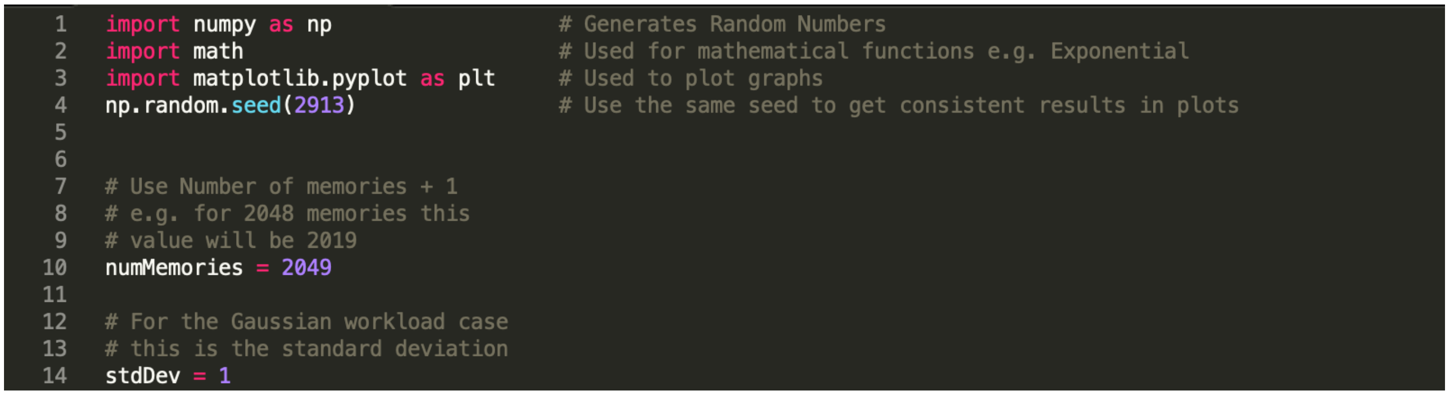
Simulating the Working of Processors and Memory Modules

Muhammad Talha Khan (46281174)
CS-230 Distributed Computer Systems

Simulation Code & Explaination

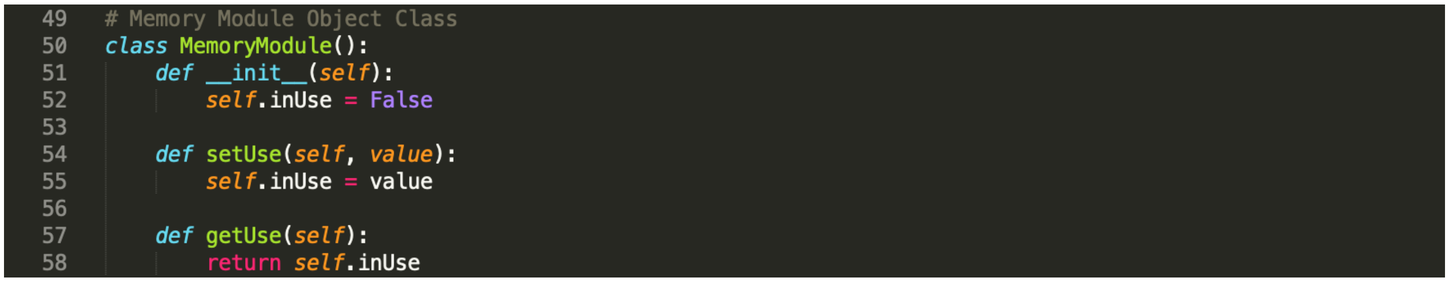
The code for this assignment was written in Python. An object oriented approach was used to simulate this problem in particular. The code snippets below help understand the logic that was used to build this simulation. Most of the code is explained with comments used inside of the code snippets.

Imports & Global Variables



```python
import numpy as np                  # Generates Random Numbers
import math                         # Used for mathematical functions e.g. Exponential
import matplotlib.pyplot as plt     # Used to plot graphs
np.random.seed(2913)                # Use the same seed to get consistent results in plots


# Use Number of memories + 1
# e.g. for 2048 memories this
# value will be 2019
numMemories = 2049

# For the Gaussian workload case
# this is the standard deviation
stdDev = 1
```

The Memory Module Class



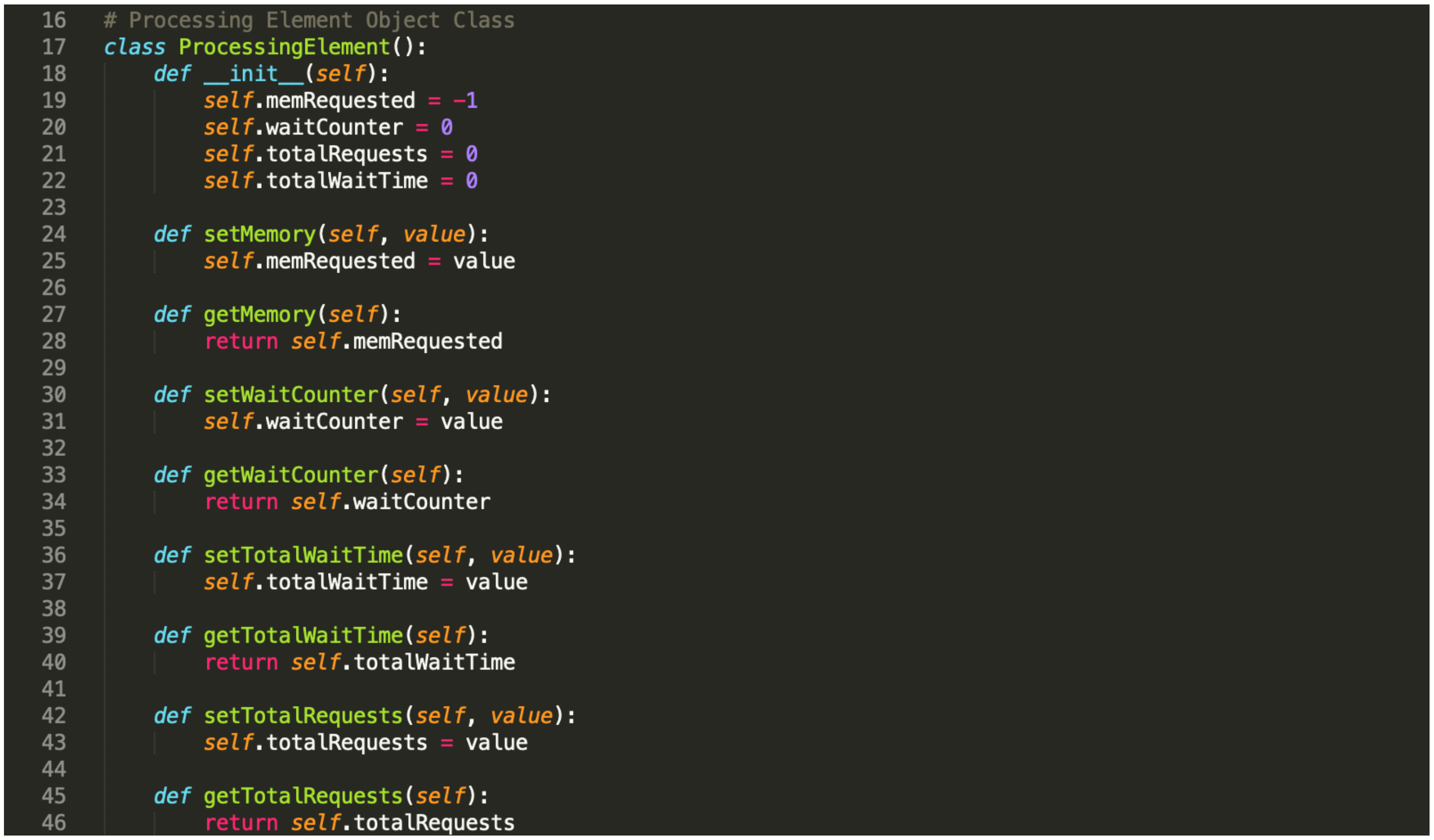
```python
# Memory Module Object Class
class MemoryModule():
    def __init__(self):
        self.inUse = False

    def setUse(self, value):
        self.inUse = value

    def getUse(self):
        return self.inUse
```

The memory module object contains only one attribute named ***inUse***, which is of type boolean with a default value of ***False***. It tells us whether a certain memory module is currently being used by a processor or not. The ***setUse*** and ***getUse*** methods defined in the Class are just the setter and getter methods of the attribute ***inUse*** respectively.

Cycle Number	1	2	3	4	5	6	7
Access (1) / No Access (0)	0	0	0	1	0	0	1
waitCounter	1	2	3	0	1	2	0
totalWaitTime	1	2	3	3	4	5	5

Table 1: Understanding Different Wait Times

The Processing Element Class



```python
# Processing Element Object Class
class ProcessingElement():
    def __init__(self):
        self.memRequested = -1
        self.waitCounter = 0
        self.totalRequests = 0
        self.totalWaitTime = 0

    def setMemory(self, value):
        self.memRequested = value

    def getMemory(self):
        return self.memRequested

    def setWaitCounter(self, value):
        self.waitCounter = value

    def getWaitCounter(self):
        return self.waitCounter

    def setTotalWaitTime(self, value):
        self.totalWaitTime = value

    def getTotalWaitTime(self):
        return self.totalWaitTime

    def setTotalRequests(self, value):
        self.totalRequests = value

    def getTotalRequests(self):
        return self.totalRequests
```

The processing element object contains 4 attributes:

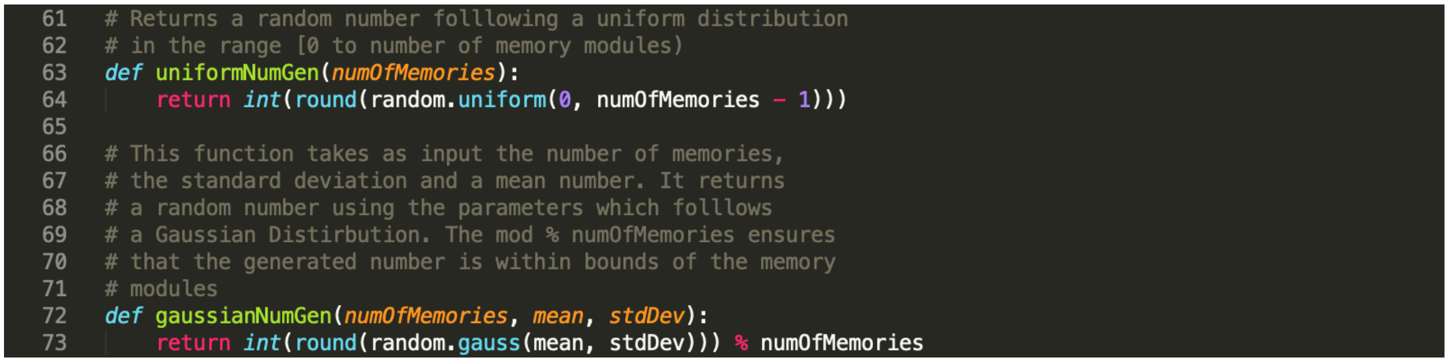
1) The ***memRequested*** attribute stores the memory module requested by the processor. If in a particular cycle a processor is not granted access to its requested memory module then ***memRequested*** stores the id(number) of the requested memory module so that the same processor can request the same memory module the next cycle. On the contrary, If a processor is granted access to its desired memory module then ***memRequested*** stores a value -1 indicating that the processor will generate a new request the next cycle. The setter and getter methods for ***memRequested*** are ***setMemory*** and ***getMemory*** respectively.

2) The ***waitCounter*** attribute stores the number of memory cycles a processor has been waiting since it was last granted access to its requested memory module. Think of it as a counter which counts the number of cycles before a processor is granted access to its requested memory module after which this counter restarts from 0. **Table 1** above clarifies this. The setter and getter methods for ***waitCounter*** are ***setWaitCounter*** and ***getWaitCounter*** respectively.

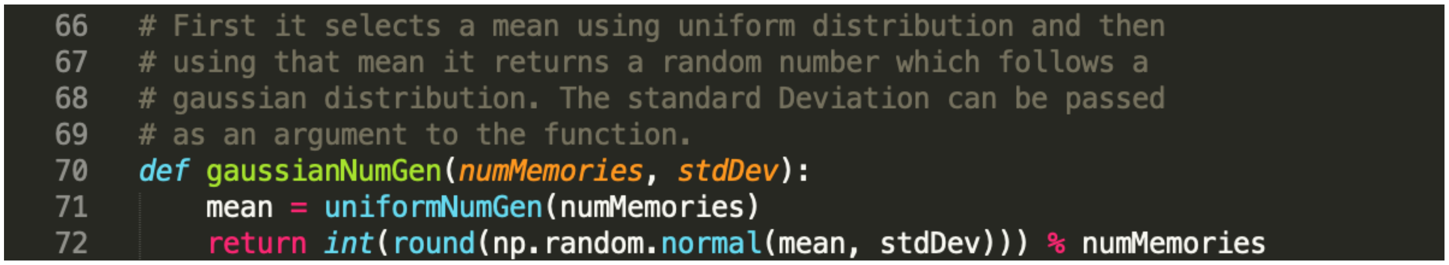
3) The ***totalWaitTime*** attribute stores the total number of memory cycles a processor has been waiting so far. Think of it as a counter which counts the number of memory cycles a processor was rejected access or had to wait for its requested memory module. **Table 1** above clarifies this. The setter and getter methods for ***totalWaitTime*** are ***setTotalWaitTime*** and ***getTotalWaitTime*** repectively.

4) The ***totalRequests*** attributes counts the number of new memory requests a processor has made so far. Continuing requests from previous cycles do not increment this. The setter and getter methods for ***totalRequests*** are ***setTotalRequests*** and ***getTotalRequests*** respectively.

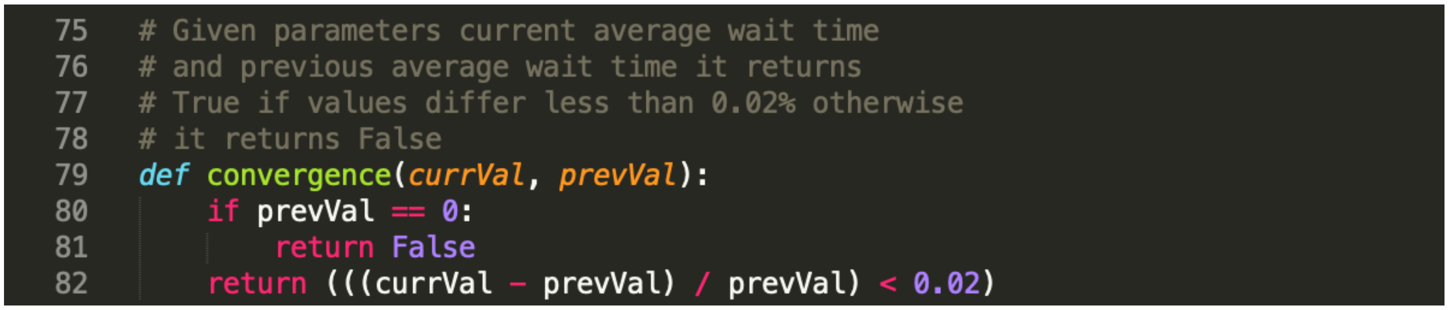
Functions



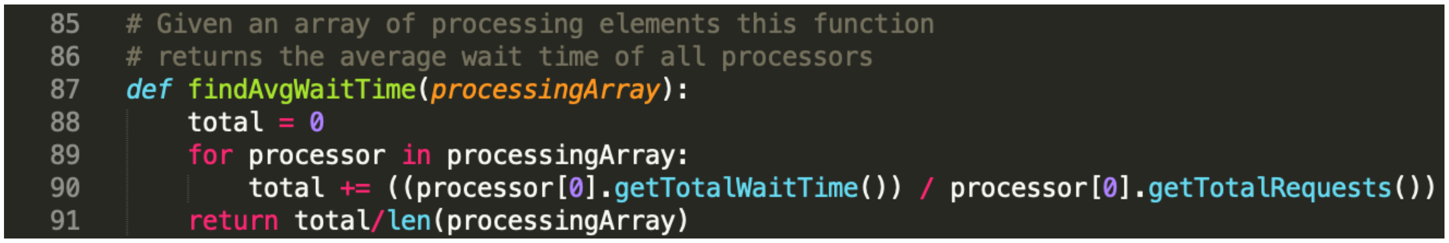
```python
# Returns a random number folllowing a uniform distribution
# in the range [0 to number of memory modules)
def uniformNumGen(numOfMemories):
    return int(round(random.uniform(0, numOfMemories - 1)))

# This function takes as input the number of memories,
# the standard deviation and a mean number. It returns
# a random number using the parameters which folllows
# a Gaussian Distirbution. The mod % numOfMemories ensures
# that the generated number is within bounds of the memory
# modules
def gaussianNumGen(numOfMemories, mean, stdDev):
    return int(round(random.gauss(mean, stdDev))) % numOfMemories
```



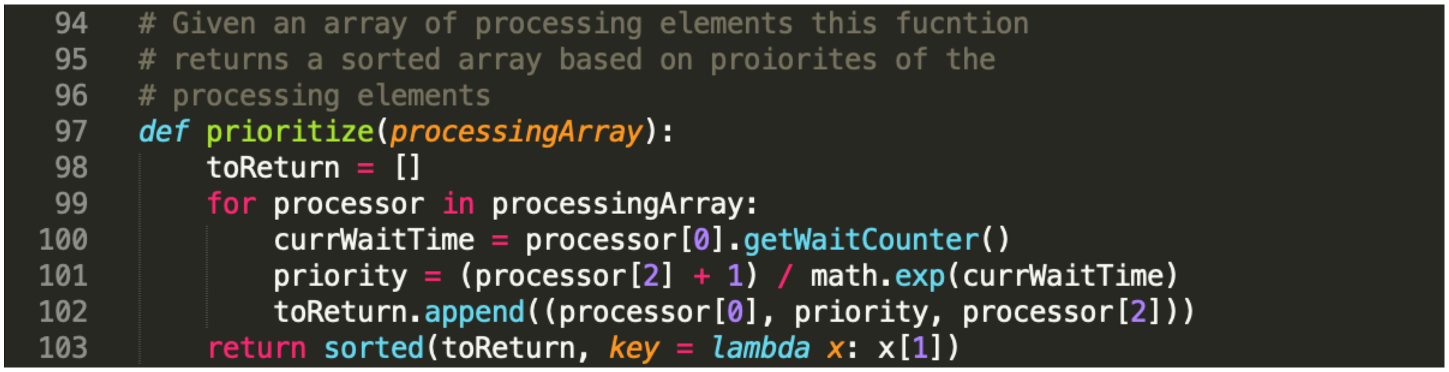
```python
# First it selects a mean using uniform distribution and then
# using that mean it returns a random number which follows a
# gaussian distribution. The standard Deviation can be passed
# as an argument to the function.
def gaussianNumGen(numMemories, stdDev):
    mean = uniformNumGen(numMemories)
    return int(round(np.random.normal(mean, stdDev))) % numMemories
```



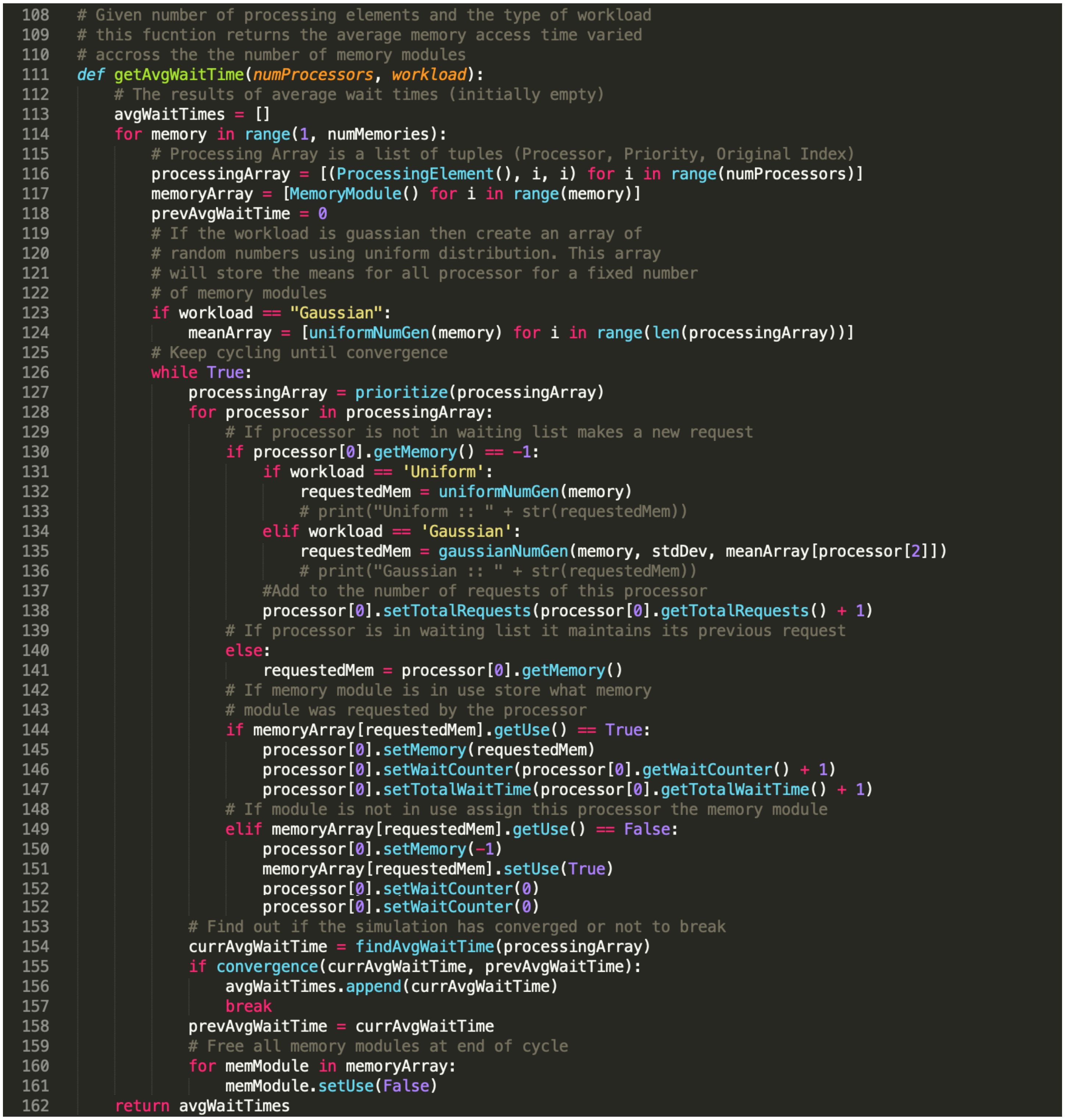
```python
# Given parameters current average wait time
# and previous average wait time it returns
# True if values differ less than 0.02% otherwise
# it returns False
def convergence(currVal, prevVal):
    if prevVal == 0:
        return False
    return (((currVal - prevVal) / prevVal) < 0.02)
```



```python
# Given an array of processing elements this function
# returns the average wait time of all processors
def findAvgWaitTime(processingArray):
    total = 0
    for processor in processingArray:
        total += ((processor[0].getTotalWaitTime()) / processor[0].getTotalRequests())
    return total/len(processingArray)
```



```python
# Given an array of processing elements this fucntion
# returns a sorted array based on proiorites of the
# processing elements
def prioritize(processingArray):
    toReturn = []
    for processor in processingArray:
        currWaitTime = processor[0].getWaitCounter()
        priority = (processor[2] + 1) / math.exp(currWaitTime)
        toReturn.append((processor[0], priority, processor[2]))
    return sorted(toReturn, key = lambda x: x[1])
```



```
108 # Given number of processing elements and the type of workload
109 # this fucntion returns the average memory access time varied
110 # accross the the number of memory modules
111 def getAvgWaitTime(numProcessors, workload):
112     # The results of average wait times (initially empty)
113     avgWaitTimes = []
114     for memory in range(1, numMemories):
115         # Processing Array is a list of tuples (Processor, Priority, Original Index)
116         processingArray = [(ProcessingElement(), i, i) for i in range(numProcessors)]
117         memoryArray = [MemoryModule() for i in range(memory)]
118         prevAvgWaitTime = 0
119         # If the workload is guassian then create an array of
120         # random numbers using uniform distribution. This array
121         # will store the means for all processor for a fixed number
122         # of memory modules
123         if workload == "Gaussian":
124             meanArray = [uniformNumGen(memory) for i in range(len(processingArray))]
125         # Keep cycling until convergence
126         while True:
127             processingArray = prioritize(processingArray)
128             for processor in processingArray:
129                 # If processor is not in waiting list makes a new request
130                 if processor[0].getMemory() == -1:
131                     if workload == 'Uniform':
132                         requestedMem = uniformNumGen(memory)
133                         # print("Uniform :: " + str(requestedMem))
134                     elif workload == 'Gaussian':
135                         requestedMem = gaussianNumGen(memory, stdDev, meanArray[processor[2]])
136                         # print("Gaussian :: " + str(requestedMem))
137                     #Add to the number of requests of this processor
138                     processor[0].setTotalRequests(processor[0].getTotalRequests() + 1)
139                 # If processor is in waiting list it maintains its previous request
140                 else:
141                     requestedMem = processor[0].getMemory()
142                 # If memory module is in use store what memory
143                 # module was requested by the processor
144                 if memoryArray[requestedMem].getUse() == True:
145                     processor[0].setMemory(requestedMem)
146                     processor[0].setWaitCounter(processor[0].getWaitCounter() + 1)
147                     processor[0].setTotalWaitTime(processor[0].getTotalWaitTime() + 1)
148                 # If module is not in use assign this processor the memory module
149                 elif memoryArray[requestedMem].getUse() == False:
150                     processor[0].setMemory(-1)
151                     memoryArray[requestedMem].setUse(True)
152                     processor[0].setWaitCounter(0)
152                     processor[0].setWaitCounter(0)
153             # Find out if the simulation has converged or not to break
154             currAvgWaitTime = findAvgWaitTime(processingArray)
155             if convergence(currAvgWaitTime, prevAvgWaitTime):
156                 avgWaitTimes.append(currAvgWaitTime)
157                 break
158             prevAvgWaitTime = currAvgWaitTime
159             # Free all memory modules at end of cycle
160             for memModule in memoryArray:
161                 memModule.setUse(False)
162     return avgWaitTimes
```

Discussion

getAvgWaitTime is the most important function of the simulation and uses all of the functions described above. Given a number of processors it returns an array of average wait times, initialized in (line 113), for each memory module ranging from 1 to the number that we have defined ***numMemories*** (line 10). The function consists of three loops. The most outer loop (line 114) traverses over the number of memory modules , the middle loop (line 126) traverses over the memory cycles and the inner most loop (line 128) traverses over the processing elements array or processors.

The processing elements array (line 116) is a list of tuples. Where each tuple consists of 3 values: (Processing Element, Priority, Index). The memory modules array (line 117) is just a list containing memory module objects. Also, before starting the memory cycles there is a check to see whether the workload is Gaussian (line 123). If that is the case than an array of means is generated which stores the mean of each processor. The mean is generated using a Uniform distribution. We generate this array at the very beginning of the simulation. The only time the mean array changes in when the number of memory modules change.

At the start of each memory cycle the processing element array is sorted based on priorities of the processors, the ***prioritize*** function (line 97) does this. Prioritizing helps to avoid starvation of higher indexed processors. Since in our model processors with lower index have priority over processors with higher index. E.g. P_1 has priority over P_5. This can lead to starvation and hence the need for prioritizing arises. Instead of the total wait time (***totalWaitTime***) of a processor the current wait time is used to prioritize. The current wait time (***waitCounter***) corresponds to the number of memory cycles that have passed since the last time the processor was granted access to its requested memory module. Priority is calculated based on **equation 1** below. The +1 in the numerator of the equation ensures that the processor at index 0 not always remains at top priority and the exponential in the denominator ensures that it is always greater than 0. This method to calculate priority lead to quite low average wait times. I compared this with other methods E.g. one was just assigning priority based on the total wait time and the other was just the same as I am using now but had the squared value of the wait counter in the denominator instead of the exponent. After sorting the processing elements the first if (line 130) checks if the processor needs to make a new request. If that is the case then based on the specified workload a random number is generated which corresponds to a memory module. Also the total requests so far of a processor are incremented by one. If the processor does not need to make a new request (line 140), meaning it is still waiting for its previous request, than a new request is not generated as the processor will maintain its previous request.

$$priority = \frac{ProcessorIndex + 1}{exp(waitCounter)} \tag{1}$$

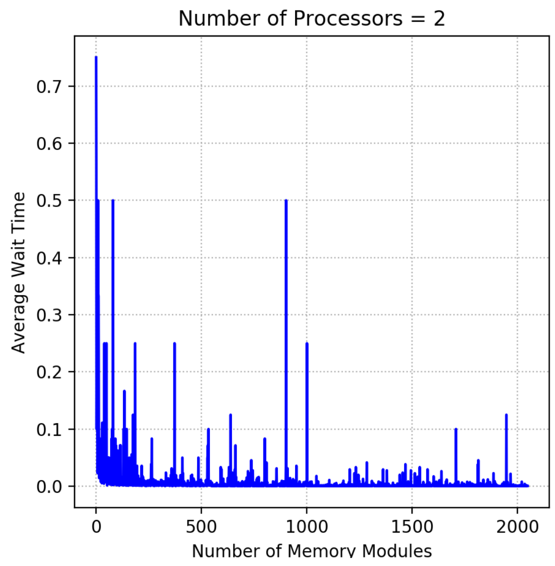
After the processor generates a request we look in the memory module array, if the requested module is currently in use or not. If it is in use (line 144) the processing element stores the requested memory module number, so it can be used the next cycle rather than generating a new request, also the ***waitCounter*** and ***totalWaitTime*** are incremented by one. If the memory module is free, it is assigned to the processor and is flagged as busy, the processor stores -1 as its requested memory module so that the next cycle we know it has to generate a new request (line 130) and ***waitCounter*** of the processor is set to zero.

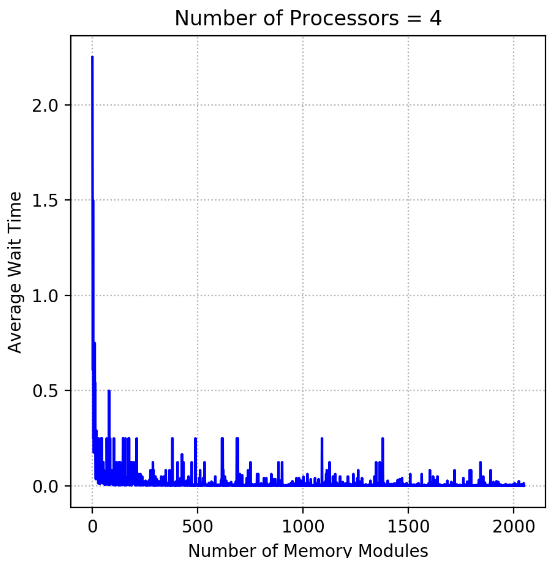
Once we have traversed over all the processing elements meaning completed a cycle, the average wait time of all processors is calculated. The function ***findAvgWaitTime*** (line 87) is used to calculate this. **Equation 2** below shows how the average wait time is calculated. N in the equation corresponds to the total number of processors.

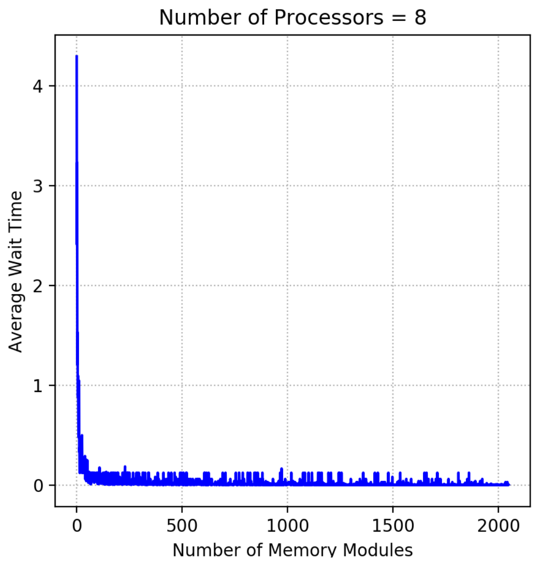
$$averageWaitTime = \frac{\sum_{p=1}^{N} \frac{totalWaitTime}{totalRequests}}{N} \tag{2}$$

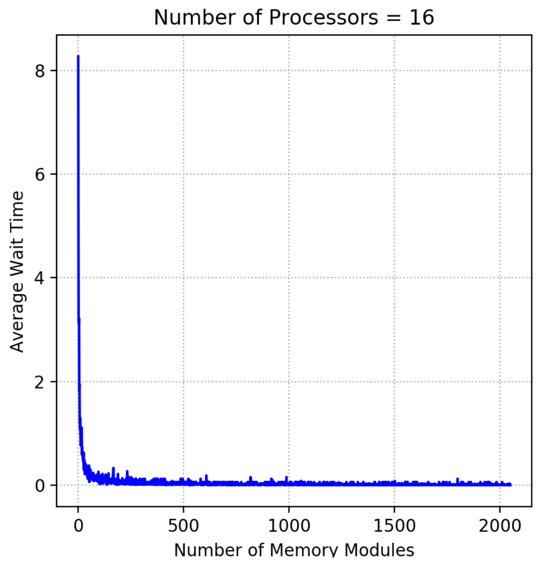
After calculating the current average wait time, the previous average wait time and the current average wait time are compared. If the percentage change is less than 0.02% the result is appended to the average wait times array (which is returned at the end) and the memory cycles are stopped for the current configuration. The number of memory modules is incremented by one and the memory cycles start again until this is done for all number of memories (***numMemories*** (line 10). The resulting array from the function ***getAvgWaitTime*** is used to plot graphs. The average wait times are plotted on the y-axis against the number of memory modules. The plots for both the workloads are shown on the pages below. The standard deviation for the Gaussian workload was kept at 20 for each processor.

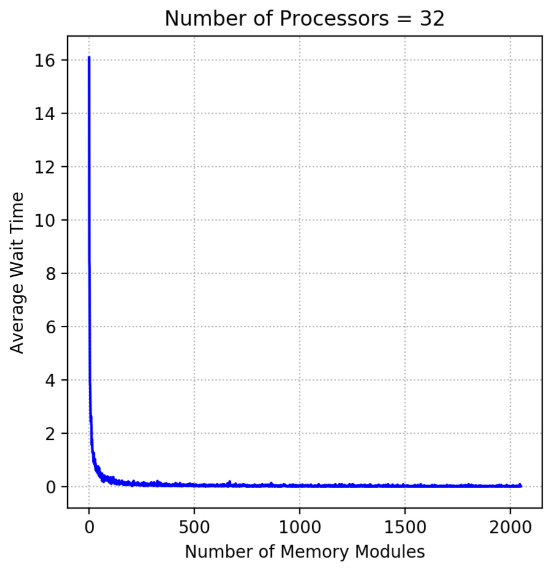
Uniform Workload Plots

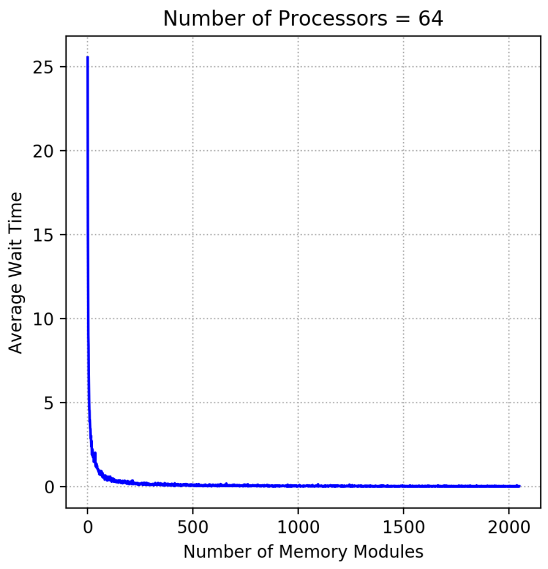




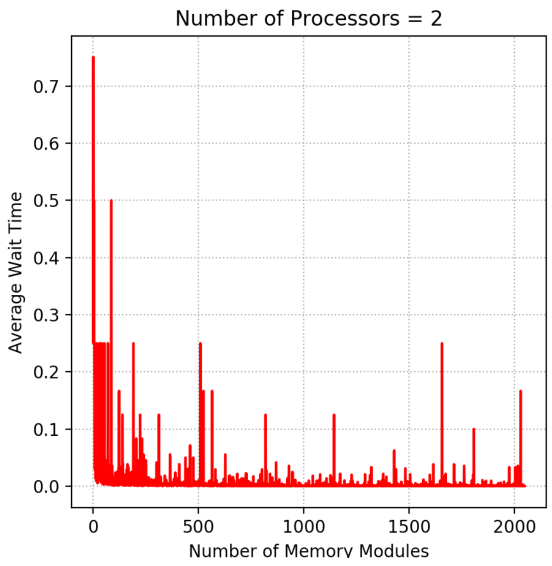


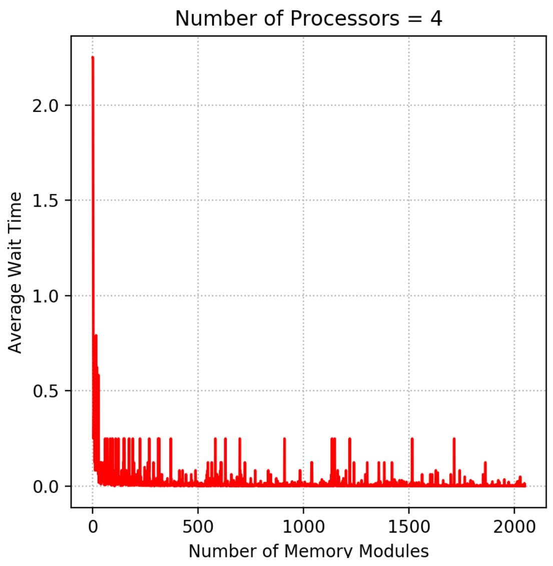


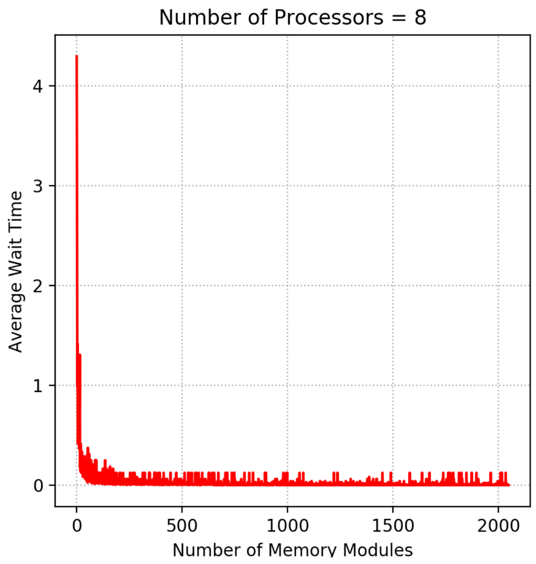


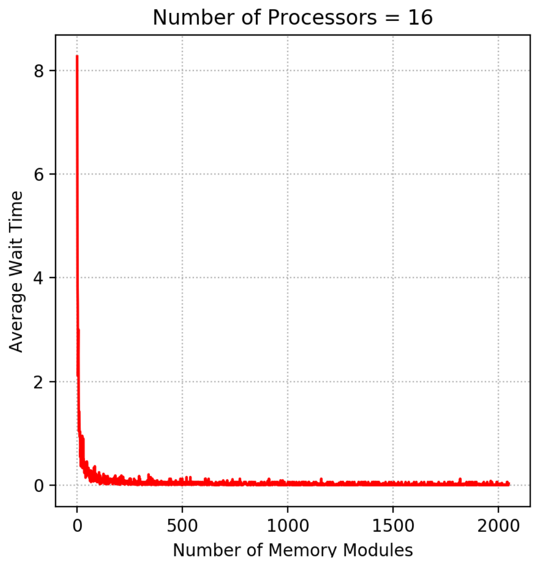


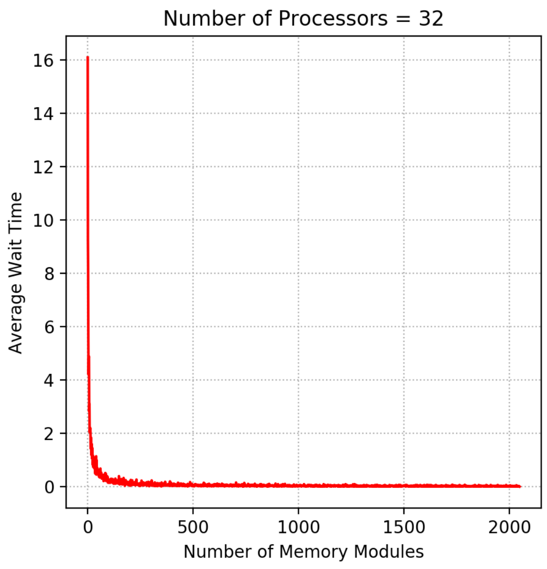
Gaussian Workload Plots

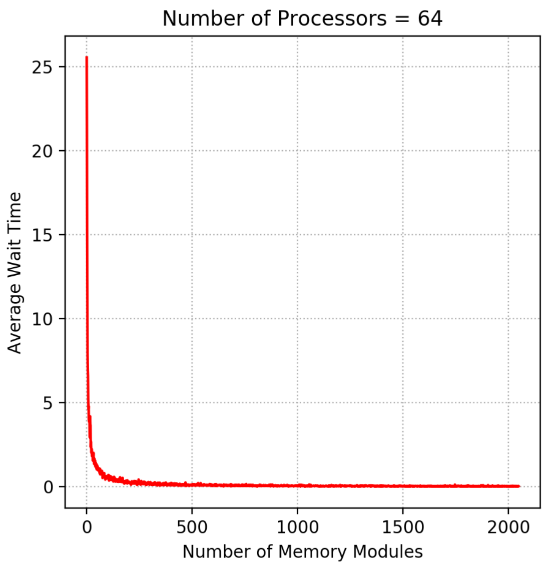




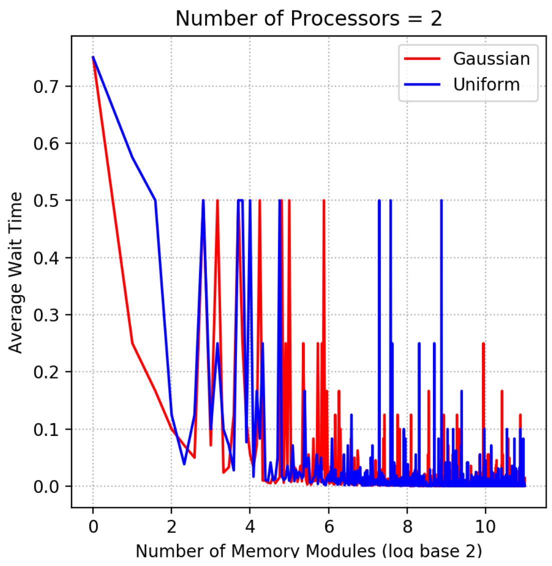


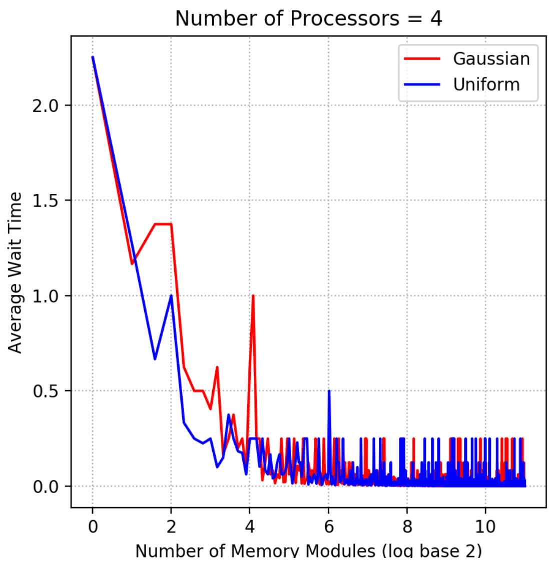


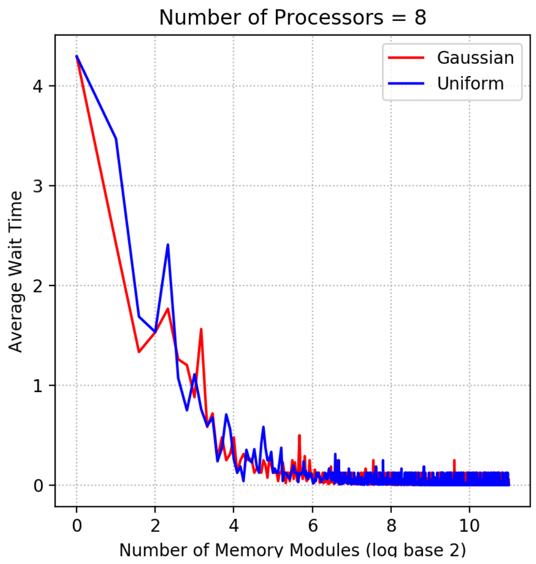


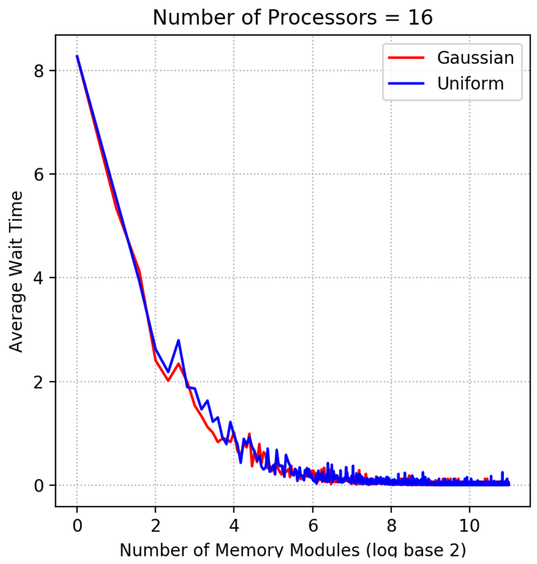


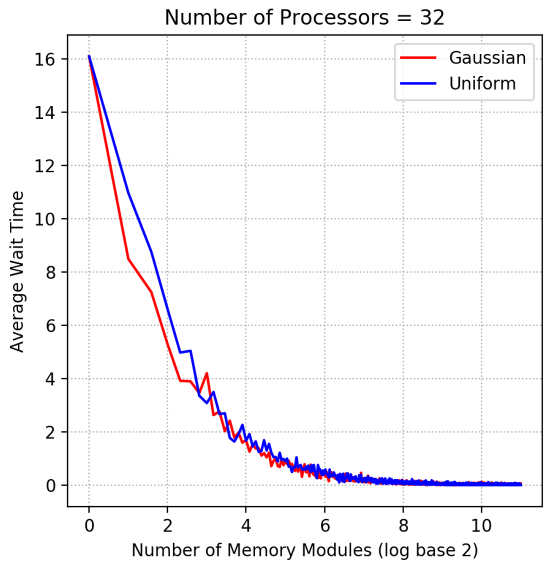
Superposed Workload Plots

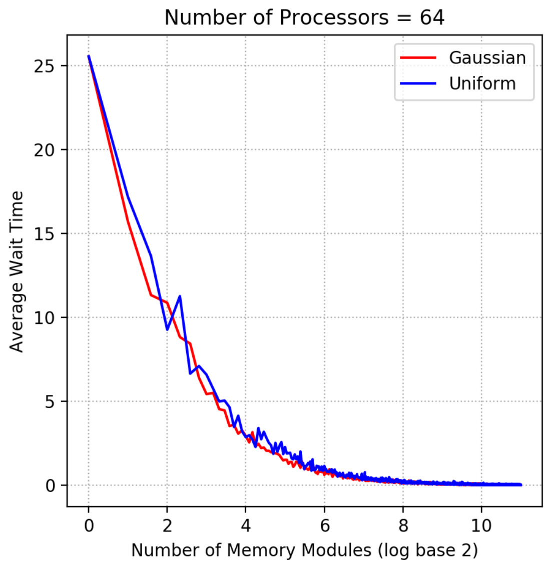












Interpretation of Results

For both the Uniform and Gaussian workload plots above, the pattern that is obvious is that if the number of processors is fixed, increasing the number of memory modules decreases the average wait time. Also from the plots it is evident that the decreasing rate falls at an exponential rate as the number of memory modules is increased. This totally makes sense because if there are more memory modules than the probability of two processors requesting the same memory module in a cycle decreases. Suppose for the Uniform distribution workload with 10 memory modules, the probability that 2 processors request the same memory module is $\frac{1}{10} * \frac{1}{10} = 0.01$ but if the number of memory modules are increased to 100 the probability that 2 processors request the same memory module is now $\frac{1}{100} * \frac{1}{100} = 0.0001$. So by increasing the number of memory modules the chance of 2 processors requesting the same memory module goes down which leads to less average wait time.

Another trend that is evident from the plots above is that, if the number of memory modules is fixed and the number of processors are varied. Then, increasing the number of processors increases the average wait time. This happens because the more the number of processors, the more the number of memory requests per cycle and hence more the chance of those requests being similar.

One result that can be extrapolated from the plots is that after a certain limit, increasing the number of memory modules does not help in decreasing the average wait time that much. This is evident from the long tail distribution of the plots. This is true because when the number of memory modules is increased the probability that each processor gets its requested memory module during a cycle reaches quite near to 1, meaning average wait time goes to 0. Hence, it is not feasible to spend more resources on using more memory modules when the average wait time is not decreasing significantly. In my opinion, the point where the elbow forms (after which the long tail starts), the number of memory modules corresponding to that elbow point should be used for the underlying system in order to utilize maximum efficiency.

By looking at the super imposed plots we can see that for most of the simulations the Gaussian workload average wait time converges faster compared to the Uniform workload average wait time. This phenomenon can be explained by spatial locality. In the Uniform workload distribution case, the processor can request any memory module with equal probability. So it can be the case than in the $Cycle_{n-1}$ the processor requests a memory module numbered 10 and in the $cycle_n$ it requests memory module 300. This can lead to disturbance within the system. In the case of the Gaussian workload, each processor has a mean value which remains unchanged until the number of memory modules are changed. In addition to this mean value, there is also a standard deviation associated with each processor, which in our case was 20. The fixed mean value and standard deviation ensures that processors generate requests for memory modules which are in the near vicinity of their mean value. E.g. if in $Cycle_{n-1}$ a processor with mean value 47 generates a request for memory module numbered 60 then in $Cycle_n$ the probability that this processor requests the memory module 310 is quite low as compared to the probability that it requests the memory module 30. In a Gaussian distribution the values which lie further away from the mean are less likely to occur. This leads to less disturbance in the system and in turn less average wait times. In other words a Gaussian distribution leads to a more ordered system.

There are some anomalies in the plots e.g. some local maxima can be observed even after convergence. In my opinion these peaks occur because of the random numbers we are generating. Sometimes for a given seed the computer tends to generate quite similar random numbers. E.g. it can be possible that cycle after cycle a bunch of processors keep asking for the same memory modules. This disturbance in randomness causes those peaks. But those peaks can be ignored.