**Basic Functions**

* We need to ask the user to input the **Memory Size** and store into an integer
* We need to ask the user to input the **Page Size** as either 1, 2 or 3 (1:100, 2:200, 3:400) which will also be the **Frame Size**.
* We can assume the memory size will always be a multiple of the page size
* We need to ask the user for the **File Name** of a workload file
  + The first line of this file will have the integer value **N** that tells us how many processes are defined in this file
  + First line is the **Unique Process ID,**
  + Second line contains **Arrival Time** which is the time it is submitted to the system.
    - Process will be listed in arrival order.
    - If two processes have the same arrival time, the process listed first in the file goes on the input queue first
  + Second line contains the **Lifetime in Memory**, the lifetime after it is admitted to the main memory.
    - How long the process will run ONCE IT HAS BEEN GIVEN SPACE IN MAIN MEMORY
    - If a process is submitted to the system at time = 100 with **Lifetime in Memory** = 2000, but it is not admitted to the memory until time = 1500, then it will complete at time = 1500 + **Lifetime in Memory** = 3500
    - The Memory Space for a process will be freed by the memory manager when it completes.
  + Third line contains the **Address Space**, the memory requirements
    - It is a sequence one or two more integers separated by a single blank.
    - The first integer gives the number of ‘pieces’ of memory on the line.
    - Simply sum these integers to get the overall space requirement
* Memory is limited. No guarantee that a process will be admitted to the memory as soon as it arrives.
  + It will have to wait until the system can accommodate its memory requirements
* Our simulator must generate output that shows the trace of important events that modify the memory contents and input queue
  + Process Arrival
  + Process admission to the main memory
  + Process completion
  + Simulation terminates when time reaches 100,000 or when there are no more process in the input queue
  + Refer to **output files** for more details

**Implementation**

* We can represent our **virtual clock** with an integer or long variable
  + We will increment its value until all the process have completed, making appropriate memory management decisions along the way as processes arrive and complete
* Each arriving process will be first put to Input Queue.
  + Process in Input Queue will be ordered according to their arrival times (FCFS ordering).
* Whenever a process completes or a new process arrives, the **Memory Manager (MM)** must be invoked
  + In case of completion, MM will first adjust the memory map to reflect the fact that the memory region(s) previously allocated to the process is/are now free.
  + After that, (and also when a new process arrives), MM will check to see if it can move the process at the head of the input queue to the memory.
    - If so, it will make the allocation and it will update the input queue.
    - Then it will check whether the head of the updated input queue **(new head)** can be also moved to the main memory, and so on.
    - Even if current commitments in memory do not allow MM to admit the process at position X of Input Queue, MM should try to load other process at positions X+1, X+2, … , and in that order.
* Throughout the project, we will assume the entire address space of a process must be brought to the main memory for execution. MM should not bring the pages of a process partially.
* Program can ignore any process whose total address space is larger than Memory Size
  + A large process may have to wait in Input Queue until sufficient space is available in memory.
* When the MM is allocating a free memory region (hole) to one process/page, it may have to choose between the lower end and the upper end of the region.
  + In this case, always use the lower end
* The turnaround time for each process will be computed as the difference between its completion time and its arrival time
* Since the focus of this project is on memory management, do not be concerned about the details of CPU scheduling or I/O device management.
  + Just assume that the process will be completed after staying in memory for **Lifetime in Memory** time units
* Feel free to adopt any convenient data structure to represent your memory map
  + Explain it in your write-up

This document contains the process that went into creating this project

To begin, we wrote down all the requirements above and used it as a checklist to make sure everything was taken care of. We then discussed the project to make sure we both had an understanding of what it was asking until we both knew exactly what was needed.  
  
We then began to discuss the data structures needed. We created a structure for MM and process and created a class that contained the actual memory manager. We then created vectors of processes, the queue, and the memory map.

From there we created the functions needed and what needed to be in them and split the work to write the pseudocode. Once it was complete, we began to write the code and split it up the same way.

We then debugged and got it working and made sure the requirements were met

**PSEUDOCODE(copied from txt file)**

**ofstream out;**

**int memSize**

**float numOfProcesses**

**int const max\_Time = 100000**

**int max\_Memory**

**int page\_Size**

**int current\_Time**

**float totalArrivalTime**

**struct MM{**

**bool free = true**

**int pID**

**int page**

**}**

**struct Process{**

**int processID**

**int arrivalTime**

**int memSize**

**float lifeTime**

**float lifeTimeInMemory**

**bool que = false**

**bool life = true**

**bool mm = false**

**}**

**class Simulator{**

**vector<Process> components**

**vector<Process> queue**

**Simulator(string file)**

**vector<MM> MM**

**void memoryManager()**

**void addMM(int pID, int pMem)**

**void removeMM( int pID, int pMem)**

**void outputMM()**

**}**

**Simulator::Simulator(string file){**

**fstream inData**

**inData.open(file)**

**int tempProcessID**

**float tempArrivalTime**

**int tempMemSize**

**float totalProcesses**

**float tempLifeTimeInMemory**

**Process temp;**

**inData >> numOfProcesses**

**while(!inData.eof()){**

**int totalMem = 0;**

**int mem = 0;**

**inData >> tempProcessID;**

**inData >> tempArrivalTime >> tempLifeTimeInMemory;**

**inData >> totalProcesses;**

**while (totalProcesses > 0){**

**inData >> mem;**

**totalMem += mem;**

**totalProcesses--;**

**}**

**tempMemSize = totalMem;**

**if(!inData.good()) {**

**//checks for end of file**

**break;**

**}**

**tempLifeTimeInMemory += tempArrivalTime;**

**temp.processID = tempProcessID;**

**temp.arrivalTime = tempArrivalTime;**

**temp.memSize = tempMemSize;**

**temp.lifeTimeInMemory = tempLifeTimeInMemory;**

**temp.lifeTime = tempLifeTimeInMemory;**

**components.push\_back(temp);**

**}**

**inData.close();**

**}**

**void Simulator::memoryManager() {**

**//Initialize basic variables**

**float it = numOfProcesses, totalTime, average**

**int current\_Memory**

**bool print = true**

**//Main Loop**

**while current\_Time is less than or equal to max\_Time and it is greater than 0**

**counter = 0**

**print = true**

**//Loop to check if a process has reached life time and removes from MM**

**for i in the range of i and components.size()**

**if components.at(i).lifeTime is equal to current\_Time and components.at)i).mm is true**

**if print is true**

**output current time process completes**

**else**

**output process completes**

**print = false**

**components.at(i).life = false**

**totalArrivalTime+= components.at(i).arrivalTime**

**totalTime+= components.at(i).lifeTime**

**current\_Memory -= components.at(i).memSize**

**removeMM(components.at(i).processID, components.at(i).memSize)**

**outputMM()**

**it--**

**if components.at(i).life is true**

**increase counter**

**it = counter**

**//Loop to check when processes arrive and adds them into the queue**

**for i in the range of i and components.size - 1**

**if components.at(i).arrivalTime equals current\_Time**

**ouput process arrivalTime**

**omponents.at(i).que = true;**

**queue.push\_back(components.at(i));**

**output input queue**

**outputMM()**

**//Statement and Loop to check if it is valid to add process from queue into MM**

**if queue.size is less than 0**

**for i in the range of i and queue.size**

**if queue.at(i).memSize + current\_Memory is less than or equal to max\_Memory and queue.at(i).que equals true and queue.at(i).arrivalTime == current\_Time**

**output that MM moves process to memory**

**addMM(queue.at(i).processID, queue.at(i).memSize);**

**current\_Memory += queue.at(i).memSize;**

**queue.at(i).mm = true;**

**for j in the range of j and components.size()**

**if queue.at(i).processID equals components.at(j).processID**

**components.at(j).mm = true**

**for i in range of i and queue.size**

**if queue.at(i).mm equals false**

**output process id of input queue**

**else**

**if queue.at(i).memSize + current\_Memory is less than max\_Memory and queue.at(i).que equals true and queue.at(i).mm equals false**

**output that MM moves process to memory**

**addMM(queue.at(i).processID, queue.at(i).memSize);**

**current\_Memory += queue.at(i).memSize;**

**queue.at(i).mm = true;**

**for j in the range of j and components.size()**

**if queue.at(i).processID equals components.at(j).processID**

**components.at(j).mm = true**

**for i in range of i and queue.size**

**if queue.at(i).mm equals false**

**output process id of input queue**

**//Statement to check if queue has been added to MM and then removes it from queue if valid**

**if queue.size is greater than 0**

**erase queue**

**current\_Time+=100**

**//Outputting total average turnaround time after loop is complete and checks whether loop exited via time or via completion**

**if it is greater than 0**

**average = (totalTime - totalArrivalTime) / (numOfProcesses - it);**

**else**

**average = (totalTime - totalArrivalTime) / numOfProcesses;**

**output average turnarount time**

**}**

**//function to add to MM**

**void Simulator::addMM(int pID, int pMem) {**

**loop = (pMem + page\_Size - 1) / page\_Size**

**page = 1**

**i = 0**

**while loop is greater than 0 and i is less max\_Memory / page\_Size**

**if MM.at(i).free is true**

**MM.at(i).free = false**

**MM.at(i).pID = pID**

**MM.at(i).page = page**

**page++**

**loop--**

**i++**

**}**

**//function is remove from MM**

**void Simulator::removeMM(int pID, int pMem) {**

**loop = (pMem + page\_Size - 1) / page\_Size**

**int i = 0**

**while loop is greater than 0 and i is less than max\_Memory / page\_Size**

**if MM.at(i).pID equals pID**

**MM.at(i).free equals true**

**loop--**

**i++**

**}**

**}**

**//function is output to file the MM**

**void Simulator:: outputMM() {**

**count = 0**

**for i in the range of i and MM.size()**

**if MM.at(i).free equals true**

**count++**

**if count equals MM.size()**

**output 0-(page\_Size \* MM.size()) - 1 Free frame(s)**

**if (count does not equal 20 and i equals MM.size() - 1**

**output (i - count + 1) \* page\_Size "-" (page\_Size \* MM.size() - 1) Free frame(s)**

**}**

**}**

**else**

**{**

**if count is greater than**

**output ((i - count) \* page\_Size) "-" ((i \* page\_Size) - 1) Free frame(s)**

**output (i \* page\_Size) "-" (((i \* page\_Size) + page\_Size) - 1) Process MM.at(i).pID Page MM.at(i).page**

**count = 0**

**}**

**else**

**{**

**(i \* page\_Size) "-" (((i \* page\_Size) + page\_Size) - 1) Process MM.at(i).pID Page MM.at(i).page**

**}**

**}**

**}**

**}**

**main() {**

**int max\_Memory = 0**

**int page\_Size = 0**

**int input = 0**

**bool loop = true**

**string file**

**get input from user**

**Simulator test(file)**

**test.max\_Memory = max\_Memory**

**test.page\_Size = page\_Size**

**test.MM.resize(max\_Memory/page\_Size)**

**test.out.open("output.txt")**

**test.memoryManager();**

**}**