Implementing Threads

CS149
Lecture 4

The Big Picture

- **Mantra:**
  - Any sequential overhead in a parallel program limits parallel scalability
  \[
  \frac{1}{(1-P)^2 + P/S}
  \]
  - Even minor costs can become very important in highly parallel programs

- Thread implementations must be done well
  - If the goal is lots of parallelism

Outline

- Thread control contexts
- Locks
- Java memory model
  - again!
- Memory management
- Thread pools
  - again!

Topic 1: Threads of Control

- A thread consists of
  - A program counter (PC)
    - The current instruction the thread is executing
  - A stack of activation records (the stack)
    - An activation records the local variables, return address, etc. for the current procedure/method call
    - The stack records all the activation records of the chain of function calls from the initial call to main() to the currently executing procedure

Fork/Join, OS Style

- On encountering a **fork()**
  - Create a new thread (process)
  - Copy the stack of the old thread
  - Copy the program counter of the old thread

- Add new thread to the OS process queue
Comments

• Note new thread/process has the same lexical context as the old thread

• But no variables are shared
  - The new thread gets a copy of the old state

• Copying the stack is expensive

Fork/Join Lightweight Thread Style

• On fork()
  - Copy PC
  - Copy stack pointer
    • Threads share stack

• Requires different stack representation
  - Each AR has a pointer to its parent
  - A cactus stack

Recall: Creating/Destroying Threads in Java

class MyThreads extends Thread {
  public void run() { .... }
}
Thread thread = new MyThreads();
thread.start(); // thread executes its run() method concurrently
thread.join();

Java Threads

• A thread is an object with a run() method
  - Invoked by the system at top-level

• Implies child’s stack is initially empty
  - All thread stacks are independent
    • No need for cactus stack
  - JVM provides interface for setting size of stack
    • i.e., stacks are contiguous memory, like hardware stack

Topic 2: Locks

• A lock consists of
  - State for the lock
    • Boolean or counter
    • Thread-id
  - Mode
    - Read/write
    - Read only

• Blocked thread queue
• Waiting thread queue
  - If wait/notify supported

Example OS Implementation

class Lock {
  bool bState;
  Queue oQueue;
  void Lock (void) {
    DisableInterrupts();
    if (bState) {
      Queue.Put (ThisProcess());
      Sleep();
    } else bState = TRUE;
    EnableInterrupts();
  }
  void Unlock (void) {
    DisableInterrupts();
    if (Queue.Empty()) {
      bState = FALSE;
    } else {
      Wake (Queue.Get ());
    }
    EnableInterrupts();
  }
};

Source: Petr Tuma
Example Spin-Lock Implementation

class Lock {
  bool bState;

  void Lock (void) {
    while (Exchange (bState, TRUE)) { }
  }

  void Unlock (void) {
    bState = FALSE;
  }
};

Puzzle

• How to:
  - Implement blocking lock/unlock operations
  - Using Exchange as the primitive

Java Locks

• Recall: every object in Java is a lock

• A feature!
  - Very convenient
  - Natural lock for any object is always available

• A bug!
  - Locks are heavy
  - Every object has two private queues?!

An Insight

• It turns out that, empirically:
  - Most locks acquired once by one thread
    - And released before any contention occurs
    - 80% case

  - A thread trying to acquire a held lock is usually the thread that holds that lock
    - Most of the remaining 20%

Thin Locks

• Initially, a lock is one word
  - Inlined in the object header

• Unlocked: word is 0

• Locked: word is id of locking thread
  - An integer

Thin Locks (Cont.)

• When attempting to lock a thin lock:
  - Check if word is 0 Yes: write thread id
  - Check if word is my thread id Yes: have lock

• If word is some other thread id
  - Inflate the lock
  - Replace integer with pointer to a lock object
    - With queues, state, etc.
  - Once inflated, lock remains inflated
Topic 3: Java Memory Model

- We know enough now to explain the Java concurrent memory model in more detail
- Important to understand at least informally what the rules are
  - Can affect program semantics & performance

Review: How Memory Works

- Consider $X = X + 1$
- Typical steps:
  - Load $X$ from memory to a register $A$
  - $A = A + 1$
  - Store $X$ back to memory

Java Terminology

- There is a master copy of each variable
  - The one in memory
- Threads use working copies in local storage
  - i.e., registers

The Problem

- Consider $X = X + 1$
- Possible steps:
  - Load $X$ from memory to a register $A$
  - $A = A + 1$
  - Another thread comes in and reads the copy in memory

What Do We Want?

Thread 1
...
lock(o)
$X = ...$
unlock(o)

Thread 2

lock(o)
$... = X$
onlock(o)

Rules

- Rule 1
  Before performing an unlock(\_), all shared variables in local memory that have been written must be stored back to memory

- Rule 2
  After performing a lock(\_), all shared variables must be loaded from memory before they are read
Discussion

- Note that
  - Only writes are required on unlocking
  - Only reads are required on locking

- All shared variables are written (unlock) or reloaded (lock)
  - There is no system-maintained relationship between locks and particular objects
  - Any such relationship is purely a convention of the programmer

Discussion (Cont.)

- Locking even heavier operation than expected
  - Not sufficient to begin writes when unlocking
  - Writes must complete before releasing the lock

- And nothing is guaranteed without locking

- Shared memory = single address space
  - But the concept of local memory seems necessary to explain it!

Correction: Long Ints and Doubles

- The current standard does not require long ints and doubles to be written atomically
  - Unless they are declared volatile

- Standard states this is a concession to current hardware
  - And may change in the future

Topic 4: Memory Management

- New objects are allocated out of free memory

- If centralized in a single memory manager, potential bottleneck
  - Memory allocation and deallocation is serialized
  - Remember Amdahl's law!

Serial Garbage Collection vs. Parallelism

- Each thread has its own block of memory
  - All allocation done out of local block
  - Deallocation returns memory to local block
  - Instead of 1 memory manager, N memory managers

- When thread runs out of memory
  - Local garbage collection
  - If still not enough space, initiate global collection
  - Global collections are still synchronization points
Parallel Garbage Collection

- Often the garbage collector runs in a separate thread
  - Works in parallel with client program or mutator
- Tricky
  - Should only collect unreachable objects
    - Objects with no chain of pointers to them, beginning at a global or local variable
  - But mutator can allocated and update pointers while garbage collector is running

Sketch of Parallel Collection Algorithm

- Every object is in one of three states:
  - White: Unreachable, so far as we know
  - Black: Reachable from a root, so far as we know
  - Grey: Reachable, successors not yet processed

- Invariant:
  \[ A \text{ black node never points to a white node} \]

Sketch of Parallel Collection (Cont.)

- Initially all objects are white
- Color objects \( o \) pointed to by variables black
  - Color white successors of \( o \) gray
- Repeat
  - Pick a gray object \( o \)
  - Color \( o \) black
  - Color \( o \)'s white successors gray

Sketch of Parallel Collection (Cont.)

- Mutator
  - Newly allocated objects are colored black
  - On field update \( x.f = y \)
    - If \( x \) is black and \( y \) is white, mark \( y \) gray
- When there are no gray objects,
  - Algorithm halts; nothing to do
  - Every object is either black or white
  - White objects are definitely unreachable
    - deallocate

Comments

- Minimal/no synchronization required
  - Mutator and collector may access the same objects
  - But modify different fields
    - "color field" of object is reserved for GC
  - Every memory manager does as much local collection as possible
    - Within their private memory
    - But global collections still required sometimes
    - These require synchronization between collectors
More Comments

• Some applications work better with parallel automatic memory management than others
• Applications with low memory pressure
  – Not much data, not much management of the data
• Applications with little shared data
  – Mostly thread-private data, few global collections
• Hard case: lots of data and sharing

Thought Experiment

• 1,000,000 “small” but independent jobs
  – Cost per job equals cost to create/destroy 1 thread
• Approach
  – Create 1,000,000 threads
• What is the machine utilization?

Topic 5: Thread Pools

• Remember: Control is a resource!
• Threads consume resources
  – Memory for the state of the thread
  – Scheduling time/logic for managing threads
• Too many threads will kill performance
  – All time and memory spent managing threads
  – And not doing useful work

Thread Pools

• An abstraction for
  – Decoupling number of threads from amount of work
  – Matching number of threads to hardware capacity
• Two parts
  – A number of threads (“the pool”)
  – A worklist of “tasks” (things to do)
• Threads repeatedly take an item from the worklist, do it, repeat

The Trade-Off

• Easiest programming solution
  – One thread per “task”
  – But this may easily swamp the system
• Instead a thread is a loop
  – Repeatedly get something to do from worklist
• Advantages
  – Don’t set up/tear down a thread for each task
  – Set # of threads separately from # of tasks
Conclusions

- Thread implementations are sophisticated
- Forced to be, because of Amdahl's law
- Overheads can creep in from many sources
  - Memory management
  - Lock management
  - Other forms of implicit synchronization