Goal of this chapter

- Understand implementation issues and solutions for a more complicated example
- Develop the concept of a future event set and its use in a simulation
  - Along with appropriate, fast data structures for such sets
- Using object orientation in simulation programs
- Develop a typical programming style for simulations, based on object-oriented design of simulation programs
- Some reasons and cures for some subtle programming bugs
Overview

- **A complex example: single queue, multiple servers**
- Future event set
- Data structures for future event sets
- Objectifying simulation design
- Race conditions

---

**Single queue, multiple servers**

- Consider a system where multiple servers are used to serve a single queue
  - Often found in, e.g., check-in lanes at airports
- **Model:**
  - If at least one server is empty, arriving job will go to that server (ties are broken arbitrarily, e.g., in increasing numerical order)
  - If all servers are busy, arriving job will join end of queue
  - If server becomes idle, and queue not empty, first job in queue will be assigned to that server
  - Arrival process and service time as before
  - All servers are assumed to be identical and independent of each other
- This is called an M/M/k queue (k = # servers)
Implementing M/M/k

- Reusing classes `Tqueue`, `task` evidently possible (without any changes)
- How to structure the main program?
  - Changes against M/M/1 version discussed
- State information
  - Actual number of servers to use
  - State of every server must be described -> array
  - Statistic-gathering variables need to be extended: keep utilization of every server separate
    - How to interpret this? Think of tie-breaking rules between idle servers!
  - Fairly straightforward, really

Implementing M/M/k – Program structure

- Some obvious changes
  - Initialize all server-relevant variables, not just one
- Recall: Main parts of the program were
  - Identify next event
  - Process event
  - Generate new events
  - Update statistics whenever suitable
Identify next event

- Which happens first – new task or task completion?

- Up to $k$ tasks could be in progress
  - Identify the first task to complete

- Compare this task’s departure time to arrival time of new task

- This gives the next clock value

Process event: Task arrives

- When customer arrives, check all servers to see if idle server exists
- If idle server exists, assign this task to idle server
  - Tie breaking!
- Otherwise, put into queue
Process event: Task finishes service

- Pretty much identical to M/M/1 version
- Main difference: Server that becomes idle has to be identified and manipulated
  - When determining which server will finish first, also remember the index of this server for reference here

Discussion

- Updating statistics is straightforwardly extended from simpler version
- Processing the events is also reasonably increasing in complexity
- Identifying the next event, however, becomes inordinately more complex
  - Not clear how to generalize that even further without getting lost in application-specific details
- This is version 3 of the example program – look at source code on the course web page
- Can we not do any better?
Overview

- A complex example: single queue, multiple servers
- Future event set
- Data structures for future event sets
- Objectifying simulation design
- Race conditions

Restructuring event management

- Look closely how the next event is determined
  - A set of variables describe the times for all the next events
  - Always the nearest event is used
  - The kind of event determines which code is used to process that event
    - This is highly application-specific
  - Sometimes, additional information is also provided
    - E.g., the number of the server on which the task has been running
    - Also, highly application-specific information
Future event set

- What is actually needed?
- A means to describe a set of events that will occur in the future (future event set, FES)
- Each event is associated with
  - The time at which it will occur
  - The particular procedure that shall be called when this event occurs (to handle this event)
    - The so-called “handler function”
  - Additional information as parameters for this procedure

Future event set

- Operations to be performed on this set
  - Enter a new event
    - Along with additional information
    - In particular, the time of occurrence of the event
  - Remove (and return) the first event
    - Irrespective of the order in which events have been added, order of removal is only dictated by the explicitly given time for each event
  - If desired: functions for statistical purposes
- In essence, this is a priority queue
  - Well-known data structure; more details soon
Organization of the main program

- Use a priority queue to hold the future event set
- Main program is a loop that continues as long as stopping rule is not true (and there are events to be processed)
  - Extract next event from queue holding the FES
  - Set time to the time of this event, update statistics
  - Call the handler function for this event (included with the event)
    - Passing parameters as included in the event information to this procedure
- New events are generated by the handler functions themselves
  - Just put new events in the future event set, specifying their time of occurrence
  - Handler functions might event delete events from the FES, does not happen here
- Initialization: Just put one or more events in the future event set
  - No need to “poison” some kinds of events as they do not even exist yet

Implementation issues – Version 4

- Have a look at version 4 of our example program
- SIMEvent.hpp contains declaration of
  - Prototype for handler functions:
    typedef void (*handler_fct) (int);
    Handler functions only take a single integer as parameter
  - Class SIMEvent, containing time of occurrence, a handler function pointer, and some arbitrary data that is to be passed to the handler function
- SIMPriorityQueue.hpp defines a priority queue
  - Similar to Tqueue
  - Method push has a parameter “priority”, according to which the queue is ordered
  - Usually, priority and time of the event will be identical – priority is introduced here to make SIMPriorityQueue more general
Implementation issues – Version 4

- Main program has a SIMPriorityQueue of events
- Main loop as described above
- Event handler functions for arrival of task and departure of task
  - Essentially identical to the code blocks that were originally in the main loop
- Explicit function schedule_event
  - Puts an entry into the SIMPriorityQueue
  - Called by the handler functions
- All in all, a much clearer structure

Overview

- A complex example: single queue, multiple servers
- Future event set
- Data structures for future event sets
- Objectifying simulation design
- Race conditions
Implementing a priority queue

- PriorityQueue implements a simple sorted, single-linked list
  - Simple to implement
  - Remove of first element happens in $O(1)$
  - However, inserting an element takes $O(\text{length of queue})$ - expensive
  - Appropriate if the future event set is small

- Reduce search time during inserting
  - Subdivide the single list in a number of lists, each one only containing events for a certain time interval, in consecutive order
  - So-called indexed linear lists
  - A number of variations how to choose these intervals
    - All span the same time (equidistant)
    - Dynamically adjusted so that the number of elements in each list is constant
    - Only sort events that are “near in time”

Implementing a priority queue

- Faster: Heap
  - Keep the entries only semi-sorted
  - Divide-and-conquer approach: Think of the data as arranged in a tree
  - Invariant: Every node is smaller than its children
  - NO requirement on the relative priority of siblings!
  - Efficiently representable in an array (children of $i$ are in $2i$ and $2i+1$)

Array representation: 15, 19, 28, 50, 23, 39, 40
Implementing a priority queue

Operations on Heaps
- Remove top node: Choose the smaller of its children and move it to the top. Continue recursively in that child’s subtree.
- Inserting an entry: Add new entry to an arbitrarily chosen leaf. Check if the new node is larger than its parent. If yes, terminate. If no, exchange places with parent and recursively check with the new parent.
- Both operations are $O(\log(\text{number of elements in the heap}))$, fast implementations possible

Proper choice of data structure/algorithm for priority queue depends on different factors
- Average size of the future event set
  - Linked lists good for small FES
  - Heaps appropriate for large FES
  - Indexed lists are somewhere in the middle (and are the most complicated to implement)
- Distribution of the event hold time (simulated time between inserting an event and its occurrence)
- Why bother at all?
  - Generating and consuming events is the most basic activity of a discrete event simulation
  - Future event set algorithm is usually crucial for the required running time of a simulation program
Example from [Len96]

![Bar chart](image)

Fig. 2. Execution time of 10000 simulation steps, DEC ALPHA proc.

Overview

- A complex example: single queue, multiple servers
- Future event set
- Data structures for future event sets
- **Objectifying simulation design**
- Race conditions
Generalizing the program structure

- Let us come back to the overall program structure of version 4
- This structure – having a FES with events that define their handler functions – is very flexible
- New functionality could be added by defining new handlers and inserting events for these handlers
  - The underlying machinery of handling events remains the same
- However, much problem-specific functionality is still included in the main program

Generalizing program structure

- Would it not be nice to have such a machinery in place and only
  - Write own handlers,
  - Generate own events,
  - And provide own data structure for these handlers to work upon?
- How to organize the program to represent
  - General simulation functionality
  - Problem-specific functionality
  - separately?
Objectifying program structure

- Possible solution: Use object orientation
- Use classes/objects to represent entities of the simulation model
- Have objects communicate by exchanging messages
- General functionality is only used to
  - Organize exchange of these messages,
  - Create problem-specific objects,
  - Provide utility functions for statistics, etc.

Objects in an M/M/k model

- Separate functionality included in the model:
  - Load is generated independently of remainder of the model
  - Servers are independent entities
  - Some logic is necessary that manages the actual queue and assigns jobs to empty servers
- Hence, use objects of three different classes:
  - SIMLoadGen
  - SIMDispatcher
  - SIMServer
- Such objects are separate modules of a simulation program
Communication between objects

- These modules communicate *only* by exchanging messages
- Arrivals of messages are events
- Delivery of events/invocation of handler functions is organized by a general-purpose simulation framework
  - Independent of particular classes
- Have a look at such an implementation – it is version 5 of our simulation program!
  - For simplicity, the collection of statistics is not shown here, but it is straightforward to implement

A closer look: simulation framework

- Main program in `SIMApp.cpp`
  - Only used to create objects, generates an initial event to get simulation going, and execute event loop
  - Uses the well-known `PriorityQueue` class to manage event queue
- Important functions that are generally necessary
  - `const double& now()`: representing the current simulated time
  - `scheduleEvent(SIMEvent* e, const double& t)`: enqueue event `e` into the `PriorityQueue` priority queue such that event `e` will happen at time `t`
A closer look: simulation framework

- Events are represented using a class hierarchy
  - Starting with a simple class `SIMEvent`, containing a pointer to the recipient (destination object) of the event
  - Event-based classes contain methods to enable identification of the derived class
    - Maybe done even if only a pointer to a base class is available (dynamic_cast) – used e.g. in `SIMServer` (slow!!!)
    - Better: add IDs
  - Other types of events can be subclassed from this class, adding additional information to the event as needed
  - Here, classes `taskArrives` and `taskDone` are defined as special types of events

A closer look: implementing modules

- Any class that implements problem-specific functionality (providing modules) needs to be derived from abstract base class `module`
- Class `module` only defines a handler function to be called when the module receives an event:
  ```cpp
  virtual void handleEvent(SIMEvent* e) = 0;
  ```
- The classes `SIMLoadGen`, `SIMDispatcher`, and `SIMServer` are derived from `SIMModule`
A closer look: module SIMLoadGen

- A load generator contains the two random number generators for interarrival time and required service time
- It also needs to be told the dispatcher module: to whom shall the generated load (represented by SIMTaskArrives events) be delivered?
  - Done in the constructor of the load generator
- Event handler is simple: Any time an event arrives, do the following:
  - Generate a SIMTaskArrives event to be delivered to the dispatcher immediately (containing service time as parameter)
  - Generate a simple event to be delivered to the load generator itself after the randomly generated interarrival time has passed

A closer look: module SIMServer

- A server module knows about three things
  - Its identity
  - Whether it is idle or not
  - The dispatcher module it works for (in order to return task completion events to the dispatcher)
- A server module knows how to do two things
  - What to do when a new task arrives: handleTaskArrives()
  - What to do when the currently assigned task finishes: handleTaskDone()
A closer look: module SIMServer

- Tasks are assigned to a server by sending a corresponding event to it from the dispatcher
  - Server sets itself to BUSY
  - Schedules a SIMTaskDone event for itself
- At SIMTaskDone, server sets itself to IDLE and passes the event on to the dispatcher (to be delivered immediately)
- handleEvent() decides which kind of event arrived and calls the appropriate method

A closer look: module SIMDispatcher

- While load generator and server are fairly simple, dispatcher contains actual logic of the simulation
- Load generator sends SIMTaskArrives events to the dispatcher
  - Dispatcher scans set of servers by checking their idle status
  - If idle server is found, SIMTaskArrives event is immediately sent to the corresponding server
  - If all servers are busy, an entry in a SIMTimedQueue object is made
- At arrival of SIMTaskDone event, dispatcher attempts to assign a queued job (if any) to a now idle server
- Similar to the server, dispatcher’s handleEvent() calls appropriate methods depending on the type of the arriving event
That’s it! Object-oriented simulation!

- Take a look at the code – it really is much simpler than the description sounds
- Overall structure is simple and straightforward
- Most important points:
  - Strict separation of simulation engine from problem-specific modules and events
  - New event types and module types can easily be used by subclassing the corresponding classes, without the actual simulation framework even being aware of this

Programming style used here?

- The modules themselves are also rather simple and follow a certain pattern:
  - Wait for an event to arrive,
  - Decide what type of event it is, call the corresponding method that knows how to handle that event
  - Event handler modifies some of the module’s state (sets it to IDLE, makes entries in a queue, …)
  - Generate some new events
  - And again wait for a new event to arrive
- You should recognize this pattern: the modules are extended finite state machines!
  - Recall: Communication protocols are typically designed as eFSMs!
- **Typical programming style for discrete event simulation:** communicating extended finite state machines
Overview

- A complex example: single queue, multiple servers
- Future event set
- Data structures for future event sets
- Objectifying simulation design
- Race conditions

But… does it work?

- Trying to run the code shown as version 5, you will notice some error messages from the server!
- At some points in time, a task is assigned when the server is not idle
- Impossible!?! Dispatcher first checks a server’s idle status before it sends a SIMTaskArrives event to a server!
- How can it then be busy when a task arrives?
Assigning work to a busy server?

- Looking more closely at an execution, the problem only seems to happen when (at a certain time)
  - The queue is not empty
  - The load generator generates a job and sends it to the dispatcher
  - A server, at the same time, finishes a job
- Let us take a careful look at the event queue
  - Important point: in this implementation of the priority queue, events that happen at the same time are executed in the order in which they were entered into the queue!
  - Recall the discussion about tie breaking for simultaneous events?

Event queue snapshots at time t

- Server receives `SIMTaskDone`, sets itself idle, sends `SIMTaskDone` to dispatcher
  ```
  toServer: taskDone, t
toLoadGen: GenLoad, t
  ```

- Load generator generates load, sends `SIMTaskArrives` to dispatcher, sends `LoadGen` event to itself (t’ > t)
  ```
  toDispatcher: taskDone, t
toDispatcher: taskArrives, t
toLoadGen: GenLoad, t'
  ```
Event queue snapshots at time t

- Dispatcher receives `SIMTaskDone`, retrieves job from job queue, generates a `SIMTaskArrives` for server (which is idle at this moment!)

- Dispatcher receives `SIMTaskArrives`, scans server, finds the still empty server (the `SIMTaskArrives` event has not yet arrived at the server, even though everything happens “simultaneously”), and sends this job to the server as well!

Race condition

- Server will receive the two `SIMTaskArrives` events
  - The first one is ok
  - The second one would assign a job to an already busy server, which is impossible and generates an error message
- This is a typical example of a race condition
  - The two jobs (one from the queue, one from the load generator) are “simultaneously” assigned to the same server, because the state information in the server has not been updated in time (“immediately, but not soon enough”)
- Dangerous and often difficult to find problem in simulation programs!
  - Sometimes even in commercial tools, where – even worse – no source code is available
Race conditions – ways out?

- Assign priorities to different types of events
  - In this example, SIMTaskArrives message to servers are more important than SIMTaskArrives messages to the dispatcher, as state information in the server needs to be updated to reflect the decision already taken by the dispatcher
  - Priority queue is not only sorted by time, but also by priority (which one is the primary key?)
  - Often simple to do in small simulations, difficult to handle in large cases

Race conditions – ways out?

- Break information hiding
  - In a sense, being idle/busy is a property of a server and should be represented there and no where else
    - Avoid redundant representation of state is a basic rule of object orientation
  - However, this caused the dispatcher’s need to communicate with the server to obtain its idle status
  - Representing this state in the dispatcher (which does know the state) removes this need
  - Again: simple in small programs, but can become a nightmare in larger systems when the redundant information is not updated correctly in all replicas
Race conditions – ways out?

- Be extremely careful about mixing different ways of information flow
  - In this example, the dispatcher assigned jobs by means of scheduling events, yet retrieved information from the server module via C++ method calls
  - These types of information flows are not synchronized, hence the query “Are you idle?” could overtake the assignment of the job
  - Querying the server with special events is possible, server answers with corresponding events – should avoid race conditions of this type
  - Disadvantage: can multiply the number of event/message types, runtime overhead for scheduling events is considerably higher than method invocations

Repairing this example?

- All three methods are feasible here – no cure-all to all problems
- Solutions not shown – left as an exercise 😊
  - I will be glad to include any good solutions into next year’s course
- But this is quite a mess, really: Could we not have a tool that takes care of some of these issues?
- Yes – we will look at such a tool in the following chapter
  - Nevertheless, simulation program logic must be sound no matter which tool is used
Conclusion

- Concept of future event set and event handlers
  - Think of it in terms of individual events, along with appropriate handlers
  - Structure program as to manage events in a general fashion
- Future event set is a priority queue, with time of occurrence of events as priority
- Algorithms for implementing priority queues
- Modules communicating via events, based on object-oriented simulation design, and simple to use and represent communicating extended finite state machines
- Race conditions can introduce subtle bugs into simulations, often because of bad design, often no really clean solution
- And what about your program’s results?