Simulation

Discrete Event Simulation

Terminology

- Entity - a component of the system that requires explicit representation in the model
- Attribute - a property of an entity
- State - a variable describing the system at a given point in time
- Event - instantaneous occurrence that changes the state of the system
- Activity - a duration of time started and terminated by related events
- Set - a collection of associated entities, ordered in some logical fashion

Event Scheduling Approach

- Concentrate on events and their effect on system state
- Future events are ordered according to event time in an event list
- Simulated time is kept in *clock* variable

Simulation Program skeleton

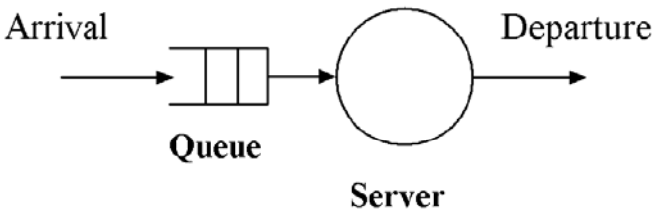
- Initialization
  - initialize *clock* to zero
  - initialize state variables, sets, and statistical counters
  - initialize event list (with known future events)
- Main loop (repeat until condition for terminating simulation is met)
  - determine the most imminent event and remove it from the event list (suppose this event is of type *i*)
  - advance *clock* to the time of this event
  - invoke event routine for type *i*

Simulation Program skeleton

- Event routine (a separate routine for each event type)
  - update state variables and sets
  - update statistical counters
  - when required, add future events to event list
- Report generator
- invoked when simulation is terminated
- compute and output performance measures of interest

Example - Single Server Queue

Infinite population model
(open model)



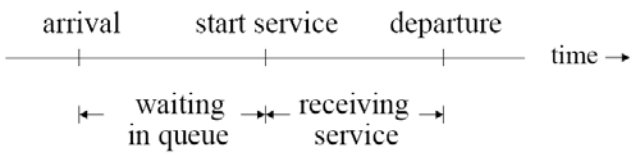
Example - Single Server Queue

Entities	**Attributes**
server	service rate
customer	service requirement
system	interarrival time
	note: service time = service requirement/service rate
State variables:	status of server (busy/idle) number of customers in system

Example - Single Server Queue

Events

Activities



Set customers in queue

Single Server Queue Model

- Assumptions
- interarrival times are independent of system state (similarly for service times)
- interarrival times are independent of each other and have identical probability distribution (similarly for service times)
- FCFS scheduling
- system is empty at time zero
- arrival of first customer occurs after the first interarrival time
- simulation terminates when the *m-th* customer starts service

Single Server Queue Model

- Input parameters
  - interarrival time distribution
  - service time distribution
- Performance measures of interest
  - mean waiting time, $\overline{W}$
  - mean number of customers in system, $\overline{N}$

Single Server Queue Model

- State variables
  - *status* = server status (busy or idle)
  - *n* = number of customers in system
- Statistical counters
  - *nw* = number of waiting times accumulated
  - *sw* = sum of accumulated waiting times
  - *sa* = sum of accumulated areas (for calculating $\overline{N}$)
  - *last_event* = time of last event when accumulating area

Single Server Queue Model

- Sets
  - *event_list*
  - *queue*
- Event types
  - type 1: *arrival*
  - type 2: *start_service*
  - type 3: *departure*

Single Server Queue Model

- Initialization
  - *clock* = 0
  - *status* = idle
  - *n* = 0
  - *nw* = *sw* = 0
  - *last_event* = 0
  - *sa* = 0
  - initialize *queue* to empty
  - initialize *event_list* to empty
  - determine *inter_t*, the first interarrival time
  - schedule an arrival event to occur at *clock* + *inter_t*

Single Server Queue Model

- Main loop (repeat until condition for terminating simulation is met)
  - determine the most imminent event and remove it from the event list (suppose this event is of type *i* and occurs at time *t*)
  - *clock* = *t*
  - *sa* = *sa* + (*clock* - *last_event*) * *n*
  - *last_event* = *clock*
  - invoke event routine for type *i*

Single Server Queue Model

- *arrival* event – type 1
  - determine *inter_t*, the interarrival time between the current and next arrivals
  - schedule an arrival event to occur at *clock* + *inter_t*
  - $n = n + 1$
  - enter arriving customer to end of *queue*, and save its time of arrival (given by *clock*)
  - if *status* is idle, invoke routine for *start_service* event

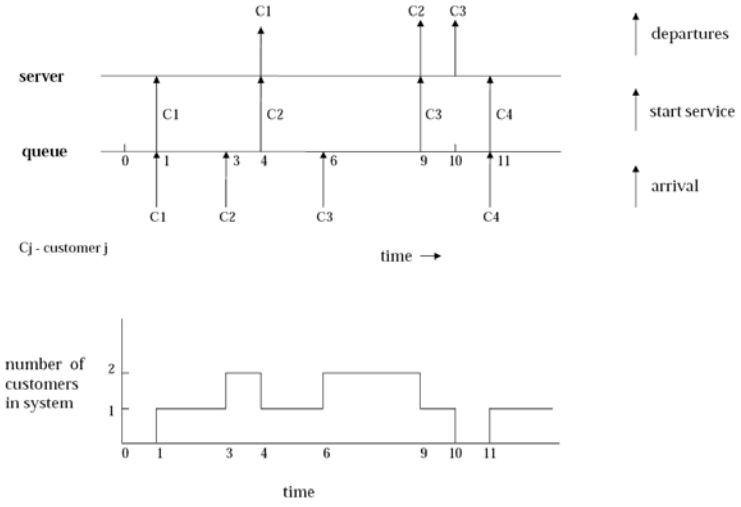
Single Server Queue Model

- *start_service* event – type 2
  - remove customer from front of *queue*, and retrieve time of arrival (*t_arrival*)
  - $nw = nw + 1$
  - $sw = sw + (clock - t\_arrival)$
  - if $nw = m$ (condition for terminating simulation), exit main loop
  - *status* = busy
  - determine *serv_t*, the service time of customer
  - schedule a *departure* event to occur at *clock* + *serv_t*

Single Server Queue Model

- *departure* event – type 3
  - $n = n - 1$
  - *status* = idle
  - if $n > 0$, invoke event routine for *start_service* event
- Report generator
  - mean waiting time $\overline{W} = sw / nw$
  - mean number of customers in system $\overline{N} = sa / clock$
  - output results

Single Server Queue Model

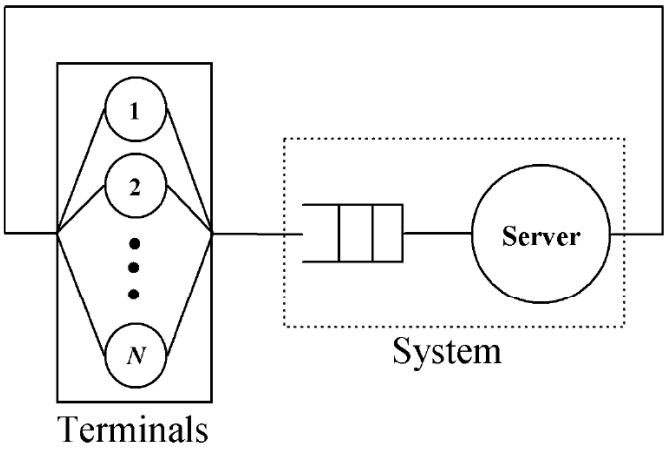


Notation: A - arrival event, D - departure event
(Cj, x) - customer j in queue, time of arrival of this customer is x
n - number of customers in system

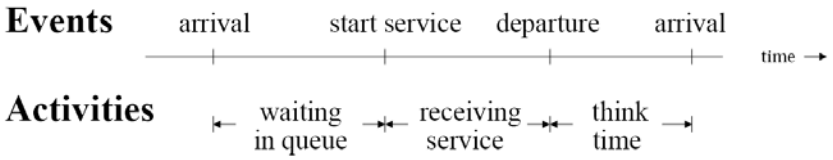
clock	event	status	n	event list	queue	nw	sw
0	--	idle	0	A at 1	empty	0	0
1	A	busy	1	A at 3, D at 4	empty	1	0
3	A	busy	2	D at 4, A at 6	(C2, 3)	1	0
4	D	busy	1	A at 6, D at 9	empty	2	1
6	A	busy	2	D at 9, A at 11	(C3, 6)	2	1
9	D	busy	1	D at 10, A at 11	empty	3	4
10	D	idle	0	A at 11	empty	3	4
11	A	busy	1	A at 15, D at 17	empty	4	4

mean waiting time = sw/nw = 1.0

Example - Single Server Queue (Finite population, closed model)



Finite Population Model



Finite Population Model

- Assumptions
  - think times are independent of system state (similarly for service times)
  - think times are independent of each other and have identical probability distribution (similarly for service times)
  - FCFS scheduling system is empty at time zero
  - for each of the *N* users, the first request is submitted after a think time
  - simulation terminates at time = *term_sim*

Finite Population Model

- Initialization
  - *clock* = 0
  - *status* = idle
  - *n* = 0
  - initialize *queue* to empty
  - initialize *event_list* to empty
  - for user *j* (*j* = 1 to *N*)
    determine *think_t*, a think time of user *j*, and
    schedule an *arrival* event at *clock* + *think_t*
    end for
  - schedule an *end_simulation* event at *term_sim*

Finite Population Model

- *arrival* event
  - *n* = *n* + 1
  - enter arriving customer to end of *queue*
  - if *status* is idle, invoke routine for *start_service* event
- *start_service* event
  - remove customer from front of *queue*
  - *status* = busy
  - determine *serv_t*, the service time of customer
  - schedule a *departure* event to occur at *clock* + *serv_t*

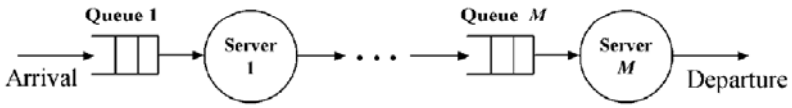
Finite Population Model

- *departure* event
  - $n = n - 1$
  - *status* = idle
  - if $n > 0$, invoke event routine for *start_service*
  - determine *think_t*
  - schedule an *arrival* event at *clock* + *think_t*
- *end-simulation* event
  - exit main loop

Model Development for Complex Systems

- Basic technique
  - identify the subsystems and develop model for each subsystem
  - characterize the interaction among subsystems

Example - Tandem Queue, *M* Stages

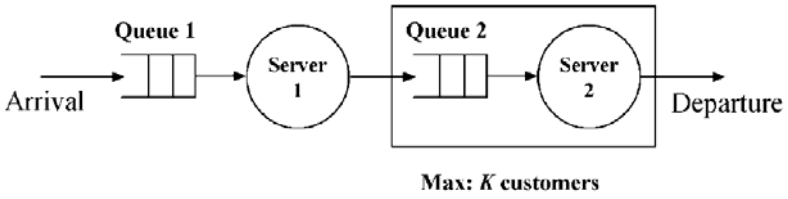


- Subsystems and interaction
  - *M* subsystems - one for each stage
  - a departure from stage *i* becomes an arrival to stage *i*+1 (*i* = 1 to *M*-1)

Simulation Program

- Use event routines for single server queue model for each of the *M* stages
- Modifications to implement tandem queue:
  - *departure* event : for stage *i* (*i* = 1 to *M*-1) - add the step
    - invoke routine for *arrival* event at stage *i*+1
  - *arrival* event : for stage *i* (*i* = 2 to *M*) - do not schedule the next *arrival* event

Example - Tandem Queue with Blocking



Tandem Queue with Blocking

- Two stages
- Finite waiting room at stage 2 (number of customers in system $\leq K$)
- Blocking
  - server 1 is blocked if a customer completing service at stage 1 finds no waiting room at stage 2

Tandem Queue with Blocking

- Subsystems and interaction
  - 2 subsystems -one for each stage
  - server 1 is blocked if a customer completing service at stage 1 finds no waiting room at stage 2
  - if server 1 is in the "blocked" state, it becomes "not blocked" when a departure occurs at stage 2
  - a departure from stage 1 becomes an arrival to stage 2

Simulation Program

- Use event routines for single server queue (infinite population model) for each of the 2 stages
- Modifications to implement tandem queue with blocking:
  - add state variable b
    - *b*= 1 if server 1 is blocked and 0 if server 1 is not blocked
  - Initialization : add the step
    - *b*= 0

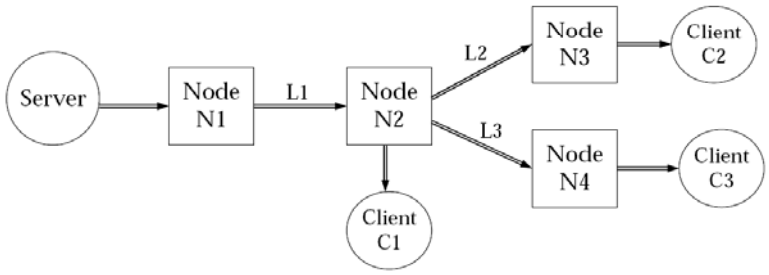
Simulation Program

- Modifications (cont.):
  - *start_service* event at stage 1 : add the step
    - schedule an *end_service* event at stage 1 instead of a *departure* event at stage 1
  - *end_service* event at stage 1 : add the step
    - if number in system at stage 2 < K then invoke routine for *departure* event at stage 1 else $b = 1$

Simulation Program

- Modifications (cont.):
  - departureevent at stage 1 : add the step
    - invoke routine for *arrival* event at stage 2
  - *arrival* event at stage 2 : do not schedule the next *arrival* event
  - *departure* event at stage 2 : add the step
    - if $b = 1$, then $b = 0$ and invoke routine for *departure* event at stage 1

Example - Computer Network

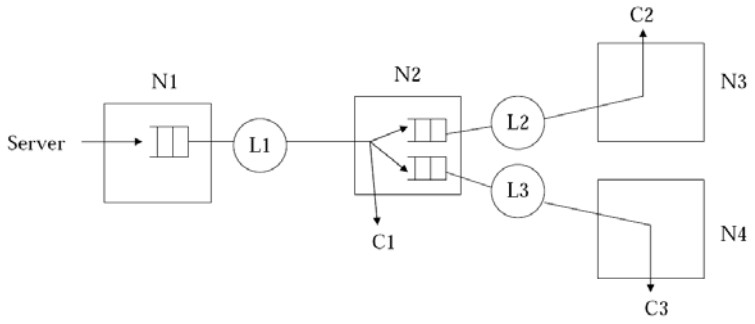


L1, L2, L3 - communication channels

Open Network Model

- Assumptions
  - infinite population model
  - multiple classes (or flows of packets to be transmitted)
  - three servers, each representing a communication channel
  - path oriented routing (each class follows a fixed path)
- Example
  - class F1 (server to C1) -routed along L1
  - class F2 (server to C2) -routed along L1, L2
  - class F3 (server to C3) -routed along L1, L3

Open Network Model



Different classes of customers (one class per flow)
Each class is routed along its own path

Subsystems and Interaction

- Three subsystems, one for each channel
- Interaction is defined by packet routing, e.g., for class F2, a departure from L1 becomes an arrival to L2, etc.
- Simulation program
  - use event routines for single server queue for each channel
  - initialization: schedule arrival events for class F1, F2 and F3 at server 1 (these are external arrivals), no arrival events scheduled for servers 2 and 3
  - arrival event: schedule next arrival for the same class, but at server 1 only

Departure Event

- Use routing table to determine next stage in packet routing (three cases)
  - invoke arrival event to next stage
  - packet leaves the system (delivered to client)
  - Error
- Routing table -based on paths taken by the various classes

Routing Table

- Subsystem 1 (L1)

Class	F1	Next stage	Client C1 (leaves system)
	F2		L2
	F3		L3

- Subsystem 2 (L2)

Class	F1	Next stage	error
	F2		Client C2 (leaves system)
	F3		error

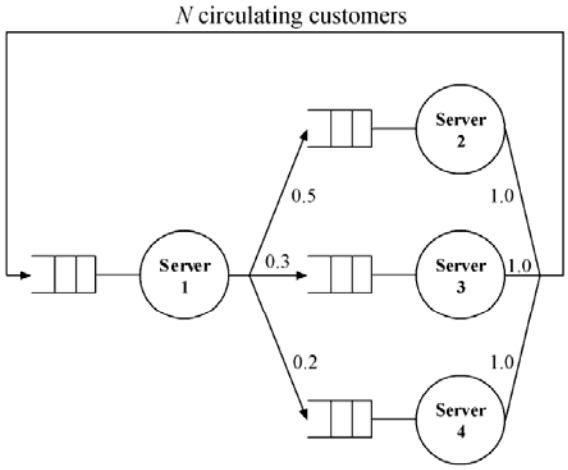
- Subsystem 3 (L3)

Class	F1	Next stage	error
	F2		error
	F3		Client C3 (leaves system)

Simulation

Example Models

Example 1 – Central Server Model



Subsystems and Interaction

- Four subsystems, one for each server
- Interaction is defined by transition probabilities
  - a customer departing from server 1 has
    - 50% probability of arriving at server 2
    - 30% probability of arriving at server 3
    - 20% probability of arriving at server 4
  - a customer departing from server 2, 3 or 4 has 100% probability of arriving at server 1

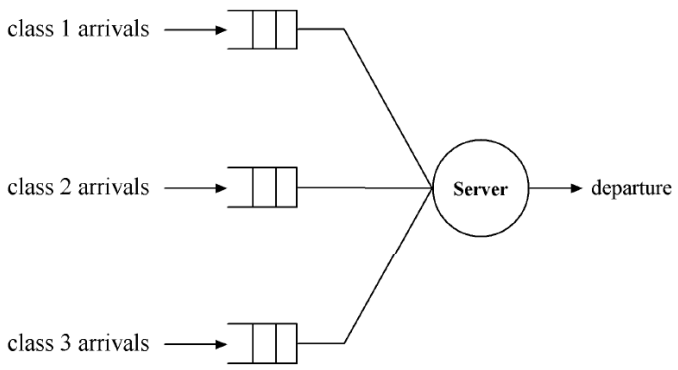
Simulation Program

- Single server queue model for each subsystem with modifications to model the interaction
- Initialization
  - *clock*= 0
  - for *i*= 1 to 4
    - Initialize *queue(i)* to empty
    - *status(i)* = idle
    - *n(i)* = 0
  - Initialize *event_list* to empty
  - enter *N* customers at end of queue(1)
  - invoke *start_service* event at server 1

Simulation Program

- *departure* event from server 1
  - determine *k*, the ID of the next server for the departing customer (*k* = 2 : 50%, *k* = 3 : 30%, *k* = 4 : 20%)
  - invoke *arrival* event to server *k*
  - *departure* event from server 2, 3 or 4
    - invoke *arrival* event to server 1
  - *arrival* event at server 1, 2, 3 or 4
    - do not schedule the next *arrival* event

Example 2 – Fair Queuing Model



Fair Queuing Model

- Suppose the server has a service rate of 1 unit of work per second
- At any point in time
  - the service rate seen by the customer at the head of each non-empty queue is equal to 1/*m* units of work per second, where *m* is the number of non-empty queues
- Interpretation
  - the server visits each non-empty queue in a cyclic manner, and provides a very small amount of service to the customer at the head of the queue

Fair Queuing Model

- Assumptions
  - for each class of customers, interarrival times are independent of each other and have identical probability distribution (similarly for service times)
  - system is empty at time zero
  - for each class of customers, arrival of first request occurs after the first interarrival time
  - simulation terminates at time = *term_sim*

Fair Queuing Model

- State variable
  - m - number of non-empty queues
  - $N(r)$ - number of class r customers in system, $r = 1, 2, ..., R$
    - in the context of the model, $N(r)$ is also the number of customers in queue r
- Event types
  - type 1: *arrival* (parameter: class membership)
  - type 2: *start_service* (parameter: class membership)
  - type 3: *departure* (parameter: class membership)
  - type 4: *end_simulation*

Simulation Program

- Initialization
  - $clock = 0$
  - $m = 0$
  - initialize *event_list* to empty
  - for $r = 1, 2, ..., R$
    $N(r) = 0$
    initialize $queue(r)$ = empty
    determine *inter_t*, an interarrival time of class r
    schedule an *arrival* (class r) event at $clock + inter\_t$
    end for

Simulation Program

- *arrival* event (class *r*)
  - $N(r) = N(r) + 1$
  - determine *inter_t*, an interarrival time of class *r*
  - schedule an *arrival* (class *r*) event at *clock* + *inter_t*
  - enter arriving customer to end of *queue*(*r*)
  - if $N(r) = 1$, then

```
m = m + 1
for each departure (class j) event in event_list
    remove that event from event_list
    retrieve the event time (let this be x)
    schedule a departure (class j) event at clock + (x - clock)*m /(m-1)
end for
invoke start_service (class r) event
end if
```


Simulation Program

- *start_service* event (class *r*)
  - determine *serv_req*, a service requirement for class *r*
  - schedule a *departure* event (class *r*) at *clock* + *serv_req* m*
- *end_simulation* event
  - exit the main loop

Simulation Program

- *departure* event (class *r*)
  - remove customer from head of *queue*(*r*)
  - $N(r) = N(r) - 1$
  - if $N(r) > 0$ then

    invoke *start_service* (class *r*) event

    else

    $m = m - 1$

    for each *departure* event (class *j*) in *event_list*

    remove event from *event_list*

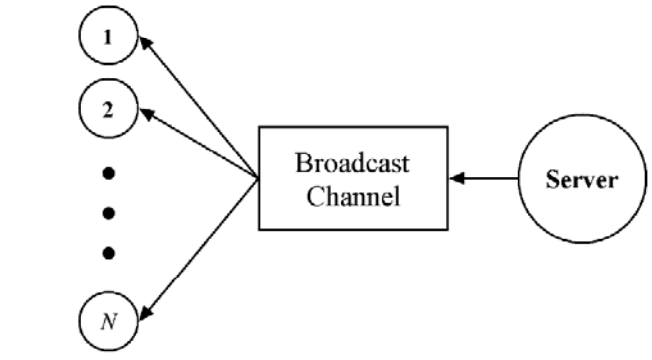
    retrieve the event time (let this be *x*)

    schedule a *departure* (class *j*) event at *clock* + (*x* - *clock*) * *m* / (*m*+1)

    end for

    end if

Example 3 – Broadcast Delivery



user workstations

Broadcast Delivery Model

- N users requesting information pages
- Pr [page j is requested] = p_j, j = 1, 2, …, M
- The server transmits pages continuously using a broadcast cycle (length of cycle = K)
  - $j(1), j(2), \ldots, j(K)$, where $j(k)$ is ID of k-th page in cycle
- Workstation operates as follows
  - user request is held at the workstation and not forwarded to the service computer
  - workstation examines the broadcast data until the requested page is transmitted; this page is captured and displayed
- "Push" model

Broadcast Delivery Model

- Assumptions
  - think times are independent of each other and have identical probability distribution (similarly for page transmission times)
  - for each of the N users, the first request is submitted after a think time
  - at time zero, the server starts transmitting the first page of the broadcast cycle
  - simulation terminates at time = *term_sim*

Broadcast Delivery Model

- State variables
  - k - page with ID = $j(k)$ is to be transmitted next
  - nt - number of users in the thinking state
- Statistical counters
  - nr - number of response times accumulated
  - sr - sum of accumulated response times
- Data structure
  - *event_list*
  - *request_list* - each entry contains *t_arrival* (time of arrival) and *page_req* (ID of requested page) of a user's request

Broadcast Delivery Model

- Event types
  - type 1: *arrival*
  - type 2: *departure* (parameter: *t_arrival*)
  - type 3: *start_transmit* (parameter: *k*)
  - type 4: *end_simulation*

Simulation Program

- Initialization
  - *clock* = 0
  - initialize *event_list* to empty
  - initialize *request_list* to empty
  - *nt* = *N*
  - *nr* = *sr* = 0
  - for *n* = 1, 2, ..., *N*
    determine *think_t*
    schedule an *arrival* event at *clock* + *think_t*
    end for
  - *k* = 1
  - schedule a *start_transmit* (*k*) event at *clock*
  - schedule an *end_simulation* event at *term_sim*

Simulation Program

- *arrival* event
  - *nt= nt-1*
  - determine *page_req*, the ID of page requested by user
  - enter entry (*clock*, *page_req*) to *request_list*

Simulation Program

- *start_transmit* (k) event
  - determine *transmit_t*, the page transmission time
  - for each entry in *request_list*
    if *page_req* = $j(k)$, then
    remove entry from *request_list*
    retrieve *t_arrival* for this entry
    schedule a *departure* (*t_arrival*) event at *clock* + *transmit_t*
    end if
    end for
  - $k = k + 1$
  - if $k > K$, then $k = 1$
  - schedule a *start_transmit* (k) event at *clock* + *transmit_t*

Simulation Program

- *departure* (*t_arrival*) event
  - *nt= nt + 1*
  - *nr= nr + 1*
  - *sr= sr + (clock - t_arrival)*
  - determine *think_t*
  - schedule an *arrival* event at *clock+ think_t*

Simulation Program

- *end_simulation* event
  - exit main loop
- Report generator
  - mean response time = *sr/nr*