Operating Systems Synchronization Issues

CS239
Advanced Operating Systems
October 19, 2000
Dr. Richard Guy

Outline

- Motivation & Challenge
- Sources of Concurrency
- Synchronization Overview
- Intra-kernel synchronization study

Lecture Source Materials

- Singhal, ch 2: Synchronization Mechanisms
- Singhal, ch 3: Process Deadlocks
- Silberschatz & Galvin
  – Ch. 6: Process Synchronization
  – Ch. 7: Deadlocks
- Small and Manley, A Revisitation of Kernel Synchronization Schemes, USENIX 1997.

Motivation

- Computer system resources are commonly shared by multiple entities
  – serially, disjoint over time
  – concurrently, at the “same time”
- If concurrency is even a remote possibility, a concern for maintaining “consistent” or “correct” behavior arises
  – joint interference of concurrent activities

Challenge

- Understand concurrency
  – sources
  – types
- Identify effective abstract solutions
  – synchronization techniques
- Design/implement mechanisms
  – correct, sufficient, efficient

Sources of Concurrency

- Multiprocessors
- Uniprocessors
  – hardware
  – software

Multiprocessors

- Physical concurrency of process execution
- Main issue is concurrent access to shared memory regions
- A function of atomicity defined by memory management hardware
  - – atomicity is typically a single byte/word
  - – access to any other quantity may be a problem
- Delay further discussion until Ch. 17

Uniprocessors

- Hardware concurrency
  - – not the CPU by itself
  - – not programmed I/O-type components
  - – memory access by the CPU and DMA from
    - disk
    - network interface
  - – CPU doesn't act on it's own behalf...

Uniprocessors, continued

- Software concurrency
  - – logical (virtual) concurrent execution of
    - process(es)
    - kernel thread(s)
  - – causes of concurrent execution
    - time-sharing
    - asynchronous interrupt handling

Processes and Timesharing

- Multiple processes using shared memory
- Context switch can occur at *any* time in a process' user-mode execution
  - – system call execution usually non-preemptive
- Apparent atomicity of high-level language statements is often just that, and not actual
- Without additional measures, unintended interaction may occur

Processes and Interrupts

- Asynchronous hardware interrupt handling diverts CPU's attention away from process execution
- ISR may want to access memory belonging to "currently executing" process
  - – reading partially updated region(s)
  - – writing region(s) in midst of process read/write

Kernel thread(s) and interrupts

- Currently executing system call thread is interrupted
- ISR may want to read/write kernel data structure in use by "current" syscall thread
- If kernel is itself multithreaded, situation is similar to multiple processes and shared memory
  - – traditional kernel is single-threaded...

Synchronization Overview

- Concepts
- Mechanisms
- Deadlocks
- OS role and issues

Synchronization Concepts

- Atomicity
  - behavior of entity (usually data structure) is indivisible from perspective of outside observer
- *Mutual exclusion* is one way to ensure atomicity
  - only allow serial usage of entity
- *Critical section* is code segment that accessed shared resource

Mutual Exclusion

- Effective ME solution must guarantee
  - only one process executing in critical section
  - if no process currently in CS, any process that wants to be in CS experiences no delay
  - if >1 processes competing for entry to CS, can't wait indefinitely to select a process
  - no process can prevent another from entering CS indefinitely (no starvation/livelock)

Mutual Exclusion Mechanisms

- Busy waiting (spin lock)
  - wastes cycles in uniprocessor
- Disabling interrupts
  - doesn't work for multiprocessor
  - may miss important I/O events
- Test-and-Set or Compare-and-Swap
  - atomic hardware implementation of two higher-level steps; common in multiprocessors

ME mechanisms, continued

- Semaphores
  - high-level construct useful in ME (and other synchronization problem solutions)
  - integer variable with two defined atomic operations, wait and signal
  - atomicity achieved by implementing either in hardware (via test-and-set) or in software (disabling interrupts)

Semaphore downsides

- Semaphore participants need to understand each other's intended usage of semaphore
- Requires careful usage to avoid
  - inconsistency
  - deadlock
- Difficult to formally verify correctness of programs that use semaphores

Other Synchronization Problems (besides ME)

- Dining Philosophers
  - multiple resource allocation
- Producer-Consumer
  - shared buffer access (put/get by multiple users)
- Readers-Writers
  - exclusive write access
  - priority choices to be made

Language-based Mechanisms

- Motivation: avoid semaphore pitfalls
- Examples
  - Monitors
  - Serializers
  - Path-expressions
  - Communicating Sequential Processes
  - Ada rendevous

Deadlocks

- Occurs when set of entities is blocked, waiting for event that will never occur
- Requires four conditions to hold
  - exlusive access
  - hold-and-wait
  - non-preemptive
  - circular wait

Deadlock Handling Strategies

- Prevention
  - grant resource only if <4 conditions hold
- Detection
  - no a priori effort on resource granting
  - often significant algorithmic costs to analyze
- Avoidance
  - examine potential resource requests and grant only if a non-deadlocking execution exists

OS Role and Issues

- Generally provides basic synchronization tools to processes
  - locks
  - semaphores
  - deadlock support
  - relatively expensive to access (system call)
- Must be sensitive to intra-kernel synchronization needs

Intra-kernel Synchronization

- Background
- Key Questions
- Strategies
- Performance Comparison
- Lessons

Background

- Traditional BSD/Unix kernel is single-threaded, but must handle asynchronous interrupts
- Nested interrupts generally avoided
  - certainly for single device driver
  - priority assignment may allow limited nesting
- Interrupt disabling/enabling commonly used to protect critical sections within kernel

Key Questions

- How often are critical sections entered?
- How long do critical sections last?
- How often are they interrupted?
- What is the frequency of interrupts?
- What is the cost of various interrupt-based synchronization schemes?

CS entry and duration; interrupt frequency

- Across a range of workloads, on a 120MHz Pentium, saw 25K-50K CS's per second on average, >80K peak
- Typical duration was ~3µs per CS
- Majority <1µs per CS; few >4µs per CS
- <1% CS interruptions

Synchronization Strategies

- Two basic dimensions to explore
  - where to mask interrupts
    - on-chip (disable all interrupts)
    - off-chip (disable selected interrupts)
  - how to mask interrupts
    - pessimistically (before entering CS)
    - optimistically (after entering CS)

On-chip vs Off-chip

- On-chip
  - disables all interrupts
  - takes 5-10 cycles
- Off-chip
  - disable selected interrupts (esp. specific device)
  - takes 300 cycles

Pessimistic vs Optimistic

- Pessimistic
  - disable interrupts before entering CS
  - enable interrupts after leaving CS
- Optimistic
  - use variable to store interrupt flag value
  - all ISRs consult variable as first step, and then disable interrupts as needed before proceeding

Scheme Overhead Comparison

	MAB-L	MAB-R	Wstone	Serial
Off-P	9.0%	6.5%	8.6%	11.9%
Off-O	0.2%	0.1%	0.3%	0.3%
On-P	0.7%	0.5%	0.6%	0.9%
On-O	0.2%	0.1%	0.2%	0.2%

Lessons

- Intra-kernel synchronization is common
- High overhead is easy; so is low overhead!
- Overhead is *very* hardware-specific

Summary

- Synchronization issues are real
  - at process level
  - within kernel
- Synchronization is messy, often expensive
- Careful tuning can reduce some overhead
- Question: what happens when kernels become modularized into user-level servers???