Barrier Synchronization on a Loaded SMP using Two-Phase Waiting Algorithms

Dan Tsafrir and Dror G. Feitelson
School of Computer Science and Engineering
The Hebrew University, 91904 Jerusalem, Israel
{dants,feit}@cs.huji.ac.il

2016-04-20
Introduction

• The main synchronization constructs: locks and barriers
• The synchronization dilemma: spin or block?
• Common wisdom for barriers: Always-block
Two-phase blocking

• Another solution to the dilemma: two phase blocking
• Spinning for the duration of a context-switch: 2-competitive
**Problem: lacking a global view**

<table>
<thead>
<tr>
<th>standard 2-phase</th>
<th>P1</th>
<th>A1</th>
<th>S+F</th>
<th>CS</th>
<th>A2</th>
<th>S+F</th>
<th>CS</th>
<th>A1</th>
<th>S+F</th>
<th>CS</th>
<th>A2</th>
<th>S+F</th>
<th>CS</th>
</tr>
</thead>
<tbody>
<tr>
<td>P2</td>
<td>B1</td>
<td>S+F</td>
<td>CS</td>
<td></td>
<td>B2</td>
<td>S+F</td>
<td>CS</td>
<td></td>
<td>B1</td>
<td>S+F</td>
<td>CS</td>
<td></td>
<td>B2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>optimal offline</th>
<th>P1</th>
<th>A1</th>
<th>cs</th>
<th></th>
<th>A2</th>
<th>cs</th>
<th></th>
<th>A1</th>
<th>cs</th>
<th></th>
<th>A2</th>
<th>cs</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>P2</td>
<td>B1</td>
<td>cs</td>
<td></td>
<td></td>
<td>B2</td>
<td>cs</td>
<td></td>
<td></td>
<td>B1</td>
<td>cs</td>
<td></td>
<td></td>
<td>B2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>real optimal</th>
<th>P1</th>
<th>A1</th>
<th>S</th>
<th>S</th>
<th>...A1...</th>
<th>cs</th>
<th>...B1...</th>
<th>cs</th>
<th>...A1...</th>
</tr>
</thead>
<tbody>
<tr>
<td>P2</td>
<td>B1</td>
<td>cs</td>
<td>A2</td>
<td>cs</td>
<td>...A2...</td>
<td>cs</td>
<td>...B2...</td>
<td>cs</td>
<td>...A2...</td>
</tr>
</tbody>
</table>
Methodology - The SMP simulator

- An event driven program
- Distinguishes between sync\non-sync jobs
- Simulated applications use two-phase waiting alg.
- Input: machine & workload description
- Output: how well the synchronization policy performed
Methodology - scheduling policy & workloads

- A detailed emulation of the Linux scheduler
- Linux supports 3 policies: FIFO/RR/OTHER
- For general-purpose systems:
  OTHER is a priority-based preemptive scheduler
- Extensive simulations of six combinations:
  - two scheduling policies: RR and OTHER, and
  - three workloads:
    1) One sync job against backdrop of non-sync threads
    2) Homogeneous set of identical sync-jobs
    3) Heterogeneous mixture
Methodology - performance metric

- Successful-spin-rate (SSR)
  % of cases threads succeed to sync while spinning, excluding last thread to arrive

\[
S = \text{set of synchronizing threads}
\]

\[
totalSpin(t) = \text{num of times thread } t \text{ spun, waiting for synchronization}
\]

\[
successSpin(t) = \text{num of times sync achieved before } t \text{ blocked}
\]

\[
SSR = \frac{\sum_{t \in S} successSpin(t)}{\sum_{t \in S} totalSpin(t)} \times 100
\]
Methodology - performance metric

- Intuitively, if SSR < 50%, probably better to block
SCHED_RR, homogenous collection (32-cores machine)

- SSR is above 50% in the "intermediate" load range (32-64 threads = 1-2x the size of the machine)
- After, SSR converges to some value <50%
Why?
“alternating synchronization”

- The answer to the puzzle: alternating synchronization. Jobs fall into pattern of two alternating sets of threads.
- The consequences:
  - SSR converges to positive value < 50%
  - Low effective CPU utilization
  - Common assumption of Poisson arrivals incorrect
SCHED_OTHER, single 11-threads sync job

- With increasing number of CPU-bound jobs
- Results are similar to RR
- But SSR shoots up to near 100% for load = 16n
- Why?
“Transition point”

- The synchronizing-job gains 11 CPUs on a periodic bases
- But threads perform “rapid alt-sync” because:
  - running: $\mu$ (compute) + CS (spin)
  - awakened: CS (schedule) + $\mu$ (compute)
- Rarely, all awakened threads are “faster” than the waiting peers
- Multiple of 16 allows machine to be “partitioned” henceforth
Increase spin duration beyond CS (= context switch)

- Achieve transition regardless of load with spin duration > CS
- CS+ shows improvement, but not optimal. Why?
- Worst case: a thread must wait for its peers to complete blocking, and then to unblock
- 2CS+ provides best combination of stability & performance

![Graph showing SSR vs. Number of Threads for Fine grain and Medium grain with CS, CS+, and 2CS+ conditions]
Synthetic job mixes

- A single synchronizing job: allowed to make good progress even when system is heavily overloaded
- Homogeneous job mix: exactly the opposite (like RR)
- Heterogeneous job mix: somewhere in between
SSR of heterogeneous job mix
Spin vs. always-block

- SSR is not a perfect metric
  - E.g., always spin achieves SSR=100%
- Use speedup for comparison
Spin vs. always-block, single sync job
Spin vs. always-block, heterogeneous mix
Conclusions

• Load is a dominant factor in the decision whether to spin or block
• Spinning for CS is not enough. 2CS+ is optimal
• Alternating-synchronization computation pattern
• Barrier are fundamentally different than locks
• Bug in the Linux scheduler - processors get lost