OS (BCSE303L)

DA-1

PRESENTED BY:24BCE540,TARUN KRISHNA MANIVANNAN

2)

```
#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <math.h>


#define NUM_PROCS 25

#define NUM_CORES 4

#define MAX_TIME 10000

#define EVAL_INTERVAL 5

#define RR_QUANTUM 4


typedef enum {ALG_SJF=0, ALG_RR=1, ALG_PRIO=2} alg_t;


typedef struct {

   int pid;

   int arrival;

   int burst;

   int remaining;

   int priority;

   int start_time;

   int finish_time;
```

```c
    int assigned_core;
    int last_run_time;
} proc_t;

typedef struct {
    int id;
    double busy_time;
    double idle_time;
    double energy_consumed;
    double energy_threshold;
    double power_coeff;
    int served_count;
    alg_t current_alg;
    int last_eval;
    int q_len;
    int q_cap;
    proc_t **queue;
} core_t;

typedef struct {
    double total_energy;
    double core_energy[NUM_CORES];
    double sim_time;
    double avg_response_time;
    double avg_turnaround_time;
    double throughput_per_core[NUM_CORES];
```

```
    double load_stddev;
    double max_core_load;
    double min_core_load;
} run_stats_t;

void swap_ptr(proc_t **a, proc_t **b) { proc_t *t=*a; *a=*b; *b=t; }

void core_enqueue(core_t *c, proc_t *p) {
    if (c->q_len >= c->q_cap) {
        c->q_cap = c->q_cap ? c->q_cap*2 : 8;
        c->queue = (proc_t**)realloc(c->queue, sizeof(proc_t*) * c->q_cap);
    }
    c->queue[c->q_len++] = p;
}

void core_remove_at(core_t *c, int idx) {
    if (idx < 0 || idx >= c->q_len) return;
    for (int j = idx; j+1 < c->q_len; ++j) c->queue[j] = c->queue[j+1];
    c->q_len--;
}

int core_pick_sjf(core_t *c) {
    int best = -1;
    int best_rem = 1e9;
    for (int i = 0; i < c->q_len; ++i) {
        if (c->queue[i]->remaining > 0 && c->queue[i]->remaining < best_rem) {
```

```
            best_rem = c->queue[i]->remaining;
            best = i;
        }
    }
    return best;
}

int core_pick_prio(core_t *c) {
    int best = -1;
    int best_pr = 1e9;
    for (int i = 0; i < c->q_len; ++i) {
        if (c->queue[i]->remaining > 0 && c->queue[i]->priority < best_pr) {
            best_pr = c->queue[i]->priority;
            best = i;
        }
    }
    return best;
}

int core_pick_rr(core_t *c) {
    for (int i = 0; i < c->q_len; ++i) {
        if (c->queue[i]->remaining > 0) return i;
    }
    return -1;
}
```

```
double placement_metric(core_t *c, proc_t *p) {
    double util = c->busy_time / (1.0 + c->busy_time + c->idle_time);
    double energy_frac = c->energy_consumed / (1e-6 + c->energy_threshold);
    double q = (double)c->q_len;
    double pr_norm = (6 - p->priority) / 5.0;
    double w1 = 0.6, w2 = 0.5, w3 = 0.8, w4 = 0.9;
    double m = w1*util + w2*energy_frac + w3*q - w4*pr_norm;
    return m;
}

double compute_stddev(double arr[], int n, double mean) {
    double s = 0.0;
    for (int i=0;i<n;i++) s += (arr[i]-mean)*(arr[i]-mean);
    return sqrt(s/n);
}

run_stats_t simulate(proc_t procs_in[], int n, int mode, int verbose) {
    proc_t procs[NUM_PROCS];
    for (int i=0;i<n;i++) procs[i] = procs_in[i];

    core_t cores[NUM_CORES];
    for (int i = 0; i < NUM_CORES; ++i) {
        cores[i].id = i;
        cores[i].busy_time = cores[i].idle_time = 0.0;
        cores[i].energy_consumed = 0.0;
        cores[i].energy_threshold = 50.0 + 10.0*i;
```

```
        cores[i].power_coeff = 1.0 + 0.1*i;
        cores[i].served_count = 0;
        cores[i].current_alg =
(mode==0?ALG_PRIO:(mode==1?ALG_SJF:(mode==2?ALG_RR:ALG_PRIO)));
        cores[i].last_eval = 0;
        cores[i].q_len = 0;
        cores[i].q_cap = 8;
        cores[i].queue = (proc_t**)malloc(sizeof(proc_t*) * cores[i].q_cap);
    }

    int completed = 0;
    int t = 0;
    double context_switch_energy = 0.05;
    double base_power = 1.0;

    for (int i = 0; i < n; ++i) {
        procs[i].remaining = procs[i].burst;
        procs[i].start_time = -1;
        procs[i].finish_time = -1;
        procs[i].assigned_core = -1;
        procs[i].last_run_time = -1;
    }

    int running_idx[NUM_CORES];
    int rr_remaining_quantum[NUM_CORES];
    for (int i=0;i<NUM_CORES;i++) { running_idx[i] = -1; rr_remaining_quantum[i] = 0; }
```

```
    while (completed < n && t < MAX_TIME) {
        for (int i = 0; i < n; ++i) {
            if (procs[i].arrival == t) {
                int best_core = 0;
                double best_metric = placement_metric(&cores[0], &procs[i]);
                for (int c=1;c<NUM_CORES;c++) {
                    double m = placement_metric(&cores[c], &procs[i]);
                    if (m < best_metric) { best_metric = m; best_core = c; }
                }
                procs[i].assigned_core = best_core;
                core_enqueue(&cores[best_core], &procs[i]);
            }
        }

        for (int c=0;c<NUM_CORES;c++) {
            if (mode == 0) {
                if (t - cores[c].last_eval >= EVAL_INTERVAL) {
                    cores[c].last_eval = t;
                    double util = (cores[c].busy_time) / (1.0 + cores[c].busy_time + cores[c].idle_time);
                    int qlen = cores[c].q_len;
                    double energy_frac = cores[c].energy_consumed / (1e-9 +
cores[c].energy_threshold);
                    alg_t choose = cores[c].current_alg;
                    if (cores[c].energy_consumed > cores[c].energy_threshold) {
                        choose = ALG_SJF;
```

```
        } else if (util > 0.7 && qlen > 4) {
          choose = ALG_RR;
        } else {
          choose = ALG_PRIO;
        }
        if (choose != cores[c].current_alg) {
          cores[c].energy_consumed += context_switch_energy;
          cores[c].current_alg = choose;
          rr_remaining_quantum[c] = 0;
          running_idx[c] = -1;
        }
      }
    }
  }

  for (int c=0;c<NUM_CORES;c++) {
    core_t *core = &cores[c];
    int pick = -1;
    if (core->q_len == 0) {
      running_idx[c] = -1;
    } else {
      alg_t alg = core->current_alg;
      if (alg == ALG_SJF) {
        pick = core_pick_sjf(core);
      } else if (alg == ALG_PRIO) {
        pick = core_pick_prio(core);
```

```
        } else {
            if (running_idx[c] >= 0 && running_idx[c] < core->q_len &&
                core->queue[running_idx[c]]->remaining > 0 && rr_remaining_quantum[c] > 0) {
                pick = running_idx[c];
            } else {
                pick = core_pick_rr(core);
                rr_remaining_quantum[c] = RR_QUANTUM;
            }
        }
        if (pick != running_idx[c]) {
            if (running_idx[c] != -1) core->energy_consumed += context_switch_energy;
            running_idx[c] = pick;
        }
    }
}

for (int c=0;c<NUM_CORES;c++) {
    core_t *core = &cores[c];
    int ridx = running_idx[c];
    if (ridx == -1) {
        core->idle_time += 1.0;
        continue;
    }
    proc_t *p = core->queue[ridx];
    if (p->remaining <= 0) {
        core_remove_at(core, ridx);
```

```c
            running_idx[c] = -1;
            continue;
        }
        if (p->start_time == -1) p->start_time = t;
        p->remaining -= 1;
        core->busy_time += 1.0;
        core->energy_consumed += base_power * core->power_coeff;
        if (core->current_alg == ALG_RR) rr_remaining_quantum[c]--;
        if (p->remaining == 0) {
            p->finish_time = t+1;
            core->served_count += 1;
            completed++;
            core_remove_at(core, ridx);
            running_idx[c] = -1;
            rr_remaining_quantum[c] = 0;
        } else {
            if (core->current_alg == ALG_RR && rr_remaining_quantum[c] <= 0) {
                proc_t *tmp = core->queue[ridx];
                core_remove_at(core, ridx);
                core_enqueue(core, tmp);
                running_idx[c] = -1;
                rr_remaining_quantum[c] = 0;
            }
        }
    }
```

```
    t++;
}

run_stats_t stats;
stats.sim_time = (double)t;
stats.total_energy = 0.0;
double response_sum = 0.0, turnaround_sum = 0.0;
for (int c=0;c<NUM_CORES;c++) {
    stats.core_energy[c] = cores[c].energy_consumed;
    stats.total_energy += cores[c].energy_consumed;
    stats.throughput_per_core[c] = cores[c].served_count / stats.sim_time;
}
for (int i=0;i<n;i++) {
    if (procs[i].start_time >= 0) {
        response_sum += (procs[i].start_time - procs[i].arrival);
        turnaround_sum += (procs[i].finish_time - procs[i].arrival);
    } else {
        response_sum += (stats.sim_time - procs[i].arrival);
        turnaround_sum += (stats.sim_time - procs[i].arrival);
    }
}
stats.avg_response_time = response_sum / n;
stats.avg_turnaround_time = turnaround_sum / n;
double loads[NUM_CORES];
double total_load = 0.0;
for (int c=0;c<NUM_CORES;c++) { loads[c] = cores[c].busy_time; total_load += loads[c]; }
```

```
    double avg_load = total_load / NUM_CORES;
    stats.max_core_load = loads[0]; stats.min_core_load = loads[0];
    for (int c=1;c<NUM_CORES;c++) {
        if (loads[c] > stats.max_core_load) stats.max_core_load = loads[c];
        if (loads[c] < stats.min_core_load) stats.min_core_load = loads[c];
    }
    stats.load_stddev = compute_stddev(loads, NUM_CORES, avg_load);
    for (int c=0;c<NUM_CORES;c++) free(cores[c].queue);
    return stats;
}

void generate_processes(proc_t procs[], unsigned seed) {
    srand(seed);
    for (int i = 0; i < NUM_PROCS; ++i) {
        procs[i].pid = i;
        procs[i].arrival = rand() % 21;
        procs[i].burst = 1 + (rand() % 20);
        procs[i].priority = 1 + (rand() % 5);
    }
}

void print_stats(const char *title, run_stats_t *s) {
    printf("=== %s ===\n", title);
    printf("Sim time: %.0f\n", s->sim_time);
    printf("Total energy: %.3f\n", s->total_energy);
    for (int i=0;i<NUM_CORES;i++) {
```

```c
        printf("  Core %d energy: %.3f  throughput: %.4f jobs/unit\n", i, s->core_energy[i], s->throughput_per_core[i]);
    }
    printf("Average response time: %.3f\n", s->avg_response_time);
    printf("Average turnaround time: %.3f\n", s->avg_turnaround_time);
    printf("Load stddev: %.4f  (max load %.2f min load %.2f)\n", s->load_stddev, s->max_core_load, s->min_core_load);
    printf("\n");
}

int main() {
    proc_t base_procs[NUM_PROCS];
    unsigned seed = 123456;
    generate_processes(base_procs, seed);
    printf("Generated workload (pid, arrival, burst, priority):\n");
    for (int i=0;i<NUM_PROCS;i++) {
        printf(" %2d: arr=%2d burst=%2d pr=%d\n", base_procs[i].pid, base_procs[i].arrival, base_procs[i].burst, base_procs[i].priority);
    }
    printf("\n");
    run_stats_t hybrid_stats = simulate(base_procs, NUM_PROCS, 0, 0);
    run_stats_t sjf_stats = simulate(base_procs, NUM_PROCS, 1, 0);
    run_stats_t rr_stats = simulate(base_procs, NUM_PROCS, 2, 0);
    run_stats_t prio_stats = simulate(base_procs, NUM_PROCS, 3, 0);
    print_stats("Hybrid Scheduler", &hybrid_stats);
    print_stats("Forced SJF (all cores)", &sjf_stats);
    print_stats("Forced RR (all cores)", &rr_stats);
```

```
    print_stats("Forced Priority (all cores)", &prio_stats);

    printf("=== Comparison Summary ===\n");

    printf("Total energy: Hybrid=%.3f | SJF=%.3f | RR=%.3f | PRIO=%.3f\n",
           hybrid_stats.total_energy, sjf_stats.total_energy, rr_stats.total_energy,
prio_stats.total_energy);

    printf("Avg response time: Hybrid=%.3f | SJF=%.3f | RR=%.3f | PRIO=%.3f\n",
           hybrid_stats.avg_response_time, sjf_stats.avg_response_time,
rr_stats.avg_response_time, prio_stats.avg_response_time);

    printf("Avg turnaround time: Hybrid=%.3f | SJF=%.3f | RR=%.3f | PRIO=%.3f\n",
           hybrid_stats.avg_turnaround_time, sjf_stats.avg_turnaround_time,
rr_stats.avg_turnaround_time, prio_stats.avg_turnaround_time);

    printf("Load stddev: Hybrid=%.4f | SJF=%.4f | RR=%.4f | PRIO=%.4f\n",
           hybrid_stats.load_stddev, sjf_stats.load_stddev, rr_stats.load_stddev,
prio_stats.load_stddev);

    printf("\n");

    return 0;

}
```

Output:

```
Generated workload (pid, arrival, burst, priority):

  0: arr= 8 burst= 4 pr=2

  1: arr= 2 burst=13 pr=2

  2: arr= 1 burst= 8 pr=1

  3: arr=17 burst=10 pr=2

  4: arr= 7 burst= 3 pr=2

  5: arr=16 burst= 9 pr=5

  6: arr= 7 burst= 1 pr=1

  7: arr=19 burst= 9 pr=3
```

```
  8: arr=10 burst=11 pr=5
  9: arr=10 burst= 5 pr=2
 10: arr= 6 burst=16 pr=2
 11: arr=16 burst= 7 pr=4
 12: arr= 3 burst=11 pr=1
 13: arr=17 burst=15 pr=3
 14: arr=20 burst= 8 pr=5
 15: arr= 1 burst= 5 pr=5
 16: arr= 0 burst=12 pr=3
 17: arr=12 burst=15 pr=3
 18: arr=19 burst=10 pr=3
 19: arr= 8 burst=20 pr=4
 20: arr=14 burst= 3 pr=1
 21: arr=17 burst=10 pr=2
 22: arr= 7 burst= 5 pr=5
 23: arr=10 burst=18 pr=1
 24: arr= 1 burst= 6 pr=3

=== Hybrid Scheduler ===
Sim time: 66
Total energy: 270.650
  Core 0 energy: 54.100  throughput: 0.0758 jobs/unit
  Core 1 energy: 70.450  throughput: 0.0758 jobs/unit
  Core 2 energy: 61.350  throughput: 0.1061 jobs/unit
  Core 3 energy: 84.750  throughput: 0.1212 jobs/unit
Average response time: 9.120
```

Average turnaround time: 25.920

Load stddev: 6.1033 (max load 65.00 min load 51.00)

=== Forced SJF (all cores) ===

Sim time: 74

Total energy: 272.950

  Core 0 energy: 51.000 throughput: 0.0676 jobs/unit

  Core 1 energy: 48.450 throughput: 0.0946 jobs/unit

  Core 2 energy: 87.650 throughput: 0.0811 jobs/unit

  Core 3 energy: 85.850 throughput: 0.0946 jobs/unit

Average response time: 8.480

Average turnaround time: 18.960

Load stddev: 11.5434 (max load 73.00 min load 44.00)

=== Forced RR (all cores) ===

Sim time: 71

Total energy: 271.300

  Core 0 energy: 56.000 throughput: 0.0845 jobs/unit

  Core 1 energy: 58.300 throughput: 0.0845 jobs/unit

  Core 2 energy: 66.000 throughput: 0.0845 jobs/unit

  Core 3 energy: 91.000 throughput: 0.0986 jobs/unit

Average response time: 5.600

Average turnaround time: 28.120

Load stddev: 6.7268 (max load 70.00 min load 53.00)

=== Forced Priority (all cores) ===

Sim time: 66

Total energy: 270.300

  Core 0 energy: 54.050 throughput: 0.0758 jobs/unit

  Core 1 energy: 70.400 throughput: 0.0758 jobs/unit

  Core 2 energy: 61.250 throughput: 0.1061 jobs/unit

  Core 3 energy: 84.600 throughput: 0.1212 jobs/unit

Average response time: 11.760

Average turnaround time: 24.760

Load stddev: 6.1033 (max load 65.00 min load 51.00)

=== Comparison Summary ===

Total energy: Hybrid=270.650 | SJF=272.950 | RR=271.300 | PRIO=270.300

Avg response time: Hybrid=9.120 | SJF=8.480 | RR=5.600 | PRIO=11.760

Avg turnaround time: Hybrid=25.920 | SJF=18.960 | RR=28.120 | PRIO=24.760

Load stddev: Hybrid=6.1033 | SJF=11.5434 | RR=6.7268 | PRIO=6.1033

Hybrid vs Forced Single-Core Schedulers

1. Energy

- Hybrid uses **less energy than SJF and RR**.
- Priority-only is slightly better, but difference is very small.

2. Throughput

- Hybrid = Priority → **highest throughput**.
- SJF lowest throughput.

3. Response Time

- RR is **best** (quickest response).
- Hybrid is **better than Priority**, but worse than SJF and RR.

4. Turnaround Time

- SJF is **best** (lowest completion time).
- Hybrid is worse than SJF, better than RR.

5. Load Balance

- Hybrid = Priority → **most balanced** (lowest stddev).
- SJF is worst (cores unevenly loaded).

Overall Conclusion

- **Hybrid is balanced**:
    - Good energy efficiency,
    - High throughput,
    - Excellent load balance.
- **Not best for any single metric**, but avoids extremes of SJF (bad balance), RR (high turnaround), or Priority (slow response).
- Best choice when **all factors (energy + performance + fairness)** matter together.