Optimizing Memory Locality

Memory Access, in Formal Model

- \( p_0 \): test\&set
- \( p_1 \): write
- \( p_2 \): read, write

\( X \) and \( Y \)
In Reality: Memory Interconnect

Memory is accessed through an interconnection network (e.g., a bus)

Local Memory: CC model

Interconnect traffic is expensive
Store copies of data in local memory (cache)
Keep caches coherent with memory and each other (cache coherence model)
Local Memory: DSM model

Larger memory banks are located at the processors *(distributed shared memory model)*
Local Spinning

- An algorithm is **local-spin** if all busy waiting is in read-only loops of local-accesses, which do not cause interconnect traffic.

- An algorithm may be local-spin on one model (DSM or CC) and not local-spin on the other!

- The **remote memory references (RMR)** complexity of an algorithm is the number of remote accesses.

R/W 2-Process Mutex

```plaintext
Want[1] = 0
wait until Want[1-i] == 0 or Priority == 1
Want[1] = 1
if (Priority == 1-i) then
  if (Want[i-1] == 1) then goto Line 1
else wait until (Want[i-1] == 0)
```

- Is this algorithm local-spin?
  - In the DSM model? No
  - In the CC model? Yes

- What is its RMR complexity?
  - In the DSM model? Unbounded
  - In the CC model? Constant
Test&Set Mutex

Entry section

\[
\text{wait until test&set(lock) == 0}
\]

Exit section:

\[
\text{reset(lock)}
\]

- Is this algorithm local-spin?
  - No (in both models)
- What is its RMR complexity?
  - Unbounded (in both models)

Test&Test&Set Mutex

Entry section

\[
\text{while ( test&set(lock) == 1 )}
\]
\[
\text{wait until (lock == 0)}
\]

Exit section:

\[
\text{reset(lock)}
\]

Less traffic in CC model, still not local-spin.
Recall Anderson’s Algorithm

entry section:

```c
myPlace = rmw(\text{Last,Last+1 mod n})
wait until Flags[myPlace] == 1
Flags[myPlace] = 0
```

exit section:

Is this algorithm local-spin?
In the CC model? Yes
In the DSM model? No

Local-Spin Mutex w/ Swap

Atomic register-to-memory swap operations,
also called fetch-and-store
More common

Each process spins on its own location in array

Array contains the queue of waiting processes
Each entry in the array holds a pointer to the next process in line.

```c
\text{swap}(w, \text{new})
\quad \text{prev} = w
\quad w = \text{new}
\quad \text{return prev}
```
Local-Spin Mutex w/ Swap

**Shared variables:**
Flags[0..n-1], binary; all initially 1
Tail {binary, {0,..,n-1}}, initially {0,0}

**Local variables:**
myRecord, prev {binary, {0,..,n-1}},
temp binary

entry section:

```plaintext
myRecord.value = Flags[i]
myRecord.slot = i
prev = swap(tail, myRecord)
wait until(Flags[prev.slot] ≠ prev.value)
```

exit section:

```plaintext
Flags[i]= 1 - Flags[i]
```

Is this algorithm local-spin?
In the CC model? Yes
In the DSM model? No
CLH Lock

[Craig 1993] and [Landin & Hagers, 1994]

- Also a queue, but does not allocate space for all processes
- Instead, “thread” records in a (virtual) linked list

```
entry section:
new myNode
pred = getAndSet(tail, myNode)
wait until ! pred

exit section:
myNode = false
```

```
tail
false
true
true
```
CLH Lock

entry section:

```java
new myNode
pred = getAndSet(tail, myNode)
wait until ¬ pred
```

exit section:

```java
myNode = false
```

Pointers are kept in local memories

Is this algorithm local-spin?
- In the CC model? Yes
- In the DSM model? No

MCS Lock

[Mellor-Crummey and Scott, 1991]

- Maintain a more explicit queue of waiting processes
- Small space overhead
- Local spinning in CC & DSM models
  - Each process has a dedicated record that is enqueued and dequeued
MCS Lock: Enqueing for the lock

- Set tail to point to your record (with compare&set)

MCS Lock: Enqueing for the lock

- Set tail to point to your record (with compare&set)
- Make last element point to your record
MCS Lock: Enqueing for the lock

- Set tail to point to your record (with compare&set)
- Make last element point to your record
- Spin on your own record

MCS Lock: Unlock

- Notify next in line that is can go into the critical section
MCS Lock: Unlock

• Notify next in line that is can go into the critical section
  – Pi sets pj’s flag to false
• Dequeue own record from the list
  – clear the next pointer

MCS Lock: Unlock Subtleties

• Another thread might be joining the list at the same time
  – No thread will be enabled for the critical section
  – Exception (p_k accesses p_i’s reclaimed memory)
MCS Lock: Unlock Subtleties

• Another thread might be joining the list at the same time
• Can be detected since tail is not null
  – Wait for next to be filled before proceeding