Parallelization and Synchronization

CS165 – Section 8
Multiprocessing & the Multicore Era

- Single-core performance stagnates (breakdown of Dennard scaling)
- Moore’s law continues → use additional transistors to duplicate logic → additional cores
Concurrent Computing

- Single-core programs usually cannot automatically benefit from modern multi-core hardware.
- Adding multiple processors does not help, if the programmer is not aware of how to use them → programmers need to structure problems as a set of parallel tasks
- **Parallel computing** uses multiple processors to solve problems by dividing it into subtasks that can be executed at the same time.
Race Conditions & Critical Sections

- **Race Condition**: A situation in which the output depends on the sequence/timing of threads/processes. Race conditions are often difficult to debug as the output is nondeterministic and often disappears when using debuggers or extensive logging (see “Heisenbug”).

- **Critical Section**: Some parts of a parallel program may not be concurrently executed by more than one process. Those parts are called *critical section*.
Example: Linked List Delete Operation

```
Initial State
```

![Diagram of initial state of a linked list with nodes i-1, i, i+1, and i+2]
Example: Linked List Delete Operation (cont’)
Example: Linked List Delete Operation (cont’)

Linked List After Second (Parallel) Delete Operation
Example: Linked List Delete Operation (cont’)

Resultant Linked List
Example: Linked List Delete Operation (cont’)

Resultant Linked List
Barriers

Idea: Synchronizes threads so that no threads continues execution until all threads have passed the synchronization point.

Example:

```c
// Something (e.g. parallel insert)...
barrier(b);
// Something else (e.g. parallel reads)...
```
Mutual Exclusion - Locks

**Problem:** How to ensure that no two concurrent processes are in their critical section at the same time.

**Solution:** Different ways of handling mutual exclusion. Either with locks or atomic operations.

**Traditional Solution:**
```c
mutex_lock(m);
// Sequential part ...
mutex_unlock(m);
```
Mutual Exclusion - Locks (cont’)

Be careful to avoid deadlocks!

Thread 1

mutex_lock(m1);
...
mutex_lock(m2);
...
// Do something...
...
mutex_unlock(m2);
...
mutex_unlock(m1);
Mutual Exclusion - Locks (cont’)

Be careful to avoid deadlocks!

```c
Thread 1
mutex_lock(m1);
...
mutex_lock(m2);
...
// Do something...
...
mutex_unlock(m2);
...
mutex_unlock(m1);
```

```c
Thread 2
...
mutex_lock(m2);
...
mutex_lock(m1);
...
// Do something
...
mutex_unlock(m1);
...
mutex_unlock(m2);
```
Mutual Exclusion - Locks (cont’)

Be careful to avoid deadlocks!

Thread 1

mutex_lock(m1);
...
mutex_lock(m2);
...
// Do something...
...
mutex_unlock(m2);
...
mutex_unlock(m1);

Thread 2

...
mutex_lock(m2);
...
mutex_lock(m1);
...
// Do something
...
mutex_unlock(m1);
...
mutex_unlock(m2);
Mutual Exclusion - Locks (cont’)

Be careful to avoid deadlocks!

Thread 1

mutex_lock(m1);
...
mutex_lock(m2);
...
// Do something...
...
mutex_unlock(m2);
...
mutex_unlock(m1);

Thread 2

...  
mutex_lock(m2);
...
mutex_lock(m1);
...
// Do something
...
mutex_unlock(m1);
...
mutex_unlock(m2);
Be careful to avoid deadlocks!

Thread 1

```c
mutex_lock(m1);
...
mutex_lock(m2);
...
// Do something
...
mutex_unlock(m2);
...
mutex_unlock(m1);
```

Thread 2

```c
...
mutex_lock(m2);
...
mutex_lock(m1);
...
// Do something
...
mutex_unlock(m1);
...
mutex_unlock(m2);
```
Condition Variables

Threads often need to wait for other threads to finish a specific task → condition variables allow a thread to wait for a particular condition to become true

Typical operations:

- `wait(condition, lock)` - release lock, put thread to sleep until condition is signaled; when thread wakes up again, re-acquire lock before returning.
- `signal(condition, lock)` - if any threads are waiting on condition, wake up one of them. Caller must hold lock, which must be the same as the lock used in the wait call.
- `broadcast(condition, lock)` - same as signal, except wake up all waiting threads.
Example: Concurrent List

define(prev_node, node){
    mutex_lock(m);
    // Check that prev_node is still pointing to node
    if(prev_node && (prev_node.next == node)){
        prev_node.next = node.next;
        free(node);
    }
    mutex_unlock(m)
}
Atomic Operations

Idea: Avoid mutual exclusion using atomic operations that are guaranteed to be executed as an atomic unit.

Typically employ direct hardware support

Benefits:
- Relatively quick
- No deadlocks

Typical atomic operations:
- `x.fetch_and_store(y)` - do `x=y` and return the old value of `x`
- `x.fetch_and_add(y)` - do `x+=y` and return the old value of `x`
- `x.compare_and_swap(y, z)` - if `x` equals `z`, then do `x=y`. In either case, return old value of `x`.

NOTE: Atomic operations provided the basis for most modern higher level synchronization primitives
Example: Lock-free Concurrent List

delete(prev_node, node){
    // If the ‘prev_node’ was not changed, atomically skip ‘node’
    if(prev_node.next.compare_and_swap(node.next,node) == node){
        free(node);
    }
}
Project Milestone 3

- Move execution to the next level, by parallelizing execution of incoming queries
- Numerous opportunities:
  - Use a thread pool to manage incoming queries
  - Use Intra- and Inter-query parallelism
- Scans & Filters:
  - Chop up data into chunks that can be scanned in parallel (NOTE: data placement matters → see memory hierarchy slides for more information about NUMA)
  - **Scan Sharing:** Share intermediate results between queries by executing multiple queries as a batch (batching also has benefits in other components, such as overlapping I/O in latency bound operations)
Parallelism in Data Systems

Why:
- Better utilization of resources through overlap of I/O and computation
- Efficient utilization of modern server hardware (multi-core and multi-socket)
- Avoid idle times during long-running transactions

How:
*Parallel query processing*
- Inter-query parallelism
- Intra-query parallelism
Inter-Query Parallelism

Run multiple queries and/or transactions in parallel

“Easy” to implement in shared memory parallel systems
(Lock/latch shared resources, keep the general flow the same)
Intra-query Parallelism

Divide single queries into parallelizable sub-tasks

**Techniques:**
- I/O-aware parallel scheduling of independent sub-operations
- Partition base- and/or intermediate data
Simple Server Loop

```java
server()
    {
        while(true)
        {
            request = accept_request();
            handle_request(request);
        }
    }

Idea: Sequentially handle incoming requests (one at a time)

Pros: Straight forward design; only one execution path no synchronization needed
Cons: Cannot exploit multi-core hardware, Long I/O wait times → Hardware under-utilized
Multithreaded Server Loop

server()
    while(true){
        request = accept_request();
        thread_create(handle_request(), request);
    }
}

Idea: Spawn a separate thread for every incoming request

Benefits:
- Allows us to handle multiple requests in parallel → better utilization of multi-processors
- Overlap I/O and computation, hide memory access latencies, better utilize caches
Thread Pools

**Problem:** Unbounded number of threads can reduce overall throughput

**Solution:** Allocate a bounded pool of threads to limit amount of parallelism, queue up pending tasks

Thread Pools (cont’)

**Server Thread**

```java
server() {
    queue = new ConcurrentQueue();
    allocate_threads(queue);
    while(true){
        // Enqueue into work queue
        request = accept_request();
        enqueue(queue, request);
        wake_up(queue);
    }
}
```

**Worker Threads**

```java
worker() {
    while(true){
        // Dequeue form work queue
        if(r = dequeue(queue, request)){
            handle_request(r);
        } else {
            sleep_on(queue);
        }
    }
}
```
More Things to Consider

- **NUMA**: Data partitioning & replication to increase bandwidth and reduce latency on NUMA architectures

- **Multiple Machines**: Scaling out to multiple machines in a network poses its own unique challenges (varying latencies/bandwidths, fault-tolerance, geographically distributed data centers etc.)