Virtualization

How can we execute independent modules on the same hardware?

Learning Objectives:

- Examine the distinction between a physical and a virtual resources
- Multiplexing, Aggregation, and Emulation
- Introduce virtualized fundamental abstractions of computer systems
- Cover time multiplexing

Motivation [10 Min]:

1. Client-Server (or, enforced modularity) with each module on its own machine has:
    a. Great isolation!
    b. Very poor utilization!
2. What if we instead naively deploy modules on the same machine?
    a. m1 and m2 coordinate:
        i. Manually coordinate processor time (somehow)
        ii. Must split RAM manually
    b. All of this is insanely complex and dangerous given bugs, security!
3. Tradeoffs seem to be Simplicity vs. Utilization and Isolation vs. Utilization.
4. Can we somehow get the best of both worlds?
5. Virtualization:
    a. Each module gets the *abstraction* of its own computer (like client server)
    b. Each module runs on the same hardware (like naive deployment)

Virtualization Approaches [10 Min]:

1. Multiplexing: Enable multiple users the "abstraction" of a complete resource
2. Aggregation: Combine multiple resources into a single abstraction
3. Emulation: Take a resource of one type and make it look like another
4. Sometimes, you use more than one at once:

Virtualization of Fundamental Abstractions of Computer Systems [20 Min]:

1. Interpreters: Virtualized as threads
    a. Thread: The execution state of a processor or interpreter
    b. Must include sufficient state to be started/stopped

c. Example of… Multiplexing!
d. Note: threads can be layered:

2. Memory: virtualized as virtual memory (BOO bad name)
    a. Virtual memory: give each module its own virtual address space
    b. Each module can pretend like it can address all of the bytes!
    c. Each module can use the SAME ADDRESS!
        i. Fun trick to make program start work well, and older non-PIC binaries
3. If we put a thread and a virtual memory together, we get a **process**
4. Communication Link: virtualized as bonded buffers
    a. Bounded Buffer: constant sized buffers created by low-level of OS stack
    b. Enables cross thread communication.
    c. Q: What hardware resources are we virtualizing?
5. These are often layered:
    a. Threads: OS, Language, App, coroutine, promises, etc.
    b. Memory: OS (mmap), Language (malloc)
    c. We'll discuss these abstractions at the Operating System layer.
    d. We won't cover a lot about Operating Systems, though.

Threads and Thread Managers[30 Min]:

1. What state is required for enabling start/stop?
    a. Environment reference (stack, registers)
    b. Instruction Reference (program counter)
    c. Pointer to which virtual memory address space (AS_ID).
    d. Threads can share memory! But each have separate stack and registers
2. Notion of processor layer and thread layer
3. What happens if we have more threads than processors?
    a. Time Multiplexing: split up timeline into epochs and run each thread for an epoch
    b. Separate epochs by having threads call yield
    c. How does Yield work?

```
// Yield on a single processor
#define MAX_THREADS xxx
#define RUNNABLE 0
#define RUNNING 1

static int current_id;
Typedef struct {
    int state;
    int stack;
    int AS_ID;
} Thread;
static Table TTable[MAX_THREADS];

void yield() {
    // Save current running thread
    TTable[current_id].state = RUNNABLE;
    TTable[current_id].stack = SP; // register of current stack
    TTable[current_id].AS_ID = AS_ID; //register of current AS

    //Choose a thread such that state == RUNNABLE.
    Int new_id = scheduling_algorithm();

    //update state in TTable and in registers.
    TTable[new_id].state = RUNNING;
    AS_ID = TTable[new_id].AS_ID;
    SP = TTable[new_id].stack;
    current_id = new_id;

    return;
}
```

4. Thread Manager:
    a. How do we create a thread?
        i. Get index of empty entry in TTable
        ii. Allocate new stack in AS_ID
        iii. Assign Values in TTable
        iv. Return index
    b. Thread creation forms a "lineage tree" of parent-child relationships
    c. How do we kill a thread? ("Parents kill their children..")
        i. Deallocate Stack.
        ii. Reset entry in TTable to be 0.
    d. How do we exit a thread?
        i. How do you deallocate a stack that you are using right now!!
        ii. Where do you put your return code?
        iii. Solution: Defer cleanup!
            1. Mark the thread for later cleanup in TTable (called a zombie)
            2. Parent "reaps" zombie to cleanup and get return code
            3. Parentless zombies (i.e. Orphan Zombies) reaped during future thread manager call
5. Concerns:
    a. What about interrupts?
        i. Each interrupt initiates an interrupt handler that operates at processor layer:
            1. Interrupts might be intended for the current thread at thread layer (e.g., divide by 0). Pass them along to "exception handler".
            2. Interrupts might be intended for a different thread (e.g., a disk interrupt). Make a note of it for the next time that thread executes.
    b. Is yield the only time you switch? [What about other "waiting" systemcalls? (e.g., write()?]
        i. Allow context switch (or, yield logic) after almost any systemcall!
    c. What if a thread never gets off the processor?
        i. Use a Clock Interrupt!
    d. Which Thread should you switch to?
        i. Super well explored, still active area of research? Generally, you try to be "fair"
    e. What happens if you have multiple processors? Stay tuned to find out...

Virtual Memory :

1. Virtual memory–give a separate memory to each module

    a. Each “memory” named with a special virtual address space
2. Space Multiplexing: split space into chunks, assign each chunk to a module:
    a. A “chunk” is a page, a fixed-size (4096, usually) region of memory
3. Page Table:
    a. Mapping from a virtual address space’s pages to physical pages.
    b. Separate page table for each module
    c. Managed by operating system using privileged instructions to prevent each module from direct access
4. Challenges:
    a. There are many (one per process) large (Ms of entries) Page Tables!
        i. Solution: Many entries will be empty!
        ii. Solution: Use multiple Levels!
    b. Who does the translation?
        i. Very slow to context switch to go to operating system for each memory access!
        ii. There are now three accesses for each memory access (the outer page table, inner page table, and the actual physical address)!
        iii. Solution: hardware cache called a Translation Lookaside Buffer (TLB).
        iv. Solution: Store current “page table” in a register, make updating register a privileged instruction
    c. How do we create Address Spaces?
        i. In Unix, Address space lifetime is tied to a thread’s lifetime through the process abstraction
        ii. Cannot create an address space without creating a thread.
        iii. We use fork() to create a thread, which copies all of the content of its parent.