On Microkernel Construction

Lei Yan
(Slides adopted from Marios Kogias)

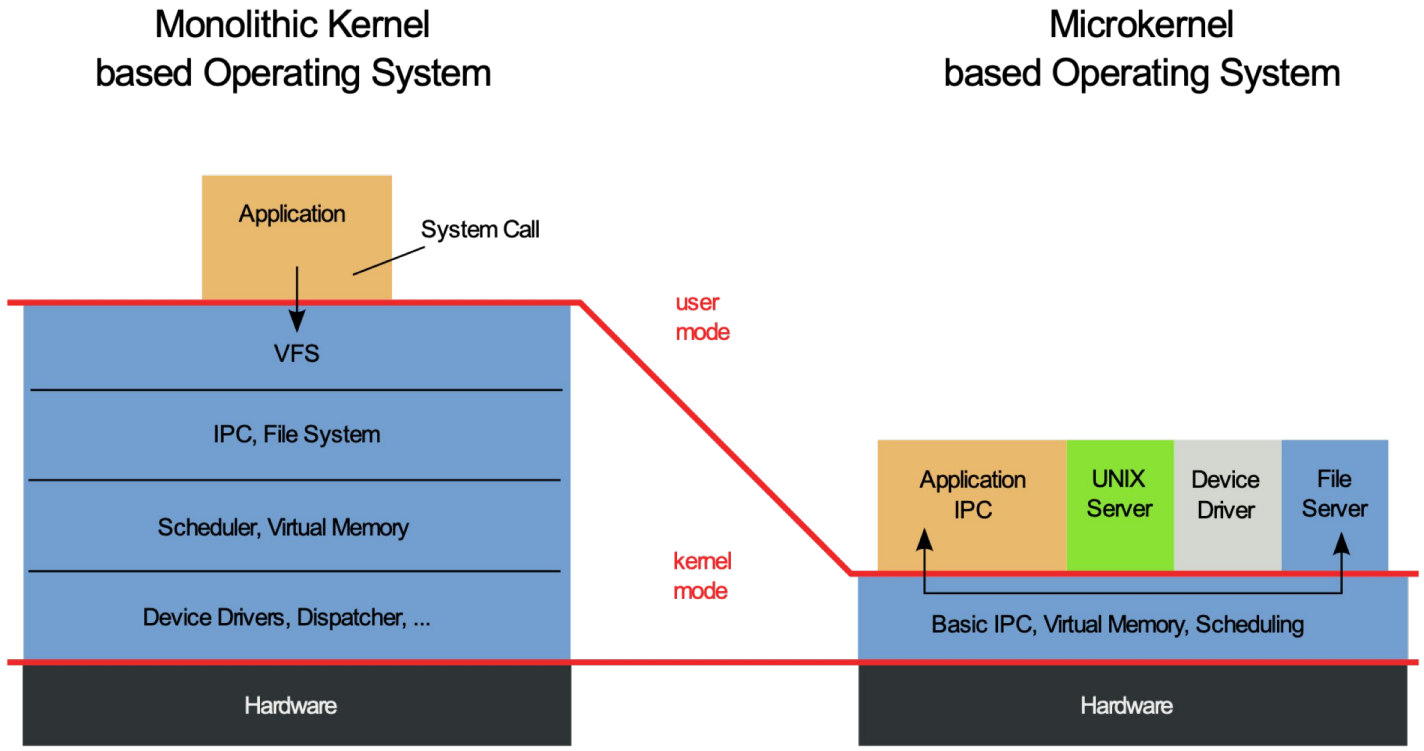
µ-kernels

What is a µ-kernel and what are the advantages of the µ-kernel design?

- Enforced modularity
  - Fault isolation

On µ-Kernel Construction

- Main Ideas
  - Minimality Principle
    - Address Spaces
    - Threads & IPC
    - Unique Identifiers
  - Misconceptions:
    - Performance



Principles

- Independence

  A programmer must be able to implement an arbitrary subsystem S in such a way that it cannot be disturbed or corrupted by other subsystems S’

- Integrity

  There must be a way for S_1 to address S_2 and establish a communication channel which can neither be corrupted or eavesdropped by S’

How does a µ-kernel guarantee integrity and authentication for the IPC?

What is the underlying assumption in the paper in order to guarantee the two principles?

Check Reflections on Trusting Trust

Is the microkernel design in the paper free of performance overhead compared to monolithic kernel with today's hardware?

- IPC vs. Procedure call:
    - Min. 2x syscall vs. 1x function call (100 vs. 10 cycles)
    - Function calls can even have 0 overhead (e.g., inlining by smart compiler)

Enforced modularity is not free (given today's hardware)

- There are designs with much less performance overhead but comes with other trade-offs
    - E.g., eBPF is kind of turning Linux into a microkernel (see the discussion on duality paper for details)

Compare the µ-kernel with the exokernel design

“The presented design shows that it is possible to achieve well performing µ-kernels through processor-specific implementations of processor-independent abstractions.”

BACKUP

Microkernel is not inherently slow

- IPC

  L3 IPC is 23x faster than Mach on 486 (250 vs. 5750 cycles)

- Mode-switch

  Theoretcically fast (107 cycles on 486), Mach is just not well-optimized (900 cycles on 486).

- Address-space switch

  Can be fast (< 50 cycles on e.g., 486) with hardware support (e.g., tagged-TLB) and various kernel implementation tricks (e.g., emulating tagged-TLB with segmentation).

- Micro-kernel architecture does not inherently lead to memory system degradation (increasing cache capacity miss)

Enforcing Modularity

Rishabh Iyer

21. 10. 2021

Paper Recap

- Two rough models of OS designs
  - Message-oriented vs Procedure-oriented
- Two models are duals
- Dual programs:
  - Are logically identical
  - Can be implemented to have similar performance

Underlying hardware (not app) should determine design

Shared Memory vs Message Passing

- Decades-old debate on the right IPC mechanism
  - Have also been proven to be duals
- Shared memory:
  - Writes to local memory/registers are globally visible
  - Communication is implicit
- Message passing:
  - Communication must be explicitly specified.
  - Must communicate with a process to share data with it

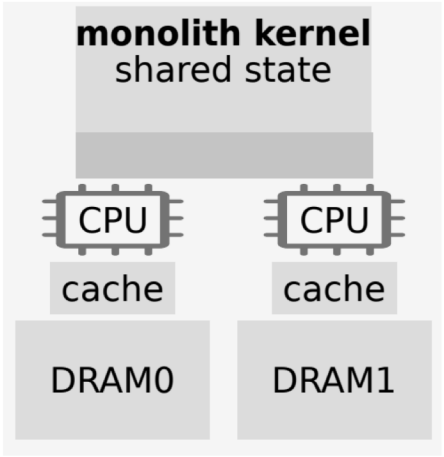
How is this relevant to modularity?

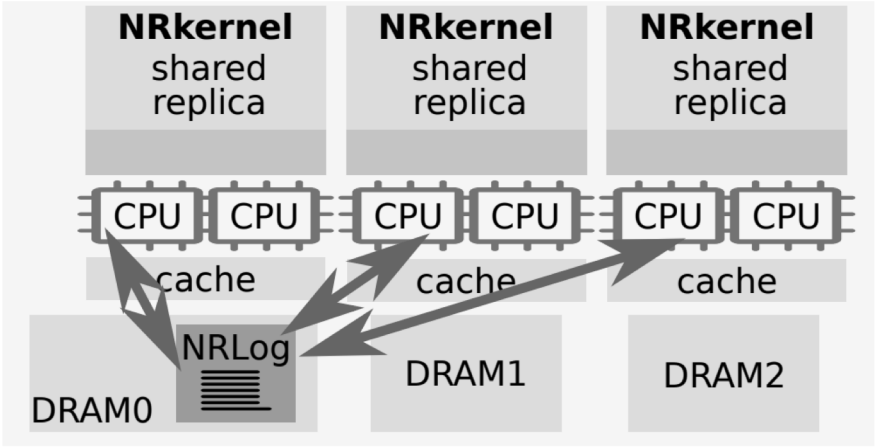
- Message Passing enforces modularity
  - ❖ All communication via explicit messages
  - ❖ Modules are isolated
  - ❖ Propagation of errors is reduced
- Primary disadvantage of enforced modularity?
  - ❖ Performance (marshalling, unmarshalling of messages)
    - 10% of CPU cycles in Google's DC spent running protobuf operations
  - ❖ Semantic coupling may render functional decoupling moot

Revisiting duality for modern HW

- How to build an OS that scales across NUMA nodes?
  - NrOS [OSDI'21]
- How to build an OS for heterogeneous CPU cores?
  - Barrelfish [SOSP'09]
- How to build an OS for disaggregated resources?
  - LegoOS [OSDI'18]

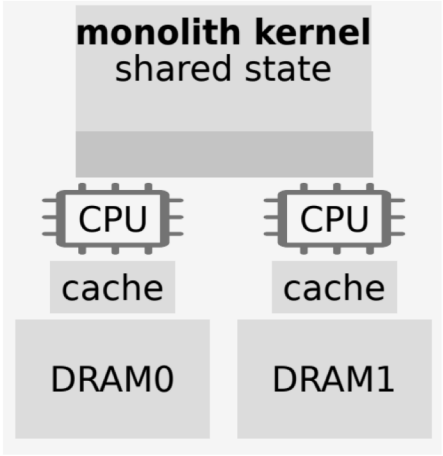
Node-replicated (Nr) OS

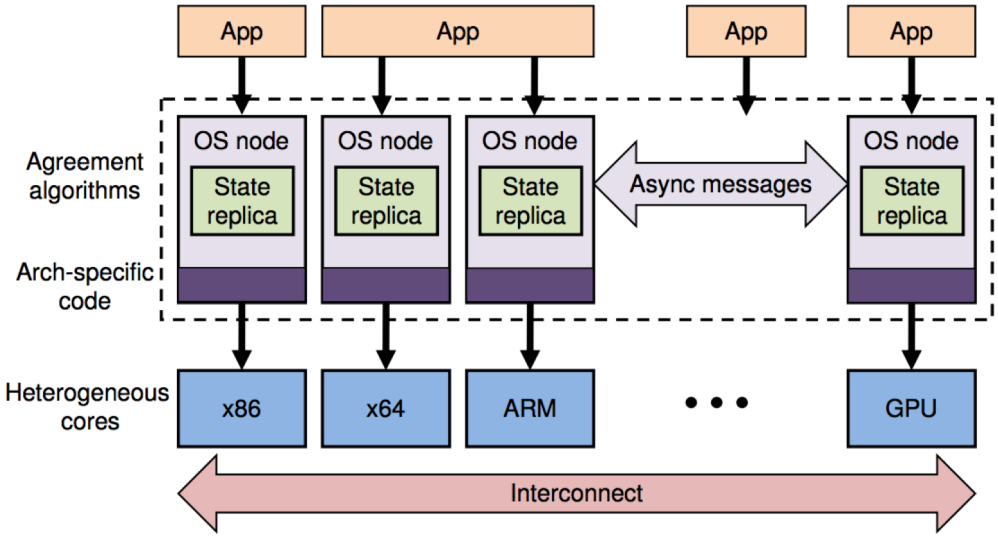




- Motivation?
- Main challenges?
- Tradeoffs?

Barrelfish





- Motivation?
- Main challenges?
- Tradeoffs?

Revisiting microkernels with modern PL

- Linux kernel extensibility is important
  - e.g. Docker relies on OVS, AppArmor, OverlayFS extensions
- Originally implemented using kernel modules
  - What security/isolation guarantees do these provide?
- Modern solution?
  - Extended Berkeley Packet Filter (eBPF)

eBPF 101

- Framework to run sandboxed programs within linux
  - Why would you want to run these programs within linux?
- How is sandboxing done? What is the provided interface?
  - Tradeoffs?
- Justify/argue against the below statement:
  - "The Linux kernel continues its march towards becoming BPF runtime-powered microkernel"

Further Reading (Optional)

- LegoOS [OSDI'18]
- Practical, safe, Linux kernel extensibility [HotOS'19]
- An Incremental Path Towards a Safer OS Kernel [HotOS'21]

Backup slides

Duality

Message-oriented system	*Procedure-oriented system*
Processes, **CreateProcess**	Monitors, NEW/START
Message Channels	External Procedure identifiers
Message Ports	ENTRY procedure identifiers
SendMessage; AwaitReply (immediate)	simple procedure call
SendMessage; . . . AwaitReply (delayed)	FORK; . . . JOIN
SendReply	RETURN (from procedure)
main loop of standard resource manager, **WaitForMessage** statement, **case** statement	monitor lock, ENTRY attribute
arms of the **case** statement	ENTRY procedure declarations
selective waiting for messages	condition variables, WAIT, SIGNAL

Designing good interfaces

- We discussed how server interfaces are defined (IDLs)
- Design considerations for message passing systems:
  - How do I name processes I want to communicate with?
  - What is the message format?
  - Semantics of asynchronous operations?