

COSC 519 Assignment 1

1. $\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n$ where τ_0 is initial predicted value, t_n actual length of nth cpu bursts, τ_{n+1} predicted value of the next cpu burst
 - a. $\alpha = 0$ and $\tau_0 = 100ms$

When $\alpha = 0$, $\tau_{n+1} = \tau_n$ Recent history does not count. Therefore, $\tau_1 = \tau_0 = 100ms$, $\tau_2 = 100ms$. The prediction is forever 100 ms regardless of actual bursts. Since SJF uses these predicted next bursts, every job looks the same length. Tie-breaking reduces it to arrival order. For example, using FCFS for scheduling.
 - b. $\alpha = 0.99$ and $\tau_0 = 10ms$

Expanding the formula shows:

$$\tau_{n+1} = \alpha t_n + (1 - \alpha)\alpha t_{n-1} + \dots + (1 - \alpha)^j \alpha t_{n-j} + \dots + (1 - \alpha)^{n+1} t_0$$

With $\alpha = 0.99$, $(1 - \alpha) = 0.01$, so the predictor is dominated by the most recent burst ($\approx 0.99t_n$) and forgets the past very quickly; the initial $\tau_0 = 10ms$ has vanishing influence after even a couple of updates. This makes the estimate highly responsive to recent value, only the actual cpu burst counts.
2. Explaining the differences in how much the following scheduling algorithms discriminate in favor of short processes:
 - a. FCFS: It does not favor short jobs. Short jobs can get stuck behind long ones; this effect is called convoy effect. FCFS is non-preemptive, so once a long job starts it runs until it blocks or finishes.
 - b. RR: It moderately favors short jobs. Each job gets at most onetime quantum before being preempted. Small jobs often finish in one- or a few-time quantum, so RR gives better response to short jobs while remaining fair. The effect strengthens as the quantum is reduced.
 - c. Multilevel Feedback queue: It strongly favors short (and I/O-bound) jobs. By demoting CPU-bound jobs to lower queues and keeping short/I-O-bound jobs near the top.
3. Code:

```
// Build:
// gcc -o assin1_3 assin1_3.c
//
// Run (file mode):
// ./assin1_3 fcfs file ex5_19.txt
// ./assin1_3 rr file ex5_19.txt 1
// ./assin1_3 pri file ex5_19.txt
//
// Run (random mode; only N is required; RR quantum if not passed default is 1):
// ./assin1_3 fcfs random 6
// ./assin1_3 rr random 6 2
```

```

//
//
// File format (one per line; priority optional):
// PID arrival burst [priority]
// P1 0 5 2

#include <stdio.h> // printf, FILE, fopen, fgets, sscanf, fclose
#include <stdlib.h> // atoi, exit, rand, srand
#include <string.h> // strcmp, snprintf
#include <ctype.h> // isspace
#include <stdbool.h> // bool
#include <time.h> // time

// constants
#define MAXP 64 // maximum number of processes supported
#define PIDLEN 32 // maximum length of a PID string like

// ----- data models -----
typedef struct {
    char pid[PIDLEN]; // process id
    int arrival; // arrival time in integer
    int burst; // total CPU time required
    int remain; // remaining time for RR
    int priority; // smaller number => higher priority
    int completion; // finish time
    int turnaround; // completion - arrival
    int waiting; // turnaround - burst
    int idx; // original input order (0..N-1)
} Proc;

typedef struct {
    Proc v[MAXP]; // storage for processes
    int n; // count (0..MAXP)
} ProcList;

// functions

/*
 * copies the PID text into the Proc struct's pid field.
 * Inputs : src[] (NUL-terminated string)
 * Output : dst is filled with up to PIDLEN-1 characters; returns nothing.
 */
static void str_copy_fixed(char dst[PIDLEN], const char src[]) {
    int j = 0;

```

```

while (src[j] && j < PIDLEN - 1)
{
    dst[j] = src[j]; j++;
}
dst[j] = '\0';
}

/*
 * compute_metrics Fill turnaround and waiting for every process in 'list'.
 * Inputs  : list with completion times already set for each process.
 * Output  : the same list, with turnaround = completion - arrival,
 *           waiting = turnaround - burst (clamped at 0).
 */
static ProcList compute_metrics(ProcList list) {
    for (int i = 0; i < list.n; ++i) {
        list.v[i].turnaround = list.v[i].completion - list.v[i].arrival;
        list.v[i].waiting    = list.v[i].turnaround - list.v[i].burst;
        if (list.v[i].waiting < 0)
        {
            list.v[i].waiting = 0;
        }
    }
    return list;
}

/*
 * sort_by_arrival_then_idx using Insertion Sort for FCFS and RR.
 * Inputs  : list unsorted or partially sorted
 * Output  : new list sorted by arrival first, then by original input order.
 */
static ProcList sort_by_arrival_then_idx(ProcList list) {
    for (int i = 1; i < list.n; ++i) {
        Proc key = list.v[i];
        int j = i - 1;
        while (j >= 0 &&
            (list.v[j].arrival > key.arrival ||
             (list.v[j].arrival == key.arrival && list.v[j].idx > key.idx))) {
            list.v[j + 1] = list.v[j];
            j--;
        }
        list.v[j + 1] = key;
    }
    return list;
}

```

```

}

/*
 * For displaying sort_by_idx using Insertion Sort to display to the original input order.
 * Inputs : list
 * Output : list sorted by idx ascending.
 */
static ProcList sort_by_idx(ProcList list) {
    for (int i = 1; i < list.n; ++i) {
        Proc key = list.v[i];
        int j = i - 1;
        while (j >= 0 && list.v[j].idx > key.idx)
        {
            list.v[j + 1] = list.v[j];
            j--;
        }
        list.v[j + 1] = key;
    }
    return list;
}

/*
 * print_table per-process metrics in the original input order
 * Inputs : title as heading, list have completion/turnaround/waiting
 * Output : Prints to stdout and returns nothing.
 */
static void print_table(const char title[], ProcList list) {
    printf("\n=== %s ===\n", title);

    ProcList view = sort_by_idx(list); // show results in original input order

    printf("\n%-6s %-7s %-6s %-9s %-10s %-7s\n",
           "PID", "Arrive", "Burst", "Complete", "Turnaround", "Wait");
    for (int i = 0; i < view.n; ++i) {
        Proc p = view.v[i];
        printf("%-6s %-7d %-6d %-9d %-10d %-7d\n",
               p.pid, p.arrival, p.burst, p.completion, p.turnaround, p.waiting);
    }
}

// queue for RR

```

```

typedef struct { int a[MAXP]; int h; int t; } Queue; // circular buffer indices
typedef struct { Queue q; int value; } Pop; // returns both queue and popped value

/*
 * q_init / q_empty / q_push / q_pop Minimal FIFO ready-queue for Round Robin.
 * Inputs : q by value, v value to push
 * Output : Updated queue and popped value.
 */
static Queue q_init(void) { Queue q; q.h = 0; q.t = 0; return q; }
static bool q_empty(Queue q) { return q.h == q.t; }
static Queue q_push(Queue q, int v) { q.a[q.t % MAXP] = v; q.t++; return q; }
static Pop q_pop (Queue q) { Pop r; r.value = q.a[q.h % MAXP]; q.h++; r.q = q; return r; }

```

// Algorithms

```

/*
 * FCFS
 * Inputs : ProcList
 * Output : ProcList with completion/turnaround/waiting.
 */
static ProcList run_fcfs(ProcList input) {
    ProcList list = input;
    for (int i = 0; i < list.n; ++i) { list.v[i].remain = list.v[i].burst; list.v[i].completion = 0; }
    list = sort_by_arrival_then_idx(list);

    int t = 0;
    for (int i = 0; i < list.n; ++i) {
        if (t < list.v[i].arrival)
        {
            t = list.v[i].arrival; // idle until next job arrives
        }
        t += list.v[i].burst; // run to completion
        list.v[i].completion = t; // record finish time
    }
    return compute_metrics(list);
}

```

```

/*
 * priority at each step, among arrived and not-done jobs, choose the one with the
 * smallest 'priority' ties earlier arrival, then smaller idx; run to completion.
 * Inputs : ProcList
 * Output : ProcList with completion/turnaround/waiting .
 */
static ProcList run_priority(ProcList input) {

```

```

ProcList list = input;
for (int i = 0; i < list.n; ++i) { list.v[i].remain = list.v[i].burst; list.v[i].completion = 0; }

bool done[MAXP]; for (int i = 0; i < MAXP; ++i) done[i] = false;
int finished = 0;
int t = 0;

while (finished < list.n) {
    // find best candidate at time t
    int best = -1;
    for (int i = 0; i < list.n; ++i) {
        if (!done[i] && list.v[i].arrival <= t) {
            if (best == -1)
            {
                best = i;
            }
            else {
                Proc a = list.v[i], b = list.v[best];
                bool better = (a.priority < b.priority) ||
                    (a.priority == b.priority && a.arrival < b.arrival) ||
                    (a.priority == b.priority && a.arrival == b.arrival && a.idx < b.idx);
                if (better)
                {
                    best = i;
                }
            }
        }
    }
    if (best == -1) { // nothing ready yet jumping to earliest future arrival
        int next_t = 1e9;
        for (int i = 0; i < list.n; ++i)
            if (!done[i] && list.v[i].arrival < next_t)
            {
                next_t = list.v[i].arrival;
            }
        t = next_t;
        continue;
    }

    t += list.v[best].burst;    // run chosen job to completion
    list.v[best].completion = t; // mark finish
    done[best] = true; finished++; // mark as done
}
return compute_metrics(list);

```

```

}

/*
 * Round Robbin with a fixed time quantum. Ready jobs are served in FIFO order each gets up to
 'quantum' time.
 * Inputs : ProcList, quantum ( if <=0, treated as 1 meaning not stated)
 * Output : ProcList with completion/turnaround/waiting filled.
 */
static ProcList run_rr(ProcList input, int quantum) {
    if (quantum <= 0)
    {
        quantum = 1; // Default
    }

    ProcList list = input;
    for (int i = 0; i < list.n; ++i)
    {
        list.v[i].remain = list.v[i].burst;
        list.v[i].completion = 0;
    }
    list = sort_by_arrival_then_idx(list);

    int t;
    //If list.n is non-zero there is at least one process then
    //set t to the first process's arrival time
    if (list.n > 0)
    {
        t = list.v[0].arrival;
    }
    else
    {
        t = 0;
    }

    int arrived = 0;    // next index to enqueue
    int finished = 0;
    Queue rq = q_init();

    // enqueue whoever has arrived by time t
    while (arrived < list.n && list.v[arrived].arrival <= t) { rq = q_push(rq, arrived); arrived++; }

    while (finished < list.n) {
        if (q_empty(rq)) { // no ready jobs: jump to next arrival
            if (arrived < list.n) {

```

```

        t = list.v[arrived].arrival;
        while (arrived < list.n && list.v[arrived].arrival <= t)
        {
            rq = q_push(rq, arrived); arrived++;
        }
        continue;
    }
    else
    {
        break;
    }
}
Pop pr = q_pop(rq); rq = pr.q;
int i = pr.value;

int slice;
if (list.v[i].remain < quantum)
{
    slice = list.v[i].remain;
}
else
{
    slice = quantum;
}
t += slice;
list.v[i].remain -= slice;

// enqueue any processes that arrived during this slice
while (arrived < list.n && list.v[arrived].arrival <= t)
{
    rq = q_push(rq, arrived); arrived++;
}

if (list.v[i].remain == 0) {
    list.v[i].completion = t;    // finished
    finished++;
}
else {
    rq = q_push(rq, i);        // back of the queue
}
}
return compute_metrics(list);
}

```

```

// input (file + random)

/*
 * read_file_into_list Load processes from a text file each line PID arrival burst priority.
 * Inputs : path
 * Output : ProcList with pid/arrival/burst/priority/idx set and n is the count loaded.
 * Lines starting with '#' or blank are ignored.
 */
static ProcList read_file_into_list(const char path[]) {
    ProcList list; list.n = 0;

    FILE* fp = fopen(path, "r"); // library FILE*
    if (!fp) { perror("fopen"); exit(1); }

    char line[256];
    while (fgets(line, sizeof(line), fp)) {
        // skip comment lines
        bool blank = true;
        for (int i = 0; line[i]; ++i)
            if (!isspace((unsigned char)line[i]))
                {
                    blank = false; break;
                }
        if (blank || line[0] == '#')
            {
                continue;
            }

        char pid[PIDLEN]; int a, b, pri = 0;
        int k = sscanf(line, "%31s %d %d %d", pid, &a, &b, &pri);
        if (k >= 3) { // require pid, arrival, burst
            int i = list.n;
            str_copy_fixed(list.v[i].pid, pid);
            list.v[i].arrival = a;
            list.v[i].burst = b;
            // default 0 if omitted
            if (k >= 4)
                {
                    list.v[i].priority = pri;
                }
            else
                {
                    list.v[i].priority = 0;
                }
        }
    }
}

```

```

        list.v[i].idx    = i; // remembering input order
        list.n++;
        if (list.n >= MAXP)
        {
            break;
        }
    }
}
fclose(fp);
return list;
}

/*
 * randint Return a random integer in [lo, hi].
 * Inputs  : lo, hi assume lo <= hi
 * Output  : int in range.
 */
static int randint(int lo, int hi) { return lo + (rand() % (hi - lo + 1)); }

/*
 * gen_random_list N processes with random arrival/burst/priority; PIDs P1..PN.
 * Inputs  : N (1..20).
 * Output  : ProcList with fields filled; idx = input order (0..N-1).
 */
static ProcList gen_random_list(int N) {
    ProcList list; list.n = 0;
    if (N <= 0 || N > 20)
    {
        printf("Choose N in 1 to 20\n"); exit(1);
    }

    // Seed RNG with current time
    static bool seeded = false;
    if (!seeded)
    {
        srand((unsigned)time(NULL));
        seeded = true;
    }

    for (int i = 0; i < N; ++i) {
        char pid[PIDLEN];
        (void)snprintf(pid, sizeof(pid), "P%d", i + 1);
        str_copy_fixed(list.v[i].pid, pid);
    }
}

```

```

        list.v[i].arrival = randint(0, 10); // fixed range per spec
        list.v[i].burst  = randint(1, 20); // fixed range per spec
        list.v[i].priority = randint(1, 5); // 1 = highest
        list.v[i].idx    = i;
        list.n++;
    }
    return list;
}

/*
 * Show the randomly generated processes before scheduling.
 * Inputs : list
 * Output : Prints PID, arrival, burst, priority.
 */
static void print_generated_list(ProcList list) {
    printf("Generated processes (PID arrival burst priority):\n");
    for (int i = 0; i < list.n; ++i) {
        printf("%-6s %7d %6d %8d\n",
            list.v[i].pid, list.v[i].arrival, list.v[i].burst, list.v[i].priority);
    }
}

int main(int argc, char* argv[]) {
    if (argc < 4) {
        printf("Usage:\n");
        printf(" %s <fcfs|rr|pri> file <path> [quantum]\n", argv[0]);
        printf(" %s <fcfs|rr|pri> random <N> [quantum]\n", argv[0]);
        return 1;
    }

    const char* alg = argv[1];
    const char* mode = argv[2];

    ProcList input; input.n = 0;
    int quantum = 1;

    if (strcmp(mode, "file") == 0) {
        const char* path = argv[3];
        input = read_file_into_list(path);
        if (argc >= 5)
        {
            quantum = atoi(argv[4]); // used by RR

```

```

    }
} else if (strcmp(mode, "random") == 0) {
    int N = atoi(argv[3]);           // only N is required now
    if (argc >= 5) quantum = atoi(argv[4]); // optional: RR quantum otherwise default is 1
    input = gen_random_list(N);
    print_generated_list(input);
} else {
    printf("Unknown mode: %s (use 'file' or 'random')\n", mode);
    return 1;
}

if (strcmp(alg, "fcfs") == 0) {
    ProcList r = run_fcfs(input);
    print_table("FCFS", r);

} else if (strcmp(alg, "rr") == 0) {
    ProcList r = run_rr(input, quantum);
    char title[64]; snprintf(title, sizeof(title), "Round Robin (q=%d)", quantum);
    print_table(title, r);

} else if (strcmp(alg, "pri") == 0) {
    ProcList r = run_priority(input);
    print_table("Priority (non-preemptive, lower number = higher)", r);

} else {
    printf("Unknown algorithm: %s (use fcfs|rr|pri)\n", alg);
    return 1;
}

return 0;
}

```

Output:

```

tiger@tucis-ubuntu:~/myDir/assign1$
tiger@tucis-ubuntu:~/myDir/assign1$ vi assign1_3.c
tiger@tucis-ubuntu:~/myDir/assign1$
tiger@tucis-ubuntu:~/myDir/assign1$ vi input.txt
tiger@tucis-ubuntu:~/myDir/assign1$ gcc -o assign1_3 assign1_3.c
tiger@tucis-ubuntu:~/myDir/assign1$ ./assign1_3 fcfs file input.txt

```

=== FCFS ===

PID	Arrive	Burst	Complete	Turnaround	Wait
P1	0	10	10	10	0
P2	0	1	11	11	10
P3	0	2	13	13	11
P4	0	1	14	14	13
P5	0	5	19	19	14

```

tiger@tucis-ubuntu:~/myDir/assign1$ ./assign1_3 rr file input.txt 1

```

=== Round Robin (q=1) ===

PID	Arrive	Burst	Complete	Turnaround	Wait
P1	0	10	19	19	9
P2	0	1	2	2	1
P3	0	2	7	7	5
P4	0	1	4	4	3
P5	0	5	14	14	9

```

tiger@tucis-ubuntu:~/myDir/assign1$ ./assign1_3 pri file input.txt

```

=== Priority (non-preemptive, lower number = higher) ===

PID	Arrive	Burst	Complete	Turnaround	Wait
P1	0	10	16	16	6
P2	0	1	1	1	0
P3	0	2	18	18	16
P4	0	1	19	19	18
P5	0	5	6	6	1

```

tiger@tucis-ubuntu:~/myDir/assign1$ ./assign1_3 fcfs random 5

```

Generated processes (PID arrival burst priority):

PID	Arrive	Burst	Priority
P1	4	9	2
P2	1	8	3
P3	2	7	3
P4	4	17	5
P5	7	3	1

=== FCFS ===

PID	Arrive	Burst	Complete	Turnaround	Wait
P1	4	9	25	21	12
P2	1	8	9	8	0
P3	2	7	16	14	7
P4	4	17	42	38	21
P5	7	3	45	38	35

```

tiger@tucis-ubuntu:~/myDir/assign1$ ./assign1_3 rr random 5 1

```

Generated processes (PID arrival burst priority):

PID	Arrive	Burst	Priority
P1	6	17	5
P2	0	6	3
P3	6	20	1
P4	0	16	1
P5	5	5	2

=== Round Robin (q=1) ===

PID	Arrive	Burst	Complete	Turnaround	Wait
P1	6	17	60	54	37
P2	0	6	18	18	12
P3	6	20	64	58	38
P4	0	16	53	53	37
P5	5	5	26	21	16

```

tiger@tucis-ubuntu:~/myDir/assign1$ ./assign1_3 pri random 5

```

Generated processes (PID arrival burst priority):

PID	Arrive	Burst	Priority
P1	1	1	1
P2	7	17	2
P3	2	19	2
P4	6	13	2
P5	9	12	4

=== Priority (non-preemptive, lower number = higher) ===

PID	Arrive	Burst	Complete	Turnaround	Wait
P1	1	1	2	1	0
P2	7	17	51	44	27
P3	2	19	21	19	0
P4	6	13	34	28	15
P5	9	12	63	54	42

```

tiger@tucis-ubuntu:~/myDir/assign1$ ./assign1_3 pri random 23

```

Choose N in 1 to 20