I. Concurrency

1. Write a function **`getpositions(fnom, chunksize)`** where **`fnom`** is the name of a file and **`chunksize`** is a positive integer. The function should return a list of file positions that would divide the file contents into segments of length **`chunksize`**, with the possible exception that the last segment could have a positive size less than **`chunksize`**.

```
def getpositions(fnom, chunksize):

    # List to store positions (first index 0)
    positions = [0]

    # Open file
    with open(fnom, 'r') as file:

        # Loop until end of file
        while True:

            # Read chunk from file
            if not file.read(chunksize):

                # End loop
                break

            # Get position of cursor
            position = file.tell()

            # Append position to list
            positions.append(position)

    # Return list
    return positions
```

2. Write a function **chunk_counter(f, pos, csize, b)**, where **f** is a file object and **csize** is a positive integer, that returns the number of occurrences of byte **b** in the segment of **f** starting at position **pos** and having length at most **csize**.

```
# Global count variable
count = 0

def chunk_counter(f, pos, csize, b):

    # Allow function to modify global variable
    global count

    # Move cursor position
    f.seek(pos)

    # Read data from file
    data = f.read(csize)

    # For each byte in data read
    for byte in data:

        # If byte matches the one being searched for
        if byte == b:

            # Increment count
            count += 1

    # Return count
    return count
```

3. Write a function **`total_counter(fnom, b)`** that returns the total number of occurrences of byte **`b`** in the file named **`fnom`**. Your function should use multithreading with and use the functions **`getpositions`** and **`chunk_counter`**; assume that your processor has 8 cores.

```
def total_counter(fnom, b):

	# Get size of file
	fsize = os.path.getsize(fnom)

	# Chunk size (assuming 100 threads)
	csize = ceil(fsize / 100)

	# Get thread starting indices
	idxs = getpositions(fnom, csize)

	# List containing running threads
	threads = []

	# Open file
	with open(fnom, 'rb') as file:

		# For each thread
		for x in range(100):

			# Create new thread
			thread = threading.Thread(target=chunk_counter,
				args=(file, idxs[x], csize, b))

			# Append to list of threads
			threads.append(thread)

		# For each thread
		for x in range(100):

			# Start each thread
			threads[x].start()

		# For each thread
		for x in range(100):

			# Wait for each thread
			threads[x].join()

	# Return count
	return count
```

4. Would using multiprocessing instead of multithreading run slower or faster than the multithreaded version?

   For small files, the difference in performance between multithreading and multiprocessing will either be negligible or slightly in favor of multithreading, due to the slightly longer time it takes to spawn processes. For larger files, where searching is more resource intensive, multiprocessing would be the clear winner, since the negligible time it takes to spawn processes is outweighed by the performance they afford.

II. Regular Expressions

5. For each of the following create a single regular expression that:

   a. recognizes the following strings: “bat”, “bit”, “but”, “hat”, “hit”, or “hut”.

      ```
      ^[bh][aiu][t]$
      ```

   b. matches any word and single letter separated by a comma and single space, as in last name, first initial.

      ```
      ^[A-Za-z]+\, [A-Za-z]$
      ```

   c. matches a one or two digit number string representation of a month of the year (January,..., December).

      ```
      ^[1-9]$|^0[1-9]$|^1[0-2]$
      ```