ADVANCED OPERATING SYSTEMS

On Microkernel Construction

The UNIX Time-Sharing System

On micro-Kernel Construction (by Jochen Lietdke, 1995)

- The topic of how to structure an OS implementation is a religious debate
- Other religious debates:
  - Threads vs events
  - CATOCS (group communication)
  - Windows vs Linux
  - Unix vs Unix
  - PL vs PL
  - Others?

What kind of paper is this?

- New big idea?
- Measurement paper?
- Experiences/lessons learnt paper?
- A system description?
- Performance study?
- Refute-conventional wisdom?
- Survey paper?

What kind of paper is this?

- Refute-conventional wisdom
- Big idea (microkernels are fast enough)

Brief summary

- There are two widely held beliefs:
  - Microkernel systems are inherently inefficient
  - Microkernel systems are not sufficiently flexible
- Paper's claim: these are not correct beliefs

Microkernel concept

- What is the kernel?

Microkernel concept

- What is the kernel?
  - The part of the OS that is mandatory and common to all other software
- Microkernel concept:
  - Minimize the kernel
    - Implement outside the kernel whatever is not absolutely necessary
    - Advantages?

Paper's main message

- MYTH: microkernels are inefficient and inflexible due to
  - increased user-kernel mode switches
  - address space switches (aka context switches)
- Lietdke states
  - "maybe it's the developers at fault – poor implementation, not a poor concept"
  - If you build a microkernel correctly, then you get good performance AND flexibility
  - So what should you do?
    - Reason carefully about microkernel concepts

What must go in the microkernel?

- Only put in the kernel that which, if moved outside, would prohibit functionality
- Assumptions?
  - Require security
    - subsystems must be isolated from one another
  - Require page-based VM
  - Must be able to communicate between 2 subsystems without interference from a 3rd

What exactly should go in the kernel?

What exactly should go in the kernel?

- Address spaces
- Threads and IPC
- Unique ID (so you can identify address spaces, threads, and messages)

Address spaces

- All through inheritance
- One master address space (physical memory)
- All others are selections from this space
- Interface
  - Grant: move a page from your addr space to a new one
  - Map: share a page with another addr space
  - Flush: remove a page from someone's addr space
- This approach leaves memory mgt and paging decisions outside the kernel

Threads

- Threads execute in an address space
- Included in micro-kernel because a thread is associated with a particular addr space
  - Although association may change over time
  - Changes to state must be managed by kernel
- Thread state includes registers and addr space
- Communication among threads (IPC) must also be a microkernel feature

IPC

- Represents an agreement
  - Sender sends and receiver agrees to receive
- Interrupts are IPC messages with no payload
  - Only purpose is to supply the sender ID so that the interrupt can be associated with a particular hw device
- Kernel must turn real hw interrupts into message events
  - But kernel need NOT be involved in device-specific interrupt-handling

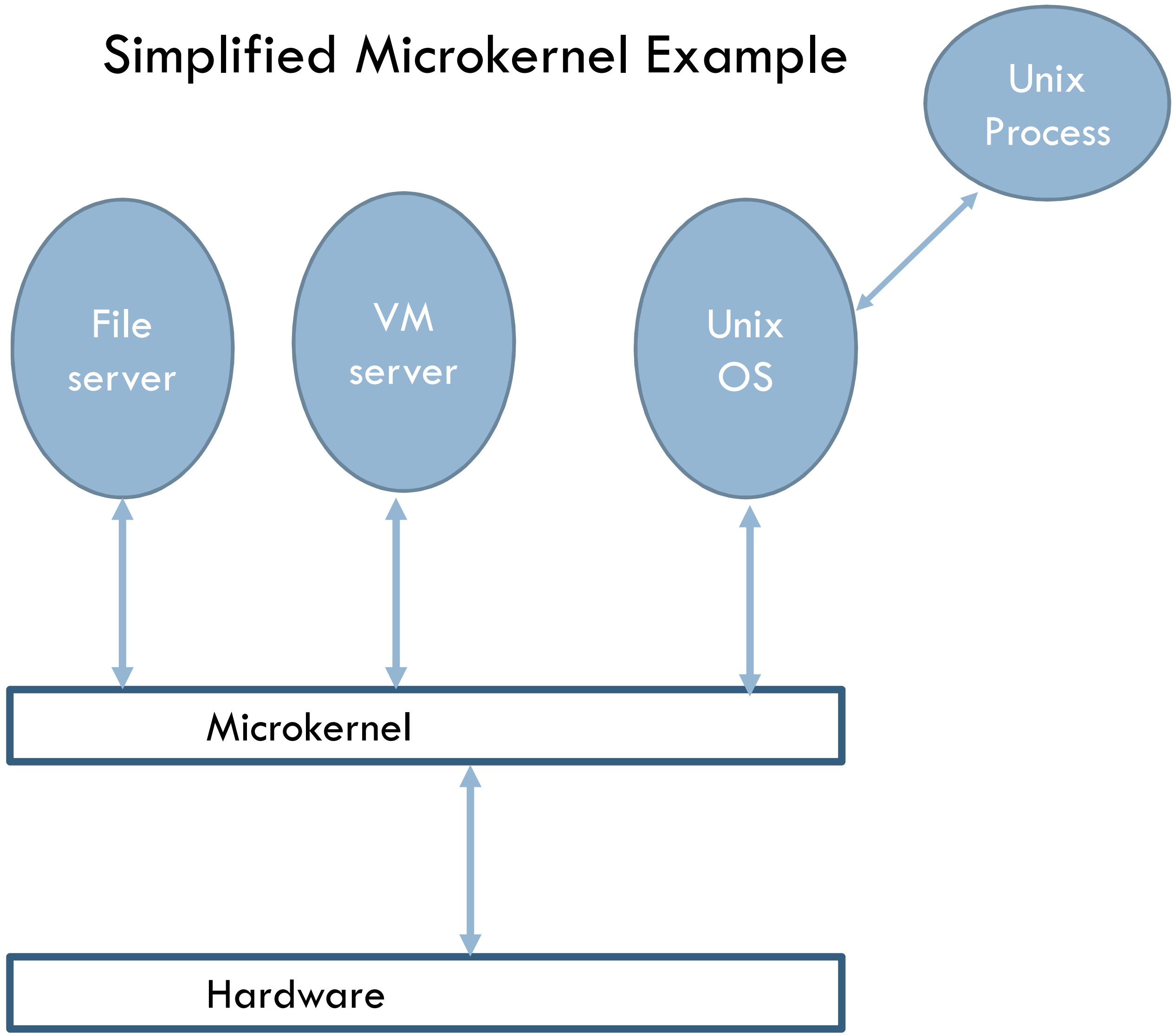
Implementing user-level services on a microkernel

- Memory manager/Pager
  - Grant, map, flush are the basic mechanisms
  - Policies for how and when you issue these calls can be made in a user-level manager
  - Each address space can have its own manager
  - Can have app-specific mgrs. (e.g., multimedia resource allocator) coordinate with other managers to make appropriate guarantees

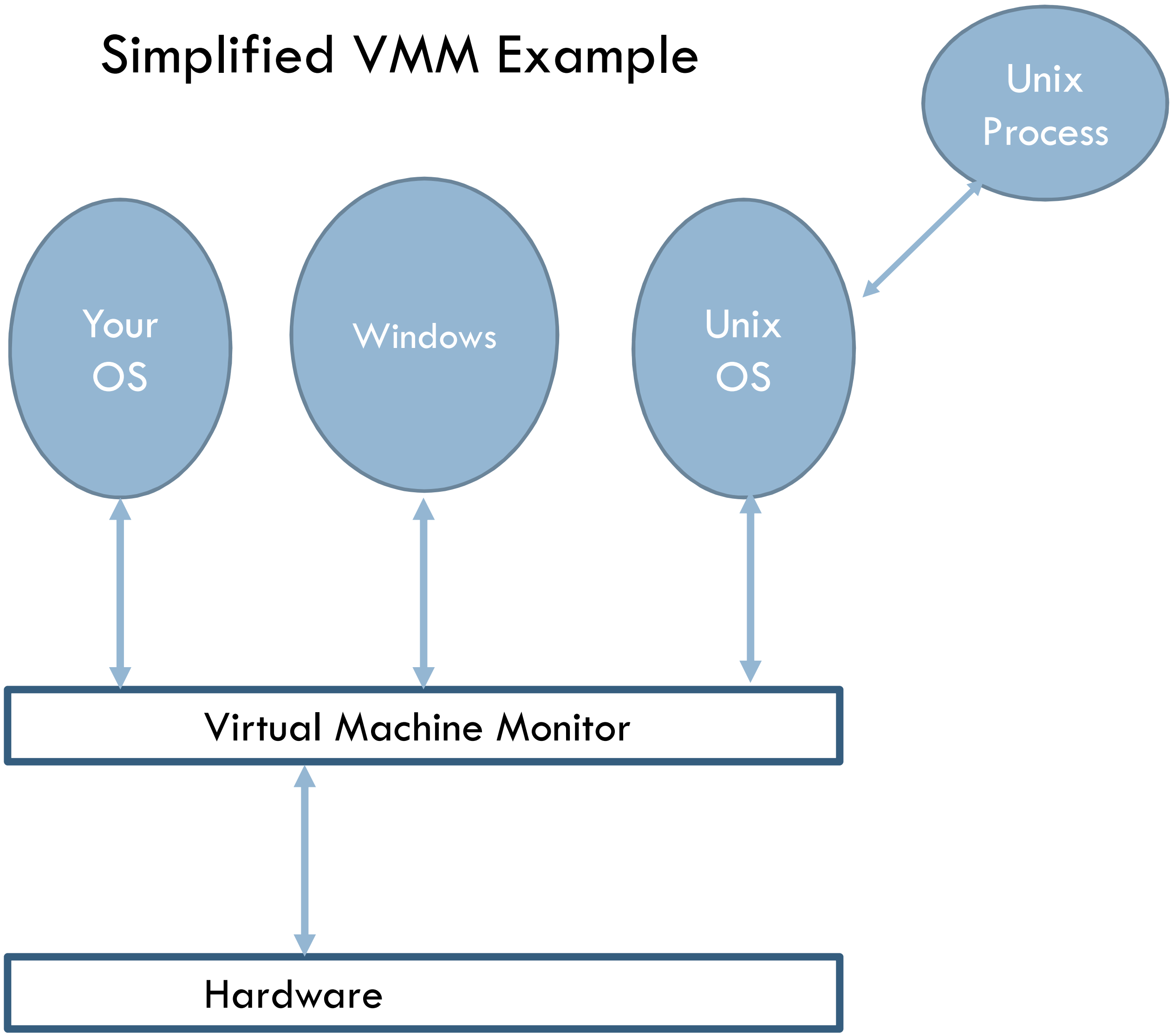
Implementing user-level services on a microkernel

- Device drivers
  - Can live outside the kernel because they don't access hw directly
    - They send/receive mesgs from the thread that represents the hw
- Cache and TLB handling
  - User pagers to implement whatever policies you like
  - In practice, 1^{st} level TLB handling still needs to be in the kernel for performance
- What about Unix server?

Simplified Microkernel Example



Simplified VMM Example



Performance

- User-kernel switches (e.g. system calls) shouldn't be that expensive
  - Conventional systems pay almost 90% of their switch time in "overhead"
  - L3 does not
  - Why?
- Similarly, context switches between address spaces shouldn't be so expensive

Context switches between addr spaces

- Similarly, context switches between addr spaces shouldn't be so expensive
  - Lietdke includes thread and addr space switching in the discussion because that's what people measure
  - If caches are physical, these don't affect context switch time
  - If TLBs are untagged, an addr space switch requires a flush of the TLB
  - Use PowerPC hacks to get rid of TLB reload problem, by use of segment registers
  - Tailor context switch code to the hw and figure out how to get it fast

Thread switches and IPC

- Empirically shows that microkernel can have fast IPC
- Graph compares multiplexing idea vs standard address space switch on L4

Memory Effects

- Chen and Bershad "showed" that microkernels had significantly worse memory behavior (higher MCPI)
- Lietke shows that the difference in memory overhead is in the cache miss behavior (system cache misses)
- Capacity or conflict misses?
  - What would each imply?
  - Ratio of conflict to capacity much lower in Mach than Ultrix
- Conclusion: problem is simply too much code in Mach; soln: reduce the cache working set of microkernel

Portability

- Microkernels should NOT be portable
  - They _are_ the hw compatibility layer
  - Must take advantage of specific hw features
  - Must take precautions to avoid problems of specific hw features
- Example: implementation decision between 486 and Pentium is different if you are going for high performance
  - Suggest significant rewriting to port from 486 to Pentium even though they are binary compatible!!!

The Unix Time-sharing system

What kind of paper is this?

- New big idea?
- Measurement paper?
- Experiences/ lessons learnt paper?
- A system description?
- Performance study?
- Refute-conventional wisdom?
- Survey paper?

The Unix Time-Sharing System

- Compare with MULTICS
  - Multics: a visionary system, but a bit ungainly
    - Proposed lots of new ideas
    - Wasn't able to fit them together harmoniously
  - Unix: craftsmanship, elegance, taste
    - A few key concepts that fit together well
    - Few new ideas, but Unix made them work

Unix

- Entire systems include UI and implementation takes 10 pages to describe
- Kernel size: then 42 kilobytes, Today many megabytes!!!
- First OS to run on "low-cost" hw
  - Only $40K
  - Interactive
  - General purpose
  - Goals: simplicity, elegance, ease of use

Unix

- What was the key service offered by Unix?

Key service of Unix: a File system

- Most of what we do can use files
- Key ideas
  - Hierarchical file system
  - Record-less I/O: a file is just an array of bytes
  - Directories just like regular file
    - only writable by the system
    - Links (name, i-number pairs) place files
      - Special links "." and ".."
  - Device-independent I/O
    - I/O devices appear as files
      - Same interface as files, same protection scheme
      - Substitute for filenames in programs

Key ideas (cont'd)

- Mount system call
  - Mount file systems on top of existing directory
  - Build file systems into single hierarchy
    - Disk (volume) structure is hidden
- Uniform I/O calls
  - Open/read/write/seek/ioctl/close
  - Bytes, no records (for simplicity – causes issues for DB developers
  - No "random" and "sequential"

Protection

- User-world, RWX bits (7 bits indicating permission for user and rest of world to access file)
- Super user is just special user id
- One new idea: set user id bit
  - Simple mechanism for rights amplification
  - Available to user programs as well

Processes and images

- Text, data, and stack segments
- Process swapping
- Process management: fork() and exec()
  - Inherit open files from parent
    - Used for shell (pipelines and redirection)
  - Separate fork and exec
    - Simple to create new processes
    - Easy to control child's initial state
      - Configure communication channels

The shell

- Users communicate with system and ask for services via the shell
- User issues command with arguments
  - Shell spawns process to execute command (via fork and exec)
  - Spawned process inherits open files (shell by default opens a file for reading and writing with fds 0 and 1, known as standard input and output files)
  - Very elegant
    - I/O redirection is transparent to the command which just uses fd 0 and 1

Filters (pipes)

- Allow combination of various tasks to be achieved through simple I/O redirection
- Otherwise, what would happen with example on p. 371?

Lessons Learned

- If you really want to build a system that works, USE it!
- Systems design comes from careful selection of design choices
- Keep it simple!

- Van Jacobson quote on Windows vs Unix

"Words and grammar vs sentences"