Recap: Virtual Memory

- Virtual memory support standard on general-purpose processors
  - Gives each user (or program) illusion of a separate large protected memory
  - Programs can be written independent of machine memory configuration
- Hierarchical page tables exploit sparseness of virtual address usage to reduce size of mapping information
- TLB caches translation/protection information to make VM practical
  - Would not be acceptable to have multiple additional memory references for each instruction
- Interaction between TLB lookup and cache tag lookup
  - Want to avoid inconsistencies from virtual address aliases
Déjà vu all over again?

“... today’s processors ... are nearing an impasse as technologies approach the speed of light...”


- Transputer had bad timing (Uniprocessor performance↑)
  ⇒ Procrastination rewarded: 2X seq. perf. / 1.5 years
- “We are dedicating all of our future product development to multicore designs.
  ... This is a sea change in computing”
  Paul Otellini, President, Intel (2005)
- All microprocessor companies switch to MP (2X CPUs / 2 yrs)
  ⇒ Procrastination penalized: 2X sequential perf. / 5 yrs

<table>
<thead>
<tr>
<th>Manufacturer/Year</th>
<th>AMD/’07</th>
<th>Intel/’07</th>
<th>IBM/’07</th>
<th>Sun/’07</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processors/chip</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>Threads/Processor</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>Threads/chip</td>
<td>4</td>
<td>2</td>
<td>4</td>
<td>64</td>
</tr>
</tbody>
</table>
**Other Factors ⇒ Multiprocessors**

- Growth in data-intensive applications
  - Data bases, file servers, ...
- Growing interest in servers, server perf.
- Increasing desktop perf. less important
  - Outside of graphics
- Improved understanding in how to use multiprocessors effectively
  - Especially server where significant natural TLP
- Advantage of leveraging design investment by replication
  - Rather than unique design

---

**Flynn’s Taxonomy**


- Flynn classified by data and control streams in 1966

<table>
<thead>
<tr>
<th>Single Instruction, Single Data (SISD) (Uniprocessor)</th>
<th>Single Instruction, Multiple Data SIMD (single PC: Vector, CM-2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiple Instruction, Single Data (MISD) (?????)</td>
<td>Multiple Instruction, Multiple Data MIMD (Clusters, SMP servers)</td>
</tr>
</tbody>
</table>

- SIMD ⇒ Data-Level Parallelism
- MIMD ⇒ Thread-Level Parallelism
- MIMD popular because
  - Flexible: N programs or 1 multithreaded program
  - Cost-effective: same MPU in desktop & MIMD machine
Back to Basics

- “A parallel computer is a collection of processing elements that cooperate and communicate to solve large problems fast.”
- Parallel Architecture = Computer Architecture + Communication Architecture

Two Models for Communication and Memory Architecture

1. Communication occurs by explicitly passing messages among the processors: message-passing multiprocessors (aka multicomputers)
   - Modern cluster systems contain multiple stand-alone computers communicating via messages
2. Communication occurs through a shared address space (via loads and stores): shared-memory multiprocessors either
   - UMA (Uniform Memory Access time) for shared address, centralized memory MP
   - NUMA (Non-Uniform Memory Access time multiprocessor) for shared address, distributed memory MP
   - In past, confusion whether “sharing” means sharing physical memory (Symmetric MP) or sharing address space
Centralized vs. Distributed Memory

Centralized Memory

Distributed Memory

Centralized Memory Multiprocessor

- Also called symmetric multiprocessors (SMPs) because single main memory has a symmetric relationship to all processors
- Large caches ⇒ single memory can satisfy memory demands of small number of processors
- Can scale to a few dozen processors by using a switch and by using many memory banks
- Although scaling beyond that is technically conceivable, it becomes less attractive as the number of processors sharing centralized memory increases
Distributed Memory Multiprocessor

- **Pro:** Cost-effective way to scale memory bandwidth
  - If most accesses are to local memory
- **Pro:** Reduces latency of local memory accesses
- **Con:** Communicating data between processors more complex
- **Con:** Software must be aware of data placement to take advantage of increased memory BW

Challenges of Parallel Processing

- Big challenge is % of program that is inherently sequential
  - What does it mean to be inherently sequential?
- Suppose 80X speedup from 100 processors. What fraction of original program can be sequential?
  a. 10%
  b. 5%
  c. 1%
  d. <1%
Communication and Synchronization

• Parallel processes must co-operate to complete a single task faster
• Requires distributed communication and synchronization
  – Communication is for data values, or "what"
  – Synchronization is for control, or "when"
  – Communication and synchronization are often inter-related
    » i.e., "what" depends on "when"

• Message-passing bundles data and control
  – Message arrival encodes "what" and "when"
• In shared-memory machines, communication usually via coherent caches & synchronization via atomic memory operations
  – Due to advent of single-chip multiprocessors, it is likely cache-coherent shared memory systems will be the dominant form of multiprocessor
  – Today’s lecture focuses on the synchronization problem

CS252 Administrivia

• Haven’t received many project website URLs, please forward to both Rose and me
  – We will use this for 2nd project meetings, week of October 22
• Midterm #1, Thursday, in class, 9:40AM-11:00AM
  – Closed book, no calculators/computers/iPhones…
  – Based on material and assigned readings from lectures 1-9
  – Practice problems and solutions on website
• Meet in La Vals for pizza/drinks at 7pm after midterm on Thursday
  – Show of hands for RSVP
• Slight change in course calendar
  – All final project presentations on Thursday December 6th (no class on December 4th)
  – Gives all groups same amount of time before final presentation
  – Reminder: final report due Monday December 10th, no extensions
Symmetric Multiprocessors

- All memory is equally far away from all processors
- Any processor can do any I/O (set up a DMA transfer)

Synchronization

The need for synchronization arises whenever there are concurrent processes in a system (even in a uniprocessor system)

- Forks and Joins: In parallel programming, a parallel process may want to wait until several events have occurred
- Producer-Consumer: A consumer process must wait until the producer process has produced data
- Exclusive use of a resource: Operating system has to ensure that only one process uses a resource at a given time
A Producer-Consumer Example

Producer posting Item x:
Load $R_{tail}$ (tail)
Store ($R_{tail}$), x
$R_{tail}=R_{tail}+1$
Store (tail), $R_{tail}$

Consumer:
Load $R_{head}$ (head)
spin:
Load $R_{tail}$ (tail)
if $R_{head}=R_{tail}$ goto spin
Load $R_{x}$ ($R_{head}$)
$R_{head}=R_{head}+1$
Store (head), $R_{head}$
process(R)

The program is written assuming instructions are executed in order.

Problems?

A Producer-Consumer Example

Producer posting Item x:
1 Load $R_{tail}$ (tail)
   Store ($R_{tail}$), x
   $R_{tail}=R_{tail}+1$
2 Store (tail), $R_{tail}$

Can the tail pointer get updated before the item x is stored?

Programmer assumes that if 3 happens after 2, then 4 happens after 1.

Problem sequences are:
2, 3, 4, 1
4, 1, 2, 3
Sequential Consistency

A Memory Model

“A system is **sequentially consistent** if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in the order specified by the program”

*Leslie Lamport*

Sequential Consistency =

*arbitrary order-preserving interleaving*

of memory references of sequential programs

Sequential Consistency tasks: T1, T2

Shared variables: X, Y (initially X = 0, Y = 10)

T1:

- Store (X), 1 (X = 1)
- Store (Y), 11 (Y = 11)

T2:

- Load R₁, (Y)
- Store (Y’), R₁ (Y’ = Y)
- Load R₂, (X)
- Store (X’), R₂ (X’ = X)

What are the legitimate answers for X’ and Y’?

\[(X’, Y’) \in \{(1,11), (0,10), (1,10), (0,11)\}\]

*If y is 11 then x cannot be 0*
Sequential Consistency

Sequential consistency imposes more memory ordering constraints than those imposed by uniprocessor program dependencies (→)

*What are these in our example?*

T1:
- Store (X), 1 (X = 1)
- Store (Y), 11 (Y = 11)

T2:
- Load R₁, (Y)
- Store (Y'), R₁ (Y' = Y)
- Load R₂, (X)
- Store (X'), R₂ (X' = X)

Does (can) a system with caches or out-of-order execution capability provide a *sequentially consistent* view of the memory?

more on this later

---

Multiple Consumer Example

*Critical section: Needs to be executed atomically by one consumer ⇒ locks*

Producer posting Item x:
- Load R\text{tail}, (tail)
- Store (R\text{tail}), x
- R\text{tail} = R\text{tail} + 1
- Store (tail), R\text{tail}

Consumer:
- Load R\text{head}, (head)
- Load R\text{tail}, (tail)
- if R\text{head} == R\text{tail} goto spin
- Load R, (R\text{head})
- R\text{head} = R\text{head} + 1
- Store (head), R\text{head}
- process(R)

*What is wrong with this code?*

10/9/2007
Locks or Semaphores
E. W. Dijkstra, 1965

A semaphore is a non-negative integer, with the following operations:

\[ P(s): \text{if } s > 0, \text{ decrement } s \text{ by } 1, \text{ otherwise wait} \]

\[ V(s): \text{increment } s \text{ by } 1 \text{ and wake up one of the waiting processes} \]

P’s and V’s must be executed atomically, i.e., without

- interruptions or
- interleaved accesses to s by other processors

\[ \text{Process } i \]

\[ P(s) \]
\[ \quad \text{<critical section>} \]
\[ V(s) \]

\[ \text{initial value of } s \text{ determines the maximum no. of processes in the critical section} \]

Implementation of Semaphores

Semaphores (mutual exclusion) can be implemented using ordinary Load and Store instructions in the Sequential Consistency memory model. However, protocols for mutual exclusion are difficult to design...

Simpler solution:

\textit{atomic read-modify-write instructions}

Examples: \textit{m is a memory location, R is a register}

\[ \text{Test\&Set (m), R):} \]
\[ R \leftarrow M[m]; \]
\[ \text{if } R = 0 \text{ then} \]
\[ M[m] \leftarrow 1; \]

\[ \text{Fetch\&Add (m), R, } R:\]
\[ R \leftarrow M[m]; \]
\[ M[m] \leftarrow R + R_v; \]

\[ \text{Swap (m), R):} \]
\[ R_t \leftarrow M[m]; \]
\[ M[m] \leftarrow R; \]
\[ R \leftarrow R_t; \]
**Multiple Consumers Example using the Test&Set Instruction**

P: Test&Set (mutex), R_{temp}
   if (R_{temp}! = 0) goto P

spin:
   Load R_{head}, (head)
   Load R_{tail}, (tail)
   if R_{head} == R_{tail} goto spin
   Load R, (R_{head})
   R_{head} = R_{head}+1
   Store (head), R_{head}

V: Store (mutex), 0
   process(R)

Other atomic read-modify-write instructions (Swap, Fetch&Add, etc.) can also implement P’s and V’s

*What if the process stops or is swapped out while in the critical section?*

**Nonblocking Synchronization**

Compare&Swap(m), R_t, R_s:
   if (R_s == M[m])
      then M[m] = R_s;
      R_s = R_t;
      status ← success;
   else status ← fail;

try:
   Load R_{head}, (head)
   Load R_{tail}, (tail)
   if R_{head} == R_{tail} goto spin
   Load R, (R_{head})
   R_{newhead} = R_{head} + 1
   Compare&Swap(head), R_{head}, R_{newhead}
   if (status == fail) goto try
   process(R)
Load-reserve & Store-conditional

Special register(s) to hold reservation flag and address, and the outcome of store-conditional

Load-reserve R, (m):
<flag, adr> ← <1, m>;
R ← M[m];

Store-conditional (m), R:
if <flag, adr> == <1, m>
then cancel other procs’ reservation on m;
M[m] ← R;
status ← succeed;
else status ← fail;

Performance of Locks

Blocking atomic read-modify-write instructions
e.g., Test&Set, Fetch&Add, Swap
vs
Non-blocking atomic read-modify-write instructions
e.g., Compare&Swap,
Load-reserve/Store-conditional
vs
Protocols based on ordinary Loads and Stores

Performance depends on several interacting factors:
degree of contention,
caches,
out-of-order execution of Loads and Stores

later ...
Issues in Implementing Sequential Consistency

Implementation of SC is complicated by two issues

- **Out-of-order execution capability**
  
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load(a); Load(b)</td>
<td>yes</td>
</tr>
<tr>
<td>Load(a); Store(b)</td>
<td>yes if a ≠ b</td>
</tr>
<tr>
<td>Store(a); Load(b)</td>
<td>yes if a ≠ b</td>
</tr>
<tr>
<td>Store(a); Store(b)</td>
<td>yes if a ≠ b</td>
</tr>
</tbody>
</table>

- **Caches**
  
  Caches can prevent the effect of a store from being seen by other processors

Memory Fences

*Instructions to sequentialize memory accesses*

Processors with *relaxed or weak memory models* (i.e., permit Loads and Stores to different addresses to be reordered) need to provide *memory fence* instructions to force the serialization of memory accesses

*Examples of processors with relaxed memory models:*

- Sparc V8 (TSO,PSO): Membar
- Sparc V9 (RMO):
  - Membar #LoadLoad, Membar #LoadStore
  - Membar #StoreLoad, Membar #StoreStore
- PowerPC (WO): Sync, EIEIO

*Memory fences are expensive operations, however, one pays the cost of serialization only when it is required*
Using Memory Fences

Producer posting Item x:
Load \( R_{tail} \) (tail)
Store \( (R_{tail}), x \)
Membar_{SS} \( R_{tail} = R_{tail} + 1 \)
Store (tail), \( R_{tail} \)

ensures that tail ptr is not updated before x has been stored

Consumer:
Load \( R_{head} \) (head)
spin:
Load \( R_{tail} \) (tail)
if \( R_{head} = R_{tail} \) goto spin
Membar_{LL} \( R_{head} = R_{head} + 1 \)
Load R, \( (R_{head}) \)
Store (head), \( R_{head} \)
process(R)

ensures that R is not loaded before x has been stored

Data-Race Free Programs
a.k.a. Properly Synchronized Programs

Process 1

\[ ... \]
\[ \text{Acquire}(\text{mutex}); \]
\[ < \text{critical section}> \]
\[ \text{Release}(\text{mutex}); \]

Process 2

\[ ... \]
\[ \text{Acquire}(\text{mutex}); \]
\[ < \text{critical section}> \]
\[ \text{Release}(\text{mutex}); \]

Synchronization variables (e.g. mutex) are disjoint from data variables
Accesses to writable shared data variables are protected in critical regions
\[ \Rightarrow \text{no data races except for locks} \]

*(Formal definition is elusive)*

In general, it cannot be proven if a program is data-race free.
Fences in Data-Race Free Programs

Process 1

... Acquire(mutex);
membar;
< critical section>
membar;
Release(mutex);

Process 2

... Acquire(mutex);
membar;
< critical section>
membar;
Release(mutex);

- Relaxed memory model allows reordering of instructions by the compiler or the processor as long as the reordering is not done across a fence
- The processor also should not speculate or prefetch across fences

10/9/2007

Mutual Exclusion Using Load/Store

A protocol based on two shared variables c1 and c2. Initially, both c1 and c2 are 0 (not busy)

Process 1

... c1=1;
L: if c2=1 then go to L
  < critical section>
  c1=0;

Process 2

... c2=1;
L: if c1=1 then go to L
  < critical section>
  c2=0;

What is wrong? Deadlock!

10/9/2007
**Mutual Exclusion: second attempt**

To avoid *deadlock*, let a process give up the reservation (i.e. Process 1 sets c1 to 0) while waiting.

**Process 1**

```
... L:  c1=1;
     if c2=1 then
       { c1=0; go to L}  < critical section>
     c1=0
```

**Process 2**

```
... L:  c2=1;
     if c1=1 then
       { c2=0; go to L}  < critical section>
     c2=0
```

- Deadlock is not possible but with a low probability a *livelock* may occur.
- An unlucky process may never get to enter the critical section ⇒ starvation

---

**A Protocol for Mutual Exclusion**

*T. Dekker, 1966*

A protocol based on 3 shared variables c1, c2 and turn. Initially, both c1 and c2 are 0 (*not busy*).

**Process 1**

```
... c1=1;
     turn = 1;
     L:  if c2=1 & turn=1
         then go to L
         < critical section>
     c1=0;
```

**Process 2**

```
... c2=1;
     turn = 2;
     L:  if c1=1 & turn=2
         then go to L
         < critical section>
     c2=0;
```

- turn = i ensures that only process i can wait
- variables c1 and c2 ensure *mutual exclusion*

*Solution for n processes was given by Dijkstra and is quite tricky!*

---
Analysis of Dekker’s Algorithm

Scenario 1

... Process 1
  c1=1;
  turn = 1;
  L: if c2=1 & turn=1
  then go to L < critical section>
  c1=0;

... Process 2
  c2=1;
  turn = 2;
  L: if c1=1 & turn=2
  then go to L < critical section>
  c2=0;

Scenario 2

... Process 1
  c1=1;
  turn = 1;
  L: if c2=1 & turn=1
  then go to L < critical section>
  c1=0;

... Process 2
  c2=1;
  turn = 2;
  L: if c1=1 & turn=2
  then go to L < critical section>
  c2=0;

N-process Mutual Exclusion

Lamport’s Bakery Algorithm

Process i

Initially num[j] = 0, for all j

Entry Code

choosing[i] = 1;
num[i] = max(num[0], ..., num[N-1]) + 1;
choosing[i] = 0;

for(j = 0; j < N; j++) {
  while( choosing[j] );
  while( num[j] &&
         ( ( num[j] < num[i] ) ||
           ( num[j] == num[i] && j < i ) ) );
}

Exit Code

num[i] = 0;