Nathan Tung

CS111 Chapter 4-5 Outline #4

Chapter 4, Section 4.1-4.1.2 (pages 147-162), Chapter 5, Section 5.1 (pages 199-210), Section 5.3-5.3.4 (pages 230-237)

- Enforced Modularity
  - Idea: errors can propagate even through modules
  - Client/service organization comprises modules that communicate by message only
    - Messages are the only way…
      - To provide a service, preventing the programmer from convention violations
      - For errors to propagate, thus allowing independent failure
      - For an attacker to penetrate module; checking messages can block attacks
  - Effective implementation: run client/service module on separate computers with a communication wire
    - Prevents errors from propagating
- Client/Service Organization
  - Soft modularity limits interactions of certain modules to their interfaces, but implementation errors might bypass the interfaces (example: caller and callee use the same address space and stack)
  - Need hard boundaries between modules (client/server: sweeping simplification regarding interfaces)
    - Errors propagate only via messages
    - Errors can be found via message checking
  - MEASURE and GET_TIME example of soft modularity
    - In MEASURE, caller and callee don't have a full contract that addresses errors
      - They modify shared arguments and their own variables in stack; callee could potentially corrupt caller's area of the stack (since the caller leaves the stack pointer where it is)
      - Callee returns to where caller specifies (even if it makes a mistake)
      - Callee stores values in register R0; if it's misplaced, the caller will grab a wrong value
      - If callee divides by zero, the caller might terminate (known as "fate sharing")
      - Caller and callee are only allowed to modify global variables shared between them (otherwise, other modules might fail in computation or crash)
  - Soft modularity is found in specifications, but it is not enforced
  - Enforced modularity by an external mechanism is desired; it limits the user of interactions
    - Benefitted with a high-level language due to compiler/runtime system (Java, C#)
    - Other shortcomings that demonstrate the importance of client/service
      - Different programming languages in modules
      - Implementation error in interpreter can lead to incorrectly-restored registers, etc.
- Client/Service Components
  - Client: module that initiates request (builds "request" message containing data to send to service)
  - Service: modules that responds (takes arguments from message, executes operations, sends back "response/reply" message, waits for next request)
  - Designers use "message timing diagrams" to represent interactions
  - Two separate computers (can be geographically remote)
    - Can avoid absolute failure during power outages; costs more
    - A service of multiple computers is called a server (more fault tolerance)
  - Marshaling: the conversion of arguments between the client and service computer(s) (like endian-ness)
    - Unmarshaling converts it back to a language object
  - Client/service organization separates functions (abstraction) and enforces it (enforced modularity)
  - Benefits:
    - No shared state other than messages
    - Transaction is at arm's length; arguments can be marshaled and thus checked
    - Client protects itself from even failing services by ignoring infinite loops after a set time
    - Encourages explicit, well-defined interfaces (messages that are specific)
    - If the service has a disaster, the client has a controlled problem (only problem is lack of result)
    - Sweeping simplification: only messages are passed (and essentially between a firewall)
  - Detriments:
    - Service might still return an erroneous result

        - System designer and client need a good interface for catching certain problems
    - To use or not to use?
        - Tradeoff between ease of data access and ease of error propagation
        - Client and service units become separate programs that are harder to manipulate shared data
        - Determine plan for recovery such that fate sharing is reduced
        - Example: the world wide web (browser is client, site is service)

- Enforcing Modularity with Virtualization
    - Client/service: enforces modularity to prevent errors, achieves security, better fault tolerance
        - Unfortunately, uses a computer for each module
        - Virtualization with three new abstractions
            - SEND and RECEIVE using bounded buffers
            - Virtual memory
            - Threads
    - Client/Service Organization via Virtualization
        - Use one physical computer to run multiple virtual computers, each running a module
        - Virtualization
            - Multiplexing: program simulates interface by multiplexing a physical instance into many virtual objects
                - Example: server into web sites, processors into threads
            - Aggregating: program simulates interface by aggregating many physical instances into a large virtual object
                - Example: disks into RAID, communication channels into bonding
            - Emulation: implement virtual object from a different kind of physical object
                - Example: disk to RAM, Mac to virtual PC
        - Virtual computers don't enforce modularity as well (less fault tolerance, i.e. to power failures)
    - Threads (virtualized processors)
        - Abstraction that encapsulates execution state of an active computation
            - Reference to next program step
            - References to environment
        - Threads can be stopped and restarted later; used for virtualizing processors
        - One thread (serial executions) follows the principle of least astonishment
        - More than one thread (say, per device) allows concurrent operations
        - Thread abstraction implemented by thread manager
            - Multiplexing multiple threads on limited physical processors to prevent interference
        - Threads allow interrupts to be processed concurrently
            - Exceptions are interrupts on current thread
    - Virtual memory
        - Sharing memory has benefits, but can be easy to make mistakes
        - Threads of one module cannot overwrite another module's data
            - Virtual memory gives each module its own virtual address space and virtual addresses
            - These are controlled by virtual memory manager
    - SEND/RECEIVE with bounded buffers (virtualized communication links)
        - Calling SEND fills buffer until it is full; then it waits until the bounded buffer has more space
        - Calling RECEIVE waits (the calling thread waits) if the buffer is empty
    - OS Interface
        - Memory
            - Create/delete address space
            - Allocate/free block
        - Interpreter
            - Allocate/delete/exit thread
            - Yield, await, advance, etc
        - Communication
            - Allocate/deallocate bounded buffer

    - Send/receive
  - Emulation and Virtual Machines
    - Emulation simulates physical hardware so that the emulated hardware can run the same software the physical hardware can run
    - Microcode interpreter inside a processor can simulate instructions of other processors/instructions from other processors
      - Allows software development before chip is manufactured
    - Emulation is slow because interpreting emulated machine instructions has substantial overhead
      - Costs up to a factor of 10 in performance
    - Fast emulation uses virtual machines; physical processor implements many virtual instances
      - Less portability
      - Better performance

- Enforcing Modularity in Memory
  - Bounded buffers take advantage of threads sharing same physical memory
    - Enforced modularity means threads of one module cannot overwrite another module's data
  - Enforcing Modularity with Domains
    - Restrict thread references to a domain, or contiguous range of memory addresses
    - Each processor has a domain register to restrict memory references to that domain
    - Memory manager (a special interpreter) checks that each memory reference is between the range within the domain register
      - Otherwise, triggers a memory reference exception
    - Shortcomings:
      - Threads might need more than one domain
      - Threads should not be able to change its own domain
- Controlled Sharing Using Several Domains
  - Allow threads to have several domains; processors have several domain registers
  - Multiple threads can use shared bounded buffer domain, but not other (private domain) references
  - Interface:
    - base_address ← ALLOCATE_DOMAIN(size): allocate new domain, return base address
    - MAP_DOMAIN(base_address): add domain starting at base_address to thread's domains
    - Permissions can allow more controlled sharing
      - MAP_DOMAIN(base_address, permission)
      - Load bounds from domain table into calling thread's registers with specified permission
      - LOAD requires READ permission; STORE requires WRITE permissions
      - Must be within domain with permissions
    - For domain register's the manager calls CHECK_DOMAIN (with parameters: address request, permissions needed, domain register) to determine whether to issue the request
  - Demultiplexing memory reference exceptions can be hardware or software
    - Hardware: illegal memory reference/permission error exception
    - Software: memory manager signals memory reference exception
  - Only some permission combinations are possible: R, R/W, R/E, R/W/E
- More Enforced Modularity with Kernel and User Mode
  - Modify the processor to prevent threads from changing its domains (low, high, permission fields)
    - Add a bit to the processor to indicate kernel mode vs. user mode
      - Only kernel mode can change domain register
    - Extend set of permissions for domain to KERNEL-ONLY, and make KERNEL-ONLY permission levels return an exception when in user mode
    - Use handler to process interrupts/exceptions while switching to kernel mode
- Gates and Changing Modes
  - Threads cannot invoke procedures to change between modes in a controlled manner
  - Gates: certain addresses where a thread can enter kernel domain
  - Supervisor call instruction (SVC)
    - Changes processor mode from user to kernel

        - Sets an address (into PC) for entry point of the gate manager (to facilitate mode change)
    - Gate names are generally numbers; gate manager records gate numbers with their procedures
    - SVC instructions must happen all without interruption in order to change modes successfully
        - Processors don't have to perform these as before-or-after actions
    - SVC can then return to user mode
        - Changes mode from kernel to user
        - Load program counter from top of stack into PC
    - Entering vs. leaving kernel mode
        - Leaving: value loaded in the program counter isn't predefined; kernel sets to saved address
    - Special gates also exist to handle interrupts and exceptions