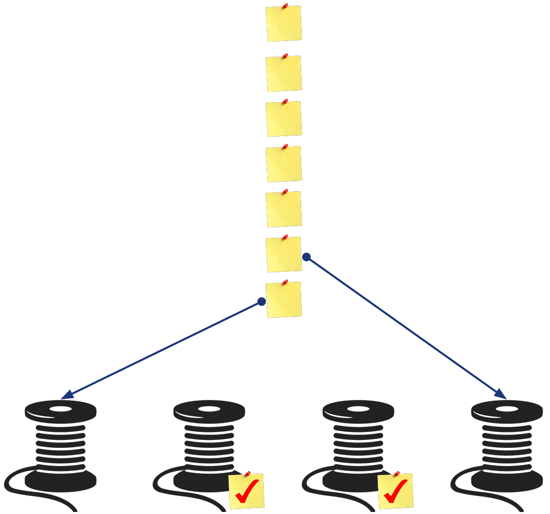
Threads Kept and Better Managed

NUMA Aware Thread Pool

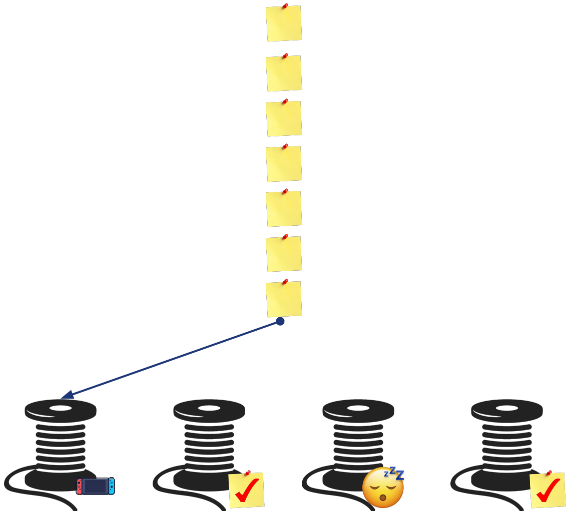
Ricky, Deepayan, & Emmanuel

Traditional Thread Pool



- Maintain a set of threads and a queue
- Threads pull tasks ( ) from the queue and execute them ( )
- Tasks are usually queries to be executed and are added to the queue from the execution layer

Brief Overview of Design, based on Hyper's morsels

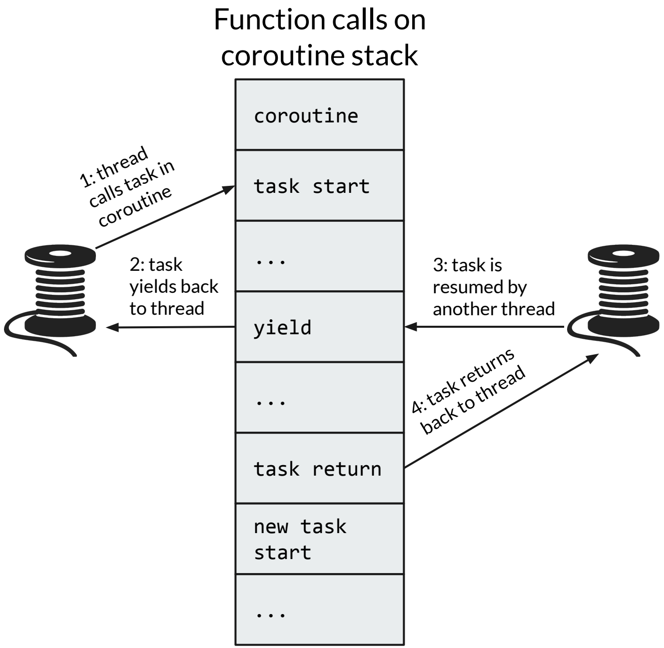


- Every thread has a state:
  - Busy: working on task (✅)
  - Switching: finding another task (🎮)
  - Parked: sleeping, no tasks available (😴)

Brief Overview of Design, based on Hyper's morsels

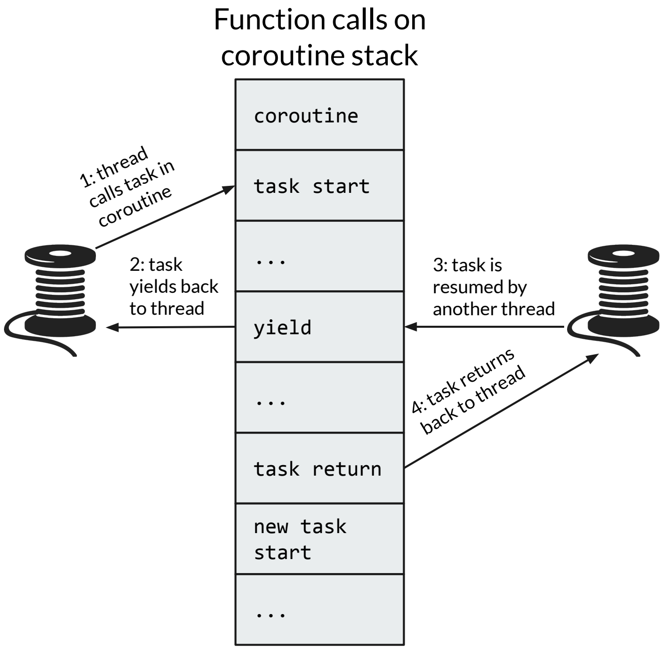
- Every thread has a state:
  - Busy: working on task (📝)
  - Switching: finding another task (🎮)
  - Parked: sleeping, no tasks available (😴)
- Maintain per-NUMA region queues
  - Threads pull from their region's queue if able
  - Pull from another region's queue if no tasks in region
- Tasks are tagged by region and added to appropriate queue

How tasks yield



- Originally tasks were just `std::functions`
- But we wanted tasks to be able to pause and resume their execution
  - Ex. task tries to get lock, disk I/O
- Tasks become coroutines
  - Specifically stackful coroutines
  - We want tasks to be able to call functions that can also be able to yield
  - Every coroutine must allocate its own stack :(
  - Fix this by pooling stacks across the thread pool
- We still want the user of pool to be able to write `std::functions`
  - Coroutine calls function
  - Function takes in context argument that allows it to yield coroutine

How tasks yield



- When new task is added it is assigned to a stack from stack pool
- Threads in pool then execute:
  1. Pull task from queue
  2. Run task until it yields or returns
  3. If it yields, return it to queue
  4. If it returns, return coroutine stack to stack pool

Status Update

- ✓ Make Terrier NUMA Aware in the Thread Pool (100%)
  - ✓ Track where blocks are kept
  - ✓ Enable scanning of tables by NUMA region (75%, last update)
  - ✓ Ideally execute scanning of a table in parallel by NUMA region
  - ✓ Do this in cores in the NUMA region
- ✓ Integrate Latches with Thread Pool (125%)
  - ✓ Add coroutine support to thread pool and latches
  - ✓ Add latching support to DataTable
  - ✓ Enable stack recycling between tasks

Our APIs

- Scanning by NUMA Region:

```
// r is a numa_region_t in scope
for (DataTable::NumaIterator it = table->begin(r); it != table->end(r); it++) {...}
```

- Adding tasks to a thread pool

```
void ExecutionThreadPool::SubmitTask(promise<void> *promise,  function<void(PoolContext *)> &task,
numa_region_t numa_hint = UNSUPPORTED_NUMA_REGION)
void ExecutionThreadPool::SubmitTask(promise<void> *promise,  function<void()> &task, numa_region_t
numa_hint = UNSUPPORTED_NUMA_REGION)
```

- Yielding inside of a task

```
// ctx is a PoolContext* in scope
ctx->YieldToPool();
```

Testing

- Check whether data is stored on the NUMA region that the metadata stored in the block indicates that it is
  - Ensures that our region tracking is accurate
- Check behavior of the thread pool
  - Check that threads are assigned to the right cores
  - Check that tasks are executed on the right cores
  - Check that the right tasks are executed in the right order
- Check that context switching is correctly executed
  - Make sure that a context switched task is started and switched out correctly

Quality

- High quality: Thread pool
  - Really clean code, easy to understand
  - Implements multiple interfaces to easily integrate with the rest of the system
- Medium quality: NUMA Awareness of RawBlocks
  - Due to the large OS dependency of NUMA APIs, this code is not very clean
- Low quality: Coroutines
  - This code seems clean but requires a great deal of oddities
  - The boost library does some weird stuff
    - Ex: signals exception to unroll the stack to enable deconstruction of coroutine (breaks ASAN)

Benchmarks

- We initially created a set of benchmarks that measured the performance of the different implementations we built:
  - Thread Pool and NUMA Awareness
    a) Baseline single-threaded iteration benchmark to determine the performance of a workload using the ThreadPool interface to scan through the table (~90M items/s)
    b) Same as *(a)* but we divide the scans to be NUMA aware, so the thread will read all tuples located on one NUMA region before switching to a different region (2.33x improvement over *a*)
    c) Same as *(b)* but we divide each NUMA region's workload to operate in parallel (4.15x improvement over *a*)
    d) Same as *(c)* but we now define assign the task associated with each NUMA region to operate on the specified region (4.5x improvement over *a*)
  - Context Switching Tasks
    a) We measure a benchmark for each iterator to scan its associated table using the thread pool interface (~1.7M items/s)
    b) We do the same as in *(a)* but allow the tasks to use the defined coroutines methods to switch upon encountering a lock (~600x improvement over *a*)

DEMO TIME!!!

Demo Results

Thread pool and NUMA Awareness

DataTableBenchmark/SingleThreadedIteration/manual_time	641 ms	0 ms	1	89.2675M items/s
DataTableBenchmark/NUMASingleThreadedIteration/manual_time	274 ms	0 ms	3	208.58M items/s
DataTableBenchmark/NUMAMultiThreadedIteration/manual_time	154 ms	0 ms	5	370.599M items/s
DataTableBenchmark/NUMAMultiThreadedNUMAAwareIteration/manual_time	140 ms	0 ms	5	408.135M items/s

Context Switching between Tasks

DataTableBenchmark/ConcurrentIterationNoContextSwitching/manual_time	682527 ms	11016 ms	1	1.67672M items/s
DataTableBenchmark/ConcurrentIterationWithContextSwitching/manual_time	1141 ms	10597 ms	1	1002.99M items/s

Benchmarks

- We run the following series of modifications to a final benchmark, which parallely iterates through a series of tables using the SlotIterator interface with high contention, to outline the performance of our implementation:
    a. Standard C++ threads, one thread per task
    b. TerrierThreads that use our defined ThreadPool and execution model (~7M items/s)
    c. Same as (*a*) but every task is associated with NUMA region (~3-5x improvement over *b*)
    d. Same as (*a*) but every task is able to context switch (~60-70x improvement over *b*)
    e. Same as (*a*) but every task is associated with NUMA region and is able to context switch (slight improvement over *d*)

DEMO TIME!!!

Demo Results

```
emmanuee@dev5:~/p3/terrier/build$ ./release/execution_thread_pool_benchmark
2020-05-02 16:19:51
Running ./release/execution_thread_pool_benchmark
Run on (40 X 2201 MHz CPU s)
CPU Caches:
  L1 Data 32K (x20)
  L1 Instruction 32K (x20)
  L2 Unified 1024K (x20)
  L3 Unified 14080K (x2)
-----------------------------------------------------------------------------------------------------------------------------------------
Benchmark                                                                                                  Time             CPU   Iterations
-----------------------------------------------------------------------------------------------------------------------------------------
ExecutionThreadPoolBenchmark/ConcurrentWorkload/min_time:3.000/manual_time                               82731 ms           74 ms            1   377.73k items/s
ExecutionThreadPoolBenchmark/ConcurrentThreadPoolWorkload/min_time:3.000/manual_time                      4099 ms           55 ms            1  7.44513M items/s
ExecutionThreadPoolBenchmark/ConcurrentNUMAThreadPoolWorkload/min_time:3.000/manual_time                  1439 ms           51 ms            3  21.2124M items/s
ExecutionThreadPoolBenchmark/ConcurrentThreadPoolWithYieldingWorkload/min_time:3.000/manual_time            65 ms           71 ms           57  468.111M items/s
ExecutionThreadPoolBenchmark/ConcurrentNUMAThreadPoolWithYieldingWorkload/min_time:3.000/manual_time        63 ms           69 ms           68  481.037M items/s
emmanuee@dev5:~/p3/terrier/build$ git pull
```

Future Work

- Intelligent block allocation policy
  - Have intelligent block allocation policy that decides which region blocks are allocated
  - Rebalance and correct for OS moves of blocks in GC
- Swap Space Awareness
  - Interact with block compacting
  - Lock hot blocks in RAM (mlock)
  - Allow cold blocks to be put into swap space (munlock)
- Interface with more concurrency primitives
  - Let conditional variables yield back to the thread pool