Nathan Tung

CS111 Chapter 5.5 Outline #5

- Virtualizing Processors Using Threads
    - Each thread does NOT necessarily have its own processor
    - Thread manager allows sharing of processors
        - One slow thread might slow down the progress of other threads on same processor
- Sharing procedure among multiple threads
    - Thread: abstraction that encapsulates the state of a running module
    - thread_id ← ALLOCATE_THREAD(starting_procedure, address_space_id)
        - Not enough space results in an error
        - Allows many threads to share a limited amount of processors
        - Solution depends on the fact that most threads spend most of their lives waiting for events
    - Virtualizing the processor
        - When thread is waiting, processor switches to another thread by saving the previous state and loading the new thread state
        - Also known as time-sharing, processor multiplexing, etc.
        - YIELD: entry point to thread manager, allows sender to add new item (thread?) to buffer
            - Thread manager hands calling thread's processor to some other thread
            - RUNNING (executing on processor) vs. RUNNABLE (ready to run, waiting for processor)
            - Three steps
                - Save: save thread state to resume later
                - Schedule: schedule other thread to run on current processor
                - Dispatch: dispatch processor to thread
    - YIELD procedure:
        - Thread running in thread layer calls YIELD
        - YIELD enters processor layer, which saves its current state
        - When processor later exits processor layer, it runs new thread
        - New thread might run in the same or different address space
    - YIELD is usually written in low-level instructions as a kernel procedure via supervisor call
    - Interrupts invoke interrupt handler, which always runs in processor layer
        - Does not invoke procedures like YIELD
        - No relation to current thread
    - Exceptions occur in thread layer, accessing interrupted thread's state
        - Invokes procedures
        - Specific to current thread
- Implementing YIELD
    - Relies on GET_THREAD_ID, ENTER_PROCESSOR_LAYER, EXIT_PROCESSOR_LAYER, SCHEDULER
    - Involves processor_table (info on processor) and thread_table (array with one entry per thread)
    - Steps
        - Virtualize register CPUID to create virtual ID register for each thread
        - YIELD calls ENTER_PROCESSOR_LAYER, releasing predecessor
            - Stack pointer is stored
        - Scheduling searches until it finds a runnable thread
        - EXIT_PROCESSOR_LAYER loads saved stack pointer from ENTER
    - Flow of control results in abandoning the two stack frames allocated to SCHEDULER and EXIT
    - Thread releases processor by ENTER call, then resumes immediately after the same ENTER call
        - Known as a "co-routine"
    - Enforcing atomicity: ENTER thread acquires thread_table_lock; EXIT thread releases it
- Creating and terminating threads
    - Supporting management of a variable number of threads requires
        - EXIT_THREAD(): destroys and cleans up calling thread to release its state
        - DESTROY_THREAD(id): destroy specified id thread; a thread might need to terminate another
    - At least as many threads as processors need to be supported, so a processor has a separate thread
        - Known as processor-layer thread, or processor thread

        - Runs SCHEDULER procedure
        - Each processor needs its own processor thread to deallocate thread layer thread stack spaces
    - ALLOCATE_THREAD
        - Allocate memory space for new thread
        - Place empty frame on new stack with EXIT_THREAD as return address
        - Place second empty frame with starting_procedure return address
        - Find FREE entry in thread table
        - Incompletion of these steps returns an error
    - DESTROY_THREAD
        - Calling thread cannot free target stack; only processor running that thread can do that
            - Set kill_or_continue variable to KILL
            - Processor thread checks variable during YIELD, possibly releasing resources
- Enforcing modularity with threads: preemptive scheduling
    - Non-preemptive scheduling: thread runs until it gives up processor
        - New thread that called YIELD is then run
        - Problematic: that thread might never give up control
    - Cooperative scheduling (cooperative multitasking)
        - Every thread will call YIELD occasionally
        - Not robust; programmer might forget to put in YIELD, or program errors might arise
    - Preemptive scheduling
        - Thread manager can force a thread to give up the processor based on time (ex: 100 ms)
        - Clock might trigger an interrupt to switch processor layer to kernel mode
            - Interrupt handler invokes exception handler, forcing current thread to YIELD
        - Thread manager needs to allow interrupt handler to invoke procedures
            - But concurrent execution within processor layer with calling YIELD can cause deadlock
            - Thread layer has its own lock mechanism
            - One solution: enable/disable interrupts
                - Make both streams separate (before-or-after actions)
        - Preemptive scheduling isolates thread behavior, making it like each thread has its own process
- Enforcing modularity with threads and address spaces
    - Sharing a single address space between threads can cause accidental memory changes
    - Thread manager needs to switch address spaces when threads are switched
    - One solution: map instructions and data of thread manager into same virtual addresses in every virtual address space
    - Another solution: use hardware to load PMAR, SP, PC as single before-or-after action
        - Returns control to thread in new virtual address space at saved location with saved pointer
- Layering threads
    - Create several threads in thread layer from a single thread in processor layer
    - Interrupt support
        - Processor as a hard-wired thread manager with a processor thread (for SCHEDULER) and interrupt thread (runs interrupt handlers in kernel mode, capable of YIELD)
    - OS layer uses processor threads of processor layer to make a second layer of threads
        - Each application module is given at least one preemptively scheduled virtual processor
        - Each application module can implement its own third-layer thread manager
            - Usually nonpreemptive
            - Threads don’t need to be protected, as they belong in the same application module
    - Hardware processor at lowest layer creates a processor thread and an interrupt thread