# CIS 520, *Operating Systems Concepts*

Lecture 3

*Simulation of Asynchronous Processes*

(Queuing Systems)

# Agenda

- Two paradigms in the design of operating systems
- Queuing model
- Simulation mechanisms
  - Random number generation
  - Event generation
  - Event processing
  - Statistics gathering
- Homework: The First Programming Assignment

# Two Paradigms
## *(after A. Tannenbaum)*

```
main( )
{
        int ... ;

        init( );
        do_something( );
        read(...);
        do_something_else( );
        write(...);
        keep_going( );
        exit(0);
}
```

**I: Algorithmic code**

```
main()
{
 event_t event;
 while (get_event(event))
   {
     switch (event.type)
       {
                    case 1: ...;
                    case 2: ...;
                    case 3: ...;
       }
   }
}
```

**II: Event-driven code**

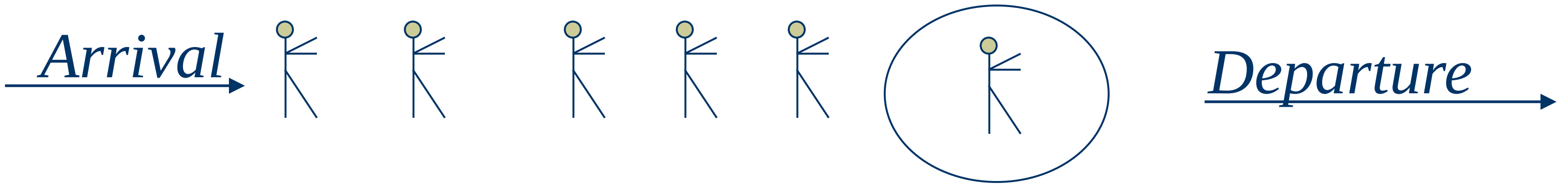
# Even better!

```
#define Total_Event_Number ... ; /* some constant */

void Event_0();
void Event_1();
...

*p [Total_Event_Number] Event_Handler ();

Event_Handler([0] = Event_0;
Event_Handler([1] = Event_1;
...
main()
{
 event_t event;
 while TRUE
     Event_Handler[get(event)]
 }
```

# The Model

The Customer Queue

Mean rate: μ customers/sec




Mean rate: λ customers/sec

The Server

Steady State: $\lambda < \mu$

(takes certain time to process a customer, so the queue is formed)

# What does *mean* mean?

Let the probability density of a random variable $\xi$ be $p_\xi(x)$ defined on an interval [a, b]

Then the *mean* E[$\xi$] is defined as

$$E[\xi] = \int_a^b x p_\xi(x) dx$$

For a discrete variable $\xi$ with $p_\xi(k) = P(\xi=k)$,

$$E[\xi] = \sum_{i=0}^{\infty} i p_\xi(i)$$

# The *Poisson* Distribution

The probability of exactly *k* arrivals within the interval *[0, t]* is independent from the previous arrival history:

$$\boxed{P\{t,k\} = e^{-\lambda t}\frac{(\lambda t)^k}{k!}}$$

# Inter-arrival Time

- For the *Poisson* distribution, the inter-arrival probability density is distributed exponentially, too:

$$P[\textit{inter-arrival time} < t] = 1 - e^{-\lambda t}$$

# Simulations

- Analytical approach does not work very well for all classes of queuing disciplines
- The low price of computing permits us to use computer simulations of very complex queuing systems (it is equivalent to integrating complex difference-differential equations) on a computer
- Simulations are widely used in modeling (and thus predicting) computer, communications, financial, and biological systems
- From now on, we will use simulations systematically in our homework

# First Comes First: Random Number Generation

- Well, one could generate them using Geiger counter, for example, or—better yet—using the data from past experiences
- But very often—and this is what we will do– what is used is *pseudo-random* number generators
- You could download C++ packages from the WWW (e.g., http://www.agner.org/random/, or http://mathworld.wolfram.com/ ) , or check with your local systems administrator for the packages already on your machine. (A good one is available on the class site.)
- If you are interested in the subject, the best place to start (and finish!) is *D. Knuth, "The Art of Computer Programming, " Volume II*

# Pseudo-Random Number Generation (The *Linear Congruential Method*)

- We start with four natural “magic numbers”:
  - $m$, the *modulus*
  - $a$, the *multiplier* ($a < m$, *a is co-prime with m*)
  - $c$, the *increment* ($c < m$)
  - $X_0$, the *starting value* (or *seed*) ($X_0 < m$)

The desired sequence (which should have a long *period*) is

$$X_{n+1} = (aX_n + c) \bmod m,\ n > 0$$

# Pseudo-Random Number Generation (The *Linear Congruential Method*)

- Again, you can find (and you should, if you have time) much better magic numbers, but just for the purpose of the exercise here is a choice of magic numbers (and they don't require the use of long integers):

$$a = 25173, c = 13849, m = 65536.$$

# Uniformly Distributed Pseudo-Random Numbers

We can obtain a *uniformly-distributed* sequence $Y_n$ in the interval $[0, 1)$ by scaling the sequence

$$X_{n+1} = (aX_n + c) \bmod m:$$

$$Y_{n+1} = X_{n+1} / m$$

# Exponentially Distributed Pseudo-Random Numbers

Finally, we can obtain an *exponentially-distributed* sequence $Z_n$ in the interval, with the mean arrival rate $\lambda$ (or mean inter-arival rate $1/\lambda$):

$X_{n+1} = (aX_n + c) \bmod m$:

$$Z_{n+1} = -\frac{1}{\lambda}\ln\left(\frac{X_{n+1}+1}{m}\right)$$

# To Make a Long Story Short

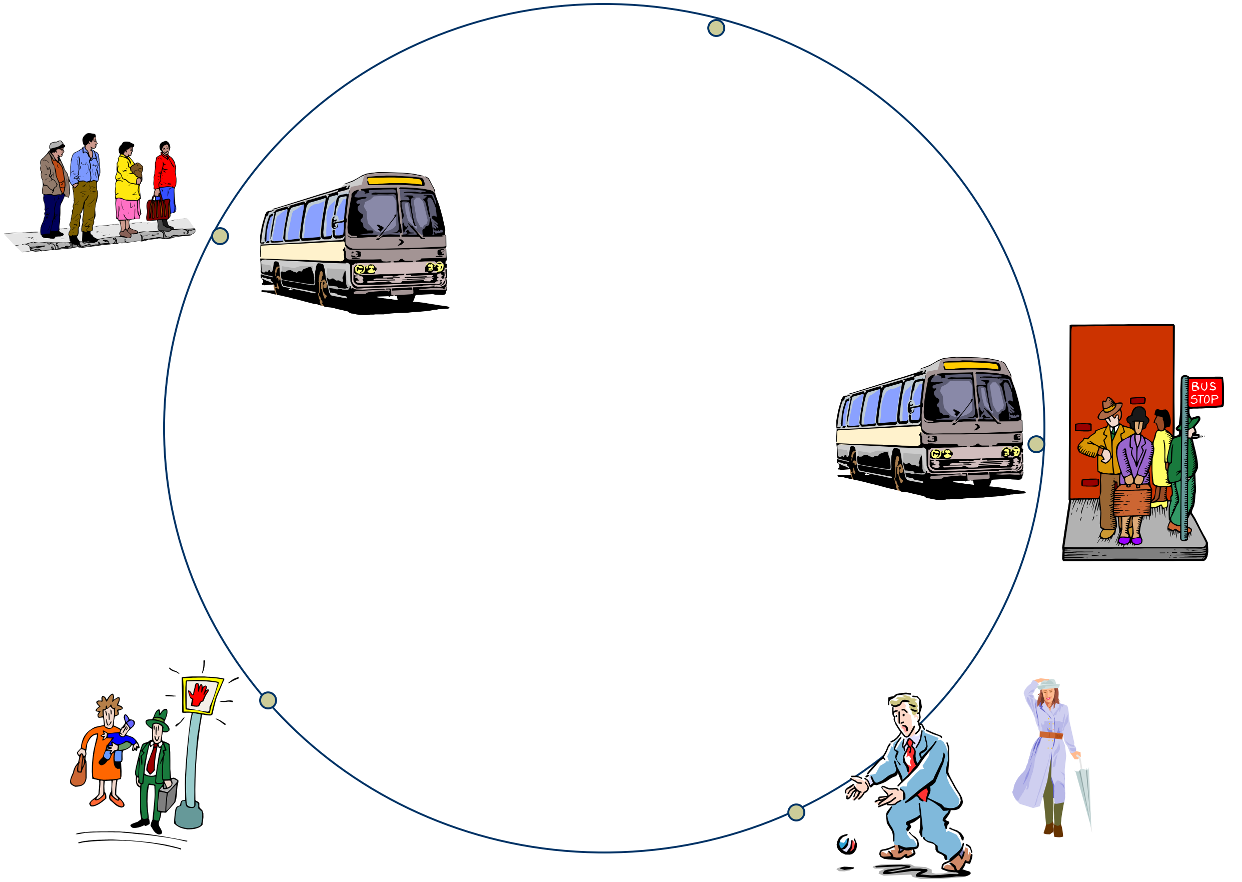
- An example:

```
int seed = 1000; /* or, better yet, read it */
float random ()
   float x;
    {
        x = - log ((seed + 1) / 65536);
        seed = (25173 * seed + 13849) % 65536;
              return x;
     }
```

# Writing Simulations

- We *create* a data structure (a balanced tree, or simply a doubly-linked ring), which keeps the *events* sorted by time; this is initialized by a dummy event scheduled for time *0*
- We *generate* events based on the system we are simulating; as the result, a generated event enters the structure in an appropriate place
- We execute the program by *repeating* the following steps until simulated current time exceeds the defined time limit
  1. Take the next event from the event structure and update the current time
  2. Process the event; (that will likely generate other events)

# We Will Simulate a Bus Service



# Three Kinds of Events

1. ***person:*** A person arrives in the queue at a bus stop

   Action: After a random (exponentially-distributed inter-arrival) time, another person is scheduled to arrive in the queue
2. ***arrival:*** A bus arrives at a bus stop

   Action: If there is no one in the queue, the bus proceeds to the next stop, and the event of its arrival there is generated; otherwise, the event to be generated (at present time!) is the first person in the queue boarding the bus;
3. ***boarder:*** A person boards the bus

   Action: The length of the queue diminishes by 1; If the queue is now empty, the bus proceeds to the next stop, otherwise the next passenger boards the bus

# Assumptions

- It takes everyone the same time to enter the bus
- As many people (on average) exit the bus as enter it, and the time to exit the bus is negligible. *Consequence: we do not consider the* ***exit*** *event in our model*
- The stops are equally spaced in a circle
- The buses may not pass one another

# The Event Record

This is an entry in the event structure. It contains

1. The time of the event
2. The type of the event
3. The rest of the information, which is event-dependent: the name (number) of the stop, the number of the bus, etc.

# The Main Program

```
Initialization();
do
  {
    Get the next event;
    clock = event time;
    switch event_kind
      person:
        {
          update the queue[stop_number];
          generate_event (person, stop_number, bus_number); /* No polymorphism in "C"! */
        }
      arrival: {...} /* board the bus
      boarder: {...}
} while clock < = stop_time
```

# Initialization

- Read the *number of buses*, the *number of stops*, the *driving time between stops*, the *boarding time*, the *stop time*, and the *mean arrival rate* from an **initialization file**.
  - For the beginning, assume there are *15* bus stops, *5* buses, the (uniform) time to drive between two contiguous stops is *5* min., mean arrival rate at each stop is *2* persons/min, and boarding time is *3* seconds
- Start with the buses distributed uniformly along the route (by generating appropriate *arrival* events) and generating one *person* event for each stop.

# Event Generation

- When the bus *arrival* event occurs, if the queue is empty, generate the *arrival* at the next bus stop at *clock* + *drive_time*. If the queue is not empty, generate the *boarder* event (at *clock*)
- When the *boarder* event occurs, if the queue is empty (i.e., the last person boarded), generate the arrival at the next bus stop at *clock* + *drive_time*. If the queue is not empty, generate the *boarder* event (at *clock* + *boarding_time*)
- At the *person* event, generate the next *person* event at the same stop at *clock* + *(mean_inter-arrival_rate) * random (*exponential*)*

***Note**:* Keep time in *float* or *double* (a better way of doing things). The *mean_inter-arrival_rate = 1/ mean_arrival_rate*

# The Goal

- The purpose of the simulation is to observe the behavior of the system, and answer the following questions:
  - Do the distances between two consecutive buses keep uniform? If not, what should be done to ensure they are uniform?
  - What is the average size of a waiting queue at each stop (and what are its maximum and minimum)?
  - Plot the positions of buses as a function of time (you will need to generate periodic snapshots of the system for that)