Ayush Patwari

Operating Systems [CS503]

Lab2b Answers and Discussion

Note: I have added "ayush edit" tags in the comments where changes have been made. To check the files edited please do a grep with "ayush"

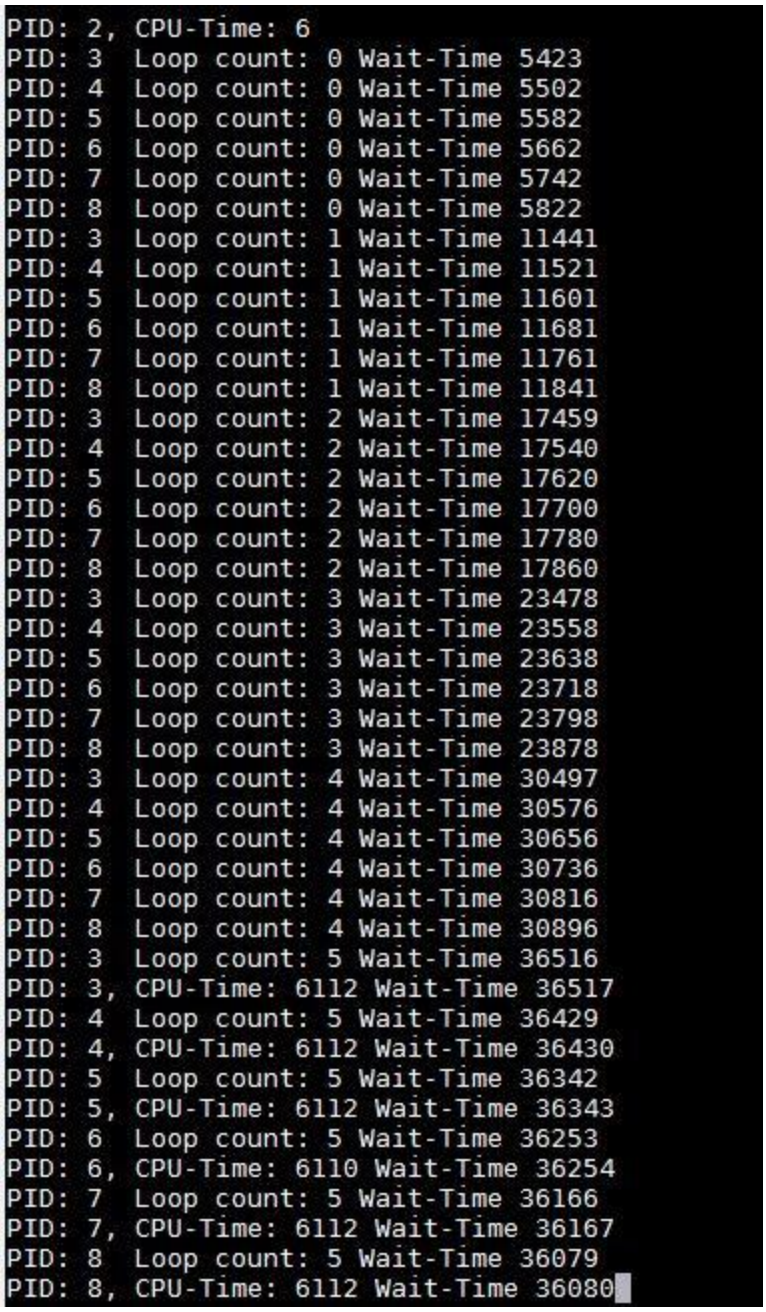
2. XINU Kernel Modification:

The following modifications have been made:

1. Calculation of prcputime: A variable was declared in the procent structure in process.h as prcputime which will maintain the cputime occupied by each process. It is initialized to 0 in initialize.c and update per ms in clkhandler().
2. TS data structure: The data structure as declared in the ts_init and initialized with NUMLEVELS=60 in the initialize.c. A function tsinit is created to populate the entries with appropriate values
3. Multi-level queue: An array of queues was defined in the ready.c and initialized in initialize.c. There are (NUMLEVELS + 1) – one for each user space priority level and the last level for higher priority kernel processes to which the TS scheduling policy doesn't apply to. The NQENT in queue.h was updated to allow for 61 different queues.
4. Scheduling implementation: The resched.c and clkhandler are modified to implement TS scheduling policy.
    a. Resched.c : When resched is called it will first calculate the highest level which is non-empty.
        i. If the entering process is in PR_CURRENT state – it will continue to execute and function returns else it is added to the readylist at the appropriate level according to the prprio field.
        ii. If it is desired to be in PR_SLEEP state it's priority is updated according to dispatcher table ts_slpret so it will wake up with that priority.
        iii. The first process from highest priority list is chosen and made current. If the process has priority 0 (=NULLPROC) then it is checked whether the list is empty. If yes then nullproc is scheduled, else the top of list is dequeued again and nullproc enqueued at the tail of level=0 list.
    b. Clkhandler.c
        i. If the preempt counter reaches zero, indicating that a process actually completed its time slice, the priority is updated according to the ts_tqexp for the current priority. When resched is called the process will already have the updated priority.

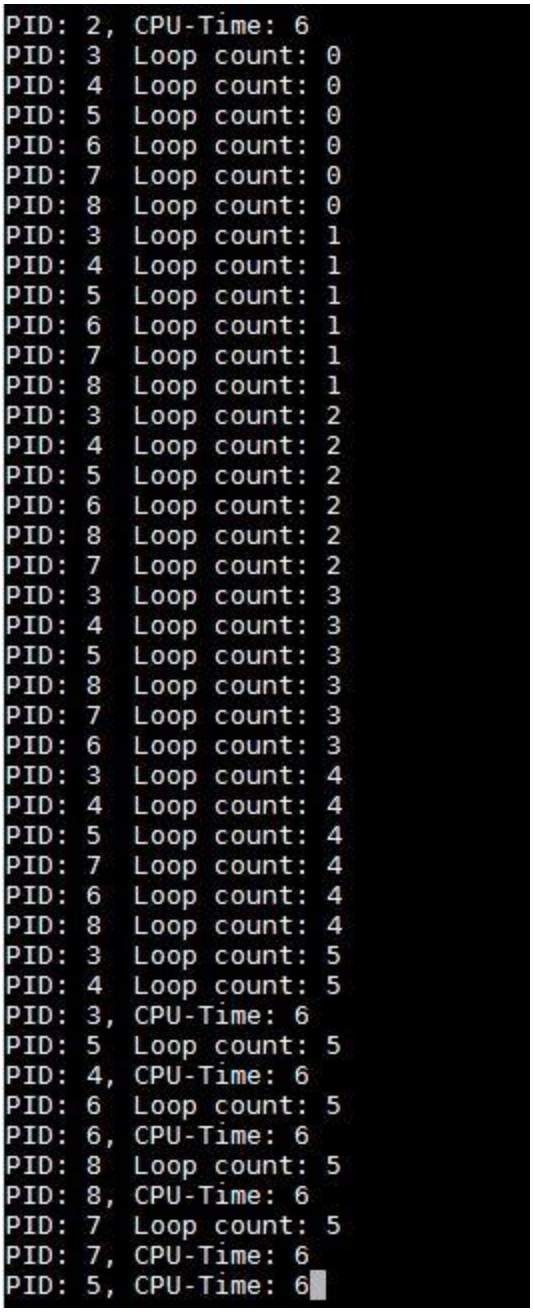
3. Benchmark Evaluation of Performance:

1. CPU-Intensive Processes: The CPU intensive processes were created are stated in the question. After testing for different values of LOOP1 and LOOP2 I settled on LOOP1 = 5, LOOP2=1000000 as they allowed to view the output cleanly and without too much delay. A sample output can be seen below for allCPU processes:



The wait-times are also shown which calculates the total time since starting that each process has been in the system. (Note: it doesn't show the wait time in readylist, just the total time from beginning of process including the waiting time). We can see that the processes are sharing the CPU since each loop is completed in batches and no process has completed significantly earlier or later than any other. Also, if we see the total time (Wait-time) it shows that all of them completed around the same time-frame. Hence there is fairness amongst CPU-bound processes.

2. IO-Intensive Processes: The IO intensive processes were created are stated in the question. Sleep time for this experiment was set to 10 ms. After testing for different values of LOOP1 and LOOP2 I settled on LOOP1 = 5, LOOP2=10 as they allowed to view the output cleanly and without too much delay. A sample output can be seen below for allIO processes:



We can see that the processes are sharing the CPU since each loop is completed in batches and no process has completed significantly earlier or later than any other. The loops are being completed in order. Hence there is fairness amongst CPU-bound processes.

3. Half-n-Half v1: The CPU and IO processes are created with parameters are stated above. Sleeptime = 20ms. We can see a sample output below:

```
PID: 2, CPU-Time: 6
PID: 6  Loop count: 0
PID: 7  Loop count: 0
PID: 8  Loop count: 0
PID: 6  Loop count: 1
PID: 7  Loop count: 1
PID: 8  Loop count: 1
PID: 6  Loop count: 2
PID: 8  Loop count: 2
PID: 7  Loop count: 2
PID: 8  Loop count: 3
PID: 7  Loop count: 3
PID: 6  Loop count: 3
PID: 7  Loop count: 4
PID: 6  Loop count: 4
PID: 8  Loop count: 4
PID: 6  Loop count: 5
PID: 8  Loop count: 5
PID: 8, CPU-Time: 6
PID: 7  Loop count: 5
PID: 7, CPU-Time: 6
PID: 6, CPU-Time: 6
PID: 3  Loop count: 0 Wait-Time 2852
PID: 4  Loop count: 0 Wait-Time 2956
PID: 5  Loop count: 0 Wait-Time 2937
PID: 3  Loop count: 1 Wait-Time 5870
PID: 4  Loop count: 1 Wait-Time 5975
PID: 5  Loop count: 1 Wait-Time 5955
PID: 3  Loop count: 2 Wait-Time 8888
PID: 5  Loop count: 2 Wait-Time 8973
PID: 4  Loop count: 2 Wait-Time 9393
PID: 3  Loop count: 3 Wait-Time 11906
PID: 5  Loop count: 3 Wait-Time 11991
PID: 4  Loop count: 3 Wait-Time 12411
PID: 5  Loop count: 4 Wait-Time 15009
PID: 3  Loop count: 4 Wait-Time 15324
PID: 4  Loop count: 4 Wait-Time 15429
PID: 5  Loop count: 5 Wait-Time 18027
PID: 5, CPU-Time: 6109 Wait-Time 18028
PID: 3  Loop count: 5 Wait-Time 18300
PID: 3, CPU-Time: 6114 Wait-Time 18301
PID: 4  Loop count: 5 Wait-Time 18238
PID: 4, CPU-Time: 6110 Wait-Time 18239
```

The wait-times are also shown for CPU Processes which calculates the total time since starting that each process has been in the system. (Note: it doesn't show the wait time in readylist, just the total time from beginning of process including the waiting time). We can see that the IO-bound processes [6,7,8] are dominating the CPU in the initial part. There is also fair scheduling amongst IO processes since each loop is completed in batches and no process has completed significantly earlier or later than any other. After the IO processes complete with loop-count = 5, only then we see the CPU processes [3, 4, 5] starting to

complete their loops. There is also fair scheduling amongst CPU processes since each loop is completed in batches and no process has completed significantly earlier or later than any other. Also, if we see the total time (Wait-time) it shows that all of them completed around the same time-frame. Hence there is fairness amongst CPU-bound processes.

4. Half-n-Half v2: The CPU and IO processes are created with parameters are stated above. Sleeptime = 7, 13, 29ms respectively are chosen as they are mutually prime and hence less chances of overlap exists. We can see a sample output below: (Sorry for the long output. Please note only the sleep-diffs and statements which have loop completed outputs)

Half V2:

PID: 2, CPU-Time: 6

PID 6, Sleep Diff 0

PID 7, Sleep Diff 0

PID 6, Sleep Diff 0

PID 6, Sleep Diff 0

PID 7, Sleep Diff 0

PID 8, Sleep Diff 0

PID 6, Sleep Diff 0

PID 6, Sleep Diff 0

PID 7, Sleep Diff 0

PID 6, Sleep Diff 0

PID 7, Sleep Diff 0

PID 6, Sleep Diff 1

PID 8, Sleep Diff 0

PID 6, Sleep Diff 0

PID 7, Sleep Diff 0

PID 6, Sleep Diff 0

PID 6, Sleep Diff 0

PID 7, Sleep Diff 0

PID 6, Sleep Diff 0

PID 8, Sleep Diff 0

PID: 6 Loop count: 0

PID 7, Sleep Diff 0

PID 6, Sleep Diff 0

PID 6, Sleep Diff 0

PID 7, Sleep Diff 0

PID 6, Sleep Diff 0

PID 8, Sleep Diff 0

PID 6, Sleep Diff 0

PID 7, Sleep Diff 0

PID 6, Sleep Diff 0

PID 6, Sleep Diff 0

PID 7, Sleep Diff 0

PID 6, Sleep Diff 0

PID 8, Sleep Diff 0

PID 7, Sleep Diff 0

PID: 7 Loop count: 0

PID 6, Sleep Diff 2

PID 6, Sleep Diff 0

PID 7, Sleep Diff 0

PID 6, Sleep Diff 0

PID 8, Sleep Diff 0

PID 6, Sleep Diff 0

PID: 6 Loop count: 1

PID 7, Sleep Diff 0

PID 6, Sleep Diff 0

PID 7, Sleep Diff 0

PID 6, Sleep Diff 1

PID 6, Sleep Diff 0

PID 8, Sleep Diff 0

PID 7, Sleep Diff 1

PID 6, Sleep Diff 0

PID 6, Sleep Diff 0

PID 7, Sleep Diff 0

PID 6, Sleep Diff 0

PID 6, Sleep Diff 0

PID 7, Sleep Diff 0

PID 8, Sleep Diff 0

PID 6, Sleep Diff 0

PID 6, Sleep Diff 0

PID 7, Sleep Diff 0

PID 6, Sleep Diff 0

PID 7, Sleep Diff 0

PID 8, Sleep Diff 0

PID 6, Sleep Diff 1

PID: 6 Loop count: 2

PID 6, Sleep Diff 0

PID 7, Sleep Diff 0

PID 6, Sleep Diff 0

PID 7, Sleep Diff 0

PID 6, Sleep Diff 1

PID 8, Sleep Diff 0

PID 6, Sleep Diff 0

PID 7, Sleep Diff 0

PID: 7 Loop count: 1

PID 6, Sleep Diff 0

PID 6, Sleep Diff 0

PID 7, Sleep Diff 0

PID 6, Sleep Diff 0

PID 8, Sleep Diff 0

PID: 8 Loop count: 0

PID 7, Sleep Diff 0

PID 6, Sleep Diff 1

PID 6, Sleep Diff 0

PID 7, Sleep Diff 0

PID 6, Sleep Diff 0

PID 8, Sleep Diff 0

PID 6, Sleep Diff 0

PID: 6 Loop count: 3

PID 7, Sleep Diff 0

PID 6, Sleep Diff 0

PID 7, Sleep Diff 0

PID 6, Sleep Diff 1

PID 6, Sleep Diff 0

PID 8, Sleep Diff 0

PID 7, Sleep Diff 0

PID 6, Sleep Diff 0

PID 6, Sleep Diff 0

PID 7, Sleep Diff 0

PID 6, Sleep Diff 0

PID 8, Sleep Diff 0

PID 7, Sleep Diff 0

PID 6, Sleep Diff 1

PID 6, Sleep Diff 0

PID 7, Sleep Diff 0

PID 6, Sleep Diff 0

PID 6, Sleep Diff 0

PID 8, Sleep Diff 0

PID 7, Sleep Diff 1

PID 6, Sleep Diff 0

PID: 6 Loop count: 4

PID 6, Sleep Diff 0

PID 7, Sleep Diff 0

PID: 7 Loop count: 2

PID 6, Sleep Diff 0

PID 8, Sleep Diff 0

PID 7, Sleep Diff 1

PID 6, Sleep Diff 0

PID 6, Sleep Diff 0

PID 7, Sleep Diff 0

PID 6, Sleep Diff 0

PID 6, Sleep Diff 0

PID 8, Sleep Diff 0

PID 7, Sleep Diff 1

PID 6, Sleep Diff 0

PID 6, Sleep Diff 0

PID 7, Sleep Diff 0

PID 6, Sleep Diff 0

PID 6, Sleep Diff 0

PID 7, Sleep Diff 0

PID 8, Sleep Diff 0

PID 6, Sleep Diff 0

PID: 6 Loop count: 5

PID: 6, CPU-Time: 72

PID 7, Sleep Diff 0

PID 7, Sleep Diff 0

PID 8, Sleep Diff 0

PID 7, Sleep Diff 0

PID 7, Sleep Diff 0

PID 8, Sleep Diff 0

PID 7, Sleep Diff 0

PID 7, Sleep Diff 0

PID: 7 Loop count: 3

PID 8, Sleep Diff 0

PID 7, Sleep Diff 0

PID 7, Sleep Diff 0

PID 8, Sleep Diff 0

PID: 8 Loop count: 1

PID 7, Sleep Diff 0

PID 7, Sleep Diff 0

PID 8, Sleep Diff 0

PID 7, Sleep Diff 0

PID 7, Sleep Diff 0

PID 8, Sleep Diff 0

PID 7, Sleep Diff 0

PID 7, Sleep Diff 0

PID 8, Sleep Diff 0

PID 7, Sleep Diff 1

PID 7, Sleep Diff 0

PID 7, Sleep Diff 0

PID 8, Sleep Diff 0

PID: 7 Loop count: 4

PID 7, Sleep Diff 0

PID 7, Sleep Diff 0

PID 8, Sleep Diff 0

PID 7, Sleep Diff 0

PID 7, Sleep Diff 0

PID 8, Sleep Diff 0

PID 7, Sleep Diff 0

PID 7, Sleep Diff 0

PID 8, Sleep Diff 0

PID 7, Sleep Diff 0

PID 7, Sleep Diff 0

PID 8, Sleep Diff 0

PID 7, Sleep Diff 0

PID 7, Sleep Diff 0

PID 8, Sleep Diff 0

PID 7, Sleep Diff 0

PID: 7 Loop count: 5

PID: 7, CPU-Time: 72

PID 8, Sleep Diff 0

PID 8, Sleep Diff 0

PID: 8 Loop count: 2

PID 8, Sleep Diff 19

PID 8, Sleep Diff 0

PID 8, Sleep Diff 0

PID 8, Sleep Diff 0

PID 8, Sleep Diff 0

PID 8, Sleep Diff 0

PID 8, Sleep Diff 19

PID 8, Sleep Diff 19

PID 8, Sleep Diff 0

PID 8, Sleep Diff 0

PID 8, Sleep Diff 0

PID: 8 Loop count: 3

PID 8, Sleep Diff 0

PID 8, Sleep Diff 0

PID 8, Sleep Diff 0

PID 8, Sleep Diff 0

PID 8, Sleep Diff 0

PID 8, Sleep Diff 0

PID 8, Sleep Diff 0

PID 8, Sleep Diff 0

PID 8, Sleep Diff 0

PID 8, Sleep Diff 0

PID 8, Sleep Diff 0

PID: 8 Loop count: 4

PID 8, Sleep Diff 0

PID 8, Sleep Diff 0

PID 8, Sleep Diff 0

PID 8, Sleep Diff 0

PID 8, Sleep Diff 0

PID 8, Sleep Diff 0

PID 8, Sleep Diff 0

PID 8, Sleep Diff 0

PID 8, Sleep Diff 0

PID 8, Sleep Diff 0

PID 8, Sleep Diff 0

PID 8, Sleep Diff 0

PID: 8 Loop count: 5

PID: 8, CPU-Time: 72

PID: 3 Loop count: 0 Wait-Time 3132

PID: 4 Loop count: 0 Wait-Time 3402

PID: 5 Loop count: 0 Wait-Time 3577

PID: 3 Loop count: 1 Wait-Time 6070

PID: 4 Loop count: 1 Wait-Time 6421

PID: 5 Loop count: 1 Wait-Time 6595

PID: 3 Loop count: 2 Wait-Time 9088

PID: 4 Loop count: 2 Wait-Time 9439

PID: 5 Loop count: 2 Wait-Time 9614

PID: 3 Loop count: 3 Wait-Time 12106

PID: 4 Loop count: 3 Wait-Time 12458

PID: 5 Loop count: 3 Wait-Time 12632

PID: 3 Loop count: 4 Wait-Time 15124

PID: 4 Loop count: 4 Wait-Time 15476

PID: 5 Loop count: 4 Wait-Time 15651

PID: 3 Loop count: 5 Wait-Time 18142

PID: 3, CPU-Time: 6150 Wait-Time 18143

PID: 5 Loop count: 5 Wait-Time 18411

PID: 5, CPU-Time: 6146 Wait-Time 18412

PID: 4 Loop count: 5 Wait-Time 18546

PID: 4, CPU-Time: 6148 Wait-Time 18547

We can see that the IO-bound processes [6,7,8] are dominating the CPU in the initial part. There is also fair scheduling amongst IO processes since each loop is completed in batches and no process has completed significantly earlier or later than any other. The measure of fairness can also be seen from the sleep-diff output which is shown. It is the difference in the desired sleep time and the actual time after which scheduler scheduled this process again after sleeping. From the output we can see that sleep diffs are 0 in most of the cases. They are one only when two processes wake-ip at the same time. Since I have chosen mutually prime-time sleep times they occur rarely but do occur since at some point that is bound to happen. This shows that the scheduler is able to keep the sleeptime promises to IO bound processes. After the IO processes complete with loop-count = 5, only then we see the CPU processes [3,

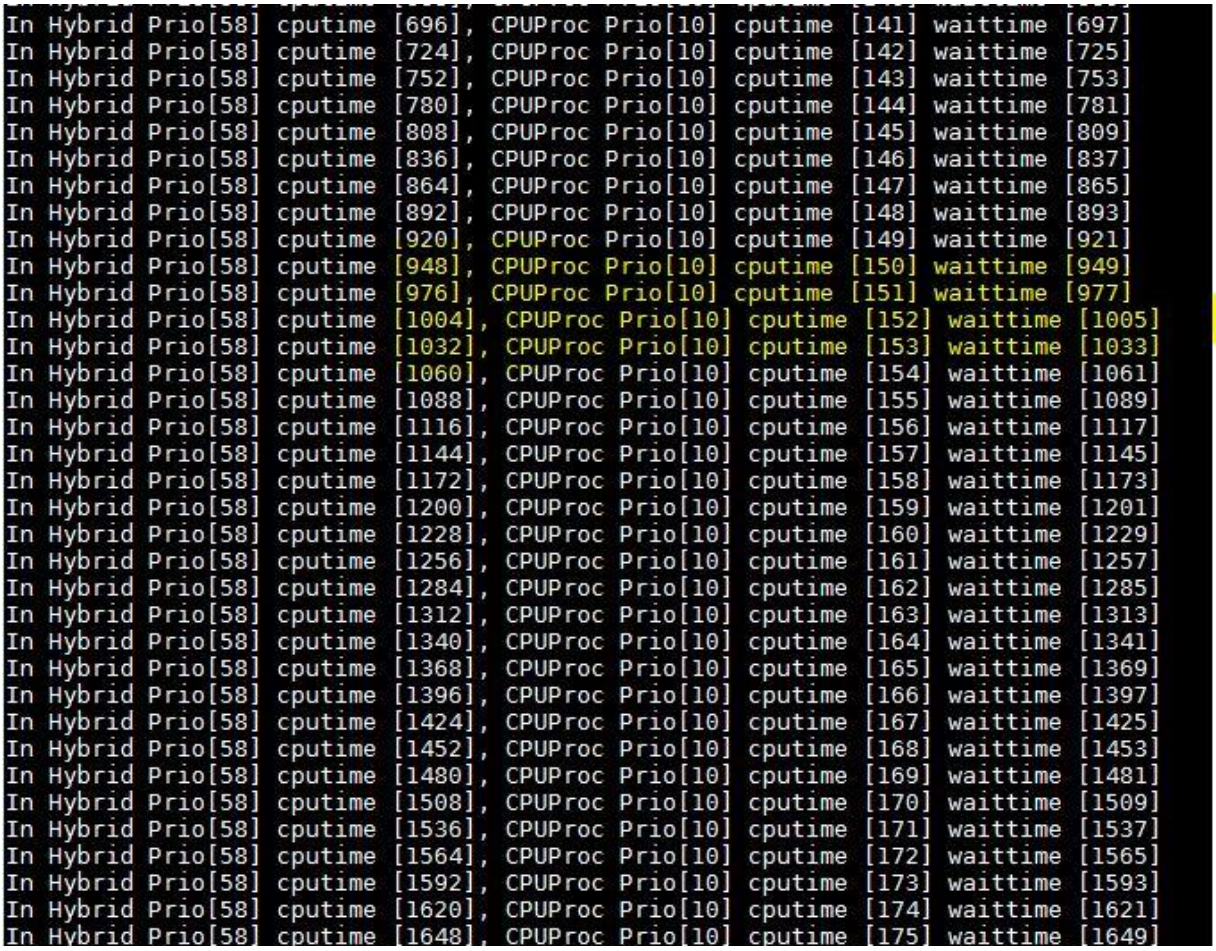
4, 5] starting to complete their loops. There is also fair scheduling amongst CPU processes since each loop is completed in batches and no process has completed significantly earlier or later than any other. Also, if we see the total time (Wait-time) it shows that all of them completed around the same time-frame. Hence there is fairness amongst CPU-bound processes.

Starvation (BONUS): To see the effects of starvation the following hybrid process was created:

```
void hybridprocess(pid32 cpu_intensive) {
        /* ayush edit */


        struct procent *ptr, *mptr;
        /* If the cpu_intensive process in not running -> exit */

        if(!isbadpid(cpu_intensive))
                ptr = &proctab[cpu_intensive];
        else {
                kprintf("\nBad CPU Process ID %d! Aborting..");
                return;
        }

        /* Method to garner lion's share:
         * 1. Get current time slice and define usable time as 90% of that.
         * 2. Run a infinite while loop
         * 3. Run a big inner loop if cpu_intensive process is still active else exit.
         * 4. Inside the loop check if usable time exceeded and call sleepms(1)
         *    Reset the usable time and starttime = myglobalclock
         */

        mptr = &proctab[getpid()];
        uint32 starttime = myglobalclock, usabletime = tstab[mptr->prprio].ts_quantum;
        usabletime = (uint32)((float)usabletime * 0.9f);


        int loop = 100000000;
        while(1) {

                if(isbadpid(cpu_intensive)) {

                        kprintf("CPU Process complete..Hybrid exiting!");
                        break;
                }
                int j;
                for(j = 0; j <= loop; j++) {

                        uint32 used = myglobalclock - starttime;
                        if(used  > usabletime) {

                                sleepms(1);
                                starttime = myglobalclock;
                                usabletime = tstab[mptr->prprio].ts_quantum;
                                usabletime = (uint32)((float)usabletime * 0.7f);

                                //kprintf("\nIn Hybrid Prio[%d] cputime [%d], CPUProc Prio[%d] cput
                        }

                        // Random computations
                        int a = 1, b = 2;
                        int c = a + b;
                        a = c;
                }

        }
}
```

The idea adopted is as follows. The process runs in an infinite outer loop and a large inner loop. Inside the inner loop whenever the processes exceeds 70% of current time slice it goes to sleepms(1). This

ensures that when it wakes up it will have a higher priority and be rescheduled again. A cpu-bound process was run along with this process concurrently. Output is shown below:



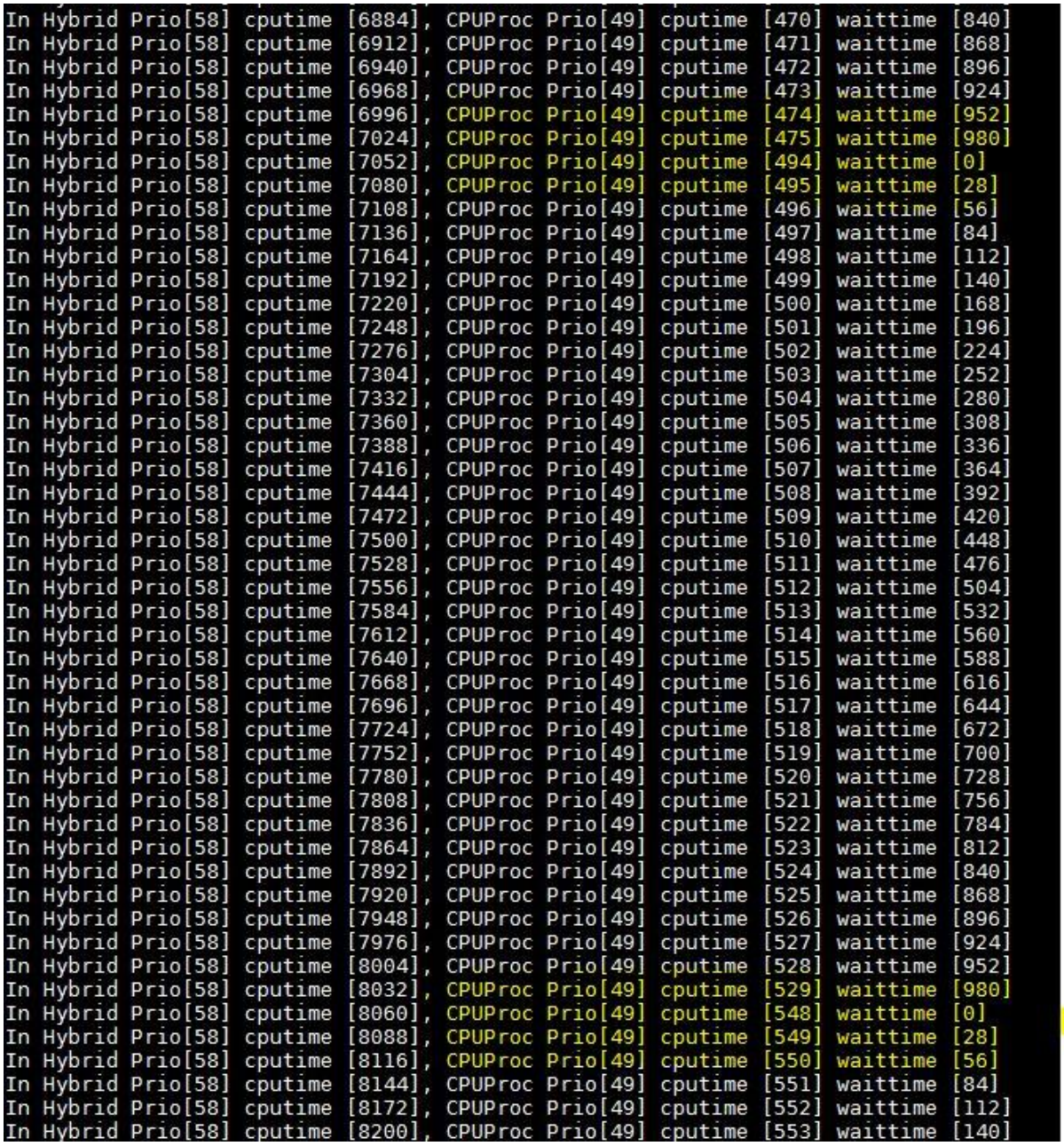
The figure shows a output which is printed whenever the hybrid process has completed its usable time and returns from the sleepms. The yellow region shows that even though the CPU processes has been waiting for 1000ms to get a full time slice, it is not able to get so. If we see that in each iteration the hybrid processes cputime increases by around 28 (.7 * 40) ms, whereas when it goes to sleep it the CPUProc is able to execute for only 1 ms after which it is preempted again.

To implement the policy to prevent this starvation following edits were made:

1. A waittime variable is introduced in process structure to keep track of time since a process could complete a full time slice. It is initialized to zero in initialize.c and set to 0 in ready when a process has been just created or becomes ready after sleep state.
2. The variable is updated in the clkhandler as shown below.

```
int i;
struct procent *ptr;
for(i = 2; i < NPROC; i++) {
        if(!isbadpid(i)) {

                ptr = &proctab[i];
                if(ptr->prstate == PR_READY) {
                        ptr->waittime++;

                        if(BONUS == 1) {
                                int waitseconds = ptr->waittime / 1000;
                                if(waitseconds > tstab[ptr->prprio].ts_maxwait) {

                                        pid32 pid = getitem(i);
                                        ptr->prprio = tstab[ptr->prprio].ts_lwait;
                                        insert(pid, multiqueue[ptr->prprio], ptr->prprio);

                                }
                        }
                }

        }
}
```

   For each process which is ready the waittime is incremented. If the waittime (in seconds) exceeds as that of ts_maxwait in the tstab the process is removed from current level, priority updated to ts_lwait and added to that level. The tsentry data structure was also updated with ts_maxwait and ts_lwait and initialized in tsinit(). The effect of this modification can be seen below:



Once a the CPU Proc has waited for 1000ms its priority is bumped to 59, it executes for full 20ms, its priority is reduced to 49 and trasnfers control to the Hybrid process again. It can be seen in the highlighted section that cputime jumps from 475 to 494 for the CPUProc. Also as a measure of total gain in waittime please look at the two experimental outputs, first is without starvation policy and second with the policy in place.

```
PID: 2, CPU-Time: 7
PID: 3  Loop count: 0 Wait-Time 26357
PID: 3  Loop count: 1 Wait-Time 55966
PID: 3  Loop count: 2 Wait-Time 85575
PID: 3  Loop count: 3 Wait-Time 115184
PID: 3  Loop count: 4 Wait-Time 144793
PID: 3  Loop count: 5 Wait-Time 174402
PID: 3, CPU-Time: 6126 Wait-Time 174431CPU Process complete..Hybrid exiting!
```

```
PID: 2, CPU-Time: 6
PID: 3  Loop count: 0 Wait-Time 18152
PID: 3  Loop count: 1 Wait-Time 38239
PID: 3  Loop count: 2 Wait-Time 58214
PID: 3  Loop count: 3 Wait-Time 78189
PID: 3  Loop count: 4 Wait-Time 98164
PID: 3  Loop count: 5 Wait-Time 118139
PID: 3, CPU-Time: 6115 Wait-Time 118168CPU Process complete..Hybrid exiting!
```

We can easily see that the wait-times have decreased sigficantly since after every 1000ms the CPUProc is able to execute for full 20ms time slice. Note: to on/off the starvation policy set the BONUS macro to 1/ anything else in the clkhandler.