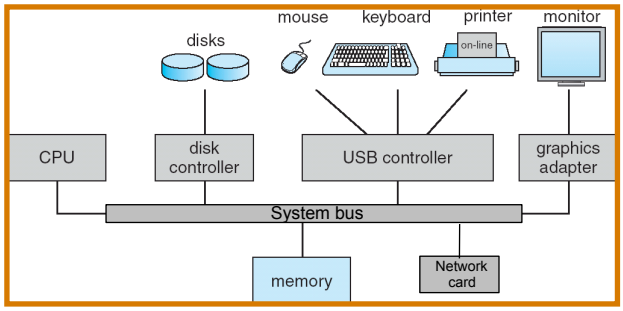
Last Class: OS and Computer Architecture



- CPU, memory, I/O devices, network card, system bus

Last Class: OS and Computer Architecture

OS Service	Hardware Support
Protection	Kernel/user mode, protected instructions, base/limit registers
Interrupts	Interrupt vectors
System calls	Trap instructions and trap vectors
I/O	Interrupts and memory mapping
Scheduling, error recovery, accounting	Timer
Syncronization	Atomic instructions
Virtual memory	Translation look-aside buffers

Today: Microkernel Design

- Brief History of UNIX, Minix and Linux
- Microkernel Organizagtion and Minix
- Device driver Reliability and Microkernels

- **Readings for today's lecture**
- A S. Tannenbaum, "Lessons learned from 30 Years of Minix," *CACM*, March 2016
- J. Herder, et. al "Construction of a Highly Dependable OS," *Proc. Sixth EDCC,* 2006
- J. Herder, et. al., "Reorganizing UNIX for Reliability," *Proc. ASSAC*, 2006.
- [Optional] A S Tannenbaum, "CAn we make OSes Reliable and Secure?" IEEE Computer 2006.

Module 1

- Paper 1: A S. Tannenbaum, "Lessons learned from 30 Years of Minix," *CACM*, March 2016
-

UNIX History

- **1964**: MULTICS (MIT, Bell Labs, GE)
- Bell Labs: Thompson reimplemented stripped down Multics on old discarded PDP-7 (UNICS)
- **1972**: Thompson, Ritchie (designer of C) reimplement UNIX on PDP-11
  - 1975: UNIX V6 released to universities for $300
- Adopted by many universities to teach OS
- AT&T V7 license: forbids use of UNIX for teaching
- Spurred other efforts to fix this restriction
- **1977**: Bill Joy creates BSD1 at Berkeley (BSD Unix)

MINIX

- **1984**: AST created Minix to reimplement V7
- Minix-UNIX (Minix) designed for IBM PC
  - Designed for 256KB RAM, 360KB 5.25 inch floppy disk
- Design goals
  - Open source, clean design that students can understand, small micro-kernel, rest of OS are user processes, communicate using synchronous message passing
- Small micro-kernel: scheduler, low-level process mgmt, IPC, device drivers
- Initial version: crashed after an hour/ INT 15 heat issue
- Led to Linux, Android, …

Linus Torvalds Buys a PC

- **Jan 1991**: Linus student at U. Helsinki
  - Bought "fast" 33MHz i386 PC for studying Minix
- March 3 Post on USENET
  - "Hello everybody, I have had Minix for a week now…"
- In ten days, read/understood Minix code to comment in bulletin boards
- Aug 25, 1991 "Hello everybody out there using minix—I'm doing a (free) operat- ing system (just a hobby, won't be big and professional like gnu) for 386(486) AT clones."
  - Rest is history…
- For over a year, continued to study Minix and develop Linux

Tanenbaum-Torvalds Flame Wars

- Jan 29, 1992 comp.os.minix post" "Linux is obsolete"
  - "Microkernels are better than monolithic design, except for performance"
- Read it here: https://www.oreilly.com/openbook/opensources/book/appa.html
- Part II: A Security argument re: microkernels
  - https://www.cs.vu.nl/~ast/reliable-os/

Module 2

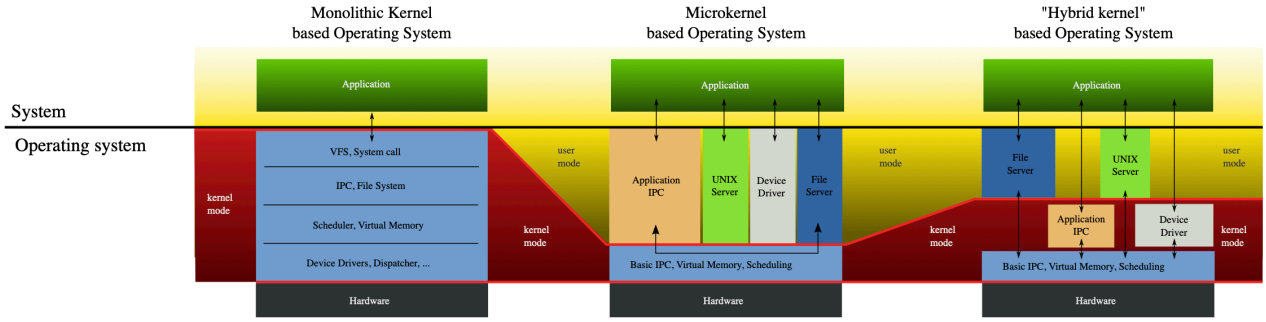
- Why microkernels?
  - "Construction of a Highly Dependable Operating System" J. Herder
- Old Unix joke
  - "Unix is structured like an onion. If you look closely at its insides, you will cry." [Oscar Vivo on Twitter]

Why do OSes crash?

- Commodity OS: monolithic design
  - kernel runs entire OS in a single address space and in privileged mode
  - any module can corrupt any other part of the OS
  - Windows "blue screen of death"
- Software bugs: code contains 6-16 bugs per 100 LOC
- Linux kernel: 2.5 million LOC => ~25,000 code
  - Windows probably 2X
- Device drivers typically 70% of OS code
  - More buggy than rest of OS: 3-7 times more buggy
- Most OS crashes: device drivers (80% of WinXP crashes)

Different OS Structures

- Monolithic, Micro-kernel, Hybrid
  - Different implications of device driver crashes

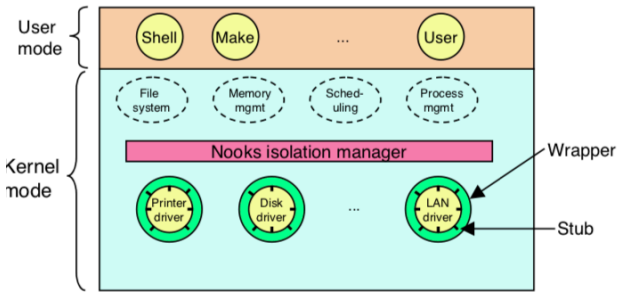


Key Idea

- Move device drivers out of the OS kernel
  - Isolate faults to a device driver process that runs independently of the OS kernel

Ideas to fix Monolithic Kernels: Nooks

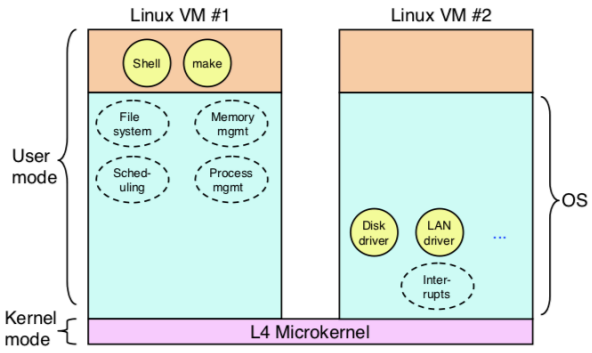
- Nooks project for Linux [Univ of Washington, 2006]
- Keep device drivers in the kernel, enclose them in a lightweight wrapper
  - Prevents bugs from propagating to other parts of kernel
  - "traffic" between kernel & driver inspected by reliability layer
  - Recover automatically when drivers fail



1) Make pages read-only when driver runs
2) Interposition/wrappers to inspect fn params etc

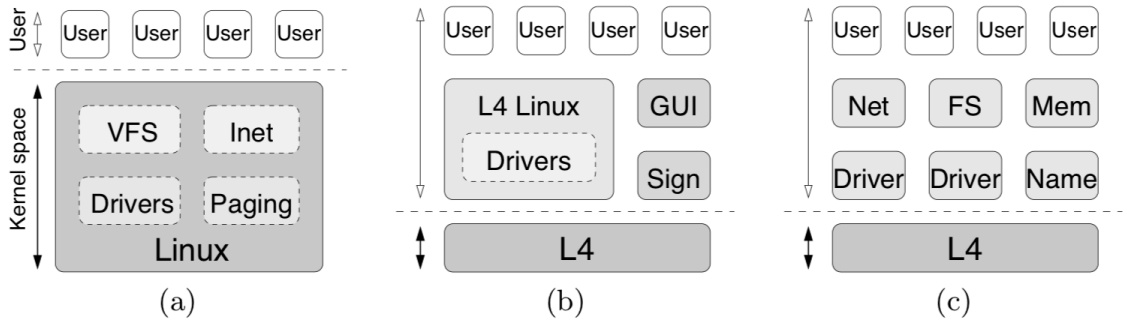
Idea #2: Use virtualization

- Run on top of a virtual machine hypervisor
- Run OS kernel and device driver in different VMs
- Device driver crash => only that VM fails



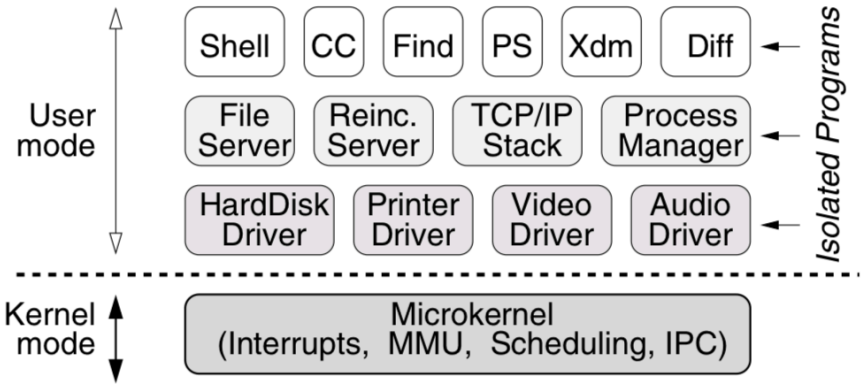
L4 Microkernel

- L4 micro-kernel: run L4linux on top of u-kernel
- Sawmill Linux (abandoned project) envisioned a full multiserver design for L4



Micro-kernel Approach

- Multiserver Operating System
  - Run device drivers outside the kernel (kernel is a compact u-kernel)
  - Many OS components also run as user-level processes



Singularity

- Use **type-safe languages** to write OS kernels
  - Type-safe sing# (based on C# with message passing)
- Language safety for system and user processes
- *All* processes (OS + user process) runs in one virtual address space
  - Compiler does not allow one process to access another
- Restrictions
  - Dynamic process extensions forbidden
  - Loadable modules (drivers, browser plug-ins) not permitted
    - to avoid introducing unverified code

Singularity Microkernel

- Micro-kernel and user processes: one address space
- kernel: access to hardware, rem allocation
- Device driver: separate process per driver
- Verification
  - Each system component has meta-data to describe
    - dependencies, resources, behavior
    - 3 step verification
      - compiler checks for type- safety, ownership, etc
      - Generates intermediate code, like JVM byte code
      - Finally compiled to x86 code with possible run-time check
- Not the same as formal verification (e.g., SEL4)

Drivers in Minix 3

- Minix2: hybrid micro-kernel
  - Drivers inside kernel, but FS, me. mgr, networking outside
- Minix 3: Remove all drivers from kernel
  - Each driver: separate user-mode process, private address space
  - New driver specific kernel calls
  - Separate the driver code from interrupt handlers

Kernel Call	Purpose
SYS_VDEVIO	Read or write a vector of I/O ports
SYS_VIRCOPY	Safe copy between address spaces
SYS_IRQCTL	Set or reset an interrupt policy
SYS_GETINFO	Get a copy of kernel information
SYS_SETALARM	Set or reset a synchronous alarm
SYS_PRIVCTL	Restrict a process' privileges

Summary

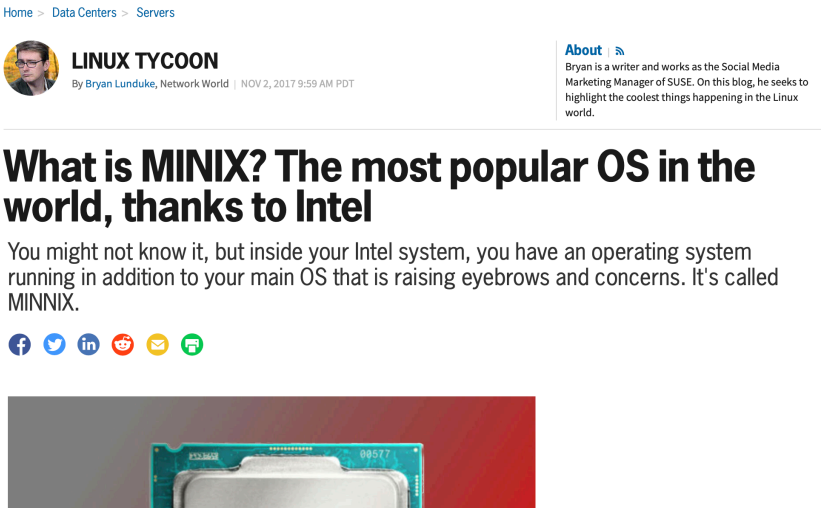
- Device driver bugs most frequent cause of OS crashes
- Many approaches to isolate the device driver
  - Nooks wrapper for Linux
  - Virtualization: run drivers in a different VM
  - Hybrid L4 micro-kernel: run all drivers in a separate process
  - Multi-server OS: run each driver in a separate process
  - Singularity: Language-based protection
- Windows : printer driver isolation mode
- Embedded OS routinely use micro-kernels (QNX OS)
  - runs on > 1 Billion radio chips for phones

Module 3

- The Minix 3 Micro-kernel architecture
- Paper 3: Reorganizing UNIX for Reliability

Minix: The world's most popular OS?

- Minix micro-kernel runs on management engine of all Intel processors (separate from the actual OS)
- https://www.networkworld.com/article/3236064/minix-the-most-popular-os-in-the-world-thanks-to-intel.html



Home > Data Centers > Servers

LINUX TYCOON

By Bryan Lunduke, Network World | NOV 2, 2017 9:59 AM PDT

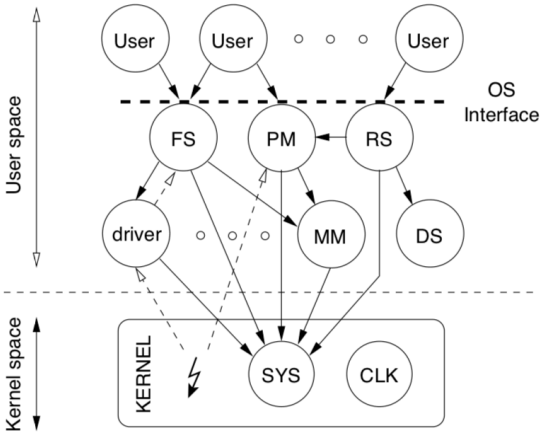
About

Bryan is a writer and works as the Social Media Marketing Manager of SUSE. On this blog, he seeks to highlight the coolest things happening in the Linux world.

What is MINIX? The most popular OS in the world, thanks to Intel

You might not know it, but inside your Intel system, you have an operating system running in addition to your main OS that is raising eyebrows and concerns. It's called MINNIX.

Minix 3 Design



- FS: file server
- PM: process manager
- MM: mem manager
- DS: data store
- RS: reincarnation server
- Kernel: SYS & Clock
  - 4000 Lines of code
- Each server ~ 1K - 3K

Example OS functions

- Create a process via fork()
  - Send message to PM, PM asks MM to allocate memory and requests SYS to create a copy of the process.
  - All IPC hidden to process that made the request
- read() call for disk I/O
  - Send request to FS. IF block in buffer cache, FS asks SYS To copy it to user. Otherwise send message to disk driver. Driver sets alarm, commands disk controller through SYS, await INT.
- RS reincarnation server: start servers or drivers on the fly. Fork process, set privileges in SYS, publish on DS.

Microkernel Components

- Three key component: IPC, SYS, Clock

- IPC: reliable inter-process communication via messages
  - IPC_REQUEST: send a message, blocks sender
  - IPC_REPLY: send a reply
  - receiver: IPC_SELECT to wait (block) for a message
  - IPC_NOTIFY: non-blocking notifications
- Kernel maintains lists to limit powers of system processes
  - allowed IPC primitives, who can talk to whom, kernel calls available, I/O ports, IRQ lines, memory regions
  - Policies set by RS (reincarnation server)

Kernel SYS Task

- Interface to the kernel for all user-mode servers/drivers
- All kernel calls are transformed into a request msg
  - SYS handles the request if caller authorized
  - SYS always blocked waiting for a request
- Five groups of kernel calls
  - process management, memory management, access to kernel data structures, interrupt management, clock services
  - Examples
    - SYS_DEVIO: device I/O
    - SYS_VIRCOPY: copy data t
    - SYS_SETALARM: schedule alarm
    - SYS_PRIVCTL: set process privilege

Kernel CLOCK Task

- Accounting for CPU usage
- Schedule another process when quantum expires
- Manage watchdog timers
- Interact with hardware clock
- Registers an interrupt handler for each clock tick
  - increments process CPU usage, decrement scheduling quantum
  - Quantum expiration/alarm: sent notification to Clock
- SYS_SETALARM: a process can schedule a synchronous alarm; timeout upon expiration

Process Manager

- POSIX interfaces: servers + drivers use IPC to provide functionality of an ordinary UNIX OS
- Process manager
  - *Management*: create/remove processes, assign PIDs, track parent-child relations
  - *Scheduling*: schedule highest-priority ready process
  - *POSIX Signal handling*
    - User process registers signal handler to catch signals;
    - PM interrupts process, puts signal frame on stack; run handler

Memory Manager

- Handware-independent memory model
- User process: text segment, stack and data segment
  - Text read-only, stack/data non-executable (avoid buffer overflow)
- System process: can be given access to video RAM
- Management
  - Track free memory region, allocate/release memory
    - Some of this is integrated into PM (but future versions of Minix will separate it out to MM).

File Server

- Previously: offers one file system only
- Virtual File server
  - support for multiple file systems
  - VFS and each file server run as user process
  - Virtual file system: a Unix mechanism to support multiple files system
  - Each file server tracks its mapping onto devices/ device drivers

Reincarnation Server

- Managers all OS servers and drivers
- PM: manage user processes, RS: manage system proc.
- service: CLI to interface with RS
  - start / stop services; change policies
- Fault recovery
  - RS adopts all system processes as its children
  - System process exits: SIGCHLD signal delivered by PM to RS
  - Problem detected: take action for fault component
    - Policy: restart, log to sys log, core dump,
    - Repeated crashes: exponential backoff

Data Store

- Database with pub-sub functionality
- System processes: store their data/state privately
  - Provides redundancy; restart: load state from DS
- Pub-sub: glue between OS components
  - Allows components to subscribe to events
- Provides Naming Service
  - Each process has IPC end-point; IPC endpoint registered with DS
  -

Minix Fault Isolation

- Servers and drivers: private address space
  - protected by MM hardware (can not access memory of another process).
- User mode processes do not run with root privileges
  - limit access to file system and POSIX calls
  - RS sets restrictions policy for each server/driver
- Ordinary process: can not request kernel services, contact Posix servers
-

Failure Resilience

- DS: pub-sub to notify changes
  - Key component for fault tolerance
  - FS subscribes to disk drivers events;
  - Driver crash: RS registers a new one and notifies FS to take further action
  - RS handles driver failures and restarts failed drivers
    - does not yet handle OS server failures / auto-restart

Performance

- Moving drivers out of kernel: 5-10% slowdown
- Disk-bound process: 7% overhead
- Restart failed ethernet driver: 1 to 15 sec
- What about normal process operations?