Avoiding Single Process Bottlenecks By Using Ets Concurrency

Jay Nelson @duomark https://github.com/duomark (Sponsored by TigerText)

Erlang Factory San Francisco, March 6, 2014

Disclaimer: Architectures are Neither Good nor Bad

- Architecture is a language of patterns
- A language encourages a preferred style
- Architect author's task is to express a style simply
- Pairing a style to an ill-suited problem IS bad

Patterns of erlang

- Watch for two patterns in this discussion
  - Cooperating set of OTP components
  - Replacing OTP components with alternatives
    - concurrently accessible data structures
    - collection of data in place of OTP constructs

- The ideas here relate to high-volume multicore erlang
  - Some issues may never occur for your environment

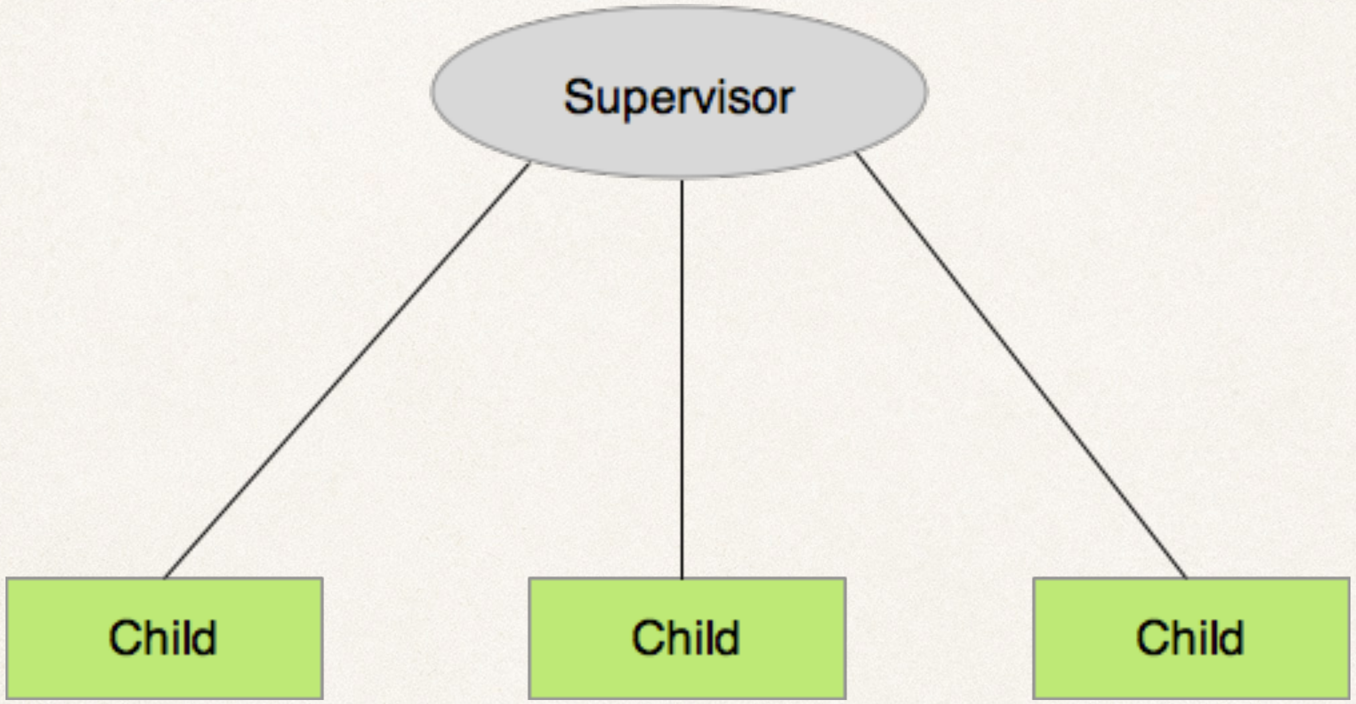
Part I: OTP Encourages Communicating Sequential Processes (CSP)

Erlang code is inherently single-threaded

- Many processes; each independent
- Per process features
  - separate control, stack and heap
  - self-contained memory space
  - dedicated garbage collector
  - process dictionary
  - message mailbox / queue

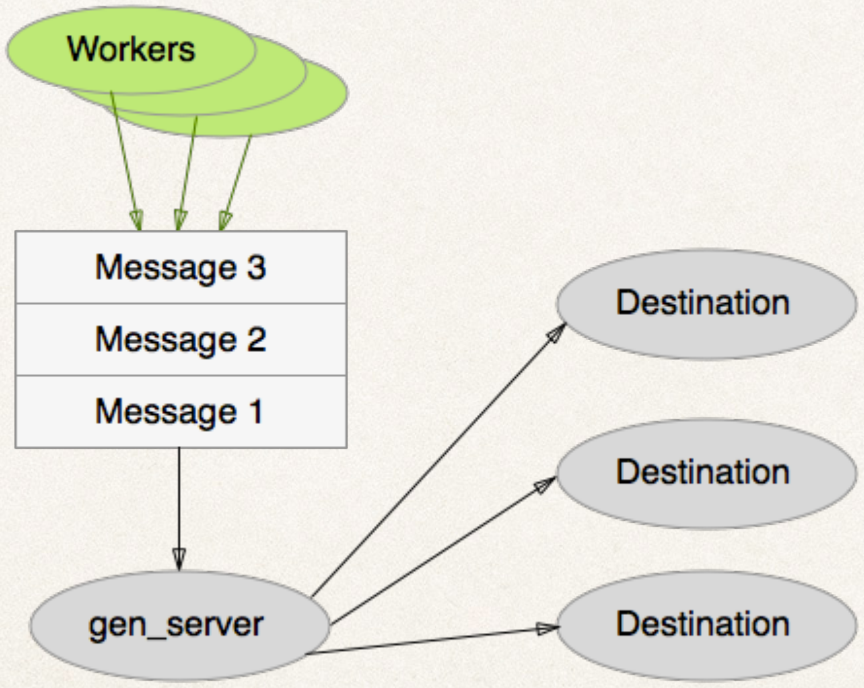
OTP encourages single process bottlenecks

- Process supervisors manage children individually



OTP encourages single process bottlenecks (cont.)

- Servers are central architectural concepts
  - serialize transactions
  - provide transaction independence



SIngle Process Bottleneck

OTP encourages single process bottlenecks (cont.)

- Servers organize computation
  - simplify reasoning about processing
  - support multi-process join / synch (i.e., wait)
- Servers are single process bottlenecks
  - limit transaction volume
  - choke systems under heavy load
  - long message queues result in timeouts and crashes

Language constructs are process local

- All data structures are in one process' memory space
- Caveat: binaries can be stored in shared heap
    - binary ref is transparent to erlang code
    - reference to binary is process local
    - memory optimization only, read-only construct

Part II: Erlang Term Storage (ets)

Ets features

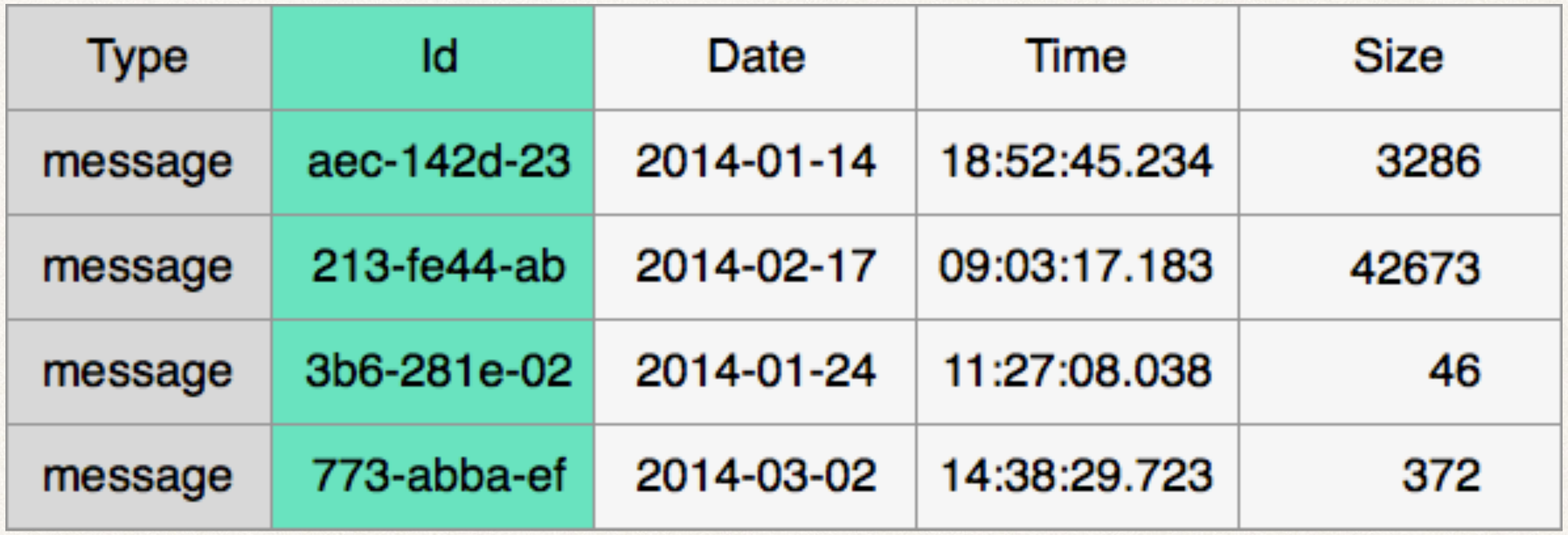
- Comes with the VM, part of OTP
- The one truly concurrent, cross-process data structure
  - key / value store (tuples hashed on one element)

- Lives in memory separate from processes
  - each tuple is accessible concurrently with others
  - simultaneous access to the same tuple is serialized

Ets table behavior

- An ets table is on par with a process
  - VM implements as in-memory data store using C
- Creating process is owner of the ets table
  - table is eliminated when owning process dies
- Is not garbage collected
  - user-managed with insert, update, delete semantics
  - allows multi-process access
  - excellent for large datasets

Ets table illustrated



Type	Id	Date	Time	Size
message	aec-142d-23	2014-01-14	18:52:45.234	3286
message	213-fe44-ab	2014-02-17	09:03:17.183	42673
message	3b6-281e-02	2014-01-24	11:27:08.038	46
message	773-abba-ef	2014-03-02	14:38:29.723	372

Key:	Id

Ets concurrency mechanics

- Table-level write lock
  - Use write_concurrency for multiple writers
  - Use read_concurrency on multicore infrequent write
  - Use both if access is big bursts of either
- Read/write locks for each tuple
- Multi-core simultaneous access on separate keys
- Accessing value copies data to process space
  - record with binary field values is very efficient

Caveats for ets concurrency

- Atomic operations: update_counter, update_element
- Beware read + update, NOT concurrent-safe
- Fold, select, first/next, et al are also NOT safe
- Be aware of which process is the owner of table

Ets concurrency strategies (partitioning)

- Kill owner process/delete table for fast garbage collect
- Split read data and write data to different tables
  - specify read_concurrency or write_concurrency
- Partition data by key
  - shard on key ranges
  - cascade tables for tree-based partitioning
  - separate pid per key for collision-free concurrency

Ets concurrency strategies (data access)

- Use update_counter / update_element for shared keys
  - Inc by zero to read int value from write-only table
- Meta-data / data set separation

- Public tables allow for read and write concurrency
  - requires cooperative trust of all functions
  - protected tables introduce single process bottleneck

Part III: Designing an Ets Solution

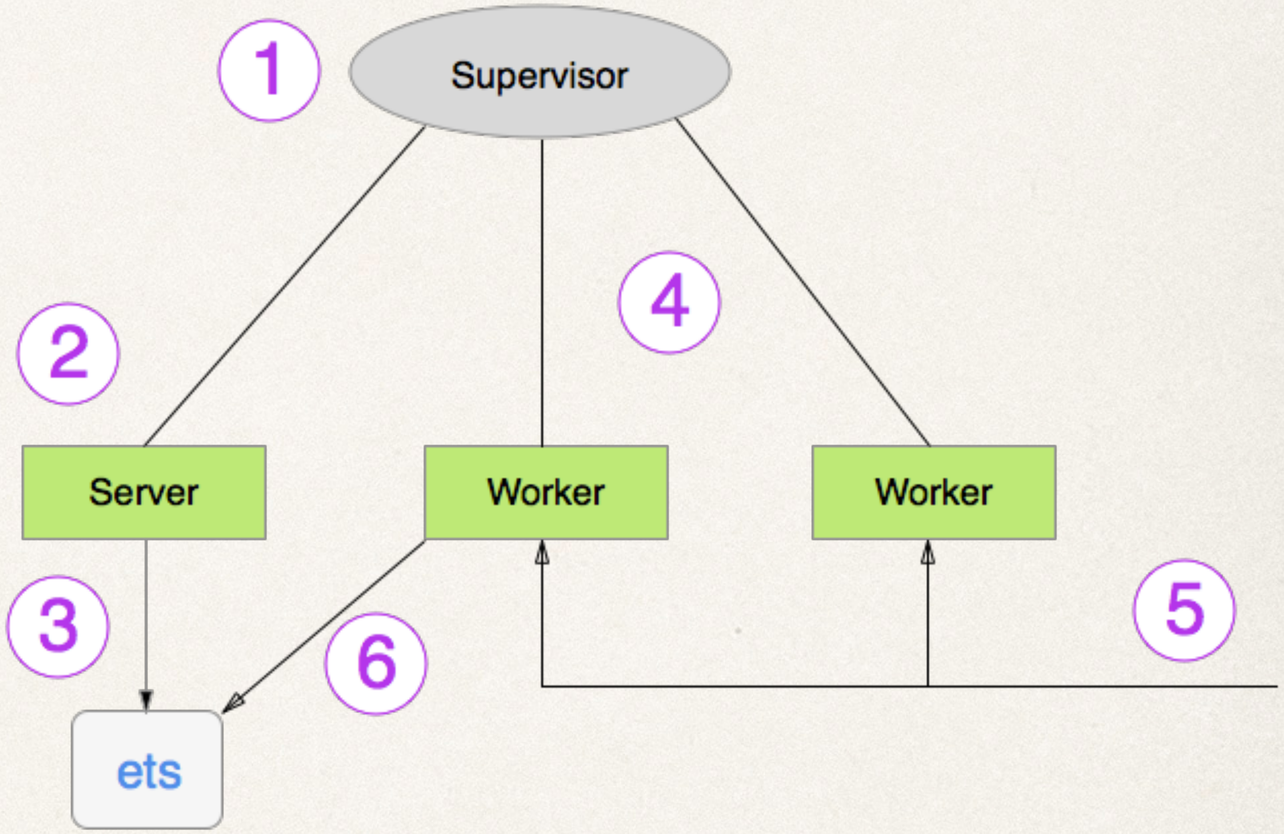
Use supervised processes to create ets

- Control when tables are created and destroyed
- Ensure a specific process is the guaranteed owner
- Don't ever create from random function calls

- You can probably avoid using owner inheritance
  - little benefit for added complexity
  - supervisors with rest-for-one are sufficient

Supervising ets creation

1. Rest-for-one supervisor
2. First server is created
3. Create ets table(s)
4. Create ets table workers
5. Signal kicks off worker processing
6. Workers access ets table(s)



Use atomic table-level operations

- New and rename can atomically create a new table
- Insert / insert_new atomically creates 1 to N objects
- Delete removes a single table atomically

Use atomic update_counter

- Only works on tables of set or ordered_set
- Cannot update the key element of a tuple
- Update_counter can add / sub integer from current value(s)
  - don't read ets before calling; returns int value after action
  - guaranteed atomic across all simultaneous access
  - allows multiple updates but only on same tuple
- Conditionals limited to replacing overmax/undermin result
  - dependent field decisions not possible
  - designed for warping counters that reach max value

Use atomic update_element

- Only works on tables of set or ordered_set
- Cannot update the key element of a tuple

- Clobbers existing value(s)
  - forced updates only, not based on current value
  - allows multiple updates but only on same tuple

Use reserve/write/publish semantics

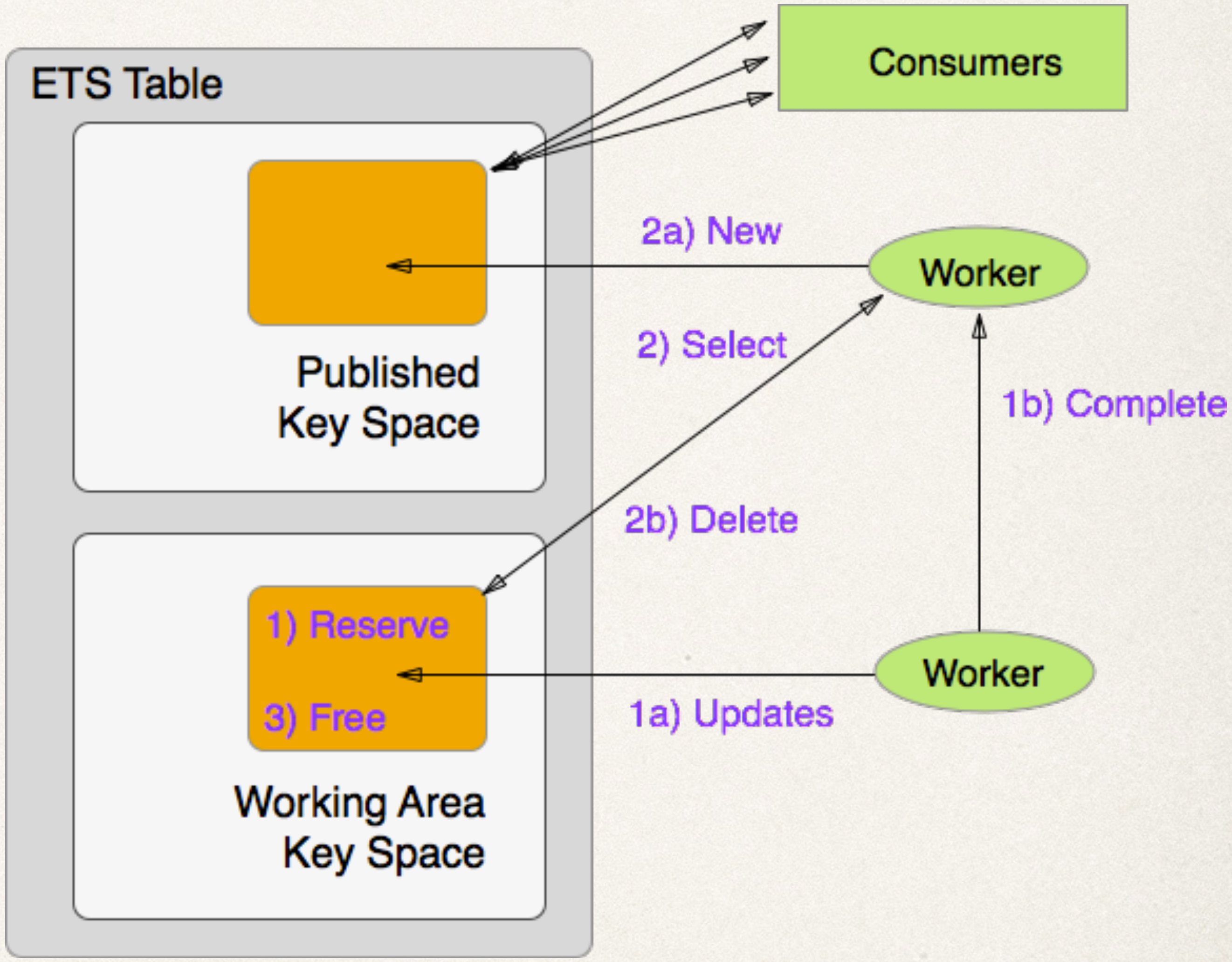
- Use update_counter to reserve int range of key(s)
- Non-atomic updates can be used to prepare new data
- Careful reservations allow rollbacks at any time
  - clear or delete intermediate data
  - unreserve atomically
    - mark to skip invalid data using update_element
    - adjust reservation indices with update_counter
- Use update_counter / update_element to publish

Use key partition for working area

- Reserve a portion of the key space
- Assemble working data in reserved space
- Signal unambiguously that data is complete
- Select finished data from working area
- Insert_new atomically creates all in active partition
- Common partition strategies (more are possible):
  - By pid or registered name prefix
  - By data content

Key Partition reserve area

1. Working partition reserved
    a) Working area updates
    b) Work completion signaled
2. Working copies selected
    a) New entries are created
    b) Working copies deleted
3. Working partition freed



Beware of non-atomic access

- Init_table and all multiple object deletes
- Iterators such as first/next and fold
  - includes all lookup/select/match functions
  - tab2list/tab2file
  - from_dets/to_dets
- **Any** write dependent on and subsequent to a read

Part IV: Erlang Patterns of Concurrency

https://github.com/duomark/epocxy

Github open source project

- OTP compatible library
    - Running in production at TigerText since Aug
    - Use as an included_application in *.app.src
- Implements ets-based concurrency constructs
- Hides complexity of correct atomic operations
- Provides an architectural API for concurrency

Firehose of data

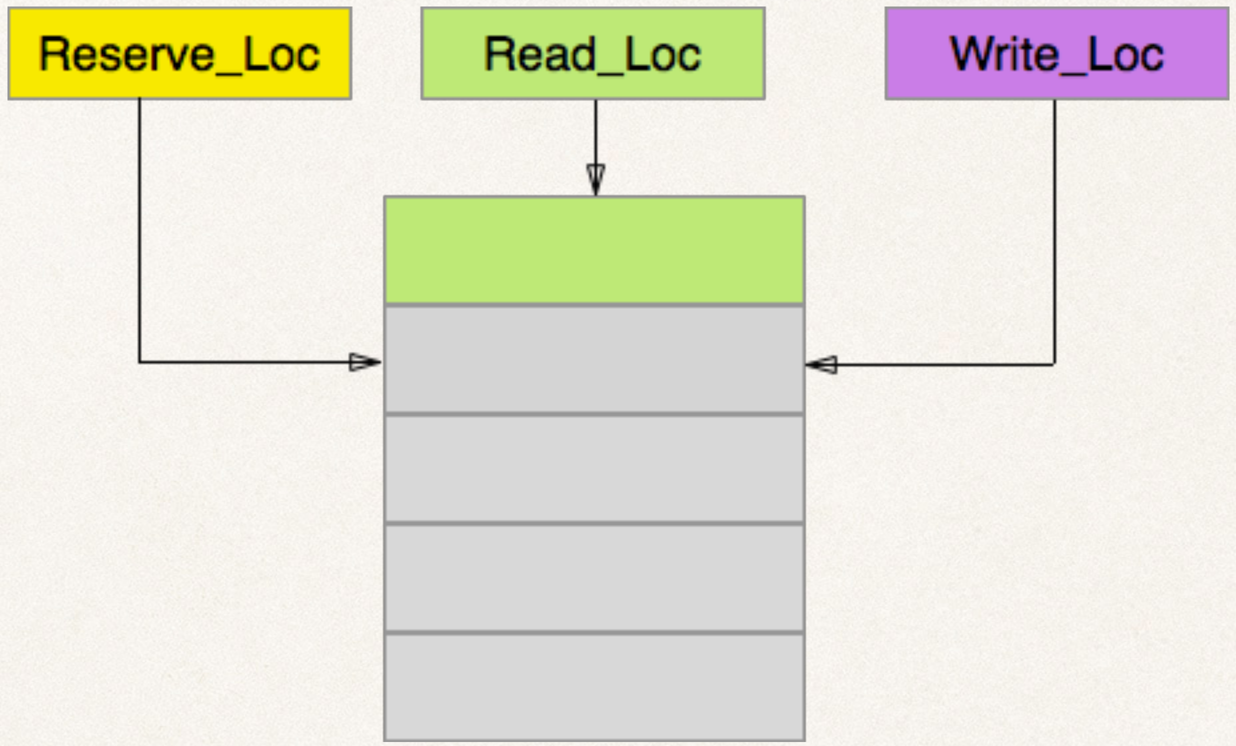
Controlled capture of concurrently arriving data

- FIFO, LIFO and Ring ets_buffer
- Implemented as an array in an ets ordered_set table
- Meta-data key space partitioned from data key space
  - {{meta, Task_Type}, Size, High_Water, Type, …}
  - {{Task_Type, Array_Index}, Create_Time, Data}
- All task_types share a single ets (named ‘ets_buffer’)
  - Non-dedicated buffers store data in metadata table
  - Dedicated buffers use separate ets table for content

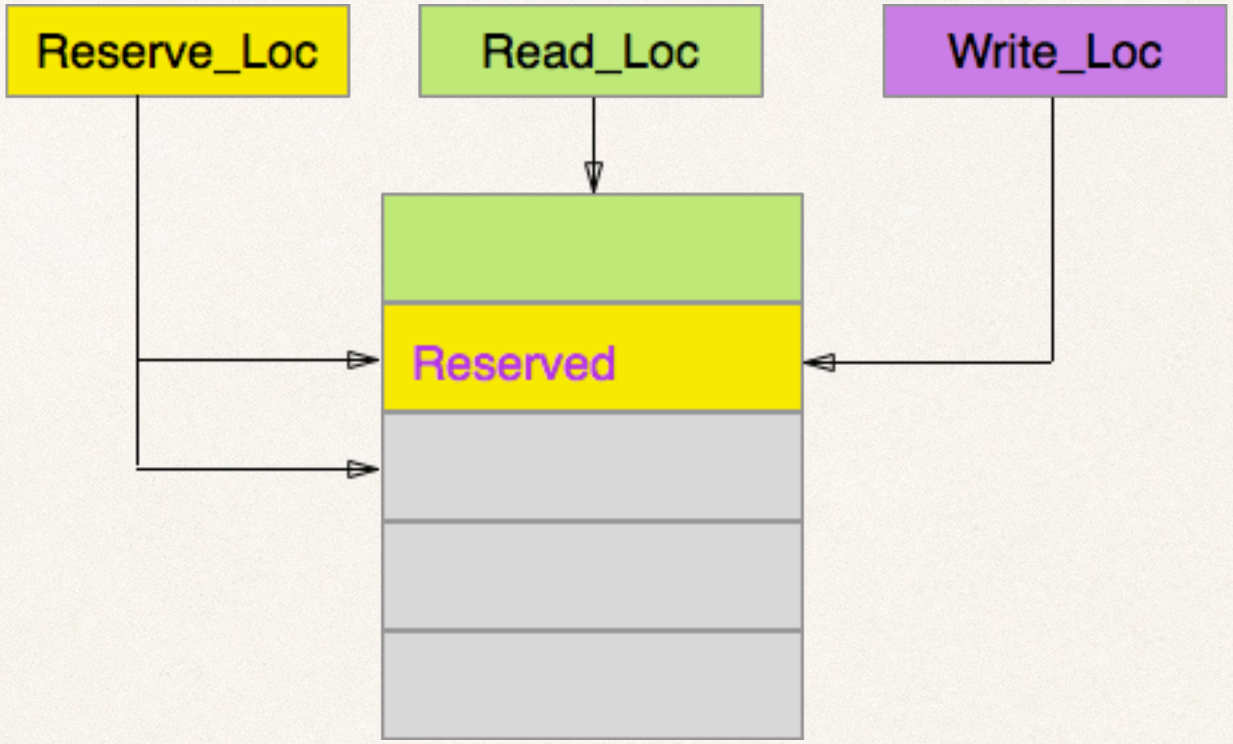
Ets_buffer implementation

- Write uses reserve / publish
  - Array index increment to reserve
  - Insert new value(s)
  - Publish new top of array
- Read uses reserve / retry
  - Bump array index to reserve
  - Read / retry entry later
  - Delete

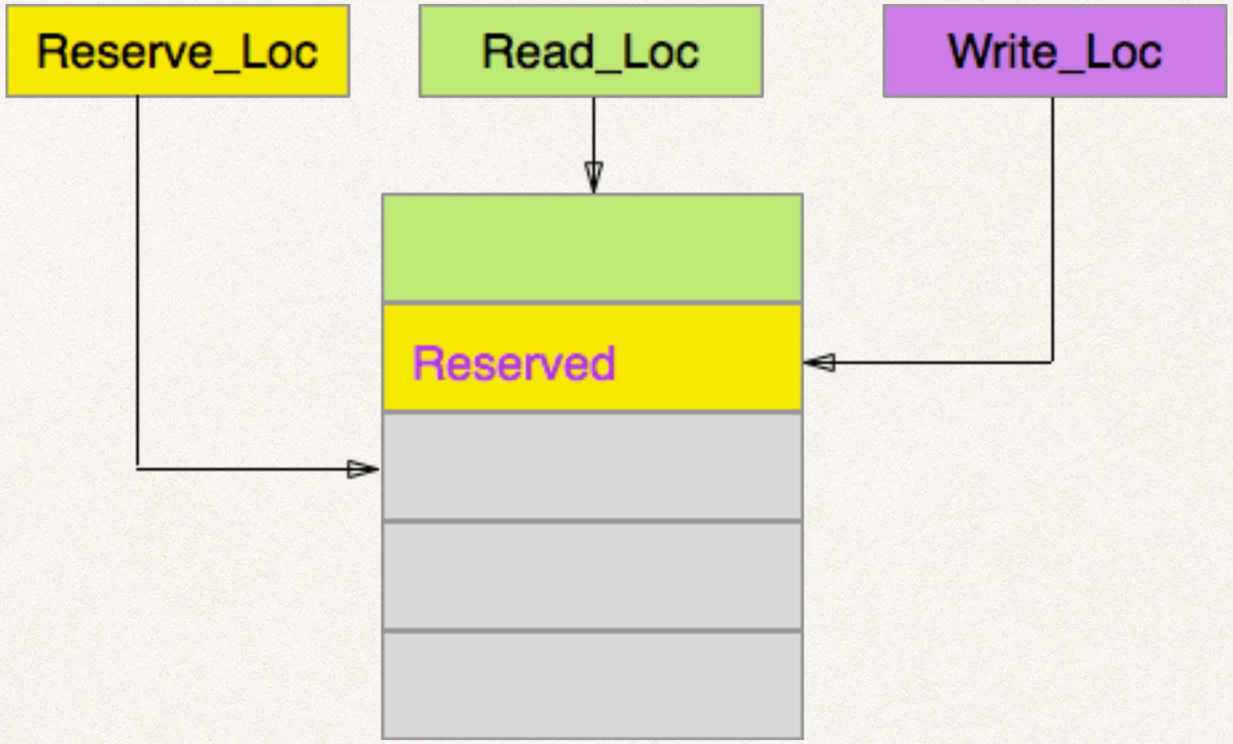
Ets_buffer illustrated



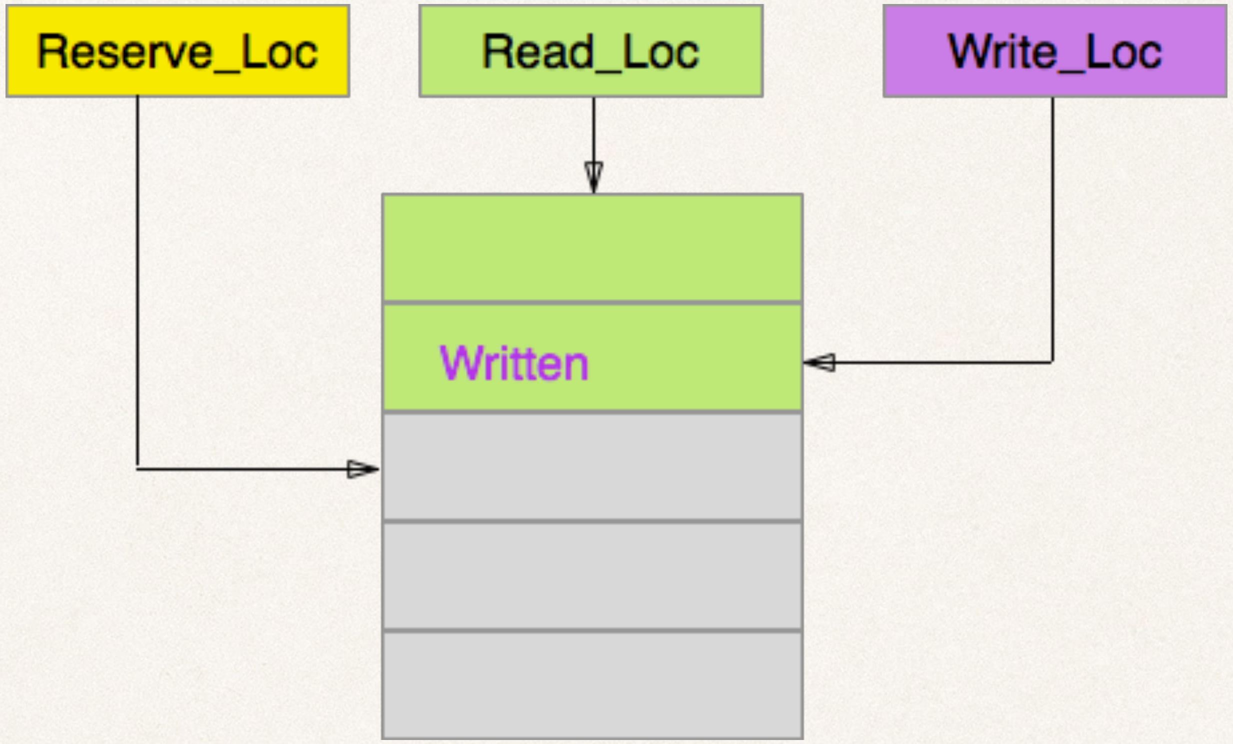
Ets_buffer illustrated



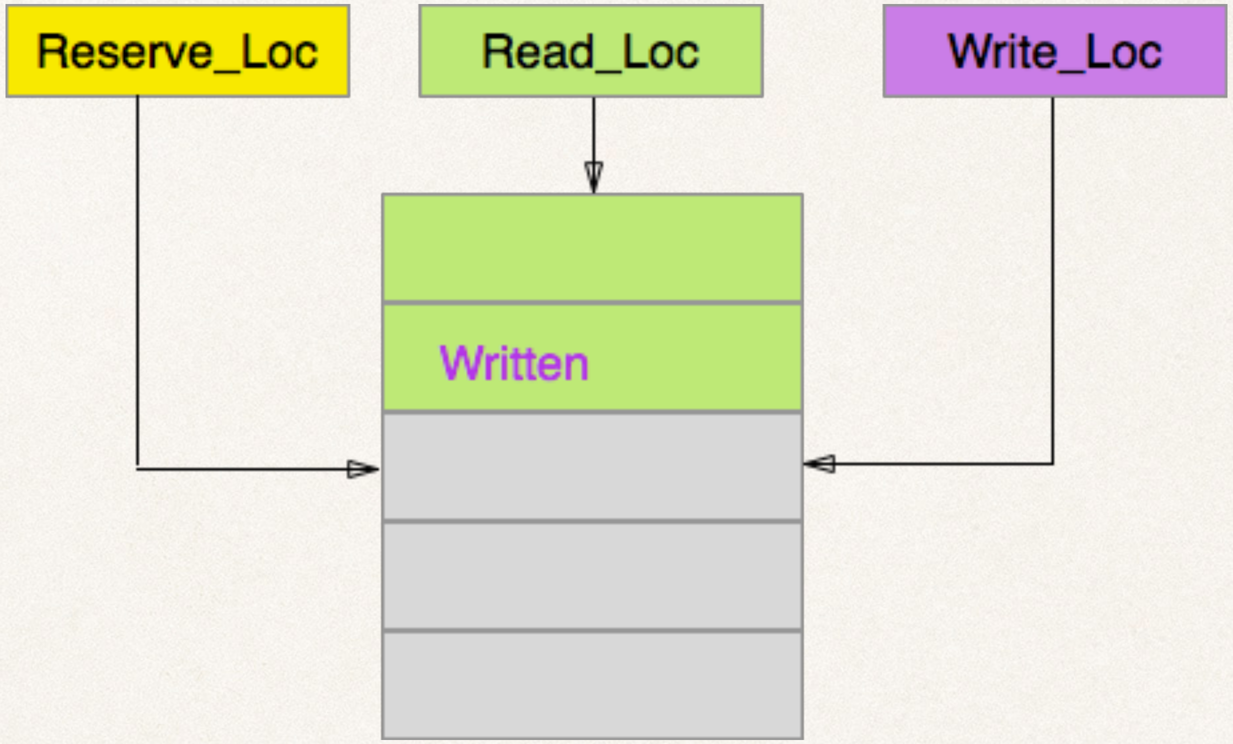
Ets_buffer illustrated



Ets_buffer illustrated



Ets_buffer illustrated



Ets_buffer issues

- Array index may use bignums if running long enough
- Currently all three share same code base
  - Distributed, concurrent LIFO arrays are hard
  - FIFO and Ring know number of elements
  - LIFO does not
- Potential enhancements
  - Linked list LIFO implementation instead
  - Ring read vs write is not well distinguished yet

Unbridled Concurrency

Erlang Factory San Francisco, March 6, 2014

Unbridled concurrency (cont.)

- Use spawn whenever concurrency needed
  - Concurrency can exceed CPU capabilities
  - No back pressure on requests
  - Load spikes can cause VM exhaustion
- Leads to erroneous use of worker pools
  - fraught with single process bottleneck symptoms
  - timeouts / restarts create cascading storms of data

Bridled concurrency

Bridled concurrency (cont.)

- Configure concurrency by type (atom)
  - Invoke spawns using concurrency type
  - Each type has a max simultaneous concurrency limit
  - Can spawn (async) or execute to get return value (sync)
- Timing of execution optionally recorded automatically
- M:F(A) captured on spawns (dangerous memory usage potential)
- Options when limit exceeded
  - execute inline (CPU back pressure)
  - refuse to execute (user-provided back pressure logic)

Cxy_ctl implementation

- Central ets for all concurrency limits
- Separate ets for init args and spawn / execute timing
  - init arg recording could cause OOM
  - timing recorded when process ends
  - library may need to record timing incrementally

Cache Expiration Overload

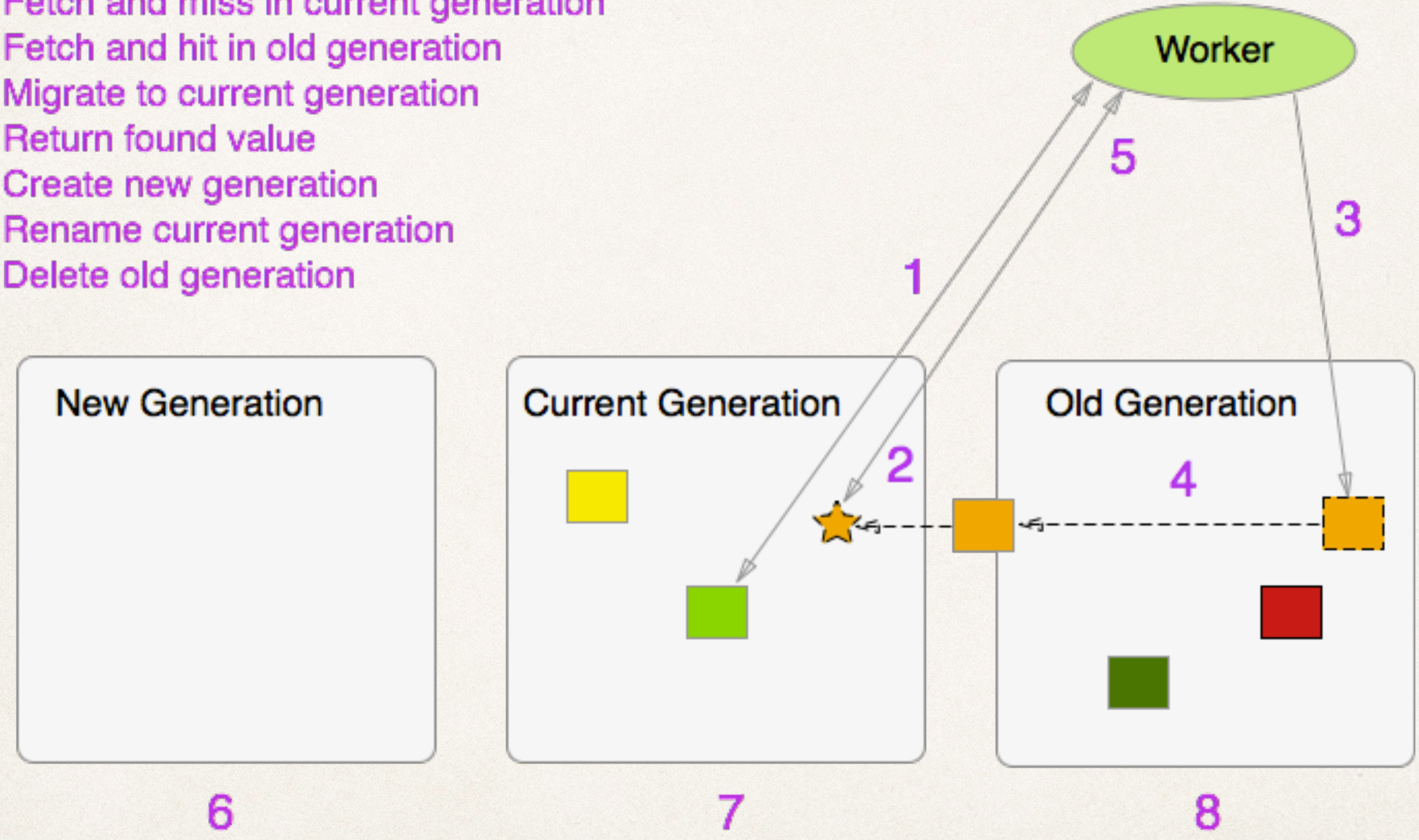
© Matteo Ianeselli / Wikimedia Commons / CC-BY-3.0

Generational Caching

- Avoid setting timers for each cached object
- Use two generations of cache
  - Return hit from newest generation
  - Miss? Then search older generation
    - Hit causes migration of datum to new generation
    - Returns matching datum
    - Miss causes DB fetch to newest generation
- Expiration is create new, then delete old generation

Generational caching illustrated

1) Fetch and hit in current generation cache
2) Fetch and miss in current generation
3) Fetch and hit in old generation
4) Migrate to current generation
5) Return found value
6) Create new generation
7) Rename current generation
8) Delete old generation



Cxy_cache (ets generations)

- Cxy_cache table with metadata about generations
  - Tuple per cache name with table ids for gens
  - Maintains hit/miss statistics per generation
- One unnamed ets table for each generation
- New generation triggers
  - Periodic time basis (e.g., every 5 minutes)
  - Number of generation accesses threshold
  - User function on name, access count, time

Cxy_cache (cont.)

- Generation checking done by polling
  - Supervised FSM owns the ets tables
  - Defaults to polling every 60 seconds
  - User can override polling frequency
  - Avoids overhead on cache fetch / insert
  - Avoids race conditions on new generation create
  - Option for no new generations
    - User determines that cache always fits in RAM

Future plans

- Synchronization barriers
  - Limiting access to resources (c.f. ferd/dispcount)
  - *Any* (1 of N), *Some* (M of N) and *All* (N of N)
- Higher-level compositions of existing patterns
  - Active task queues (ets_buffer plus cxy_ctl)
    - Dynamic workers concurrently consume tasks
  - Pipeline of active queues to manage staged progress
- Open Source Community pull requests / suggestions

Conclusion

- Ets will help increase concurrency
- Design concurrent elements of architecture
    - Partition the data set
    - Employ reserve / write / publish semantics
    - Use atomic operations to advantage
- Prefer community built libraries
    - Getting concurrency right is difficult
    - Consider tools like Concuerror