

Міністерство освіти і науки України

Національний технічний університет України

«Київський політехнічний інститут»

**Лабораторна робота №1**

***з дисципліни «Введення до операційних систем»***

**«Планування процесів»**

Виконав студент групи: КВ-11

ПІБ: Терентьєв Іван Дмитрович

Перевірив: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Київ 2024**

*Індивідуальне завдання за варіантом 23(8)*

Багаторівневі черги (2 рівні):

1. черга інтерактивних процесів – алгоритм RR
2. черга фонових процесів – алгоритм SJF

Час розподіляється між чергою фонових і чергою інтерактивних процесів. Наприклад, для інтерактивних процесів – 80 % і для фонових – 20 % часу. Якщо при досягненні ліміту часу виконується фоновий процес, він переривається. Його продовження відбуватиметься при наступному переході на обслуговування черги фонових процесів.

*Код програми:*

[*cpu.c* 4](#_Toc160622987)

[*cpu.h* 5](#_Toc160622988)

[*list\_moves.c* 6](#_Toc160622989)

[*list\_moves.h* 7](#_Toc160622990)

[*main.c* 8](#_Toc160622991)

[*process.c* 10](#_Toc160622992)

[*process.h* 11](#_Toc160622993)

[*queue\_basic.c* 12](#_Toc160622994)

[*queue\_basic.h* 12](#_Toc160622995)

[*queue\_rr.c* 13](#_Toc160622996)

[*queue\_rr.h* 13](#_Toc160622997)

[*queue\_sjf.c* 14](#_Toc160622998)

[*queue\_sjf.h* 14](#_Toc160622999)

[*scheduler.c* 15](#_Toc160623000)

[*scheduler.h* 17](#_Toc160623001)

[*scheduler\_struct.h* 17](#_Toc160623002)

*cpu.c*

#include "cpu.h"

#include "stdbool.h"

void create\_CPU(CPU \*cpu, float CLK) {

cpu->process = NULL;

cpu->curr\_tick = 0;

cpu->curr\_interactive\_tick = 0;

cpu->curr\_background\_tick = 0;

cpu->CLK = CLK;

}

void set\_proc(CPU \*cpu, Process \*process) { cpu->process = process; }

unsigned short int exec(CPU \*cpu) {

cpu->curr\_tick += cpu->CLK;

if (cpu->process->interactivity) {

cpu->curr\_interactive\_tick += cpu->CLK;

} else {

cpu->curr\_background\_tick += cpu->CLK;

}

execute(cpu->process, cpu->CLK);

if (cpu->process->curr\_state == FINISHED) {

cpu->process->end\_time = cpu->curr\_tick;

cpu->process->wait\_time = cpu->process->end\_time -

cpu->process->start\_time -

cpu->process->work\_time;

return FINISHED;

}

return WAITING;

}

*cpu.h*

#include "process.h"

#include <stdbool.h>

#ifndef CPU\_H

#define CPU\_H

struct cpu {

Process \*process;

float curr\_tick;

float curr\_interactive\_tick;

float curr\_background\_tick;

bool running;

float CLK;

};

typedef struct cpu CPU;

void create\_CPU(CPU \*cpu, float CLK);

void set\_proc(CPU \*cpu, Process \*process);

unsigned short int exec(CPU \*cpu);

#endif

*list\_moves.c*

#include "list\_moves.h"

void resize\_list(SCH \*sch, unsigned short int to\_list, size\_t count) {

switch (to\_list) {

case RR\_Q:

sch->interactive.procs =

(Process \*)realloc(sch->interactive.procs, sizeof(Process) \* count);

break;

case RR\_Q\_DONE:

sch->interactive.completed\_procs = (Process \*)realloc(

sch->interactive.completed\_procs, sizeof(Process) \* count);

break;

case SJF\_Q:

sch->background.procs =

(Process \*)realloc(sch->background.procs, sizeof(Process) \* count);

break;

case SJF\_Q\_DONE:

sch->background.completed\_procs = (Process \*)realloc(

sch->background.completed\_procs, sizeof(Process) \* count);

break;

}

}

void add\_process\_to\_list(SCH \*sch, Process process,

unsigned short int to\_list) {

switch (to\_list) {

case RR\_Q:

sch->interactive.procs\_count++;

resize\_list(sch, to\_list, sch->interactive.procs\_count);

sch->interactive.procs[sch->interactive.procs\_count - 1] = process;

break;

case RR\_Q\_DONE:

sch->interactive.completed\_procs\_count++;

resize\_list(sch, to\_list, sch->interactive.completed\_procs\_count);

sch->interactive

.completed\_procs[sch->interactive.completed\_procs\_count - 1] = process;

break;

case SJF\_Q:

sch->background.procs\_count++;

resize\_list(sch, to\_list, sch->background.procs\_count);

sch->background.procs[sch->background.procs\_count - 1] = process;

break;

case SJF\_Q\_DONE:

sch->background.completed\_procs\_count++;

resize\_list(sch, to\_list, sch->background.completed\_procs\_count);

sch->background.completed\_procs[sch->background.completed\_procs\_count - 1] =

process;

break;

};

}

void clean\_from\_finished(SCH \*sch, unsigned short int to\_list) {

if (to\_list == RR\_Q) {

for (size\_t i = 0; i < sch->interactive.procs\_count; i++) {

if (sch->interactive.procs[i].curr\_state == FINISHED) {

for (size\_t k = i + 1; k < sch->interactive.procs\_count; k++) {

sch->interactive.procs[k - 1] = sch->interactive.procs[k];

}

sch->interactive.procs\_count--;

Process \*new\_procs =

(Process \*)malloc(sch->interactive.procs\_count \* sizeof(Process));

for (size\_t k = 0; k < sch->interactive.procs\_count; k++) {

new\_procs[k] = sch->interactive.procs[k];

}

}

}

} else {

for (size\_t i = 0; i < sch->background.procs\_count; i++) {

if (sch->background.procs[i].curr\_state == FINISHED) {

for (size\_t k = i + 1; k < sch->background.procs\_count; k++) {

sch->background.procs[k - 1] = sch->background.procs[k];

}

sch->background.procs\_count--;

Process \*new\_procs =

(Process \*)malloc(sch->background.procs\_count \* sizeof(Process));

for (size\_t k = 0; k < sch->background.procs\_count; k++) {

new\_procs[k] = sch->background.procs[k];

}

}

}

}

}

*list\_moves.h*

#ifndef LIST\_MOVES\_H

#define LIST\_MOVES\_H

#include "process.h"

#include "scheduler\_struct.h"

void resize\_list(SCH \*sch, unsigned short int to\_list, size\_t count);

void add\_process\_to\_list(SCH \*sch, Process process, unsigned short int to\_list);

void clean\_from\_finished(SCH \*sch, unsigned short int to\_list);

#endif

*main.c*

#include "printer.h"

#include "process.h"

#include "scheduler.h"

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

struct param {

float min\_i;

float max\_i;

float min\_b;

float max\_b;

float chance\_i;

float util\_time\_i;

float expected\_f;

size\_t count;

bool skip;

float clk;

};

typedef struct param Param;

int main(int argc, char \*argv[]) {

Param params = {3.0, 10.0, 2.0, 5.0, 0.5, (float)0.8, 1.5, 10, false, 1.0};

for (int i = 1; i < argc; i++) {

if (strcmp(argv[i], "-mini") == 0 && i + 1 < argc) {

params.min\_i = (float)atof(argv[i + 1]);

i++;

} else if (strcmp(argv[i], "-maxi") == 0 && i + 1 < argc) {

params.max\_i = (float)atof(argv[i + 1]);

i++;

} else if (strcmp(argv[i], "-minb") == 0 && i + 1 < argc) {

params.min\_b = (float)atof(argv[i + 1]);

i++;

} else if (strcmp(argv[i], "-maxb") == 0 && i + 1 < argc) {

params.max\_b = (float)atof(argv[i + 1]);

i++;

} else if (strcmp(argv[i], "-chancei") == 0 && i + 1 < argc) {

params.chance\_i = (float)atof(argv[i + 1]);

i++;

} else if (strcmp(argv[i], "-utilk") == 0 && i + 1 < argc) {

params.util\_time\_i = (float)atof(argv[i + 1]);

i++;

} else if (strcmp(argv[i], "-expf") == 0 && i + 1 < argc) {

params.expected\_f = (float)atof(argv[i + 1]);

i++;

} else if (strcmp(argv[i], "-clk") == 0 && i + 1 < argc) {

params.clk = (float)atof(argv[i + 1]);

i++;

} else if (strcmp(argv[i], "-c") == 0 && i + 1 < argc) {

char \*endptr;

params.count = strtoul(argv[i + 1], &endptr, 10);

if (\*endptr != '\0') {

exit(-1);

}

i++;

} else if (strcmp(argv[i], "--help") == 0 || strcmp(argv[i], "-h") == 0) {

printf(

"-mini <float>\t\t- Minimal interactive proccess remaining time\n");

printf(

"-maxi <float>\t\t- Maximum interactive proccess remaining time\n");

printf("-minb <float>\t\t- Minimal background proccess remaining time\n");

printf("-maxb <float>\t\t- Maximum background proccess remaining time\n");

printf("-expf <float>\t\t- Jitter for min and max proccess remaining "

"time\n");

printf(

"-chancei <float(0 to 1)>\t\t- Chance of born interactive process\n");

printf(

"-utilk <float(0 to 1)>\t\t- Target utilization time proportion\n");

printf("-clk <float>\t\t- Clock speed k\n");

printf("-c <size\_t>\t\t- Count of proccesses\n");

printf("-v \t\t- Version\n");

printf("-h or --help \t\t- This help\n");

params.skip = true;

} else if (strcmp(argv[i], "-v") == 0) {

printf("0.1 Release by @t3ry4|Terentiev I.D. KV-11|14\n");

params.skip = true;

}

}

if (!params.skip) {

SCH sch1;

create\_SCH(&sch1, params.count, params.util\_time\_i, params.min\_i,

params.max\_i, params.min\_b, params.max\_b, params.expected\_f,

params.chance\_i, params.clk);

run\_SCH(&sch1);

print\_completed(&sch1);

}

return 0;

}

*process.c*

#include "process.h"

#include "scheduler\_struct.h"

#include <stdlib.h>

size\_t last\_id = 1;

void create\_process(Process \*process, float expected\_time, float remaining\_time,

bool interactivity, float start\_time, float end\_time,

unsigned short int state) {

if (state != NOT\_CREATED)

process->id = last\_id++;

else

process->id = 0;

process->expected\_time = expected\_time;

process->remaining\_time = remaining\_time;

process->interactivity = interactivity;

process->start\_time = start\_time;

process->end\_time = end\_time;

process->curr\_state = state;

process->work\_time = 0;

process->wait\_time = 0;

}

void execute(Process \*process, float CLK) {

if (process->remaining\_time > 0) {

process->remaining\_time -= CLK;

process->work\_time += CLK;

}

if (process->remaining\_time <= 0) {

process->remaining\_time = 0;

process->curr\_state = FINISHED;

}

}

*process.h*

#include <stdbool.h>

#include <stdlib.h>

#ifndef PROCESS\_H

#define PROCESS\_H

#define WAITING 0

#define WORKING 1

#define FINISHED 2

#define NOT\_CREATED 3

#define RR\_Q 0

#define RR\_Q\_DONE 1

#define SJF\_Q 3

#define SJF\_Q\_DONE 4

struct process {

size\_t id;

float expected\_time;

float remaining\_time;

bool interactivity;

float start\_time;

float end\_time;

float work\_time;

float wait\_time;

unsigned short int curr\_state;

};

typedef struct process Process;

void create\_process(Process \*process, float expected\_time, float remaining\_time,

bool interactivity, float start\_time, float end\_time,

unsigned short int state);

void execute(Process \*process, float CLK);

#endif

*queue\_basic.c*

#include "queue\_basic.h"

#include "list\_moves.h"

void create\_queue(Queue \*q) {

q->procs = NULL;

q->procs\_count = 0;

q->completed\_procs = NULL;

q->completed\_procs\_count = 0;

q->next\_proc\_index = 0;

}

*queue\_basic.h*

#include "process.h"

#include <stdlib.h>

#ifndef QUEUE\_BASIC

#define QUEUE\_BASIC

struct queue {

Process \*procs;

size\_t procs\_count;

Process \*completed\_procs;

size\_t completed\_procs\_count;

size\_t next\_proc\_index;

};

typedef struct queue Queue;

void create\_queue(Queue \*q);

#endif

*queue\_rr.c*

#include "queue\_rr.h"

Process get\_next\_proc\_rr(SCH \*sch) {

if (sch->interactive.procs\_count != 0) {

clean\_from\_finished(sch, RR\_Q);

if (sch->interactive.next\_proc\_index < sch->interactive.procs\_count - 1) {

sch->interactive.next\_proc\_index++;

} else

sch->interactive.next\_proc\_index = 0;

return sch->interactive.procs[sch->interactive.next\_proc\_index];

} else {

Process process;

create\_process(&process, 0, 0, 0, 0, 0, NOT\_CREATED);

return process;

}

}

*queue\_rr.h*

#include "list\_moves.h"

#ifndef QUEUE\_RR

#define QUEUE\_RR

Process get\_next\_proc\_rr(SCH \*sch);

#endif

*queue\_sjf.c*

#include "queue\_sjf.h"

Process get\_next\_proc\_sjf(SCH \*sch) {

if (sch->background.procs\_count != 0) {

if (sch->background.procs[sch->background.next\_proc\_index].curr\_state ==

FINISHED) {

clean\_from\_finished(sch, SJF\_Q);

float min = sch->background.procs[0].expected\_time;

sch->background.next\_proc\_index = 0;

for (size\_t i = 0; i < sch->background.procs\_count; i++) {

if (min > sch->background.procs[i].expected\_time) {

min = sch->background.procs[i].expected\_time;

sch->background.next\_proc\_index = i;

}

}

}

return sch->background.procs[sch->background.next\_proc\_index];

} else {

Process process;

create\_process(&process, 0, 0, 0, 0, 0, NOT\_CREATED);

return process;

}

}

*queue\_sjf.h*

#include "list\_moves.h"

#ifndef QUEUE\_SJF

#define QUEUE\_SJF

Process get\_next\_proc\_sjf(SCH \*sch);

#endif

*scheduler.c*

#include "scheduler.h"

#include "printer.h"

#include "queue\_rr.h"

#include "queue\_sjf.h"

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

// #define VERBOSE

// uncomment for real-time

float randfrom(float min, float max) {

float range = (max - min);

float div = (float)RAND\_MAX / range;

return min + ((float)rand() / div);

}

void create\_SCH(SCH \*sch, size\_t tasks\_count, float k\_interactive, float minI,

float maxI, float minB, float maxB, float ExpectedFloating,

float ChanceOfInteractive, float CLK) {

CPU new\_cpu;

create\_CPU(&new\_cpu, CLK);

srand((unsigned int)time(0));

sch->cpu = new\_cpu;

sch->tasks\_count = tasks\_count;

sch->k\_interactive = k\_interactive;

create\_queue(&sch->interactive);

create\_queue(&sch->background);

sch->tasks\_count\_backup = tasks\_count;

sch->min\_r\_gen\_rr = minI;

sch->max\_r\_gen\_rr = maxI;

sch->min\_r\_gen\_sjf = minB;

sch->max\_r\_gen\_sjf = maxB;

sch->e\_float = ExpectedFloating;

sch->chance\_of\_i = ChanceOfInteractive;

}

void create\_new\_task(SCH \*sch) {

Process process;

float remaining\_time = 0;

float expected\_time = 0;

if (randfrom((float)0.01, 1) >= sch->chance\_of\_i) {

remaining\_time = randfrom(sch->min\_r\_gen\_rr, sch->max\_r\_gen\_rr);

expected\_time =

remaining\_time + randfrom(sch->e\_float \* (float)(-1.0), sch->e\_float);

create\_process(&process, expected\_time, remaining\_time, true,

sch->cpu.curr\_tick, 0, WAITING);

add\_process\_to\_list(sch, process, RR\_Q);

} else {

remaining\_time = randfrom(sch->min\_r\_gen\_sjf, sch->max\_r\_gen\_sjf);

expected\_time =

remaining\_time + randfrom(sch->e\_float \* (float)(-1.0), sch->e\_float);

create\_process(&process, expected\_time, remaining\_time, false,

sch->cpu.curr\_tick, 0, WAITING);

add\_process\_to\_list(sch, process, SJF\_Q);

}

sch->tasks\_count--;

}

void choose\_queue(SCH \*sch) {

if (sch->interactive.procs\_count != 0 || sch->background.procs\_count != 0) {

Process next\_interactive = get\_next\_proc\_rr(sch);

Process next\_background = get\_next\_proc\_sjf(sch);

// interactive ticks / sum of interactive ticks and background ticks >=

// interactive koef

if (GET\_K\_INTERACTIVE(sch->cpu.curr\_interactive\_tick,

sch->cpu.curr\_background\_tick) >=

sch->k\_interactive) {

if (next\_background.curr\_state != NOT\_CREATED)

sch->cpu.process =

&sch->background.procs[sch->background.next\_proc\_index];

else

sch->cpu.process =

&sch->interactive.procs[sch->interactive.next\_proc\_index];

} else {

if (next\_interactive.curr\_state != NOT\_CREATED)

sch->cpu.process =

&sch->interactive.procs[sch->interactive.next\_proc\_index];

else

sch->cpu.process =

&sch->background.procs[sch->background.next\_proc\_index];

}

} else {

Process process;

create\_process(&process, 0, 0, 0, 0, 0, NOT\_CREATED);

sch->cpu.process = &process;

}

}

void run\_SCH(SCH \*sch) {

while (1) {

if (sch->tasks\_count > 0)

create\_new\_task(sch);

else if (sch->interactive.completed\_procs\_count +

sch->background.completed\_procs\_count ==

sch->tasks\_count\_backup)

break;

choose\_queue(sch);

if (sch->cpu.process->curr\_state != NOT\_CREATED &&

sch->cpu.process->curr\_state != FINISHED) {

sch->cpu.process->curr\_state = WORKING;

if (exec(&sch->cpu) == FINISHED) {

if (sch->cpu.process->interactivity) {

#ifdef VERBOSE

print\_element(

&sch->interactive.procs[sch->interactive.next\_proc\_index]);

#endif

add\_process\_to\_list(

sch, sch->interactive.procs[sch->interactive.next\_proc\_index],

RR\_Q\_DONE);

} else {

#ifdef VERBOSE

print\_element(

&sch->background.procs[sch->background.next\_proc\_index]);

#endif

add\_process\_to\_list(

sch, sch->background.procs[sch->background.next\_proc\_index],

SJF\_Q\_DONE);

}

sch->cpu.process = NULL;

} else

sch->cpu.process->curr\_state = WAITING;

}

}

}

*scheduler.h*

#ifndef SCHEDULER\_H

#define SCHEDULER\_H

#define RAND\_BACKGROUND\_TASK\_GENERATOR\_PROBABILITY 0.5

#define INTERACTIVE\_BOOL 1

#define BACKGROUND\_BOOL 0

#define GET\_K\_INTERACTIVE(x, y) x / (x + y)

#include "list\_moves.h"

void create\_SCH(SCH \*sch, size\_t tasks\_count, float k\_interactive, float minI,

float maxI, float minB, float maxB, float ExpectedFloating,

float ChanceOfInteractive, float CLK);

void add\_to\_completed(SCH \*sch, Process \*process);

void run\_SCH(SCH \*sch);

#endif

*scheduler\_struct.h*

#include "cpu.h"

#include "queue\_basic.h"

#include <stdlib.h>

#ifndef SCHEDULER\_STRUCT\_H

#define SCHUDULER\_STRUCT\_H

struct scheduler {

size\_t tasks\_count;

size\_t tasks\_count\_backup;

float k\_interactive;

Queue interactive;

Queue background;

CPU cpu;

float min\_r\_gen\_rr;

float min\_r\_gen\_sjf;

float max\_r\_gen\_rr;

float max\_r\_gen\_sjf;

float e\_float;

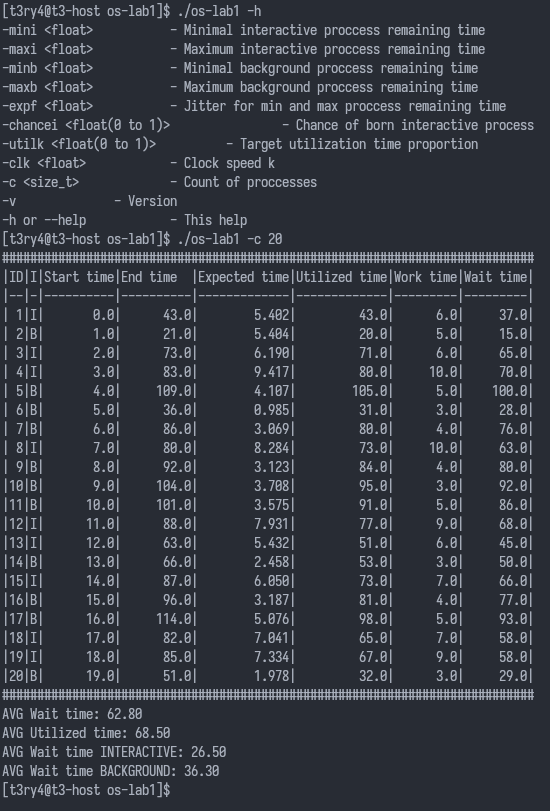
float chance\_of\_i;

};

typedef struct scheduler SCH;

#endif

*Скріншот програми:*

**

*Висновок:*

Під час виконання роботи ознайомились з основними алгоритмами планування процесів в операційних системах. Було проведено моделювання роботи планувальника за допомогою побудови програмної моделі за заданим алгоритмом. В моєму випадку була багаторівнева черга, що складалась з двох рівнів, черга для інтерактивних процесів та черга для фонових процесів. Для черги інтерактивних процесів використовувався алгоритм Round Robin – кругове(карусельне) планування, а для фонових алгоритм SJF – найкоротша робота виконується першою. Авжеж черга інтерактивних процесів була пріоритетною, та забирала на себе ініціативу від фонових, коли фонові забирали більше ніж 20% процесорного часу, згідно з варіантом, перериваючи виконання фонового процесу. Моделювання показало, що черга інтерактивних процесів має менший середній час очікування, коли кількість інтерактивних процесів дорівнює, або менше кількості фонових процесів, але якщо інтерактивних процесів більше ніж фонових, то через особливості алгоритму середній час очікування сильно збільшується у інтерактивних процесів, бо вони кожний такт процеси виконуються одним за одним і при збільшенні кількості процесів сильно збільшується час очікування, в свою чергу алгоритм для фонових процесів в середньому має доволі низький час очікування, але тільки у випадку коли процеси мають короткий час виконання, також для реалізації такого алгоритму потрібно, щоб ОС максимально точно розраховувала очікуваний час виконання. Поєднання цих двох алгоритмів дає можливість інтерактивним процесам мати мінімальний час відгуку притому одночасно усіх інтерактивних процесів, а фоновим мати мінімальний середній час очікування. Розподіл процесорного часу 80/20 між інтерактивними та фоновими процесами не дає фоновим процесам «голодувати» та зберігає більшу частину ініціативи при інтерактивних процесах.