System design issues

- **Systems often have many goals:**
  - Performance, reliability, availability, consistency, scalability, security, versatility, modularity/simplicity
- **Designers face trade-offs:**
  - Availability vs. consistency
  - Scalability vs. reliability
  - Reliability vs. performance
  - Performance vs. modularity
  - Modularity vs. versatility
  - Latency vs. throughput

Engineering vs. research

- **Engineering:**
  - Find the right design point in the trade-off
  - Minimize cost/benefit, etc.
- **Research:**
  - Fundamentally alter the trade-offs
  - Ideally get “best of both worlds”

Example: Scheduler activations

- **Problem: Kernel-level threads suck**
  - Many expensive context switches
  - Kernel doesn’t know about application-specific priorities
- **Problem: User-level threads suck**
  - Scheduler doesn’t know which system calls block
- **Solution: New kernel interface**
  - Expose information needed by user-level scheduler: preemption, blocking system calls, I/O completion, …
  - Provides the best of both worlds
  - Facilitates other abastractions, too! (async I/O)

Example: Receive livelock

- **Problem: Interrupt-driven network stack bad**
  - Causes livelock under heavy load
- **Problem: Polling adds too much latency**
- **Solution: Re-architect kernel for best of both worlds**
  - Mogul and Ramakrishnan show several techniques
  - Hybrid interrupt/polling, queue-length feedback, CPU quotas
  - Could maybe unify some with BVT to reduce need for tuning

The end-to-end principle

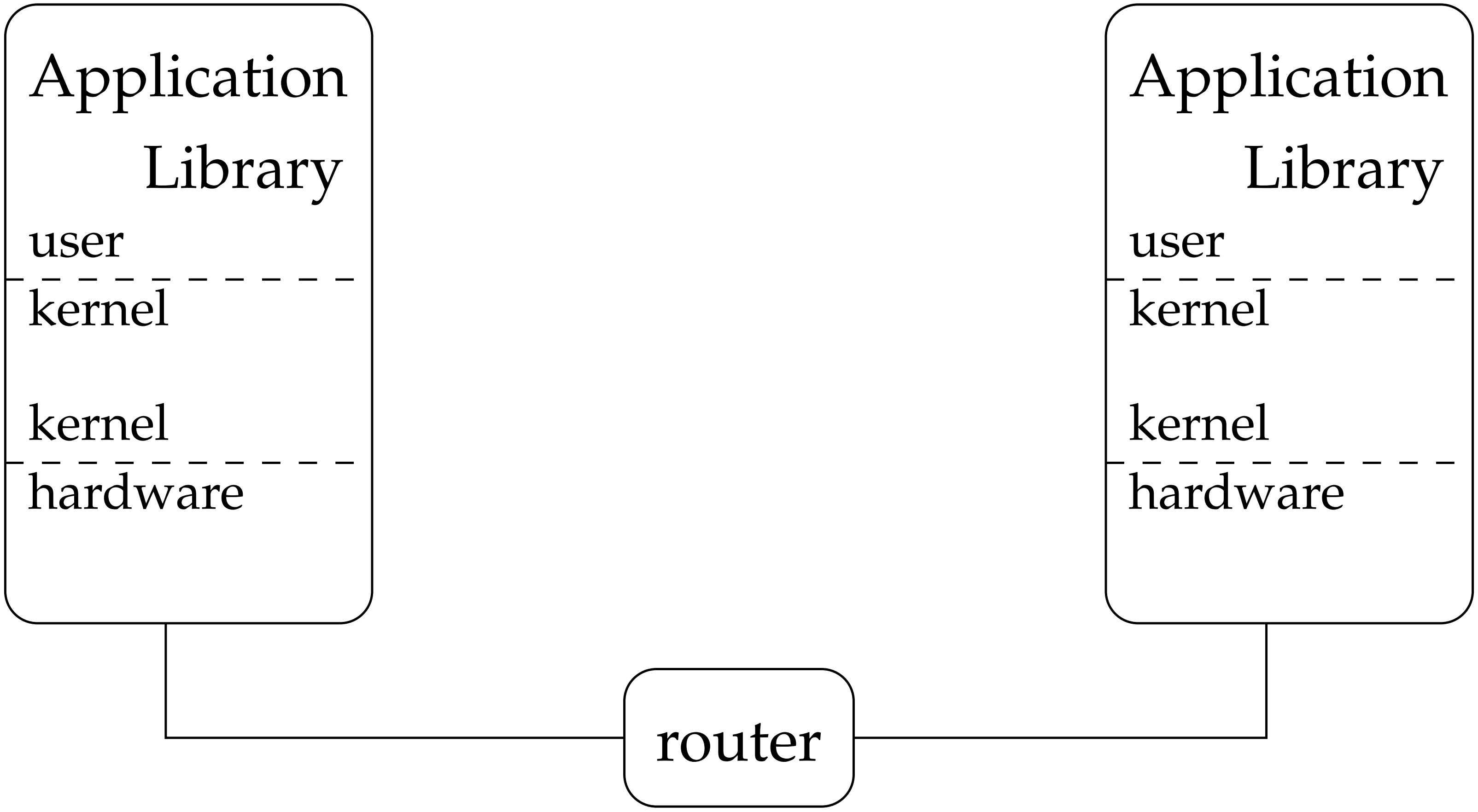
- **Seminal paper by Saltzer, Reed, and Clark**
- **Articulated principle for where to place functionality**
- **Profoundly influenced the design of TCP/IP**
  - Allowed Internet to evolve so much faster than phone network

The function in question can completely and correctly be implemented only with the knowledge and help of the application standing at the end points of the communication system. Therefore, providing that questioned function as a feature of the communication system itself is not possible. (Somtimes an incomplete version of the function provided by the communication system may be useful as a performance enhancement.)

Example applications of principle

- **Link-by-link reliable message delivery**
  - Often ensured by application (higher-level reply)
  - Can't trust every component of network
  - Inappropriate for many applications (e.g., voice over IP)
- **FIFO message delivery, duplicate suppression**
  - Redundant, just slows down two-phase commit, etc.
- **Security and data integrity checks**
  - Only make sense end-to-end

End-to-end principle generalized



- **Place functionality closer to the endpoints**
- **Often leads to insight that yields best of both worlds!**

Applying the end-to-end argument

- **Keep lower-level functionality only for performance**
  - E.g., Ethernet tries several times after a collision
  - Avoids unnecessarily triggering TCP retransmits
- **Provide “least common denominator” abstractions**
  - Can implement threads on events, but not vice versa
  - Can implement threads or events on sched. activations
  - Can implement many tricks on top of good VM primitives

Hints for low-level abstraction design

- **Expose information**
  - Lets applications/libraries make intelligent decisions
    (Is thread runnable? How much memory is available?)
- **Expose hardware and other low-level functionality**
  - Appel & Li: Exposing VM helps applications
  - Frangipani: Exploits low-level block protocol, locks
- **Avoid "outsmarting" higher-level software**
  - We still sometimes see papers on buffer cache management
  - Databases routinely bypass the buffer cache
  - Maybe OS shouldn't dictate the policy

Things to think about

- **System designers face many trade-offs**
- **When possible, gain the best of both choices**
  - Rethink layer interfaces and abstractions
  - Push functionality upwards (end-to-end priciple)
- **High-performance servers particularly demanding**
  - Often uncomfortable fit on traditional OS abstractions
- **Use “OS techniques” at application level**

[OS Security]

DAC vs. MAC

- **Most people familiar with discretionary access control (DAC)**
  - Example: Unix user-group-other permission bits
  - Might set a file `private` so only group `friends` can read it
- **Discretionary means anyone with access can propagate information:**
  - `Mail sigint@enemy.gov < private`
- **Mandatory access control**
  - Security administrator can restrict propagation
  - Abbreviated MAC (NOT a message authentication code)

Bell-Lapadula model

- **View the system as subjects accessing objects**
  - The system input is requests, the output is decisions
  - Objects can be organized in one or more hierarchies, *H* (a tree enforcing the type of decendents)
- **Four modes of access are possible:**
  - execute – no observation or alteration
  - read – observation
  - append – alteration
  - write – both observation and modification
- **The current access set, *b*, is (subj, obj, attr) tripples**
- **An access matrix *M* encodes permissible access types (subjects are rows, objects columns)**

Security levels

- **A *security level* is a (c, s) pair:**
  - $c =$ classification – E.g., unclassified, secret, top secret
  - $s =$ category-set – E.g., Nuclear, Crypto
- (c_1, s_1) ***dominates*** (c_2, s_2) **iff** $c_1 \geq c_2$ **and** $s_2 \subseteq s_1$
  - L_1 *dominates* L_2 sometimes written $L_1 \sqsupseteq L_2$ or $L_2 \sqsubseteq L_1$
  - levels then form a lattice
- **Subjects and objects are assigned security levels**
  - level(S), level(O) – security level of subject/object
  - current-level(S) – subject may operate at lower level
  - level(S) bounds current-level(S) (current-level(S) $\sqsubseteq$ level(S))
  - Since level(S) is max, sometimes called S's *clearance*

Security properties

- **The simple security or *ss-property*:**
  - For any $(S, O, A) \in b$, if A includes observation, then level(S) must dominate level(O)
  - E.g., an unclassified user cannot read a top-secret document
- **The star security or **-property*:**
  - If a subject can observe O_1 and modify O_2, then level(O_2) dominates level(O_1)
  - E.g., cannot copy top secret file into secret file
  - More precisely, given $(S, O, A) \in b$:
    if $A = r$ then current-level(S) $\sqsupseteq$ level(O) ("no read up")
    if $A = a$ then current-level(S) $\sqsubseteq$ level(O) ("no write down")
    if $A = w$ then current-level(S) $=$ level (O)

Straw man MAC implementation

- **Take an ordinary Unix system**
- **Put labels on all files and directories to track levels**
- **Each user U has a security clearance (level(U))**
- **Determine current security level dynamically**
    - When U logs in, start with lowest curent-level
    - Increase current-level as higher-level files are observed (sometimes called a *floating label* system)
    - If U's level does not dominate current, kill program
    - If program writes to file it doesn't dominate, kill it
- **Is this secure?**

No: Covert channels

- **System rife with *storage channels***
  - Low current-level process executes another program
  - New program reads sensitive file, gets high current-level
  - High program exploits covert channels to pass data to low
- **E.g., High program inherits file descriptor**
  - Can pass 4-bytes of information to low prog. in file offset
- **Labels themselves can be a storage channel**
  - Arrange to raise process p_i's label to communicate i
- **Other storage channels:**
  - Exit value, signals, terminal escape codes, …
- **If we eliminate storage channels, is system secure?**

No: Timing channels

- **Example: CPU utilization**
  - To send a 0 bit, use 100% of CPU is busy-loop
  - To send a 1 bit, sleep and relinquish CPU
  - Repeat to transfer more bits
- **Example: Resource exhaustion**
  - High prog. allocate all physical memory if bit is 1
  - If low prog. slow from paging, knows less memory available
- **More examples: Disk head position, processor cache/TLB polution, . . .**

Reducing covert channels

- **Observation: Covert channels come from sharing**
  - If you have no shared resources, no covert channels
  - Extreme example: Just use two computers
- **Problem: Sharing needed**
  - E.g., read unclassified data when preparing classified
- **Approach: Strict partitioning of resources**
  - Strictly partition and schedule resources between levels
  - Occasionally reapportion resources based on usage
  - Do so infrequently to bound leaked information
  - In general, only hope to bound bandwidth of covert channels
  - Approach still not so good if many security levels possible

Declassification

- **Sometimes need to prepare unclassified report from classified data**
- **Declassification happens outside of system**
  - Present file to security officer for downgrade
- **Job of declassification often not trivial**
  - E.g., Microsoft word saves a lot of undo information
  - This might be all the secret stuff you cut from document

Biba integrity model

- **Problem: How to protect integrity**
  - Suppose text editor gets trojaned, subtly modifies files, might mess up attack plans
- **Observation: Integrity is the converse of secrecy**
  - In secrecy, want to avoid writing less secret files
  - In integrity, want to avoid writing higher-integrity files
- **Use integrity hierarchy parallel to secrecy one**
  - Now *security level* is a (c, s, i) triple, $i =$integrity
  - Only trusted users can operate at low integrity levels
  - If you read less authentic data, your current integrity level gets raised, and you can no longer write low files

DoD Orange book

- **DoD requirements for certification of secure systems**
- **4 Divisions:**
  - D – been through certification and not secure
  - C – discretionary access control
  - B – mandatory access control
  - A – like B, but better verified design
  - Classes within divisions increasing level of security

Divisions C and D

- **Level D: Certifiably insecure**
- **Level C1: Discretionary security protection**
  - Need some DAC mechanism (user/group/other, ACLs, etc.)
  - TCB needs protection (e.g., virtual memory protection)
- **Level C2: Controlled access protection**
  - Finer-graunlarity access control
  - Need to clear memory/storage before reuse
  - Need audit facilities
- **Many OSes have C2-security packages**
  - Is, e.g., C2 Solaris “more secure” than normal Solaris?

Division B

- **B1 - Labeled Security Protection**
  - Every object and subject has a label
  - Some form of reference monitor
  - Use Bell-LaPadula model and some form of DAC
- **B2 - Structured Protection**
  - More testing, review, and validation
  - OS not just one big program (least priv. within OS)
  - Requires covert channel analysis
- **B3 - Security Domains**
  - More stringent design, w. small ref monitor
  - Audit required to detect imminent violations
  - requires security kernel + 1 or more levels *within* the OS

Division A

- **A1 – Verified Design**
  - Design must be formally verified
  - Formal model of protection system
  - Proof of its consistency
  - Formal top-level specification
  - Demonstration that the specification matches the model
  - Implementation shown informally to match specification

Limitations of Orange book

- **How to deal with floppy disks?**
- **How to deal with networking?**
- **Takes too long to certify a system**
  - People don't want to run *n*-year-old software
- **Doesn't fit non-military models very well**
- **What if you want high assurance & DAC?**