Summary: Multithreaded Categories

<table>
<thead>
<tr>
<th>Time (processor cycle)</th>
<th>Time (processor cycle)</th>
<th>Time (processor cycle)</th>
<th>Time (processor cycle)</th>
<th>Time (processor cycle)</th>
<th>Time (processor cycle)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Superscalar</td>
<td>Fine-Grained</td>
<td>Coarse-Grained</td>
<td>Multiprocessing</td>
<td>Simultaneous</td>
<td>Multithreading</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Thread 1</td>
<td>Thread 3</td>
<td>Thread 5</td>
<td>Thread 2</td>
<td>Thread 4</td>
<td>Idle slot</td>
</tr>
</tbody>
</table>

4/16/2009
Uniprocessor Performance (SPECint)


- VAX: 25%/year 1978 to 1986
- RISC + x86: 52%/year 1986 to 2002
- RISC + x86: ??%/year 2002 to present

Parallel Processing: 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’/09</th>
<th>IBM’/07</th>
<th>Sun’/09</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processors/chip</td>
<td>4</td>
<td>8</td>
<td>2</td>
<td>16</td>
</tr>
<tr>
<td>Threads/Processor</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>Threads/chip</td>
<td>4</td>
<td>16</td>
<td>4</td>
<td>128</td>
</tr>
</tbody>
</table>
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.

---

A Producer-Consumer Example continued

### Producer posting Item x:
1. Load $R_{tail}$, (tail)
2. Store ($R_{tail}$), $x$
3. $R_{tail} = R_{tail} + 1$
4. 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

Sequential concurrent 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 \( R_1 \), (Y)
Store (Y'), R1 \( (Y' = Y) \)
Load \( R_2 \), (X)
Store (X'), R2 \( (X' = X) \)

what are the legitimate answers for X' and Y' ?

\((X', Y') \in \{(1,11), (0,10), (1,10), (0,11)\}\);?
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 $R_{1}$, (Y)
- Store ($Y'$), $R_{1}$ ($Y' = Y$)
- Load $R_{2}$, (X)
- Store ($X'$), $R_{2}$ ($X' = X$)

→ additional SC requirements

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

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

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

*What is wrong with this code?*
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

Process i
\[
P(s) \rightarrow \text{critical section} \rightarrow V(s)
\]

initial value of s 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: atomic read-modify-write instructions

Examples: m is a memory location, R is a register

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

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

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

P: Test&Set (mutex), $R_{\text{temp}}$
   if ($R_{\text{temp}}! = 0$) goto P
   Load $R_{\text{head}}$, (head)
   spin: Load $R_{\text{tail}}$, (tail)
   if $R_{\text{head}} == R_{\text{tail}}$ goto spin
   Load $R_{\text{head}}$, ($R_{\text{head}}$)
   $R_{\text{head}} = R_{\text{head}} + 1$
   Store (head), $R_{\text{head}}$

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

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

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

try:
  Load Rₜₐₜ (head)
  Load Rₜₐₜ (tail)
  if Rₜₐₜ==Rₜₐₜ goto spin
  Load R, (Rₜₐₜ)
  Rₜₐₜ = Rₜₐₜ + 1
  Compare&Swap(head), Rₜₐₜ, Rₜₐₜ
  if (status==fail) goto try
  process(R)

status is an implicit argument

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;

try:
  Load-reserve Rₜₐₜ (head)
  Load Rₜₐₜ (tail)
  if Rₜₐₜ==Rₜₐₜ goto spin
  Load R, (Rₜₐₜ)
  Rₜₐₜ = Rₜₐₜ + 1
  Store-conditional (head), Rₜₐₜ
  if (status==fail) goto try
  process(R)
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

Issues in Implementing Sequential Consistency

Implementation of SC is complicated by two issues

- Out-of-order execution capability
  Load(a); Load(b)  yes
  Load(a); Store(b) yes if a ≠ b
  Store(a); Load(b) yes if a ≠ b
  Store(a); Store(b) yes if a ≠ b

- 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\(_{\text{SS}}\)
- \( R_{tail} = R_{tail} + 1 \)
- Store (tail), \( R_{tail} \)

**Consumer:**
- Load \( R_{head} \) (head)
- Spin:
  - Load \( R_{tail} \) (tail)
  - If \( R_{head} = R_{tail} \) goto spin
  - Membar\(_{\text{LL}}\)
  - Load \( R \) (\( R_{head} \))
  - \( R_{head} = R_{head} + 1 \)
  - Store (head), \( R_{head} \)

\( R_{tail} \) ensures that *tail ptr is not updated before x has been stored*.
\( R_{head} \) ensures that *R is not loaded before x has been stored*.
**Data-Race Free Programs**  
*a.k.a. Properly Synchronized Programs*

**Process 1**
```
...  
Acquire(mutex);  
< critical section>  
Release(mutex);  
```

**Process 2**
```
...  
Acquire(mutex);  
< critical section>  
Release(mutex);  
```

Synchronization variables (e.g. mutex) are disjoint from data variables

Accesses to writable shared data variables are protected in critical regions  

⇒ *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);  
\textcolor{gray}{membar;}  
< critical section>  
\textcolor{gray}{membar;}  
Release(mutex);  
```

**Process 2**
```
...  
Acquire(mutex);  
\textcolor{gray}{membar;}  
< critical section>  
\textcolor{gray}{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
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?

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 →
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;

Acknowledgements

• These slides contain material developed and copyright by:
  – Arvind (MIT)
  – Krste Asanovic (MIT/UCB)
  – Joel Emer (Intel/MIT)
  – James Hoe (CMU)
  – John Kubiatowicz (UCB)
  – David Patterson (UCB)

• MIT material derived from course 6.823
• UCB material derived from course CS252