Nathan Tung

CS111 Chapter 5.2.5, 5.6 Outline #9

Chapter 5, Section 5.2.5 (pages 221-222), Chapter 5, Section 5.6 (pages 273-284)

- Deadlock
  - Situations just as bad as race conditions
  - Interaction among a group of threads where each thread is waiting for other threads to make progress
  - Wait-for graph can represent deadlocks
    - Nodes are threads/resources (ex: locks)
    - Directed edges go from lock node to thread node (when thread acquires a lock)
    - IFF cycle exists, then threads are deadlocked
  - Preventing deadlocks
    - Enumerate lock usages to ensure all program threads acquire locks in same order
    - Deadlock might also occur when a sender forgets to release/acquire a single lock
      - If buffer is full, receiver will never remove a message from buffer as it can't acquire a lock from the sender, who might in turn be waiting for the receiver to change p.out
      - Avoid simple programming errors
  - Similarly, livelock occurs when a thread repeatedly performs operations but never finishes the sequence
- Thread Primitives for Sequence Coordination
  - Thread manager allows threads to share processors
  - Threads can also release processors for other threads to run (much like using sender/receiver YIELD)
  - Polling: thread continually tests a shared variable (in, out) when sender/receiver is scheduled again
    - Undesirable; if test fails, the processor has been acquired/released unnecessarily (unproductive)
  - Get thread manager to schedule only threads with useful work to perform? Sequence coordination
- Lost Notification Problem
  - Consider incorrect sender/receiver implementation
    - Includes a shared variable and WAIT and NOTIFY (primitives taking in shared variable)
    - Race condition occurs; notification might be lost when receiver hasn't called WAIT, but when receiver calls WAIT and actually waits, nothing comes
      - Sender continues adding to buffer then WAITs; now both sender/receiver are waiting
  - Operations for testing buffer room, sleep until room, and release shared lock must be before-or-after
    - Can't just lock around operations: who actually owns the lock to create before-or-after actions?
    - Kernel correctness can't depend on user program to set/release lock, nor on user lock's correct implementation
    - Invariant is needed to act as a relation between application-owned and system-owned state
- Avoiding lost notification problem with eventcounts and sequencers
  - Designers have suggested using new thread state to characterize event for which thread is waiting under protection of table lock; use event_name to avoid lost notifications
    - Condition variable: holds additional thread state
  - Here we use eventcounts and sequencers, variables to hold additional thread states
  - AWAIT, ADVANCE, TICKET, READ (before-or-after actions)
    - Since AWAIT and ADVANCE are before-or-after, problem doesn't occur
    - Implementing these requires a spinning lock to prevent ACQUIRE from calling AWAIT for an unlocked thread (thread manager is not designed for recursion – no base case)
- Polling, Interrupts, and Sequence Coordination
  - Certain threads need to interact with external devices (ex: keyboard manager → keyboard controller)
  - Flip-flop between controller and manager allows activities to be coordinated
    - Keyboard controller sets flip-flop; keyboard manager reads and tests it
      - If it's not set/reads 0, manager yields; if it's set, falls out of loop
      - After reading the flip-flop, it resets to 0 as a side-effect, providing a coordination lock
  - Keyboard manager thread also performs polling, continually checking scheduler for things to run
  - Disadvantages of polling
    - Difficult to predict time until event, so how long should thread poll?
    - Frequent polling thread wastes processor cycles due to lack of user input
    - Devices might require processor to execute mangers by a deadline

        - Ex: keyboard controller may have a single keystroke to communicate with manager
            - A second keystroke done before the first is absorbed might lose the first
    - Similar disadvantages to not using explicit primitives (like AWAIT, ADVANCE)
        - Thread scheduler doesn't know when to run receiver, which might call extra YIELDs
        - When controller has input to be processed, scheduler should be alerted for keyboard manager
    - Instead, move polling loop into hardware by using interrupts
        - Keyboard manager enables interrupts (sets processor's interrupt control register to ON) so that processor must take interrupts from controller
        - When processor sees change in flip-flop, it executes interrupt handler
        - Multiple shared flip-flops and an associated map for different handlers allows many interrupts
    - Interrupt handler gets processor control within one instruction, so it can meet deadlines
        - Handler could copy keystrokes to a buffer belonging to manager immediately rather than waiting for the chance of it running
    - Interrupt handler must carefully read/write shared variables, as it might be called between instructions
        - It can't be share of the thread state currently running on processor
    - Interrupt handler must be carefully programmed, since it isn't managed by OS
        - Might cause deadlocks
    - Exception handlers are less severe, since handler runs based on the current running thread