Java #3

GC & Concurrent

Garbage Collection

Garbage Collector

- Memory Management Mechanism
- Used in most VM / interpreted environment
- Opposed to manual allocation

Garbage Collector

Pros:

- Extremely efficient
- Collection is free
- Death detection is precise

Cons:

- Stop 1s/Gb
- Does not prevent all memory leaks
- Pauses can be delayed, not stopped

Garbage Collector

Terminology

- `Monolitic` GC: operates in a single, indivisible step.
- `Parallel` GC: uses multiple threads to operate.
- `Concurrent` GC: operates while the application operates.
- `Stop-The-World`: pauses completely the application to collect.
- `Incremental`: operates in a sequence of divisible steps.
- Mostly: IS NOT.

Garbage Collector

Terminology

- `Conservative` : if the GC is unaware or unsure about some references while collecting.
- `Precise` : 100% accuracy.
- `GC safepoint` : point where the application can safely stop to let the GC operate.
- `Global Safepoint` : all threads are un a safe point -> Stop-The-World.

Garbage Collector

Phases

- Mark
- Sweep
- Compact
- Copy

All GC algorithms are combinations of these phases

Garbage Collector

Mark

- Start from roots :
  - Thread stacks,
  - Local Variables,
  - Statics;
- Mark anything reachable as “live".
- In the end, any non-marked object is dead.
- Complexity linear to “live” objects set, not heap size.

Garbage Collector

Sweep

- Scan through the heap.
- Sweep dead objects to a free list.
- Complexity linear to the heap size.

Garbage Collector

Compact

- Over time, fragmentation happens;
- Move live references to compact them (relocate),
- And reclaim consecutive free space.
- The hard part is updating references pointing to moved objects (remap)
- Complexity linear to live set.

Garbage Collector

Copy

- Copy all live objects from one space to another.
- Starting space is then completely free.
- Complexity linear to live set.
- Oftentimes, combined with Mark phase.

Garbage Collector

Generational Hypothesis

Most objects die young.

Garbage Collector

Generational Hypothesis

- Divide objects on generation:
  - Young (can be separated further),
  - Old.
- Promote objects that live long enough to an older gen.

Garbage Collector

Generational Hypothesis

- Collect young{er} generations with Mark/Copy, Monolithic STW.
- Collect old{er} generations as they fill up.
- Use Mark/Sweep for oldest gen.

Garbage Collector

Take away

- Young generation disposal is dirt cheap,
- Old generation is stable, but disposal is expensive.
- Design accordingly.

Concurrent Programming

Process:

- Self contained execution environment
- Complete, private set of resources, including memory space
- Communication between process through IPCs:
    - Pipes
    - Sockets
    - Shared Memory

Threads:

- Lightweight process
- Exist within a process
- Every process has at least one thread
- Process resources are shared between threads

Threads:

- Manipulating threads is hard:
  - Synchronization
  - Data access

Alternatives:

- Actors
- Event Loops
- Executors
- Parallel Streams
- Fork/Join

Synchronizing data accees

- Concurrent data accesses are unsafe
- Most operations are not atomic
- even a simple ++i may cause issue
- Some problems are more complex

Synchronizing data accees

Lock/Mutex

- Monitor:
  - Object protected with a lock (mutex)
  - Invoking methods locks it
- All java object implements a monitor
- `synchronized` method are monitor methods
- Sufficient for all basic situations
- Locking is reentrant:
  - You can re-acquire a lock if you hold it already
  - Call to other synchronised methods are safe

Synchronizing data accees

Lock/Mutex

```
public class SimpleSyncCounter {

    private int counter;

    public SimpleSyncCounter() { counter = 0; }

    public synchronized void increment() {counter += 1;}
    public synchronized void decrement() {counter -= 1;}
    public synchronized int get() {return counter;}
}
```

Synchronizing data accees

Lock/Mutex

```
public class SimpleAtomicCounter {

    private AtomicInteger counter = new AtomicInteger(0);

    public int get() { return counter.get(); }

    public void increment() { counter.getAndIncrement(); }
    public void decrement() { counter.getAndDecrement(); }
}
```

Synchronizing data accees

Atomic Types

- Useful basic types in atomic version:
    - int, boolean, int[], reference …
    - Note that double and long access is not atomic.
- More efficient than locked methods
- Offers usual atomic operations
- Needed to implement efficient concurrent data structures (non-blocking algorithms).

Synchronizing data accees

Atomic Types

- All members marked as volatile can be accessed atomically.
- Any change to a volatile property creates a “happened-before” relationship.
  - IE: The vm “commits" the change to make sure it is immediately visible to all threads.

Synchronizing data accees

Synchronized Blocks

```java
public class IncCounter {

    private Integer c1 = 42;

    public void add(final Integer toAdd) {

        // More control over when synchronization happens.
        if (toAdd > 0) {
            synchronized (c1) {
                c1 += toAdd;
            }
        }
    }
}
```

Synchronizing data accees

Synchronized Blocks

```java
// Finer access control allows better liveness.
public class DualQueue<T> {

    private final Queue<T> firstQueue = new ArrayDeque<>();
    private final Queue<T> secondQueue = new ArrayDeque<>();

    public void pushFirst(final T element) {
        // Other threads can still access to secondQueue
        synchronized (firstQueue) {
            firstQueue.add(element);
        }
    }

    public void pushSecond(final T element) {
         synchronized (secondQueue) {
            // Other threads can still access to firstQueue
            secondQueue.add(element);
        }
    }
}
```

Synchronizing data accees

Synchronized Blocks

- `Object#wait`
  - Current thread wait, with the object as the monitor.
- `Object#notify`
  - Wakes one of the threads waiting on the monitor at random.
- `Object#notifyAll`
  - Wakes all the threads waiting on the monitor.

Blocking Queues

- Efficient scalable thread-safe non-blocking FIFO queue
- Async put/take if not full, can wait or throw otherwise
- Excellent for communication between threads

Blocking Queues

```
class BlockingQueueExample {
    public static void main(String[] args) throws Exception {

        BlockingQueue queue = new ArrayBlockingQueue(1024);

        Producer producer = new Producer(queue);Consumer consumer = new Consumer(queue);

        new Thread(producer).start();
        new Thread(consumer).start();

        Thread.sleep(4000);
    }
}
```

CompletableFuture

- Interface
- Most modern tool for concurrent development in the JDK
- Defines the contract for an asynchronous computation step that can be combined with other steps.
- Execution can be delegated to a scheduler thread pool
- Manage exceptions gracefully

CompletableFuture

```
class TotallyNotUsefulDuringTheRush {

    public static String doSomethingWithAFixedDelay() {…}

    public static void main(String[] args) throws Exception {
        CompletableFuture
            .supplyAsync(() -> doSomethingWithAFixedDelay())
            .get();
    }
}
```

CompletableFuture

```java
import java.util.concurrent.CompletableFuture;

class TotallyNotUsefulDuringTheRush {
    public static String doSomethingWithAFixedDelay() {…}

    public static String fetchReport(final String reportId) {…}

    public static void main(String[] args) throws Exception {
        CompletableFuture.supplyAsync(() -> doSomethingWithAFixedDelay())
                .thenApplyAsync(reportId -> fetchReport(reportId))
                .get();
    }
}
```

CompletableFuture

```
class TotallyNotUsefulDuringTheRush {

    public static String doSomethingWithAFixedDelay() {…}

    public static String fetchReport(final String reportId) {…}

    public static void main(String[] args) throws Exception {

        final long delay = 10L;

        final Executor executor =
                  CompletableFuture.delayedExecutor(delay, TimeUnit.MILLISECONDS);

        CompletableFuture
            .supplyAsync(() -> doSomethingWithAFixedDelay())
            .thenApplyAsync(reportId -> fetchReport(reportId), executor)
            .get();
    }
}
```

CompletableFuture

Recovering from errors

```
class TotallyNotUsefulDuringTheRush {

    public static String doSomethingThatMayFail() {…}


    public static void main(String[] args) throws Exception {

        final long delay = 10L;

        final Executor executor =
                 CompletableFuture.delayedExecutor(delay, TimeUnit.MILLISECONDS);

        CompletableFuture
            .supplyAsync(() -> doSomethingThatMayFail())
            .handle((value, exception)) -> return value != null ? value : "ERROR")
            .get();
    }
}
```

CompletableFuture

Exercise

- Write a sequence of completable future that:
  - Produces a random int between 0 and 10 included.
  - Wait one second.
  - Divide the generated value by another random value between -1 and 1.
  - Prints the result.
  - Errors should be recovered by generating the value 42.