Lecture 3: Introduction to Shared Memory

CS178: Programming Parallel and Distributed Systems

January 31, 2001
Steven P. Reiss

I. Overview

A. Hardware for shared memory systems

B. Fundamental problems and terms

C. Addressing synchronization

II. Architecture

A. Single processor these days

1. **CPU with pipeline, multiple ALUs, multiple instructions going on at once**
2. **L1 cache (direct mapped); write thru or flush**
3. **L2 cache (semi-associative)**
4. **L3 cache (off chip)**
5. **Main memory**
   a) Access to memory is atomic

B. Multiple CPU's in a box

1. **Workstations and servers today**
2. **Multiple CPUs on a chip coming**
3. **Problems that need to be addressed**
   a) How to access memory
   b) How to manage caches

C. UMA (uniform memory architecture)

1. **Multi-ported memory or bus-arbitrated**
2. **Snoopy cache coherency**

D. NUMA (non-uniform)

1. **Each processor has local memory**
2. **Tied together via memory bus**

III. Software Architecture

A. Threads and processes

1. A process is set of resources allocated by the OS

a) Memory

b) File handles

c) CPU

2. A process executes via a thread of control

a) This is access to the CPU with its other resources

b) Typically a process only has one thread of control

3. Processes can have multiple threads of control

a) These share the same memory/file handles/etc.

b) These can exist on multiple CPUs or share a CPU

B. Thread Models

1. Kernel threads versus user threads

2. Time slicing versus self-scheduling

3. pthreads

a) Set of library routines for UNIX

b) Call to set up a new thread

(1) Stack space problem

c) Call to start a new thread

(1) Runs a particular routine

4. java threads

a) Object that represents a thread

(1) The object is created

b) Starting the thread executes its “run” method

C. Implementation

1. Single CPU

2. Multiple CPUs

3. Network-based

D. Programming

1. Easiest to conceive

2. Most difficult to get right

IV. Basic Problems

A. Memory Synchronization

1. **All threads have access to memory**
2. **Must ensure that memory is accessed "correctly"**
3. **Consider the problem of maintaining a linked list**

B. Thread Synchronization

1. **Iterative/recursive parallelism implies multiple threads need to complete and then combine results**
2. **How to get the threads to coordinate when done**
3. **How to allocate work among threads**
   a) Input event processor/work threads

V. Memory Synchronization

A. Where do we have problems

1. **Read - read :: no problem**
2. **Write - write :: problem (which gets written)**
   a) Think of our event counter
3. **Read - write :: possible problem**
   a) Single read -- might not be a problem
   b) In general, view this as a problem
4. **Intermediate steps**
   a) Inconsistencies in the middle of updating a structure

B. Basic solution

1. **Identify "CRITICAL REGIONS"**
   a) These are portions of the code where memory problems might occur, i.e. where threads access shared data structures
   b) These can also be portions of the code where inconsistencies might arise
   c) Examples
2. **Provide for "MUTUAL EXCLUSION" for these regions**
   a) Various techniques can be used here
3. **This provides for "SAFETY"**

a) The final result is “correct”

4. What do we mean by correctness

a) Proving properties of programs

b) Generally enough to insure noninterference

VI. Thread Synchronization

A. Where do we have problems

1. Threads might have to wait for other threads to finish

a) Producer-consumer problems

2. Threads might have to coordinate with each other

a) Graphics loop and updating

3. Must ensure that threads get to execute

a) Don’t want one thread to block all the others

b) Don’t want threads to block each other

B. Basic solutions

1. Provide ways for threads to wait for conditions or termination of other threads

2. Provide ways for scheduling threads that are “FAIR”

a) Fairness implies that if a thread can be run it will be

b) Weak fairness implies that if a thread is executable and remains so, it will eventually be run

c) Strong fairness implies that if a thread is executable infinitely often, it will eventually be run

3. Code to prevent “DEADLOCK” and “LIVELOCK”

a) Deadlock occurs when some subset of the threads are all waiting on one another so that none can execute

b) Livelock is a situation when threads can still execute but where no progress can be made

VII. Mutual Exclusion

A. How can we create code that ensures two threads don’t interfere with each other

1. Simple example :: boolean flag set in critical region

2. Suggestions for other solutions

3. Workable solution?

```
boolean enter1 = false;
boolean enter2 = false;
int turn = 1;

PROCESS i:

   Entry_Protocol:
      enter1 = true;
      turn = 3-i;
      while enter2 && turn == 3-i do;

   Critical region

   Exit_Protocal:
      enter1 = false;
```

B. This has several problems

1. Very difficult to get right

2. Difficult to scale to varying #s of processors

3. Spin locks waste CPU cycles

4. May or may not be fair

C. Solutions

1. Provide a machine instruction to make protocol easier

a) Test and set is one example

2. Move the lock into the OS

a) OS can maintain set of threads waiting on lock

b) OS can idle a thread waiting on lock

c) OS can ensure fairness

d) Protocol is done right in the OS, not by the programmer

3. Problems with this

a) Can be quite expensive (entering OS is)

b) Do in a library with OS help

VIII. Next Time

A. We will look at the different solutions that are provided here

B. We will look at the Java programming model