OS DESIGN PATTERNS II

CS124 – Operating Systems

Spring 2024, Lecture 4

Last Time

- Began discussing general OS design patterns
  - Simple structure (MS-DOS)
  - Layered structure (The THE OS)
  - Monolithic kernels (initial versions of UNIX)
  - Modular kernels (Linux)
- Also mentioned separation of policy and mechanism as a key design goal
  - Can already see that not all design patterns achieve this successfully…

Microkernels

- What OS facilities actually *require* kernel-mode access?
- Only ones that must use privileged CPU capabilities
  - e.g. managing the virtual memory system or interrupt controller
  - e.g. receiving interrupts from the computer hardware
- Another OS structural approach: **microkernels**
  - Restrict the kernel to contain only a minimal set of capabilities
  - *Most* operating system services are provided as user-mode processes
- What facilities should be included in the kernel?
- Jochen Liedtke's "minimality principle":

  "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."

Microkernels (2)

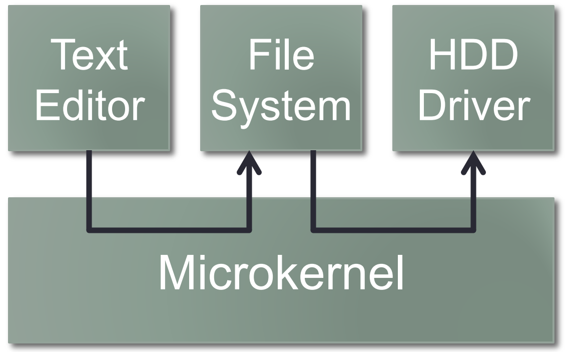
- Minimal set of capabilities provided in microkernels:
  - Process abstraction: context switches, CPU scheduling, interrupt handling, etc.
  - Memory abstraction: process address-space isolation, kernel/user memory separation
  - Inter-process communication facilities: required to allow user-mode processes to work together to implement system facilities
- These facilities <u>must</u> be provided by the OS kernel
  - Not really possible to support "multiple competing implementations" without making the OS unusable…
- Microkernels tend to be extremely small (e.g. < 10K LOC)

Microkernels (3)

- All other facilities are provided as user-mode programs
  - Device drivers, filesystems, virtual memory pagers, etc.
- Microkernels take "separation of policy and mechanism" approach to its furthest extent
  - Generally, any component that implements a system policy should be implemented as a user-mode process
  - (Want to allow multiple competing implementations, following Liedtke's minimality principle.)
- Note: drivers often require privileged I/O port access
  - Microkernel can provide facilities for granting processes access to required I/O ports
- Note: process scheduling is often implemented in-kernel
  - A great candidate for multiple competing implementations, but want to avoid requiring context-switches to and from the scheduler...

Microkernels (4)

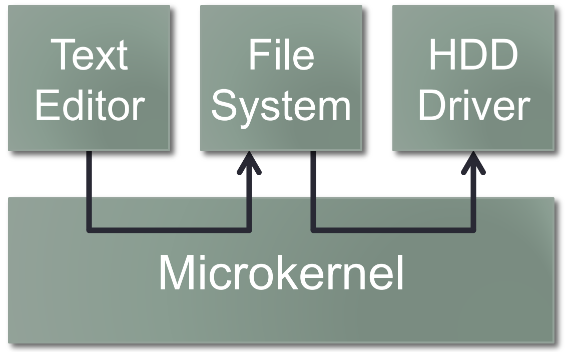
- Example: use a text editor to create and save a file
- Text editor is a user-mode process (duh)
- Filesystem implementation and HDD driver are also user-mode processes
- When text editor wants to create a file, it sends an IPC message to the filesystem service



- Filesystem service interacts with HDD driver via IPC to create the file on the physical disk
- When file has been created (or in event of failure), filesystem service sends an IPC message back to editor

Microkernels: Implementation Notes

- As indicated by previous example, microkernels allow processes to communicate via message passing
  - An asynchronous, one-way mechanism: process A sends a message to process B
  - Process B can query the kernel to receive the message
- Can be implemented with kernel-level queues for each process
  - When process A sends the message, it can be copied into the kernel queue for process B
  - When process B receives the message, it is copied from kernel queue into process B's address space
- Can provide other abstractions as well, e.g. permissions to send messages to a given process



Microkernels: Benefits

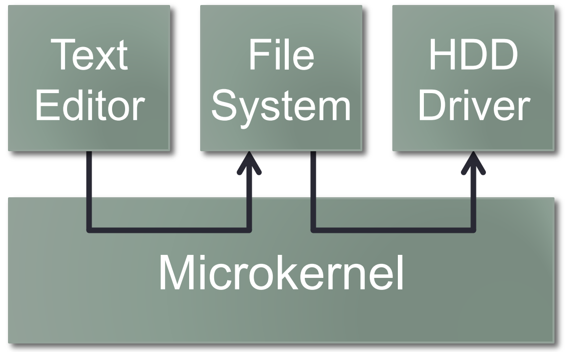
- Benefit: very reliable!
  - A very small amount of kernel code to get correct
- If a system service crashes:
  - It's a user-mode process; it won't affect overall system stability. Just restart that process.
- Caveat: reliability does not mean that state is never lost…
  - When a service crashes, it may leave inconsistent state, or it may lose state
    - e.g. a filesystem can still become corrupted
    - e.g. a connection from a remote client can be dropped
- Solution: make services more resilient
  - e.g. journaling, transacted operations, recovery, etc.
  - This is not included in the microkernel, though

Microkernels: Benefits (2)

- Another benefit of microkernels:
  - Supporting multiprocessor or multiple-computer systems becomes very easy
  - Just need to extend microkernel's IPC mechanism to support messages between processors (or between computers)
- Interestingly, can even implement non-local IPC facility with user-mode services
  - Microkernel provides primitive IPC service between local processes
  - User-mode program provides higher-level IPC mechanism between local and/or remote processes across a network
- This capability initially made microkernels <u>very</u> interesting
  - Companies were struggling to take earlier-generation single-CPU monolithic kernels and port them to multiprocessor systems

Microkernels: Drawbacks

- Microkernels have a major drawback: Performance! ☹
- Message-passing is asynchronous…
  - Text editor must trap to microkernel to send IPC message to filesystem
    - Microkernel verifies IPC arguments, then stores message in filesystem queue
  - Filesystem must trap to microkernel to receive messages



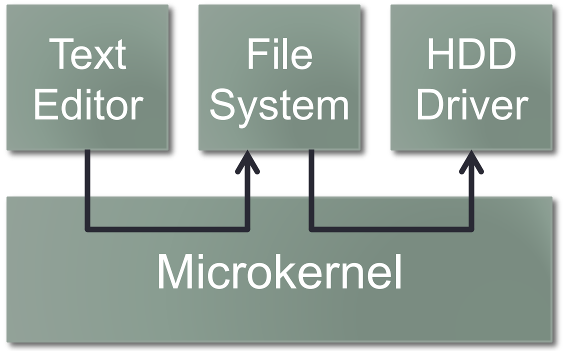
- Most interactions between services are synchronous
  - Process A makes a request to process B, and cannot progress until process B responds back to process A
  - A simple request-response interaction requires <u>four</u> system calls!
- This IPC mechanism tends to add lots of overhead ☹
  - Kernel verification of IPC call, overhead of changing hardware protection level, managing memory in kernel queues, etc.

Microkernels: CMU Mach

- Mach is a very widely known microkernel OS
  - Carnegie Mellon University project from 1985 through 1994
- Implemented on top of BSD UNIX:
  - Mach started out as a set of IPC extensions to the BSD kernel
  - Allowed the OS to be used and tested during development
  - After IPC mechanism was completed, key components of BSD OS were migrated to using the new IPC mechanism one by one
  - Finally, OS facilities were migrated out of the kernel into user space
  - Mach 3.0 was first version that was truly a microkernel OS
- Mach initially gained extensive interest, due to its ability to run on multiprocessor and multi-server systems easily
- Unfortunately, its performance was horrible ☹

Mach Inter-Process Communication

- In CMU Mach, messages are sent to **ports**: bounded queues of messages managed within the kernel
  - Processes can create ports to receive messages
  - Processes can specify permissions on ports
    - e.g. who can send them messages, how many messages can be sent
- Messages are composed of structured data:
  - Header specifying destination port, response port, message length
  - Additional data containing arrays of values, pointers to more data
- The **`mach_msg()`** system call sends and receives messages
  - May block if the port queue is currently full
  - Can be used to send, receive, or send-then-receive



Mach Inter-Process Communication (2)

- Once a message has been sent:
  - The sender must be allowed to modify the original message data without affecting the message-in-transit
  - Mach tries to avoid copying message data too many times…
- Mach 2.5:
  - Kernel maps message's virtual pages into kernel-space, marks them copy-on-write
  - If sender writes to message data after send, page is duplicated so that changes are local to the sender
  - When the message is received, kernel maps the message's virtual pages into receiver's address space
- Mach 3.0 simplified this by duplicating sender's message data into the kernel
  - On receipt, kernel copy is mapped into receiver's address space

Microkernels: CMU Mach

- Mach 3.0 revealed another drawback of microkernels:
  - Different subsystems don't always have all the details they need to make good choices
- Example: user-mode memory pager
  - When kernel needs a new page, it asks the pager to choose a victim page to evict
  - Premise: having the pager in user-space allows users to choose a pager that is appropriate for the system's usage
- Problem: the pager doesn't understand how other parts of the operating system behave
  - In low-memory situations, the Mach pager did horribly
- In a monolithic architecture, developers can tweak pager behavior based on knowledge of how rest of OS works…
  - Under Mach, the pager doesn't even know what all is part of the OS

Microkernels: L4

- Mach 4 tried to address many Mach 3 performance issues
  - Performance was still very disappointing
  - Projects tended to resolve performance issues by migrating services back into the kernel

- Jochen Liedtke began to study limitations of microkernels using inter-process communication
- Liedtke realized that the inter-process communication mechanism was doing a lot of unnecessary work…
  - Implemented L3 and L4 microkernels with completely new IPC mechanisms
  - Achieved an order-of-magnitude performance improvement (!!!)

Microkernels: L4 (2)

- L4 uses fewer system calls per IPC interaction
  - Since most IPC interactions are synchronous, why not provide a single “send and then receive” system call?
  - Reduces typical 4-syscall interaction down to just 2 system calls
- L4 eliminates a copy during message-based IPC
  - When Process A invokes kernel to pass a message to Process B, the kernel temporarily maps Process B’s target memory into Process A’s address-space
  - When the kernel copies the message from the sender, it automatically ends up in Process B’s address space
- L4 handles short and long IPC messages differently
  - Many IPC messages are very short (e.g. status/error responses)
  - Pass the message data via registers instead of using memory

Microkernels: L4 (3)

- L4 performs a *direct process switch* whenever possible
- Example:
  - Process A performs "send then receive" to Process B, which is currently blocked on receiving a message
  - Process A must wait for Process B to reply. Now A is blocked…
  - But, Process B can definitely proceed
  - *Why invoke the scheduler if we already have a process to run?*
- Produces a significant performance improvement, especially with many synchronous IPC interactions
- Also affects the OS' scheduling behavior in subtle ways
  - e.g. when used in a real-time operating system, can negatively impact real-time guarantees

Microkernels: L4 (4)

- Liedtke's work greatly revived interest in microkernels
- Problem: some of L4's IPC techniques reduce reliability benefits that microkernels are supposed to have
  - But, there are many contexts where this is actually OK
- L4 variants are now used extensively in various ways
  - Mobile devices
  - Hypervisors for managing virtual machines
- Open-source development continues
  - https://os.inf.tu-dresden.de/L4/
  - https://www.l4ka.org/

Hybrid Kernels

- Microkernels have performance issues.
- Important OS services live in user space:
  - Interactions between multiple services require context-switches
  - IPC through the kernel requires switching between kernel mode and user mode
- **Hybrid kernels** retain the same conceptual structure of microkernels, but move some services into the kernel
  - High degree of modularity, even for services within the kernel
  - Some services continue to be provided by user-mode processes, allowing for easy extension of the operating system
  - Of course, they also lose some reliability benefits of microkernels
- macOS (prev. Mac OS X) XNU kernel is a hybrid kernel
  - Based on Mach 3.0, with BSD facilities migrated into the kernel

Hybrid Kernels (2)

- Windows NT is a hybrid kernel
  - Initially, was heavily influenced by the Mach project
  - Unfortunately, performance wasn't satisfactory, so many services were migrated back into the kernel
  - NT still runs a number of services as user-mode programs
- ChorusOS and a few other microkernel OSes allowed services to be run in user mode or in kernel mode
  - A service can be run in user mode during development and testing
  - Once it is stable, it can be **co-located** into the kernel to improve its performance
  - Most microkernels that support co-location do this at compile time
  - ChorusOS could do this at compile-time or run-time

Hardware Abstractions

- Both monolithic kernels and microkernels impose an abstraction over the computer hardware
  - Only real difference is where the code that provides the abstraction actually runs
- Example: the filesystem and disk files
  - Applications generally don't care about how files are stored
  - Only interact with files via file-descriptors and various system calls (**`open`**, **`read`**, **`write`**, **`lseek`**, …)
- **Exokernels** explore a completely different approach:
  - "What if the OS didn't force us into a specific abstraction for interacting with hardware?"

Exokernels

- Concept was devised in MIT's Parallel and Distributed Operating Systems (PDOS) group in 1994-1995
- Premise:
  - Applications know better than operating systems what the goals of their resource-management decisions are
  - Therefore, give applications direct control over hardware
- Operating system provides minimal facilities necessary to <u>securely</u> multiplex hardware resources
  - It's up to applications to use the hardware however they see fit
- An exokernel might implement its OS facilities as a monolithic kernel or a microkernel
  - Being an "exokernel" simply means no abstractions are imposed
  - (Exokernels tend to be very small though.)

Exokernels (2)

- One or more “Library OSes” or libOSes run on top of the exokernel
  - Provides additional abstractions on top of the computer hardware
  - Frequently designed with a specific application’s needs in mind
- Applications run on top of a specific libOS
  - For some apps, there is a tight coupling between the application and libOS to achieve dramatically improved resource management

Exokernels (3)

- MIT created an experimental exokernel called XOS
- Implemented the ExOS libOS on top of XOS
  - Provides standard UNIX APIs for programs to use.
  - Able to run programs like emacs and gcc without any changes.
  - Applications can also ignore parts of the ExOS set of abstractions
- Created a prototype webserver called Cheetah
  - Custom filesystem management: files for a particular webpage are grouped together in the same region of the disk
  - Custom TCP/IP implementation:
    - Uses HTTP protocol state to reduce # of TCP control packets sent to clients
    - Performs TCP retransmits directly from file cache to reduce copying of data
- Result: performed 3-10x faster than comparable webservers on equivalent hardware

Exokernels (4)

- A very interesting approach to kernel structure, but still a very open area of research
- MIT Parallel / Distributed Operating Systems research on exokernels:
  - https://pdos.csail.mit.edu/archive/exo/
  - A few other exokernels have also been attempted
- Nemesis – an OS focused on multimedia processing
  - Kernel is an exokernel
  - Virtually all abstractions are implemented in the user-mode application
  - https://www.cl.cam.ac.uk/research/srg/netos/projects/archive/nemesis/

OS Design Patterns: Summary

- These design patterns generally capture the spectrum of OS design
  - Simple structure
  - Layered structure
  - Monolithic kernels, modular kernels
  - Microkernels, hybrid kernels
  - Exokernels
- Several other variants of these design patterns as well
  - Typically are blendings of the above patterns
- Note: didn’t mention hypervisors
  - Hypervisors aren’t really operating systems per se
  - Facilitate running multiple OSes on a single machine