ECS 145: Term Project

Due on March 21, 2018 at 3:10pm

Professor Norm Matoff

Justin Weich, Alex Whelan

What is DES?

DES

DES stands for "discrete event simulation". DES is a methodology for simulating a system where the system is viewed as a set of discrete events over a period of time. The events are viewed as discrete points in time and the simulation time is only progressed through the occurrence of events. The attributes and time of the system are non-continuous. This allows the simulation to run much faster than a model that has continuous time. It also has the benefit of being easier to create a model for; representing events that happen is easier than creating a function that represents a continuous model. DES frameworks will almost always have a few structures in common.

Simulation Time

There must be a system time saved in the memory. The system time is a value that represents the time after the start of the system. Typically, the system time will start at 0. When events are added, they are added in respect to the current system time.

Event List

The event list is a list that contains the events scheduled to occur. The next event is found by taking the minimum value for scheduled time. The event is then removed from the list, and the simulation is stepped forward accordingly.

State

The state is the variable or variables that are being studied with the simulation. The state is how the information is tracked over the course of the simulation.

DES Paradigms

There are a few different ways to model the interaction of the events and the system. The class focussed on the Event Oriented DES and Process-Oriented DES.

Event Oriented DES

Event-Oriented DES focuses on the creation and handling of discrete events. There are often multiple different event types for a single model. Each event type has it's own handler than runs a specific section of code. The DES.R package is a library for implementing a discrete event simulator.

Within the DES.R package, there is an object that keeps track of the different DES structures like the wait time and event list. In the user-written simulation code, they will typically schedule some number of events. These events can represent things like a machine breaking down, or an airplane leaving a runway. The user will then 'start' the simulation.

This will start off by finding the minimum time event from the event list. The system time will be adjusted according to the event time. It then activates the function that is associated with the event (the event handler). The event handler will resolve the event, and possibly schedule future events. This process will repeat until the event list is empty, or until the maximum system time has been reached.

The DES.R library also include the ability to pre-populate a timeline and load that into the simulation. This means that there can be a set of events created in a batch at the beginning that don't have to be generated by other other events.

Process Oriented DES

The process oriented DES paradigm takes the main concept of the DES and applies the ideas of processes to it. This paradigm is useful because it is a lot easier to frame the system in this view.

Essentially, each of the items in the system that are being studied are mapped to a process (which is not necessarily a separate OS-level process). There is typically a single process that manages the ones being studied. Like the Event-Oriented approach, events are added to the event queue and the main function (the manager process) finds the smallest event time in the event queue.

However, unlike the event-oriented DES, the manager process stops its own execution, and resumes the process that generated the event. The process is able to add events, but when it is done executing the code for that event, it pauses its own execution and the manager process resumes where it left off.

In the SymPy library, this is done with the use of generators. The processes are in functions that use "yield", which allows the program to resume execution at the point of the yield the next time the function is called.

RPosim

Generators

R's lack of the ability to resume execution of code in a specified location was one of the major hurdles of implementing a Process-Oriented DES. We were able to get around this issue by implementing a multi-threaded approach. Because multi-threading is also not a native part of R, this introduced it's own issues.

Starting the Threads

The first first issue was starting the treads. In order to initialize and start a thread, we decided that users of this library would have to have the code for their process in a separate R file that took arguments like the thread-id and the specific process to start up. This allowed the manager thread to use background system calls to start the program. We also decided to cause each thread to show up in a terminal window (because the information flow looked really cool).

Sharing Information

Because the threads are opened as separate processes, there is initially no way to communicate between the threads. We implemented a bigmemory matrix to allow this. This isn't a very elegant solution but it worked for our purposes. This also allowed for the threads to "relinquish" control from themselves and allow another thread be active. A busy-wait while loop is implemented to allow for this behavior. Each thread gets a slot in the bigmemory matrix that represents where or not it should be active. The application threads also use this matrix to communicate information (such as the next event) with the manager thread.
The matrix is limited to just containing numbers. Extra helper functions must be added to encode and decode non-integer data transfer. We mapped integers to certain strings to represent the different states.

The Randomness Issue

With the structure of our library, we ran into one issue that was not able to be simply fixed. Because each thread is its own process, when seeded, the random function used to generate events would generate the same numbers in the same order. That number would get added to the base time. This lead to a behavior where the processes would flip-flop when the larger outliers were generated. This really started to break down the results of the simulation because only one thread would be generating numbers at a time. When the end of the simulation came, the number of events would never be even. We were able to get around this by seeding each thread with its thread ID. However, we did not implement a way to allow the user to set a seed for their simulation.
However, we did think of a way to combat this issue. The proposed solution is to seed only the manager thread, and then have that generate a seed that is passed as an argument when it starts up the other threads. While the user isn't able to explicitly generate a seed for the threads, they are able to seed the simulation while still maintaining predictability when seeded with the same value.

Examples

Along with this report, we have the three implementations of the machine repair simulation. The first is just uses the 'hold' and is in App1.R. The second implements "request" and "release" and is located in App2.R. The third starts with "passivate" and "reactivate" and is contained within App3.R. The way to start the simulation is to run the OS.R file. To test the different models, the file name just needs to be changed on line 8 of OS.R.

General Flow

To use the library, the user will write a file for each application thread type. There will also be a file for the manager thread, called the OS. This will be the file that starts up the threads.

For the initialization of the main controller-thread object, we have the following function:

```
# Application Cols will be event specific parameters
# Application Parameters will be Simulation specific variables to observe
initOS <- function(max_time, num_threads, appcols=NULL, app_parameters=NULL, resource=NULL)
```

Besides specifiyign the maximum simulation time and number of threads, the two most important prameters are the app_parameters and resources. The applications parameters are variables that are going to be monitored/observed during the simulation and are completely user-defined. Those parameters are user-defined and each has some number of columns that are set for the bigmemory matrix. The resources are integers that are used as limiters for the request and release functions.

The bigmemory matrix that we use is defined as the "thread stack".

```
OStack$thread_stack <- bigmemory::big.matrix(nrow=OStack$num_threads, ncol = 3 +
    length(app_parameters) + length(resource), init = 1)
```

As you can see, the size of the matrix is defined partially by the user-defined variables. The first row of this matrix belongs to the OS thread, where it keeps any meta-data that needs to be shared with the other threads. Each thread gets its own row of the matrix.

Activate is used to spawn the new threads. The threads will show up in new terminal windows as a visual aide.

```
activate <- function(OStack, fname=NULL){
    id <- getID(OStack)
    system2(command = "xterm", args = sprintf("-e Rscript \"%s\" \"%s\" &",fname, id ))}
```

The main loop of the OS thread takes place in simulate.

The first thing the simulate function does is yield for all of the spawned threads to be ready. This is to make sure that if one of the processes takes a long time to become ready, the simulation doesn't start without it.

```
for (i in 2:OStack$num_threads){
    yield(wait = OStack$now, thread_id = 1, res_id= i, ts=OStack$thread_stack)}
```

Next, it enters the main functionaliy.

```
for (i in 2:OStack$num_threads){
     # Thread Has not Run not Run since last iteration
     if (OStack$thread_stack[i, "Time"] != 0) {
       if(OStack$thread_stack[i, "Time"] == -1){OStack$thread_stack[i, "Time"] <- 0}
       OStack$event_list[i, 1] <- OStack$thread_stack[i, 2] + OStack$now
       OStack$event_list[i, 2] <- OStack$thread_stack[i, 1]
       # Reset Thread's Time Parameter
       OStack$thread_stack[i, "Time"] <- 0}
     }
     events <- OStack$event_list[,'Event Time']
     if(length(events[!is.na(events)]) != 0){
     #event has occured
     #delete the event
     this_event_thread <- as.numeric(which.min(OStack$event_list[,'Event Time']))
     this_event_time <- OStack$event_list[this_event_thread,1]
     this_event_operation <- OStack$thread_stack[this_event_thread, "Operation"]
     OStack$event_list[this_event_thread,] <- NA
     #increment time
     OStack$now <- this_event_time
     OStack$thread_stack[1, 2] <- OStack$now}
     yield(wait = OStack$now, thread_id = 1, res_id= this_event_thread, ts=OStack$thread_stack)}
```

This pops the minimum time event from the event list, updates the system time. It then wakes the thread that created the minimum event with the yield command.

The yield command is the most important function to the process oriented approach.

```
yield <- function(func=NULL,resource=NULL, wait=0, thread_id, res_id, ts)
```

Depending on the paramters, the yield function is able to pause the thread for a set ammount of OS time, and update and read the user-defined limiting resources. The function takes in the "operation" ("hold", "request", etc) and performs the action on it. It then either switches to another thread, or performs one of the request or release fucntions and returns. The thread switching is done with the following command:

```
    ts[thread_id, "Active"] <- 0
    ts[res_id, "Active"] <- 1
    while(ts[thread_id, "Active"] == 0]){}
    return()
```

The res_id thread is in a busy-while loop waiting for the value to equal "1". The execution for the thread pauses at this point until another thread changes the value in the "Active" column.

We implemented the following callbacks (named funcs in our library): hold, release, request, passivate

Hold simulates the passage of time, which is recorded into the application parameters for the final analysis. It is a time-out event.

Request and **Release** are used in tandem. Request acts like a semaphore with a limited resource that prevents the program from progressing while there are no resources availble. For example, if there are only two doctors at a hospital, they would be represented as a resource. If two patients show up, they are able to be serviced immediately and "request" the doctor. They would be able to immediately go into the state of being treated, with the treatment time added to the event list. The resources are only released when the treatment event has passed. If a third patient shows up before either doctor is available, they just have to wait untill one of the other patients is done being treated before they can generate a treatment time for the event list.
Passivate is used in tandem with the **Reactivate** function. When passivate is used, it locks the thread from working untill it has been reactivated with the **reactivate** function.

Data Structures

The bigmemory matrix has the following columns. app_parameters and resources can have more than one column.

Active	Time	Operation	app_parameters	resources

The event_list is also a matrix.

Time	Thread ID

TCP/IP Approach

Instead of loading and saving the values from the matrix, the values could just be sent over TCP/IP protocol. In order to implement this, the value would have to be sent with some amount of meta-data such as the thread it is coming from and what the data is supposed to represent.

This method would have the benefit of being able to transfer more than just integers. Strings could be transmitted. Beyond that, lists could be transmitted with much more ease than the shared memory approach. The thread switching would feel a lot more intuitive than the current matrix method. While a thread is waiting, it would just constantly check the TCP stream, and stay like this until a message is received. When a thread needs to relinquish control, it sends the wake-up message and any other information it needs to transfer and then goes into the wait-for-message phase.

This method would mean that a bunch of TCP/IP specific functions would need to be implemented to the program, though that wouldn't be any different from the methods that have to handle the matrix.

simmer

Simmer makes use of an operator %<% after every line in the process object code to turn it into a list of the code that can be indexed. The object for the process also contains a value that stores which line is being executed. When the program yields, it just saves the point in the function where it was.

```
library(simmer)
setwd('.')
set.seed(12345)


# Simulation Parameters
NUM_MACHINES <- 2
MAX_SIM_TIME <- 10000.0
TOTAL_UP_TIME <- 0

# Initializes Class and Parameters for Application
Machine <- list()
Machine$up_rate <- 1.0/1.0
Machine$rep_rate <- 1.0/0.5
Machine$total_up_time <- 0.0


env <- simmer('Mach_Rep')

# Write your Run Process Method Here
Run <- function(machine){

    trajectory() %>%
      set_attribute("total_up_time", 0) %>%
      # Set Current Time
      set_attribute("start_time", function() now(env)) %>%

      seize(machine, 1) %>%

      log_("Simulate Up Time\n") %>%
      timeout(function() rexp(1, Machine$up_rate)) %>%
      # Update Total Up Time

      set_attribute('up_time', function() now(env) - get_attribute(env, "start_time")) %>%
      set_attribute('total_up_time', function() get_attribute(env, "up_time"), mod="+") %>%

      log_("Simulate Repair Time\n") %>%
      timeout(function() rexp(1, Machine$rep_rate)) %>%
      release(machine, 1) %>%
      rollback(9,Inf) # Simulate Cycle of Uptime and Repair

}



machines <- paste("machine", 1:NUM_MACHINES)

for (i in machines) {
  env %>%
  add_resource(i, 1, Inf) %>%
  add_generator(i, Run(i), at(0), mon = 2)
}

```

```
54
55 env %>% run(MAX_SIM_TIME)
56
57
58 x <- tail(get_mon_attributes(env))
59
60 TotalUpTime <- sum(x[x[,3] == 'total_up_time',4])
61
62 cat("the percentage of up time was :")
63 cat(TotalUpTime/(2*MAX_SIM_TIME))
```

Who did what?

The code was written 50/50 by both Justin and Alex. Justin primarily did the high-level programming and Alex worked with the low-level programming. Alex wrote the simmer implementation of the first machine repair example. Justin wrote the report.