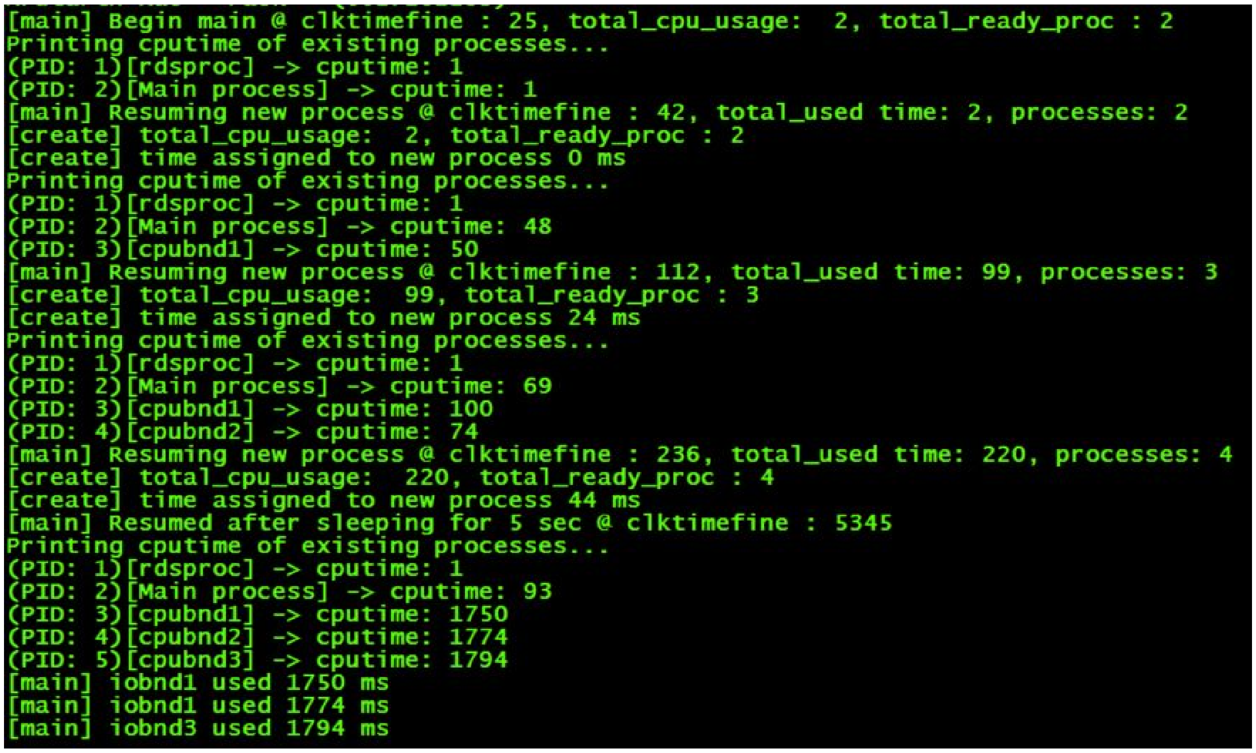
3

Handling the null process and managing average CPU time
The variable `total_cpu_usage` is incremented every time a process is added to the readylist. The implementation assigns MAXKEY as the cpu used to the null proc and never changes this assignment. Therefore the null process always stays at the end of the ready list (that is sorted by increasing cpu usage). If the current CPU usage of the running process is MAXKEY, the cpu usage is not added to the total usage. Therefore, the null process does not interfere with the scheduling and continues to run when it is the only ready process.

`Total_ready_proc` is decremented when the ready list entry is removed.

In the evaluation output below, the lines beginning with **[create]** print the cpu usage time that is assigned to the newly created process. This number is subtracted from the cpuusage field in the process table entry (when it is printed for a process being tested) to get the real amount of time that the process obtained the cpu.

CPU bound processes same time
When CPU bound processes are created at the same time, they receive equal share of the CPU. Reading the lines beginning with **[create]**, we can see the time that was assigned to the new process and subtracting that from the final time used gives us the real time each process got to run. For this example (see below), *cpubnd1* got (1750 - 0) 1750 ms, *cpubnd2* got (1774 - 24) 1750 ms and *cpubnd3* got (1794 - 44) 1750 ms. Showing that cpu bound processes created at the same time all receive equal share of the cpu.

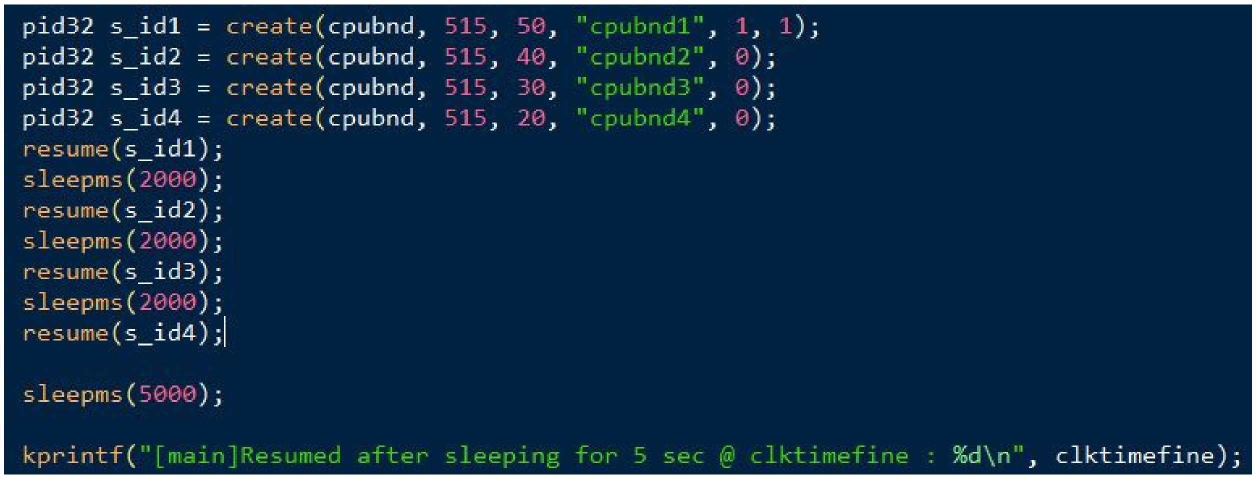


```
[main] Begin main @ clktimefine : 25, total_cpu_usage:  2, total_ready_proc : 2
Printing cputime of existing processes...
(PID: 1)[rdsproc] -> cputime: 1
(PID: 2)[Main process] -> cputime: 1
[main] Resuming new process @ clktimefine : 42, total_used time: 2, processes: 2
[create] total_cpu_usage:  2, total_ready_proc : 2
[create] time assigned to new process 0 ms
Printing cputime of existing processes...
(PID: 1)[rdsproc] -> cputime: 1
(PID: 2)[Main process] -> cputime: 48
(PID: 3)[cpubnd1] -> cputime: 50
[main] Resuming new process @ clktimefine : 112, total_used time: 99, processes: 3
[create] total_cpu_usage:  99, total_ready_proc : 3
[create] time assigned to new process 24 ms
Printing cputime of existing processes...
(PID: 1)[rdsproc] -> cputime: 1
(PID: 2)[Main process] -> cputime: 69
(PID: 3)[cpubnd1] -> cputime: 100
(PID: 4)[cpubnd2] -> cputime: 74
[main] Resuming new process @ clktimefine : 236, total_used time: 220, processes: 4
[create] total_cpu_usage:  220, total_ready_proc : 4
[create] time assigned to new process 44 ms
[main] Resumed after sleeping for 5 sec @ clktimefine : 5345
Printing cputime of existing processes...
(PID: 1)[rdsproc] -> cputime: 1
(PID: 2)[Main process] -> cputime: 93
(PID: 3)[cpubnd1] -> cputime: 1750
(PID: 4)[cpubnd2] -> cputime: 1774
(PID: 5)[cpubnd3] -> cputime: 1794
[main] iobnd1 used 1750 ms
[main] iobnd1 used 1774 ms
[main] iobnd3 used 1794 ms
```

(NOTE: last 3 print statements should read …cpubnd…)

Variable time

Processes resumed after intervals from main were tested for fair scheduling using the code below and the final prcpuused fields were printed.

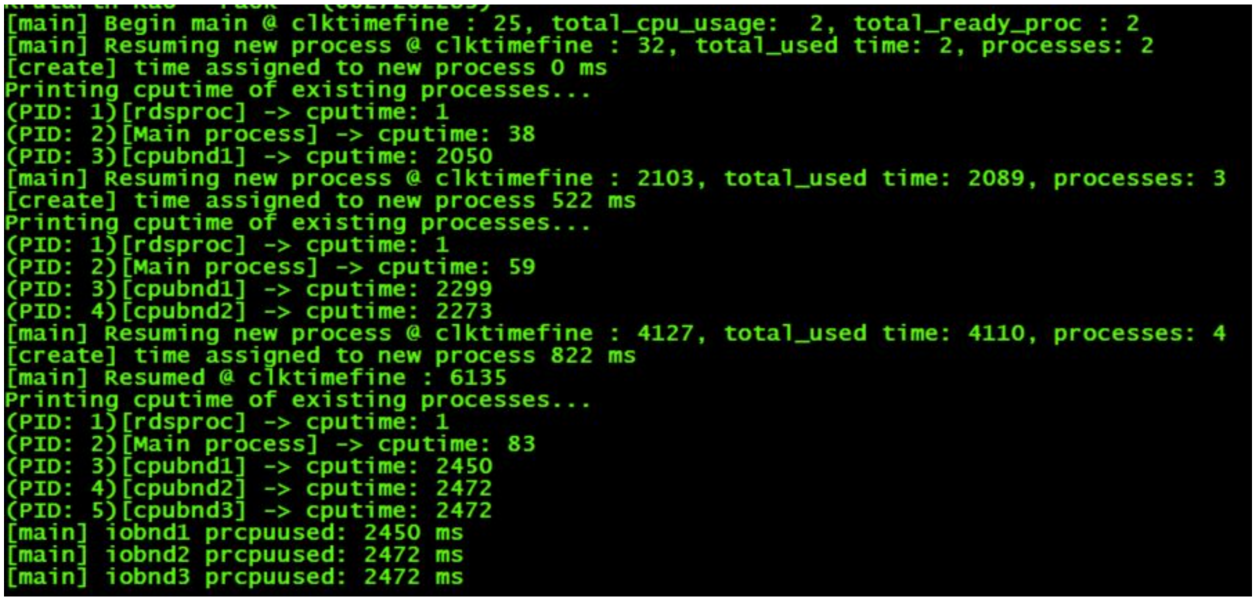


```
pid32 s_id1 = create(cpubnd, 515, 50, "cpubnd1", 1, 1);
pid32 s_id2 = create(cpubnd, 515, 40, "cpubnd2", 0);
pid32 s_id3 = create(cpubnd, 515, 30, "cpubnd3", 0);
pid32 s_id4 = create(cpubnd, 515, 20, "cpubnd4", 0);
resume(s_id1);
sleepms(2000);
resume(s_id2);
sleepms(2000);
resume(s_id3);
sleepms(2000);
resume(s_id4);

sleepms(5000);

kprintf("[main]Resumed after sleeping for 5 sec @ clktimefine : %d\n", clktimefine);
```

(Note: cpubnd4 was later removed during testing for simpler output)



```
[main] Begin main @ clktimefine : 25, total_cpu_usage:  2, total_ready_proc : 2
[main] Resuming new process @ clktimefine : 32, total_used time: 2, processes: 2
[create] time assigned to new process 0 ms
Printing cputime of existing processes...
(PID: 1)[rdsproc] -> cputime: 1
(PID: 2)[Main process] -> cputime: 38
(PID: 3)[cpubnd1] -> cputime: 2050
[main] Resuming new process @ clktimefine : 2103, total_used time: 2089, processes: 3
[create] time assigned to new process 522 ms
Printing cputime of existing processes...
(PID: 1)[rdsproc] -> cputime: 1
(PID: 2)[Main process] -> cputime: 59
(PID: 3)[cpubnd1] -> cputime: 2299
(PID: 4)[cpubnd2] -> cputime: 2273
[main] Resuming new process @ clktimefine : 4127, total_used time: 4110, processes: 4
[create] time assigned to new process 822 ms
[main] Resumed @ clktimefine : 6135
Printing cputime of existing processes...
(PID: 1)[rdsproc] -> cputime: 1
(PID: 2)[Main process] -> cputime: 83
(PID: 3)[cpubnd1] -> cputime: 2450
(PID: 4)[cpubnd2] -> cputime: 2472
(PID: 5)[cpubnd3] -> cputime: 2472
[main] iobnd1 prcpuused: 2450 ms
[main] iobnd2 prcpuused: 2472 ms
[main] iobnd3 prcpuused: 2472 ms
```

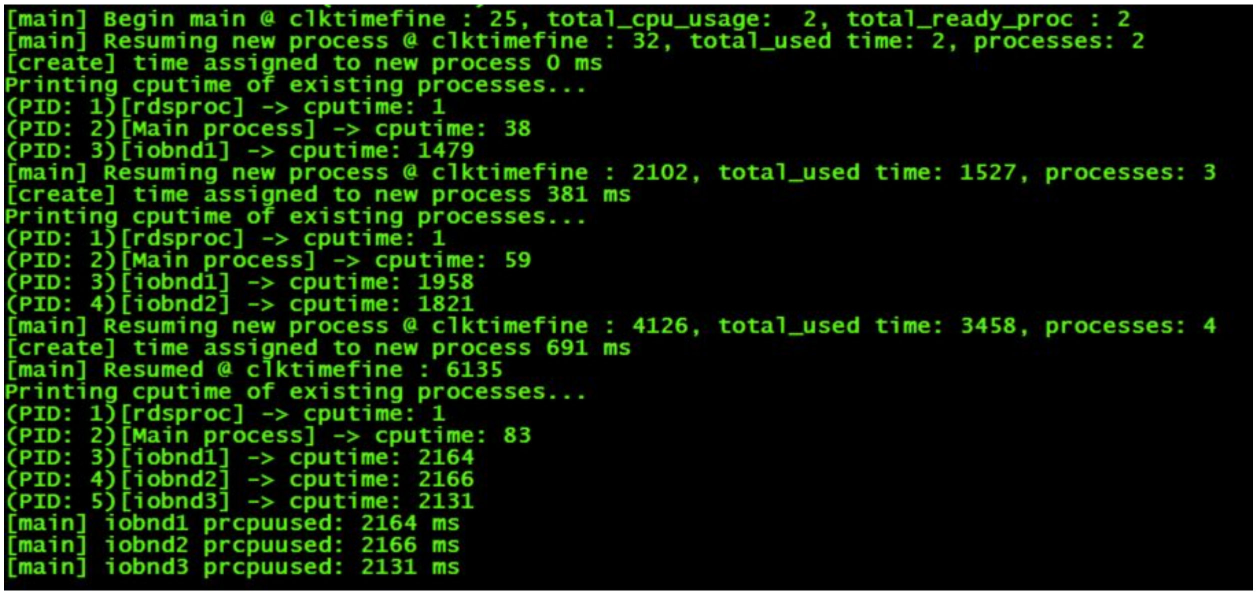
cpubnd1 got (2450 - 0) 2450 ms, *cpubnd2* got (2472 - 522) 1950 ms and *cpubnd3* got (2472 - 822) 1650 ms. At the same time, it can be seen that older process continue to receive cpu time in between and are therefore not completely neglected.

We can also see that the prcpuused fields for all cpu bound processes are becoming similar towards the end and therefore, newer process will be treated with equal/similar priority (to older processes) in the future. Hereby achieving fair scheduling.

Evaluation with CPU bound
When the time usage of processes is printed between process creation (in the output above), it is possible to verify that new processes are getting the correct initial cpu time used and that all processes eventually receive equal share of the cpu.

Lines beginning with **[create]** give the prcpuused time assigned to the new process and it is easy to verify with a simple calculation that the numbers are correct.

Evaluation with I/O bound
Similar results are seen when I/O bound processes are tested using the above strategy.



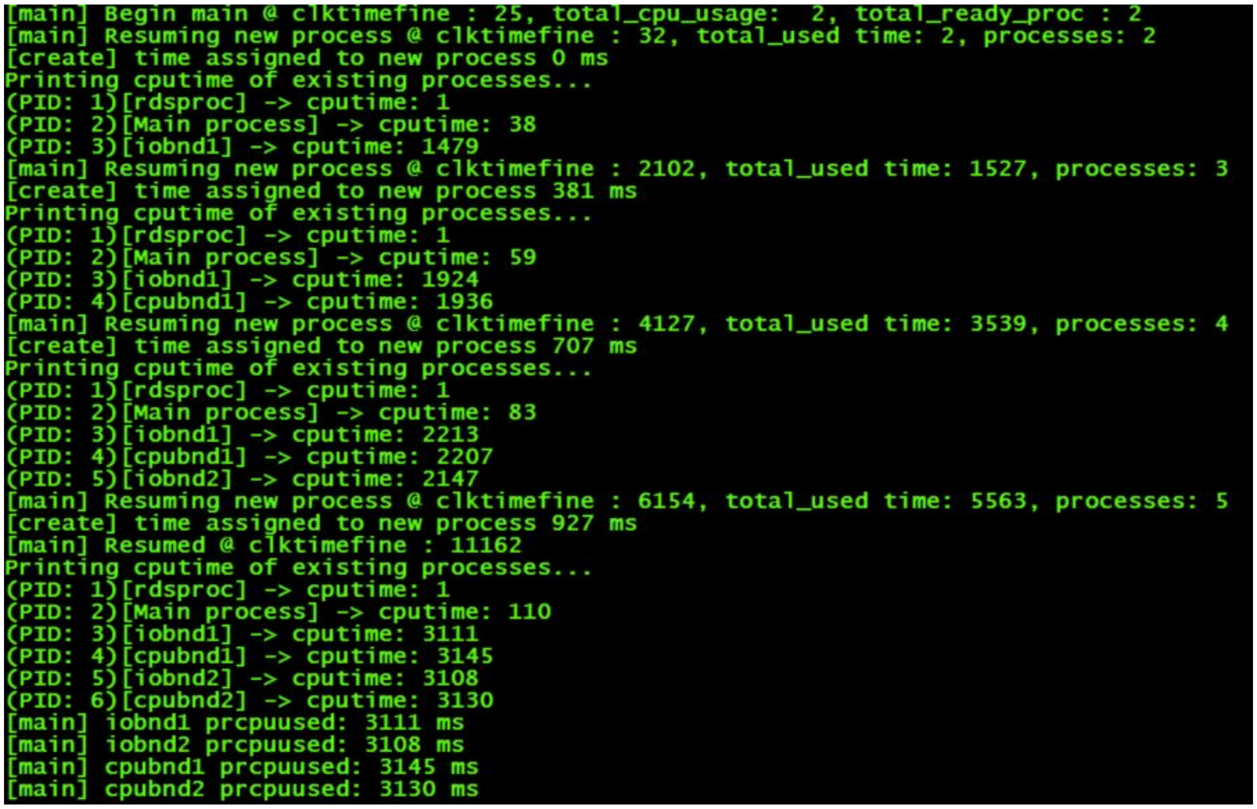
```
[main] Begin main @ clktimefine : 25, total_cpu_usage:  2, total_ready_proc : 2
[main] Resuming new process @ clktimefine : 32, total_used time: 2, processes: 2
[create] time assigned to new process 0 ms
Printing cputime of existing processes...
(PID: 1)[rdsproc] -> cputime: 1
(PID: 2)[Main process] -> cputime: 38
(PID: 3)[iobnd1] -> cputime: 1479
[main] Resuming new process @ clktimefine : 2102, total_used time: 1527, processes: 3
[create] time assigned to new process 381 ms
Printing cputime of existing processes...
(PID: 1)[rdsproc] -> cputime: 1
(PID: 2)[Main process] -> cputime: 59
(PID: 3)[iobnd1] -> cputime: 1958
(PID: 4)[iobnd2] -> cputime: 1821
[main] Resuming new process @ clktimefine : 4126, total_used time: 3458, processes: 4
[create] time assigned to new process 691 ms
[main] Resumed @ clktimefine : 6135
Printing cputime of existing processes...
(PID: 1)[rdsproc] -> cputime: 1
(PID: 2)[Main process] -> cputime: 83
(PID: 3)[iobnd1] -> cputime: 2164
(PID: 4)[iobnd2] -> cputime: 2166
(PID: 5)[iobnd3] -> cputime: 2131
[main] iobnd1 prcpuused: 2164 ms
[main] iobnd2 prcpuused: 2166 ms
[main] iobnd3 prcpuused: 2131 ms
```

The processes ultimately end up with approximately equal values in prcpuused fields and are assigned the correct values when create gives them the average of the prcpuused of all processes.

4

See turnin folder for min-heap implementation. Test method can be found in **test_hybrid()** in *main.c*. The test for the min-heap data structure can be found in **test_minheap()** in main.c.

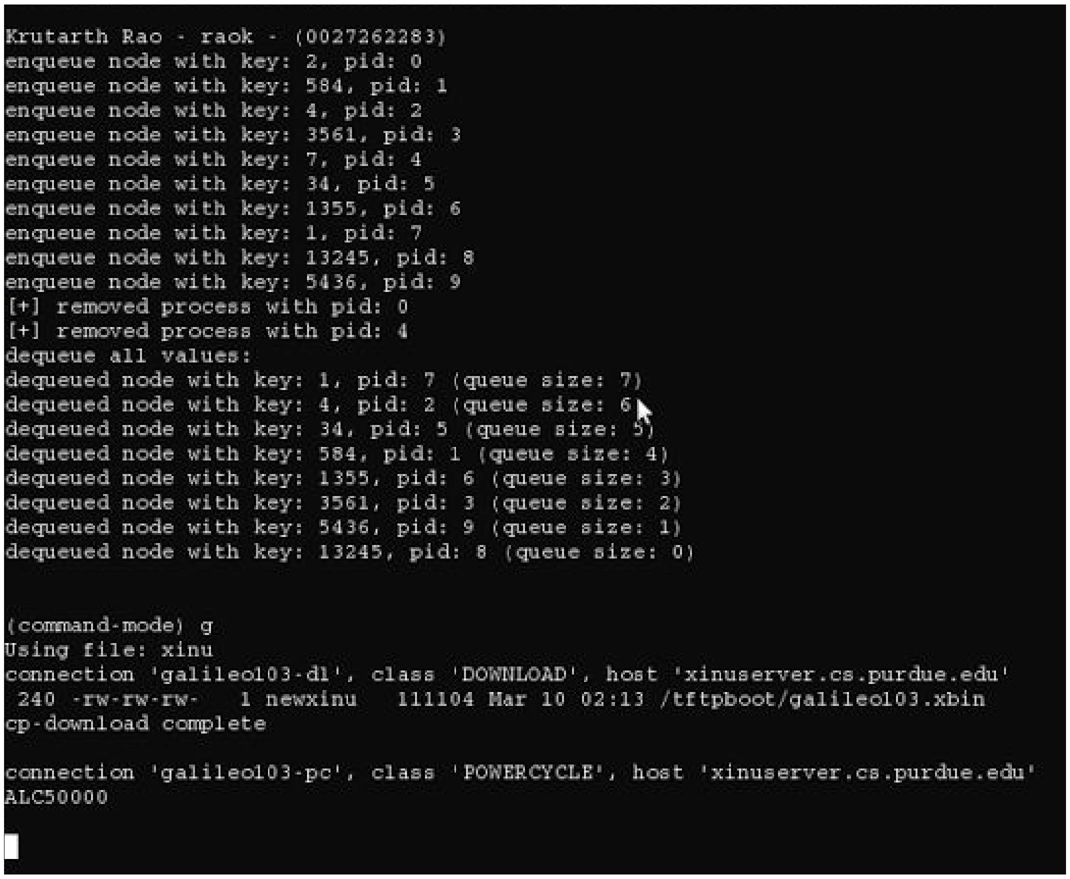
Testing the new data structure with hybrid processes and time varying workload and printing the intermediate stage information as in **3**, the output below is observed.



```
[main] Begin main @ clktimefine : 25, total_cpu_usage:  2, total_ready_proc : 2
[main] Resuming new process @ clktimefine : 32, total_used time: 2, processes: 2
[create] time assigned to new process 0 ms
Printing cputime of existing processes...
(PID: 1)[rdsproc] -> cputime: 1
(PID: 2)[Main process] -> cputime: 38
(PID: 3)[iobnd1] -> cputime: 1479
[main] Resuming new process @ clktimefine : 2102, total_used time: 1527, processes: 3
[create] time assigned to new process 381 ms
Printing cputime of existing processes...
(PID: 1)[rdsproc] -> cputime: 1
(PID: 2)[Main process] -> cputime: 59
(PID: 3)[iobnd1] -> cputime: 1924
(PID: 4)[cpubnd1] -> cputime: 1936
[main] Resuming new process @ clktimefine : 4127, total_used time: 3539, processes: 4
[create] time assigned to new process 707 ms
Printing cputime of existing processes...
(PID: 1)[rdsproc] -> cputime: 1
(PID: 2)[Main process] -> cputime: 83
(PID: 3)[iobnd1] -> cputime: 2213
(PID: 4)[cpubnd1] -> cputime: 2207
(PID: 5)[iobnd2] -> cputime: 2147
[main] Resuming new process @ clktimefine : 6154, total_used time: 5563, processes: 5
[create] time assigned to new process 927 ms
[main] Resumed @ clktimefine : 11162
Printing cputime of existing processes...
(PID: 1)[rdsproc] -> cputime: 1
(PID: 2)[Main process] -> cputime: 110
(PID: 3)[iobnd1] -> cputime: 3111
(PID: 4)[cpubnd1] -> cputime: 3145
(PID: 5)[iobnd2] -> cputime: 3108
(PID: 6)[cpubnd2] -> cputime: 3130
[main] iobnd1 prcpuused: 3111 ms
[main] iobnd2 prcpuused: 3108 ms
[main] cpubnd1 prcpuused: 3145 ms
[main] cpubnd2 prcpuused: 3130 ms
```

Four processes (2 IO bound and 2 CPU bound) are created at varying times (interval of 2sec). The prcpuused used tends towards similar values irrespective of the time the processes were created or the type of the process. The initial value of the prcpuused time given to a process by create() is also correct in the output above and therefore the fair scheduling continues to function effectively.

The min-heap implementation is tested as follows:



```
Krutarth Rao - raok - (0027262283)
enqueue node with key: 2, pid: 0
enqueue node with key: 584, pid: 1
enqueue node with key: 4, pid: 2
enqueue node with key: 3561, pid: 3
enqueue node with key: 7, pid: 4
enqueue node with key: 34, pid: 5
enqueue node with key: 1355, pid: 6
enqueue node with key: 1, pid: 7
enqueue node with key: 13245, pid: 8
enqueue node with key: 5436, pid: 9
[+] removed process with pid: 0
[+] removed process with pid: 4
dequeue all values:
dequeued node with key: 1, pid: 7 (queue size: 7)
dequeued node with key: 4, pid: 2 (queue size: 6)
dequeued node with key: 34, pid: 5 (queue size: 5)
dequeued node with key: 584, pid: 1 (queue size: 4)
dequeued node with key: 1355, pid: 6 (queue size: 3)
dequeued node with key: 3561, pid: 3 (queue size: 2)
dequeued node with key: 5436, pid: 9 (queue size: 1)
dequeued node with key: 13245, pid: 8 (queue size: 0)


(command-mode) g
Using file: xinu
connection 'galileo103-dl', class 'DOWNLOAD', host 'xinuserver.cs.purdue.edu'
 240 -rw-rw-rw-   1 newxinu   111104 Mar 10 02:13 /tftpboot/galileo103.xbin
cp-download complete

connection 'galileo103-pc', class 'POWERCYCLE', host 'xinuserver.cs.purdue.edu'
ALC50000
```

The ready list continues to be a queue like data structure with enqueue and dequeue methods but the internal nodes are maintained as an array based heap. Enqueue inserts the new element in the heap and dequeue runs a **getMin** operation on the internal heap.
(NOTE: The heap being manipulated above is not the ready list)

Bonus

Red-Black Tree over Heap

- Searching in a heap is still linear but logarithmic in a balanced tree. Therefore, removing random processes from the read list still takes linear time with a heap and can be faster with a balanced tree.

Heap over Red-Black Tree

- Heaps can be array based vs trees that are pointer based. Memory access is more efficient in array based data structures due to caching. Therefore, heaps can be aser in some cases where memory access is required.
- Building heaps requires linear time but building a tree requires nlog(n) time. This means that “maintenance” of the data structure (and therefore the ready list) is more in efficient with balanced trees.

- The constants involved in balanced tree insert/delete algorithms are larger than in heaps making trees inefficient for small number of processes.