CMS Requirements on Multi-threading

Christopher D Jones
Elizabeth Sexton-Kennedy

Present Application

CMS uses one application for all event processing

- Particle generation
- Simulation
- Online High Level Trigger
- Reconstruction
- Analysis

Each event processing algorithm is encapsulated into a 'module'

- Geant4 is wrapped by one particular module

CMS' application controls the processing

- It decides which event to process next
- It decides the order to call each module and passes it the proper event
  - Application calls specific Geant4 functions when it is Geant4's turn to do work

Multithreading

Plan for new multithreaded application

- Will process multiple events simultaneously
- Will run multiple modules processing the same event simultaneously
- This will all be controlled explicitly by the application

All parts need to work within one concurrency model

- Present application is memory resource limited
  - in future may not be able to afford 2GB / CPU core
- Each additional thread requires its own stack
  - default size on SL5 is 10MB/stack
- One concurrency model will allow use of only one thread pool
  - minimizes memory
  - avoids oversubscribing available cores
- CMS has chosen Intel Thread Building Blocks as the concurrency model

Interested in Geant-MT if it can fit with this working model

- Where concurrency is controlled by the experiment's application
  - E.g. Application calls specific Geant methods at proper time from threads controlled by application

TBB Task Model

Work to be done in parallel are encapsulated in a tbb::task object

- Object holds what ever data is needed to do the work
- TBB calls `execute()` when it is the task's turn to run

Must tell TBB how many threads should be used

For each thread, there is a work queue

`task::spawn` adds a task to the queue for the thread that calls `task::spawn`

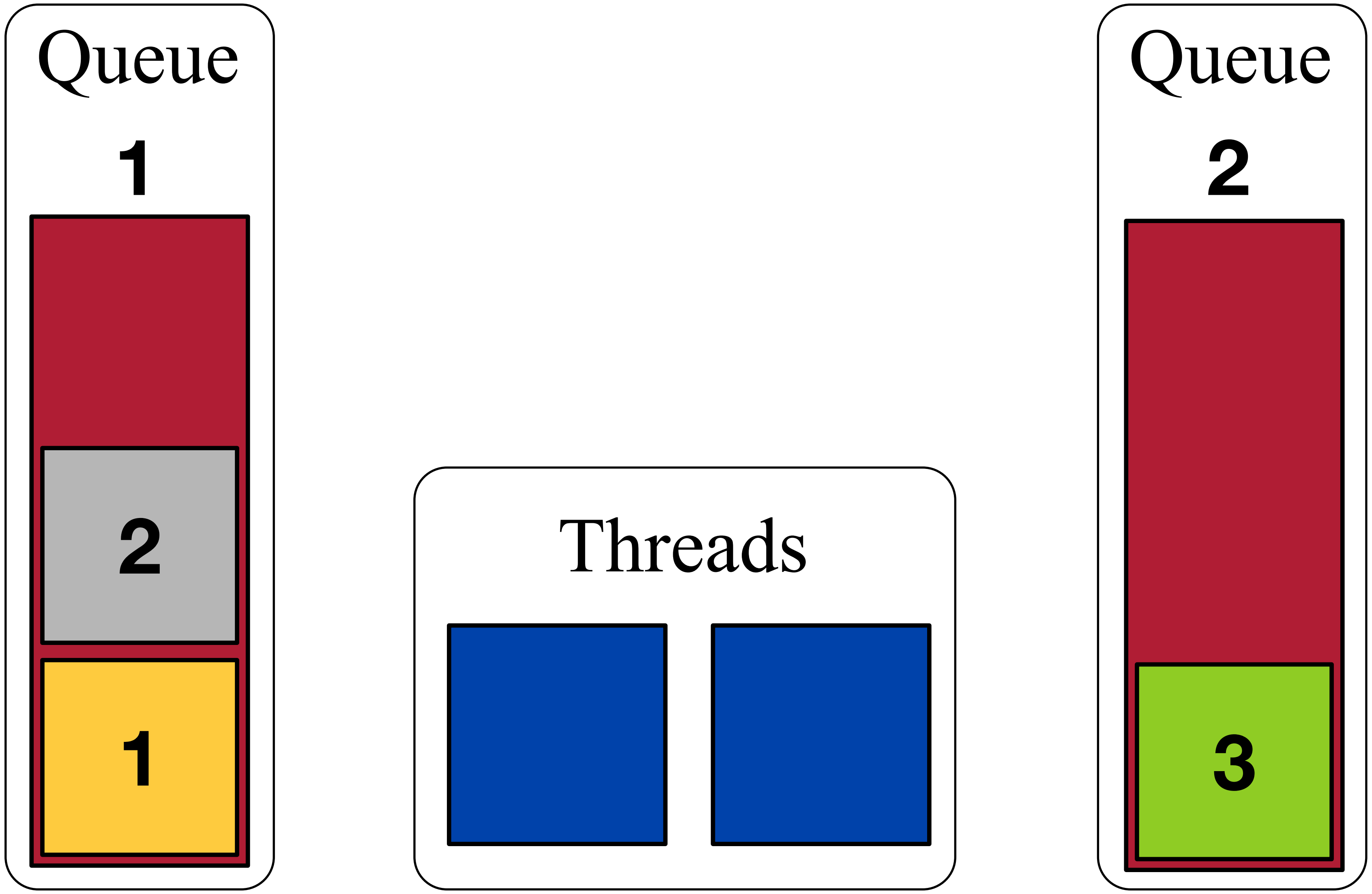
Tasks are pulled from the work queue in Last In First Out order

If a queue is empty, it will

- See if a task is on the shared list and if so take the oldest one, else
- Steal oldest task from another queue

TBB Task Model

Tasks are pulled in Last In First Out order



Queue 1

2

1

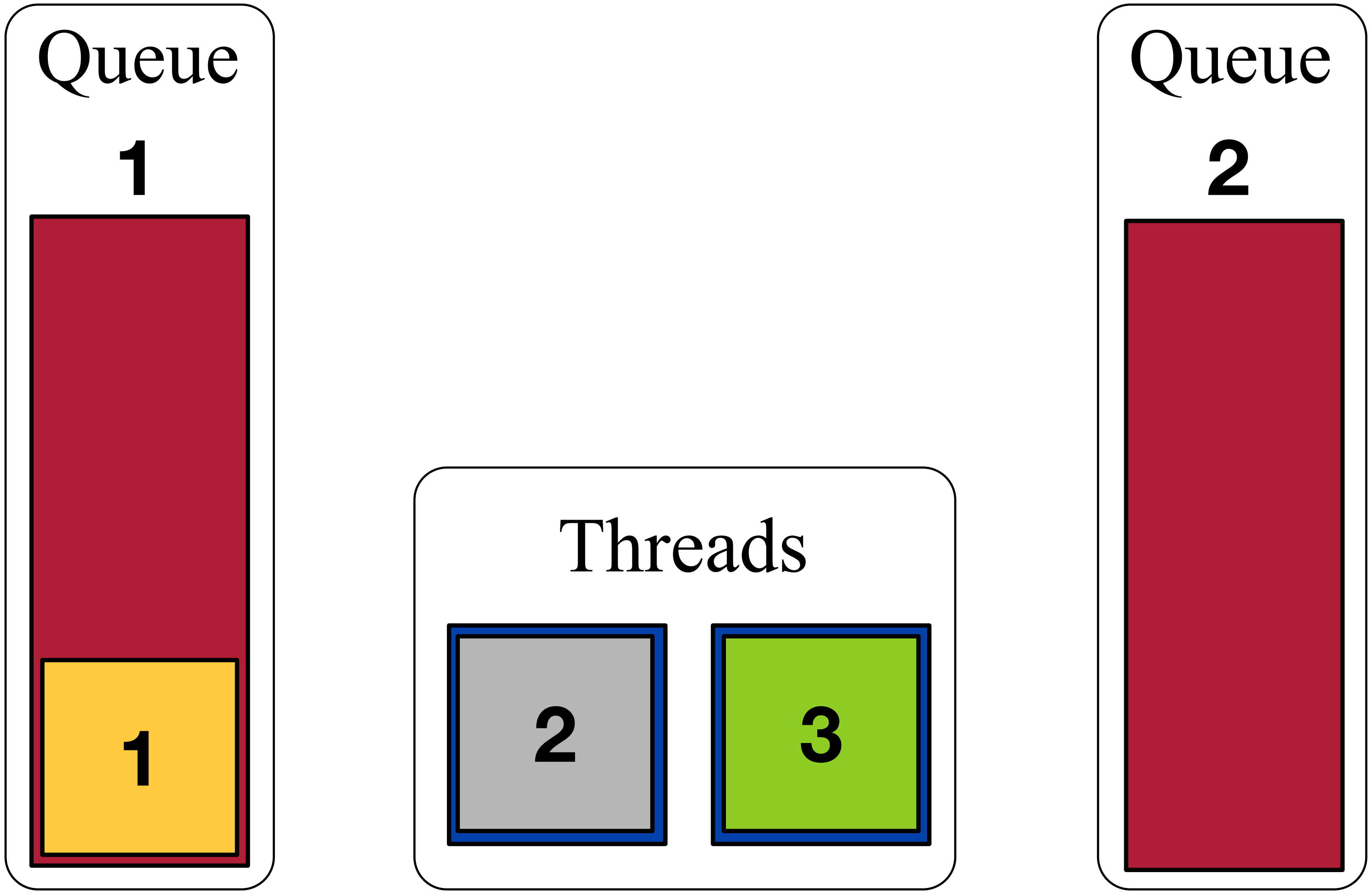
Threads

Queue 2

3

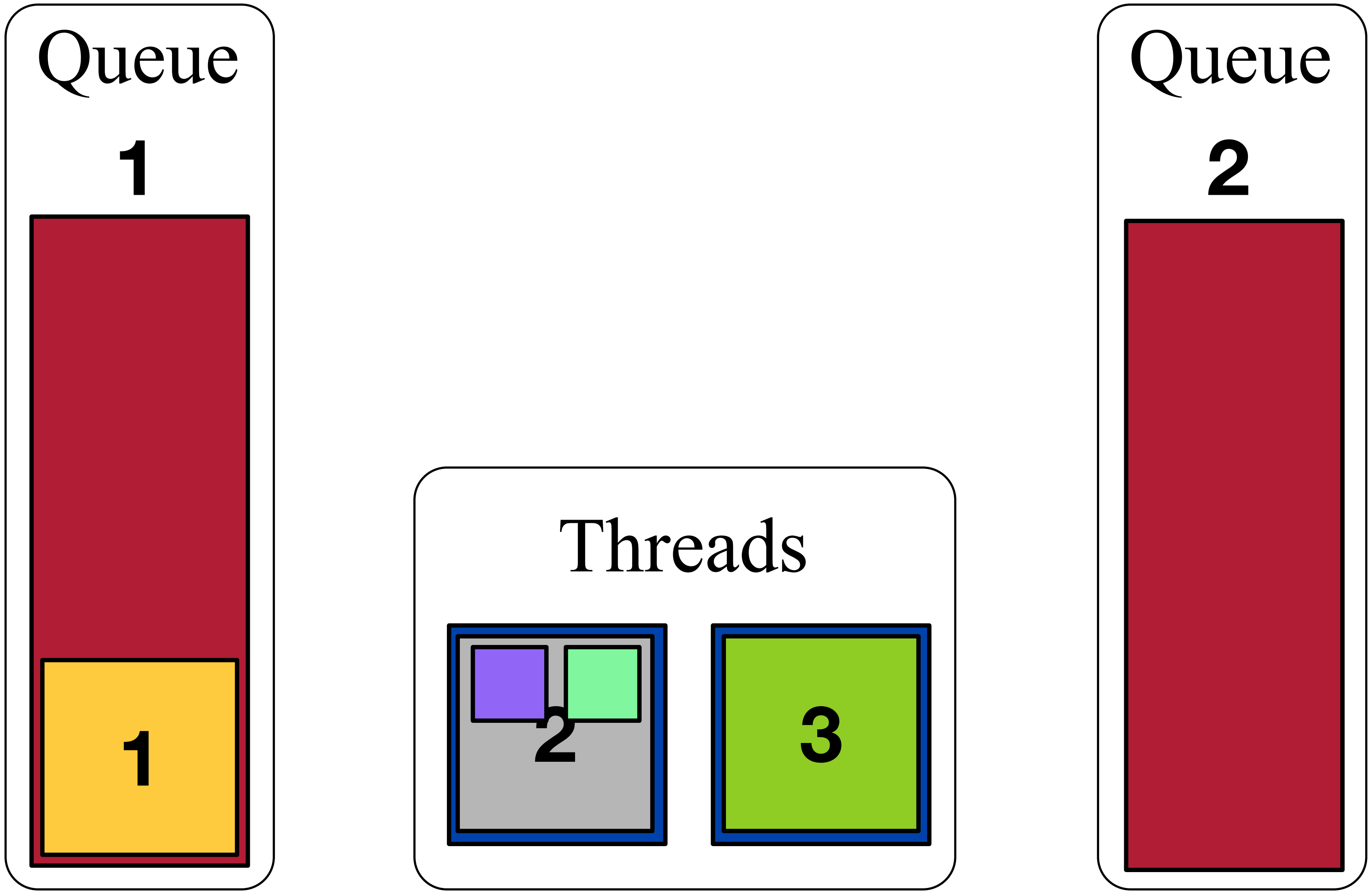
TBB Task Model

Tasks are pulled in Last In First Out order



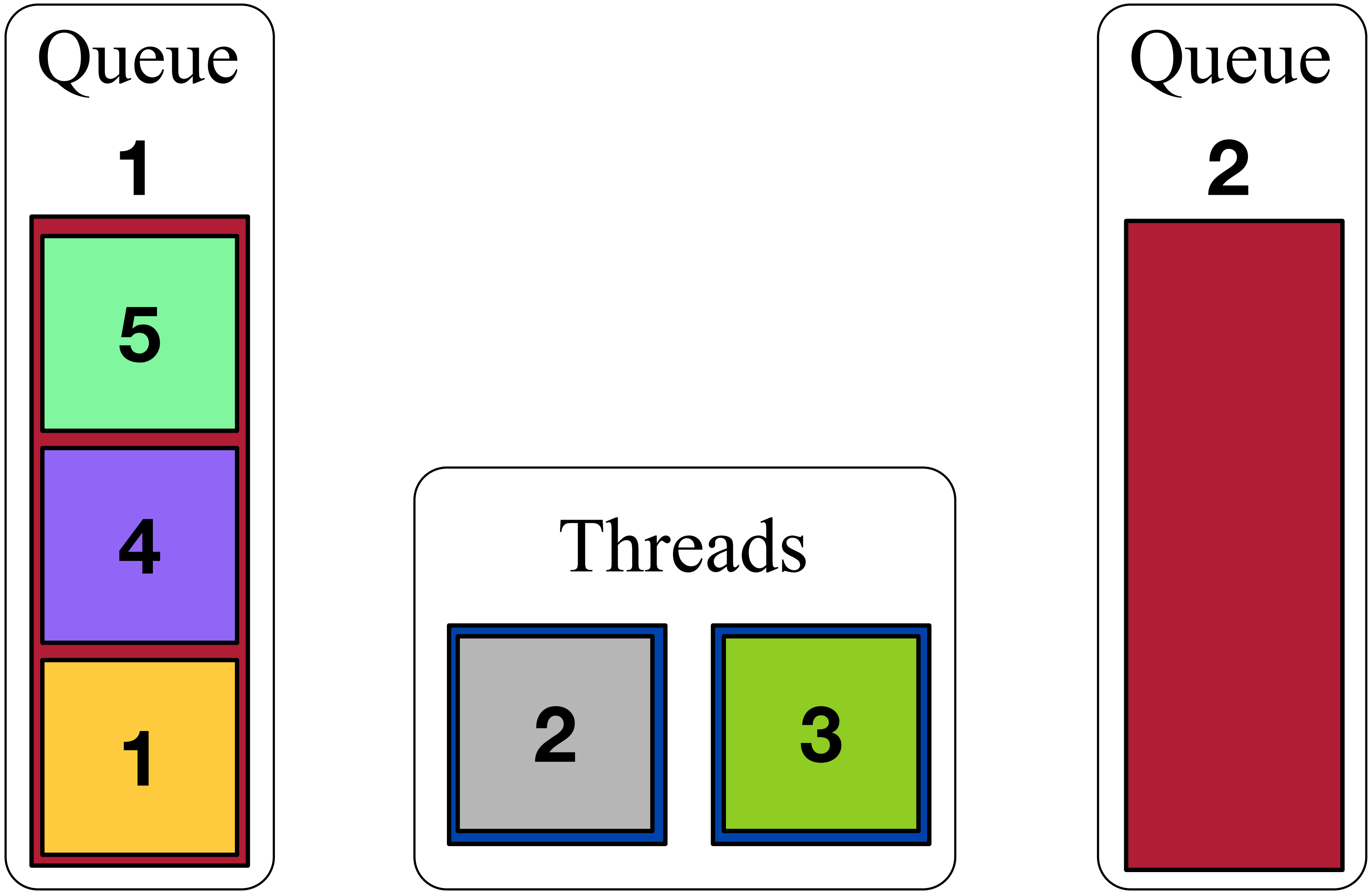
TBB Task Model

Spawned tasks go into the same thread queue as creating task



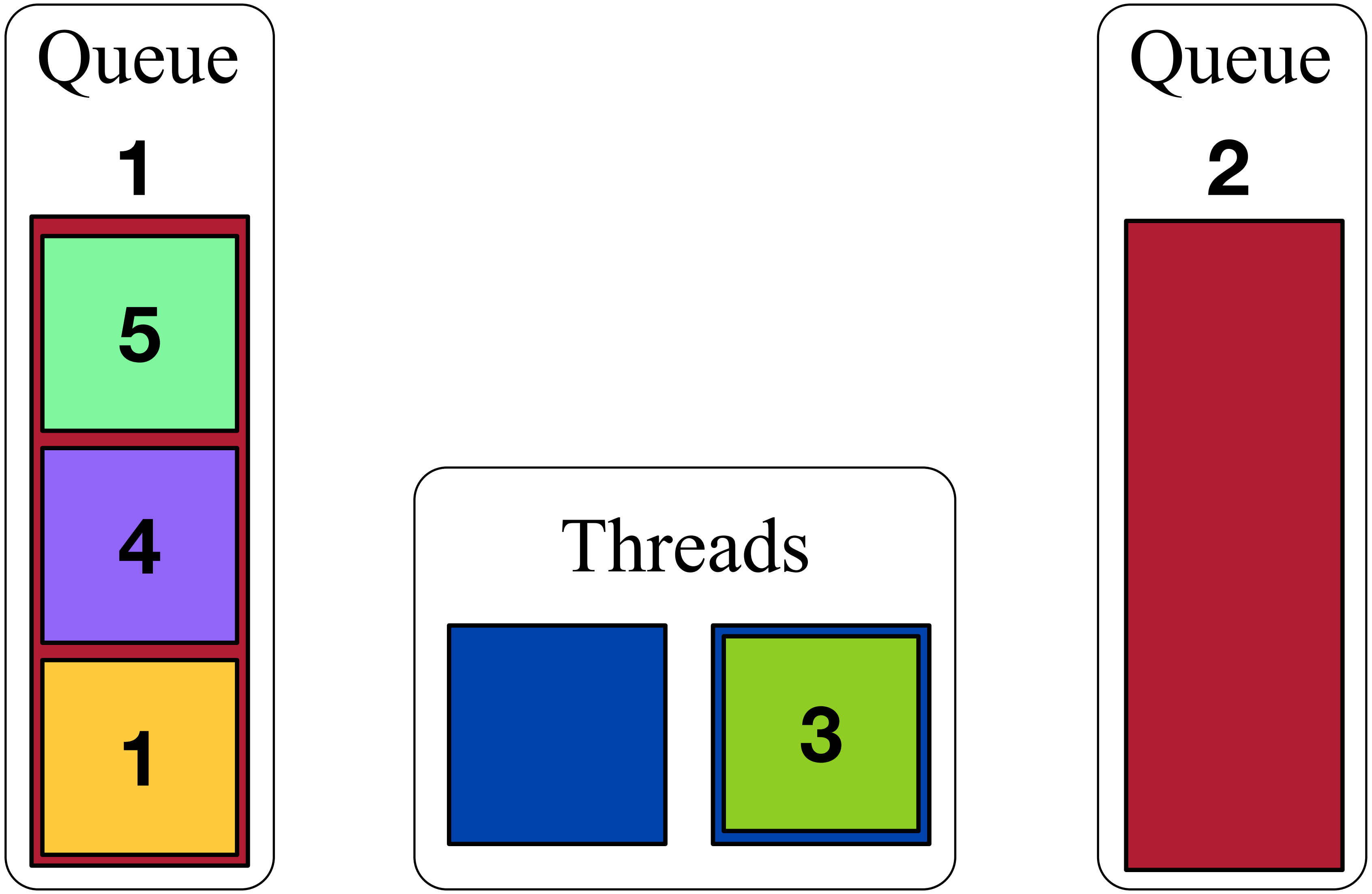
TBB Task Model

Spawned tasks go into the same thread queue as creating task



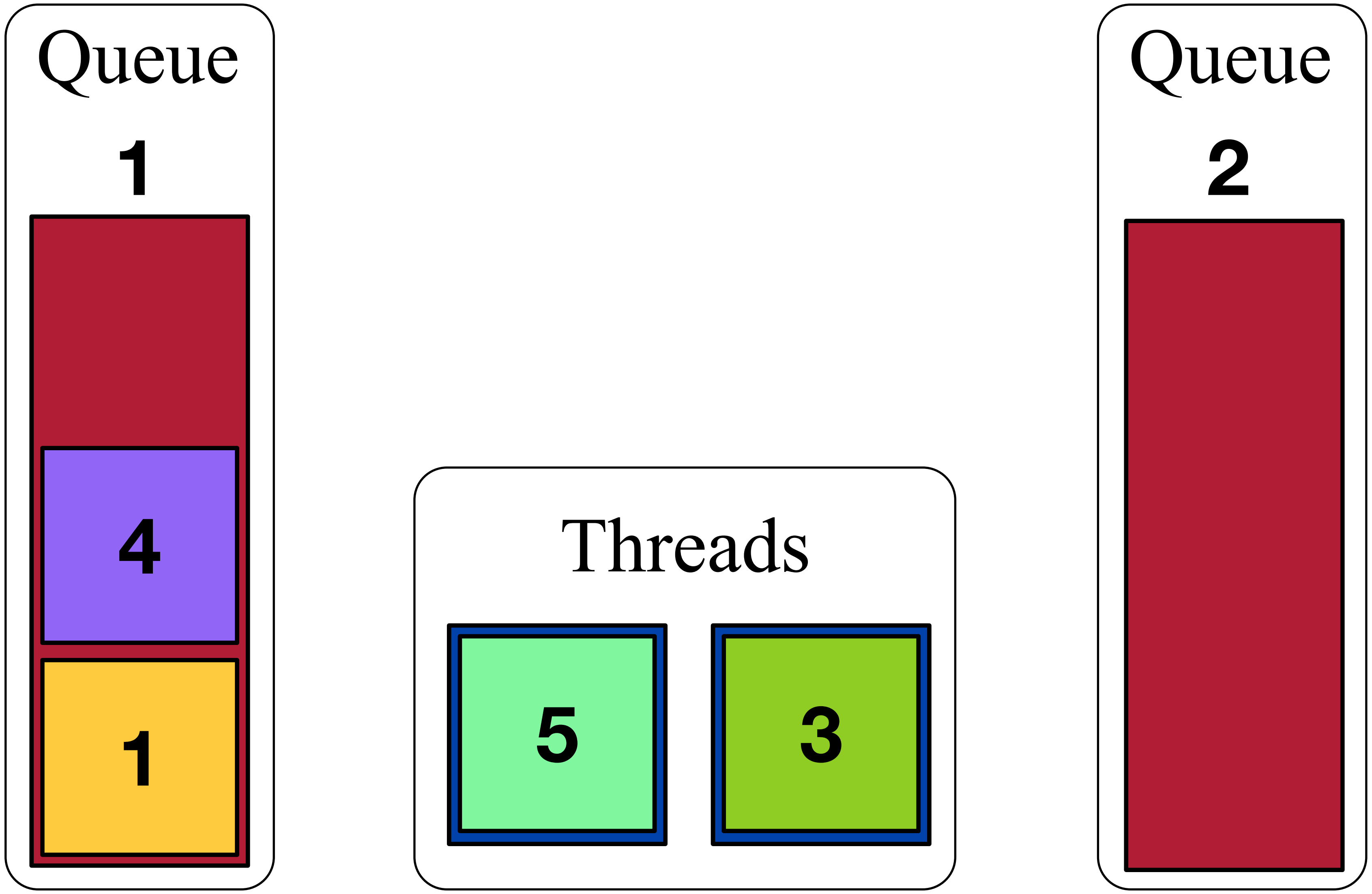
TBB Task Model

Spawned tasks go into the same thread queue as creating task



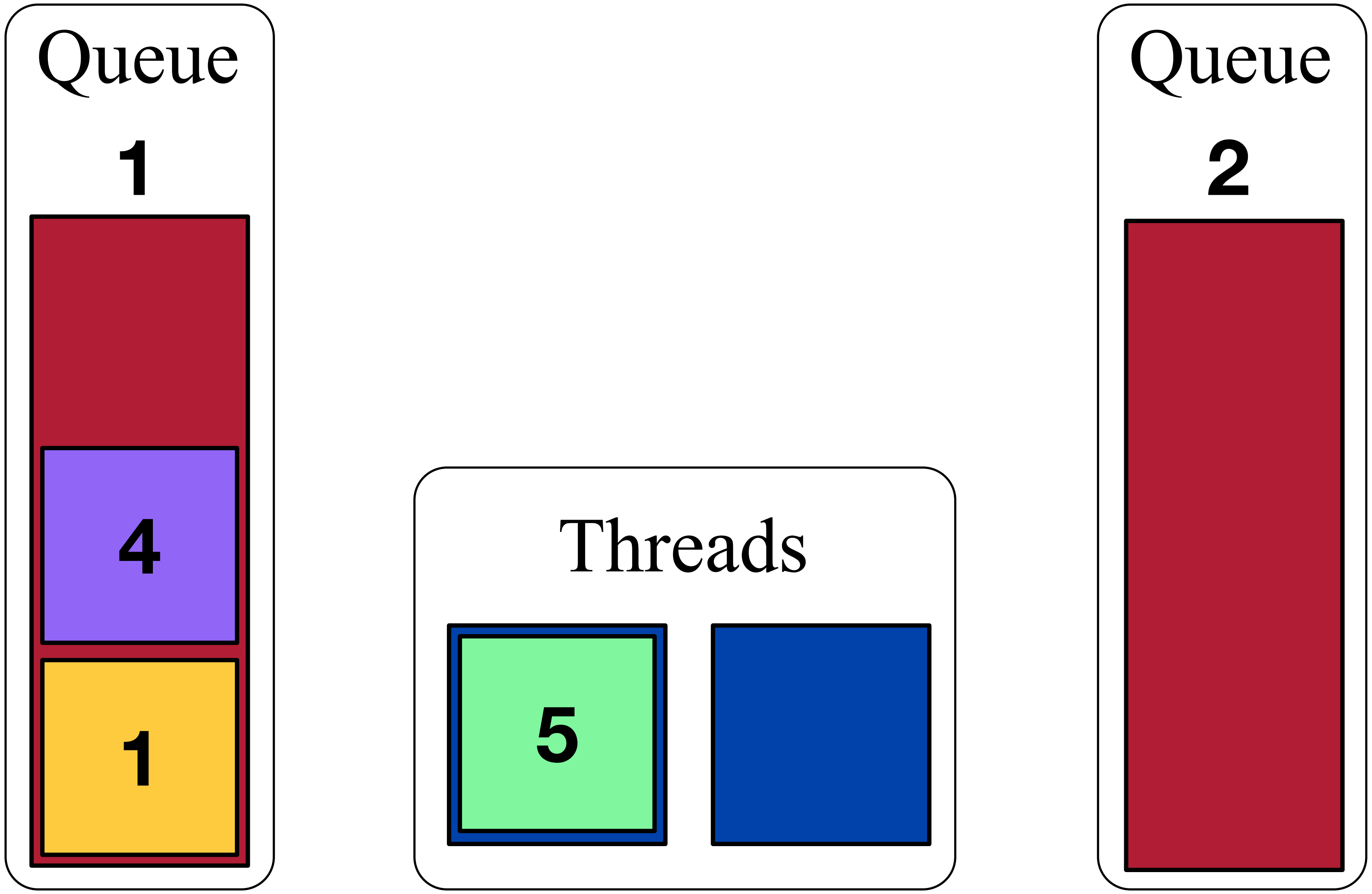
TBB Task Model

Spawned tasks go into the same thread queue as creating task



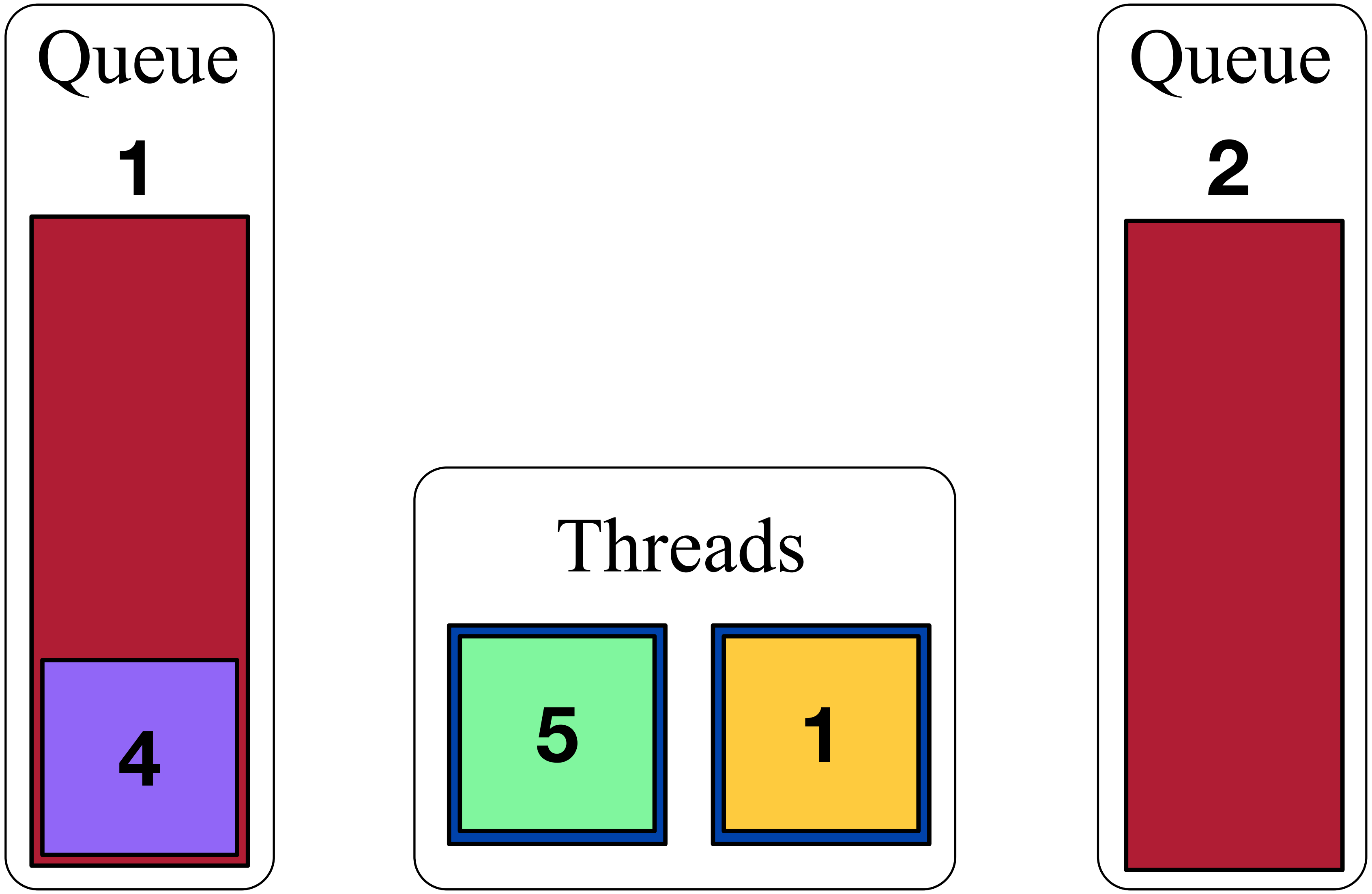
TBB Task Model

An empty thread queue steals oldest task from another queue



TBB Task Model

An empty thread queue steals oldest task from another queue



CMS' Use of TBB

- Application will be told to process N events simultaneously
- Application will be told to use M threads
  - N events <= M threads
- In beginning modules will configure themselves
  - This will be single threaded
  - Modules can setup data structures to be used by all simultaneous events
    - Must be concurrent access safe
    - e.g. a physics list or geometry
  - Modules can setup data N data structures, one per simultaneous event
    - e.g. data structures that temporarily cache per event info such as track lists
- When process an event, module will be called from a TBB task
  - Module has access to
    - the event
    - the module's shared by all simultaneous events data structure
    - the module's data structure for that event (i.e. 1 of the N structures)
  - Module is also allowed to create its own TBB tasks
    - Module must wait for all of its TBB tasks to complete before returning

Conclusion

CMS is pursuing a threaded process framework

Using Intel's TBB as the concurrency model

Model is task based not explicitly thread based

Will need future Geant4 to be amenable to TBB