# Parallel Design Patterns-L12

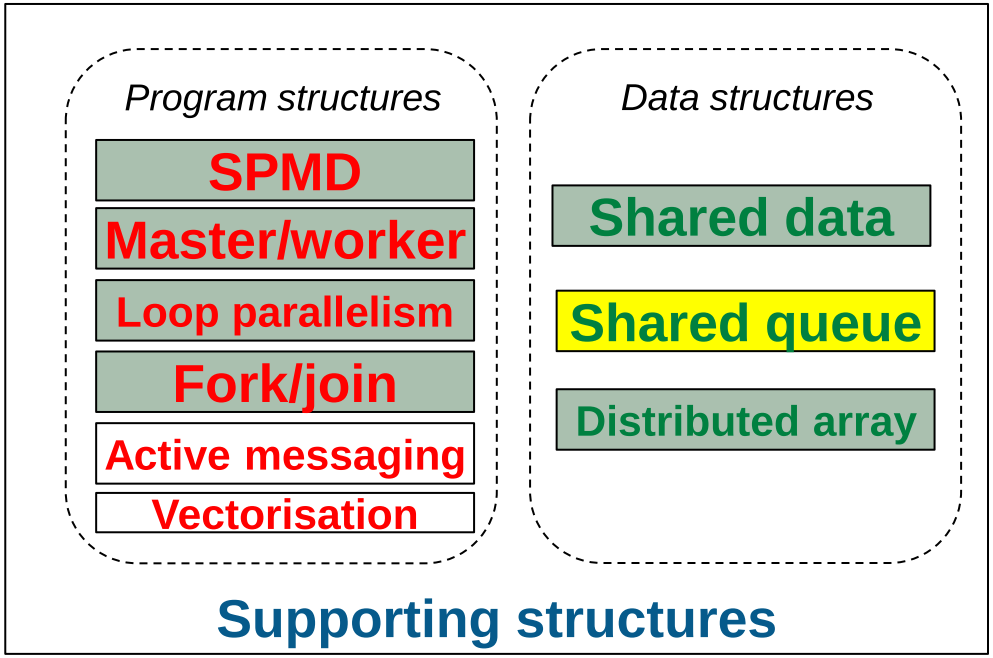
## Shared Queue

Course Organiser: Dr Nick Brown
nick.brown@ed.ac.uk
Bayes Room 2.13

# Shared Queue: The Problem

- Shared Queue is an *Implementation Pattern*
  - *Based on a data type*

- *The Problem:* How can concurrently-executing UEs safely share a queue data structure?



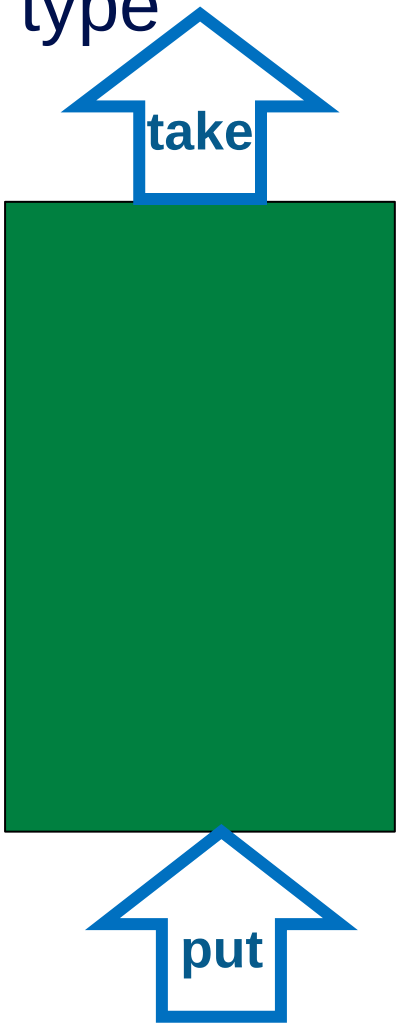
# Shared Queue: Context

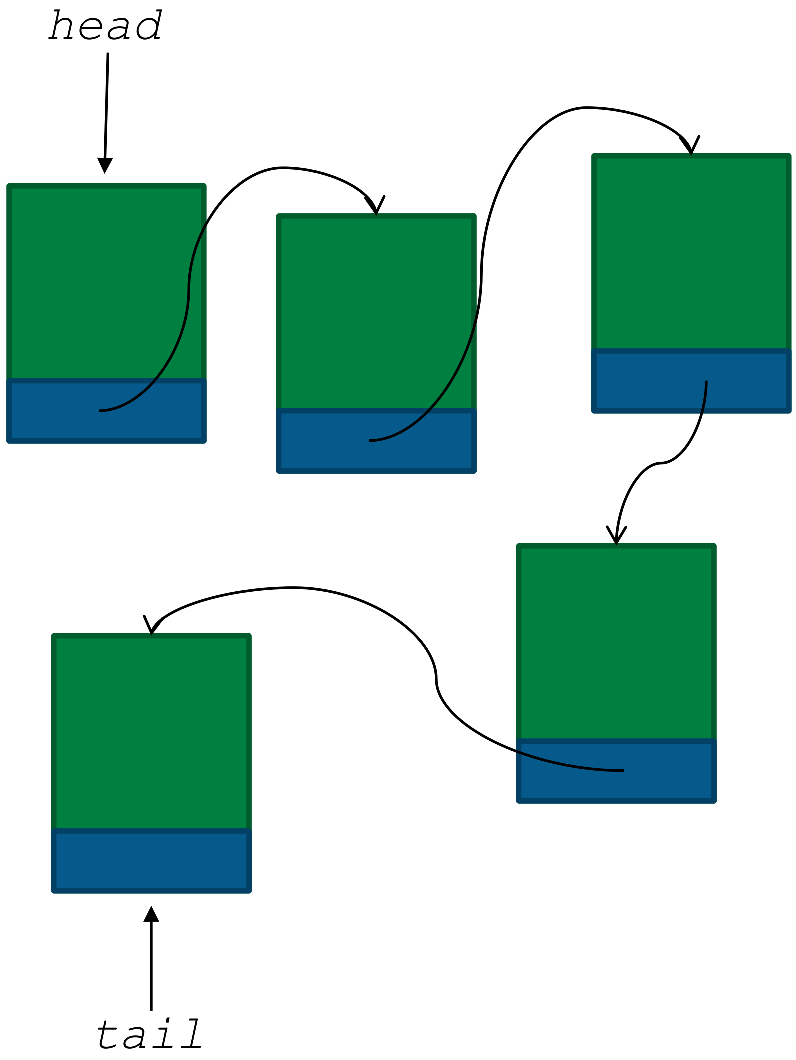
- Effective implementation of many parallel algorithms requires a queue that is to be shared among UEs

- An example we've already talked about is the "bag of tasks" in the Master-Worker pattern

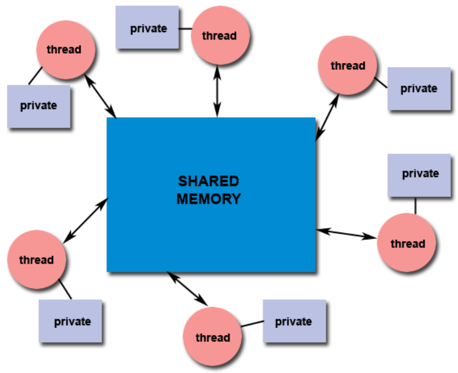
# Solution

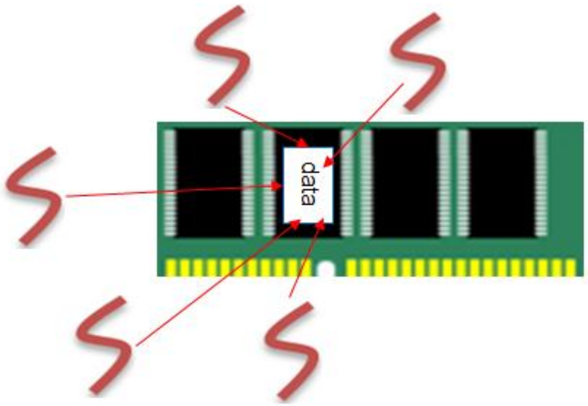
- The queue is a FIFO data type



# Effect of Concurrency-Control Protocol




- Most of the important forces relate to the choice of **concurrency-control protocol**:
  - One-at-a-time execution
  - Non-interfering sets of operations
  - Readers/Writers
  - Splitting or Shrinking the Critical Section
  - Nested Locks
  - Application specific semantic relaxation

*Simple but slow*

*Complex but fast*

# Shared Queue: Forces

- Simple concurrency-control protocols provide greater clarity of abstraction and make it easier for the programmer to verify that the shared queue has been correctly implemented
  - *Aim for clarity first, then optimise*

- Concurrency-control protocols that encompass too much of the shared queue in a single synchronisation construct increase the chances UEs will remain blocked waiting to access the queue and will limit concurrency

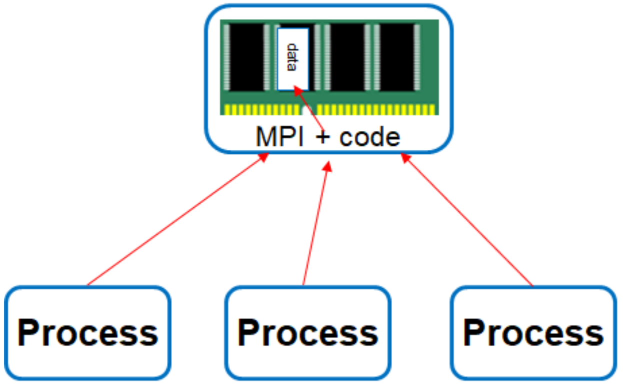
- A concurrency-control protocol finely tuned to the queue and how it will be used increases the available concurrency, at the cost of more complicated, more error-prone synchronisation constructs

# Solution

- Ideally the shared queue would be implemented as part of the target programming language
  - e.g. Java has an implementation available in java.util.concurrent

- No provided mechanism in common HPC languages (MPI, OpenMP)
- Most common use of shared queue is with *shared memory*

- Can be implemented in *message passing* by having the queue owned by one process, and putting and taking from the queue implemented by sending messages to and from the owner process

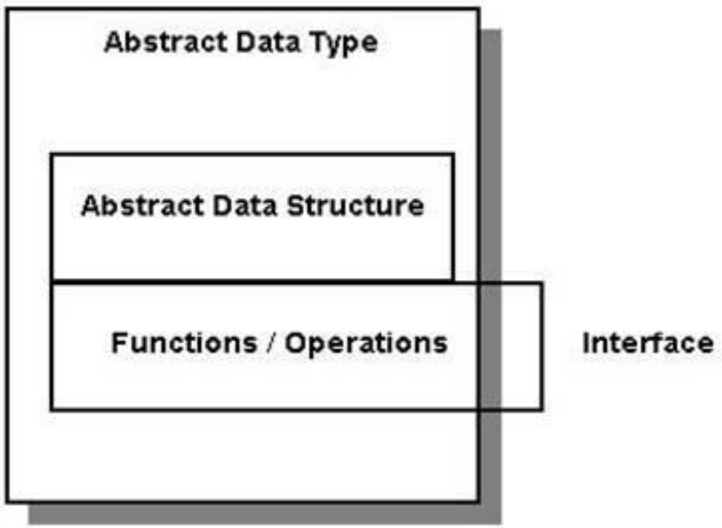


# Solution

*Apply the shared data pattern*

- Define the ADT

- Choose the concurrency protocol



# Defining the ADT

- The operations:
  - Put *(enqueue)*
  - Take *(dequeue)*
  - Other operations are possible, e.g. peek, takeall, clear, isEmpty

- Details:
  - What do you do when a queue is empty?
    - Block and wait for something to arrive
      - Could be used in Master-Worker with poison pill approach
    - Non-blocking queue: Return null or special value

# Concurrency control protocol

- Implementing a shared queue can be tricky
  - but well-written, it can be re-used widely

- Choice of protocols
  - One-at-a-time execution
  - Non-interfering sets of operations
  - Readers/Writers
  - Splitting or Shrinking the Critical Section
  - Nested Locks
  - Application specific semantic relaxation

# One at a time: Non-blocking

```
public class SharedQueue1 {
  class Node { //inner class defines list nodes {
    Object task;
    Node next;
    Node(Object task) {this.task = task; next = null;}
  }
  private Node head = new Node(null); //dummy node
  private Node last = head;

  public synchronized void put(Object task) {
    assert task != null: "Cannot insert null task";
    Node p = new Node(task);
    last.next = p;
    last = p;
  }

  public synchronized Object take() {
    //returns first task in queue or null if queue is empty
    Object task = null;
    if (!isEmpty()) {
      Node first = head.next;
      task = first.task;
      first.task = null;
      head = first;
    }
    return task;
  }
  private boolean isEmpty(){return head.next == null;} }
```

# OpenMP version

- A simple queue of ints, for illustration purposes:

```
void put (int i){
#pragma omp critical
  …
#pragma omp end critical
}

int take(){
#pragma omp critical
  …
#pragma omp end critical
}
```

# One at a time: Block on queue empty

```
public class SharedQueue2 {
  class Node {
    Object task;
    Node next;
    Node(Object task) {this.task = task; next = null;}
  }
  private Node head = new Node(null);
  private Node last = head;

  public synchronized void put(Object task) {
    assert task != null: "Cannot insert null task";
    Node p = new Node(task);
    last.next = p;
    last = p;
    notifyAll();
  }

  public synchronized Object take() {
    //returns first task in queue, waits if queue is empty
    Object task = null;
    while (isEmpty()) {
      try{wait();}catch(InterruptedException ignore){}
    }
    Node first = head.next;
    task = first.task;
    first.task = null;
    head = first;
    return task; } }
```

- Wait will release lock
  - Waits until notified
- notifyAll wakes all threads
  - In tern as lock on take method
- Pthreads has condition variables
  - Wait and signal

```
public class SharedQueue1 {
  class Node { //inner class defines list nodes {
    Object task;
    Node next;
    Node(Object task) {this.task = task; next = null;}
  }
  private Node head = new Node(null); //dummy node
  private Node last = head;

  public synchronized void put(Object task) {
    assert task != null: "Cannot insert null task";
    Node p = new Node(task);
    last.next = p;
    last = p;
  }

  public synchronized Object take() {
    //returns first task in queue or null if queue is empty
    Object task = null;
    if (!isEmpty()) {
      Node first = head.next;
      task = first.task;
      first.task = null;
      head = first;
    }
    return task;
  }
  private boolean isEmpty(){return head.next == null;} }
```

# Non-interfering operations

```
public class SharedQueue3 {
  class Node {
    Object task;
    Node next;
    Node(Object task) {this.task = task; next = null;}
  }

  private Node head = new Node(null);
  private Node last = head;

  private Object putLock = new Object();
  private Object takeLock = new Object();

  public void put(Object task) {
    synchronized(putLock) {
      assert task != null: "Cannot insert null task";
      Node p = new Node(task);
      last.next = p; last = p;
    }
  }

  public Object take() {
    Object task = null;
    synchronized(takeLock) {
      if (!isEmpty()) {
        Node first = head.next;
        task = first.task;
        first.task = null;
        head = first;
      }
    }
    return task; } }
```

- Put and take are independent as do not access the same variables
- Therefore use different locks
- Only works for non blocking
- Could be two different mutexes in pthreads

# OpenMP version

- A simple queue of ints, for illustration purposes:

```
void put (int i){
#pragma omp critical(put)
  …
#pragma omp end critical(put)
}

int take(){
#pragma omp critical (take)
  …
#pragma omp end critical (take)
}
```

# Nested locks

```
pubic class SharedQueue4 {
  class Node {
    Object task; Node next;
    Node(Object task) {
      this.task = task; next = null;}
  }
  private Node head = new Node(null);
  private Node last = head;
  private int w;
  private Object putLock = new Object();
  private Object takeLock = new Object();

  public void put(Object task) {
    synchronized(putLock) {
      assert task != null: "Cannot insert null task";
      Node p = new Node(task);
      last.next = p; last = p;
      if(w>0) putLock.notify();
    }
  }
  public Object take() {
    Object task = null;
    synchronized(takeLock) {
      //returns first task in queue, waits if queue is empty
      while (isEmpty()) {
        try { synchronized(putLock){ w++; putLock.wait();w--; }
        } catch(InterruptedException error){assert false;}
      }
      Node first = head.next;
      task = first.task;
      first.task = null; head = first;
    }
    return task; } }
```

- Blocking on empty
- Waits on the putLock lock
- Need to be very careful to avoid deadlock

# Readers and writers

```
private Node last = head;

  Rwlock rw_lock=new Rwlock();

  public void put(Object task) {
    rw_lock.writeLock();
    assert task != null: "Cannot insert null task";
    Node p = new Node(task);
    last.next = p; last = p;
    rw_lock.release();
  }

  public Object viewlast() {
    Object task = null;
    rw_lock.readLock();
    if (!isEmpty()) {
      task=last.task;
    }
    rw_lock.release();
    return task; } }
```

- Here *last* is used in both the functions
  - But one writes whilst the other reads
  - The reader can operate concurrently
  - Only one writer exclusively
- An example of this is rwlocks in pthreads

# Shrinking the critical section

```
private Node last = head;

  Rwlock rw_lock=new Rwlock();

  public void put(Object task) {
    assert task != null: "Cannot insert null task";
    Node p = new Node(task);
    rw_lock.writeLock();
    last.next = p; last = p;
    rw_lock.release();
  }

  public Object viewlast() {
    Object task = null;
    rw_lock.readLock();
    if (!isEmpty()) {
      task=last.task;
    }
    rw_lock.release();
    return task; } }
```

# Distributed shared queues

- One central queue can be a bottleneck
  - Can we split this up so there is a queue per UE and distribute the contents?
- If my local queue becomes empty then a *take* might “steal” an element from a neighbour’s queue
- If my local queue becomes full then a *put* might add the element to a neighbour’s queue
- E.g. Allocating tasks to each UE to execute, queue these up and then allow for work stealing once completed.

# Related Patterns

- Shared Data:
  - Shared Queue pattern is an instance of Shared Data pattern.

- Master/Worker:
  - Shared Queue pattern is often used to represent the task queues in algorithms that use the Master/Worker pattern.
- Fork/Join pattern:
  - Thread-pool-based implementation of Fork/Join pattern is supported by this pattern such as the tasks of OpenMP we saw in the practical

# Summary

- Idea: A shared queue encapsulates the synchronisation required inside an abstract data type
- Examples have been OO, but you can “encapsulate” inside put and take routines
- Different implementations can vary in performance and complexity
- Shared queue is a key component of various other parallel patterns