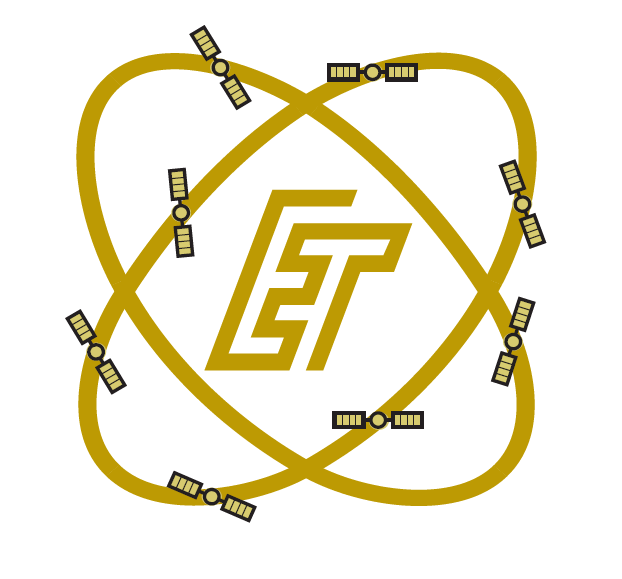
|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Method:  Activity Scanning | Date  24.01.19 | Jędrzej Kieruj | Electronics & Telecommunication | Semester  V | Gr. E1 |
| Mgr inż. Karolina Lenarska | | | Grade: | | |

Simulation Techniques  
Project Report

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**





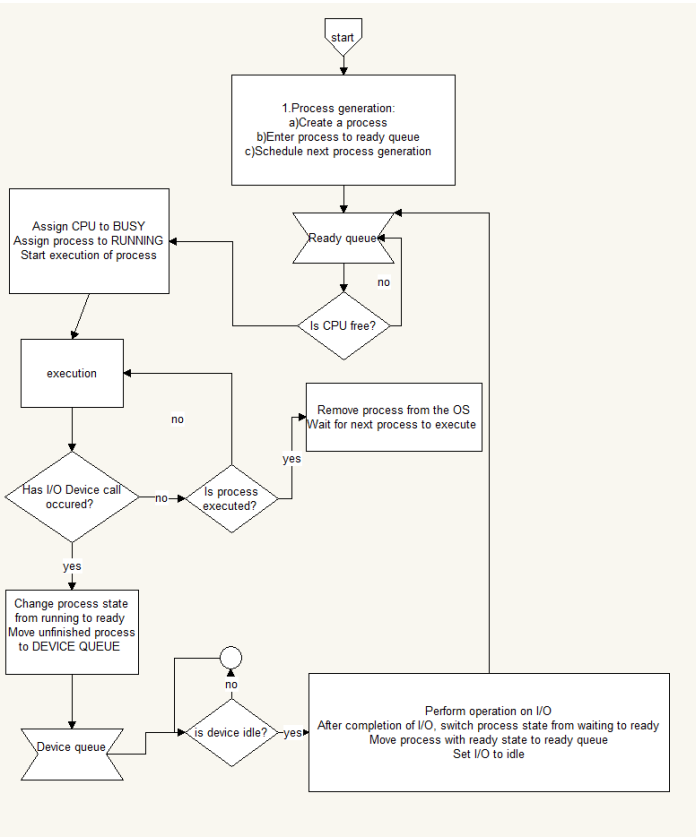
1. Project task:

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.

Simulation method – Activity Scanning

|  |  |  |
| --- | --- | --- |
| Parameter | Description | Value [**ms**] (round to natural number) |
| PGT | Process Generation Time – time before generation of a new processes | random variable with exponential distribution and intensity L |
| CET | CPU Execution Time – process execution time in CPU. Random variable with uniform distribution between | Random variable with uniform distribution between <1,50> |
| IOT | I/O Call Time – time between getting an access to the CPU and an I/O call. In case of 0, there is no I/O call. | Random variable with uniform distribution between <0, CET-1> |
| IOD | I/O Device (IOD) – indicates which I/O device is requested by the running process (NIO – number of IO). | Random variable with uniform distribution between <0, NIO-1>, |
| IOT\_occupation | • I/O Time (IOT) – I/O occupation time. Random variable with uniform distribution between | Random variable with uniform distribution between <1, 10> |

1. Simulation model scheme:



|  |  |  |
| --- | --- | --- |
| Object/class | Description | Attributes |
| CPUscheduler / ActivityScanning | Class containing all other objects implementing every attribute of simulation. Activity Scanning is main launcher of simulation. | RNG \* rng;  long sim\_time;  long total\_time;  int programModeNumb;  vector<QueueIO\*> devices\_queues; ReadyQueue CPUqueue; vector<CPU\*> cpus; vector<IOdevice\*> devices;  const int cpuNumb = 3;  const int ioNumb = 4;  long next\_process\_time;  void UpdateSimTime(); |
| Generator / kernels / rng | Description attached later. | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ |
| Process | Class generating processes with arbitrary PGT, CET, IOT, IOD for certain range. | int turnaroundTime;  int PGT;  int CET;  int IOT\_occupation; int IOT;  int IOD; int total\_ioExecTime; int ioExecTime; int processID;  long ReadyQTime;  long totalQTime;  static int ID;  int numbOfExecutedProcesses;  long bornTime; |
| ReadyQueue/QueueIO | Queue of all processes waiting for CPU/ each IO. | ReadyQueue();  Process\* getNextProcess();  void pushProcess(Process\* p);  int getLength() { return ReadyQ.size(); };  void clear() { ReadyQ.clear(); };  ~ReadyQueue();  bool isEmpty();  std::vector<Process\*> ReadyQ; |
| CPU | Vector of containers for processes. | bool cpuStatus() {  return process\_cpu != nullptr;  };  Process\* get\_process\_cpu();  void assign\_process\_to\_cpu(Process\*, long time);  void take\_out\_process\_from\_CPU();  long getServiceEnd();  long getServiceStart();  long getWorkingTime();  CPU();  ~CPU();  Process\* process\_cpu; long serviceEnd, serviceStart; long workingTime; |
| IO | Vector of vectors containing queues of each I/O Device. | void take\_out\_process\_from\_IO();  void assign\_process\_to\_IO(Process\*, long time);  long getServiceEnd();  IOdevice();  ~IOdevice();  Process\* get\_process\_io();  bool IO\_status(){return process\_io != nullptr;}  Process\* process\_io;  long serviceEnd;  long serviceStart;  long getProcessingTime(); |
| Statistics\_holder | Class to hold all statistics collected during simulation, later saved to files. | long completedProcesses;  long waitingTimeInReadyQueue;  long sum\_Turnaroundtime;  std::vector<long> cpuUtilization;  std::vector<int> turnaroundHistogram;  std::vector<std::pair<long, int>> ReadyQSize;  std::vector<std::pair<long, double>> ReadyQTime;  long totalReadyQTime; |

1. Description of time and conditional events:

Time events:

- generation of process   
- termination of process on CPU   
- termination of process on I/O device   
Time events in proper order for correctly working simulation. Each based on simulation time, or time generated to break/suspend or end of testing/suspension.

Conditional events:

- starting work on processor   
- starting work on I/O Device  
Event not based on time, performer after checking all time event which may influance performing those actions. All events are implemented in simulation.

1. Generators   
   a) Uniform distribution generator  
   With uniform distribution generator we should be able to obtain random numbers with arbitrary distribution functions is a variate uniformly distributed over the interval . The most common way to achieve that is is Lehmer’s linear congruential generator:   
   𝑋𝑛+1 = 𝑎𝑋𝑛 + 𝑐 𝑚𝑜𝑑 𝑚   
   After simplification and dealing with problem that we need to use intermediates larger than 32 bits, we can use method:   
   ℎ = 𝑋 𝑞 𝑋 = 𝑎 (𝑋 − 𝑞ℎ) − 𝑟ℎ 𝑖𝑓 𝑋 < 0,𝑡ℎ𝑎𝑛 𝑋 = 𝑋 + 2147483647 2147483647 𝑖𝑠 𝑚𝑎𝑥𝑖𝑚𝑢𝑚 𝑛𝑢𝑚𝑏𝑒𝑟 𝑟𝑒𝑝𝑟𝑒𝑠𝑒𝑛𝑡𝑒𝑑 𝑏𝑦 32 𝑏𝑖𝑡 𝑣𝑎𝑟𝑖𝑎𝑏𝑙𝑒 (2 32 − 1)  
     
     
    In my program I used that method, with set of parameters proposed in literature (J. Tyszer „Object-oriented computer simulation of discrete-event systems”) and lectures.  
    𝑎 = 16807   
   𝑞 = 127773   
   𝑟 = 2836

C++ implementation of uniform distribution generator: In generator.h file:

|  |
| --- |
| class UniformGenerator  {  public:  UniformGenerator(int kernel);  virtual ~UniformGenerator();  // Draws number between <0,1>  double Rand();  // Draws number between <start, end>  double Rand(int start, int end);  int get\_kernel() {return kernel\_;};  private:  int kernel\_;  double M;  int A;  int Q;  int R;  }; |

In generator.cpp file:

|  |
| --- |
| UniformGenerator::UniformGenerator(int kernel): kernel\_(kernel)  {  M = 2147483647.0;  A = 16807;  Q = 127773;  R = 2836;  }  UniformGenerator::~UniformGenerator()  {  }  double UniformGenerator::Rand()  {  int h = kernel\_/Q;  kernel\_ = A\*(kernel\_-Q\*h)-R\*h;  if (kernel\_ < 0)  kernel\_ = kernel\_ + static\_cast<int>(M);  return kernel\_/M;  }  double UniformGenerator::Rand(int start, int end)  {  return Rand()\*(end-start) + start; |

b) Exponential distribution generator

After making uniform distribution generator, exponential one was ease to achieve. In exponential generator we introduce variable U, which is uniform distributed.

𝐹 −1 (𝑈) = −𝜆 −1 ln(1 − 𝑈)

|  |
| --- |
| class ExpGenerator  {  public:  ExpGenerator(double lambda, UniformGenerator \*ug);  virtual ~ExpGenerator();  double Rand();  private:  double lambda\_;  UniformGenerator \*uniform\_;  }; |

C++ implementation   
In generator.h file:

In generator.cpp file:

|  |
| --- |
| ExpGenerator::ExpGenerator(double lambda, UniformGenerator \*ug): lambda\_(lambda), uniform\_(ug)  {  }  ExpGenerator::~ExpGenerator()  {  if (uniform\_ != nullptr)  delete uniform\_;  }  double ExpGenerator::Rand()  {  double k = uniform\_->Rand();  return -(1.0/lambda\_)\*log(k);  } |

1. Program input parameters:

In my simulation program, I used 4 initial variables. The first one is parameter responsible for number of how many simulations are needed, second is number of processes that simulation has to generate, the third is parameter concerning simulation mode. There are 2 simulation modes, one allows to simulate step by step and displaying particular event in form of text, another finishes simulation uninterruptedly without notifications. Last parameter is number of processes after which simulation will start gathering the results.

1. Code testing methods:

During making the program, I faced many problems. In the beginning, I had to ensure that particular time (IOT, CET etc.) in each moment of simulation is correct. Some of this values are used in further generations and in equations. I had problem that my values sometimes after equations resulted in “0” that was then used to divide by other variables. I had to fix it. The next troubles came when generators were used. I had to ensure that everything works correctly. In final stage I outstood problem with wrong CPU utilization results. I started gathering information in wrong way, resulting CPU utilization equal to 1-2%.

1. Initial phase determination:

Due to fact that there is very noticeable rise after 250 processes, I decided to analyze longer period of simulation. I carried out simulation with 2000 processes.

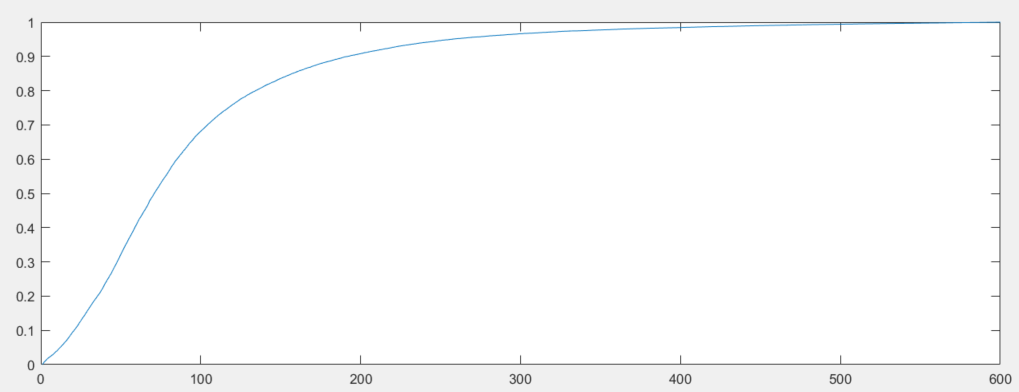
We can see that behavior of chart is relatively suitable. After 1700 processes performed, we clearly see that we have another significant rise. This leads us to conclusion, that noticeable peak is periodic, but general trend is that processes behave naturally. By this fact, I assumed that after **40 processes** system is stable. Until this time, we stay in **initial phase** and therefore before reaching stability we should omit results.

1. Determination of exponential generator intensity L:

General trend in choosing lambda is kept. The higher lambda, the bigger value of average waiting time in Ready Queue. There are noticeable wide ranges of intervals. This is caused by relatively big differences in results of each simulation. I chosen lambda equal to 0.092, which is highest parameter that gives average waiting time less than 50 ms.

FINAL RESULTS:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Simulation number | Average waiting time in Ready Queue | CPU #1 Utilization | CPU #2 Utilization | CPU #3 Utilization | Throughput | Average turnaround time |
| 1 | 49,6252 | 0,918148 | 0,859054 | 0,778189 | 0,0968108 | 96,7063 |
| 2 | 33,0954 | 0,886775 | 0,791671 | 0,699554 | 0,0930177 | 78,4365 |
| 3 | 68,9102 | 0,91714 | 0,844949 | 0,783428 | 0,0959769 | 115,209 |
| 4 | 57,2519 | 0,923858 | 0,869113 | 0,785471 | 0,0980402 | 104,756 |
| 5 | 39,1657 | 0,896774 | 0,820907 | 0,72417 | 0,0915542 | 85,0765 |
| 6 | 31,3289 | 0,911156 | 0,844925 | 0,771955 | 0,09578 | 76,7007 |
| 7 | 56,076 | 0,918061 | 0,847344 | 0,767099 | 0,0948534 | 104,127 |
| 8 | 39,0561 | 0,880174 | 0,798062 | 0,700339 | 0,093665 | 84,0178 |
| 9 | 73,0209 | 0,93986 | 0,891523 | 0,819835 | 0,100199 | 119,644 |
| 10 | 38,2769 | 0,91611 | 0,847076 | 0,76576 | 0,0971129 | 82,9179 |
| Average | 48,58072 | 0,9108056 | 0,8414624 | 0,75958 | 0,09570101 | 94,75917 |
| Confidence interval [+/-] | 11,281919 | 0,2402659 | 0,2317272 | 0,221051511 | 0,07778197 | 11,96404 |

1. Empirical cumulative distribution function:  
   I gathered turnaround times from 10 simulations and basing on them I created ECDF chart in Matlab. The plot is shown below.  
     
   
2. Conclusion:

General conclusion The project has significantly improved my programming skills, and allowed me to implement methods from the beginning to the end, which I have only read about before, but I have not used them as part of a larger program. The main problem encountered is the complexity of the entire program, and inaccurate knowledge of the topic that I had to learn step by step to complete a working program with simulation. Apart from the difficulties related to the subject of the simulation I got to know during lectures and literature, the main problems were related to the use of pointers and references to all objects / classes in the program. After completing the whole project, I came to the conclusion that it would be a great help to write a program in C # or Java, because they do not cause problems with pointers and references. Really helpful was methods introduced in C++11 standard of this language, such as:

• std::vector – simplifies operations on arrays of classes objects. It is possible to use just arrays, but it was easier to use build-in methods

• std::queue – huge simplification in this type of project, this container help to deal with FIFO queue used in this project.

Conclusion: topic of random numbers generations seems to me as the most interesting part of whole simulation. It is a topic being equally influential to simulation as main code of program. It aroused my curiosity more than other parts of this subject.