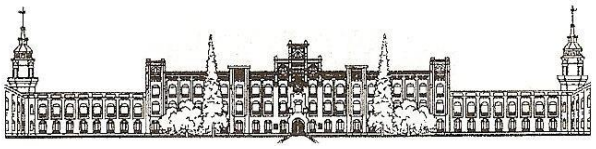

Міністерство освіти і науки України
Національний технічний університет України
«Київський політехнічний інститут»

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

```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

```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

```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

```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

```c
#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

```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:

```
[t3ry4@t3-host os-lab1]$ ./os-lab1 -h
-mini <float>           - Minimal interactive proccess remaining time
-maxi <float>           - Maximum interactive proccess remaining time
-minb <float>           - Minimal background proccess remaining time
-maxb <float>           - Maximum background proccess remaining time
-expf <float>           - Jitter for min and max proccess remaining time
-chancei <float(0 to 1)>                - Chance of born interactive process
-utilk <float(0 to 1)>          - Target utilization time proportion
-clk <float>            - Clock speed k
-c <size_t>             - Count of proccesses
-v              - Version
-h or --help            - This help
[t3ry4@t3-host os-lab1]$ ./os-lab1 -c 20
################################################################################
|ID|I|Start time|End time  |Expected time|Utilized time|Work time|Wait time|
|--|-|----------|----------|-------------|-------------|---------|---------|
| 1|I|       0.0|      43.0|        5.402|         43.0|      6.0|     37.0|
| 2|B|       1.0|      21.0|        5.404|         20.0|      5.0|     15.0|
| 3|I|       2.0|      73.0|        6.190|         71.0|      6.0|     65.0|
| 4|I|       3.0|      83.0|        9.417|         80.0|     10.0|     70.0|
| 5|B|       4.0|     109.0|        4.107|        105.0|      5.0|    100.0|
| 6|B|       5.0|      36.0|        0.985|         31.0|      3.0|     28.0|
| 7|B|       6.0|      86.0|        3.069|         80.0|      4.0|     76.0|
| 8|I|       7.0|      80.0|        8.284|         73.0|     10.0|     63.0|
| 9|B|       8.0|      92.0|        3.123|         84.0|      4.0|     80.0|
|10|B|       9.0|     104.0|        3.708|         95.0|      3.0|     92.0|
|11|B|      10.0|     101.0|        3.575|         91.0|      5.0|     86.0|
|12|I|      11.0|      88.0|        7.931|         77.0|      9.0|     68.0|
|13|I|      12.0|      63.0|        5.432|         51.0|      6.0|     45.0|
|14|B|      13.0|      66.0|        2.458|         53.0|      3.0|     50.0|
|15|I|      14.0|      87.0|        6.050|         73.0|      7.0|     66.0|
|16|B|      15.0|      96.0|        3.187|         81.0|      4.0|     77.0|
|17|B|      16.0|     114.0|        5.076|         98.0|      5.0|     93.0|
|18|I|      17.0|      82.0|        7.041|         65.0|      7.0|     58.0|
|19|I|      18.0|      85.0|        7.334|         67.0|      9.0|     58.0|
|20|B|      19.0|      51.0|        1.978|         32.0|      3.0|     29.0|
################################################################################
AVG Wait time: 62.80
AVG Utilized time: 68.50
AVG Wait time INTERACTIVE: 26.50
AVG Wait time BACKGROUND: 36.30
[t3ry4@t3-host os-lab1]$
```

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.