Synchronisation

Based on the paper "High Performance Operating Systems" by Robin Sharp, DTU August 2001

02226 High Performance Operating Systems

s002660 Bjarke Frøsig and s001686 Paul Knudsen

DTU, September 23, 2004.

Agenda

- Problem
- Usual proposals
- Multiprocessor Synchronisation
- Cache Concepts
- Barriers
- Other methods
- Conclusion
- Debate

Problem

- A basic problem in multi-process computing:

- We want to do the following with two processes: x = x + 1 and x = x + 1 yielding x = 2 Initial value of x is 0.

```
Proc1                          Proc2
ld  x,r1     (r1 = 0)          ...
add 1,r1     (r1 = 1)          ld  x,r1       (r1 = 0)
st  r1,x     (x := r1 = 1)     add 1,r1       (r1 = 1)
...                            st  r1,x       (x := r1 = 1)
```

- In the end x = 1 where it should be x = 2.

Usual proposals

- We need *atomic operations/critical regions*
- Hardware support
  - Special CPU instructions
    - Test-and-set, swap etc.
    - Impossible on small RISC CPUs
    - Locked memory access

Example

- Test-and-set

Acquire$_{spin}$*(lock)* :

```
while TestAndSet(lock) = LOCKED do
  nop();
end;
```

Release$_{spin}$*(lock)* :

```
Clear(lock);
```

Software Synchronisation

- It is possible to make synchronisation with software
- Two-process Tie-breaker Algorithm
  - Problems with out-of-order execution on SMP
- Lamport's Fast Mutual Exclusion Algorithm
- Operating-system-level (embedded)

Two-process Tie-breaker Algorithm

```
bool in1 = false, in2 = false;
int last = 1;

Process CS1 {
  while (true) {
    in1 = true; last = 1;
    while (in2 and last == 1) skip;
    critical section;
    in1 = false;
    noncritical section;
  }
}

Process CS2 {
  while (true) {
    in2 = true; last = 2;
    while (in1 and last == 2) skip;
    critical section;
    in2 = false;
    noncritical section;
  }
}
```

Multiprocessor Synchronisation

- What about SMPs and distributed systems?
  - Shared address space
  - Scaling
    - Memory consumption
    - Performance

Cache Concepts

- Snooping vs. Directory
- Write invalidation vs. Write update
- Write through vs. Write back

Snooping vs. Directory

- Snooping
  - Snoops on the activity on the bus
- Directory-based
  - Uses a single shared directory

Write invalidation vs. Write update

- Write invalidation
  - When a write is attempted, all other cached copies of the variable is invalidated
- Write update
  - The new value is sent to all cache controllers

Write through vs. Write back

- Write through
  - When data is stored in the local cache, it is also written in the memory
- Write back
  - Data is only written in the local cache – and then written when necessary
  - E.g. if another process generates a read miss

Example

CPU activity	Bus activity	P1 cache	P2 cache	X
				0
P1 reads X	Cache miss for X	0		0
P2 reads X	Cache miss for X	0	0	0
P1 writes 1 to X	Invalidate X	1		0
P2 reads X	Cache miss for X, write back to X	1	1	1

Test-and-TestAndSet

- TestAndSet behavior
  - Problem with write and invalidate signal
- Test-and-TestAndSet

Acquire$_{spin}$*(lock)* :

while *TestAndSet(lock)* = LOCKED do

while *lock* = LOCKED do end;

end;

Test-and-TestAndSet

CPU P1 + cache	CPU P2 + cache	CPU P3 + cache	Bus activity
Has lock	Spins testing *l* = LOCKED	Spins testing *l* = LOCKED	
l := FREE	Receives invalidate signal	Receives invalidate signal	Write invalidate cached copies of l
	Cache read miss when testing *l* = LOCKED	**Cache read miss when testing *l* = LOCKED**	P3 cache miss serviced, write back to memory from P1 cache
	Wait	Reads *l* = FREE	Copy fetched from memory to P3 cache
	Reads *l* = FREE	Executes TestAndSet, gets cache write miss	Copy fetched from memory to P2 cache
	Executes TestAndSet, gets cache write miss	Returns FREE *l* := LOCKED	Generate write invalidate after P3 cache miss
	Returns LOCKED	Enters critical section	Write back to memory after P3 cache miss
	Spins testing *l* = LOCKED		

Binary Exponential Backoff

Acquire$_{BEB}$*(lock)* :
  delay := 1;
  while *TestAndSet(lock)* = LOCKED do
    *pause(irand() * delay);*
    delay := 2 * *delay* + 1;
  end;

Release$_{BEB}$*(lock)* :
  Clear(lock);

- Reduces bus/network contention
- Drawback: Long waits

Graunke and Thakkar's queuing lock algorithm

```
Initialise_queue(lock) :
  lock.slots[0] := TRUE;
  lock.slots[1] := TRUE;
    …
  lock.slots[N - 1] := TRUE;
  lock.who_was_last := 0;
  lock.this_means_locked := FALSE;

Aquire_queue(lock) :
  atomicbegin
    ahead_of_me := lock.who_was_last;
    what_is_locked := lock.this_means_locked;
    lock.who_was_last := myId;
    lock.this_means_locked := lock.slots[myId];
  atomicend;
  await(lock.slots[ahead_of_me] != what_is_locked);

Release_queue(lock) :
  lock.slots[myId] := !lock.slots[myId];
```

- Excellent scaling *O(1)*
- Drawback: Memory use?

Barriers

- Classical barrier:
  - Every thread must spin when entering the barrier
  - Every thread must also spin when leaving the barrier
- Sense reversing centralised barrier
  - A flag eliminates ambiguity

Trees

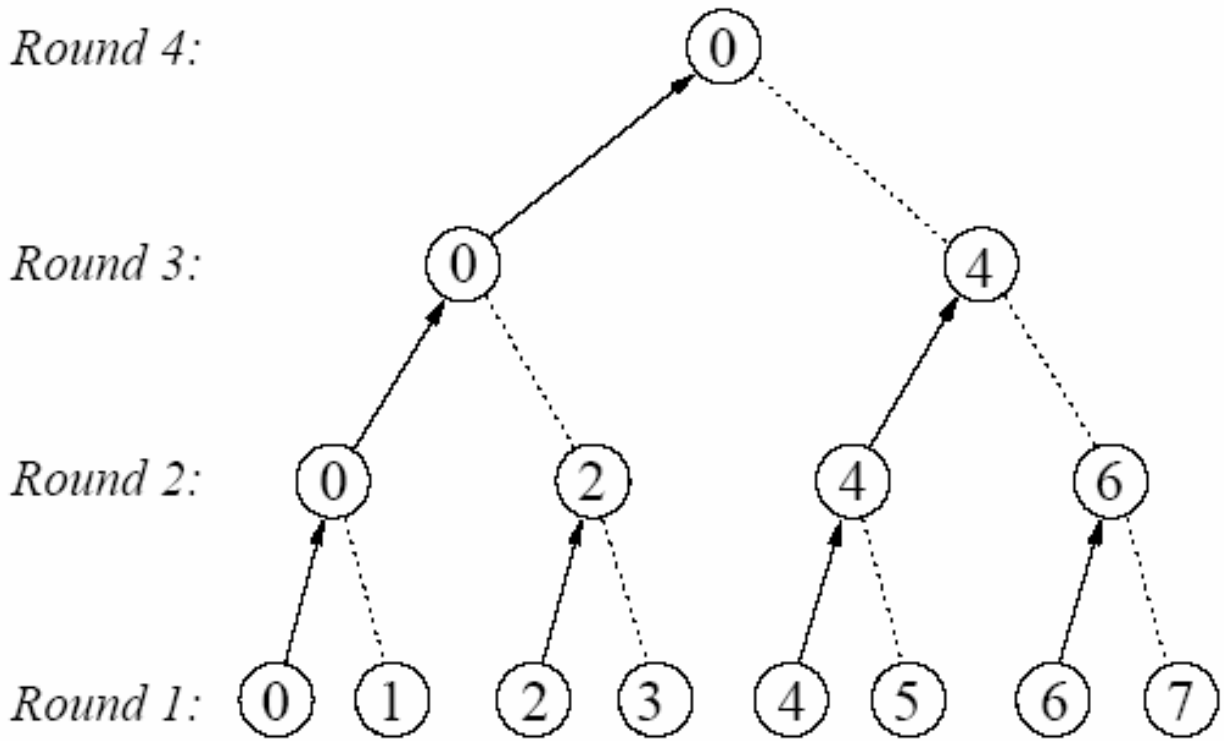

Figure 4.16 A binary tournament tree

Mellor-Crummey and Scott's tree barrier algorithm

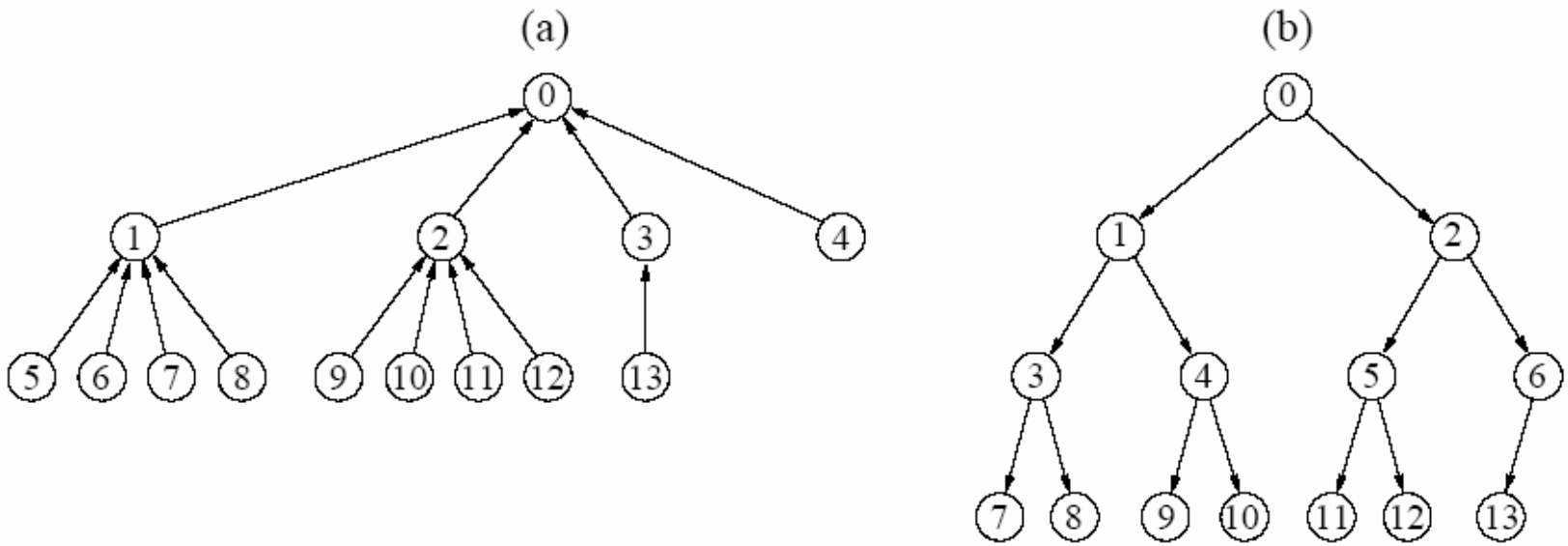

Figure 4.17 (a) Arrival and (b) wakeup trees in Mellor-Crummey and Scott's tree barrier algorithm

Other methods

- Non-blocking and Wait-free Synchronisation
  - Avoiding deadlock caused by thread-errors

Conclusion

- All parallel systems require synchronisation
- Scientific problems often need barriers
  - Differential equations solved in small time steps

Debate

- Questions?