Name: Szymon Frąszczak	Index: 131010	Task: 2	Method: M2 Event Scheduling

Description of task:

The process scheduling (also called as CPU scheduler) is the activity of the process manager that handles the removal of the running process from the CPU and the selection of another process on the basis of a particular strategy. A scheduling allows one process to use the CPU while another is waiting for I/O, thereby making the system more efficient, fast and fair. In a multitasking computer system, processes may occupy a variety of states (Figure 1). When a new process is created it is automatically admitted the ready state, waiting for the execution on a CPU. Processes that are ready for the CPU are kept in a ready queue. A process moves into the running state when it is chosen for execution. The process's instructions are executed by one of the CPUs of the system. A process transitions to a waiting state when a call to an I/O device occurs. The processes which are blocked due to unavailability of an I/O device are kept in a device queue. When a required I/O device becomes idle one of the processes from its device queue is selected and assigned to it. After completion of I/O, a process switches from the waiting state to the ready state and is moved to the ready queue. A process may be terminated only from the running state after completing its execution. Terminated processes are removed from the OS.

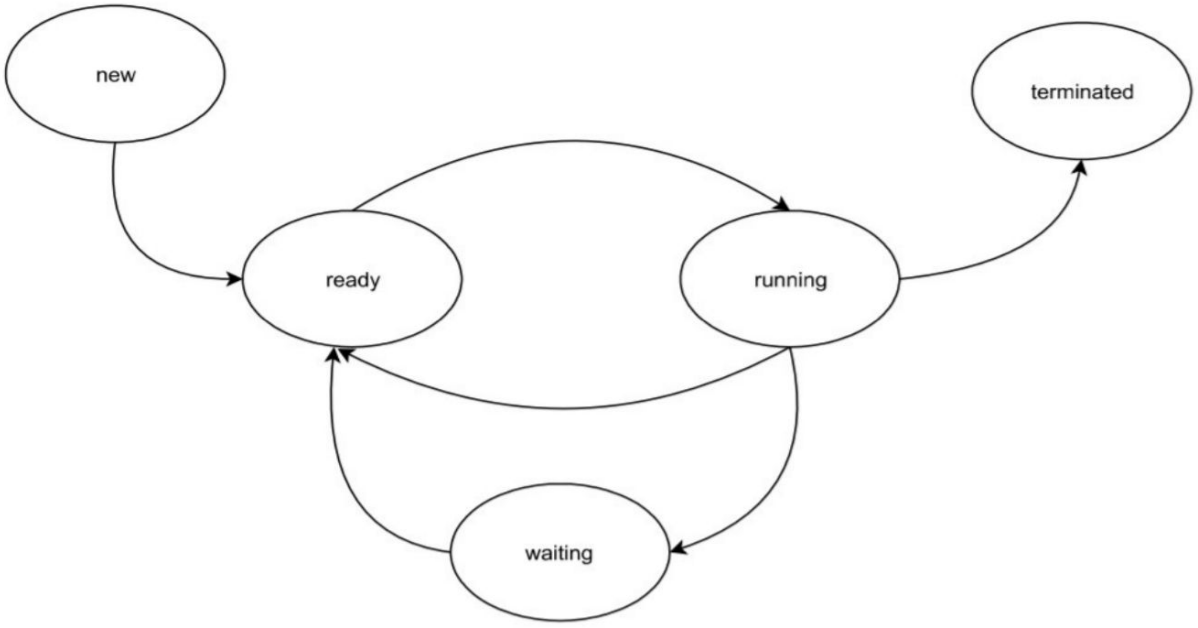


Figure 1. Process state diagram

Develop a C++ project that simulates the process scheduling described above, in accordance with the following parameters:

- Process Generation Time (PGT) [ms] – time before generation of a new processes (random variable with exponential distribution and intensity L) (round to natural number)

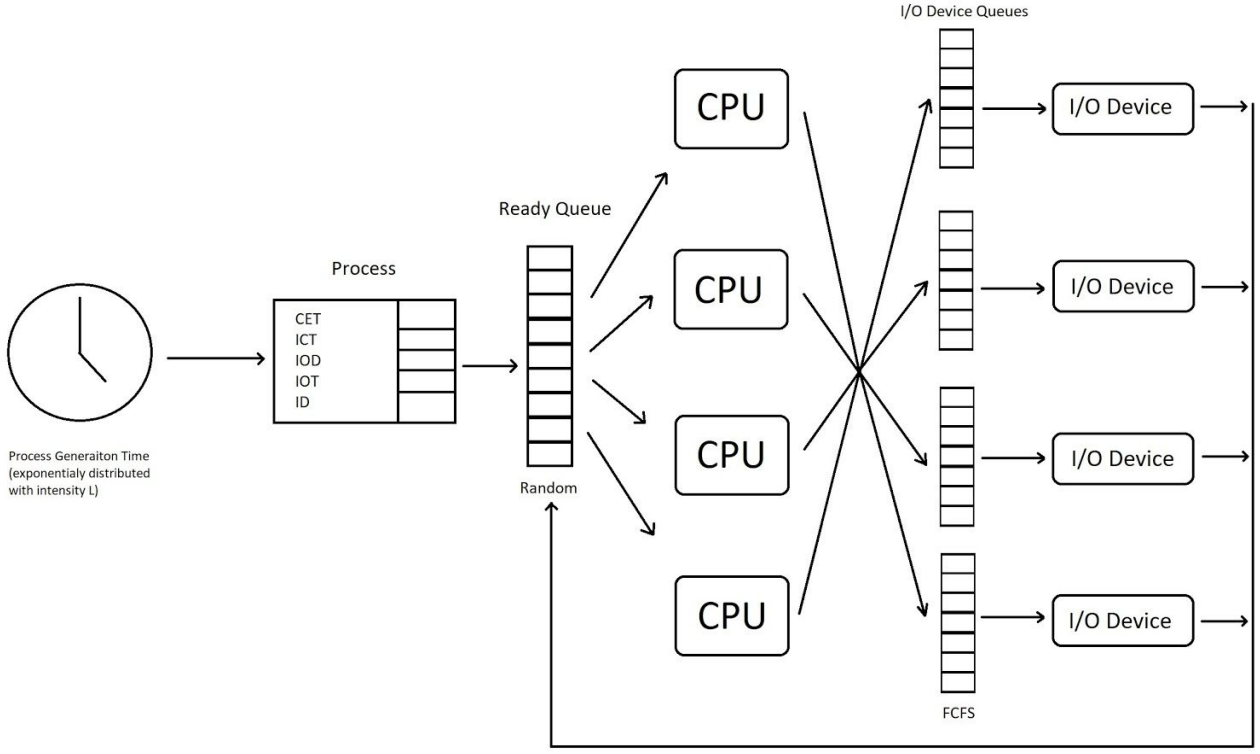
- CPU Execution Time (CET) [ms] – process execution time in CPU. Random variable with uniform distribution between <1, 50> [ms] (natural number)
- I/O Call Time (IOT) [ms] – time between getting an access to the CPU and an I/O call. Random variable with uniform distribution between <0, CET-1> [ms] (natural number). In case of 0, there is no I/O call.
- I/O Device (IOD) – indicates which I/O device is requested by the running process. Random variable with uniform distribution between <0, N_{IO}-1>, where N_{IO} is the number of I/O devices in the OS.
- I/O Time (IOT) [ms] – I/O occupation time. Random variable with uniform distribution between <1, 10> [ms] (natural number).

Determine the value of the parameter L that ensures the average waiting time in the ready queue not higher than 50 ms. Then, run at least ten simulations and determine:

- CPU utilization (for every CPU) [%]
- Throughput – number of processes completed (terminated) per unit time
- Turnaround time – time required for a particular process to complete, from its generation until termination [ms]

Draw a figure of an average waiting time in the ready queue in the function of parameter L.

Model scheme:



Description of objects and their attributes:

Object	Class name	Description	Attributes
CPU Scheduler	CpuScheduler	Class used as a container of all other objects	- kNumberOfCpus - number of CPU's of type const int - kNumberOfDevices - number of I/O Devices of type const int - cpu_list - CPU's vector of type vector<Cpu*> - io_device_list - I/O Devices vector of type vector<IoDevice*> - ready_queue - process Ready Queue of type ReadyQueue - device_queue_list - Device Queue list of type vector<DeviceQueue*> - UniGenCET, UniGenIOT, UniGenICT, UniGenIOD - uniform generators of type UnifromGenerator for generating random values of CET, IOT, ICT and IOD - UniGenPGT - uniform generator of type UniformGenerator to generate seeds to exponential generator ExpGen - ExpGen - exponential generator of type ExpGenerator to generate values of PGT
Process	Process	Class that represents a single process	- id_s - unique process identifier of type static int - cet - CPU Execution Time of type int - ict - IO Call Time of type int - iod - IO Device requested number of type int - iot - IO Occupation Time of type int - sw - start waiting time in Ready Queue of type int - ew - end waiting time in Ready Queue of type int - tw - total waiting time in Ready Queue of type int - se - start of process's existence of type int - ee - end of process's existence of type int

Ready Queue	ReadyQueue	Queue to CPU's	- process_ready_queue - queue of type vector<Process *> with random priority - UniGenRQ - uniform generator of type UniformGenerator to randomise taking processes from queue - empty - flag of type bool to check if queue is empty
Device Queue	DeviceQueue	Class that represents single Queue to I/O Device	- device_queue - process queue of type queue<Process*> with FCFS priority - empty - flag of type bool to check if queue is empty
CPU	Cpu	Class that represents single CPU	- process_served_cpu_ptr - pointer of type Process to process that is currently being served by CPU - end_service - variable of type int that holds end time of service for process currently served in CPU - busy - flag of type bool to check if CPU is busy
I/O Device	IoDevice	Class that represents single I/O Device	- process_served_io_ptr - pointer of type Process to process that is currently being served by IO Device - end_service - variable of type int that hold end time of service for process currently served in IO Device - busy - flag of type bool to check if IO Device is busy

List of events:

Time Events	Algorithm
New Process	1.Generate new process and time to generating new(another) process. 2. Check if any CPU is free: 2.1 If CPU is free assign process to CPU's process pointer. 2.2 Set CPU busy. 2.3 Set CPU's end of service time. 2.4 Create End of work in CPU Event. 3. If all CPUs are busy push the process to Ready Queue. 4. Create New Process event with generated PGT.
End of work in CPU	1.Get process from CPU. 2. Set CPU not busy. 3. If ICT equals 0 terminate process. 4. If ICT greater than 0 try to move process to proper IO Device: 4.1 If free enter process to IO Device and assign pointer. 4.2 Set IO Device busy. 4.3 Set IO Device end of service time. 4.4 Create End of work in IO Device Event. 5. If IO Device is busy push process to it's queue. 6. Check if there are any processes on Ready Queue: 6.1 If Ready Queue is not empty take process and enter it to recently released CPU by assigning it's pointer. 6.2 Set CPU busy. 6.3 Set CPU's end of service time. 6.4 Create End of work in CPU Event.
End of work in IO Device	1. Get process from IO Device. 2. Set IO Device not busy. 3. Generate randomly new ICT and IOT for process. 4. Check if any CPU is free: 4.1 If CPU is free assign process to CPU's process pointer. 4.2 Set CPU busy. 4.3 Set CPU's end of service time. 4.4 Create End of work in CPU Event. 5. If all CPUs are busy push the process to Ready Queue. 6. Check if there are any processes on Device Queue to recently released IO Device: 6.1 If Device Queue is not empty take process and enter it to device by assigning it's pointer. 6.2 Set IO Device busy. 6.3 Set IO Device end of service time. 6.4 Create End of work in IO Device Event.

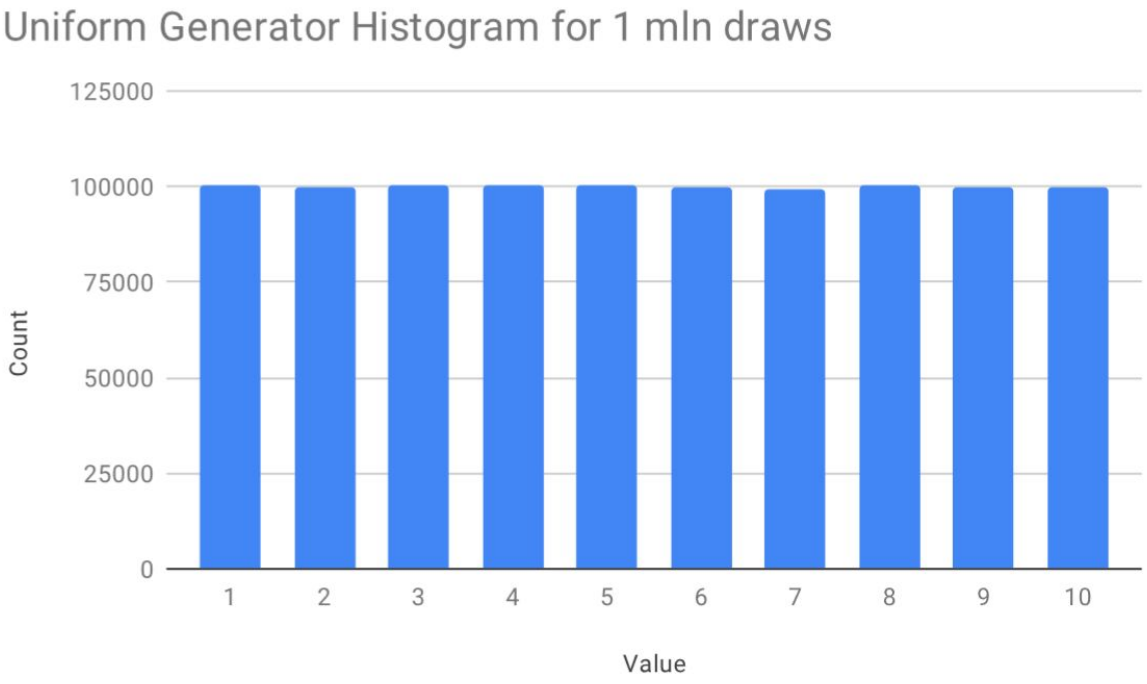
Generators:

In my simulation I am using Uniform and Exponential Generators to draw pseudo random numbers for purposes of generating particular times. To use as independent draws as possible I use many generators with prepared seeds.

Seeds

Seeds are generated by implemented generators. Using number 10 as initial seed I've saved ten seeds for each of seven generators, with span of 100 000 numbers.

1. **Uniform Distribution Generator**



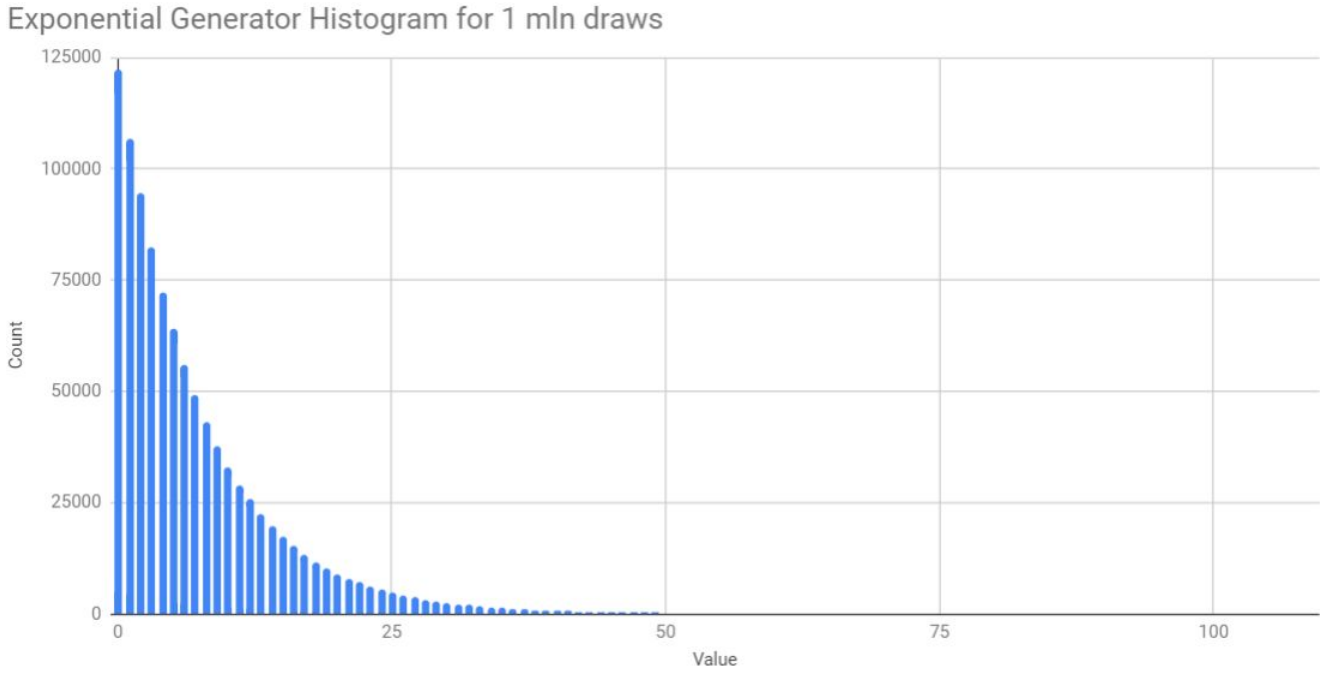
Uniform Random Generator

```
M = 2147483647.0;
A = 16807;
Q = 127773;
R = 2836;

int h = kernel_/Q;
kernel_ = A*(kernel_-Q*h)-R*h;
if (kernel_ < 0)
        kernel_ = kernel_ +static_cast(M);
```

- Variable „kernel_" is the current seed. Program calculated new value for kernel and this value will be returned as the new random number and also will be safed as the new seed.

2. Exponential Distribution Generator



Exponential Random Generator

```
double k = uniform_->Rand01();
return -(1.0/lambda_)*log(k);
```

uniform_ – uniform generator
Rand01 – function returns random number in the range 0-1

Program input parameters:

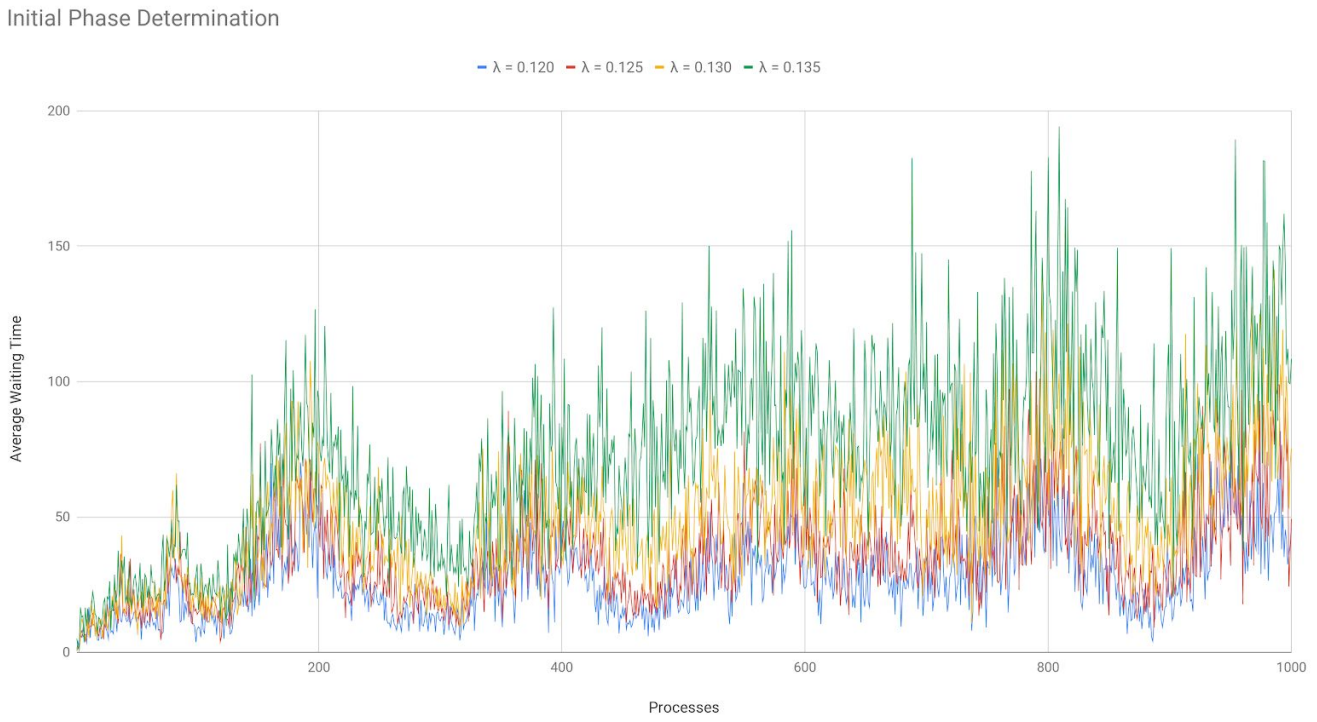
simulation time - length of simulation time
lambda - intensity of generating processes
simulation mode - ‘0’ for continuous or ‘1’ for step by step
initial phase mode - ‘0’ to not skip initial phase or ‘1’ to skip initial phase

Code testing methods

While writing a program I’ve using Compiler and Debugger to check if my new code works as it should. Some errors caused very hidden problem such that I had to use the break points and verify code line by line observing variables, pointers and objects. Finding mistakes and debugging was very hard when code was modified and inserted some new, in few classes. After some time I’ve learned that better check the code twice time as needed, than omeet it and verify not so often. When whole program was finished to verify if program works correctly I’ve been analyzing logs in step by step mode observing processes entering the CPUs, IO Devices, and respectively their Queues if they were busy. To be sure I checked in debugging mode how Future Event List Calendar behaves and that I’ve no trash pointers in used objects.

Simulation results

- Initial phase determination



To get results for each lambda I ran 10 simulations on different seeds of generators. To determine initial phase I've observed average waiting time of process in the ready queue. After generating chart from gathered data I've determined end of initial phase at 200'th process. Simulation time for this process was varying between 1500 ms and 1750 ms depending on simulation. To be sure that initial phase is finished I assumed time 1800 ms as initial phase.

- Determination of exponential generator intensity λ
    - Average value μ
        - N - number of observations in one simulation
        - M - number of simulations

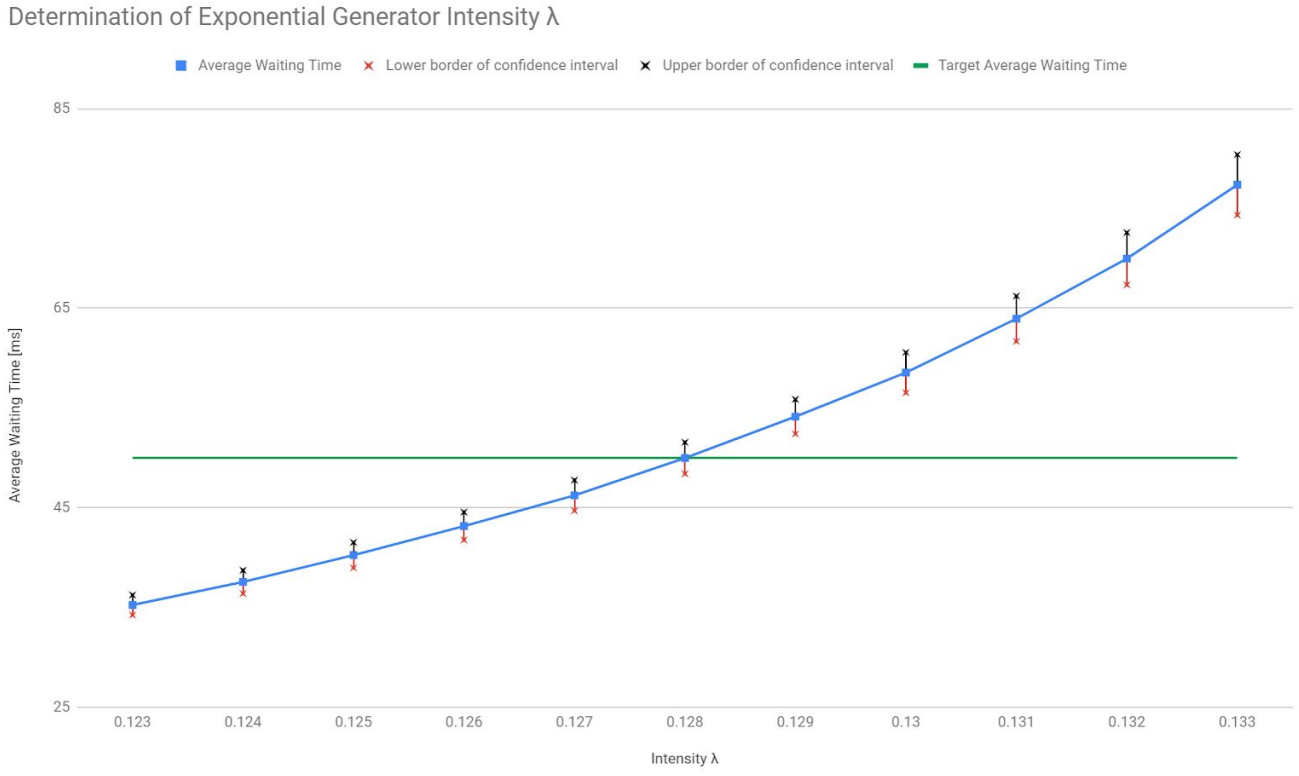
$$\mu_m = \frac{1}{N}\sum_{n=1}^{N} X_{mn} \qquad \mu = \frac{1}{M}\sum_{m=1}^{M} \mu_m$$

- Standard deviation

$$s(\mu_m) = \sqrt{\frac{1}{M-1}\sum_{m=1}^{M}(\mu_m - \mu)^2}$$

- Confidence interval

$$u \in \left[\mu - t_{M-1,\alpha}\frac{s}{\sqrt{M-1}}, \mu + t_{M-1,\alpha}\frac{s}{\sqrt{M-1}}\right]$$

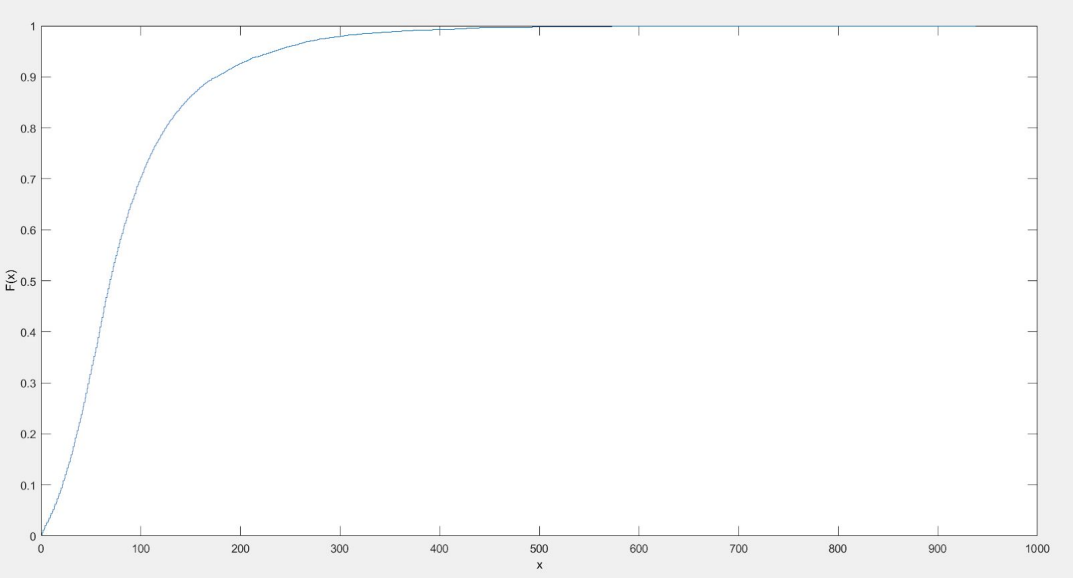


Above chart is presenting average waiting time depending on intensity of generation of new processes with confidence intervals for each lambda. To gather data needed to determine intensity of exponential generator I've observed average waiting time of process in the ready queue. Results are taken from 10 simulations with simulation length of 1 000 000 ms for each lambda intensity. Closest value of parameter to meet requirements of average waiting time lower than 50 ms is λ = 0.128, with average from 10 simulations equal to 49.987865.

- Simulation results table

Simulation number	Average waiting time in the ready queue [ms]	CPU Utilization				Throughput [prcs/ms]	Average turnaround time [ms]
		CPU #1 [%]	CPU #2 [%]	CPU #3 [%]	CPU #4 [%]		
1	48.769126	92.075135	89.188239	85.297035	80.625025	0.136354	94.502142
2	48.001926	92.113905	89.138149	85.275696	80.561711	0.136266	93.750965
3	49.213984	92.335103	89.400120	85.656983	81.246043	0.137033	95.058632
4	47.398670	92.151072	89.116610	85.272891	80.634843	0.136469	93.156236
5	52.070234	92.375075	89.674714	85.899920	81.403326	0.137190	98.055395
6	48.316827	92.269585	89.402725	85.754658	81.180525	0.136838	94.210693
7	53.274140	92.335404	89.458726	85.608495	81.016530	0.136669	99.300836
8	50.036168	92.096173	89.130535	85.086656	80.586957	0.136332	95.719922
9	49.861531	92.348327	89.553496	85.724204	81.163695	0.136594	95.706875
10	52.936041	92.471348	89.677019	86.012422	81.567321	0.136645	98.945571
Average	49.987865	92.257113	89.374033	85.558896	80.998598	0.136639	95.840727
Confidence interval +/-	3.298653 592	0.014349 8916	0.036765 92489	0.0714936 354	0.104006 63	0.000000070 79027053	3.618159 789

- Empirical cumulative distribution



X axis is turnaround times of processes
Y axis is probability of given turnaround time
Results taken from 10 simulations. Value of ECDF at any specified value of the measured variable is the fraction of observations that are less than or equal to the specified value.

- Conclusions

Performing simulation I've obtained interesting results from which I can see some clues to improve system. First processor is always busiest one, and with descending order we can observe decreasing utilization of CPUs. It has no influence on turnaround times of processes and effectiveness of system, but I can imagine that in real life model it should be balanced that processors will be equally worn of. Another aspect of CPUs is that they are not fully used due to processes which are waiting to be serviced by IO Device and cannot go further. Adding another IO Device would allow to bigger utilization of CPUs
I think that system would have bigger throughput if I would use FCFS or SJF algorithm for Ready Queue instead of random priority. That's because random algorithm in extreme circumstances can lead to situation in which on process have been waiting very long whilst another is pushed and popped almost in the same instance of time.