Goal of this chapter

- In this chapter, we will develop our first simulation program.
- It simulates a single queue, generates customers, and computes statistics.
- You should learn a few things about implementation concepts for discrete event simulation.
  - And maybe a few things about C++ 😊
- The entire source code is available on the course webpage (under Course Handouts).
  - Please do look at it!
- A first look at how to interpret results of simulation programs.
Overview

- Model and its state
- Implementing a queue
- Generating random numbers
- Program structure
- Interpreting results

Model

- Single server
- Single queue, first-in-first-out discipline
- Input parameters
  - Pattern of customer arrivals:
    - Time between customers is exponentially distributed
    - Mean $\lambda$ is a parameter for the program
  - Pattern of service times:
    - Time needed to serve a customer is exponentially distributed
    - Mean $\mu$ is a parameter for the program
- So-called M/M/1 system
- Metrics: Server utilization, average number in queue, average waiting time
- Stop simulation after 10000 customers have been processed
Necessary state

- Current simulation time
- Server state: idle or busy
- Time of next arrival
- Time of next departure
- A queue data structure to represent the queue
  - Queue is a first-in-first-out data structure – how to implement?
  - Queue entries contain required processing time and arrival time of the task (for statistics)

Variables to compute metrics

- Average waiting time
  - Every queue entry contains time of arrival
  - When task exits queue, use this value to update total waiting time, later divide that by number of customers
- Average queue length
  - Recall the representation as area under the curve of number of tasks currently in queue
  - Represent total area under this curve
- Average server utilization
  - Similar to average queue length, yet the curve only takes on values of 0 and 1, representing idle or busy
  - Represent total area under this curve as well
Open questions

- How to obtain randomness in a simulation program?
- How to implement a queue?
- How to organize the overall program structure?

Overview

- Model and its state
- **Implementing a queue**
- Generating random numbers
- Program structure
- Interpreting results
Implementing a queue

- A queue as abstract data type
  - Contains data of a certain type (at least, here)
  - Has a **push** operation that adds data to the end
  - Has a **pop** operation that removes one data element from the front and returns it (error if applied to empty queue)
  - Useful: **length** operation that returns number of elements
  - Number of elements should be unbounded

- Many simple implementations possible
  - Here: singly-linked list

- Make it reusable by making the data type a parameter of the class
  - Write a “template class”

Templates in C++

- Template: define a type that takes one (or more) types as parameter
  - E.g. define a “queue” type (a class) that contains the general logic of a queue, but is not specialized towards a certain type to be contained within the queue
  - When creating objects of such a general, parameterized class, the parameters have to be filled with concrete types, e.g., a queue of integers, or a queue of floats, or …

- Looks difficult, is difficult 😊
  - Well, not really – just a little bit awkward to write or read
  - For debugging use a recent compiler, e.g., clang!
Templates in C++

- Important point: the entire implementation of the class has to be included in the .hpp file (not just the declarations)
  - Reason: Templates in C++ are realized more or less via smart preprocessing
  - Compiler generates textual instances of code for every usage of the template class, these instances are then compiled
  - Complete source code necessary to do this

- Look at example in SIMQueue.hpp
  - SIMQueue class acts as a wrapper, containing pointers to the next elements (not needed outside)
  - queue is the actual implementation

Overview

- Model and its state
- Implementing a queue
  - Generating random numbers
    - Generators for random sequences
    - Seeds for generators
    - From random sequences to random distributions
    - How to use and reuse generators
- Program structure
- Interpreting results
Generating random numbers

- How to simulate the random process of arriving customers, needing random service times?
- Sources of true randomness exist, but are awkward to handle
  - E.g., number of particles radiating per second from radio-active material
  - Additional disadvantage: reproducing experiments would be difficult
- **Deterministically generate pseudo-random numbers** to represent service times, times between arrivals, etc.

---

Generating random numbers

- One possibility: Do it from scratch
- Many algorithms for random number generators (RNGs) exist
- Simple example: Linear congruential generators

\[ x_n = (ax_{n-1} + b) \mod c \]

where \( x_n \) is the current random number
- Example parameters: \( x_n = 5 \times x_{n-1} + 1 \mod 16 \)
- Initial value \( x_1 = 5 \)
- Resulting sequence of “random numbers”: 10, 3, 0, 1, 6, 15, 12, 13, 2, 11, 8, 9, 14, 7, 4, 5 (and here we are back again!), 10, 3, ...
- Division by 16 gives numbers between 0 and 1, to be interpreted as random numbers
Generating random numbers

- Typical properties of random number generators: a state and an initial state (seed)
  - In example: State is the current $x_i$, initial value is $x_0$
  - Modulus defines maximum number of different values, also defines required amount of space to hold state
    - A modulus of 16 needs 4 bits to represent state, but only 16 different numbers are possible (at best) -> not particularly random
- Larger state (often) gives “better” pseudo-random properties
- Designing good random number generators is difficult
  - Extensively treated in the literature, but not part of this course
  - Testing for randomness also difficult

Generator objects for random numbers

- Usually: use a library, e.g., UNIX C++ standard library (libc++)
  - Other programming environments usually provide similar mechanisms
  - Only for C++11!
- For historic projects: e.g. Communication Network Class Library CNCL
- Concept for random numbers in C++11 (and many others):
  - Separate generation of random number streams from distributions
Generator objects for random numbers

- Different classes that generate random streams, according to different algorithms
  - E.g. additive congruential generator:
    ```cpp
    std::linear_congruential_engine<
        uint32_t, LC_A, LC_B, LC_M> gen(seed);
    ```
  - gen is an object that provides a sequence of random numbers
  - Accessed via the operator() interface (operator overloading of “()”)

- All these classes return numbers that are uniformly distributed over the range of all possible numbers that fit into the given space
  - I.e., 0 .. 2^{space-1}

Seeds for generators

- How to choose seeds for generators?
- Using identical seeds will produce identical sequences, hence identical executions of the simulation, hence identical results
  - Pseudo-random generators are deterministic
- Use different seeds for different runs!
  - Choose seeds “randomly”, e.g., time of day the run is started? Usually not a good idea, because of repeatability, debugging
  - Pass seed(s) as explicit parameters to the program
Some pitfalls in seed selection

- Do not use 0
  - Some RNG algorithms are stuck at 0 (will only produce 0)
- Avoid even values as seed
  - Some generators (bad ones) will produce fewer different numbers with even seeds
  - NOTE: all RNGs can only produce a finite amount of different numbers, because they only return a finite number of bits. Good RNGs, however, will produce every possible number or at least many of them
- Avoid seed with a lot of ‘1’ or ‘0’ bits
  - May produce skewed results on some other RNGs

Distribution objects

- Obtain random numbers that follow a desired distribution
  - A number of different classes available that compute such random numbers, one class for one distribution
  - Initialize an object of such a distribution-specific class with a generator object and additional parameters (e.g., the mean of the distribution)
  - Example: exponential distribution
    ```cpp
    std::exponential_distribution<double> dist(mean);
    auto rnd = std::bind(dist, gen);
    ```
    will initialize an object `rnd`, using generator object `gen`
  - Calling `rnd()` will produce random numbers that are exponentially distributed with mean `mean`
  - Examples of available distribution: uniform, normal, lognormal
How to combine generators and distributions

- Suppose you need random numbers for different purposes, but they have the same distribution – use the same distribution object to generate them?
  - Obviously a bad idea:
    - What if the distribution (or just the parameters) do change for one of the different kinds of random processes
    - Bad self-documentation of the program
    - ...
  - Don’t do that! (Unless you know better)

One generator for different purposes?

- But suppose we do use different distribution objects – could we still use the same generator object?
  - Example: Generate stream \( \{x_1, x_2, x_3, \ldots\} \) and use even-indexed random numbers to generate the service time, odd-indexed random numbers to generate the interarrival time
    (This is called “subdividing a stream”)

- Reasonable proposal?
One generator for different purposes?

- No! – Two main disadvantages
  - Successive random numbers could be correlated –
    - Hence, the service and interarrival time would be correlated, too (large service time tends to occur together with large interarrival time).
    - This is not desirable – if correlation is the case, it needs to be modeled explicitly, not as a mistake in the implementation
    - Unsuspected correlation will in general lead to wrong conclusions
  - Impossible to change service times while keeping interarrival times the same
    - But might be desirable to study different system aspects

Multiple generators, one seed?

- Ok – so we use different random number streams for different purposes. Can we at least use the same seed for all of them?
  - (Choosing seeds is a hassle, after all)
- Again, no.
  - This would also lead to strong correlation:
  - Here it would be guaranteed that large service time means large interarrival time etc.
- Use separate streams, with separate seeds, for each separate random variable in your simulation!
  - Extensive discussion in Jain, p. 453 f.
- We have already seen that different seeds are necessary for different runs of a simulation, otherwise results would be identical
Separate RNG streams – Potential problem

- When using separate random streams, with different seeds, for different modeling purposes, one potential problem remains:
  - Generated random number sequences could overlap
    - Particularly problematic if, as is common, the same algorithm is chosen for all streams
  - Consequence: Make sure that seeds are separated sufficiently so that enough random numbers can be generated in each stream
    - Limits simulation length!

Overview

- Model and its state
- Implementing a queue
- Generating random numbers
- **Program structure**
  - Implementing next-event time advance algorithm
  - Processing events
  - Initialization
  - Updating statistics
  - Output and compare results
- Interpreting results
Program structure

- Recall the next-event time advance algorithm
- Basic structure

```java
while (stopping rule is false)
{
    determine what is the next event
    advance the time to this event
    process the event, generating and
    storing new event(s)
}
```

Determining next event

- Only two types of events here
  - New customer arrives
  - Customer finishes service
- Next event is whichever event comes first
  - Just compare the two variables describing customer arrival time and customer finish time
- What if two events happen “at the same time”?  
  - Sometimes, explicit priorities of events are specified
  - Otherwise, tie can be broken arbitrarily. Alternatively, give precedence to the event that has been generated first.
  - However, resolution algorithm should be deterministic!
Process event – New customer

- Create a new task describing a customer
  - Contains time to serve this customer
- Distinguish two cases
  - Server is currently idle
    - Set server to busy
    - Set the time of departure of this customer as one of the next events
  - Server is currently busy
    - Add this task to the queue
- Compute the arrival time of the next customer and store this as a future event
  - When better to take care of the need to have another customer?
  - This is a key technique! When event is generated, make provisions that the next event of the same type is generated!

Process event – customer finishes service

- Update number of processed customers (for stopping rule)
- Check if queue is empty
  - If yes
    - Set server to idle
    - There is no reasonable time for the “current customer” to finish service – set it to infinite -> this event will never be chosen
      - Can be obtained by calling std::numeric_limits<float>::infinity()
    - This is called poisoning an event
  - If no
    - Extract the next customer from queue
    - Schedule the departure event for this customer as a new future event (happens at current time + this customer’s service time)
How to initialize event times?

- First things that are evaluated in the actual main loop are event times
- Generate random time for the arrival of the first customer
- Set the departure time of the current = non-existent customer to infinity
  - Poison this event
- Arrival of a customer will hence be the first event that will be executed

Updating statistics

- When to update statistical counters, e.g. areas under respective curves?
- A simple way:
  - Always remember the time of the “last” event
  - Update the simulation clock
  - Compute time since last event
  - Update statistics
  - Process event, generate new events, etc.
  - Remember this simulation time as time of the last event
Updating statistics

- Some statistics have to be updated at other places
  - E.g., when extracting a customer from the queue, its waiting time is determined and should be counted into the metric
- Often useful to maintain additional information for statistical purposes
  - E.g., task description in queue contains the time it entered the queue, for easy reference

Actual code! – Version 1

- Please do look at the actual implementation for this example
- Structure
  - SIMQueue.hpp: Header file implementing (!) a template class SIMQueue, using an auxiliary class SIMQueueElement, which contains the actual information along with the possibility to link SIMQueueElement elements in a list. SIMQueue maintains a singly-linked list of SIMQueueElement elements in a fairly standard fashion
  - SIMTask.hpp/SIMTask.cpp: define and implement a pretty much trivial class SIMTask that describes a task in a SIMQueue by its arrival time and the processing time the task is going to need
  - SIMApp.cpp: the actual implementation of the main program. Instantiates a SIMQueue, dynamically creates SIMTask objects, implements next-event time advance algorithm
Overview

- Model and its state
- Implementing a queue
- Generating random numbers
- Program structure
- **Interpreting results**

Output results

- Once stopping rule is true, compute and output statistical results
- As we have simulated an M/M/1 queue, analytical results are available for comparison!
  - Let mean interarrival time = 1 / \( \lambda \), mean service time = 1 / \( \mu \)
  - Average utilization of the server = \( \lambda / \mu = \rho \)
  - Average waiting time = \( \rho (1/\mu) 1/(1-\rho) \)
  - Average length of the queue = \( \rho^2 / (1-\rho) \)
  - (Check e.g. the second part of this course or Rai Jain’s book on how to derive these values)
Output results

Some example results

<table>
<thead>
<tr>
<th>Mean interarrival time</th>
<th>Mean service time</th>
<th>Theoretical avg. utilization</th>
<th>Simulated avg. utilization</th>
<th>Theoretical avg. number in queue</th>
<th>Simulated avg. number in queue</th>
<th>Theoretical avg. waiting time</th>
<th>Simulated avg. waiting time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.5</td>
<td>0.5</td>
<td>0.49917</td>
<td>0.5</td>
<td>0.5</td>
<td>0.5</td>
<td>0.5</td>
</tr>
<tr>
<td>1</td>
<td>0.1</td>
<td>0.1</td>
<td>0.09983</td>
<td>0.01111</td>
<td>0.0111055</td>
<td>0.01111</td>
<td>0.011040</td>
</tr>
<tr>
<td>1</td>
<td>0.9</td>
<td>0.9</td>
<td>0.89588</td>
<td>8.1</td>
<td>7.17325</td>
<td>8.1</td>
<td>7.12282</td>
</tr>
<tr>
<td>2</td>
<td>0.8</td>
<td>0.4</td>
<td>0.39934</td>
<td>0.26667</td>
<td>0.271702</td>
<td>0.533333</td>
<td>0.542721</td>
</tr>
<tr>
<td>2</td>
<td>1.8</td>
<td>0.9</td>
<td>0.89589</td>
<td>8.1</td>
<td>7.17381</td>
<td>16.2</td>
<td>14.2456</td>
</tr>
</tbody>
</table>

Interpreting output results

- Not a single result matches the theoretical numbers exactly
- What happens if a single experiment is repeated with different seeds?
  - Example: mean interarrival = 1, mean service = 0.5

Seed mean interarrival time | Seed mean service time | Theoretical avg. utilization | Simulated avg. utilization | Theoretical avg. number in queue | Simulated avg. number in queue | Theoretical avg. waiting time | Simulated avg. waiting time |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>12345</td>
<td>6789</td>
<td>0.5</td>
<td>0.49121</td>
<td>0.5</td>
<td>0.482928</td>
<td>0.5</td>
<td>0.485767</td>
</tr>
<tr>
<td>4711</td>
<td>5434</td>
<td>0.5</td>
<td>0.49917</td>
<td>0.5</td>
<td>0.513099</td>
<td>0.5</td>
<td>0.51246</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>0.5</td>
<td>0.50340</td>
<td>0.5</td>
<td>0.49663</td>
<td>0.5</td>
<td>0.496924</td>
</tr>
</tbody>
</table>
Interpreting output results of multiple runs

- Some results are smaller, some are larger than theoretically predicted values
- Can many results be aggregated into a single result for a particular metric? E.g., by averaging?

<table>
<thead>
<tr>
<th>Seed mean interarrival time</th>
<th>Seed mean service time</th>
<th>Theoretical avg. utilization</th>
<th>Simulated avg. utilization</th>
<th>Theoretical avg. number in queue</th>
<th>Simulated avg. number in queue</th>
<th>Theoretical avg. waiting time</th>
<th>Simulated avg. waiting time</th>
</tr>
</thead>
<tbody>
<tr>
<td>12345</td>
<td>6789</td>
<td>0.5</td>
<td>0.49121</td>
<td>0.5</td>
<td>0.482928</td>
<td>0.5</td>
<td>0.485767</td>
</tr>
<tr>
<td>4711</td>
<td>5434</td>
<td>0.5</td>
<td>0.49917</td>
<td>0.5</td>
<td>0.513099</td>
<td>0.5</td>
<td>0.51246</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>0.5</td>
<td>0.50340</td>
<td>0.5</td>
<td>0.49663</td>
<td>0.5</td>
<td>0.496924</td>
</tr>
<tr>
<td>Average</td>
<td></td>
<td>0.4979</td>
<td></td>
<td>0.49755</td>
<td>0.49838</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Interpreting output results of multiple runs

- What is the actual meaning of such an average? Its relationship to theoretical values for a metric?
- How many simulated values are necessary to make “reasonable” statements about the real values?
  - Evidently, one value is insufficient
  - Average of three values: still off from the theoretical value
  - What does “reasonable” mean?
- Even worse: what to do when no theoretical value for a simulation metric is known?
  - Any possibilities to infer anything from simulated values?
- We will discuss this later in much detail: Stochastic confidence!
Interpreting output results: wrong numbers?

- Look at the results for high utilization scenarios:

<table>
<thead>
<tr>
<th>Mean interarrival time</th>
<th>Mean service time</th>
<th>Theoretical avg. utilization</th>
<th>Simulated avg. utilization</th>
<th>Theoretical avg. number in queue</th>
<th>Simulated avg. number in queue</th>
<th>Theoretical avg. waiting time</th>
<th>Simulated avg. waiting time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.9</td>
<td>0.8958</td>
<td>8.1</td>
<td>7.17325</td>
<td>8.1</td>
<td>7.12282</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1.8</td>
<td>0.8958</td>
<td>8.1</td>
<td>7.17381</td>
<td>16.2</td>
<td>14.2456</td>
<td></td>
</tr>
</tbody>
</table>

- Results for avg. queue length and avg. waiting time are quite different from theoretical values
  - Bug in the program?

Performance Evaluation (WS 13/14): 03 – A Simple Example

Interpreting output results: wrong numbers?

- Let’s have a look at the $1/\lambda = 1$, $1/\mu = 0.9$ case
- Consider stopping after an increasing number of customers

<table>
<thead>
<tr>
<th>Number of customers</th>
<th>Theoretical avg. utilization</th>
<th>Simulated avg. utilization</th>
<th>Theoretical avg. number in queue</th>
<th>Simulated avg. number in queue</th>
<th>Theoretical avg. waiting time</th>
<th>Simulated avg. waiting time</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>0.9</td>
<td>0.51922</td>
<td>8.1</td>
<td>0.205292</td>
<td>8.1</td>
<td>0.178378</td>
</tr>
<tr>
<td>100</td>
<td>0.9</td>
<td>0.9448</td>
<td>8.1</td>
<td>11.3715</td>
<td>8.1</td>
<td>10.4965</td>
</tr>
<tr>
<td>1000</td>
<td>0.9</td>
<td>0.8751</td>
<td>8.1</td>
<td>5.46195</td>
<td>8.1</td>
<td>5.4201</td>
</tr>
<tr>
<td>10000</td>
<td>0.9</td>
<td>0.8958</td>
<td>8.1</td>
<td>7.17381</td>
<td>8.1</td>
<td>7.12282</td>
</tr>
<tr>
<td>100000</td>
<td>0.9</td>
<td>0.8968</td>
<td>8.1</td>
<td>7.68953</td>
<td>8.1</td>
<td>7.717</td>
</tr>
<tr>
<td>1000000</td>
<td>0.9</td>
<td>0.899</td>
<td>8.1</td>
<td>7.94133</td>
<td>8.1</td>
<td>7.94867</td>
</tr>
</tbody>
</table>
Interpreting output results: longer = better?

- Longer runs seem to give better = more accurate results
- How long is long enough?
- When is it ok to stop a simulation?
- Strange things seem to happen at the beginning
  - A highly utilized server has a full queue
  - At the start of a simulation, the queue is empty
  - Is it ok to include these results into the statistics for the various metrics?
  - Not really! We are not interested in this initial phase, but rather what happens when the system is working in a steady state
  - How to detect the end of such initial phases?
- Again, we will come to back to this in detail! (“transient removal”)

Done!

- This is all that is needed to write and interpret an M/M/1 simulation!
- Any difficulties? What did you learn? How does the output of your program look like, how does it compare with theoretical results?
Some code cleanup – Version 2

- Keeping track of average queue length is a hassle
- Modify the queue class so as to keep track of the average number of elements in it
- Recall: This averaging is weighted by the \textit{time} these different levels are present in the queue
- Hence: Queue needs to know what the “current” time is that such operations take place
- \textbf{Class SIMTimedQueue (SIMTimedQueue.hpp) provides this functionality}
  - Main code is modified and cleaned up accordingly

Conclusion

- Identify necessary state and events that modify state
- Identify necessary information to store metrics
- Standard means of random number/distribution generation
  - Pay attention to seed selection
- Implement next-event time advance algorithm
- Routines to process the state changes due to an event
- Many statistics-related pitfalls: when to stop, when to start, …