業系統的規劃藍圖裡頭，本應由 Hurd 存在的位置。Linux 依賴 GCC 和 GNU 工具，其知名度隨眾多散佈套件出現而提高，自由軟體基金會開始將 Linux 視為 GNU 作業系統的核心替代品。Hurd 的開發雖然持續進行，但顯然自由軟體基金會已經選擇了不同的方向。

RMS 之後也承認，開發基於 Mach 的 GNU 核心的技術決定完全是他的責任，這個決定似乎延緩了開發工作。他認為使用 Mach 可以節省許多工作而加速開發，但他錯了。GNU Hurd 預計在 1994 年問世，並計劃於 2001 年釋出正式版本 (v1.0)，但這些承諾從未履行。

GNU 核心一開始並不叫做 Hurd。據 RMS 的說法，最初稱為 Alix，以當時他的情人 Alice 來命名，符合 UNIX 系統的命名慣例，後方加上 "X" 字母 (最類似的命名是 IBM 自己的 UNIX，稱為 AIX)。

之後 Hurd 主要開發者 Michael (後來更名為 Thomas) Bushnell 偏好 "Hurd" 這個名字，Alix 被改以指稱核心中攔截系統呼叫，並發送訊息至 Hurd 背景程式加以處理的部分。之後 Stallman 跟 Alice 分手，她也改了名字。恰巧的是，Hurd 設計也有所更動，C 程式庫會直接發送訊息至伺服器，致使 "Alix" 消失在設計中。



Bushnell 選擇 Hurd 這個名字，部分因為 Hurd 念起來像 Herd，有一群 GNU 的意思，部分因為 Hurd 是 'Hird of Unix-Replacing Daemons' 的遞迴首字母縮寫，而 Hird 是 'Hurd of Interfaces Representing Depth' 的遞迴首字母縮寫。如 Bushnell 所說，這是第一套以成對相互遞迴首字母縮寫命名的軟體。

Hurd 早期主要架構師之一的 Thomas Bushnell，在 1996 年撰寫的論文 "Towards a New Strategy of OS design" 總結：GNU Hurd 的設計讓系統程式碼區域盡可能受限，程式只能與核心少數基本部分溝通，系統其餘部分可動態更換。

Mach 微核心有許多特性，但這些對 Hurd 而言都不重要。Hurd 只須要微核心提供幾個基本而簡單的功能

- 排程
- task 建立與刪除
- IPC
- 記憶體管理

換句話說，理論上其它的微核心只要提供這些基本功能，就可以取代 Mach，所以 Hurd 有很高的移植性，過去也曾移植到 L4 microkernel 上，但後來仍是回歸 Mach 這個微核心 (這個特化的版本稱為 GNU Mach)。

除了微核心外，Hurd 另一個讓人感到興趣的技術就是 multi-server 的設計。這些環繞在微核心旁的

server 都執行於使用者模式，並提供應用程式所需要的服務。由這些 server 與其特性看來，可清楚瞭解 Hurd 整個設計。

Authentication server 在不同 task 間通訊時，提供使用者身份驗證。一個 task 可以透過 port 連到 authentication server 使用其所提供的服務，並利用這些服務來辨識其它的 task。因此 authentication server 提供 Hurd 基本的安全機制。值得注意的是，這個 server 沒有什麼特別，它與其它 Hurd 的 server 一樣，都是在 user-space 執行。因此，任何人可以設計並執行自己的 authentication server; 其如果它使用者信任這個 server，那他們可以自由的使用它，如果他們不信任這個 server，那可以不使用它。

Process server 基本上扮演一個橋樑，藉於以 UNIX 為概念的 process 及以 Mach 為概念的 task。遵循 POSIX 標準的 UNIX 有許多概念無法使用於 Mach，這些缺少的功能就由 process server 來補足。

此外，process server 提供了整個系統所有 process 的資訊 -- task 可在 process server 中註冊，如此其它的 task (例如 UNIX 下的 process server 行程) 就可以獲得系統中其它 task 的資訊。但是，這動作不是強制的，process 可以選擇不在 process server 註冊，因此其它 process 也就無法獲得這個 process 的資訊。

Hurd 還包括許多其它 server，像是以 socket 為通訊基礎的 server，NFS server 等等。這些 server 與 Hurd 一樣還在發展中。記得一點，使用者可以隨心所欲的加上他的 server，這個動作不須要特別的權限。

除了 server，Hurd 另一個重要的概念就是所謂的 translator。您可以將 translator 視為 UNIX 檔案系統的機制，諸如 mount points、devices 還有 symbolic links。

在 Hurd 檔案系統中，每個檔案都被連結到一個叫 "translator" 的程式。當使用者程式存取這個檔案時，控制權就轉交給 translator 來處理。如果存取動作是讀取 disk 中真正的檔案，則這種檔案系統格式的 translator 會向 disk 要求資料，並傳回應用程序。如果存取的是 device，則這個 device 的 driver 會處理。同樣的，在存取 symbolic link 時，translator 會直接導向到指定的位置，或是其它的檔案架構。

融合到 Mac OS X 的 Mach

Apple Inc. 現在許多技術來自 Steve Jobs 於 1985 年離開 Apple Computer 後，所創立的 NeXT 公司，後者的主力產品就是 NeXTSTEP 作業系統，以 CMU Mach 為基礎，並且整合 4.3BSD userspace 作為 Mach 上的 server。

後來 Apple Inc. 把 NeXTSTEP 的技術發揚光大，演化為 XNU (核心) / Darwin (作業系統)，造就 iPhone OS / Mac OS X 的關鍵技術。
技術簡報: The Microkernel Mach Under NeXTSTEP
延伸閱讀: Inside the Mac OS X Kernel (錄影)

邁向第二代 Microkernel

以設計的細緻度來看，microkernel 優於大型整合核心 (large-integrated kernel) 之處：

1. 清楚的微核心介面將會使系統結構更為模組化
2. 方便使用者程式的除錯
3. 系統將更有彈性及容易維護。

雖然微核心在觀念上優於大型整合核心，但經過測量及比較的結果，一些著名的微核心，如 Mach, Chorus，其執行效率卻比大型整合核心差很多。第二代的微核心則針對效率的問題，做了許多改良，捨棄傳統上把硬體資源視為抽象的物件、加以包裝的觀念，使得整體的效能平均可以比第一代的微核心好上 10 倍以上。

Monolithic Kernel 最主要的優勢為執行速度。而 Microkernel 則是 stability 與 security。

- 典型的 microkernel 負責的事情很少，只有低階 process 管理與 scheduling, IPC, interrupt, 基本的記憶體管理，以及少數其他東西。其他的 service，如 file system, device driver 等等，各自跑在不同的 process，透過 message pipe 與kernel 或其他 service 溝通。
- Monolithic Kernel 的作法完全相反，除了 user application 之外的程式，幾乎都跑在 kernel space (同一個process)。

microkernel 設計不可避免會變得龐大臃腫，因為核心的確變小，但是被剔除到核心之外的各部分互相通信的開銷則是增加了很多，所以過去的 microkernel 速度不會太快。QNX 克服了這個缺點的方式：

- Unix 使用同步系統呼叫。Mach (Gnu Hurd 的 microkernel) 和 minix 依循這設計
- QNX 使用非同步的系統呼叫

第二代 microkernel 以 L4 microkernel 為代表，以「最小化原則」為基礎。為什麼要小。Liedtke 在論文〈u-kenrels Must and Can Be Small〉解釋：

- 更小的核心使用的 cache 自然也小，由於 cache miss 而帶來的開銷自然也小
- 因此，只有 footprint 非常小的 microkernel 才可能有高效能

接著面臨設計的抉擇：系統個別功能該實做於核心模式，還是使用者模式呢？

- L4 microkernel 提供的機制是最被頻繁呼叫的程式碼和資料，大幅提昇 cache hit ，是 L4 高效率的主因
- 第一代 microkernel 之所以慢，是因為核心工作太多，而且設計到大量複雜的系統呼叫，以 Mach 來說，許多系統呼叫是非同步，但沒有充分調整，以至於系統頻繁在核心和使用者模式間切換，更麻煩的是，大量的 IPC 讓這件事變得更緩慢
- 1996 年，以效能為首要考量的第二代微核心 L4，還無法將自身置入當時 Intel Pentium 架構的 8K I-cache (L1)。整整 20 年後，ARM Cortex-A57 的 L1 I-cache 就達到 48K，於此同時，L4 實作更加精簡，可望完全運作於 I-cache (!)

- L4 Microkernels: The Lessons from 20 Years of Research and Deployment (論文)

NICTA / UNSW / OKL4 / seL4

NICTA (澳大利亞的產學合作加速器，類似台灣的工研院，但是更聚焦在資訊科技) 最成功的案例就是 Open Kernel Labs (OK Labs)，2012 年賣給美國國防部的主要軍火供應商 General Dynamics。之後

NICTA 持續和 UNSW (澳洲新南威爾斯大學), AU (澳洲大學) 等大學的研究人員開發 seL4 微核心，於是世界上最先進的微核心研究不在美國，也不在歐洲，而是在澳洲。

澳洲沒有半導體產業，沒有垂直整合的 ICT，沒關係，UNSW 專心致力於作業系統技術，在 21 世紀中以作業系統技術為主，建立了 OKLabs 這個新創公司，我們幾乎看不到第二間標榜產品就是作業系統的公司，但這群來自 UNSW 的研究人員作到了，而且被成功收購後，還發揚光大，持續影響整個產業。

NICTA/UNSW 獲得美國國防部 Defense Advanced Research Projects Agency (DARPA) 的資助，將高可靠、安全的 seL4 應用在飛行載具中。具體的應用情境已公佈，對於波音公司出品的 MH-6 Little Bird 直昇機和它的攻擊型本 AH-6，是美國陸軍用於特種作戰的單引擎輕型直昇機，隨著作戰計畫的複雜化，美國軍方正視了網路攻擊的嚴酷挑戰，因此採納了 seL4 這個號稱數學上證明過的 "Unhackable Kernel"，目前 seL4 用於 High-Assurance Cyber Military Systems (HACMS) 中，已完成測試。

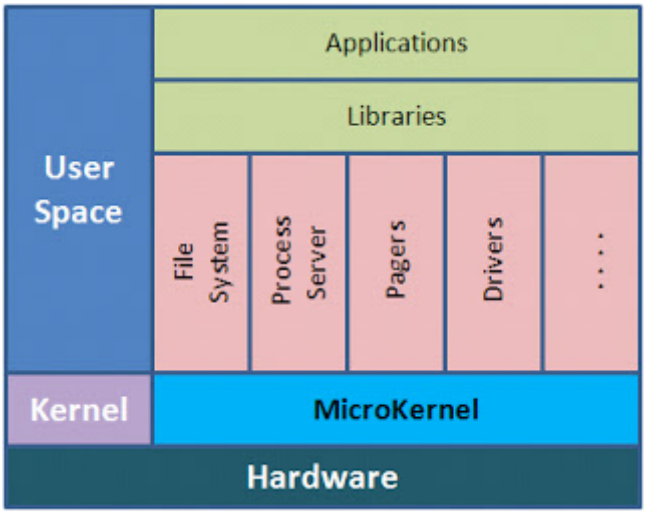
(video) Boeing helps DARPA develop more cybersecure military systems

seL4 受到 DARPA 贊助並已用於波音公司開發的 Unmanned Little Bird (ULB) 戰鬥直昇機，這份影片展示建構在 eChronos RTOS 和 seL4 的高度安全飛行控制系統，Mission board 由 ARM Cortex-A15 硬體組成，執行 seL4 + 虛擬化的 Linux，而 Flight controller 由 ARM Cortex-M4F 硬體組成，運作 eChronos RTOS，詳情可見:http://ssrg.nicta.com.au/projects/TS/SMACCM/

後來 NICTA / UNSW 有一群人再次出來創業，成立 Cog Systems (光看名字就知道很 geek!)，將 OKL4 應用於高度安全需求的領域。Cog System 宣佈將會推出發揮 ARMv8 64-bit Trusted Extensions 的 Trusted OS (TEE) 系統出來。
這是繼 seL4 後，另一個值得關注的技術。
Cog Secure: http://cog.systems/products/cog-secure-env.shtml
[YouTube: vGPU on HTC M8]

Muen Separation Kernels

[source]



Muen separation kernel 運用 Intel VMX 技術，在原本的作業系統底下建立一層抽象執行單元，於是可運作 Linux 或其他的作業系統，在Muen 核心之上，這樣能做什麼呢？要記住，Muen 核心原始程式碼才 5000 行，而且設計上經過數學驗證 "Proof of absence of runtime errors"，這樣的 trusted computing base (TCB) 是我們能夠用有限的工程資源去確保安全性，而 Linux 儘管具備強大的功能，卻無法通過各式理論和實務的檢驗。

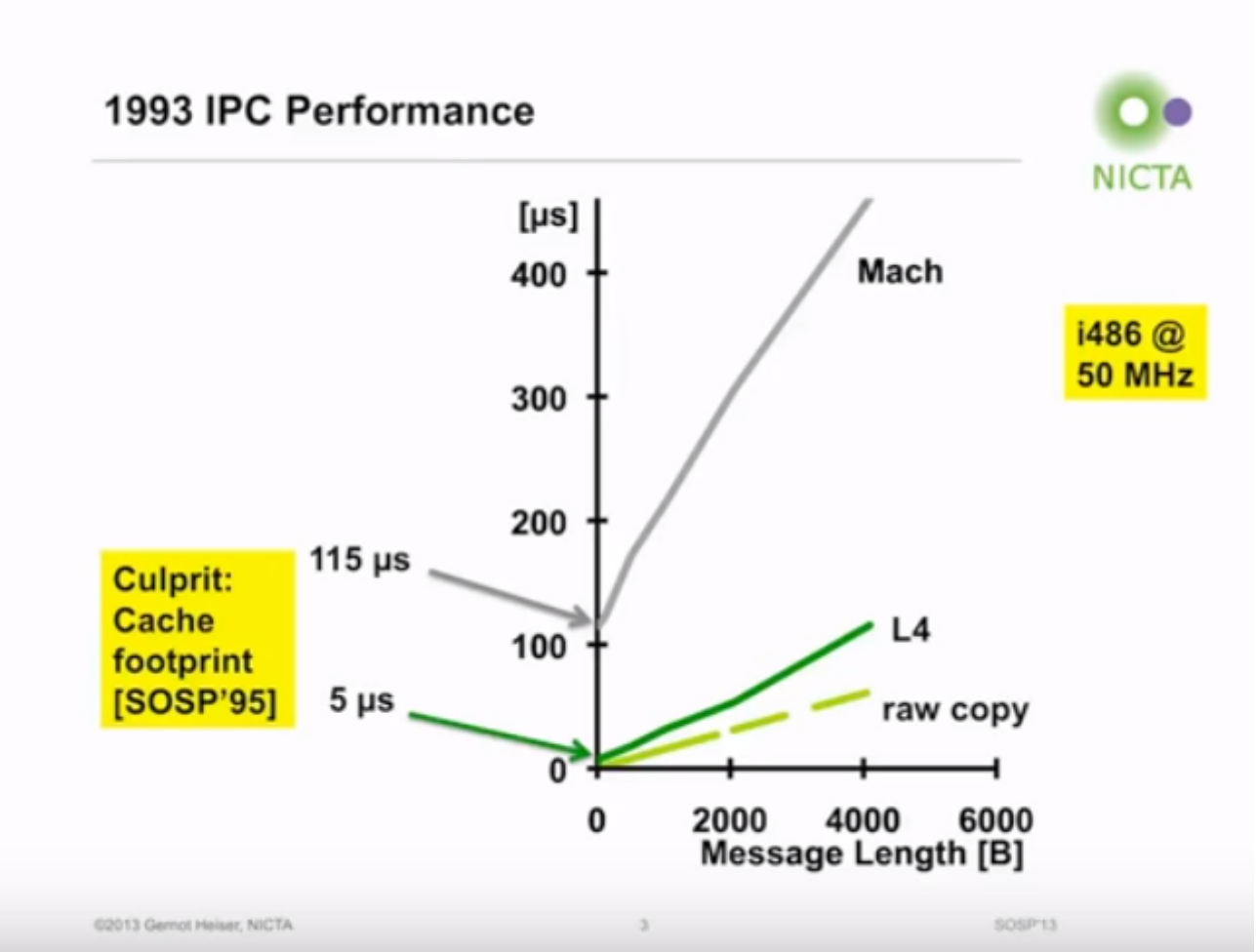
當 Muen 應用於雲端運算時，可搭配 Network Function Virtualization (NFV) 和 Software Defined Networks (SDN)，提昇更高的佈署和設定彈性，其上的通用作業系統如 Linux，可幾乎不用作什麼修改。

看了 Muen，我們再回頭看 ARM mbed，2015 年發布了 Beta 版本，值得一提的是 uvisor: http://www.mbed.com/en/technologies/security/uvisor/
這是什麼樣的技術呢？想像我們在 mbed 作業系統中得存取由廠商 (如 Nordic) 提供的 Bluetooth Low-energy stack，才能夠做出豐富的應用，但很多時候，系統的漏洞就來自 BLE stack 不夠安全，進而讓整個系統暴露在各式攻擊的危機中。倘若這時，我們能夠將類似 Muen 的技術引入於 mbed 作業系統，確保主體程式和 BLE stack (或其他私有系統) 是獨立運作，是否就能將危機控制在可預期的程度呢？

ARM Ltd. 主導的 uvisor 就是巧妙利用 ARM Cortex-M 系列的 Memory Protection Unit (MPU) 和 sandboxing 技巧，做出能夠讓兩個以上執行環境獨立運作的機制。

From L3 to seL4 what have we learnt in 20 years of L4 microkernels?

- 影片連結 : https://www.youtube.com/watch?v=RdoaFc5-1Rk
- 投影片連結 : http://www.slideshare.net/microkerneldude/from-l3-to-sel4-what-have-we-learnt-in-20-years-l4
- 論文: https://t.co/t3SLxKQcHm
- Microkernel
  - L4 Microkernel Family
- L4 kernel 只實做三個抽象概念所需的基礎建設
  - Address space
  - Thread
  - IPC
- 1993 年開始發展L4 microkernel，作為第二代的 microkernel 實做
  - Improving IPC by kernel design, by Jochen Liedtke http://www.read.seas.harvard.edu/~kohler/class/aosref/liedtke93improving.pdf
- Microkernel的概念在70年代就已經成形，80年代變得熱門，但也有致命的缺點
  - Mach microkernel https://www.wikiwand.com/en/Mach_(kernel)



- 100µs-disaster: IPC latency超過100µs
- IPC code的spatial locality太差
- L4 kernel效能增進20倍，且對message length有較好的scalability
  - High IPC performance成為L4 kernel的特點
  - 1990 年代所產出的 microkernel，效能都十分低，許多人認為無法實做出有效率的 microkernel。然而 Jochen Liedtke 證實 microkernel 的效率低落是實做方式所造成
  - 1997 年 Hermann Hartig嘗試在 L4 上面跑虛擬化的 Linux，也只有不到 10% 的 overhead
- (L4) microkernel 的 IPC 效能會受系統架構影響，例如 Pentium4 和 Itanium

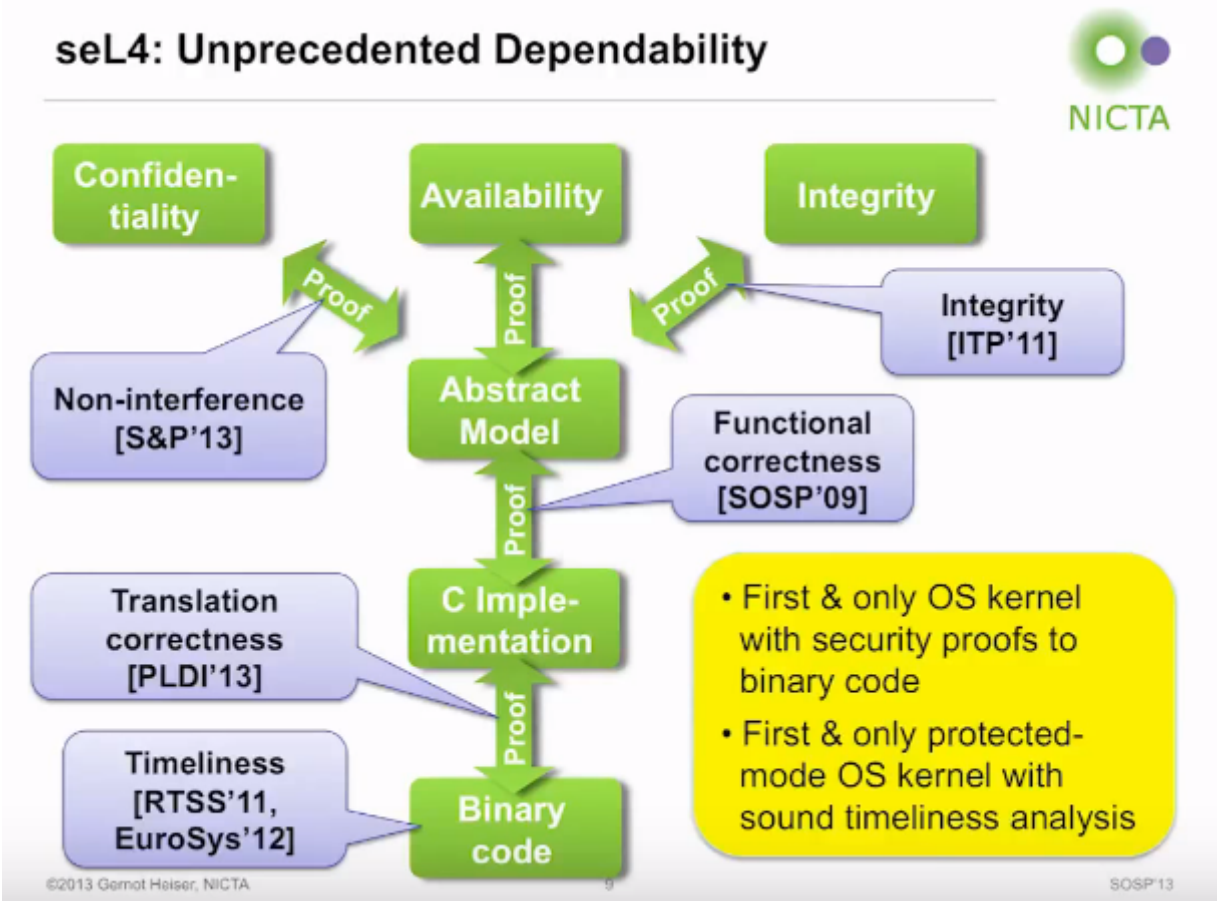
IPC Performance over 20 Years

NICTA

Name	Year	Processor	MHz	Cycles	µs
Original	1993	i486	50	250	5.00
Original	1997	Pentium	160	121	0.75
L4/MIPS	1997	R4700	100	86	0.86
L4/Alpha	1997	21064	433	45	0.10
Hazelnut	2002	Pentium 4	1,400	2,000	1.38
Pistachio	2005	Itanium	1,500	36	0.02
OKL4	2007	XScale 255	400	151	0.64
NOVA	2010	i7 Bloomfield (32-bit)	2,660	288	0.11
seL4	2013	i7 Haswell (32-bit)	3,400	301	0.09
seL4	2013	ARM11	532	188	0.35
seL4	2013	Cortex A9	1,000	316	0.32

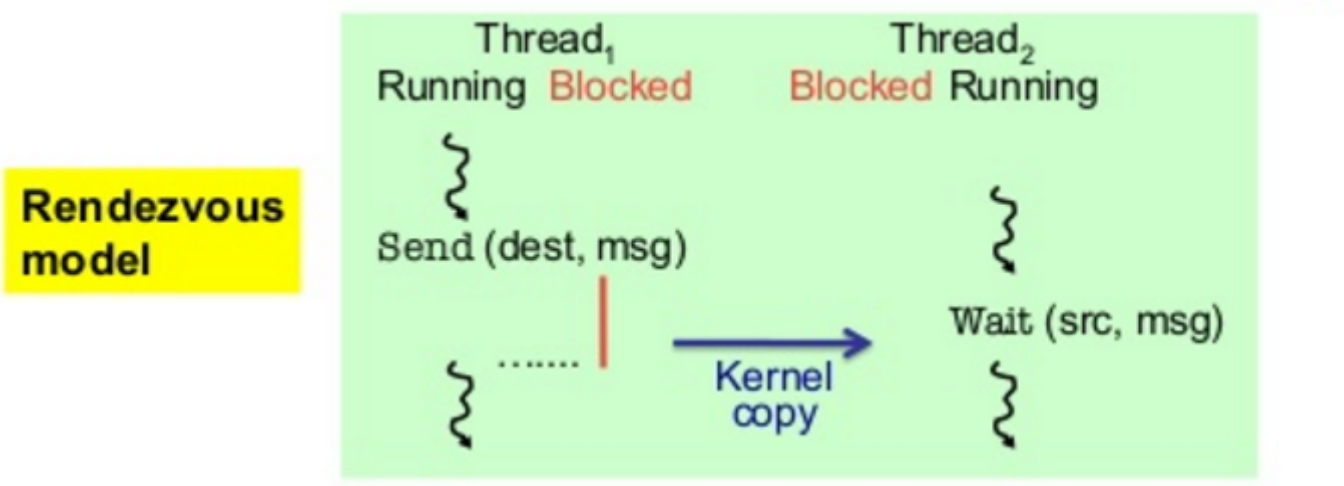
- Microkernel的核心開發理念: **最簡化(Minimality)**，把所有非必要的部分從核心去除

  A concept is tolerated inside the microkernel only if moving it outside the kernel, i.e. permitting competing implementations, would prevent the implementation of the system's required functionality.
  - 比較簡單的評估方式是看原始碼長度(SLOC, Source Lines of Code)
  - 基本上 L4 Microkernel的大小 (或功能) 在二十年間都沒有增長，API 反而更為簡化
    - 最初的 L4 microkernel 有六千多行 (全部都是組合語言!)，seL4 microkernel 則有一萬行 portable C
- L4 kernel 擁有超過十億名使用者，應用層面包括手機、交通工具 (飛機、火車、汽車) 等
- seL4 microkernel: 從高階應用到 binary code 都有經過安全性驗證



- L4 Synchronous IPC model

  In synchronous IPC, the first party (sender or receiver) blocks until the other party is ready to perform the IPC. It does not require buffering or multiple copies, but the implicit rendezvous can make programming tricky.



Kernel executes in sender's context
- copies memory data directly to receiver (single-copy)
- leaves message registers unchanged during context switch (zero copy)

- Page fault handler處於user level

相關論文

- [ ] seL4: Formal Verification of an OS Kernel
- [ ] 減少(kernel) privileged code size，以降低系統缺陷的衝擊
  - small hypervisors as minimal trust base
  - Common Criteria for IT security evaluation
- 從高階規格至C語言實做的驗證
- 假設編譯器、assembly code和硬體的正確性
- Functional correctness:實作過程嚴格遵照 kernel 設計所定義的行為規格

Q&A (請用「複製貼上」，將你的問題列於下方

- 請問使用 FPGA IP (intellectual property) cores 來實作 microkernel IPC 有何缺點？（目前看到有機會的平台是 zedboard 和 icore2）
  - https://github.com/insop/hyos
- nanokernel 和 microkernel 這二者之間的本質上的區別？
  - nanokernel is marketing name!
- 請問存不存在一種設計叫作「microkernel hypervisor」？我的意思是，是否存在有三種不一樣的設計分別是：[1] microkernel [2] hypervisor (vmm) [3] microkernel hypervisor？
  - marketing-driven (I think)
  - Google:// microvisor vmware
- 前面說 microkernel 的效能問題，可以由實作改善。不過舉的實際應用似乎多在精減的MBD環境下。看起來 realtime 跟高可靠性是 microkernel 的賣點。不曉得在一般作業環境，甚至是桌面系統，microkernel 是否也能勝任愉快？或者應該問，microkernel 除了 realtime, high reliability 的利基點？
  - L4Re: https://l4re.org/screens.html
  - Genode: https://genode.org/