Systems@ETH Zürich

Kernel construction and OS architecture

Advanced Operating Systems (263-3800-00)

Timothy Roscoe

Thursday 28 October 2010

What is “OS Architecture”?

- Coarse-grained structure of the OS
- How the complexity is factored
- Mapping onto:
  - Programming language features
  - Execution environment presented to applications
  - Address spaces
  - Hardware protection features (rings, levels, etc.)
  - Execution patterns (subroutines, threads,. coroutines)
  - Hardware execution (interrupts, traps, call gates)

Architectural models

- There are many, and they are *models!*
  - Idealized, extreme view of how system is structured
  - Real systems always entail compromises
  - Hard to convey $\Rightarrow$ it's good to build a few
- Think of these as tools for thinking about Oses
  - Each has its reasons
  - Solve particular problems at particular times

Outline

- Monolithic or component-based systems
- Kernel-based systems
- Microkernels
- Kernel thread models
  - Per-thread kernel stack
  - Single kernel stack
- Exokernel systems
  - Nemesis and Exokernel
- Multikernels
  - Barrelfish
- References

1. Monolithic/kernel-based systems

Systems@ETH Zürich

- Examples:
  - Cedar [Swinehart et al., 1986]
  - TinyOS [Hill et al., 2004]
  - Oberon
  - Singularity [Hunt and Larus, 2007]
- Hardware provides time multiplexing
  - Interrupts
  - threads (in Cedar's case)
- Language provides modularity & protection
  - Module calls
  - Inter-thread communication

Protection-based component-based systems

- Examples:
  – KeyKOS [Bromberger et al., 1992]
  – Pebble [Bruna et al., 1999]
- Even simpler kernel than microkernels
  – Kernel only mediates protection domain switches
  – Scheduling, threads, etc. implemented in “user space”
- Aimed at:
  – High security (very small TCB)
  – Embedded systems (highly configurable)

2. Kernel-based systems

- Examples:
  - Unix [Thompson, 1974],
  - VMS → Windows NT/2k/XP/Vista/7
- Hardware enforces user vs. kernel mode
- Machine in user space multiplexed into address spaces
- Kernel provides:
  - All shared services
  - All device abstraction

3. Microkernels

Systems@ETH Zürich

- Examples: L4, Mach, Amoeba, Chorus
- Kernel provides:
  - Threads
  - Address spaces
  - IPC
- All other functionality in server processes
  - Device drivers
  - File systems
  - Etc.
- Instead of syscalls, applications send IPC to servers

Kernel thread models

Systems@ETH Zürich

- Important design choices when implementing an OS:
  - *Do I support more than one execution context in the kernel?*
  - *Where is the stack for executing kernel code?*
  - *Can kernel code block? If so, how?*

- The answers determine the **kernel thread model**.

Kernel thread models

- Important design choices when implementing an OS:
  - *Do I support more than one execution context in the kernel?*
  - *Where is the stack for executing kernel code?*
  - *Can kernel code block? If so, how?*

- The answers determine the **kernel thread model**.
  - You have faced the same choices, although SOS may not be a kernel

Kernel thread models

There are two basic alternatives:

- **Per-thread kernel stack**:
  - Every thread has a matching kernel Stack
- **Single kernel stack**:
  - Only one stack is used in the kernel (per core).

Per-thread kernel stack

- Every user thread/process has its own kernel stack
- Thread's kernel state implicitly stored in kernel activation stack
- A kernel thread blocks ⇒ switch to another kernel stack
- Resuming: simply switch back to original stack
- Preemption is easy
- No conceptual difference between kernel- and user-mode

```
example(arg1, arg2) {
  P1(arg1, arg2);
  if (need_to_block) {
    thread_block();
    P2(arg2);
  } else {
    P3();
  }
  /* return to user */
  return SUCCESS;
}
```

Single kernel stack

- Challenges:
  - How can a single kernel stack support many threads?
  - How are system calls that block handled?

- Two basic approaches:
  - Continuations [Draves et al., 1991]
  - Stateless kernel [Ford et al., 1999]

Continuations

Systems@ETH Zürich

- State to resume blocked thread explicitly saved in TCB
  - Function pointer
  - Variables
- Stack can be discarded and reused for new thread
- Resuming involves discarding current stack and restoring the continuation

```
example(arg1, arg2) {
  P1(arg1, arg2);
  if (need_to_block) {
    save_context_in_TCB;
    thread_block(example_continue);
    panic("thread_block returned");
  } else {
    P3();
  }
  thread_syscall_return(SUCCESS);
}
example_continue() {
  recover_context_from_TCB;
  P2(recovered arg2);
  thread_syscall_return(SUCCESS);
}
```

Stateless kernel

- System calls simply do not block within kernel
- If a system call must block:
  - Modify user state to restart call when resources are available
  - Kernel stack content discarded
- Preemption within kernel difficult
  - Must (partially) roll back to a restart point
- Avoid page faults within kernel code
  - System call arguments in registers
  - Nested page fault is fatal

Kernel stack model summary

Per-thread kernel stack:

✓ Simple, flexible

- Kernel can always use threads
- No special technique for saving state when interrupted/blocked
- No conceptual difference between kernel and user mode

✗ Larger cache and memory footprint

- Used by L4Ka::Pistachio, UNIX, Linux, etc.

Kernel stack model summary

Single kernel stack

✓ Lower cache & memory footprint (always the same stack)

Continuations:

- ✗ Complex to program
- ✗ Must save state conservatively (whatever might be needed)
- – Used by Mach, NICTA::Pistachio

Stateless kernel:

- ✗ Also complex to program
- – Must request all resources prior to execution
- – Blocking system calls must be restartable
- ✗ Processor-provided stack management can get in the way
- – System calls need to be atomic
- – Used by Fluke, Nemesis, Exokernel, Barrelfish

Why build a stateless kernel?

- It is the simplest model, if all kernel invocations are:
  - Atomic
  - Non-blocking
  - Bounded and short-running
  - Non-preemptable
  - Guaranteed not to page fault
- Restrictive, but quite appropriate for a uniprocessor μkernel with no blocking IPC.

4. Exokernels

Systems@ETH Zürich

- Examples: Exokernel, Nemesis, Xen 3, ESX...
- Kernel provides minimal multiplexing of h/w
- All other functionality in userspace libraries
  - Unlike microkernels, where this in servers
  - “LibraryOS” concept
- Enables:
  - Strong isolation between applications
  - High degree of application-specific policies

Exokernels: Exterminate all OS abstractions!

[Engler and Kaashoek, 1995]

A traditional operating system, and also a microkernel like L4:

- **Multiplexes** physical resources
  - Shared and secure access to CPU, memory, disk, network, etc.
- **Abstracts** the same physical resources
  - Processes/threads, address spaces, virtual file system, network stack

Exokernels: Exterminate all OS abstractions!

[Engler and Kaashoek, 1995]

A traditional operating system, and also a microkernel like L4:

- **Multiplexes** physical resources
  - Shared and secure access to CPU, memory, disk, network, etc.
- **Abstracts** the same physical resources
  - Processes/threads, address spaces, virtual file system, network stack

- Multiplexing is required for security
  ...but why should an OS abstract what it multiplexes?

Exokernel systems

Systems@ETH Zürich

- Two different systems. Two different motivations:
  1. Complexity, adaptability, performance → **Exokernel**
     [Kaashoek et al., 1997]
  2. QoS crosstalk → **Nemesis**
     [Leslie et al., 1996]

Exokernel systems

- Two different systems. Two different motivations:
  1. Complexity, adaptability, performance → **Exokernel**
     [Kaashoek et al., 1997]
  2. QoS crosstalk → **Nemesis**
     [Leslie et al., 1996]
- The approach of both is similar:
  – Exterminate OS abstractions
  – Move all code possible into the application's address space
    → **library OSes**

Nemesis

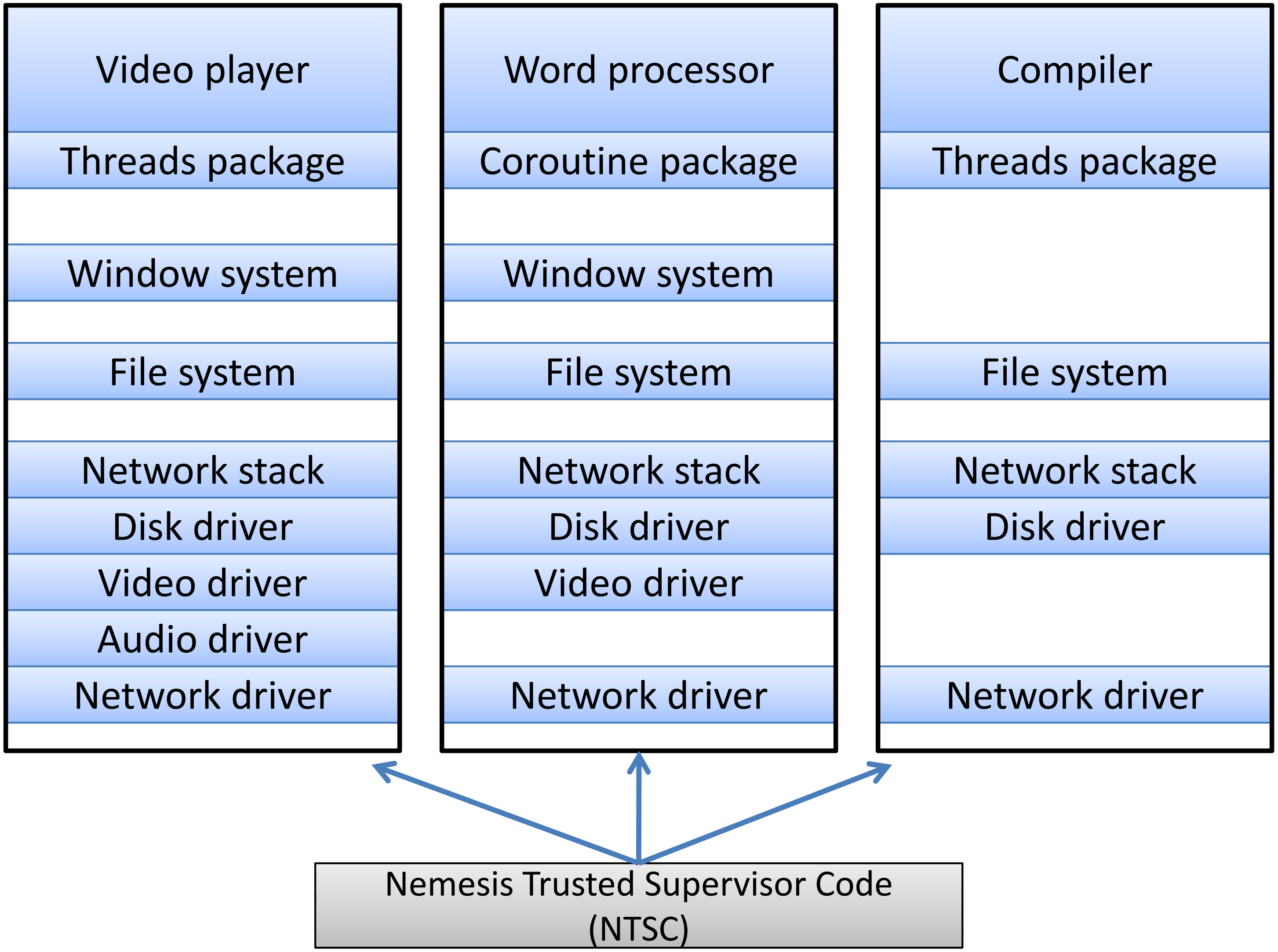
- Written for uniprocessor Alpha, 1992-95
- 64-bit single address space
  - Not a fundamental design motivation, as in Mungi
- *"Multi-service operating system"*
  - Mixture of soft real time, communication-oriented, interactive, batch jobs
  - Designed for workstations
- Strong networking influence
  - Published in JSAC!

What is an application?

In an Exokernel, functionally, *everything:*

- User code
- Network stack
- Filing system
- Window system
- Low-level I/O
- Intra-application communication

Nemesis Application Domains



Nemesis in action

Exokernel challenges

- Can you really expose all the hardware to the application and still stay sane?
- Can you multiplex the machine securely while removing (most) abstraction?

Exokernel challenges

- Can you really expose all the hardware to the application and still stay sane?
- Can you multiplex the machine securely while removing (most) abstraction?

Apparently, yes:

- Threads and processes: see *scheduler activations later*
- Networking: packet filtering
- Disks (file systems): block or track-level protection, careful management of metadata
- Window system: similar; blit tiles into protected windows

Programmability questions

- Isn’t it all rather complex to move functionality into the app?
  - **No**: libraries do what the kernel or servers used to do.

Programmability questions

Systems@ETH Zürich

- Isn't it all rather complex to move functionality into the app?
  - **No**: libraries do what the kernel or servers used to do.
- Does the flexibility impact performance?
  - **No**: protection checks are mostly off the fast path
  - Each application can efficiently implement its policy

Programmability questions

- Isn't it all rather complex to move functionality into the app?
  - **No**: libraries do what the kernel or servers used to do.
- Does the flexibility impact performance?
  - **No**: protection checks are mostly off the fast path
  - Each application can efficiently implement its policy
- What happens on a multiprocessor?
  - **Unclear**: a multiprocessor kernel requires plenty of embedded policy (e.g. locks)
  - Attempts to produce MP exokernels have not been as dramatically better at performance

6. The Multikernel

[Baumann et al., 2009]

An architecture aimed at **heterogeneous**, **manycore** machines:

- Lots of processors
- Not all of them the same
- Not all of them share memory
- Not all the shared memory is cache-coherent
- You don't know in advance what the machine looks like

Very new: published last year (though similar designs have existed in the past).

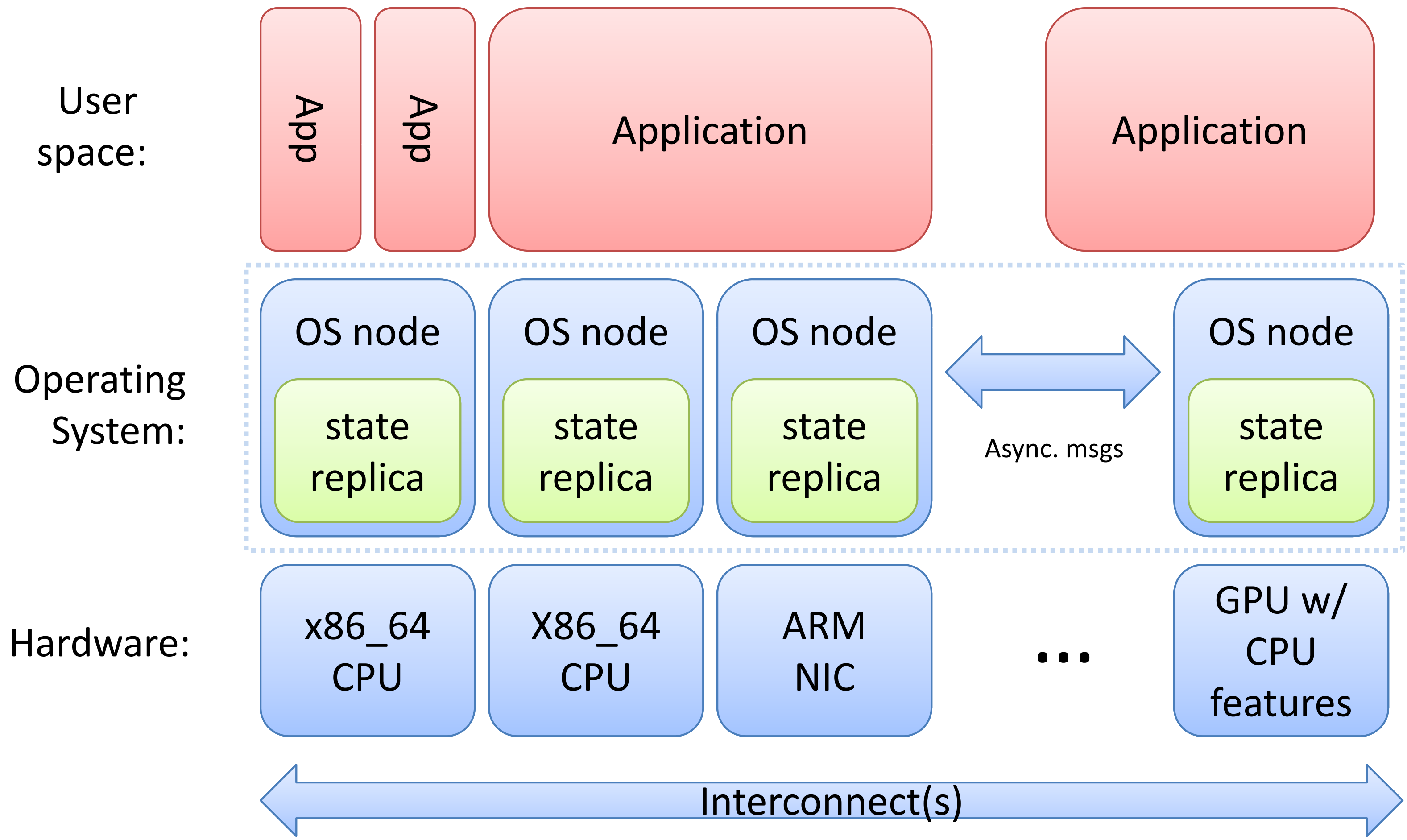
Multikernel design principles

Systems@ETH Zürich

1. Use messages, not shared memory, between cores
2. Decouple OS structure from hardware configuration
3. Treat all (potentially global) OS data as a replica

Multikernel architecture

Instead of the kernel as a multithreaded, shared-memory program, treat the machine as a **distributed system**:



Advantages

- You need this for core heterogeneity
- You need this for non-shared memory
- Handles cores coming and going (power, failure, hot-plug)
- Separating structure (algorithms) from hardware scalability tradeoffs makes the design **agile**.

Advantages

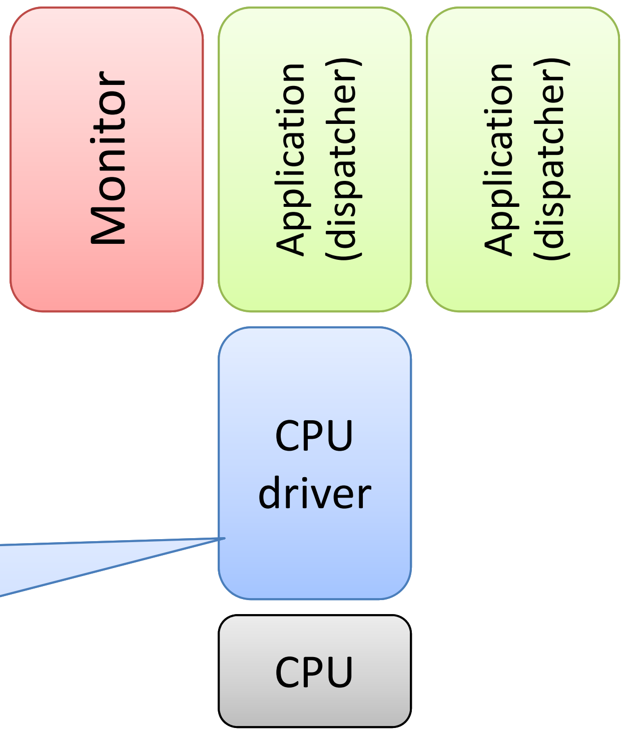
- You need this for core heterogeneity
- You need this for non-shared memory
- Handles cores coming and going (power, failure, hot-plug)
- Separating structure (algorithms) from hardware scalability tradeoffs makes the design **agile**.

Or so we hope, anyway.

Challenges

- You need to communicate between cores efficiently
  - Message transports (“interconnect drivers”) are highly specialized (=optimized)
  - We cover some known techniques later in this course
- You still need to design the kernel on each core! Key differences:
  - It is now implicitly a uniprocessor kernel
  - Highly communication-oriented (to other cores)
  - Can be highly architecture-specific (c.f. L4)

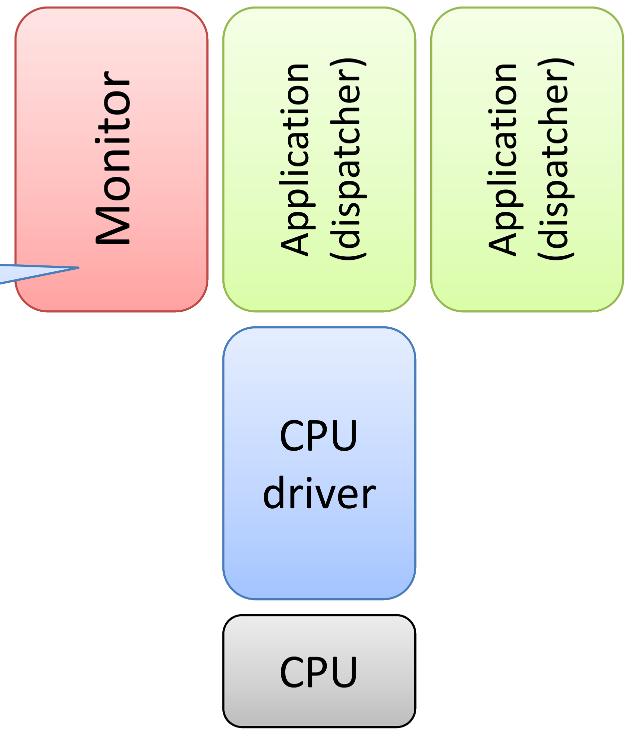
Barrelfish per-core architecture



- Kernel space
- Manages CPU, MMU, APIC
- Multiplexes core btw. *dispatchers*
- Handles interrupts
- Implements protection via *capability* validation
- Single-threaded, non-preemptable

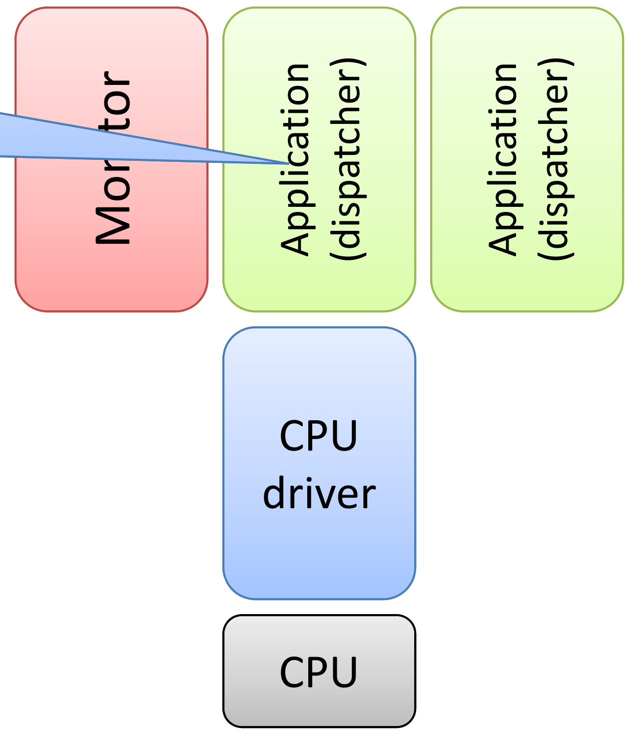
Barrelfish per-core architecture

- User space (extra privilege)
- Communicates with other monitors
- Manages distributed operations
- Performs long-running operations



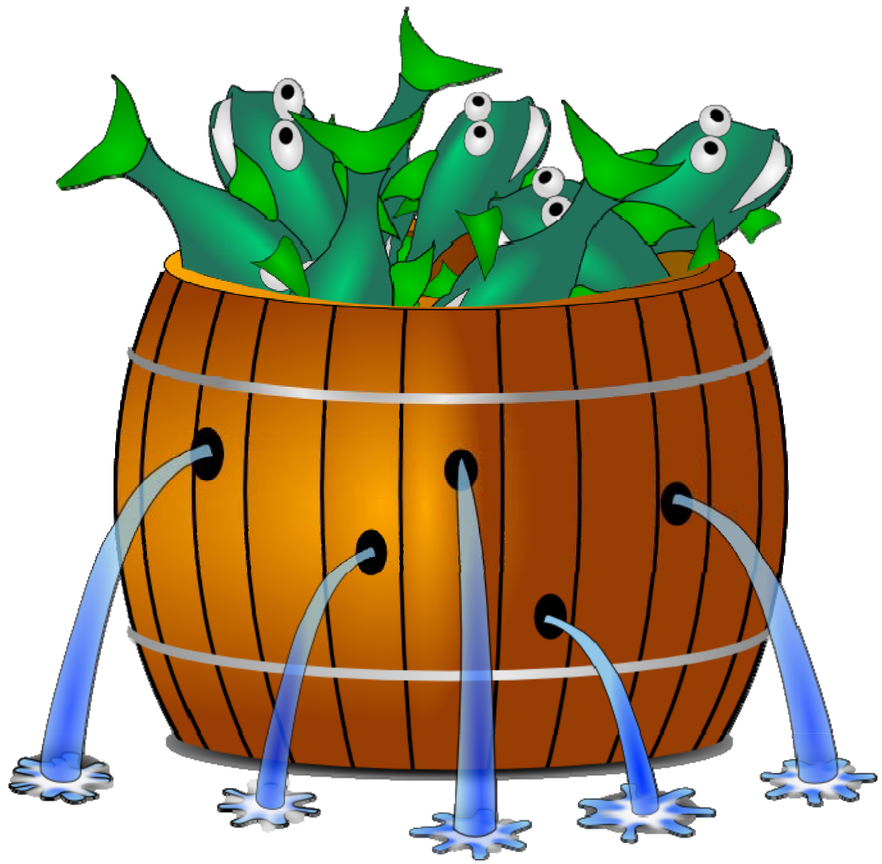
Barrelfish per-core architecture

- Representative of app. on each core (including drivers and services)
- Upcalled from CPU driver
- Local thread scheduling
- Communicates with peer dispatchers



Barrelfish

- More information at www.barrelfish.org...



References

- Baumann, A., Barham, P., Dagand, P.-E., Harris, T., Isaacs, R., Peter, S., Roscoe, T., Schuepbach, A., and Singhania, A. (2009): **The multikernel: a new OS architecture for scalable multicore systems.** In *SOSP '09: Proceedings of the 22nd ACM symposium on Operating systems principles, New York, NY, USA. ACM* Press.
- Bromberger, A. C., Frantz, A. P., Frantz,W. S., Hardy, A. C., Hardy, N., Landau, C. R., and Shapiro, J. S. (1992). **The KeyKOS nanokernel architecture.** In *Proceedings of the USENIX Workshop on Microkernels and other Kernel Architectures, pages 95–112.*
- Bruna, J., Brustoloni, J., Gabber, E., Özden, B., and Silberschatz, A. (1999). **Retrofitting quality of service into a time-sharing operating system.** In *Proceedings of the 1999 Annual USENIX Technical Conference, pages 15–26.*
- Draves, R., Bershad, B. N., Rashid, R., and Dean, R. (1991). **Using continuations to implement thread management and communication in operating systems.** In *Proceedings of the 13th ACM Symposium on Operating Systems Principles.*
- Engler, D. R. and Kaashoek, M. F. (1995). **Exterminate all operating system abstractions.** In *Proceedings of the 5th Workshop on Hot Topics in Operating Systems.*
- Ford, B., Hibler, M., Lepreau, J., McGrath, R., and Tullmann, P. (1999). **Interface and execution models in the Fluke kernel.** In *Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation, pages 101–115.*

References

- Hill, J., Horton, M., Lking, R., and Krishnamurthy, L. (2004). **The platforms enabling wireless sensor networks**. *Communications of the ACM, 47(6):41–46.*
- Hunt, G. C. and Larus, J. R. (2007). **Singularity: rethinking the software stack.** *SIGOPS Operating Systems Review, 41(2):37–49.*
- Kaashoek, M. F., Engler, D. R., Ganger, G. R., no, H. M. B., Hunt, R., Mazières, D., Pinckney, T., Grimm, R., Jannotti, J., and Mackenzie, K. (1997). **Application performance and flexibility on Exokernel systems.** In *Proceedings of the 16th ACM Symposium on Operating Systems Principles, pages 52–65.*
- Leslie, I. M., McAuley, D., Black, R., Roscoe, T., Barham, P., Evers, D., Fairbairns, R., and Hyden, E. (1996). **The Design and Implementation of an Operating System to Support Distributed Multimedia Applications.** *IEEE Journal on Selected Areas in Communications, 14(7):1280–129.*
- Swinehart, D. C., Zellweger, P. T., Beach, R. J., and Hagmann, R. B. (1986). **A structural view of the Cedar programming environment.** *ACM Transactions on Programming Languages and Systems, 8:419–490.*
- Ritchie, D. M., and Thompson, K. (1974). **The UNIX time-sharing system**. *Communications of the ACM, 17:365–375.*