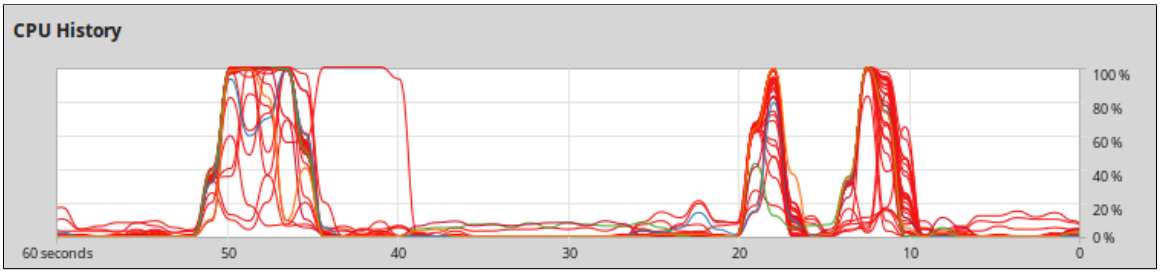
1 Overview

When you are finished submit all your work through the MyClasses page for this class. Create a directory called Project3, put each program in its own subdirectory, compress this into a single zip file, and then submit this zip file to the Project #3 assignment. Also remember to create make files for each program and make sure the programs compile and run on the lab Linux system.

Projects are to be done strictly on your own and as with all assignments the sharing of files and code is strictly prohibited and constitutes an act of Academic Misconduct. Furthermore the use of any electronic medium, such as code repositories, forums, blogs, message boards, email, etc. is strictly prohibited and constitutes an act of Academic Misconduct.

2 Introduction

The objective of this project is to use a templated priority queue class that uses the STL vector as the base storage structure, to simulate a multi-core CPU to investigate the optimal load of the CPU. Most operating systems have a nice graphical user interface that allows the user to visualize the resources usage of their machine. Below is a graph of the CPU usage during a one minute of work on the machine in my office. You will not be creating a visual representation like this but you will be simulating the load and activity of the processors numerically as well as the total load of the machine.



Each line on the graph represents a core of the processor. So it is easy to see that around 50, 18, and 13 seconds the CPU was working very hard and around 30 seconds the CPU was not working hard at all. Obviously, if you want to have your machine running efficiently you want the CPU to be working on processes. So around the 30 second mark the CPU is not processing data or running many programs, it is simply idling at very low efficiently. At the places where the cores are near 100% the CPU is very busy and processing at almost full capacity. This might not be the best scenario either. It might be that all of the processing the CPU is doing is on the higher priority processes and the lower priority processes are simply waiting, this is sometimes called process starvation, when a process is not getting enough CPU time.

3 Project: Process Simulation Programs

3.1 The Priority Queue

We are going to make one addition to our templated priority queue from the previous lab. Add in an overloaded [ ] operator. Allowing direct access to a queue structure is not standard but this will make the calculations of the final statistics of the simulation much easier.

3.2 Create a Process Class

Create the Process class. This is a class that will simulate a computer process, at least well enough for our purposes. The declaration of the class is below, you are to finish the implementation.

```
1  #ifndef PROCESS_H
2  #define PROCESS_H
3
4  #include <iostream>
5  #include <math.h>
6
7  using namespace std;
8
9  class Process {
10   private:
11     int exeTime;     // Execution time of the process.
12     int timeStamp;   // Time stamp of the process creation.
13
14   public:
15     // Constructors and Destructor
16     Process(int time = 1, int stamp = 0);
17     Process(const Process &obj);
18     ~Process();
19
20     // Acessors and Mutators
21     void setExeTime(int);
22     int getExeTime();
23     void setTimeStamp(int);
24     int getTimeStamp();
25
26     // Overloaded operator functions
27     const Process operator=(const Process &right);
28     friend ostream &operator << (ostream &, const Process &);
29 };
30
31 #endif
```

- exeTime — Represents the execution time of the process, that is the number of CPU clock cycles that must be done for the process to be completed.
- timeStamp — The time when the process was created. We will be using this to track the amount of time the process waits in the queue before it gets executed.
- setExeTime(int) — Sets the execution time of the process. This value must be at least one clock cycle (i.e. a value of 1 or higher).

- getExeTime() — Returns the execution time of the process.
- setTimeStamp(int) — Sets the time stamp of the process.
- getTimeStamp() — Gets the time stamp of the process.
- operator= — Overloaded assignment operator.
- operator << — Overloaded ostream operator. The output should look like `[7, 123]` where the first value is the execution time and the second is the time stamp.

Test this with the priority queue you created to make sure that all functions work correctly between the two classes.

3.3 Create a Process Simulation

Once both of the priority queue and process classes are complete we can create our simulation. Here is a description of the simulation and below are two runs of the simulation program.

The user will input the following information into the program. The number of CPU cores is the number of parallel execution processors of the machine. The minimum and maximum execution cycles per process is the range of how many cycles the process will take to execute. So with the settings below, 5 and 20, when a new process is created it will be given a random number between 5 and 20 for its execution time. The number of priority levels is simply the number of different priorities. So with the setting of 3, a new process will be given a random priority of 1, 2, or 3. The number of cycles for the simulation is the number of clock cycles being used in total. We will usually keep this around 100,000 to 1,000,000. The number of new processes per process is a little more difficult to explain. Note that it can be a decimal number. If this were set to 2, then during each clock cycle there would be two more processes added to the queue. With a value of 0.1, as in the example below, there would be 0.1 processes added per clock cycle. In other words there will be one process added each 10 clock cycles.

```
Input the Number of CPU Cores: 4
Input the Minimum Number of Execution Cycles per Process: 5
Input the Maximum Number of Execution Cycles per Process: 20
Input the Number of Priority Levels: 3
Input the Number of New Processes per Cycle: 0.1
Input the Length of the Simulation in Cycles: 100000
```

Before you start the simulation, you need to create the CPU. This will simply be a one-dimensional array of integers which is the size of the number of CPU cores the user selects. So in the above example it is an array of 4 cells. In a loop that runs the number of simulation clock cycles we will do the following for each clock cycle.

1. If any CPU core is ready to accept another process (i.e. the value in a cell is 0) then the next process in the priority queue will be dequeued and the execution time will be loaded into the CPU core that is ready.

2. Idle time is incremented. If any CPU core is still waiting for a process (i.e. the value in a cell is 0) then increment the idle time by 1. Note that this is per core, so if you are on a 4 core processor and three of the four cores are waiting for a process then the idle time will be incremented by a total of 3.

3. Let the CPU do one clock cycle for each of the processes currently in the CPU. That is, decrement the values of each CPU core that currently has a process being executed.

4. Add in new processes to the process priority queue. This is a little different since we could be adding several per clock cycle or we could be adding only 1 after several clock cycles. Here is a method that works for both cases, the skeleton of the code is below, you will need to finish the implementation at the comment line. Note that when you add a new process its execution time and priority are to be randomly generated within the given ranges and it is to be given the time stamp of the current cycle.

```
1 newProcessAmount += numberOfNewProcessesPerCycle;
2 while (newProcessAmount >= 1)
3 {
4     //  Add random process to the process queue.
5
6     newProcessAmount -= 1;
7 }
```

The `numberOfNewProcessesPerCycle` variable is the one that the user input and the `newProcessAmount` is a double that is tracking the accumulation of processes per cycle. To see how this works let's look at a couple examples.

First, lets say that the number per cycle is 0.25 (one every 4 cycles).

Cycle	newProcessAmount	Processes Added
1	0.25	0
2	0.5	0
3	0.75	0
4	$1 \longrightarrow 0$	1
5	0.25	0
6	0.5	0
7	0.75	0
8	$1 \longrightarrow 0$	1

First, lets say that the number per cycle is 2.5 (5 every 2 cycles).

Cycle	newProcessAmount	Processes Added
1	$2.5 \longrightarrow 0.5$	2
2	$3 \longrightarrow 0$	3
3	$2.5 \longrightarrow 0.5$	2
4	$3 \longrightarrow 0$	3

At the end of the simulation you are to report the status of the system. That is, report,

- Total idle time.
- The number of processes that were executed, that is, the number that were removed from the queue and put on the CPU.
- The total wait time of the completed processes. When a process is removed from the queue and placed on the CPU the program should subtract the time stamp from the current clock cycle value to get the number of clock cycles the process was waiting in the queue. The total wait time is the sum of all the wait times of the processes.
- The number of processes in the queue when the simulation ends.
- The total execution time of the waiting processes.
- The total wait time the queue processes were there.
- The maximum wait time of any process still in the queue.

```
============  Simulation Results  ============
Idle Time = 0
Processes Completed = 33373
Total Wait Time of Completed Processes = 8369993
Number of Processes Remaining in Queue = 66631
Total Execution Time Needed for Unprocessed Processes = 800681
Total Wait Time for Unprocessed Processes = 3321700850
Maximum Wait Time for Unprocessed Processes = 100000
```

3.3.1 Program Runs

Two program runs are below, along with an analysis of the results.

```
Input the Number of CPU Cores: 4
Input the Minimum Number of Execution Cycles per Process: 5
Input the Maximum Number of Execution Cycles per Process: 20
Input the Number of Priority Levels: 3
Input the Number of New Processes per Cycle: 0.1
Input the Length of the Simulation in Cycles: 100000

============  Simulation Results  ============
Idle Time = 279859
Processes Completed = 10003
Total Wait Time of Completed Processes = 9999
Number of Processes Remaining in Queue = 0
```

```
Input the Number of CPU Cores: 4
Input the Minimum Number of Execution Cycles per Process: 5
Input the Maximum Number of Execution Cycles per Process: 20
Input the Number of Priority Levels: 3
Input the Number of New Processes per Cycle: 1
Input the Length of the Simulation in Cycles: 100000

============  Simulation Results  ============
Idle Time = 0
Processes Completed = 33373
Total Wait Time of Completed Processes = 8369993
Number of Processes Remaining in Queue = 66631
Total Execution Time Needed for Unprocessed Processes = 800681
Total Wait Time for Unprocessed Processes = 3321700850
Maximum Wait Time for Unprocessed Processes = 100000
```

Note that in the first run the idle time is very large, only 10003 processes were completed, and the queue was empty at the end of the simulation. The number of new processes per cycle is set to 0.1, which means that a new process is added every $\frac{1}{0.1} = 10$ cycles. This indicates a lightly loaded CPU that could be using its idle time to do processes.

Note that in the second run the idle time is 0, 33373 processes were completed, and the queue had 66631 processes waiting to be processed at the end of the simulation. The number of new processes per cycle is set to 1, which means that a new process was added every clock cycle. This indicates a heavily loaded CPU that cannot keep up with the processes being added to the system. Also notice that the maximum wait time for unprocessed processes is 100000, the same length as the simulation. Hence there was at least one process that entered the system on the first clock cycle that was never processed, hence it starved.

3.4 Automate the Simulation for Different CPU Loads

After your simulation program is complete, we will use it to create a program that will investigate the simulations on different loads. The load is the number of new processes per cycle. We will use our simulation to estimate the most efficient load on a given CPU. Create a program that will take in the following information from the user.

```
Number of CPU Cores: 4
Minimum Number of Execution Cycles per Process: 5
Maximum Number of Execution Cycles per Process: 20
Number of Priority Levels: 3
Minimum Number of New Processes per Cycle: 0.25
Maximum Number of New Processes per Cycle: 0.4
New Processes per Cycle Step Size: 0.01
Length of the Simulation in Cycles: 100000
```

The meaning of the inputs are the same here as with the last program. The difference is that the user is now putting in three values for the load (Number of New Processes per Cycle), the first is the minimum, second the maximum, and third the step size. So if the user put in 0.1, 0.5, and 0.1, then the program would run simulations on loads of (0.1, 0.2, 0.3, 0.4, and 0.5). Each simulation would keep the other inputs the same. Instead of having the output as in the last program, have the

output print out in table format. Have the columns be like the example below. Note that my output does not look all that great, this is because I put the tab character between each consecutive output. This way, you will be able to copy and paste easily into a spreadsheet, as I did below.

```
============  Simulation Results  ============
Load Idle Completed Processed Wait Unprocessed Exe. Needed Unprocessed Wait Unprocessed Max. Wait
0.25 100679 25003 28175 1 5 1 1
0.26 88516 26003 31247 1 15 1 1
0.27 75560 27003 35663 1 11 1 1
0.28 63901 28003 41957 1 18 1 1
0.29 51914 29003 51305 1 10 1 1
0.3 39095 30002 67758 2 27 5 4
0.31 27580 31003 96214 1 19 1 1
0.32 16238 32003 157406 1 11 1 1
0.33 3563 32998 742179 6 83 180 58
0.34 0 33436 29383183 568 6736 1417053 4942
0.35 0 33316 71629765 1688 20395 12639180 14798
0.36 1 33409 103805061 2595 31132 28344025 21848
0.37 4 33330 129493905 3674 44008 54652081 29925
0.38 1 33268 151157348 4736 56422 87366161 36824
0.39 0 33338 159473135 5666 68219 122035900 43203
```

Load	Idle	Completed	Processed Wait	Unprocessed	Exe. Needed	Unpr. Wait	Unpr. Max. Wait
0.25	100679	25003	28175	1	5	1	1
0.26	88516	26003	31247	1	15	1	1
0.27	75560	27003	35663	1	11	1	1
0.28	63901	28003	41957	1	18	1	1
0.29	51914	29003	51305	1	10	1	1
0.3	39095	30002	67758	2	27	5	4
0.31	27580	31003	96214	1	19	1	1
0.32	16238	32003	157406	1	11	1	1
0.33	3563	32998	742179	6	83	180	58
0.34	0	33436	29383183	568	6736	1417053	4942
0.35	0	33316	71629765	1688	20395	12639180	14798
0.36	1	33409	103805061	2595	31132	28344025	21848
0.37	4	33330	129493905	3674	44008	54652081	29925
0.38	1	33268	151157348	4736	56422	87366161	36824
0.39	0	33338	159473135	5666	68219	122035900	43203

Notice that between a load of 0.33 and 0.34 there is a change in the system state. We go from having idle time to no idle time, the number of processes maxes out, the wait time makes a drastic jump, and there are far more processes left in the queue. This is indicating CPU saturation. Since the load is around 0.33, that would indicate that a load of one new process every three clock cycles is the most this system can take. Once you see where the maximum efficient load is around you should narrow the load range and use a smaller step with another simulation. For example,

Load	Idle	Completed	Processed Wait	Unprocessed	Exe. Needed	Unpr. Wait	Unpr. Max. Wait
0.32	14875	32002	174011	2	33	5	4
0.321	14864	32103	188020	0	0	0	0
0.322	14214	32203	176029	1	6	1	1
0.323	11261	32300	219383	3	23	39	19
0.324	10627	32402	254969	2	30	5	4
0.325	10539	32502	252154	2	15	11	10
0.326	9427	32601	277015	3	24	39	22
0.327	7661	32700	337962	3	43	93	52
0.328	6570	32802	369402	2	27	5	4

Load	Idle	Completed	Processed Wait	Unprocessed	Exe. Needed	Unpr. Wait	Unpr. Max. Wait
0.329	5012	32903	453473	0	0	0	0
0.33	3379	33002	874470	2	14	8	7
0.331	3732	33101	637951	2	31	17	10
0.332	712	33136	2167480	68	791	19818	642
0.333	204	33278	3476630	26	300	3545	307
0.334	188	33181	12320645	223	2645	230825	2042
0.335	236	33230	16361758	274	3232	356036	2532
0.336	1	33385	10258116	219	2556	235678	2197
0.337	108	33335	19716758	369	4309	617798	3368
0.338	38	33282	25672470	522	6279	1140334	4382
0.339	40	33402	23702000	502	5900	1155590	4425
0.34	1	33377	29101817	627	7482	1531113	4974
0.341	2	33425	32372599	679	8057	1969843	5754
0.342	0	33296	42958804	908	10817	3452157	7714
0.343	71	33325	46082013	979	11898	4175210	8572
0.344	3	33316	47546985	1088	13361	5311904	9437
0.345	0	33307	50716270	1197	14287	6237064	10311
0.346	6	33316	55386762	1288	15729	6892608	10784
0.347	1	33288	59484469	1416	17059	8751931	12381
0.348	44	33212	66217441	1592	19061	11110906	14239
0.349	0	33383	66438630	1521	18431	10005863	13086

This shows more clearly what is happening between 0.33 and 0.34.

3.5 System Analysis

Once your simulator above is working use it to analyze the following systems and draw conclusions from the data.

1. System #1: Light processes. Start with the configuration below and find the most optimal load for the system. Then increase the number of cores to 8, then 16, then 32, then 64. Find the optimal loads for each of these systems as well. Does the load scale along with the number of cores. That is, if we increase the number of cores by a factor of 2 does our load also increase by a factor of 2?

```
Number of CPU Cores: 4
Minimum Number of Execution Cycles per Process: 5
Maximum Number of Execution Cycles per Process: 20
Number of Priority Levels: 3
Length of the Simulation in Cycles: 100000
```

2. System #1: Medium processes. Start with the configuration below and find the most optimal load for the system. Then increase the number of cores to 8, then 16, then 32, then 64. Find the optimal loads for each of these systems as well. Does the load scale along with the number of cores. That is, if we increase the number of cores by a factor of 2 does our load also increase by a factor of 2?

```
Number of CPU Cores: 4
Minimum Number of Execution Cycles per Process: 20
Maximum Number of Execution Cycles per Process: 50
Number of Priority Levels: 5
Length of the Simulation in Cycles: 100000
```

3. System #3: Heavy processes. Start with the configuration below and find the most optimal load for the system. Then increase the number of cores to 8, then 16, then 32, then 64. Find the optimal loads for each of these systems as well. Does the load scale along with the number of cores. That is, if we increase the number of cores by a factor of 2 does our load also increase by a factor of 2?

```
Number of CPU Cores: 4
Minimum Number of Execution Cycles per Process: 100
Maximum Number of Execution Cycles per Process: 200
Number of Priority Levels: 5
Length of the Simulation in Cycles: 100000
```

Write a report of your findings that includes all of the data, charts, and answers to at least the questions posed above. Also, include any other observations you make with the data. Include this report in the zip file that contains all of your code for the three programs in the project. The report document should be in either Word (doc or docx) format, LibreOffice odt format or as a PDF.

4 Grading

The program itself should, of course, be nicely formatted and commented and should follow all the other rules of good programming style. Variable names should be representative of their purpose. As always, there must be our standard header comment. All variables must have a comment to their use and major blocks of code should contain brief but descriptive comments to their function. The report should be well written and formatted. Export the paper to PDF format and include it in the zip file that contains all of your code for the three programs in the project.

The grading of the project will take three forms, a sample run of all programs, an inspection of the code of all programs, and reading of the report for completeness and correct conclusions. If a program does not run you will receive a zero for that portion. So even if a program is not complete you will get a better grade for a partial program that runs verses a program that does not run. The run portion of the grading will test the user interface for usability and conforming to the specifications I have outlined above. The code inspection portion of the grade will involve commenting, readability, correct indentation, variable names, structure and style, correctness, and conforming to specifications. The report will be graded on completeness and correctness as well as readability and style.