Національний технічний університет України

«Київський політехнічний інститут»

Факультет інформатики і обчислювальної техніки

Лабораторна робота №3

З ООП

*Виконав:*

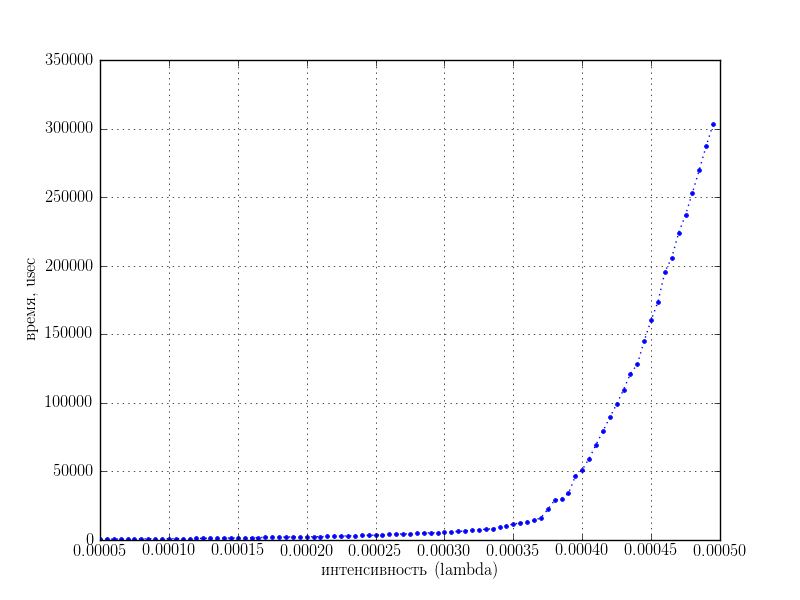
Студент групи ІC-73

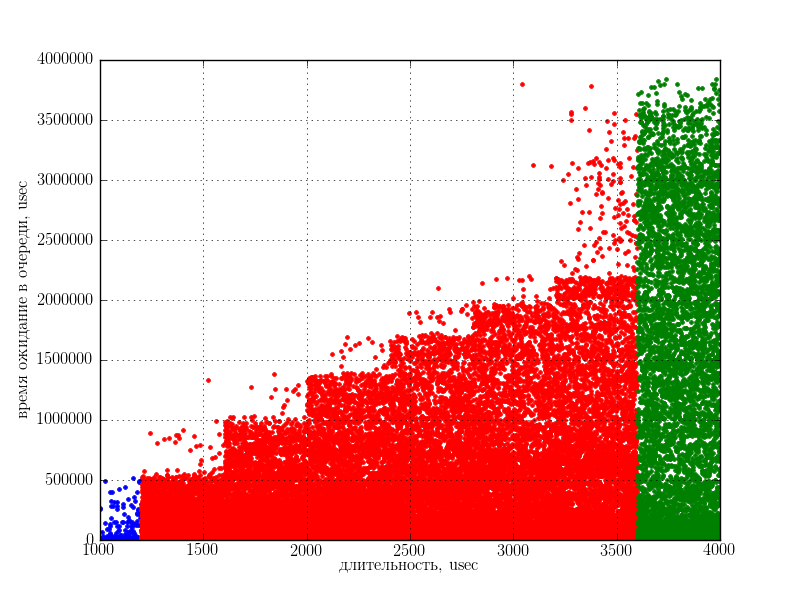
Глухов Никита

м. Київ 2020

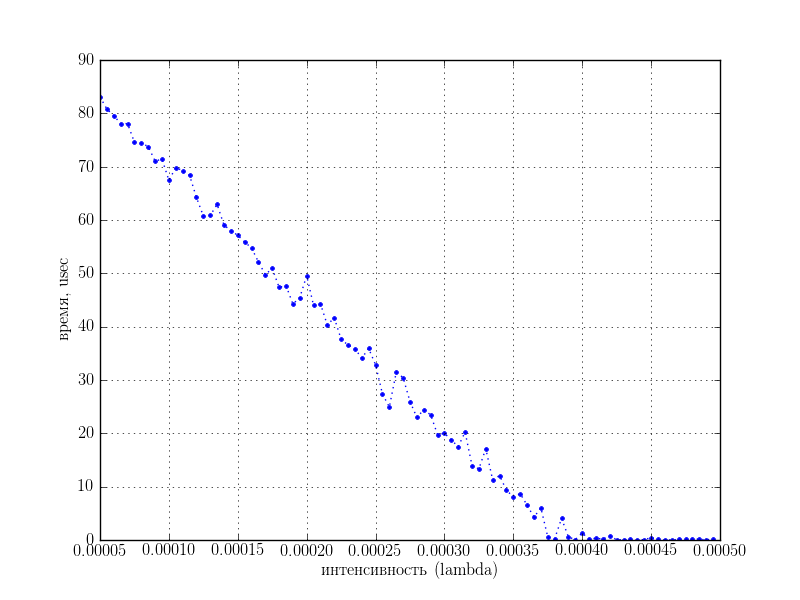
16. Змiшаний алгоритм. FIFO + АП

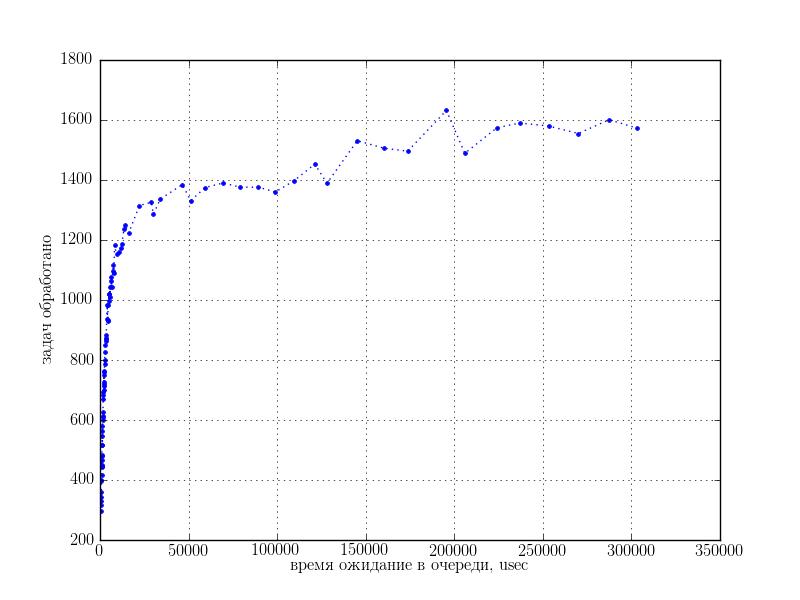
Час чекання в черзi вiд iнтенсивностi:



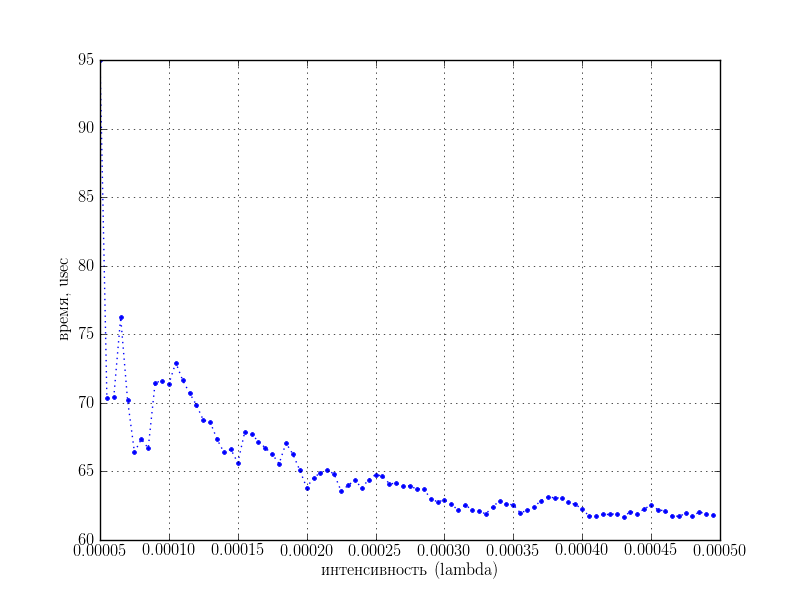


Час чекання в черзi вiд довжини задачi:

Час простою вiд iнтенсивностi заявок:

Кiлькiсть опрацьованих задач вiд часу чекання:

Середнiй час очiкування задач абсолютного приорiтету в черзi вiд iнтенсивностi

заявок:

**Код програми**

import random

import logging

import math

from pylab import arange

logging.basicConfig(level=logging.INFO)

def log(name, text):

logging.getLogger(name).debug(text)

TASK\_NEW, TASK\_PROGRESS, TASK\_SUSPENDED, TASK\_FINISHED, TASK\_KILLED = range(5)

class Task(object):

ALL\_TASKS = []

ALL\_TASKS\_AP = []

def \_\_init\_\_(self, env, duration, ap):

self.env = env

self.duration = duration

self.creation\_time = env.time

self.state = TASK\_NEW

self.ap = ap

self.last\_work\_time = self.creation\_time

self.waiting\_time = 0

if ap:

log('tsk', "[%d] task produced %ds ABSOLUTE PRIORITY" % (env.time, self.duration))

else:

log('tsk', "[%d] task produced %ds" % (env.time, self.duration))

if ap:

Task.ALL\_TASKS\_AP.append(self)

else:

Task.ALL\_TASKS.append(self)

def suspend(self):

if self.state != TASK\_SUSPENDED:

self.last\_work\_time = self.env.time

self.state = TASK\_SUSPENDED

log('tsk', "[%d] task #%d suspended" % (self.env.time, self.id))

def resume(self):

if self.state != TASK\_PROGRESS:

self.waiting\_time += self.env.time - self.last\_work\_time

self.state = TASK\_PROGRESS

log('tsk', "[%d] task #%d resumed" % (self.env.time, self.id))

def finished(self):

self.state = TASK\_FINISHED

self.finish\_time = self.env.time

self.turnaround = self.finish\_time - self.creation\_time

log('tsk', "[%d] task #%d finished. turnaround time: %d" %

(self.env.time, self.id, self.turnaround))

def kill(self):

self.state = TASK\_KILLED

log('tsk', "[%d] task #%d killed" % (self.env.time, self.id))

def task\_generator(env, task\_f, task\_d, ap):

time = task\_f() # once per period

prev\_time = env.time

diff = env.time + time

while True:

diff = env.time - prev\_time

while diff >= time:

diff -= time

task = Task(env, task\_d(), ap)

yield task

prev\_time = env.time

time = task\_f()

yield None

def poisson(l):

return -(1/float(l)) \* math.log(random.random())

def def\_task\_generator(env):

task\_f = lambda: poisson(env.LAMBDA\_DEF) # random.random()\*3\*1000+2000 # 2000..5000

task\_d = lambda: random.random()\*3\*1000+1000 # 1000..4000

return task\_generator(env, task\_f, task\_d, False)

def ap\_task\_generator(env):

task\_f = lambda: poisson(env.LAMBDA\_AP) # random.random()\*1.5\*1000+4000 # 4000..5500

task\_d = lambda: random.random()\*1\*1000+500 # 500..1500

return task\_generator(env, task\_f, task\_d, True)

class Queue(list):

pass

class BaseProcessor(object):

def \_\_init\_\_(self, environment):

self.new\_tasks = []

self.tid\_count = 1

self.environment = environment

def add\_task(self, task):

task.id = self.tid\_count

self.tid\_count += 1

task.worktime = 0

self.new\_tasks.append(task)

def tick(self):

pass

TIME\_QUANT = 600

THRESHOLDS = [3, 6, 20]

QUEUE\_NUM = 3

class Processor(BaseProcessor):

def \_\_init\_\_(self, env, quant=TIME\_QUANT, q\_num=QUEUE\_NUM, thresholds=THRESHOLDS):

super(Processor, self).\_\_init\_\_(env)

self.qs = [Queue() for q in range(q\_num)]

self.q\_ap = []

self.current = None

self.thresholds = thresholds

self.quant = quant

self.prev\_time = env.time

def get\_new(self):

self.current = None

if len(self.q\_ap):

self.current = self.q\_ap[0]

self.current.queue = -1

self.current.resume()

self.q\_ap.remove(self.current)

return

for q\_id, queue in enumerate(self.qs):

if len(queue) > 0:

self.current = queue[0]

self.current.queue = q\_id

self.current.resume()

self.qs[self.current.queue].remove(self.current)

break

def tick(self):

for task in self.new\_tasks:

task.loops = 0

task.queue = 0

if task.ap:

self.q\_ap.append(task)

else:

self.qs[0].append(task)

self.new\_tasks = []

elapsed = self.environment.time - self.prev\_time

# log('cpu', "elapsed %d" % elapsed)

self.prev\_time = self.environment.time

if not self.current:

self.get\_new()

if self.current:

self.current.session = 0

if self.current:

# elapsed = min(self.quant, self.current.duration -

# self.current.worktime)

self.current.worktime += elapsed

self.current.session += elapsed

# log('cpu', "[%d] task #%d worktime %d AFTER" %

# (env.time, self.current.id, self.current.worktime))

if self.current.worktime >= self.current.duration or \

self.current.session >= self.quant or \

(len(self.q\_ap) and not self.current.ap):

self.current.session = 0

self.current.suspend()

self.current.loops += 1

if self.current.loops >= self.thresholds[self.current.queue] and \

not self.current.ap:

self.current.loops = 0

self.current.queue += 1

if self.current.queue >= len(self.qs):

self.current.kill()

else:

log('cpu', "task #%d moved to queue %d" %

(self.current.id, self.current.queue))

if self.current.worktime < self.current.duration and \

self.current.state != TASK\_KILLED:

if self.current.queue == -1:

self.q\_ap.append(self.current)

else:

self.qs[self.current.queue].append(self.current)

else:

self.current.finished()

self.get\_new()

if self.current:

self.current.session = 0

# return elapsed # 0.6t period

def is\_empty(self):

return all(len(q) == 0 for q in self.qs) and not self.current

class Environment(object):

def \_\_init\_\_(self):

self.time = 0

self.time = 0

self.tasks = def\_task\_generator(self)

self.ap\_tasks = ap\_task\_generator(self)

self.processor = Processor(self)

self.producing = True

self.utilization\_time = 0

self.QUANT = 10

self.LAMBDA\_DEF = 0.0003

self.LAMBDA\_AP = 0.00005

def is\_empty(self):

return self.processor.is\_empty()

def tick(self):

if self.producing:

task = next(self.tasks)

while task is not None:

self.processor.add\_task(task)

log('env', "[%d] added task #%d" % (self.time, task.id))

task = next(self.tasks)

task = next(self.ap\_tasks)

while task is not None:

self.processor.add\_task(task)

log('env', "[%d] added task #%d" % (self.time, task.id))

task = next(self.tasks)

self.time += self.QUANT

if self.processor.current:

self.utilization\_time += self.QUANT

self.processor.tick()

# log('env', "[%d] ticked" % (env.time))

def test(l\_def, l\_ap, time):

env = Environment()

env.LAMBDA\_DEF = l\_def

env.LAMBDA\_AP = l\_ap

while env.time < time:

env.tick()

prod = env.time

env.producing = False

while not env.is\_empty():

env.tick()

print("SIMULATION FINISHED")

results = {}

results['overtime'] = env.time - prod

print("%d overtime" % results['overtime'])

results['utilization'] = (env.utilization\_time / env.time)\*100

results['tasks processed'] = env.processor.tid\_count

results['AP tasks'] = len(Task.ALL\_TASKS\_AP)

results['tasks/s'] = (env.processor.tid\_count / (env.time / 1000))

print("%4.2f%% utilization" % results['utilization'])

print("%d tasks processed" % results['tasks processed'])

print("%d AP tasks" % results['AP tasks'])

print("%4.4f tasks/s" % results['tasks/s'])

mean\_turnaround\_def = sum(t.turnaround for t in Task.ALL\_TASKS) / len(Task.ALL\_TASKS)

results['mean turnaround def'] = mean\_turnaround\_def

mean\_turnaround\_ap = sum(t.turnaround for t in Task.ALL\_TASKS\_AP) / len(Task.ALL\_TASKS\_AP)

results['mean turnaround ap'] = mean\_turnaround\_ap

print("%4.4f mean turnaround" % (mean\_turnaround\_def))

print("%4.4f mean turnaround AP" % (mean\_turnaround\_ap))

waiting\_time = sum(t.waiting\_time for t in Task.ALL\_TASKS) / len(Task.ALL\_TASKS)

results['waiting time'] = waiting\_time

print("%4.4f waiting time" % waiting\_time)

waiting\_time\_AP = sum(t.waiting\_time for t in Task.ALL\_TASKS\_AP) / len(Task.ALL\_TASKS\_AP)

results['waiting time AP'] = waiting\_time\_AP

print("%4.4f waiting time AP" % waiting\_time\_AP)

sorted\_tasks = sorted(Task.ALL\_TASKS, key=lambda x: x.duration)

waiting\_times = []

durations = []

last\_dur = sorted\_tasks[0].duration

vals = []

waiting\_times\_qs = [[] for x in range(QUEUE\_NUM)]

durations\_qs = [[] for x in range(QUEUE\_NUM)]

start\_q\_id = 0

for i, dur in enumerate([t.duration for t in sorted\_tasks]):

if last\_dur != dur:

durations.append(last\_dur)

waiting\_times.append(sum(vals)/len(vals))

while last\_dur > TIME\_QUANT\*(sum(THRESHOLDS[q\_id] for q\_id in range(0, start\_q\_id+1))):

start\_q\_id += 1

waiting\_times\_qs[start\_q\_id].append(sum(vals)/len(vals))

durations\_qs[start\_q\_id].append(last\_dur)

vals = []

vals.append(sorted\_tasks[i].waiting\_time)

last\_dur = dur

else:

durations.append(last\_dur)

waiting\_times.append(sum(vals)/len(vals))

results['waiting times'] = waiting\_times # [t.waiting\_time for t in sorted\_tasks]

results['durations'] = durations # [t.duration for t in sorted\_tasks]

results['waiting times qs'] = waiting\_times\_qs

results['durations qs'] = durations\_qs

return results

def make\_plot(x, y, title\_t, xlabel\_t=u'intensity (lambda)', ylabel\_t=u'time, usec', dot='b.:', clear=True):

from pylab import plot, axis, xlabel, ylabel, grid, show, savefig, cla, rc

rc('font', \*\*{'family': 'serif'})

rc('text', usetex=True)

rc('text.latex', unicode=True)

rc('text.latex', unicode=True)

rc('text.latex', preamble='\\usepackage[utf8]{inputenc}')

rc('text.latex', preamble='\\usepackage[russian]{babel}')

plot(x, y, dot)

# axis([0, 0.01, 0, 10000])

ylabel(ylabel\_t)

xlabel(xlabel\_t)

grid()

savefig('%s.png' % title\_t.replace(" ", "\_"))

if clear:

cla()

if \_\_name\_\_ == '\_\_main\_\_':

time = 5000\*600

l\_def\_r = arange(0.00005, 0.0005, 0.000005)

print

results = []

for l\_def in l\_def\_r:

print ("Test: %f" % l\_def)

res = test(l\_def, 0.00005, time)

results.append(res)

make\_plot(l\_def\_r, [x['waiting time'] for x in results], 'waiting time')

make\_plot(l\_def\_r, [100-x['utilization'] for x in results], 'idle time')

make\_plot(l\_def\_r, [x['waiting time AP'] for x in results], 'waiting time AP')

res = test(0.0002, 0.00005, time)

make\_plot(res['durations'], res['waiting times'], 'waiting by duration',

'duration, usec', 'waiting time in queue, usec', dot='b.')

make\_plot(res['durations qs'][0], res['waiting times qs'][0], 'waiting by duration with queues',

'duration, usec', 'waiting time in queue, usec', dot='b.', clear=False)

make\_plot(res['durations qs'][1], res['waiting times qs'][1], 'waiting by duration with queues',

'duration, usec', 'waiting time in queue, usec', dot='r.', clear=False)

make\_plot(res['durations qs'][2], res['waiting times qs'][2], 'waiting by duration with queues',

'duration, usec', 'waiting time in queue, usec', dot='g.', clear=True)

make\_plot([x['waiting time'] for x in results],

[x['tasks processed'] for x in results], 'tasks count by waiting time',

'waiting time in queue, usec', ' handled tasks ', dot='b.:')