Chapter 2

Modeling Complex Systems

CONTENTS

2.1 INTRODUCTION

- Complex systems usually require complex models – difficult to code from scratch in general-purpose language
- Need some help, more simulation-oriented software
- In this chapter, will discuss list processing, a central activity in most simulations (and high-level simulation software)
- Develop and illustrate a set of support routines, **`simlib`**, that does several "routine" simulation tasks
  – List processing, event-list management, random-number and variate generation, statistics collection, output reporting
  – Will do this in C … FORTRAN codes available on book's website
- **`simlib`** is not a "real" real simulation language
  – Doing it here to illustrate what's behind commercial simulation software, enable modeling of more complex systems

2.2 LIST PROCESSING IN SIMULATION

- Most simulations involve *lists*
  - Queues, event list, others
  - A list is composed of *records*
- *Record*: Usually corresponds to an object in the list
  - By convention, a record is represented as a row in a two-dimensional array (matrix) representing the list
  - A person in a queue list, an event in the event list
  - A record is composed of one or more *attributes*
- *Attribute*: A data field of each record
  - By convention, attributes are in columns
  - Examples of records (lines) of attributes (columns):
    - Queue list: [time of arrival, customer type, service requirement, priority, …]
    - Event list: [event time, event type, possibly other attributes of the event]

2.2.1 Approaches to Storing Lists in a Computer

- *Sequential* allocation – approach used in Chap. 1
  - Records are in physically adjacent storage locations in the list, one record after another
  - Logical position = physical position
- *Linked* allocation
  - Logical location need not be the same as physical location
  - Each record contains its usual attributes, plus *pointers* (or *links*)
    - *Successor link* (or *front pointer*) – physical location (row number) of the record that's logically next in the list
    - *Predecessor link* (or *back pointer*) – physical location of the record that's logically before this one in the list
  - Each list has head pointer, tail pointer giving physical location of (logically) first and last records

2.2.1 Approaches to Storing Lists in a Computer (cont'd.)

- Advantages of linked over sequential allocation
  - Adding, deleting, inserting, moving records involves far fewer operations, so is much faster … critical for event-list management
    - Sequential allocation – have to move records around physically, copying all the attributes
    - Linked allocation – just readjust a few pointers, leave the record and attribute data physically where they are
  - Reduce memory requirements without increasing chance of list overflow
    - Multiple lists can occupy the same physical storage area … can grow and shrink more flexibly than if they have their own storage area
  - Provides a general modeling framework for list processing, which composes a lot of the modeling, computing in many simulations

2.2.2 Linked Storage Allocation

- Will treat *doubly*-linked lists (could define *singly*-linked)
- Physical storage area for all lists (max of, say, 25 lists)
  – Array with (say) 15 rows, 4 columns for attributes, a column for back pointers, a column for forward pointers
  – Also need array with 25 rows, 2 columns for head and tail pointers of each list

Physical row	Backward pointer	attrib_1	attrib_2	attrib_3	attrib_4	Forward pointer
1						
2						
3						
4						
5						
6						
7						
8						
9						
10						
11						
12						
13						
14						
15						

Head and tail pointers:

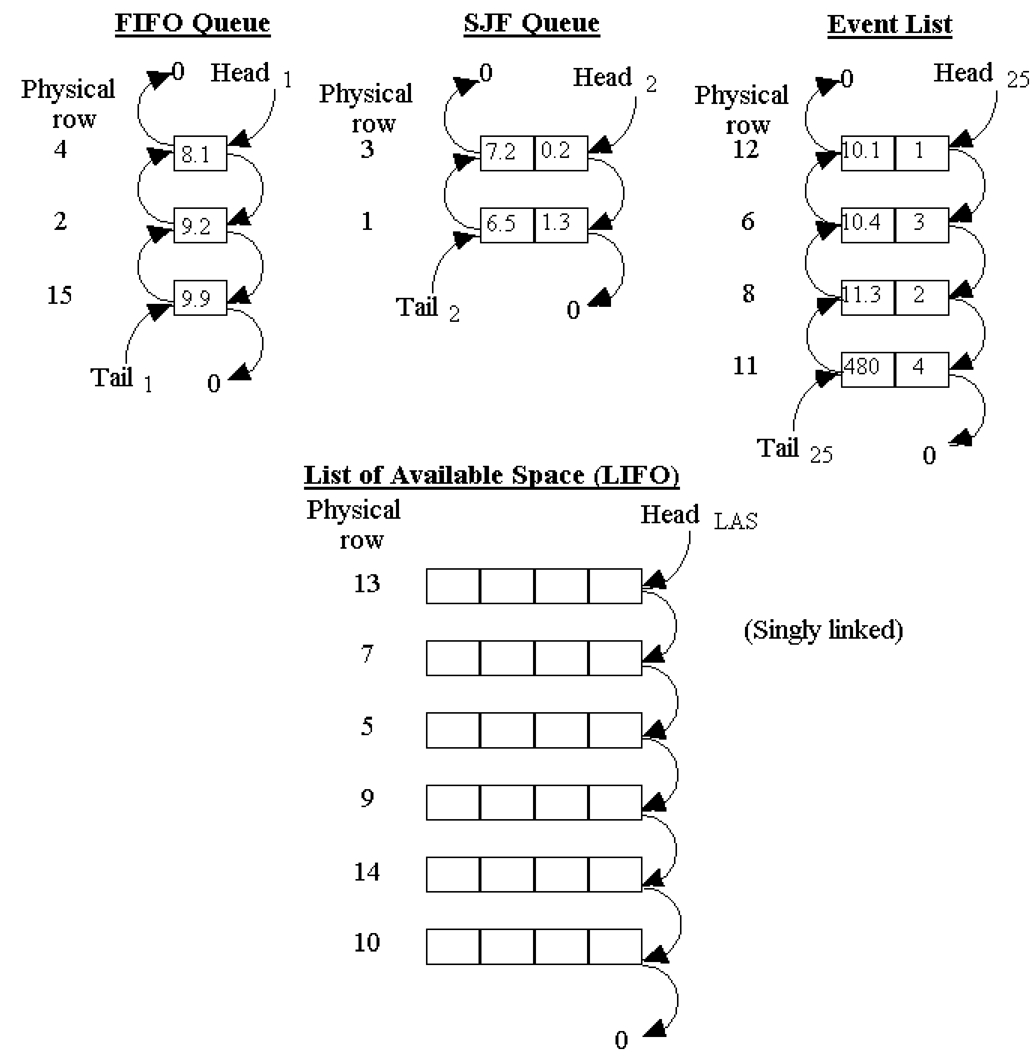
List number	Head pointer	Tail pointer
1		
2		
3		
. . .	. . .	. . .
24		
25		

2.2.2 Linked Storage Allocation (cont'd.)

- Example: Two queues (FIFO, Shortest-Job-First), event list (see text for different examples)
  - FIFO queue: attrib_1 = time of arrival
  - SJF queue: attrib_1 = time of arrival, attrib_2 = service requirement; insert new records (customers) to keep list ranked in increasing order on attrib_2; remove next customer to serve off top of list
  - Event list: attrib_1 = (future) event time, attrib_2 = event type

2.2.2 Linked Storage Allocation (cont'd.)

Logically:



Physical setup:

Physical row	Backward pointer	$attrib_1$	$attrib_2$	$attrib_3$	$attrib_4$	Forward pointer
1	3	6.5	1.3			0
2	4	9.2				15
3	0	7.2	0.2			1
4	0	8.1				2
5						9
6	12	10.4	3			8
7						5
8	6	11.3	2			11
9						14
10						0
11	8	480	4			0
12	0	10.1	1			6
13						7
14						10
15	2	9.9				0

Head and tail pointers:

List number	Head pointer	Tail pointer
1	4	15
2	3	1
3	0	0
. . .	. . .	. . .
24	0	0
25	12	11

Initially:

All lists empty:
$Head_i = 0 = Tail_i$ for all list numbers i

All rows in list of available space in order:
$Head_{LAS} = 1$
Front $pointer_j = j + 1$ for all rows $j = 1, 2, ...,$ last row – 1
Front $pointer_{\text{last row}} = 0$

Need: Utility code to manage all this …

2.3 A SIMPLE SIMULATION LANGUAGE: simlib

- Some C functions – rudimentary simulation "language"
  - Source code in Appendix 2A, and on book's website
  - Also available in "legacy" FORTRAN on book's website
- Capabilities
  - List processing (pointers, file a record, remove a record)
  - Processes event list
  - Tallies statistical counters
  - Generate random numbers, variates from some distributions
  - Provide "standard" output (optional)
- Not a "real" simulation language – incomplete, inefficient
  - But illustrates what's in real simulation software, how it works

2.3 A Simple Simulation Language: **`simlib`**

(cont'd.)

- Heart of **`simlib`** is a collection of doubly linked lists
  - All live together in dynamic memory – space allocated as new records are filed in lists, freed when records are removed from lists
  - Maximum of 25 lists
  - Records in lists can have a maximum of 10 attributes
  - Data are stored as type **`float`**
  - Uses dynamic storage allocation, so total number of records in all lists is limited only by hardware
  - List 25 always reserved for event list
    - Always: attribute 1 = (future) event time, attribute 2 = event type
    - attributes 3-10 can be used for other event attributes
    - Kept sorted in increasing order on event time – next event is always on top
- User must **`#include simlib.h`** for required declarations, definitions
  - **`simlib.h`** in turn **`#include`**s **`simlibdefs.h`**

2.3 A Simple Simulation Language: **simlib**

(cont'd.)

- Key **simlib** variables and constants
  - **sim_time** = simulation clock; updated by **simlib**
  - **next_event_type** = type of next event; determined by **simlib**
  - **transfer[i]**: float array indexed on **i** = 1, 2, …, 10 for transferring attributes of records into and out of lists
  - **maxatr** = max number of attributes in any list; defaults to 10, but user can initialize to < 10 for improved efficiency (cannot be set to < 4 due to the way **simlib** works)
  - **list_size[list]** = current number of records in list **list**; maintained by **simlib**
  - **list_rank[list]** = attribute number (if any) on which list **list** is to be ranked (incr. or decr.); must be initialized by user
  - **FIRST**, **LAST**, **INCREASING**, **DECREASING**: symbolic constants for options of filing a record into a list
  - **LIST_EVENT**, **EVENT_TIME**, **EVENT_TYPE**: symbolic constants for event-list number, attribute number of event time, attribute number of event type

2.3 A Simple Simulation Language: `simlib`

(cont'd.)

- Description of the 19 functions in **`simlib`**:

  `init_simlib`: Invoke at beginning of each simulation run from the (user-written) main function to allocate storage for lists, initialize all pointers, set clock to 0, sets event list for proper ranking on event time, defaults **`maxatr`** to 10, sets all statistical counters to 0

  `list_file(option, list)`: File a record (user must pre-load attributes into **`transfer`** array) in list **`list`**, according to **`option`**:

1 or **`FIRST`**	File record as the new beginning of the list
2 or **`LAST`**	File record as the new end of the list
3 or **`INCREASING`**	File record to keep list in increasing order on attribute **`list_rank[list]`** (as pre-set by user)
4 or **`DECREASING`**	File record to keep list in increasing order on attribute **`list_rank[list]`**

2.3 A Simple Simulation Language: `simlib`

(cont'd.)

`list_remove(option, list)`: Remove a record from list **`list`** and copy its attributes into the **`transfer`** array, according to **`option`**:

1 or **`FIRST`**	Remove the first record from the list
2 or **`LAST`**	Remove the last record from the list

`timing`: Invoke from main function to remove the first record from the event list (the next event), advance the clock to the time of the next event, and set **`next_event_type`** to its type; if attributes beyond 1 and 2 are used in the event list their values are copied into the corresponding spots in the **`transfer`** array

`event_schedule(time_of_event, type_of_event)`: Invoke to schedule an event at the indicated time of the indicated type; if attributes beyond 1 and 2 are used in the event list their values must be pre-set in the **`transfer`** array

2.3 A Simple Simulation Language: `simlib`

(cont'd.)

- **`event_cancel(event_type)`**: Cancel (remove) the first (most imminent) event of type **`event_type`** from the event list, if there is one, and copy its attributes into the **`transfer`** array
- **`sampst(value, variable)`**: Accumulate and summarize discrete-time process data
  - Can maintain up to 20 separate "registers" (**`sampst`** variables); 3 functions:
    - During the simulation: record a value already placed in **`value`** in **`sampst`** variable **`variable`**: **`sampst (value, variable)`**
    - At end of simulation: invoke **`sampst`** with the *negative* of the variable desired (**`value`** doesn't matter); get in **`transfer`** array the mean (1), number of observations (2), max (3), min (4); name **`sampst`** also has mean
    - To reset all **`sampst`** variables: **`sampst (0.0, 0)`**; normally done at initialization, but could be done at any time during the simulation

2.3 A Simple Simulation Language: `simlib`

(cont'd.)

`timest(value, variable)`: Accumulate and summarize continuous-time process data

- Can maintain up to 20 separate "registers" (**`timest`** variables), separate from **`sampst`** variables; 3 functions:
  - During the simulation: record a *new* value (after the change in its level) already placed in **`value`** in **`timest`** variable **`variable`**: **`timest (value, variable)`**
  - At end of simulation: invoke **`timest`** with the *negative* of the variable desired (**`value`** doesn't matter); get in **`transfer`** array the mean (1), max (2), min (3); name **`timest`** also has mean
  - To reset all **`timest`** variables: **`timest (0.0, 0)`**; normally done at initialization, but could be done at any time during the simulation

`filest(list)`: Produces summary statistics on number of records in list **`list`** up to time of invocation

- Get in **`transfer`** array the mean (1), max (2), min (3) number of records in list list; name **`filest`** also has mean
- Why? List lengths can have physical meaning (queue length, server status)

2.3 A Simple Simulation Language: `simlib`

(cont'd.)

- **`out_sampst(unit, lowvar, highvar)`**: Write to file **`unit`** summary statistics (mean, number of values, max, min) on **`sampst`** variables **`lowvar`** through **`highvar`**
  - Get "standard" output format, 80 characters wide
  - Alternative to final invocation of **`sampst`** (still must initialize and use **`sampst`** along the way)
- **`out_timest(unit, lowvar, highvar)`**: Like **`out_sampst`** but for **`timest`** variables
- **`out_filest(unit, lowfile, highfile)`**: Like **`out_sampst`** but for summary statistics on number of records in lists **`lowfile`** through **`highfile`**

2.3 A Simple Simulation Language: `simlib`

(cont'd.)

`expon(mean, stream)`: Returns in its name a variate from an exponential distribution with mean **`mean`**, using random-number "stream" **`stream`** (more on streams in Chaps. 7, 11)

- Uses random-number generator **`lcgrand`** (see below, and Chap. 7)

`random_integer(prob_distrib[], stream)`: Returns a variate from a discrete probability distribution with *cumulative* distribution in the array **`prob_distrib`**, using stream **`stream`**

- Assumes range is 1, 2, …, k, with $k \leq 25$
- User prespecifies **`prob_distrib[i]`** to be $P(X \leq \mathbf{i})$ for **`i`** = 1, 2, …, k
- Note that **`prob_distrib[`***k***`]`** should be specified as 1.0

`uniform(a, b, stream)`: Returns a variate from a continuous uniform distribution on [**`a`**, **`b`**], using stream **`stream`**

`erlang(m, mean, stream)`: Returns a variate from an **`m`**-Erlang distribution (see below, and Chaps. 6, 8) with mean **`mean`**, using stream **`stream`**

2.3 A Simple Simulation Language: `simlib`

(cont'd.)

`lcgrand(stream)`: Random-number generator, returns a variate from the (continuous) U(0, 1) distribution, using stream **`stream`**

- See Chap. 7 for algorithm, code
- **`stream`** can be 1, 2, …, 100
- When using **`simlib`**, no need to **`#include lcgrand.h`** since **`simlib.h`**, already **`#include`**d, has the definitions needed for **`lcgrand`**

`lcgrandst(zset, stream)`: Sets the random-number "seed" for stream **`stream`** to **`zset`**

`lcgrandgt(stream)`: Returns the current underlying integer for stream **`stream`**

- **See Chap. 7 for details**
- **Use – at end of a run, use `lcgrandgt` to get the current underlying integer**
- **Use this value as `zset` in `lcgrandst` for a later run that will be the same as extending the earlier run**

2.3 A Simple Simulation Language: `simlib`

(cont'd.)

- Using **`simlib`**
  - Still up to user to determine events, write main function and event functions (and maybe other functions), but **`simlib`** functions makes this easier
  - Determine what lists are needed, what their attributes are
  - Determine **`sampst`**, **`timest`** variables
  - Determine and assign usage of random-number streams
  - **`simlib`** variables take the place of many state, accumulator variables, but user may still need to declare some global or local variables

Numbering is largely arbitrary, but it's *essential* to:
- ***Decide ahead of time***
- ***Write it all down***
- ***Be consistent***

2.3 A Simple Simulation Language: **`simlib`**

(cont'd.)

- Typical activities in the user-written main function (roughly, but not necessarily exactly, in this order):
  1. Read/write input parameters
  2. Invoke **`init_simlib`** to initialize **`simlib`**'s variables
  3. (Maybe) Set **`lrank_list[list]`** to attribute number for ranked lists
  4. (Speed option) Set **`maxatr`** to max number of attributes per list
  5. (Maybe) **`timest`** to initialize any **`timest`** variables to nonzero values
  6. Initialize event list via **`event_schedule`** for each event scheduled at time 0
     - If using more than first two attributes in event list (time, type), must set **`transfer[3]`**, **`transfer[4]`**, … before invoking **`event_schedule`**
     - Events not initially to be scheduled are just left out of the event list
  7. Invoke timing to determine **`next_event_type`** and advance clock
  8. Invoke appropriate event function, perhaps with **`case`** statement
  9. At end of simulation, invoke user-written report generator that usually uses **`sampst`**, **`timest`**, **`filest`**, **`out_sampst`**, **`out_timest`**, **`out_filest`**

2.3 A Simple Simulation Language: `simlib`

(cont'd.)

- Things to do in your program
  - Maintain lists via **`list_file`**, **`list_remove`**, together with **`transfer`** to communicate attribute data to and from lists
  - Gather statistics via **`sampst`** and **`timest`**
  - Update event list via **`event_schedule`**
- Error checking – cannot check for all kinds of errors, but some opportunities to do some things in simulation "software" – so **`simlib`** checks/traps for:
  - Time reversal (scheduling an event to happen in the past)
  - Illegal list numbers, variable numbers
  - Trying to remove a record from an empty list

2.4 SINGLE-SERVER QUEUEING SIMULATION WITH `simlib`; 2.4.1 Problem Statement; 2.4.2 `simlib` Program

- Same model as in Chap. 1
  - Original 1000-delay stopping rule
  - Same events (1 = arrival, 2 = departure)
- **`simlib`** lists, attributes:

  1 = queue, attributes = [time of arrival to queue]

  2 = server, no attributes (dummy list for utilization)

  25 = event list, attributes = [event time, event type]
- **`sampst`** variable: 1 = delays in queue
- **`timest`** variables: none (use **`filest`** or **`out_filest`**)
- Random-number streams:
  1 = interarrivals, 2 = service times

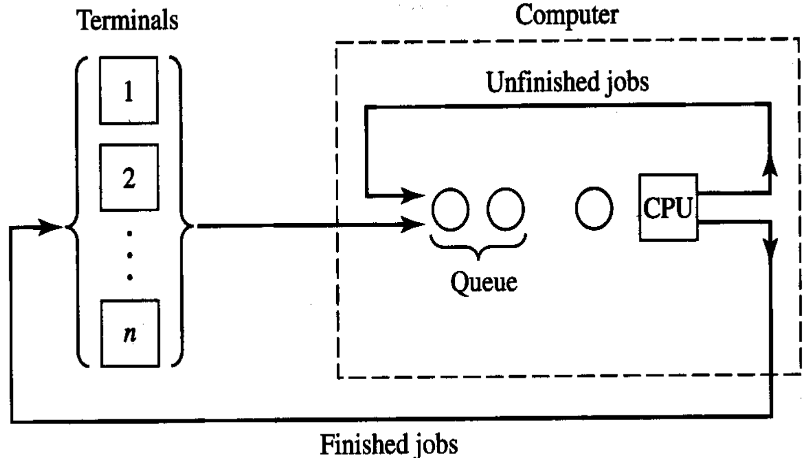
2.4.2 `simlib` Program (cont'd.); 2.4.3 Simulation Output and Discussion

- Refer to pp. 124-129 in the book (Figures 2.6-2.12) and the file **`mm1smlb.c`**
  - Figure 2.6 – external definitions (at top of file)
  - Figure 2.7 – function **`main`**
  - Figure 2.8 – function **`init_model`**
  - Figure 2.9 – function **`arrive`**
  - Figure 2.10 – function **`depart`**
  - Figure 2.11 – function **`report`**
  - Figure 2.12 – output report **`mm1smlb.out`**
    - Results differ from Chap. 1 (same model, input parameters, start/stop rules) – why?

2.5 TIME-SHARED COMPUTER MODEL; 2.5.1 Problem Statement

Think times of jobs at terminals:
Expon (mean = 25 sec.)

Fixed number of terminals (and jobs), *n*



Processing times of jobs at CPU:
Expon (mean = 0.8 sec.)

- Processing rule: *round-robin*
  - Each visit to CPU is $\leq q = 0.1$ sec. (a *quantum*)
    - If job still needs $> q$ sec., it gets q sec, then kicked out
    - If job still needs $\leq q$ sec., it gets what it needs, returns to its terminal
  - Compromise between extremes of FIFO ($q = \infty$) and SJF ($q = 0$)
- Swap time: $\tau = 0.015$ sec. Elapse whenever a job enters CPU before processing starts
- *Response time* of a job = (time job returns to terminal) – (time it left its terminal)

2.5.1 Problem Statement (cont'd.)

- Initially, computer empty and idle, all *n* jobs in the think state at their terminals
- Stopping rule: 1000 response times collected
- Output:
  - Average of the response times
  - Time-average number of jobs in queue for the CPU
  - CPU utilization
- Question: how many terminals (*n*) can be loaded on and hold average response time to under 30 sec.?

2.5.2 **`simlib`** Program

- Events

  1 = Arrival of a job to the computer (at end of a think time)

  2 = A job leaves the CPU (done or kicked out)

  3 = End simulation (scheduled at time 1000th job gets done, for *now*)

  (Also, have "utility" non-event function **`start_CPU_run`** to remove first job from queue, put it into the CPU)

- **`simlib`** lists, attributes:

  1 = queue, attributes = [time of arrival of job to computer, remaining service time]

  2 = CPU, attributes = [time of arrival of job to computer, remaining service time]

  - In CPU, if remaining service time > 0 then job has this much CPU time needed after current CPU pass; if remaining service time ≤ 0, job will be done after this pass

  25 = event list, attributes = [event time, event type]

2.5.2 `simlib` Program (cont'd.); 2.5.3 Simulation Output and Discussion

- **`sampst`** variable: 1 = response times
- **`timest`** variables: none (use **`filest`** or **`out_filest`**)
- Random-number streams:
  1 = think times, 2 = service times
- Refer to pp. 133-141 in the book (Figures 2.15-2.24) and the file **`tscomp.c`**
  - Figure 2.15 – external definitions (at top of file)
  - Figure 2.16 – function **`main`**
  - Figure 2.18 – function **`a`**rrive (flowchart: Figure 2.17)
  - Figure 2.20 – function **`start_cpu_run`** (flowchart: Figure 2.19)
  - Figure 2.22 – function **`end_cpu_run`** (flowchart: Figure 2.21)
  - Figure 2.23 – function **`report`**
  - Figure 2.24 – output report **`tscomp.out`**
    - Looks like max *n* is a little under 60 – sure?

2.8 EFFICIENT EVENT-LIST MANIPULATION

- Have seen two different ways to store, handle the event list
  - Sequential by event type – search for smallest entry for next event
    - Must always look at the whole event list
  - Ranked in increasing order by event time – insert correctly, take next event from top
    - Event insertion might not require looking at the whole list
- In large simulations with many events, event-list processing can consume much of the total computing time (e.g., 40%)
  - Generally worth it to try hard to manage the event list efficiently
  - Better methods – binary search, storing event list as a tree or heap
  - Exploit special structure of model to speed up event-list processing
    - e.g., if time a record stays on event list is approximately the same for all events, insert a new event via a bottom-to-top search