Chapter 7

Threading

7.1 Threading

Threads are the smallest program units that an operating system can execute. Programming with threads allows that several lightweight processes can run simultaneously inside the same program. Threads that are in the same process share the memory and the state of the variables of the process. This shared use of resources enables threads to run faster than execute several instances of the same program.

Each process has at least one thread that corresponds to its execution. When a process creates several threads, it executes these units as *parallel processes*. In a single-core machine, the parallelism is approximated through thread *scheduling* or *time slicing*. The approximation consists of assigning a limited amount of time to each thread repeatedly. The alternation between threads simulates parallelism. Although there is no true increase in execution speed, the program becomes much more responsive. For example, several tasks may execute while a program is waiting for a user input. Multi-core machines achieve a truly faster execution of the program. Figure 7.1 shows how threads interact with the main process.

Some examples of where it is useful to implement threads, even on single-core computers, are:

- Interfaces that interact with the user while the machine executes a heavyweight calculation process.
- Delegation of tasks that follow consumer-producer pattern, *i.e.*, jobs which outputs and inputs are related, but run independently.
- Multi-users applications, in which each thread would be in charge of the requests of each user.

Python 3 handles threads by using the *threading* library. It includes several methods and objects to manipulate threads.

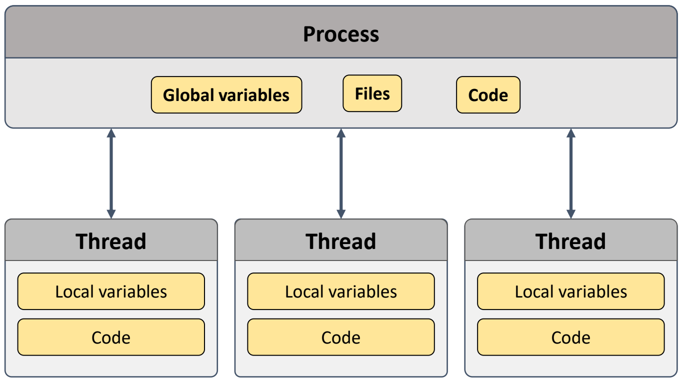

Figure 7.1: Diagram of a threading-based application

.

Creating Threads

We can create a new thread using the `Thread` class from the `Threading` library. This class requires three arguments: `target` to define the function to be executed; `name` to provide name we want to give to the thread; `args` to pass the `target` arguments. Once created, it may be executed by calling the `start()` method. In the next example, we create three threads `t1`, `w1`, and `w2`, that execute different instances of the `service` and `worker` functions.

```python
# code0.py

import threading
import time


def worker():
    print("{} starting...".format(threading.currentThread().getName()))
    # This stops the thread execution for 2 seconds.
    time.sleep(2)
    print("{} exiting...".format(threading.currentThread().getName()))


def service():
    print("{} starting...".format(threading.currentThread().getName()))
    # This stops the thread execution for 4 seconds.
    time.sleep(4)
    print("{} exiting...".format(threading.currentThread().getName()))
```

```python


# We create two named threads
t1 = threading.Thread(name='Thread 1', target=service)
w1 = threading.Thread(name='Thread 2', target=worker)

# This uses the default name (Thread-i)
w2 = threading.Thread(target=worker)

# All threads are executed
w1.start()
w2.start()
t1.start()


# The following will be printed before the threads finish executing
print('\nThree threads were created\n')
```

```
Thread 2 starting...
Thread-1 starting...
Thread 1 starting...

Three threads were created

Thread 2 exiting...
Thread-1 exiting...
Thread 1 exiting...
```

In the example, we see that once we have initialized the threads, the main program continues with the rest of the instructions while threads execute their task. The three threads end independently at different times. The main program waits until all the threads finish correctly.

The following code shows an example of how to pass arguments to the `target` function through the `args` attribute.

```python
# code1.py

import threading
import time


def worker(t):
    print("{} starting...".format(threading.currentThread().getName()))

    # Thread is stopped for t seconds
    time.sleep(t)
    print("{} exiting...".format(threading.currentThread().getName()))


# Threads are created using the Thread class, these are associated with the
# objective function to be executed by the thread. Function attributes are
# given using the 'args' keyword. In this example, we only need to give one
# argument. For this reason a one value tuple is given.

w = threading.Thread(name='Thread 2', target=worker, args=(3,))
w.start()
```

```
Thread 2 starting...
Thread 2 exiting...
```

Another way of creating a thread is by inheriting from `Thread` and redefining the `run()` method.

```python
# code2.py

import threading
import time


class Worker(threading.Thread):

    def __init__(self, t):
        super().__init__()
        self.t = t

    def run(self):
        print("{} starting...".format(threading.currentThread().getName()))
        time.sleep(self.t)
        print("{} exiting...".format(threading.currentThread().getName()))


class Service(threading.Thread):

    def __init__(self, t):
        super().__init__()
        self.t = t

    def run(self):
        print("{} starting...".format(threading.currentThread().getName()))
        time.sleep(self.t)
        print("{} exiting...".format(threading.currentThread().getName()))


# Creating threads
t1 = Service(5)
w1 = Worker(2)
w2 = Worker(4)

# The created threads are executed
t1.start()
w1.start()
w2.start()
```

```
Thread-1 starting...
Thread-2 starting...
Thread-3 starting...
Thread-2 exiting...
Thread-3 exiting...
Thread-1 exiting...
```

Join()

In certain situations, we would like to synchronize part of our main program with the outputs of the running threads. When we need the main program to wait that the execution of a thread or a group of threads finished, we must use the `join(< maximum-waiting-time >)` method after the thread starts. In this way, every time we use `join()` the main program will be blocked until the referenced threads finish correctly. If we do not define the maximum waiting time, the main program waits indefinitely until the referenced thread finishes. Figure 7.2 shows the execution of the program using `join()`.

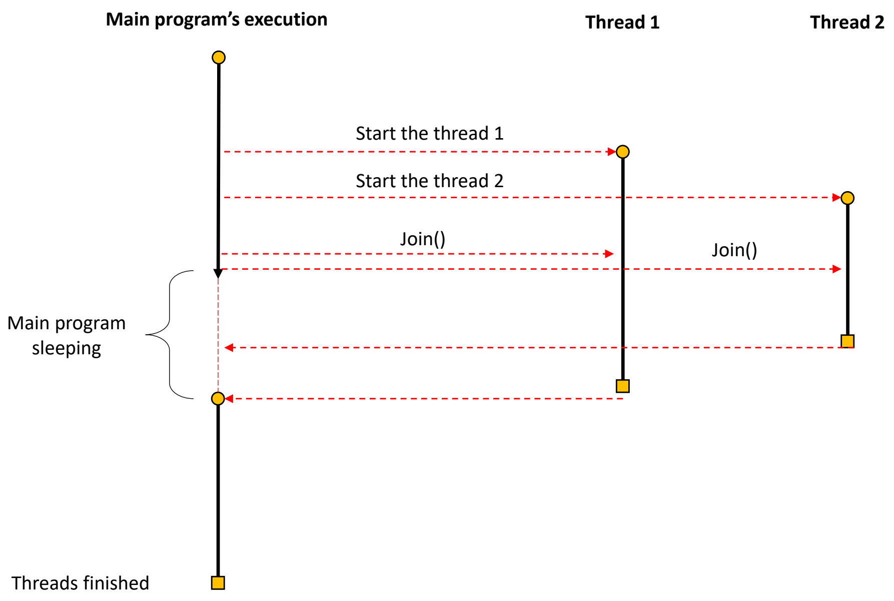

Figure 7.2: Diagram shows the program's flow when we use the `join()` method. We can see that the main program will sleep until *thread 1* finishes. The *thread 2* keeps running independently to the other thread and the main program.

Now let's see the same previous example but incorporating the `join()` method after threads start running.

```

# Creating threads
t1 = Service(5)
w1 = Worker(2)
w2 = Worker(4)

# Starting threads
t1.start()
w1.start()
```

```
w2.start()

# Here we call the join() method to block the main program.
# The other threads keep running independently
t0 = time.time()
w1.join()
print('Main program waits for: {}'.format(time.time() - t0))
```

```
Thread 1 starting...
Thread 2 starting...
Thread 3 starting...
Thread 2 exiting...
Main program waits for: 2.000131607055664
Thread 1 exiting...
Thread 3 exiting...
```

IsAlive()

We can identify if a thread finished its execution using the `IsAlive()` method or the `is_alive` attribute, for example, after using `join()`. The following example shows the way to use `IsAlive()` to check if a thread is still running after a certain amount of time.

```

t = Service(4)
t.start()

# The main program will wait 5 seconds after 't' has finished executing
# before continuing its execution.
t.join(5)

# This returns true if the thread is not currently executing
if not t.isAlive():
    print('The thread has finished successfully')
else:
    print('The thread is still executing')
```

```
Thread-1 starting...
```

```
Thread-1 exiting...
The thread has finished successfully
```

We can avoid the use of too many `prints` that help us with the tracking of threads, by using the *logging* library. Every time we make a log we have to embed the name of each thread on its log message, as shown in the following example:

```python
# code5.py

import threading
import time
import logging


# This sets ups the format in which the messages will be logged on console
logging.basicConfig(level=logging.DEBUG, format='[%(levelname)s]'
                                        '(%(threadName)-10s) %(message)s')

class Worker(threading.Thread):

    def __init__(self, t):
        super().__init__()
        self.t = t

    def run(self):
        logging.debug('Starting')
        time.sleep(self.t)
        logging.debug('Exiting')


class Service(threading.Thread):

    def __init__(self, t):
        super().__init__()
        self.t = t

    def run(self):
        logging.debug('Starting')
        time.sleep(self.t)
        logging.debug('Exiting')


# Creating threads
t1 = Service(4)
w1 = Worker(2)
w2 = Worker(2)

# Starting threads
w1.start()
w2.start()
t1.start()
```

```
[DEBUG](Thread-2  ) Starting
[DEBUG](Thread-3  ) Starting
[DEBUG](Thread-1  ) Starting
[DEBUG](Thread-2  ) Exiting
[DEBUG](Thread-3  ) Exiting
[DEBUG](Thread-1  ) Exiting
```

Daemon Threads

In general, the main program waits for all the threads to finish before ending its execution. *Daemon threads* let the main program to kill them off after other threads (and itself) finish. Daemon threads do not prevent the main program to end. In this way, daemon threads will only run until the main program finishes.

```
# code7.py

import threading
import time


class Worker(threading.Thread):

    def __init__(self, t):
        super().__init__()
        self.t = t

    def run(self):
        print("{} starting...".format(threading.currentThread().getName()))
        time.sleep(self.t)
        print("{} exiting...".format(threading.currentThread().getName()))


class Service(threading.Thread):

    def __init__(self, t):
        super().__init__()
        self.t = t
        # We can set a thread as deamon inside the class definition
        # setting the daemon attribute as True
        self.daemon = True

    def run(self):
        print("{} starting...".format(threading.currentThread().getName()))
        time.sleep(self.t)
        print("{} exiting...".format(threading.currentThread().getName()))


# Creating threads
t1 = Service(5)
w1 = Worker(2)

# Setting the working thread as daemon
# We can use this same method when we define a function as target
# of a thread.
w1.setDaemon(True)

# Executing threads
```

```
44  w1.start()
45  t1.start()
```

```
Thread 2 starting...
Thread 1 starting...
```

The previous example explains the use of daemon threads. The console output shows how threads are interrupted abruptly after the main program ends its execution. We can compare this output with the output of the next example, configuring threads as daemon (removing lines 24 and 39):

```
Thread-2 starting...
Thread-1 starting...
Thread-2 exiting...
Thread-1 exiting...
```

Note that threads complete the execution and the program did not close until both threads finished. If for any reason, we require waiting for a daemon thread during an amount of time, we can specify that amount (in seconds) in the `join()` method:

```python
# code9.py

import threading
import logging
import time

logging.basicConfig(level=logging.DEBUG,format='(%(threadName)-10s) '
                                                            '%(message)s')


class DaemonThread(threading.Thread):

    def __init__(self, t):
        super().__init__()
        self.t = t
        self.daemon = True
        self.name = 'daemon'

    def run(self):
        logging.debug('Starting')
        time.sleep(self.t)
        logging.debug('Exiting')

class NonDaemonThread(threading.Thread):

    def __init__(self, t):
        super().__init__()
        self.t = t
        self.name = 'non-daemon'

    def run(self):
        logging.debug('Starting')
        time.sleep(self.t)
        logging.debug('Exiting')

# Creating threads
d = DaemonThread(3)
t = NonDaemonThread(1)

# Executing threads
d.start()
t.start()

# Waiting thread d for 1 seconds
d.join(2)
print('is d alive?: {}'.format(d.isAlive()))
```

```
(daemon    ) Starting
(non-daemon) Starting
(non-daemon) Exiting
is d alive?: True
```

Timers

The class `Timer` is a subclass of the class `Thread` and allows us to execute a process or an action after a certain amount of time has passed. `Timer` requires as basic parameters the time in seconds after which the thread starts running, the name of the process to execute and the entry arguments for the process. The `cancel()` method allows us, if required, to cancel the execution of the timer before its begins.

```python
# code10.py

import threading

def delayed_message(msg):
    print("Message:", msg)

t1 = threading.Timer(10.0, delayed_message, args=("This is a thread t1!",))
t2 = threading.Timer(5.0, delayed_message, args=('This is a thread t2!',))
t3 = threading.Timer(15.0, delayed_message, args=('This is a thread t3!',))

# This thread will start after 10 seconds
t1.start()

# This thread will start after 5 seconds
t2.start()

# Here we cancel thread t1
t1.cancel()

# This thread will start after 15 seconds
t3.start()
```

```
Message: This is a thread t2!
Message: This is a thread t3!
```

7.2 Synchronization

Threads run in a non-deterministic way. Therefore, there are some situations in which more than one thread must share the access to certain resources, such as files and memory. During this process, **only one** thread have access to the resource, and the remaining threads must wait for it. When there is multiple *concurrence* to a resource it is possible to control the access through synchronization mechanisms among the threads.

Locks

Locks allow us to synchronize the access to shared resources between two or more threads. The *Threading* library provides us with the `Lock` class which allows the synchronization. A lock has two states: *locked* and *unlocked*. The default state is *unlocked*. When a given thread t_i attempts to execute, first it tries to acquire the lock (with the `acquire()` method). If another thread t_j takes the lock, t_i must wait for t_j to finish and release it (with the `release()` method) to have the chance to acquire the lock. Once t_i acquires the lock, it can start executing. Figure 7.3 shows a general scheme of synchronization between threads using locks.

```python
# code11.py

import threading


# This class models a thread that blocks to a file
class MyThread(threading.Thread):

    lock = threading.Lock()

    def __init__(self, i, file):
        super().__init__()
        self.i = i
        self.file = file

    # This method is the one executed when the start() method is called.
    def run(self):

        # Blocks other threads from entering the next block
        MyThread.lock.acquire()
        try:
            self.file.write('This line was written by thread #{}\n'.format(self.i))
        finally:
            # Releases the resource
            MyThread.lock.release()


if __name__ == '__main__':
    n_threads = 15
    threads = []

    # We create a file to write the output.
    with open('out.txt', 'w') as file:

        # All writing threads are created at once
        for i in range(n_threads):

            my_thread = MyThread(i, file)

            # The thread is started, which executes the run() method.
            my_thread.start()
            threads.append(my_thread)
```

```
This line was written by thread #0
This line was written by thread #1
This line was written by thread #2
This line was written by thread #3
This line was written by thread #4
This line was written by thread #5
This line was written by thread #6
This line was written by thread #7
This line was written by thread #8
This line was written by thread #9
This line was written by thread #10
This line was written by thread #11
This line was written by thread #12
This line was written by thread #13
This line was written by thread #14
```

Fortunately in Python *locks* can also work inside a *context manager* through the `with` sentence. In this case, is the

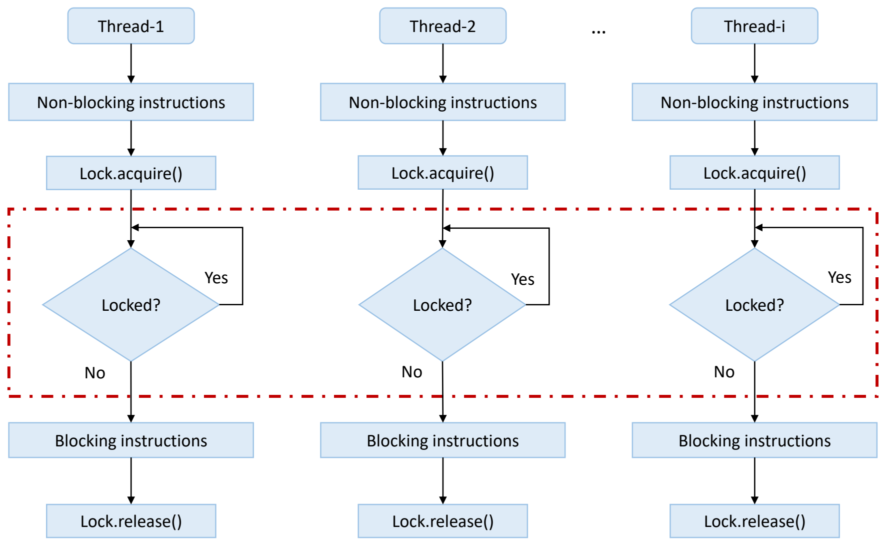

Figure 7.3: This image shows a general example where a set of *i* threads are running. Whenever a thread acquires the lock, the rest of the threads must wait to be able to execute. After the thread releases the lock, the other threads can acquire it and run their (blocking) instructions.

same `with` that is in charge of calling the `acquire()` and `release()` methods. For example, the *locks* used in the `run` method from the previous example can be implemented as shown:

```python
def run(self):
    with MyThread.lock:
        self.file.write(
            'This line was written by thread #{}\n'.format(self.i))
```

A common problem in concurrent programming is the *Producer-Consumer* pattern. This problem arises when two or more threads, known as *producers* and *consumers*, access to the same storage space or *buffer*. Under this scheme, producers put items in the *buffer* and consumers pull items out of the *buffer*. This model allows the communication between different threads. In general the *buffer* shared in this model is implemented through a *synchronized queue* or *secure queue*.

For example, let's assume that we can separate a program that processes a text file with numbers in two independent threads. The first thread is in charge of reading the file and appending the values to a queue. The second thread stores into another file the sum of the tuples of numbers previously added into the queue. We communicate both threads through a synchronized queue implemented as shown next:

```python
# code13.py

import collections
import threading

class MyDeque(collections.deque):

    # We inherit from a normal collections module Deque and
    # we add the locking mechanisms to ensure thread
    # synchronization

    def __init__(self):
        super().__init__()
        # A lock is created for this queue
        self.lock = threading.Lock()

    def append(self, element):

        # The lock is used within a context manager
        with self.lock:
            super().append(element)
            print('[ADD] queue now has {} elements'.format(len(self)))

    def popleft(self):
        with self.lock:
            print('[REMOVE] queue now has {} elements'.format(len(self)))
            return super().popleft()
```

Now let's see the rest of the implementation of the producer and the consumer. As a recommendation, we encourage the read to try the examples directly in a terminal or using a IDE such as *PyCharm*.

```python
# code14.py

import time
import threading

```

```python

class Producer(threading.Thread):
    # This thread is implemented as a class

    def __init__(self, queue):
        super().__init__()
        self.queue = queue

    def run(self):
        # We open the file using a context manager. We explain this in details
        # in Chapter 10.
        with open('raw_numbers.txt') as file:
            for line in file:
                values = tuple(map(int, line.strip().split(',')))
                self.queue.append(values)


def consumer(queue):
    # This thread is implemented as a function

    with open('processed_numbers.txt', 'w') as file:
        while len(queue) > 0:
            numbers = queue.pop()
            file.write('{}\n'.format(sum(numbers)))

            # Simulates that the consumer is slower than the producer
            time.sleep(0.001)


if __name__ == '__main__':

    queue = MyDeque()

    p = Producer(queue)
    p.start()

    c = threading.Thread(target=consumer, args=(queue,))
    c.start()
```

```
[ADD] queue now has 1 elements
[ADD] queue now has 2 elements
[ADD] queue now has 3 elements
[ADD] queue now has 4 elements
[ADD] queue now has 5 elements
```

Deadlock

In the context of multithreading-based applications, there is an innocent but dangerous situation in programs that use locks. This case is commonly called *deadlock*. A deadlock occurs when two or more threads are stuck waiting for each other to release a resource. For example, let `FirstProcess` be a thread that acquires a lock `a` and requests for a lock `b` so it can release the lock `a`. Let `SecondProcess` be another thread that already acquired the lock `b` and is waiting for the lock `a` before it releases the lock `b`. We note the deadlock because of our program will be frozen without getting a runtime error or crash. The next code example shows a template of a deadlock according to with the situation described:

```python
# code15.py

import threading


class FirstProcess(threading.Thread):

    def __init__(self, lock_a, lock_b):
        self.lock_a = lock_a
        self.lock_b = lock_b

    def run(self):
        with self.lock_a:
            # Acquire the first lock

            with self.lock_b:
```

```
            # Acquire the second lock for another concurrent task


class SecondProcess(threading.Thread):

    def __init__(self, lock_a, lock_b):
        self.lock_a = lock_a
        self.lock_b = lock_b

    def run(self):
        with self.lock_b:
            # Acquire the first lock
            # Notice that this thread require the lock_b, that
            # could be taken fot other thread previously

            with self.lock_a:
                # Acquire the second lock for another concurrent task


lock_a = threading.Lock()
lock_b = threading.Lock()

t1 = FirstProcess(lock_a, lock_b)
t2 = SecondProcess(lock_a, lock_b)
t1.start()
t2.start()
```

We can decrease the risk of a deadlock by restricting the number of locks that a threads can acquire at a time.

Queue

Fortunately, Python has an optimized library for secure queues management in *producer-consumer* models. The `queue` library has a implemented queue that safely manages multiples concurrences. It is different to the queue implemented in `collections` library used in data structures because that one does not have locks for synchronization.

The main queue methods in the `queue` library are:

- `put()`: Adds an item to the queue (push)
- `get()`: Removes and returns an item from the queue (pop)
- `task_done()`: Requires to be called each time an item has been processed
- `join()`: Blocks the queue until all the items have been processed

Recall the text file processing example shown before. The implementation using the `queue` library is as follows:

```python
# code16.py

import threading
import time
import queue


class Producer(threading.Thread):

    def __init__(self, que):
        super().__init__()
        self.que = que

    def run(self):
        with open('raw_numbers.txt') as file:
            for line in file:
                values = tuple([int(l) for l in line.strip().split(',')])
                self.que.put(values)
                print('[PRODUCER] The queue has {} elements.'.format( \
                    self.que.qsize()))

                # Simulates a slower process
                time.sleep(0.001)


def consumer(que):
    with open('processed_numbers.txt', 'w') as file:
        while True:

            # A try/except clause is used in order to stop
            # the consumer once there is no elements left in the
            # queue. If not for this, the consumer would be executing
            # for ever

            try:

                # If no elements are left in the queue, an Empty
                # exception is raised
                numbers = que.get(False)
            except queue.Empty:
                break
            else:
                file.write('{}\n'.format(sum(numbers)))
                que.task_done()

                # qsize() returns the queue size
                print('[CONSUMER] The queue now has {} elements.'.format( \
                    que.qsize()))

                # Simulates a complex process. If the consumer was faster
                # than the producer, the threads would end abruptly
                time.sleep(0.005)


if __name__ == '__main__':

    q = queue.Queue()

    # a producer is created and executed
    p = Producer(q)
    p.start()

    # a consumer thread is created and executed with the same queue
```

```python
    c = threading.Thread(target=consumer, args=(q,))
    c.start()
```

```
[PRODUCER] The queue has 1 elements.
[CONSUMER] The queue now has 0 elements.
[PRODUCER] The queue has 1 elements.
[PRODUCER] The queue has 2 elements.
[PRODUCER] The queue has 3 elements.
[PRODUCER] The queue has 4 elements.
[CONSUMER] The queue now has 3 elements.
[CONSUMER] The queue now has 2 elements.
[CONSUMER] The queue now has 1 elements.
[CONSUMER] The queue now has 0 elements.
```

7.3 Hands-On Activities

Activity 7.1

ALERT! Godzilla has arrived at Santiago! Soldiers need to simulate a battle against Godzilla. The simulation will contribute deciding whether it is better to run away or to fight him. With this purpose, Soldiers have given us a report with the specifications that we have to accomplish. These specifications are:

- There is just one Godzilla and several soldiers.
- Each soldier has an attack speed, remaining life (HP), and strength of attack (damage).
- Godzilla attacks every eight seconds, affecting all soldiers by decreasing their HP in three units.
- Each time a soldier attacks Godzilla, it attacks back decreasing one-fourth of the soldiers' attack to his HP.
- The soldiers' attack speed is random between 4 and 19 seconds.
- You must create one new soldier every x seconds, where x has to be previously defined by you.

Activity 7.2

Congratulations! Thanks to the previous simulation (7.3), the Army has realized of its superiority against Godzilla. Santiago is safe again, or that is what we believe. The truth is that the epic battle has been nothing but a simulation made by Godzilla to decide if he attacks Santiago or not. Now Santiago will be faced by Mega-Godzilla (Godzilla in its ultimate form), with all of the powers he has not shown before. The task is to simulate the battle between Mega-Godzilla and the Army so it can be written in history books. The simulation must:

- Contain several soldiers and one Mega-Godzilla.
- Each soldier has an attack speed, remaining life (HP), and strength of attack (damage).
- Mega-Godzilla attacks every N seconds, where N is a random number between 3 and 6. N has to be reset after each action.
- Mega-Godzilla attacks in the following ways:
  - Normal attack: Mega-Godzilla stomps affecting all soldiers. This attack causes a three units damage to each soldier.
  - Scaly Skin: Each time a soldier attacks Mega-Godzilla, it attacks back decreasing one-fourth of the soldiers' attack to his HP.
  - Ultimate Mega-Godzilla Super Attack: Mega-Godzilla screams causing a six units damage to every soldier. Also, the scream stuns all the soldiers for 10 seconds. During this period, soldiers can not perform any action.
- Soldiers' attack speed is random between 4 and 19 seconds.
- Soldiers' attack lasts a random number between 1 and 3 seconds.
- Only one soldier can attack Mega-Godzilla at a time.