The Need for Synchronization

- Multiprogramming
  - “logical” concurrency: processes appear to run concurrently although there is only one PC

- Multiprocessing (and multithreading)
  - Concurrency can be “logical” and “physical”

- Concurrent processes must
  - Protect some common data so that there are orderly updates
    - Competing for access (mutual exclusion; critical sections)
  - Coordinate their relative progress
    - Producer-consumer relationships; barriers
Example

Process P1

A ← A + 1
ld R1, A
addi R1, R1, 1
st R1, A

A ← A + 2
ld R1, A
addi R1, R1, 2
st R1, A

Original sequence (result either A+1, A+2, or A+3)

ld R1, A
addi R1, R1, 2
st R1, A
addi R1, R1, 1
st R1, A

A timing sequence giving the result A+1
Synchronization

• **Locking**
  – Critical sections
  – Mutual exclusion
  – Used for exclusive access to shared resource or shared data for some period of time
  – Efficient update of a shared (work) queue

• **Barriers**
  – Process synchronization -- All processes must reach the barrier before any one can proceed (e.g., end of a parallel loop).
Locking

• Typical use of a lock:
  
  ```c
  while (!acquire (lock))  /*spin*/
  ;
  /* some computation on shared data*/
  release (lock)
  ```

• Acquire based on primitive: Read-Modify-Write
  – Basic principle: “Atomic exchange”
  – Test-and-set
  – Fetch-and-add
Test-and-set

- Lock is stored in a memory location that contains 0 or 1
- Test-and-set (attempt to acquire) writes a 1 and returns the value in memory
- If the value is 0, the process gets the lock; if the value is 1 another process has the lock.
- To release, just clear (set to 0) the memory location.
Atomic Exchanges

• Test-and-set is one form of atomic exchange
• Atomic-swap is a generalization of Test-and-set that allows values besides 0 and 1
• Compare-and-swap is a further generalization: the value in memory is not changed unless it is equal to the test value supplied
Fetch-and-T

- Generic name for `fetch-and-add`, `fetch-and-store` etc.
- Can be used as test-and-set (since atomic exchange) but more general. Will be used for barriers (see later)
- Introduced by the designers of the NYU Ultra where the interconnection network allowed combining.
  - If two `fetch-and-add` have the same destination, they can be combined. However, they have to be forked on the return path
Full/Empty Bits

- Based on producer-consumer paradigm
- Each memory location has a synchronization bit associated with it
  - Bit = 0 indicates the value has not been produced (empty)
  - Bit = 1 indicates the value has been produced (full)
- A write stalls until the bit is empty (0). After the write the bit is set to full (1).
- A read stalls until the bit is full and then empty it.
- Not all load/store instructions need to test the bit. Only those needed for synchronization (special opcode)
- First implemented in HEP and now in Tera.
Faking Atomicity

• Instead of atomic exchange, have an instruction pair that can be deduced to have operated in an atomic fashion

• Load locked (ll) + Store conditional (sc) (Alpha)
  – sc detects if the value of the memory location loaded by ll has been modified. If so returns 0 (locking fails) otherwise 1 (locking succeeds)
  – Similar to atomic exchange but does nor require read-modify-write

• Implementation
  – Use a special register (link register) to store the address of the memory location addressed by ll. On context-switch, interrupt or invalidation of block corresponding to that address (by another sc), the register is cleared. If on sc, the addresses match, the sc succeeds
### Example revisited (Process P1)

<table>
<thead>
<tr>
<th>Loop: Test-and-set</th>
<th>Fetch-and-increment A</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>bnz</code> R2,loop</td>
<td>(b)</td>
</tr>
<tr>
<td><code>ld</code> R1,A</td>
<td></td>
</tr>
<tr>
<td><code>addi</code> R1,R1,A</td>
<td></td>
</tr>
<tr>
<td><code>st</code> R1,A</td>
<td></td>
</tr>
<tr>
<td><code>st</code> R0, lock</td>
<td></td>
</tr>
<tr>
<td>(a)</td>
<td></td>
</tr>
</tbody>
</table>

| (b)               |                       |
| `loop: ll R1,A    |                       |
| `addi` R1,R1,1    |                       |
| `sc` R1,A         |                       |
| `bz` R1,loop      | (c)                   |

Synchron. CSE 471
Spin Locks

- Repeatedly: try to acquire the lock
- Test-and-Set in a cache coherent environment (invalidation-based):
  - Bus utilized during the whole read-modify-write cycle
  - Since test-and-set writes a location in memory, need to send an invalidate (even if the lock is not acquired)
  - In general loop to test the lock is short, so lots of bus contention
  - Possibility of “exponential back-off” (like in Ethernet protocol to avoid too many collisions)
Test and Test-and-Set

• Replace “test-and-set” with “test and test-and-set”.
  – Keep the test (read) local to the cache.
  – First test in the cache (non atomic). If lock cannot be acquired, repeatedly test in the cache (no bus transaction)
  – On lock release (write 0 in memory location) all other cached copies of the lock are invalidated.
  – Still racing condition for acquiring a lock that has just been released. (O(n²) worst case bus transactions for n contending processes).
• Can use ll+sc but still racing condition when the lock is released
Queuing Locks

- Basic idea: a queue of waiting processors is maintained in shared-memory for each lock (best for bus-based machines)
  - Each processor performs an atomic operation to obtain a memory location (element of an array) on which to spin
  - Upon a release, the lock can be directly handed off to the next waiting processor
Software Implementation

`lock` struct {int Queue[P]; int Queuelast;} /*for P processors*/
/*Initially all Queue[i] are 1 except for Queue[0] = 0 */
Queuelast := 0;
ACQUIRE  `myplace` := fetch-and-add (`lock`->Queuelast);
   while (`lock`->Queue[`myplace` modP] == 1; /* spin*/
   `lock`->Queue[`myplace` modP] := 1;

RELEASE `lock`->Queue[(`myplace` + 1) modP] := 0;
– The Release should invalidate the cached value in the next processor that can then fetch the new value stored in the array.
Queuing Locks (hardware implementation)

- Can be done several ways via directory controllers
- Associate a syncbit (aka, full/empty bit) with each block in memory (a single lock will be in that block)
  - Test-and-set the syncbit for acquiring the lock
  - Unset to release
  - Special operation (QOLB) non-blocking operation that enqueues the processor for that lock if not already in the queue. Can be done in advance, like a prefetch operation.
- Have to be careful if process is context-switched (possibility of deadlocks)
Transactions

• Critical section behaves as a transaction (in the database sense)
  – All instructions within the transaction commits or none of them does
• Many possible implementations
• In “transactional memory” proposals, use of a hardware assisted optimistic control
  – Might become important in CMPs
  – Might become a basic construct of a “parallel programming language”
Transaction Implementation without software support (overview)

• On transaction-begin start save process state
  – Begin can be recognized by an atomic instruction (test-and-set etc.)
  – Save registers, register maps etc.
• During transaction, buffer all state changes
  – Register and memory stores
  – Registers in ROB like for branch prediction
  – For caches, use a status bit and keep old values in a log
• On transaction-end, check if conflicts with other running transactions
  – End can be recognized by “store to lock location”
  – Conflicts can be messages for invalidations (and abort of transactions that receive such invalidations)
  – Long transactions … lots of messages
• This is the tip of the iceberg (but there are papers giving full implementations)
  – For example, what to do on cache evictions etc
Barriers

• All processes have to wait at a synchronization point
  – End of parallel do loops
• Processes don’t progress until they all reach the barrier
• Low-performance implementation: use a counter initialized with the number of processes
  – When a process reaches the barrier, it decrements the counter (atomically -- fetch-and-add (-1)) and busy waits
  – When the counter is zero, all processes are allowed to progress (broadcast)
• Lots of possible optimizations (tree, butterfly etc. )
  – Is it important? Barriers do not occur that often (Amdahl’s law….)
Sequential Consistency – Example 1

Process P1
write (A);
flag := 1;

Process P2
while (flag != 1);    /*spin on flag*/
read (A);

• What does the programmer expect?
  – Producer-consumer relationship (P1 producer; P2 consumer)
  – But … what if NUMA and writing of flag is much faster than writing of A?
Sequential Consistency – Example 2

Process P1
X := 0;
....
X := 1;
If (Y = 0) then Kill P2

Process P2
Y := 0;
....
Y := 1;
If (X = 0) then Kill P1

Programmer expects P1 or P2 or both to go on
But what if loads bypass stores; or write buffers; or…
Sequential consistency

• Seq. consistency is the behavior that programmer expects:
  – Result of any execution is the same as if the instructions of each process were executed in some sequential order
  – Instructions of each process appear in program order in the above sequential order
• Equivalent to say
  – Memory operations should proceed in program order
  – All writes are “atomic” i.e., seen by all processors in same order
• What about all our optimizations (write buffers, load speculation etc)?
  – Keep them but make everything “speculative” (e.g., on a load with a previous store conflict, prefetch. If invalidated, abort before commit)
  – Provide a relaxed model of memory consistency
Relaxed Models of Memory Consistency

- Seq. consistency = total order of loads and stores
- Relaxed models
  - Weak ordering
  - Load and store selectively used as lock/unlock operations
  - Need some “fence” operations to flush out buffers
  - Reasoning needed by programmer quite subtle
- A simple relaxed model: Processor consistency
  - Stores are totally ordered (so write buffers are FIFO)
  - Loads can bypass stores
  - Works for Example 1
  - Does not work for Example 2! Programmers still need to be careful.