Simulation Modeling

- Imitation of the operation of a real-world process or system over time
- Objective: to collect data as if a real system were being observed
- Data collected from the simulation are used to estimate the performance/dependability measures of the system

Discrete Event Simulation

- modeling of the system as it evolves over time by a representation in which the state variables change only at a countable number of points in time
- terminology:
  - simulation clock: a variable that gives the current value of the simulated time
  - event: an instantaneous occurrence which may change the state of the system

Simulation Terminology

– event list: a list (data structure) consisting of event records with each record containing the time of occurrence of a particular event, e.g., the arrival time, the departure time of a client
– timing routine: a subroutine which determines and removes the most imminent event record from the event list and advances the simulation clock to the time when the corresponding event is to occur
– event routine: a subroutine which updates the state of the system when a particular type of event occurs
    - **one event routine for each type of event**

Event Scheduling

- Determine the number of event types in the system, e.g., 1: arrival, 2: request for service, 3: service completion, 4: timer, etc.
- Place one or more initial event records in the event list, each containing
  - event time, event type, customer class, etc.
- Determine the most imminent event in the event list (by the timing routine) in a loop until a specified stopping rule is satisfied
  - update the simulation clock when an event record is removed from the event list

Event Scheduling (cont.)

  - Pass the control to the event routine corresponding to the event type
  - Update the state of the system
  - Gather the statistics if necessary
- Report the simulation results when the simulation is completed
  - For example
    - **the average response time per client**
    - **the loss probability of calls**
    - **the system throughput**
    - **the average number of clients served over a time period**

Simulation using smpl

- In the smpl view of systems, there are three types of entities:
  - resources: facilities
    - smpl provides functions to define, request, release and preempt (queueing) facilities
  - tokens: active entities of the systems, e.g., tasks, users (indistinguishable or distinguishable)
  - events: a change of state of any system entity is an event
    - smpl provides functions for scheduling and for selecting events in the order of event occurrence time

Structure of An smpl Program

Initialization routine;

timing control routine to select the most imminent event from the event list (event clock is updated implicitly)

```
{
  event type 1: event routine for event type 1;
  event type 2: event routine for event type 2;
    .
    .
  event type n: event routine for event type n;
}
```

statistics reporting routine;

Initialization Routine

```
smpl(m, s)
int m=0; /* always 0 */
char *s;
```

smpl provides seeds for 15 streams for generating random numbers. To collect a set of 15 sample values of a particular performance measure, one can invoke smpl() 15 times:

```
loop: repeat 15 times
 {
  smpl(0, “hw1”);
 }
```

One can also use stream(1), stream(2), etc. to specify the stream number to be used in a simulation run

Facility Definition and Control

fd = facility(s, n)
char *s;
int n; /* # of servers */
=> define a queueing server with “n” servers;
smpl automatically manages enqueueing/dequeueing activities

r = request(fd_id, token_id, pri)
int fd_id; int token_id; int pri;
=> request a server of facility “fd_id” be reserved for the token designated by “token_id” with priority “pri” (higher is better)

r=0: facility is reserved
r=1: facility is busy and the request is blocked in the queue ordered on priority

Facility Definition and Control

r = preempt(fd_id, tkn_id, pri)
int fd_id, tkn_id, pri;
=> same as request() except that it will preempt the server if it is busy serving a task with priority < "pri"
=> the event record corresponding to the preempted token (for the service completion event) is removed from the event list and a queue entry with the residual time is created
r=0: facility is reserved
r=1: facility is busy and the request is blocked in the queue ordered on priority

create an event of the same type and put it in the event list

release(fd_id, tkn_id)
int fd_id; int tkn_id;
=> release the facility and if the queue is not empty, reschedule an event with the event occurrence time at NOW for a blocked task, and reschedule an event with the event occurrence time at NOW+ the residual time for a preempted task.

Scheduling Events

schedule(event_id, te, tkn_id)
int event_id;
real te; /* time interval relative to the current time */
int tkn_id;
=> schedule the event with id "event_id" to occur at NOW+te
=> this essentially inserts an event record with the event occurrence time NOW+te into the event list
=> part of the information in the event record is event_id, tkn_id and the event occurrence time NOW+te

Example: schedule(2, 0.0, token_id)
=> schedule event type #2 associated with token id "token_id" to occur NOW

Timing Routine

```
cause(event_id, tkn_id)
int *event_id;
int *tkn_id;
```

=> remove the most imminent event from the event list and automatically advance the simulation clock to the event occurrence time

=> return the event number (type) and token id to the caller

Typically in the smpl program, we use a select statement on the event_id returned, so as to transfer the control to the appropriate event routine.

Canceling Events

cancel(event_id)
int event_id;
=> search the event list and remove the first event with the event number = event_id

Get Current Simulation Time

t = real time()
=> return the current simulation clock value
=> real is a predefined type; it is the same as double in C

Status Functions

n= int inq(fd);
=> returning # of tokens currently in queue (not including the ones in service)

r = int status(fd)
=> r=0: facility is free; r=1: facility is busy

u = real U(fd)
=> mean # of tokens in service

n = real Lq(fd)
=> mean # of tokens in queue excluding the ones in service

b = real B(fd)
=> mean busy period = accumulated busy time/release counts

Random Variate Generation (rand.c)

r = real drand48(); /* available on UNIX machines */
=> return r in the range of (0,1)

r = real expntl(x)
double x;
=> return an exponentially distributed sample value with mean x

r = real uniform(a,b)
double a,b; => return a real number r in the range of (a,b)

k = int random(i,j)
int i, j; => return an integer k in the range of (i,j)

r = real normal(x,s)
=> return a normally distributed sample value with mean x and standard deviation s

Traces and Debugging

trace(n)
int n;
=> generate trace messages when a facility is defined, requested, or released, or whenever an event is scheduled or caused

n=0: trace is off
n=1: free-running, i.e., trace messages are generated continuously
n=2: screen by screen running (press any key to resume tracing)
n=3: message by message running (press any key to resume)

M/M/1 smpl program

```
#include “smpl.h”
main()
{
real Ta=200, Ts=100,  te=200000;
int customer=1, event, server;
smpl(0, “M/M/1 Queue”);
server = facility(“server”,1);
schedule(1, 0.0, customer);
while (time()< te)
 {
   cause(&event, &customer);
   switch(event)
    {
     case 1: /* arrival */
       schedule(2,0.0, customer);
       schedule(1, expntl(Ta), customer);
       break;
     case 2: /* request server */
         if (request(server, customer,0)==0)
          schedule(3, expntl(Ts), customer);
         break;
     case 3: /* completion */
          release(server, customer);
          break;

    }
 }
report();
}
```

Confidence Interval and Level

- Suppose we collect N sample values Y_1, Y_2, …, Y_N from N simulation runs
- sample mean $Y = (Y_1 + Y_2 + \ldots + Y_N)/N$
- true mean is μ
- Define $1-\alpha$ as the probability that the absolute value of the difference between Y and μ is equal to or less than H } Confidence interval half-width
  - that is, $\text{prob}[Y-H <= \mu <= Y+H] = 1-\alpha$

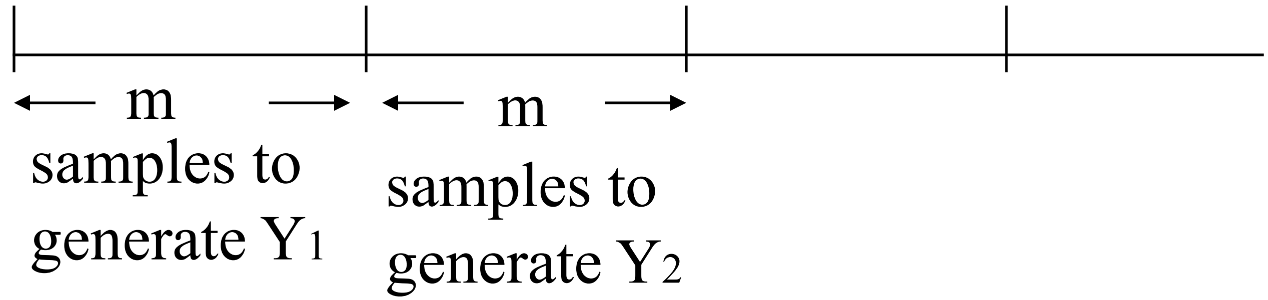
Confidence interval (Y-H to Y+H)

Confidence level ($1-\alpha$)

Confidence Interval and Level (cont.)

- When $Y_1, Y_2, \ldots, Y_N$ are independent random variables from a normal distribution with the mean μ, H is defined by $H = t_{\alpha/2;N-1} * \sigma/\mathrm{sqrt}(N)$ where t is the student's t distribution and σ^2 is the sample variance given by $\sigma^2 = \Sigma_i (Y_i - Y)^2 /(N-1)$ (and thus σ is the standard deviation).

Batch Mean Analysis by smpl



- Use a batch size m around 2000 observations to collect a sample value Y_i to justify the normal distribution assumption (by central limit theorem).
- Delete d = 0.1 m initial observations
- Collect k = 10 batches and compute the confidence interval half-width H
- If the desired accuracy has not been reached, collect another batch and compute H again. Repeat as necessary.

BMA: stat.c and bmeans.c

- Based on 95% confidence level ($\alpha = 0.05$) with 10% confidence accuracy (H/Y = 10%)
- The following three routines are provided:
- init_bm(d, m): d is number of initial observations to be discarded and m is the number of observations to collect one sample Y_i
- obs(y): y is the observation value generated out of a simulation run
  - if the returning value is 1, it means that the required confidence level and accuracy have been reached; otherwise, need to continue calling this function obs(y)
- civals(Y, H, k): Y, H and k are passed in by reference. This function returns the final result.

M/M/1 smpl program with BMA

```
#include “smpl.h”
#define TOKENS 1000
#define TRUE 1
#define FALSE 0

main()
{
real Ta=200.0,Ts=100.0,mean,hw;
int tk_id=0,customer=0,event,server,nb;
real ts[TOKENS]; /* start time stamp */
int cont=TRUE;
smpl(0,"M/M/1 Queue with BMA");
init_bm(200,2000); /* d=200; m=2000 */
server=facility("server",1);
schedule(1,0.0,tk_id);
while (cont)
    {
     cause(&event,&customer);
     switch(event)
       {
case 1:  /* arrival */
          ts[customer] = time();
          schedule(2,0.0, customer);
          if (++tk_id >= TOKENS) tk_id=0;
          schedule(1,expntl(Ta),tk_id);
         break;
case 2:  /* request server */
          if (request(server, customer,0)==0)
            schedule(3,expntl(Ts),customer);
          break;
 case 3:  /* release server */
          release(server, customer);
          if (obs(time()-ts[customer]) == 1)
             cont = FALSE;
          break;
        }
      } /* end while */
civals(&mean, &hw, &nb);
printf(”Y= %f; H= %f after %d batches\n”,
          mean, hw, nb);
}
```

Bmeans.c

```
#include "smpl.h"
#include "stat.c"

static int d,k,m,n;
static real smy,smY,smY2,Y, h;

init_bm(m0,mb)
  int m0,mb;
    { /* set deletion amount & batch size */
      d=m0; m=mb; smy=smY=smY2=0.0;
      k=n=0;
    }
obs(y)
  real y;
    {
      int r=0; real var;
      if (d) then {d--; return(r);}
      smy+=y; n++;
      if (n==m) then
      { /* batch complete:  update sums & counts */
       smy/=n; smY+=smy; smY2+=smy*smy; k++;
       printf("batch %2d mean = %.3f",k,smy);
       smy=0.0; n=0; /* reset batch variables */
       if (k>=10) then
       { /* compute grand mean & half width */
            Y=smY/k; var=(smY2-k*Y*Y)/(k-1);
            h=T(0.025,k-1)*sqrt(var/k);
            printf(", rel. HW = %.3f",h/Y);
            if (h/Y<=0.1) then r=1;
       }
         printf("\n");
        }
     return(r);
    }

civals(mean,hw,nb)
  real *mean,*hw; int *nb;
    { /* return batch means analysis results */
     *mean=Y; *hw=h; *nb=k;
    }
```