Lecture 26: Multiprocessors

- Today’s topics:
  - Synchronization
  - Consistency
  - Shared memory vs message-passing
Constructing Locks

• Applications have phases (consisting of many instructions) that must be executed atomically, without other parallel processes modifying the data.

• A lock surrounding the data/code ensures that only one program can be in a critical section at a time.

• The hardware must provide some basic primitives that allow us to construct locks with different properties.

Parallel (unlocked) banking transactions:
Synchronization

• The simplest hardware primitive that greatly facilitates synchronization implementations (locks, barriers, etc.) is an atomic read-modify-write

• Atomic exchange: swap contents of register and memory

• Special case of atomic exchange: test & set: transfer memory location into register and write 1 into memory (if memory has 0, lock is free)

• lock: t&s register, location
bnz register, lock
CS
st location, #0

When multiple parallel threads execute this code, only one will be able to enter CS
Coherence Vs. Consistency

• Recall that coherence guarantees (i) write propagation (a write will eventually be seen by other processors), and (ii) write serialization (all processors see writes to the same location in the same order)

• The consistency model defines the ordering of writes and reads to different memory locations – the hardware guarantees a certain consistency model and the programmer attempts to write correct programs with those assumptions
Consistency Example

- Consider a multiprocessor with bus-based snooping cache coherence

```
Initially A = B = 0
P1        P2
A ← 1     B ← 1
...       ...
if (B == 0) if (A == 0)
Crit.Section    Crit.Section
```
Consistency Example

- Consider a multiprocessor with bus-based snooping cache coherence

<table>
<thead>
<tr>
<th>Initially A = B = 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1 (\rightarrow) A (\leftarrow 1) (\rightarrow) P2</td>
</tr>
<tr>
<td>B (\leftarrow 1)</td>
</tr>
<tr>
<td>(\ldots)</td>
</tr>
<tr>
<td>if (B == 0) (\rightarrow) Crit.Sect. (\leftarrow) if (A == 0) (\rightarrow) Crit.Sect.</td>
</tr>
</tbody>
</table>

The programmer expected the above code to implement a lock – because of ooo, both processors can enter the critical section

The consistency model lets the programmer know what assumptions they can make about the hardware’s reordering capabilities
Sequential Consistency

- A multiprocessor is sequentially consistent if the result of the execution is achieveable by maintaining program order within a processor and interleaving accesses by different processors in an arbitrary fashion.

- The multiprocessor in the previous example is not sequentially consistent.

- Can implement sequential consistency by requiring the following: program order, write serialization, everyone has seen an update before a value is read – very intuitive for the programmer, but extremely slow.
Shared-Memory Vs. Message-Passing

**Shared-memory:**
- Well-understood programming model
- Communication is implicit and hardware handles protection
- Hardware-controlled caching

**Message-passing:**
- No cache coherence → simpler hardware
- Explicit communication → easier for the programmer to restructure code
- Software-controlled caching
- Sender can initiate data transfer
Procedure Solve(A)
begin
  diff = done = 0;
  while (!done) do
    diff = 0;
    for i ← 1 to n do
      for j ← 1 to n do
        temp = A[i,j];
        A[i,j] ← 0.2 * (A[i,j] + neighbors);
        diff += abs(A[i,j] – temp);
      end for
    end for
    if (diff < TOL) then done = 1;
  end while
end procedure
int n, nprocs;
float **A, diff;
LOCKDEC(diff_lock);
BARDEC(bar1);

main()
begin
  read(n); read(nprocs);
  A ← G_MALLOC();
  initialize (A);
  CREATE (nprocs,Solve,A);
  WAIT_FOR_END (nprocs);
end main

procedure Solve(A)
  int i, j, pid, done=0;
  float temp, mydiff=0;
  int mymin = 1 + (pid * n/procs);
  int mymax = mymin + n/nprocs -1;
  while (!done) do
    mydiff = diff = 0;
    BARRIER(bar1,nprocs);
    for i ← mymin to mymax
      for j ← 1 to n do
        ...
      endfor
    endfor
    LOCK(diff_lock);
    diff += mydiff;
    UNLOCK(diff_lock);
    BARRIER (bar1, nprocs);
    if (diff < TOL) then done = 1;
    BARRIER (bar1, nprocs);
  endwhile
Message Passing Model

```c
main()
    read(n); read(nprocs);
    CREATE (nprocs-1, Solve);
    Solve();
    WAIT_FOR_END (nprocs-1);

procedure Solve()
    int i, j, pid, nn = n/nprocs, done=0;
    float temp, tempdiff, mydiff = 0;
    myA \leftarrow \text{malloc}(\ldots)
    initialize(myA);
    while (!done) do
        mydiff = 0;
        if (pid != 0)
            SEND(&myA[1,0], n, pid-1, ROW);
        if (pid != nprocs-1)
            SEND(&myA[nn,0], n, pid+1, ROW);
        if (pid != 0)
            RECEIVE(&myA[0,0], n, pid-1, ROW);
        if (pid != nprocs-1)
            RECEIVE(&myA[nn+1,0], n, pid+1, ROW);
        for i \leftarrow 1 to nn do
            for j \leftarrow 1 to n do
                \ldots
            endfor
        endfor
        if (pid != 0)
            SEND(mydiff, 1, 0, DIFF);
        RECEIVE(done, 1, 0, DONE);
        else
            for i \leftarrow 1 to nprocs-1 do
                RECEIVE(tempdiff, 1, *, DIFF);
                mydiff += tempdiff;
            endfor
        if (mydiff < TOL) done = 1;
            for i \leftarrow 1 to nprocs-1 do
                SEND(done, 1, I, DONE);
            endfor
        endif
    endwhile
```

Multithreading Within a Processor

• Until now, we have executed multiple threads of an application on different processors – can multiple threads execute concurrently on the same processor?

• Why is this desireable?
  ➢ inexpensive – one CPU, no external interconnects
  ➢ no remote or coherence misses (more capacity misses)

• Why does this make sense?
  ➢ most processors can’t find enough work – peak IPC is 6, average IPC is 1.5!
  ➢ threads can share resources → we can increase threads without a corresponding linear increase in area
Title

• Bullet