# COMP3520 Assignment 2: Discussion Document

## Question 1

1. Initialize data structures
    i. Create three ready queues: Level-0 (L0), Level-1 (L1), Level-2 (L2).
    ii. Create a Job Dispatch list from the input.
    iii. For each job record store: arrival, burst_total, burst_left, init_prio, pid, first_start, finish_time, ran_in_level, wait_total, wait_since_tail.
2. Read parameters
    i. Ask user to enter integer values for t0, t1, t2, W (quanta for L0/L1/L2 and aging window).
3. Sort the Job Dispatch list by non-decreasing arrival time.
4. Main loop
   While there is a currently running process or any of {Job Dispatch, L0, L1, L2} is non-empty:
    i. Unload arrivals at current time nowRepeatedly dequeue from Job Dispatch while arrival == now:
        (a) If init_prio == 0, enqueue to tail of L0 and set level=0.
        (b) Else if init_prio == 1, enqueue to tail of L1 and set level=1.
        (c) Else enqueue to tail of L2 and set level=2.
        (d) For each enqueued job set ran_in_level=0, wait_since_tail=0.
    ii. Immediate preemption on higher-priority arrival
        (a) If a process is currently running:
            I. If it runs in L2 and any L0 or L1 job arrived at step
                1. Send SIGTSTP to the running process; waitpid(..., WUNTRACED).
                2. Enqueue the process back to the head of L2 (do not reset ran_in_level).
                3. Clear current running process.
            II. Else if it runs in L1 and any L0 job arrived at step
                1. Send SIGTSTP; waitpid(..., WUNTRACED).
                2. Enqueue the process to the head of L1 (keep ran_in_level).
                3. Clear current running process.

iii. Starvation prevention (aging by W)

(a) If L1 is non-empty and the head of L1 has wait_since_tail ≥ W:

I. Move all jobs in L1 (preserving order) to the tail of L0 and set their level=0, ran_in_level=0, wait_since_tail=0.

II. Then move all jobs in L2 (preserving order) to the tail of L0 with the same resets.

(b) Else if L2 is non-empty and the head of L2 has wait_since_tail ≥ W:

I. Move all jobs in L2 (preserving order) to the tail of L0 with the same resets.

iv. Dispatch if CPU idle

(a) If there is no running process:

I. If L0 non-empty: dequeue head of L0 as running.

II. Else if L1 non-empty: dequeue head of L1 as running.

III. Else if L2 non-empty: dequeue head of L2 as running.

IV. Else (no ready jobs):

1. If Job Dispatch still has future arrivals, set now ← next arrival time and continue to step i.

2. Otherwise break (done).

v. (Re)start or resume the selected job

(a) If running has never started:

I. pid = fork();

1. Child: execv("./sigtrap", argv_for_id); (on error, exit).

2. Parent: store pid, mark job as started.

II. Send SIGCONT to running (safe even if still running).

III. If first_start is unset, set first_start ← now.

vi. Execute for one tick (1 second)

(a) Increase wait_total and wait_since_tail by 1 for every job in L0, L1, L2.

(b) Sleep for one second (a scheduling tick).

(c) Update:

I. running.burst_left −= 1

II. running.ran_in_level += 1

III. now += 1

vii. Completion check

(a) If running.burst_left == 0:

I. Send SIGINT to running.pid to terminate; then waitpid(pid, 0).

II. Set finish_time ← now.

III. Clear current running; continue to next iteration.

viii. Quantum/queue rules (end-of-slice actions)

I. If running. level == 0 and ran_in_level ≥ t0:

1. Send SIGTSTP; waitpid(..., WUNTRACED).
2. Reset ran_in_level ← 0, wait_since_tail ← 0, set level ← 1.
3. Enqueue to tail of L1; clear running.

II. Else if running. level == 1 and ran_in_level ≥ t1:

1. Send SIGTSTP; waitpid(..., WUNTRACED).
2. Reset ran_in_level ← 0, wait_since_tail ← 0, set level ← 2.
3. Enqueue to tail of L2; clear running.

III. Else if running. level == 2 and ran_in_level ≥ t2:

1. (Round-Robin in L2)
2. Send SIGTSTP; waitpid(..., WUNTRACED).
3. Reset ran_in_level ← 0, wait_since_tail ← 0.
4. Enqueue to tail of L2; clear running.

IV. Else:

1. Keep running for the next tick (go back to step i).

5. Cleanup (safety)

i. For any started but unfinished child, send SIGINT and waitpid to reap.

6. Compute and print metrics

i. For each job:

(a) Turnaround = finish_time − arrival

(b) Response = first_start − arrival

(c) Waiting = wait_total

ii. Print Average Turnaround Time, Average Waiting Time, Average Response Time.

# Question 2

## Overall approach

I verified correctness in three layers:

- Local invariants & unit tests
  Validate queue operations and bookkeeping never go out of sync (no duplicates, counters add up, etc).
- Scenario-based functional tests
  Hand-crafted small workloads that isolate each rule from the spec (dispatch order, preemption, demotion, L2 RR, aging W). For a few tiny cases I compute the schedule and metrics by hand.
- Differential testing (oracle)
  I wrote a deterministic simulator (no forking, can be seen in simulation.c) that implements the same rules. For dozens of workloads (both fixed and random), I compared:
  - Per-job finish time, first start, waiting, and
  - Average Turnaround/Waiting/Response.
    The signal-driven version must match the simulator bit-for-bit. If it didn't, I treated the simulator as the oracle and debugged the discrepancy.

## What constitutes "correct output"

For each job j:

- Turnaround = finish_time - arrival.
- Response = first_start - arrival.
- Waiting = the sum of unit ticks while queued (never while running).

Global metrics are just the averages. The implementation increments wait totals for all queued jobs once per tick and never for the running job, which matches the spec's intent.

# Question 3

## Parameters:

t0 = 6,

t1 = 6,

t2 = 32,

W = 60

## Pick t0 = 6 to finish most jobs at Level-0

From the CPU times [3, 50, 2, 6, 12, 4, 6, 11, 3, 2, 4, 6, 5, 6, 1, 41, 3, 2, 6],
15/19 jobs ( ≈ 79%) are ≤ 6.

With t0 = 6, the vast majority completes at L0 in a single dispatch. This: keeps response time low (they start immediately or soon), avoids extra context switches/demotions, which helps turnaround.

Smaller t0 improves response further but hurts turnaround by fragmenting medium jobs and a larger t0 doesn't help response and delays the few long jobs.

### Make t1 = 6 to tidy up "mediums" without hogging CPU

Jobs that overflow L0 (e.g., 11 and 12) will typically finish in one L1 dispatch (or two short ones if preempted by new L0 arrivals). Using the same slice as L0: maintains predictable latency when L0 goes empty, preventing long L1 bursts from blocking a fresh L0 arrival (since L1 is preempted immediately on L0 arrival anyway).

### Use a large L2 quantum t2 = 32 for the few heavy hitters

Only a handful are true long runners (41 and 50 dominate; the rest are ≤12). L2 only runs when both L0 and L1 are empty; giving large quanta there:cuts down RR churn and context-switch costs and lets the big jobs make meaningful progress when they finally get CPU while reducing turnaround without affecting response (because new L0/L1 arrivals still preempt L2 instantly). (Anything ≥32 performs equivalently on this workload so 32 is a tidy, safe choice.)

### Set W = 60 to avoid noisy promotions

With many L0 arrivals, an aggressive W would keep yanking L2 work up to L0, which increases head-of-line blocking and worsens both ATT and ART. W = 60 is high enough to avoid gratuitous aging events while there's a steady L0 stream and still protects against pathological starvation (if arrivals dry up, the waiting budget ticks up and promotion triggers).

### Comparisons (to show the trade-off space)

Latency-leaning (snappier first responses):

t0=3, t1=6, t2=32, W=30

ATT: 33.74  AWT: 24.63  ART: 3.58

Balanced guess:

t0=6, t1=12, t2=20, W=30

ATT: 32.16  AWT: 23.05  ART: 12.00

Bigger slices: t0=8, t1=8, t2=32, W=60

ATT: 33.47  AWT: 24.37  ART: 10.26

## Question 4

No bugs were found.

# Running the Program

**NOTE:** It is assumed that there already exists a binary called ./process

Compile using

```
make
```

Then run:

```
./main
```

This will prompt the user to enter t0 t1 t2 W. separated by spaces, see below for example:

```
Enter t0 t1 t2 W: 5 5 5 10
```

Afterwards, press Enter and this will prompt the user to enter the job list, see below for example:

```
Paste jobs (arrival cputime priority), commas optional. Ctrl-D when done:
0, 8, 2
0, 2, 0
0, 4, 1
```

Finally, press Ctrl + D after all jobs have been entered.