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;

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);

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++) {

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.