CSE 454
Synchronization, Monitors, Deadlocks

Course Overview

- Info Extraction
- Data Mining
- E-commerce
- P2P
- Security
- Web Services
- Semantic Web
- Case Studies: Nutch, Google, Altavista
- Information Retrieval
- Precision vs Recall
- Inverted Indices
- Crawler Architecture
- Synchronization & Monitors
- Systems Foundation: Networking & Clusters

Reading

- **Focused Crawling: A New Approach To Topic-Specific Web Resource Discovery.**
- **Efficient Crawling Through URL Ordering,**
  - Ideas may well help your crawler find webcams
  - Read to the extent that they are helpful

- **The Anatomy Of A Large-Scale Hypertextual Web Search Engine**
  - “Must” reading for everyone

Threads and processes

- **Most modern OS’s support two entities:**
  - the **process** defines the address space and general process attributes (such as open files, etc.)
  - the **thread** defines a sequential execution stream within a process
- **A thread is bound to a single process**
  - processes can have multiple threads executing within them
  - sharing data between threads is cheap: all see same address space
- **Threads become the unit of scheduling**
  - processes are just **containers** in which threads execute

Thread Design Space

- **MS-DOS**
  - one thread/process
  - one process

- **Java**
  - many threads/process
  - one process

- **older UNIXes**
  - one thread/process
  - many processes

- **Mach, NT, Chorus, Linux, …**
  - many threads/process
  - many processes
Synchronization

• Threads cooperate in multithreaded programs
  – to share resources, access shared data structures
    • e.g., threads accessing a memory cache in a web server
    • also, to coordinate their execution
    • e.g., a disk read thread hands off a block to a network writer
• For correctness, we have to control this cooperation
  – must assume threads interleave executions arbitrarily and at different rates
    • scheduling is not under application writers’ control
    • we control cooperation using synchronization
    • enables us to restrict the interleaving of executions
• Note: this also applies to processes, not just threads
  – and it also applies across machines in a distributed system

Shared Resources

• Focus on coordinating access to shared resources
  – basic problem:
    • two concurrent threads are accessing a shared variable
    • if the variable is read/modified/written by both threads, then access to the variable must be controlled
    • otherwise, unexpected results may occur
• Overview:
  – mechanisms to control access to shared resources
    • low level mechanisms like locks
    • higher level mechanisms like monitors and condition variables
  – patterns for coordinating access to shared resources
    • bounded buffer, producer-consumer, …

The classic example

• Suppose we have to implement a function to withdraw money from a bank account:

```c
int withdraw(account, amount) {
  balance = get_balance(account);
  balance -= amount;
  put_balance(account, balance);
  return balance;
}
```

• Now suppose that you and your S.O. share a bank account with a balance of $100.00
  – what happens if you both go to separate ATM machines, and simultaneously withdraw $90.00 from the account?

Example continued

• What’s the problem with this?
  – what are the possible balance values after this runs?

Interleaved Schedules

• The problem is that the execution of the two threads can be interleaved, assuming preemptive scheduling:

```c
balance = get_balance(account);
balance -= amount;
put_balance(account, balance);
```

• What’s the account balance after this sequence?
  – who’s happy, the bank or you? ;)

The crux of the matter

• The problem: two concurrent threads access a shared resource (account) without any synchronization
  – creates a race condition
  – output is non-deterministic, depends on timing
• We need mechanisms for controlling access to shared resources in the face of concurrency
  – so we can reason about the operation of programs
  – essentially, re-introducing determinism
• Synchronization is necessary for any shared data structure
  – buffers, queues, lists, hash tables, …
When are Resources Shared?

- **Local variables are not shared**
  - refer to data on the stack, each thread has its own stack
  - never pass/share/store a pointer to a local variable on another thread’s stack
- **Global variables are shared**
  - stored in the static data segment, accessible by any thread
- **Dynamic objects are shared**
  - stored in the heap, shared if you can name it
    - in C, can copy up the pointer
    - e.g. void *p = (void *)0xDEADBEEF
    - in Java, strong typing prevents this
  - must pass references explicitly

Critical Section Requirements

- **Mutual exclusion**
  - at most one thread is in the critical section
- **Progress**
  - if thread T is outside the critical section, then T cannot prevent thread S from entering the critical section
- **Bounded waiting**
  - if thread T is waiting on the critical section, then T will eventually enter the critical section
  - assumes threads eventually leave critical sections
- **Performance**
  - the overhead of entering and exiting the critical section is small with respect to the work being done within it

Mechanisms for Building Critical Sections

- **Locks**
  - very primitive, minimal semantics; used to build others
- **Semaphores**
  - basic, easy to get the hang of, hard to program with
- **Monitors**
  - high level, requires language support, implicit operations
  - easy to program with;
  - E.g., Java “synchronized()”
- **Messages**
  - simple model of communication and synchronization based on (atomic) transfer of data across a channel
  - direct application to distributed systems

Locks

- A lock is an object (in memory) that provides the following two operations:
  - acquire( ): a thread calls this before entering a critical section
  - release( ): a thread calls this after leaving a critical section
- Threads pair up calls to acquire() and release( )
  - between acquire() and release(), the thread holds the lock
  - acquire( ) does not return until the caller holds the lock
  - at most one thread can hold a lock at a time (usually)
  - so: what can happen if the calls aren’t paired?
- Implementation requires hardware support
  - atomic test-and-set instruction
  - disable interrupts

Using Locks

- What happens when green tries to acquire the lock?
- Why is the “return” outside the critical section?
  - is this ok?
Deadlock

• When two threads are waiting on a lock held by the other

Dan: “Please get your clothes on Galen”
Galen: “Give me another math problem, Dad.”
Dan: “I’ll do that after you start getting your clothes on.”
Galen: “I won’t get my clothes on until you give me a problem.”

Avoiding Deadlock

• Simplest method
• Focus on lock order
• Every procedure should get locks in same order
  – What if use overlapping sets of locks?

Monitors

• A programming language construct that supports controlled access to shared data
  – synchronization code added by compiler, enforced at runtime
  – why does this help?
• Monitor is a software module that encapsulates:
  – shared data structures
  – procedures that operate on the shared data
  – synchronization between concurrent threads invoking those procedures
• Monitor protects the data from unstructured access
  – guarantees one may only access data through procedures
  – hence in legitimate ways

A monitor

waiting queue of threads trying to enter the monitor

shared data

operations (procedures)

at most one thread in monitor at a time

Monitor facilities

• Mutual exclusion
  – only one process can be executing inside at any time
  – thus, synchronization implicitly associated with monitor
  – if a second process tries to enter a monitor procedure, it blocks until the first has left the monitor
  – more restrictive than locks, semaphores!
  – but easier to use most of the time
• Once inside, a process may discover it can’t continue, and may wish to sleep
  – or, allow some other waiting process to continue
  – condition variables provided within monitor
    – processes can wait or signal others to continue
    – condition variable can only be accessed from inside monitor
### Condition Variables

- **A place to wait; sometimes called a rendezvous point**
- **Three operations on condition variables**
  - `wait(c)`
    - wait for somebody else to signal condition
    - thus, condition variables have wait queues
  - `signal(c)`
    - release monitor lock, so somebody else can get in
    - wake up at most one waiting process/thread
    - if no waiting processes, signal is lost
  - `broadcast(c)`
    - release monitor lock
    - wake up all waiting processes/threads

### Bounded Buffer using Monitors

Monitor bounded_buffer {
  buffer resources[N];
  condition not_full, not_empty;

  procedure add_entry(resource $x$) {
    while (array "resources" is full)
      wait(not_full);
    add "$x" to array "resources"
    signal(not_empty);
  }

  procedure get_entry(resource $*$x) {
    while (array "resources" is empty)
      wait(not_empty);
    $*$x = get resource from array "resources"
    signal(not_full);
  }
}

### Two Kinds of Monitors

- **Hoare monitors: `signal(c)` means**
  - run waiter immediately
  - signaler blocks immediately
    - condition guaranteed to hold when waiter runs
  - but, signaler must restore monitor invariants before signaling!
- **Mesa monitors: `signal(c)` means**
  - waiter is made ready, but the signaler continues
    - waiter runs when signaler leaves monitor (or waits)
    - condition is not necessarily true when waiter runs again
  - signaler need not restore invariant until it leaves the monitor
  - being woken up is only a hint that something has changed
    - must recheck conditional case

### Examples

- **Hoare monitors**
  - if (notReady)
    - `wait(c)`
- **Mesa monitors**
  - while (notReady)
    - `wait(c)`
- **Mesa monitors easier to use**
  - more efficient
  - fewer switches
  - directly supports broadcast

### Synchronization in the 454 Project

- **Multiple crawler threads**
  - More efficient than requesting, waiting for single page to download while doing nothing else (interleave I/O with computation)
- **What are the shared resources?**
  - Page Repository?
  - Queues?
    - Only one thread may take pages off a queue, but
    - What about adding to a thread’s queue
  - Everything?
    - Consistent view during checkpointing