Lab 3

3.1

To gauge the correctness of my implementation, I created a function, *donothing()*, that has a for-loop performing a simple arithmetic operation and loops for 100,000 iterations. After the loop ends, it prints the CPU usage of the executing process.

I first called *donothing()* from *main()* and then created and readied two processes, A and B, executing *donothing()* with different priorities, one the same as *main()*'s and one slightly higher. As expected, all three processes printed approximately the same CPU usage after the loop in *donothing()* ended. This shows that CPU usage for each process is recorded separately and not all together. Additionally, I printed A's CPU usage after it's been context-switched out back to *main()* and it showed correct behavior. Furthermore, as process B had a higher priority than process A, it terminates the loop and prints its CPU usage in *donothing()* before process A prints its CPU usage.

To further test that CPU usage is being recorded accordingly, I increased the bounds of the for-loop in *donothing()* to 500,000 iterations. As expected, the CPU usage of all processes executing the function increased too.

The “volatile” qualifier is used to tell the compiler that the value of the variable may change at any time without any action being taken by the code the compiler finds nearby. This is often the case when multiple threads are accessing a variable or when it represents hardware that is external to the computer. Since we are not working with concurrency or external hardware in this lab, the “volatile” qualifier is not necessary.

3.2

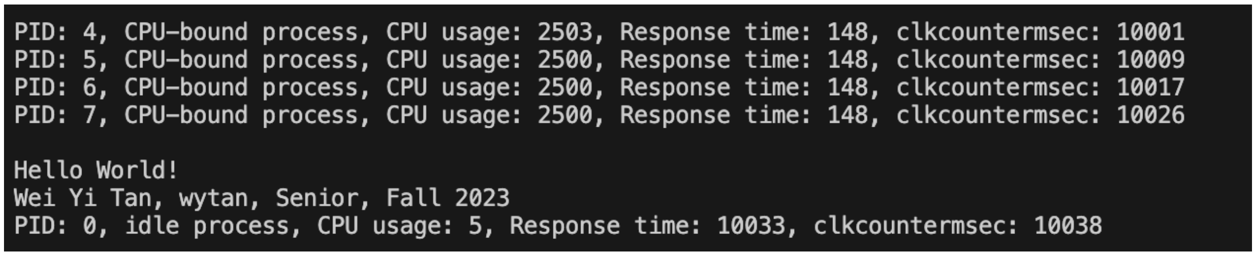
To assess the correctness of my implementation, I created 3 different processes to execute *donothing()* and of priorities higher than *main()*'s. After each process was created and readied, I called *avgresponse()* on *main()*'s PID to check its response time. I also added a print statement in *avgresponse()* to print the process's *prtotresp* and *prtotready*, since there might be cases where *main()* was previously in the readylist before I created the processes.

As expected, *main()*'s total response time increased approximately by the CPU usage time of each executing process for *donothing()*. *main()*'s *prtotready* also incremented each time a new process was created, which is correct. Then, by printing both *prtotresp* and *prtotready*, I was also able to check that the output of *avgresponse()* was correctly rounded up. I also checked that the response time of a process in PR_CURR state does not increase by printing the response time before and after the for-loop in *donothing()*.

As an extra precaution, I created 3 more processes to execute *donothing()*, each of different priorities but all lower than *main()*'s. I then added a print statement in *donothing()* to print the executing process's response time. As expected, the process with the lowest priority has the longest response time and the process with the highest priority has the shortest response time.

When a newly created process becomes ready for the first time, the above method for calculating response time is inaccurate. This is because when the process becomes ready for the first time, and assuming it has the highest priority of all eligible processes, it is immediately context-switched in to be the current process in *resched()*. However, when a process becomes current, we subtract *prreadystart* from *clkcountermsec* and set the difference to 1 (msec) if it is 0. This overestimates the response time since the process does not actually spend 1 msec in the readylist as it is immediately being context-switched in. I have also tested that the response time is 1 by printing the response time immediately after *donothing()* starts executing for a process with higher priority than *main()*.

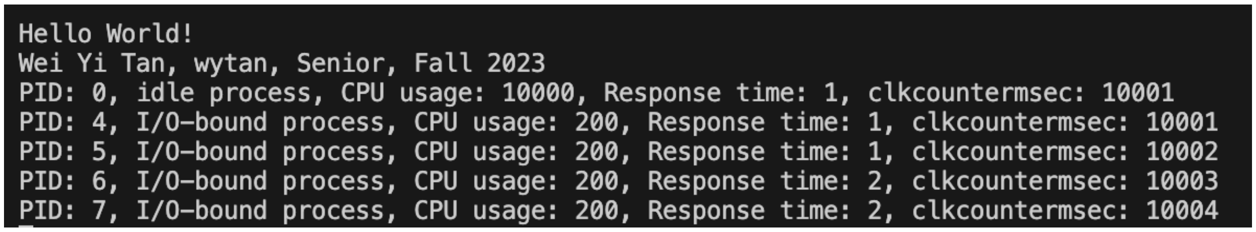
4.3
Benchmark A


```
PID: 4, CPU-bound process, CPU usage: 2503, Response time: 148, clkcountermsec: 10001
PID: 5, CPU-bound process, CPU usage: 2500, Response time: 148, clkcountermsec: 10009
PID: 6, CPU-bound process, CPU usage: 2500, Response time: 148, clkcountermsec: 10017
PID: 7, CPU-bound process, CPU usage: 2500, Response time: 148, clkcountermsec: 10026

Hello World!
Wei Yi Tan, wytan, Senior, Fall 2023
PID: 0, idle process, CPU usage: 5, Response time: 10033, clkcountermsec: 10038
```

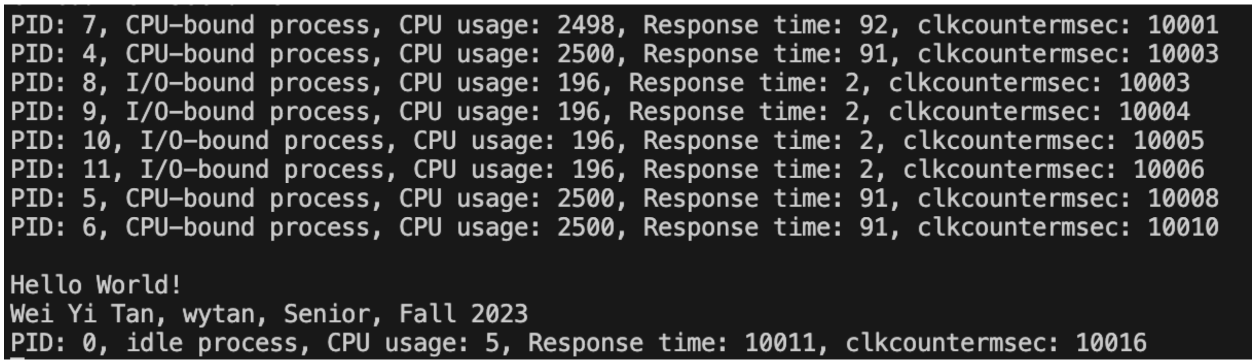
Each CPU-bound process has a CPU usage time of approximately 2500 milliseconds and an average response time of 148 milliseconds. This shows that there is fair allocation of CPU cycles to each CPU-bound process. The total sum of all CPU usage times is approximately 10 seconds, which is correct since it's approximately equal to *clkcountermsec* after all processes are done executing. Each process has a higher CPU usage than the idle process because it has a higher priority so the idle process rarely excutes. As a result of this, the idle process has a very high average response time because it keeps staying in the readylist. Additionally, each CPU-bound process's average response time is similar because they all enter the ready list at approximately equal amounts and stay there for approximately the same amount of time. Each CPU-bound process' total response time are similar because they'll always wait for 3 other CPU-bound process to deplete its time slice every time they enter the readylist due to round-robin scheduling.

Benchmark B


```
Hello World!
Wei Yi Tan, wytan, Senior, Fall 2023
PID: 0, idle process, CPU usage: 10000, Response time: 1, clkcountermsec: 10001
PID: 4, I/O-bound process, CPU usage: 200, Response time: 1, clkcountermsec: 10001
PID: 5, I/O-bound process, CPU usage: 200, Response time: 1, clkcountermsec: 10002
PID: 6, I/O-bound process, CPU usage: 200, Response time: 2, clkcountermsec: 10003
PID: 7, I/O-bound process, CPU usage: 200, Response time: 2, clkcountermsec: 10004
```

Each I/O-bound process has a CPU usage time of approximately 200 milliseconds and an approximate average response time of 2 milliseconds. This shows that there is fair allocation of CPU cycles to each I/O-bound process. The I/O-bound processes' low CPU usage time could be because of its smaller time slice compared to the time slice allocated for CPU-bound process. Similarly, the I/O-bound processes' low response time could be because it keeps entering the readylist after it is woken up from the *sleepms()* call. Since, we're only printing the average response time, the total response time and the number of time it enters the readylist might be equally high. On the other hand, the idle process has a CPU usage of 10 seconds and an average response time of 1 millisecond. This could be because when all IO-bound processes are in the sleep queue, the idle process is context-switched in and starts executing. Additionally, the idle process has a larger time slice compared to the I/O-bound process, further explaining its higher CPU usage time.

Benchmark C

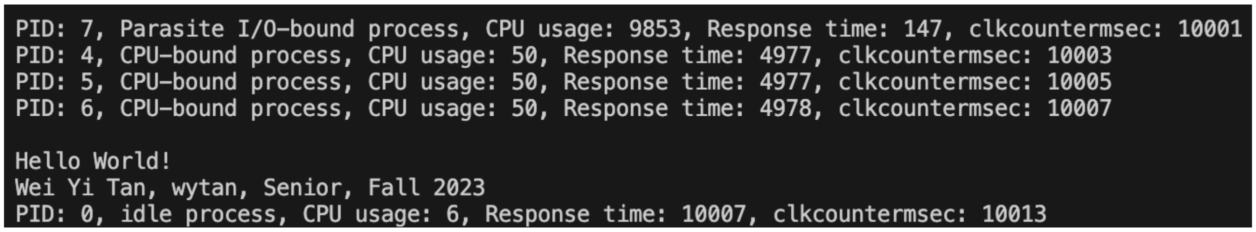


```
PID: 7, CPU-bound process, CPU usage: 2498, Response time: 92, clkcountermsec: 10001
PID: 4, CPU-bound process, CPU usage: 2500, Response time: 91, clkcountermsec: 10003
PID: 8, I/O-bound process, CPU usage: 196, Response time: 2, clkcountermsec: 10003
PID: 9, I/O-bound process, CPU usage: 196, Response time: 2, clkcountermsec: 10004
PID: 10, I/O-bound process, CPU usage: 196, Response time: 2, clkcountermsec: 10005
PID: 11, I/O-bound process, CPU usage: 196, Response time: 2, clkcountermsec: 10006
PID: 5, CPU-bound process, CPU usage: 2500, Response time: 91, clkcountermsec: 10008
PID: 6, CPU-bound process, CPU usage: 2500, Response time: 91, clkcountermsec: 10010

Hello World!
Wei Yi Tan, wytan, Senior, Fall 2023
PID: 0, idle process, CPU usage: 5, Response time: 10011, clkcountermsec: 10016
```

All CPU-bound processes has a CPU usage time of roughly 2500 milliseconds and an average response time of 91 milliseconds, whereas all I/O-bound processes has a CPU usage time of 196 milliseconds and an average response time of 2 milliseconds. This seems correct since CPU-bound processes has a larger time slice so they would have a larger share of the CPU cycles among themselves. Likewise, I/O-bound processes has a smaller time slice so they have lower CPU usage times. As CPU-bound processes have lower priorities, they spend more time in the readylist, which explains the higher average response time. Similarly, I/O-bound processes has a higher priority, so they spend less time in the readylist and have a lower average response time. Lastly, each type of process shows similar behavior because they are assigned the same priority, either in *sleepms()* or *clkhandler()*, and are allocated the same time.

Benchmark D

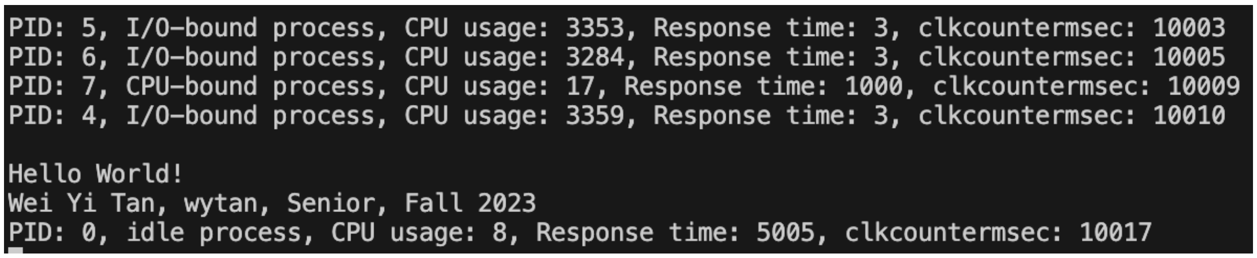


```
PID: 7, Parasite I/O-bound process, CPU usage: 9853, Response time: 147, clkcountermsec: 10001
PID: 4, CPU-bound process, CPU usage: 50, Response time: 4977, clkcountermsec: 10003
PID: 5, CPU-bound process, CPU usage: 50, Response time: 4977, clkcountermsec: 10005
PID: 6, CPU-bound process, CPU usage: 50, Response time: 4978, clkcountermsec: 10007

Hello World!
Wei Yi Tan, wytan, Senior, Fall 2023
PID: 0, idle process, CPU usage: 6, Response time: 10007, clkcountermsec: 10013
```

The parasite process has a CPU usage time of 9853 milliseconds and an average response time of 147 milliseconds, while all CPU-bound processes has a CPU usage time of 50 milliseconds and an average response time of 4977 milliseconds. This is because the parasite process tricks the kernel into thinking it is I/O-bound by calling *sleepms()*. By doing so, it will be assigned a higher priority than the CPU-bound process. Despite the parasite process being allocated a smaller time slice, it still gets the most CPU time because it has a higher priority and will always preempt any CPU-bound process that is executing. Similarly, it has a lower average response time because it's only in the readylist briefly before it becomes the executing process again. This also explains why the CPU-bound process has a high average response time, since even if it's the executing process, it will be preempted by the parasite process and re-inserted into the readylist again.

Bonus



```
PID: 5, I/O-bound process, CPU usage: 3353, Response time: 3, clkcountermsec: 10003
PID: 6, I/O-bound process, CPU usage: 3284, Response time: 3, clkcountermsec: 10005
PID: 7, CPU-bound process, CPU usage: 17, Response time: 1000, clkcountermsec: 10009
PID: 4, I/O-bound process, CPU usage: 3359, Response time: 3, clkcountermsec: 10010

Hello World!
Wei Yi Tan, wytan, Senior, Fall 2023
PID: 0, idle process, CPU usage: 8, Response time: 5005, clkcountermsec: 10017
```

To starve a CPU process, I increased the inner for-loop bounds to 100000 and reduced the sleep time to 2 milliseconds. The reduced sleep time meant that as soon as an I/O-bound process is in the sleep queue, another I/O-bound process is awake and is ready to execute. As a result, all I/O-bound processes has a CPU usage time of approximately 3300 milliseconds and an average response time of 3 milliseconds, whereas the CPU-bound process only has a CPU usage time of 17 milliseconds and an average response time of 1000 milliseconds. This is because each of the I/O-bound processes will keep executing in round-robin style due to its reduced sleep time and higher priority, so all of them have similarly high CPU usage time. Likewise, since the CPU-bound process is always stuck in the readylist because of its lower priority, it has a higher average response time.