RICE UNIVERSITY

Electrical & Computer Engineering Department

YACSIM

Reference Manual

Version 2.1

March 1993

J. Robert Jump

ECE Dept., Rice University
P.O. Box 1892
Houston, TX 77251-1892
email: jrj@rice.edu
Phone: (713) 527-8101 ext. 3576

This manual describes a simulator that has not been thoroughly tested and may contain bugs. Suggestions, criticisms, questions, or reports of any problems, errors, or bugs with the manual or the simulator are welcome and encouraged. Please send them to J. R. Jump at the above address.

Credits

YACSIM is derived from XSIM, a C++ discrete event simulator written by J. R. Jump. XSIM was influenced by, and is a significant enhancement of CSIM, a discrete event simulator developed at Rice University by Richard Covington. J. B. Sinclair wrote the YACSIM random number generator code and the machine dependent code for context switching on the IBM RISC/SYSTEM 6000 and the SPARC architectures. J. R. Jump wrote the other YACSIM code.

TABLE OF CONTENTS

1. INTRODUCTION

YACSIM is a discrete-event simulation language based on the C programming language. It is implemented as a collection of data structures and a library of C subroutines that can be linked with any program written in C. Extending an existing language by adding discrete-event simulation routines is a common approach. The main advantages of this over designing a completely new language are that the user can write most of the code of a simulation in a well-known and widely-used language, and a new compiler is not required. A user who knows the base language only needs to learn how to use the simulation library routines in order to write simulation programs.

There are at least two other simulation languages based on the C programming language, both called CSIM. One was developed at Rice University as part of a parallel processing testbed, and the other was developed at MCC in Austin, TX. YACSIM is Yet Another CSIM that has features in common with both of these. It is more closely related to the Rice CSIM and was developed to replace that language for the parallel processing performance work at Rice.

This manual describes the simulation routines in the YACSIM library. It does not give details of their implementation except in a few cases where such knowledge would be helpful for improving the efficiency or accuracy of a simulation. The presentation assumes that the reader is familiar with the C programming language. It also assumes a familiarity with discrete-event simulation including both event-driven and process-oriented techniques. The manual gives the minimal amount of information needed to use the simulators. It is not a tutorial, since it has very few examples and most of the descriptions are brief. On the other hand, it should provide sufficient information for an experienced C programmer to write simulations in the YACSIM language.

1.1. SIMULATION OBJECTS

The YACSIM extensions to C are organized as a set of objects, each with an underlying data structure and a set of operations for manipulating that structure. These objects will be called *simulation objects* in this manual. There are several types of simulation objects, and the user can declare multiple instances of each.

We group the simulation objects into three categories: *activities*, *queues*, and *statistics records*. Activities model the active components of a simulation. They account for the passage of time and modify the state of a simulation. There are two types of queues, those used for synchronization and those used to model resources. We use the term queue for this group of objects because they all contain an internal queue to hold waiting activities. Statistics records simplify the collection and presentation of information generated by a simulation.

There are two types of activities: *events* and *processes.* They are used to represent activity in a simulated system. Some simulation languages only have events for this purpose, and the only way a simulation can advance time is to schedule an event to occur at some time in the future. These languages are called *event-driven*. Simulation languages that are based on processes are said to be *process-oriented*. In process-oriented simulations, the processes can account for the passage of time by delaying themselves for some time interval. In general, event-driven simulators are more efficient than those that are process-oriented. On the other hand, process-oriented simulators are usually thought to be easier and more natural to use. Since YACSIM has both events and processes, the user can write either event-driven or process-oriented simulations or can mix the two. Furthermore, YACSIM is implemented so that the extra overhead of process-oriented simulation is not incurred unless processes are used.

The queues used for synchronization are *semaphores*, *barriers*, *flags*, and *conditions*. They are used to delay the progress of a process or the occurrence of an event until some conditions are met. A process or event waits in the internal queues of these objects until this happens. Semaphores, barriers, and flags are similar to synchronization objects found in many parallel programming languages and operating systems. Conditions are a generalized form of these other three. They use expressions involving simulation objects called *state variables.* There are two types of state variables: *integer state variables* and *floating point state variables*.

Resources are included to simplify the simulation of queuing systems consisting of queues and servers. In these systems, processes request service from a resource server. If all servers are busy, the process waits in the resource's internal queue until one becomes free. The order in which processes are removed from a resource queue is determined by rules called the resource's queuing discipline. Several common queuing disciplines are implemented.

There is only one type of statistics record. It is a simulation object that can be used to collect information about a simulated system. The statistics records in YACSIM work on sequences of weighted values sent them during a simulation. They can be used to compute the mean, variance, max, min, and histogram of such a sequence. Operations for displaying the information collected by statistics records are provided. They can be used to generate a report in a standard form, or to construct one in a user-defined format.

In addition to the procedures used to manipulate the simulation objects, there are a few other procedures provided to control the simulation. The user uses these to initiate the execution of a simulation, to reset the simulator for another run, and to interrupt the simulator during a simulation.

1.2. USING SIMULATION OBJECTS

A simulation object consists of a data structure and operations that can be used to manipulate that data structure. YACSIM has been implemented to encourage the user to perform all manipulation of these data structures through the operations provided for that purpose. The details of the data structures are hidden from the user and should not be needed to use the simulation objects. This approach is a standard feature of object oriented systems and is generally viewed as a desirable way to structure programs. Unfortunately, C does not have the ability to hide global names from the user. Therefore, all internal YACSIM global names begin with the characters "YS__". To avoid name conflicts, the user should avoid using any names that start this way.

All of the simulation object operations access the simulation objects through pointers. For each object there is an operation that creates a new instance of the object and returns a pointer to it. Since these operations perform necessary initialization of the objects, the user should always use them when a new object is needed.

For each simulation object, a new data type (implemented as a C typedef) is available to the user. These new types are summarized in the following table. Although the IVAR and FVAR objects do not contain queues, we have grouped them with the queue objects, since they are only used with conditions that do contain queues. Since all access to the objects should be through the operations provided by YACSIM, and they only work on pointers to the objects, the user should only declare pointers to these new data types instead of declaring instances of them. These pointers must then be initialized with the operation that creates and returns a pointer to a new object of that type.

Activities	Queues	Statistics Records
PROCESS	SEMAPHORE	STATREC
EVENT	BARRIER	
	FLAG	
	CONDITION	
	IVAR	
	FVAR	
	RESOURCE	

Table 1. Simulation Object Types

1.3. THE FUNCTIONS main() AND UserMain()

Unlike standard C programs, your program must not contain the function main(). In its place, you must use a function called *UserMain().* The function main() is in the simulation library. As usual, it will be called to start the simulation. It will perform several initializations and then call the function UserMain(). When UserMain() terminates, it will return to main() which will also terminate. The usual way to get a simulation started is to create one or more events or processes and schedule them from within UserMain(). Then transfer to the driver using the DriverRun() operation (see Chapter 5).

1.4. COMMAND LINE ARGUMENTS

There are some pre-defined command line arguments that can be used to control certain aspects of a simulation. They are stripped off by the simulator at the start of a simulation and used to control the display of messages at the beginning and end of a simulation, the level of program tracing, and the type of event list that will be used. If used, they must appear at the very beginning of the list of command line arguments. The pre-defined command line arguments are listed below:

+t*i* sets the level of trace output produced to *i*. Tracing is explained in Section 6.3. The default level is 0, which turns off all tracing.

+b*i* Sets the number of bins for the event list's calendar queue to *i*. If $i = 1$, a simple sequential search event list will be used instead of the calendar queue. The calendar queue implementation of the event list is described in Section 5.2. The default event list is a calendar queue with automatic bin sizing.

+w*x* Sets the bin size for the event list's calendar queue to *x*. The calendar queue implementation of the event list is described in Section 5.2. The default event list is a calendar queue with automatic bin sizing.

+l Selects the simple linear linked list implementation of the event list. Using this argument has exactly the same effect as the +b1 argument.

-h Suppresses messages at the beginning and end of a simulation.

-i Suppresses the printing of all unique ID numbers in the trace output. The number 0 is printed instead. The ability to do this is sometimes useful during debugging for comparing two traces for similar simulations where the unique ID is different for almost all objects, but there are few other differences.

Once the pre-defined control arguments have been read and processed, all the remaining command line arguments are passed to UserMain() unchanged. They are accessed using *argc* and *argv* in the same way command line arguments are accessed in main().

1.5. COMPILING YACSIM SIMULATIONS

To compile a simulation program under the UNIX operating system, you need access to two files, *sim.h* and *yacsim.o*. The file *sim.h* should be included in all the files that make up your program. It contains useful pre-defined symbols, declarations of all the YACSIM operations available to the user, and definitions of the simulation object types. The file *yacsim.o* is the library of all YACSIM operations.

To compile a simulation program, use a command line of the form:

cc "your options for the compiler" "your files" yacsim.o

You may include any options you want for the C compiler such as -g, -o, etc. To use this command, you must put *sim.h* and *yacsim.o* where the compiler can find them, for example in the same directory as the source code for the simulation program, or use full path names for them.

An alternative way to compile a simulation program is to use a command line of the form:

yacsim "your options for the compiler" "your files"

To use this form, both sim.h and yacsim.o, along with the command file yacsim, must all be in the same directory, and that directory must have been passed to the command program yacsim when it was compiled. The full name of this directory must be on your search path.

2. ACTIVITIES

There are two types of activities: processes and events. When you create a process or event, you assign it a user-defined C procedure that specifies its action. We call this procedure the *body* of the process or event.

The main difference between a process and an event is that the body of a process can temporarily suspend execution, while the body of an event can not. That is, once the body of an event starts executing, it must continue until it terminates. As a result, processes can have a lifetime that extends over a period of simulation time, while an event occurs at one instant in simulation time. Simulation time can advance during the lifetime of a process, but not while the body of an event executes.

An important characteristic common to both types of activities is that you can schedule them to "happen" in the future. You do this by specifying the conditions which must be met for the activity to happen. Once scheduled, the activity is in limbo until these conditions are met. At that time it starts executing its body.

This chapter describes the properties and uses of the two types of YACSIM activities. Processes and events are scheduled in the same way. Therefore, we will describe the scheduling operations and those other operations that are common to all activities in Section 2.1. Sections 2.2 and 2.3 describe those features that are unique to processes and events, respectively.

2.1. OPERATIONS COMMON TO ALL ACTIVITIES

There are five different ways to specify the conditions that determine when an activity will happen. You can schedule an activity to happen:

1. after a given time delay,
2. when a given semaphore value is positive,
3. when a given flag is set,
4. when a given condition holds, or
5. after a requested amount of service from a resource.

There is a separate scheduling operation for each of these five ways to schedule an activity.

The detailed operation of semaphores, flags, conditions, and resources is the subject of a later chapter. For now, it is not necessary to know how to change semaphore values or to set flags in order to discuss the scheduling of events. For the following discussion, it is also sufficient to know that a condition is defined by an expression that is either true or false. If this expression is true, we say that the condition holds. Resources contain a queue of activities waiting for service and one or more servers. The order in which activities waiting in a resource's queue receive service from its server(s) is determined by the resource's queuing discipline.

You can schedule an activity at any point in a program that has access to the activity. In particular, you can schedule it from within the body of a process or event as well as from within code segments that are not part of any activity. If you schedule an activity from within a process, this can have one of the following three effects on that process:

1. It can cause the scheduling process to suspend until the scheduled activity terminates. In this case we say that the scheduling process is *blocked* waiting for the activity to terminate.

2. It can link the scheduled activity with other activities scheduled by the same process. We say that all such linked activities are *forked* by the process. Later, the scheduling process can execute a "join" operation that will cause it to suspend until all of its forked activities have terminated. The process that performs a forking schedule is called the *parent* of the activities it forks, and the forked activities are called *children* of the parent process.

3. Finally, the action of an activity scheduled by a process need not have any effect on the process. In this case, we say that the scheduled activity and the scheduling process are *independent*.

Processes terminate when their bodies terminate. The suspension of a process is not a termination. Events normally terminate when their bodies terminate (i.e., the body routine returns). However, an event can be designated as non-deleting, in which case it can be scheduled multiple times before it terminates. These events are not viewed as terminated by a forking or blocked parent process until they are converted to a deleting type event as explained in Section 2.3.

It is frequently useful to pass an argument to an activity when it is scheduled. This is similar to passing arguments to subroutines. YACSIM uses the operations ActivitySetArg(), ActivityGetArg(), and ActivityArgSize() for this purpose.

Operations:

The following five scheduling operations all use the arguments *aptr* that points to the activity to be scheduled and *blkflg* to specify the effect of a scheduled activity's termination on a scheduling process as described above. A non-deleting event (defined in the section on events below) may reschedule itself by calling the schedule operation from within its body with a NULL *aptr* argument. The pre-defined symbol ME can also be used for *aptr* in this case. Otherwise, *aptr* must point to some activity or the simulator will generate an error message and terminate. The allowable values for *blkflg* are INDEPENDENT, BLOCK, and FORK. Scheduling a forked or blocking activity (i.e., *blkflg* = BLOCK or FORK) other than from within the body of a process will generate an error message and terminate the simulation.

void ActivitySchedTime(aptr, timeinc, blkflg)
*ACTIVITY *aptr;*
double timeinc;
int blkflg;

> This operation schedules the activity pointed to by *aptr* to happen in *timeinc* units of time. If timeinc is 0.0, the activity happens at the current simulation time. If it is less than 0.0, an error termination will occur.

void ActivitySchedSema(aptr, semptr, blkflg)
*ACTIVITY *aptr;*
*SEMAPHORE *semptr;*
int blkflg;

> This operation schedules the activity pointed to by *aptr* on the semaphore pointed to by *semptr*. If that semaphore is positive at the time the activity is scheduled, then the activity is initiated immediately and the semaphore value decremented. If the

semaphore's value is less than or equal to zero, the initiation of the activity is delayed until the semaphore becomes positive. See the section on semaphores below for a description of what happens when several activities are waiting at a semaphore when it becomes positive.

void ActivitySchedFlag(aptr, flgptr, blkflg)
*ACTIVITY *aptr;*
*FLAG *flgptr;*
int blkflg;

This operation schedules the activity pointed to by *aptr* on the flag pointed to by *flgptr*. If that flag is in the set state at the time the activity is scheduled, then the activity is initiated immediately and the flag is cleared. If the flag is in the cleared state when the activity is scheduled, the initiation of the activity is delayed until a FlagSet() or FlagRelease() operation is performed on the flag. At that time all activities waiting on that flag are initiated.

void ActivitySchedCond(aptr, condptr, blkflg)
*ACTIVITY *aptr;*
*CONDITION *condptr;*
int blkflg;

This operation schedules the activity pointed to by *aptr* on the condition pointed to by *condptr*. If that condition holds (i.e., has value TRUE) at the time the activity is scheduled, then the activity is initiated immediately. If the condition does not hold (i.e., has value FALSE) when the activity is scheduled, the initiation of the activity is delayed until the condition does hold. At that time all activities waiting on that condition are initiated.

void ActivitySchedRes(aptr, rptr, timeinc, blkflg)
*ACTIVITY *aptr;*
*RESOURCE *rptr;*
double timeinc;
int blkflg;

This operation schedules the activity pointed to by *aptr* for *timeinc* units of service from the resource pointed to by *rptr*. The activity will wait in the resource's queue for its turn to use one of the resource's servers.

The following operations are used to set and access the arguments of an activity. The argument *aptr* points to that activity. If a process or event uses any of these operations with *aptr* equal to NULL or ME, it sets or accesses its own argument. Otherwise, *aptr* must point to an activity or the simulation will terminate with an error message.

void ActivitySetArg(aptr, argptr, argsize)
*ACTIVITY *aptr;*
*char *argptr;*
int argsize;

The variable *argptr* can point to an arbitrary structure and *argsize* is the size of that structure in bytes. Execution of this operation passes this pointer to the activity pointed to by *aptr*. Note that this operation only passes a pointer to the process, not a value. Therefore, when the process accesses the argument pointed to by this pointer, it will get the value of the argument at the time it is accessed, and this may not be the same as its value when the argument was set.

*char *ActivityGetArg(aptr)*
*ACTIVITY *aptr;*

This operation returns a pointer to the arguments of the activity pointed to by *aptr*. It is usually executed within the body of a process or event (with aptr = NULL or ME), and is the way these activities access their arguments. The activity must know what is pointed to by this pointer and use a cast to change the pointer from a character pointer. Moreover, this operation will return the value of the argument at the time the operation is executed, and this may not be the same as its value when the argument pointer was set.

int ActivityArgSize(aptr)
*ACTIVITY *aptr;*

This operation returns the size of the argument of the activity pointed to by *aptr*. A return value of -1 means the size is unknown.

The following two operations provide an activity with a way to get a pointer to itself and to its parent, if it has one.

*ACTIVITY *ActivityGetMyPtr()*

This operation returns a pointer to the currently active activity. It must be called from within the body of a process or an event, and it returns a pointer to that process or event.

*ACTIVITY *ActivityGetParPtr()*

This operation returns a pointer to the parent of the currently active activity. It must be called from within the body of a process or an event. If that process or event has been scheduled with its block flag set to FORK (i.e., if the activity is the child of some parent process), the operation returns a pointer to the parent process. Otherwise it returns a NULL pointer.

Statistics can be collected on the states of an activity over its lifetime. The following operations are used to activate this feature and to access the collected statistics. As with other operations on activities, these operations can be called from within the body of a process or an event by setting the pointer *aptr* to NULL or ME. Note that when an activity terminates its associated statistics record is deleted.

void ActivityCollectStats(aptr)
*ACTIVITY *aptr;*

This operation activates statistics collection for the activity pointed to by *aptr*. It does this by creating a statistics record (see Chapter 4) that records the time the activity spends in each state. The possible states are:

Processes and Events:

LIMBO - The activity has just been created.
READY - The activity is ready to execute its body, but has not yet started.
RUNNING - The activity is executing its body.
DELAYED - The activity is scheduled for activation in the future.
WAIT_SEMAPHORE - The activity is waiting for a semaphore to go positive.
WAIT_FLAG - The activity is waiting for a flag to be set.
WAIT_CONDITION - The activity is waiting for a condition to hold.

WAIT_RESOURCE - The process is in a resource queue.
USING_RESOURCE - The process is being served by a resource.

Processes only:

BLOCKED - The process is waiting for an activity it scheduled to occur.
WAIT_JOIN - The process is waiting for its forked activities to finish.
WAIT_BARRIER - The process is waiting at a barrier.
WAIT_MESSAGE - The process is blocked waiting for a message.

void ActivityStatRept(aptr)
*ACTIVITY *aptr;*

This operation prints a report on the statistics of the activity pointed to by *aptr*. This report will give the time and the percentage of the total time that the activity spends in each state.

*STATREC *ActivityStatPtr(aptr)*
*ACTIVITY *aptr;*

This operation returns a pointer to the statistics record associated with the activity pointed to by *aptr*. Chapter 4 describes the various operations that can be performed on a statistics record once you have a pointer to it.

Examples:

ActivitySchedTime(procptr, 0.0, INDEPENDENT);

Schedule the independent process pointed to by procptr to start immediately.

ActivitySchedTime(procptr, 4.5, INDEPENDENT);

Schedule the independent process pointed to by *procptr* to start in 4.5 time units.

ActivitySchedSema(procptr, semptr, BLOCK);

Schedule the process pointed to by *procptr* to start when the semaphore pointed to by *semptr* is positive, and then suspend the calling process until the scheduled process terminates. This blocking form of ActivitySchedSema() can only be invoked by a process.

ActivitySchedCond(procptr, condptr, FORK);

Fork the process pointed to by *procptr* to start when the condition pointed to by *condptr* holds. This forking form of ActivitySchedCond() can only be invoked by a process.

ActivitySchedFlag(evptr, flgptr, INDEPENDENT);

Schedule the independent event pointed to by *evptr* to occur when the flag pointed to by *flgptr* is set.

ActivitySchedTime(ME, 18.4, INDEPENDENT);

If this operation is executed from within a non-deleting event's body, then that event will be rescheduled to occur in 18.4 time units. Execution from any place other than from within a non-deleting event body is an error.

ActivitySchedTime(evptr, 0.0, BLOCK);

Schedule the event pointed to by *evptr* to occur immediately and then suspend the calling process until the event occurs.

float x = 5.4;
ActivitySetArg(procptr, &x, sizeof(float));

A pointer to a floating point argument with value 5.4 is passed to the process pointed to by *procptr*.

*float y = *((float*)ActivityGetArg(ME));*

If this statement is executed from within the body of the process pointed to by *procptr* in the previous example, it will set y to the value 5.4. The cast is necessary to prevent a C compile time warning message.

Comments:

Although the scheduling operations work for both types of activities, not all possibilities of scheduling activities make sense. For example, you can not fork or schedule a blocking process from any place except within the body of a process, since processes are the only objects that can suspend.

Once you have scheduled a process, you can not schedule it again. However, under certain conditions, you can schedule an event any number of times. For example, you can reschedule a non-deleting event once it occurs. Attempts to schedule a process more than once or an event that has not yet occurred will generate error messages. Section 2.3 discusses the conditions for rescheduling events in more detail.

It would probably be preferable to pass arguments to processes and events on a stack in the same way they are passed to subroutines. While we could do this for processes, events do not have a stack until they occur, and there is no other convenient place to put their arguments. In order to treat both processes and events as uniformly as possible, we have chosen this somewhat more cumbersome way of passing arguments using ActivitySetArg(), ActivityGetArg(), and ActivityArgSize().

If you only need one integer argument, the best way to pass it is to set the argument pointer to NULL and the argument size to the value of the argument to be passed. The process can then access this argument with the ActivityArgSize() operation. This is similar to passing the argument on a stack, since a value is passed instead of a pointer to a value. A situation where this is very useful is when you want to create several processes with the same body by using a loop. In this case it is difficult to give each process a different name or to pass a different argument value to each one using argument pointers. If you need some way of distinguishing between the different processes, passing a single integer that is incremented for each process created is a simple way to do this.

2.2. PROCESSES

Process-oriented simulators use processes to model the components of a simulated system. The usual approach to writing such a simulation is to identify the main components of the system to be simulated, and then, for each one, design a process to simulate its behavior. An important characteristic of processes is that they can model the behavior of components that operate concurrently. The simulator coordinates the advancement of simulation time as the processes execute in a way that accounts for this concurrent behavior.

Another important characteristic of processes is that their execution can be suspended and then resumed at a later simulation time. This is one way that the passing of time is modeled during a simulation. For example, a process can delay itself for a given time increment by executing a "delay" operation. This halts the execution of the process and puts it on a list of delayed processes. When simulation time advances by the amount of the time increment, the process is removed from this list and its execution restarted at the point of interruption. Processes can also delay their execution until some other process performs some specified action. The main mechanisms for this second type of delay are the synchronization and resource objects described in Chapter 3.

For each YACSIM process the user must specify a procedure that defines its behavior. We call this procedure the *body* of the process. It can be any C procedure that returns void (i.e., does not return a value) and has no arguments. You must specify its body when you create a process, and you can not change it after that. You can create several processes that use the same procedure as their bodies.

Operations:

*PROCESS *NewProcess(pname, bodyname, stksz)*
*char *pname*
func bodyname;
int stksz;

> This operation creates a new process and returns a pointer to it. The argument *pname* is a user-specified name for the process that is used in debugging traces. *Func* is a typedef specifying a pointer to a function that returns void and has no arguments. *Bodyname* is any pointer to such a function (i.e., the name of the function). Each process has its own private stack, and the argument *stksz* specifies the size of this stack in bytes. If *stksz* is 0 or the pre-defined symbol DEFAULTSTK, a default stack size will be used.

void ProcessSetStkSz(stksz)
int stksz;

> The simulator has a built in default stack size that should be satisfactory for most simulations. This operation allows the user to change that default value to *stksz*. The operation must be executed at the very beginning of a simulation, before the operation NewProcess() is called for the first time. Otherwise, it will print a warning message and leave the default stack size unchanged.

void ProcessDelay(timeinc)
double timeinc;

> You can only invoke this operation from within the body of a process. Its execution causes the suspension of the process for *timeinc* units of simulation time. If the argument is 0.0, the process execution is halted and other processes scheduled for execution at the same time are allowed to continue. If there are no other such processes, the process continues immediately. This option gives the programmer a little control over the order that processes scheduled for the same time will actually execute on a uniprocessor. A negative time increment is not allowed.

void ProcessSleep()

> You can only invoke this operation from within the body of a process. Its execution causes the suspension of that process for an indefinite amount of simulation time. The

only way to wake up a sleeping process is to schedule it using one of the activity scheduling operations.

void ProcessJoin()

This is an operation used for synchronization with forked activities. You can only invoke it from within the body of a process. Once executed it will suspend the calling process until all activities forked by that process are finished. That is, all forked processes must have terminated and all forked events must have occurred. Once this happens, the suspended process continues with the next instruction in its body. If there are no outstanding forked activities when ProcessJoin() is executed, the process continues without interruption.

void ProcessSetPriority(procptr, p)
*PROCESS *procptr;*
double p;

This operation sets the priority of the process pointed to by *procptr* to *p*. Priorities are only used when a process requests service from a resource. Resources are explained in Section 3.5. If ProcessSetPriority() is called from within the body of a process with *procptr* = NULL or ME, that process sets its own priority.

Activities can send messages to processes. These messages are similar to the messages found in many message-based operating systems. Activities use the operation ProcessSendMsg() to deliver messages to a process. Once a message has been delivered, it waits in a queue with other messages delivered to the same process. A process must explicitly receive each delivered message (taking it out of the queue) with a separate invocation of ProcessReceiveMsg().

void ProcessSendMsg(dest, buf, bytes, blkflg, type)
*PROCESS *dest;*
*char *buf;*
int bytes;
int blkflg;
int type;

This operation can only be called from within the body of an activity. It creates a message consisting of *bytes* characters taken from the buffer pointed to by *buf* and delivers it to the process pointed to by *dest*. It also assigns the message a type equal to the argument *type.* This must be an integer greater than or equal to 0 or an error termination of the simulation will occur. The allowable values for *blkflg* are BLOCK and NOBLOCK. If *blkflg* equals BLOCK, this operation will not return until the message has been delivered to, and received by, the destination process. Setting *blkflg* to NOBLOCK will allow the operation to return as soon as the message has been delivered to the destination process. If *buf* is NULL or *bytes* is 0, a null message (i.e., one with no contents) is sent. Null messages are only used for synchronization, not to send information.

int ProcessReceiveMsg(buf, bytes, blkflg, type, sender)
*char *buf;*
int bytes;
int blkflg;
int type;
*PROCESS *sender;*

A process uses this operation to receive delivered messages. Since only processes can receive messages, you can only invoke the operation from within the body of a process. The operation receives a message by moving the message contents into a buffer pointed to by *buf*. If the size of the message is less than or equal to the argument *bytes,* all the characters in the message are copied into the buffer, and the operation returns the number of characters received. If the message is larger than *bytes,* only its first *bytes* characters will be moved, and the operation will return the value -2. The remaining characters of the message can be obtained with subsequent calls to ProcessReceiveMsg(). If the operation returns 0, then a null message was received. A blocking sender will not be released until the entire message it sent has been received. The arguments *type* and *sender* can be used to restrict the type and origin of the message that will be received. *Type* must match the type of the message as set by the sender, and *sender* must match a pointer to the sending process, before a message will be received. The operation will search the entire queue of delivered messages, in the order of their delivery, looking for the first one that matches these arguments. The pre-defined symbols ANYTYPE and ANYSENDER may be used to accept messages of any type and from any sending process. If the value of *blkflg* is BLOCK, and there no messages available that match the *type* and *sender* arguments, then the calling process will suspend until one is delivered. If *blkflg* is NOBLOCK and no such message has been delivered, then ProcessReceiveMsg() will return the value -1.

*int *ProcessCheckMsg(type, sender)*
int type;
*PROCESS *sender;*

A process can use this operation to check whether or not it has a message of type *type* from the process pointed to by *sender* in its queue of delivered messages. It can only be used from within the body of a process. It returns -1 if there are no delivered messages of the indicated type and sender waiting to be received. If there is such a message, the operation returns the size of the message.

Examples:

procptr1 = NewProcess("SW1", switch, DEFAULTSTK);

This statement creates a new process with the default stack size and with the function *switch* as its body and the character string "SW1" as its name. The variable *procptr1* must have been previously declared as a pointer to PROCESS, and *switch* must be the name of a function with no arguments and no return value.

procptr2 = NewProcess("SW2", switch, 1000);

This statement creates a second process with the same body function as the previous one, but with a 1000 byte stack.

ProcessDelay(10.3);

This statement delays the calling process for 10.3 units of time. Note that this must be called only from within a process.

ProcessJoin();

The calling process waits for all forked activities to terminate.

ProcessSendMsg(prptr, sndbuf, 20, NOBLOCK, 0);

Twenty characters from the buffer pointed to by *sndbuf* will be sent to the process pointed to by *prptr*. The operation will return as soon as the message has been delivered, but before it has been received. The type of the message is set to 0.

ProcessSendMsg(pp, sb, 100, BLOCK, 3);

This instance of the ProcessSendMsg() operation will send a message of type 3 consisting of 100 characters from the buffer pointed to by *sb* to the process pointed to by *pp*. It is a blocking send so that the calling process will suspend until all of the message has been received.

i = ProcessReceiveMsg(rbuf, 1000, BLOCK, ANYTYPE, ANYSENDER);

The calling process tries to receive a message from any sender and of any type. If there is such a message and its size is less than or equal to 1000 characters, the whole message is copied to the buffer pointed to by *rbuf,* and *i* is set to the number of characters in the message. if there are more than *1000* characters in the message, the operation only copies *1000* characters to the buffer and then sets *i* to -2. The remaining characters can be read by another call to Process ReceiveMsg(). If there are no messages available, the operation will suspend until one is delivered.

i = ProcessReceiveMsg(rbuf, 250, NOBLOCK, 4, rptr);

This operation attempts to receive a message of type 4 from the process pointed to by *rptr*. If none is available, it will set *i* to -1 and return. If there is such a message and its size is less than or equal to 250, the whole message is copied to the buffer pointed to by *rebuf,* and *i* is set to the number of characters in the message. if there are more than 250 characters in the message, the operation only copies 250 characters to the buffer and then sets *i* to -2. The remaining characters can be read by another call to Process ReceiveMsg().

Comments:

It is possible to overflow a process' stack during a simulation. This usually results in the simulation crashing for no apparent reason. It can also crash at different points if the simulation code is changed (e.g., by putting in *printf* statements to find out what is happening). A good rule to follow is to try increasing the stack size of suspect processes if there is no other obvious reason for a crash. On some implementations the maximum size of the stack used by a process is printed out in the debugging trace when that process terminates. Unfortunately, this is very machine dependent and may not be implemented on some machines.

2.3. EVENTS

Events are similar to processes. When you create an event, you must assign it a body in the same way you assign a process body. When an event occurs, the body is executed. The significant difference is that the body of an event can not suspend execution. This means that once an event body has started execution, it will continue until it reaches a return point. More importantly, if the same event is activated again, it will start executing its body at the same entry point as before. It can not remember where it stopped and restart at that point the way a process can. To put it another way, execution of an event body is the same as a subroutine call, where processes use coroutine linkage to implement suspensions.

The conditions for the body function of an event are the same as for a process. It must be a C function that returns void and has no arguments. Like processes, several events can use the same body function. When you create an event, you can designate it to be a *deleting* or a *non-deleting* event. Just as processes are automatically deleted when they finish executing, a deleting event is automatically deleted after it occurs. A non-deleting event, on the other hand, is not deleted after it occurs and can be used again. Another useful feature of non-deleting events is that they can reschedule themselves. That is, any of the activity scheduling operations for an event can appear in the body of that event.

You can also assign a type to an event. An event's type is an arbitrary integer value that you can assign at the time of creation or later with the event operation EventSetType(). Event types are not currently used by the simulator; they are provided for users to use anyway they want. For example, a user could collect statistics on event types to count the number of event occurrences of a given type.

Operations:

*EVENT *NewEvent(ename, bodyname, delflg, etype)*
*char *ename;*
func bodyname;
int delflg;
int etype;

> This operation creates a new event and returns a pointer to it. The argument *bodyname* specifies the body function and *ename* names the object. The third argument determines whether the event will be deleting or non-deleting. The only two possible values for this argument are DELETE and NODELETE. The last argument is the event's type. The event's state, used for rescheduling (see event rescheduling operations below), is set to 0.

void EventSetType(eptr, etype)
*EVENT *eptr;*
int etype;

> Execution of this operation sets the type of the event pointed to by *eptr* to *etype*. You can use it to change the type of a non-deleting event before reusing it. You can invoke it within the body of an event with *eptr* = NULL or ME, enabling an event to change its own type. This probably only makes sense for non-deleting events that reschedule themselves.

int EventGetType(eptr)
*EVENT *eptr;*

> This is a function that returns the type of the event pointed to by *eptr*. You can also invoke it from within the body of an event with *eptr* = NULL or ME, which allows events to know their types.

int EventGetDelFlag(eptr)
*EVENT *eptr;*

> This function allows the programmer to determine whether or not the event pointed to by *eptr* is a deleting event by returning the value of its delete flag. You can also invoke it from within the body of an event with *eptr* = NULL or ME.

The following operations allow a deleting event to reschedule itself and save its state for use during the next occurrence of that event. The state was set to 0 when the event was created. The saved state can be used to give an event a limited capability to suspend and resume operation in a way similar to processes. This is illustrated in the last example below. All of these operations are similar to the corresponding activity scheduling operations with their activity pointer argument set to ME. The only difference is that they also save a state value and can only be called from within a non-deleting event body.

void EventReschedTime(timeinc, stval)
double timeinc;
int stval;

If this operation is executed from within the body of a non-deleting event, it will reschedule the event to occur again in *timeinc* time units and save the value *stval* for use during the next occurrence. Executing the operation from anywhere other than the body of a non-deleting event is an error.

void EventReschedSema(semptr, stval)
*SEMAPHORE *semptr;*
int stval;

This operation reschedules an event on a semaphore and saves the value *stval* for use during the next occurrence of the event. It can only be used within the body of a non-deleting event.

void EventReschedFlag(flgptr, stval)
*FLAG *flgptr;*
int stval;

This operation reschedules an event on a flag and saves the value *stval* for use during the next occurrence of the event. It can only be used within the body of a non-deleting event.

void EventReschedCond(condptr, stval)
*CONDITION *condptr;*
int stval;

This operation reschedules an event on a condition and saves the value *stval* for use during the next occurrence of the event. It can only be used within the body of a non-deleting event.

void EventReschedRes(resptr, timeinc, stval)
*CONDITION *condptr;*
double timeinc;
int stval;

This operation reschedules an event to use *timeinc* units of service from a resource and saves the value *stval* for use during the next occurrence of the event. It can only be used within the body of a non-deleting event.

int EventSetState(stvat)
int stval;

This operation must be executed from within the body of an event. It sets the state of that event to *stval*.

int EventGetState()

This operation returns the state value saved by a previously executed event rescheduling operation. It must be executed from within the body of an event.

void EventSetDelFlag()

This operation can only be called from within the body of an active event. If the event is non-deleting, this operation changes it to a deleting event. It has no effect on a deleting event. Once a non-deleting event's delete flag is set, it will terminate and be deleted the next time it returns as a subroutine. If there were blocked parent processes waiting for this event to terminate, they will then be released.

Examples:

evptr1 = NewEvent("PK1", packet, DELETE, 0);

This creates a new event with the function *packet* as its body and *"PK1"* as it name. Here *evptr1* is a pointer to type EVENT. The event will be a deleting event of type 0.

evptr2 = NewEvent("PK2", packet, DELETE, 0);

This creates a deleting event with the same body function as the previous one.

evp = NewEvent("SP", sndpck, NODELETE, 3);

This statement declares a non-deleting event with function *sndpck* as its body and 3 as its type.

EventSetType(evptr1, 6);

This sets the type of the event pointed to by *evptr1* to 6.

EventSetType(ME, 1);

This is the form of the operation if it is executed within the body of the event itself. In this way it is possible for an event to change its type and then reschedule itself.

i = EventDeleteFlag(evptr);

This sets the integer *i* to the value of the delete flag of the event pointed to by *evptr*.

j = EventGetType(evptr);

This sets the integer *j* to the type of the event pointed to by *evptr*.

```
eventbody() {
        switch (EventGetState()) {
        Case 0:
                statement1
                EventReschedTime(3.0, 1)
                return;
        Case 1;
                statement2
                EventReschedSema(sptr, 2)
                return;
        Case 2:
                statement3
                EventSetDelFlag()
                return;
        }
}
```

An event using this function as its body would be similar to a process that

1. executes statement1,
2. delays for 3.0 time units,
3. resumes and executes statement2,
4. waits at the semaphore pointed to by sptr,
5. resumes and executes statement3,
6. terminates.

The event must be created as a non-deleting event. The EventSetDelFlag() operation causes the event to be deleted when it terminates the next time.

Comments:

Although there are a number of differences between processes and events, probably the most important one is that events can be implemented more efficiently than processes. This is because every process must have its own private stack, which must exist during the lifetime of the process. This can severely limit the size of a simulation.

The reason for including the event rescheduling operations is to give an event some of the appearances of a process. While it is not nearly as powerful as a process, it does not need a separate stack. The biggest limitation of this approach to using events is that they can not easily resume operation at a point within a subroutine called from the event's body. A process can suspend anywhere and then easily resume execution at that point. Another limitation is that the local variables of an event's body are not saved when the body returns, and static variables can not be used, since different events may use the same body.

3. QUEUES

Queues are used for two purposes, to synchronize activities and to model the use of resources by processes. There are four types of queues used for synchronization: *semaphores*, *barriers*, *flags*, and *conditions*. They all contain a simple queue in which an activity can wait until some synchronization condition holds. You can use them to synchronize processes and events. Resources model the behavior of an activity when it requests service from a resource. If the resource is busy when an activity requests service, that activity waits in the resource's queue.

3.1. SEMAPHORES

Each semaphore has a value and two special operations SemaphoreWait() and SemaphoreSignal(). When a process executes SemaphoreWait() on a semaphore, there are two possible results. First, if the value of the semaphore is greater than zero, that value is decremented and the process continues. On the other hand, if the semaphore value is equal to zero, the process suspends and enters the semaphore's internal queue at its tail. In this case, the value of the semaphore does not change. You can only execute SemaphoreWait() within the body of a process, because processes are the only activities that can suspend.

You can signal a semaphore from anywhere within the code of a simulation. The operation SemaphoreSignal() first checks to see if there are any processes in the semaphore's queue. If there are, it then removes the one at the head of the queue and restarts it at the point where it suspended, but does not change the value of the semaphore. If there are no waiting processes, the only effect of SemaphoreSignal() is to increment the value of the semaphore. Since processes are removed from the head of a semaphore's queue and entered at its tail, the queue uses a FIFO queuing discipline.

You can use the operation ActivitySchedSema() to place an activity at the tail of a semaphore's queue. Since ActivitySchedSema() is an operation defined for all activities, its use for this purpose was explained in Section 2.1.

Operations:

*SEMAPHORE *NewSemaphore(sname, i)*
*char *sname;*
int i;

> This operation creates and returns a pointer to a new semaphore with name *sname* and initial value *i.*

int SemaphoreInit(sptr, i)
*SEMAPHORE *sptr;*
int i;

> This operation sets the value of the semaphore pointed to by *sptr* to *i*, if the semaphore's queue is empty. Otherwise, it does not affect the status of the semaphore in any way. It returns the number of activities in the semaphore's queue. Therefore, the operation is successful if and only if it returns 0.

void SemaphoreSignal(sptr)
*SEMAPHORE *sptr;*

> This operation removes the activity at the head of the queue of the semaphore pointed to by *sptr*, if the queue is not empty. If that activity is an event, it starts executing at the

beginning of its body. If the removed activity is a suspended process it continues executing its body at the instruction after the instance of SemaphoreWait() where it suspended. If it is a process that entered the queue due to an invocation of ActivitySchedSema(), it starts executing at the beginning of the process' body. If there are no activities in the queue, this operation increments the value of the semaphore.

void SemaphoreSet(sptr)
*SEMAPHORE *sptr;*

This operation is similar to SemaphoreSignal() in that it removes the activity at the head of the queue of the semaphore pointed to by *sptr* and treats it the same way as SemeaphoreSignal(), if the queue is not empty. However, instead of incrementing the semaphore value if the queue is empty, this operation sets it to 1. If its value was 1 before, it remains 1. Therefore, this operation can be used to implement binary-valued semaphores, since its value will be either 0 or 1.

void SemaphoreWait(sptr)
*SEMAPHORE *sptr;*

You can only use this operation within the body of a process. If the value of the semaphore pointed to by *sptr* is zero, this operation will suspend the calling process. If the value of the semaphore is greater than zero, this operation will decrement the semaphore's value and the process will continue executing.

int SemaphoreDecr(sptr)
*SEMAPHORE *sptr;*

If the value of the semaphore pointed to by *sptr* is greater than 0, it is decremented. If the value is equal to 0, it is left unchanged. The operation returns the new value of the semaphore.

int SemaphoreValue(sptr)
*SEMAPHORE *sptr;*

This operation returns the value of the semaphore pointed to by *sptr*.

int SemaphoreWaiting(sptr)
*SEMAPHORE *sptr;*

This operation returns the number of activities in the queue of the semaphore pointed to by *sptr*.

Examples:

semptr = NewSemaphore("sem.2", 4);

This statement sets *semptr* to point to a new semaphore named "sem.2" with initial value 4.

SemaphoreSignal(semptr);

This statement signals the semaphore pointed to by *semptr*. If there are any activities in its queue, the one at the head is released. If not, the semaphore value is incremented.

SemaphoreWait(semptr);

This statement can only appear in the body of a process. If the semaphore's value is greater than zero, this operation decrements the value, and the process continues. Otherwise, the process suspends and waits in the semaphore's queue.

Comments:

SemaphoreWait() and SemaphoreSignal() are sometimes called P and V. We have chosen not to use these names since they are not as descriptive of the actions they perform as SemaphoreSignal() and SemaphoreWait().

3.2. BARRIERS

Barriers implement the barrier synchronization operation found in many parallel programming languages and operating systems. You can only use barriers to synchronize processes; you can not use them in any way with events. Like semaphores, barriers have a value that determines whether a process that synchronizes on it will suspend or continue. If the value of a barrier is *k*, then *k* processes must perform a synchronization operation on the barrier before any of them can proceed. It is used to guarantee that *k* concurrently executing processes all reach a synchronization point in their code before any one of them can progress further.

Operations:

*BARRIER *NewBarrier(bname, i)*
*char *bname;*
int i;

This operation creates and returns a pointer to a new barrier. It sets the name of the newly created barrier to *bname* and its value to *i*.

int BarrierInit(bptr, i)
*BARRIER *bptr;*
int i;

This operation sets the value of the barrier pointed to by *bptr* to *i*, if its queue is empty. Otherwise, it does not affect the status of the barrier in any way. It returns the number of processes in the barrier's queue. Therefore, the operation is successful if and only if it returns 0.

void BarrierSync(bptr)
*BARRIER *bptr;*

You can only use this operation within the body of a process. It puts the calling process in the queue of the barrier pointed to by *bptr* and checks to see if the number of processes in the queue is now equal to the barrier's value. If it is, all processes waiting in the queue, including the calling process, are removed and restarted. If not, the calling process suspends.

int BarrierNeeded(bptr)
*BARRIER *bptr;*

This function returns the number of additional of processes that must perform a BarrierSync() operation on the barrier pointed to by *bptr* before the barrier will release all waiting processes. The value of the barrier pointed to by *bptr* is BarrierNeeded(bptr) + BarrierWaiting(bptr).

int BarrierWaiting(bptr)
*BARRIER *bptr;*

This is a function that returns the number of processes waiting at the barrier pointed to by *bptr*. The value of the barrier pointed to by *bptr* is BarrierNeeded(bptr) + BarrierWaiting(bptr).

Examples:

barptr = NewBarrier ("syncpt",4);

This creates a barrier named "syncpt" that can be used to synchronize four processes and sets barptr to point to it.

i = BarrierInit(barptr, 3);

This changes the number of processes that can synchronize on the barrier pointed to by *barptr* to 3, if the barrier's queue is empty. It has no effect on the barrier if its queue is not empty. The return value *i* is set to the number of processes in the barrier's queue.

BarrierSync(barptr);

This must be executed from within the body of a process. That process will suspend unless it is the k-th process to synchronize on the barrier, where k is the value of the barrier. If it is, all waiting processes are released.

Comments:

There is a form of non-determinism inherent in the BarrierSync() operation. To see this, assume that the barrier needs *i* more processes to execute a Sync() operation before all waiting processes are released. If more than *i* processes execute the Sync() at the same simulation time, then the order in which the simulator actually executes these operations determines which processes will be released. While there does not appear to be any obvious way to avoid this situation, the system could check for it and issue a warning message. However, this is not done in the current version.

Synchronization similar to barrier synchronization is possible by forking processes and waiting for them to terminate with the operation ProcessJoin(). In one sense, this is more restrictive, since the processes must terminate before they can synchronize. Using barriers, the processes can synchronize any number of times before they terminate. You can also use barriers to synchronize non-terminating processes. However you can use the fork-join synchronization with events as well as processes, while you can only use barrier synchronization with processes.

3.3. FLAGS

Flags are similar to barriers in that they provide a way to make several processes wait at a synchronization point until something happens. In a barrier, all waiting processes are released when the number of waiting processes equals the barrier value. In a flag, all waiting processes are released by explicit operations on the flag. There are two such operations, FlagSet() and FlagRelease(). These operations also control the state of the flag as explained in the following section on operations. There are two possible states, *set* and *cleared*.

You can also use the operations ActivitySchedFlag() and EventReschedFlag() to place an activity at the tail of a flag's queue. Since ActivitySchedFlag() is an operation defined for all activities, its use for this purpose was explained in Section 2.1.

Operations:

*FLAG *NewFlag(fname)*
*char *fname;*

This operation creates and returns a pointer to a new flag. It sets the name of the flag to *fname* and its state to *cleared*.

void FlagWait(fptr)
*FLAG *fptr;*

You can only invoke this operation within the body of a process. If the flag pointed to by *fptr* is in the cleared state when this operation is executed, then the calling process suspends and enters the tail of the flag's queue, and the flag remains in the cleared state. If the flag is in the set state, then the calling process continues and the flag is cleared.

int FlagSet(fptr)
*FLAG *fptr;*

This operation sets the flag pointed to by *fptr*. Then, if there are any activities waiting in the flag's queue, they are removed from the queue and scheduled for execution, and the flag is cleared. If there are no waiting activities, the flag remains set. The return value is the number of activities released.

int FlagRelease(fptr)
*FLAG *fptr;*

This operation is similar to FlagSet(). Its execution releases any waiting activities and clears the flag. However, if there are no waiting activities, FlagRelease() clears the flag, while FlagSet() sets it. The return value is the number of activities released.

int FlagWaiting(fptr)
*FLAG *fptr;*

This operation returns the number of activities in the queue of the flag pointed to by *fptr*.

Examples:

flagptr = NewFlag("F1");

This creates a new flag with name "F1" and sets *flagptr* to point to it.

FlagWait(flagptr);

This statement is only valid within the body of a process. It suspends the process if the flag pointed to by *flagptr* is in the cleared state. If that flag is in the set state, the process continues, and flag is put in the cleared state.

FlagSet(flagptr);

This operation sets the flag pointed to by *flagptr*. Then, if there are any activities waiting at that flag, they are released and the flag is cleared. If there are no waiting activities, the flag remains set.

FlagRelease(flagptr);

This operation clears the flag pointed to by *flagptr*. If there are any activities waiting at the flag, they are released. Note that the state of the flag is always clear after this operation is performed.

Comments:

Some simulators and parallel programming languages use the term "event" instead of "flag" and the term "post" for the operation FlagSet(). Then, to say that "a process waits until an event is posted" is the same as saying that "a process waits until a flag is set." Since the activity we call an event is a central YACSIM concept, we have chosen to use the less conventional term "flag" to describe the synchronization primitive.

The operations FlagSet() and FlagRelease() have similar effects on a flag; either one will release processes that are waiting at the flag when it is applied. The difference occurs when you apply them to a flag with no waiting processes. In this case, FlagSet() is remembered, while FlagRelease() is not. Consider the application of FlagSet() to a flag with no waiting processes. If a process then invokes FlagWait() on that flag, it will not suspend, since it finds the flag set. If you apply FlagRelease() instead of FlagSet(), a following FlagWait() will suspend its calling process, since FlagRelease() does not set the flag.

When the simulator runs on a uniprocessor host and several flag operations are scheduled at the same time, non-deterministic behavior is possible. Since FlagSet() and FlagRelease() take effect at the point they are actually executed, a process that waits on a flag at the same simulation time the flag is released would depend on the order that the host actually executed the operations. The process will suspend if FlagWait() is executed after FlagRelease() and will not suspend if they are executed in the opposite order. We observe a similar effect if both FlagSet() and FlagRelease() execute at the same simulation time. Whether or not the flag is set will depend on the order of execution of the two operations.

3.4. CONDITIONS AND STATE VARIABLES

Conditions offer another way to synchronize the execution of processes and schedule activities. You assign a logical expression to each condition when it is created. When you later schedule an activity on a condition, the value of its associated expression determines when the activity takes place. If the expression is true, we say that the condition holds, and the activity takes place at the current simulation time. If the value of the expression is false, the activity is delayed in the condition's queue until the expression becomes true. Similarly, when a process waits on a condition, it continues if the condition holds and suspends otherwise. When a condition changes value from false to true, all waiting activities are released.

We call the variables used in a condition's logical expression *state variables.* They are special in that whenever one changes value, all conditions that use it are reevaluated. If an expression evaluates true and there are activities waiting, they are released.

3.4.1. State Variables

There are two types of state variables, those with integer values and those with floating point values. You must use the special operations provided below to access and change the values of state variables. The statements that change the value of a state variable have a side effect of causing all conditions that depend on that state variable to be reevaluated.

There is one pre-defined floating point state variable that always equals the current simulation time. SIMTIME is defined as a pointer that points to this state variable and is always available to the user. The state variable pointed to by SIMTIME is updated automatically whenever simulation time is advanced.

Operations:

*IVAR *NewIvar(ivname, i)*
*char *ivname;*
int i;

> This operation creates and returns a pointer to a new integer state variable. It assigns the state variable the name *ivname* and the initial value *i.*

*FVAR *NewFvar(fvname, x)*
*char *fvname;*
double x;

> This operation creates and returns a pointer to a new floating point state variable. It assigns the state variable the name *fvname* and the initial value *x.*

void Iset(ivptr, i)
*IVAR *ivptr;*
int i;

> This operation sets the value of the integer state variable pointed to by *ivptr* to *i*.

void Fset(fvptr, x)
*FVAR *fvptr;*
double x;

> This operation sets the value of the floating point state variable pointed to by *fvptr* to *x.* The user should not attempt to set the special state variable SIMTIME with this command.

int Ival(ivptr)
*IVAR *ivptr;*

> This operation returns the value of the integer state variable pointed to by *ivptr*.

double Fval(fvptr)
*FVAR *fvptr;*

> This operation returns the value of the floating point state variable pointed to by *fvptr*.

Examples:

iv = NewIvar("iv1", 0);

Creates a new integer state variable and sets its value to 0.

fv = NewFvar("fv1", 0.0);

Creates a new floating point state variable and sets its value to 0.0.

Fset(fv, 4.5);

Sets the floating point state variable pointed to by *fv* to 4.5.

Iset(iv, Ival(iv)+3);

Increments the integer state variable pointed to by *iv* by 3.

3.4.2. Conditions

You must use two statements to define a condition, one to define its expression, and one to name it and link it to its expression. The expression is specified as a C function that returns a logical value (actually an integer in C) and has the following form:

int *exname*(argptr, argsize)
char *argptr;
int argsize;
{ statements; return (*expr*); };

where *exname* is an arbitrary name you assign to the expression. *Expr* can be any expression that evaluates to true or false. The arguments *argptr* and *argsize* are optional. They are set when the condition is created or by the operation ConditionSetArg(). They are intended to be used for passing arguments to the condition's expression is much the same way as arguments are passed to the body of a process or an event.

Operations:

*CONDITION *NewCondition(cname, exname, sv1, ..., svn, NULL, argptr, argsize)*
*char *cname;*
cond exname;
*STVAR *sv1;*
. . .
*STVAR *svn*
char argptr;
int argsize;

This operation creates and returns a pointer to a new condition. The first argument *cname* names the condition. The second argument *exname* links the condition to its expression. *Cond* is a typedef for the expression function described above. That is, it is the name of an integer valued function that returns the value of an expression involving state variables. The arguments following the first two are pointers to the state variables used in the expression *exname*. The list of state variables must be terminated with a null pointer. You must list pointers to all state variables that are used in the expression as arguments to CONDITION, and you must end with the argument NULL. This is required so that the condition can establish links with its state variables. The last two arguments are passed to the expression that defines the condition every time that expression is reevaluated. For an expression named *exname* on *n* state variables and a condition named *condname*, this operation would be called in the following way:

NewCondition(condname, exname, sv1, sv2, ... , svn, NULL, aptr, asize)

where *sv1, sv2, ... , svn* are pointers to the state variables used in the expression *exname*. Note that the state variable names and the expression name must be visible to this operation. The easiest way to guarantee this is to declare them as global variables. Also, all state variables referenced in the NewCondition() operation must be created prior to the execution of this operation.

void ConditionWait(cptr)
*CONDITION *cptr;*

If the condition pointed to by *cptr* holds when this operation is executed, the calling process continues. If it does not hold, then the calling process suspends and enters the tail of the condition's queue. You can only invoke this operation within the body of a process.

int ConditionState(cptr)
*CONDITION *cptr;*

This operation returns the status of the condition pointed to by *cptr*. There are two possible values: FALSE (i.e., 0) and TRUE (i.e., 1).

int ConditionWaiting(cptr)
*CONDITION *cptr;*

This operation returns the number of activities in the queue of the condition pointed to by *cptr*.

void ConditionSetArg(cptr, aptr, asize)
*CONDITION *cptr;*
*char *aptr;*
int asize;

This saves the pointer *aptr* and the integer *asize* in the descriptor of the condition pointed to by *cptr*. They are passed to the condition's expression every time it is evaluated. These arguments can also be set when the NewCondition() is called. ConditionSetArg() provides a way of changing the arguments after the condition has been created. Note that this operation only passes a pointer to the condition, not a value. Therefore, when the condition's expression accesses the argument pointed to by this pointer, it will get the value of the argument at the time it is accessed, and this may not be the same as its value when the argument was set.

Examples:

```
int contest(nullptr, i)
char *nullptr;
int i;
{ return (Ival(ivptr[i]) != 0); };
cptr1 = NewCondition("con", contest, ivptr, NULL, NULL, 4);
```

The condition *con* will hold if and only if the state variable pointed to by *ivptr[i]* is not equal to zero. The first argument to contest is not used, but the second is used to select one state variable from an array of state variables.

```
int expr1(){ return ((Fval(x) > Fval(y)) && (Fval(y) != 2.43)); };
cptr2 = NewCondition("C1", expr1, x, y, NULL, NULL, 0);
```

The condition *C1* will hold if and only if the state variable pointed to by *x* is greater than the state variable pointed to by *y*, and the state variable pointed to by *y* is not equal to 2.43. A NULL argptr and arsize of 0 are passed to NewCondition(), but not used by contest().

Iset(ivptr, 1);
ConditionWait(cptr1)

If *ivptr* and *cptr1* are as defined above, then the process invoking ConditionWait() will not suspend because the condition holds. If we had set the state variable pointed to by *ivptr* to 0, the process would suspend when it invoked ConditionWait().

i = CondStatus(condptr);

This statement sets the integer *i* to the status of the condition pointed to by *condptr*.

Comments:

The reason for declaring the expression separately from the condition is to avoid a preprocessing step that would be needed to parse the expression. Implementing it this way, the C compiler does the parsing.

The last two arguments to the operation NewCondition() can probably to omitted if they are not used within the condition's expression. NewCondition() will look for them on the stack, save whatever it finds there in its descriptor, and pass them to the condition's expression every time the condition is evaluated. If they are omitted, NewCondition will load garbage in the descriptor and pass it to the expression. However, if the expression does not use or even declare its arguments, no harm is done. This would mean that the arguments passed to NewCondition do not match its declared arguments, and there is some chance that an attempt to access them would result in a segmentation fault. Probably the safest thing to do is to always pass the last two arguments to NewCondition whether they are used by the expression or not.

Conditions are reevaluated each time one of their state variables changes value. This introduces a form of non-deterministic behavior. Indeed, if several statements executing at the same simulation time can alter the state of a condition, the behavior of the simulation could depend on the order that the simulator actually executes the statements. This situation can only happen if a state variable experiences several changes at the same simulation time, and simulations that do this are probably inherently non-deterministic. Therefore, the simulator should probably issue a warning message when this happens, since it is likely to be a mistake in the simulation program. This is not yet implemented in the current version of YACSIM.

3.5. RESOURCES

A resource consists of a queue and a set of servers. The only thing that an activity can do with a resource is to request service time. When an activity makes a request, the resource assigns it to a free server, if one is available, and puts it in the queue otherwise. A resource makes no distinction among its different servers and simply picks the first free one it can find to satisfy a request for service. The activity is in a suspended state when it requests service and does not wake up until its request has been fulfilled. When a server finishes serving an activity, it looks for another activity in the queue to serve. If there are none it returns to a pool of free servers and waits for another activity to serve. The set of rules used to decide how the resource puts activities in its queue and takes them out for service is called the *queuing discipline* for the resource. Most of the standard queuing disciplines have been implemented. They are described later in this section.

Operations:

*RESOURCE *NewResource(rname, qdisc, nserv, slice)*
*char *rname;*
int qdisc;
int nserv;
double slice;

> The operation creates and returns a pointer to a resource. The argument *qname* is the name assigned to the resource. The argument *qdisc* is a integer that specifies the queuing discipline for the resource to use. The choices for this argument are described in the following section. *Nserv* is the number of servers for the resource. The last argument *slice* is only used with the "round robin" and "round robin preemptive resume with priority" queuing disciplines and is ignored for all others. It is explained in the following section on queuing disciplines.

void ResourceUse(rptr, timeinc)
*RESOURCE *rptr;*
double timeinc;

> This is the operation a process uses to request service from the resource pointed to by *rptr*. The argument *timeinc* is the amount of service time requested. The operation can only be invoked from within a process body.

int ResourceWaiting(rptr)
*RESOURCE *rptr;*

> This operation returns the number of activities in the queue of the resource pointed to by *rptr*.

int ResourceServicing(rptr)
*RESOURCE *rptr;*

> This operation returns the number of activities that are currently receiving service from the resource pointed to by *rptr*.

Queuing Disciplines:

Ten different queuing disciplines are implemented. Currently, there is no way for the user to add new ones, although that capability may be added later. The user specifies the queuing discipline for a resource by passing an integer code to the operation NewResource(). Abbreviated names have been defined for all the implemented disciplines, and these can be used in place of the integer code to improve readability. These abbreviated names are listed in bold after the full names of the disciplines in the following descriptions.

First Come First Served - **FCFS**

> The resource inserts processes into its queue at the tail and removes them from the head. Once the resource assigns a process to a server, that process receives all of its requested service without interruption.

Last Come First Served **- LCFS**

> The resource inserts processes into its queue at the head and removes them from the head. Once the resource assigns a process to a server, that process receives all of its requested service without interruption.

Last Come First Served Preemptive Resume - **LCFSPR**

This discipline will only use one server no matter how many are specified. When the resource receives a request and its server is busy, it preempts the process receiving service and lowers its requested time by the amount of service time already received. The resource then puts the preempted process at the head of its queue and assigns the new requesting process to the server. When the server finishes with one process it takes the process from the head of the queue, if any, and proceeds to complete its service time.

First Come First Served Preemptive Resume With Priorities - ***FCFSPRWP***

This discipline will only use one server no matter how many are specified. Users can assign priorities to processes using ProcessSetPriority(). This discipline uses a process' priority to determine where it should be inserted in the queue. When a resource receives a request and its server is busy, it reacts in the following way:

- If the priority of the process being served is greater than or equal to the priority of the new process requesting service, the resource inserts the new process into the queue in the order of its priority and behind all other processes that have the same priority.
- If the process being served has lower priority, the resource preempts it and lowers its requested time by the amount of service time already received. The resource then inserts the preempted process in the queue in the order of its priority and in front of all other processes that have the same priority. Finally, the resource assigns the new requesting process to the server.

When the server finishes with one process it takes the process from the head of the queue, if any, and proceeds to complete its service time.

Last Come First Served Preemptive Resume With Priorities - **LCFSPRWP**

This discipline is the same as the FCFSPRWP discipline described above, except that a process requesting service is inserted in the queue in the order of its priority, in front of all processes with the same priority instead of behind them.

Processor Sharing - **PROCSHAR**

This discipline uses as many servers as it needs to service all request without delay. That is, all processes start receiving service as soon as they request it and they are never delayed in a queue. However, when there are k processes receiving service the remaining service time for each is increased by a factor of k. As processes arrive at and leave the resource, the requested service time of each of the remaining processes is altered to account for the new value of k.

Round Robin - **RR**

This queuing discipline is one of the two that use the *slice* argument. Processes requesting service are put at the tail of the queue. Whenever there is a free server, the resource takes the process from the head of the queue and assigns it to that server for a service time equal to the slice time. When a server completes a time slice the resource reduces the process' requested service time by the slice amount and puts it back at the tail of its queue. If the process taken from the head of the queue has a remaining service request less than the slice time, the resource only assigns it to a server for its remaining requested time.

Random - **RAND**

The resource inserts processes requesting service randomly into its queue. When a server is free, it starts serving the process at the head of the queue.

Shortest Job Next - **SJN**

This discipline will only use one server no matter how many are specified. Whenever a job finishes service, the shortest job (i.e., the one with the smallest service time request) will be selected next to receive service from the resource.

Round Robin Preemptive Resume With Priority - **RRPRWP**

This discipline will only use one server no matter how many are specified. It uses the round robin queuing discipline to service processes in its queue until a service request is received. It uses the *slice* argument in the same way as the round robin queuing discipline. When a process requests service from the resource, its priority is compared to the priority of the process currently receiving service. If the new processes' priority is higher, it preempts the one receiving service. Otherwise it enters the resource's queue in the order of its priority, behind all other processes that have the same priority.

Examples

rsptr1 = NewResource("Res 2", FCFS, 3, 0.0)

This statement sets the pointer *rsptr1* to a new resource named "Res 2" that has three servers and uses the First Come First Served queuing discipline. The slice time is ignored.

rsptr2 = NewResource("Res 1", LCFSPRWP, 1, 0.0)

This declaration creates a resource, names it "Res 1", and sets its queuing discipline to Last Come First Served Preemptive Resume With Priority.

resptr3 = NewResource("RRrsc", RR, 9, 20.0)

This resource uses the Round Robin queuing discipline, has 9 servers, a service slice of 20 time units, and is named "RRrsc."

resptr4 = NewResource("PR", LCFSPR, 3, 5.0);

This statement creates a resource named "PR" that uses the Last Come First Served Preemptive Resume queuing discipline. Note that even though the declaration specifies 3 servers and a time slice of 5.0, these arguments will be ignored, because this discipline will only use one server and does not need a time slice.

ResourceUse(rptr, 25.0);

This statement requests 25.0 time units of service from the resource pointed to by *rptr*. The statement can only be executed from within the body of a process.

Comments

The effect of the operation ResourceUse() on a uniprocessor host is delayed until all activities scheduled at the same time have been initiated. This means that, for the preemptive disciplines, if ResourceUse() is invoked more than once at the same simulation time, it will be the process with highest priority that calls ResourceUse() most recently in real time that gets service.

3.6. QUEUE STATISTICS

Statistics can be automatically collected on queues. The following operations are used to activate this feature and to access the collected statistics. Note that when a queue is deleted its associated statistics record is also deleted.

Operations:

void QueueCollectStats(qptr, type, meanflg, histflg, nbin, low, high)
*RESOURCE *qptr;*
int type;
int meanflg;
int histflg;
int nbin;
double low;
double high;

This operation activates statistics collection for the queue pointed to by *qptr*. It does this by creating a statistics record (see Chapter 4) that is updated automatically whenever the queue's status is changed. Statistics can be collected on the following three parameters: queue length, time in the queue, and server utilization (for resources only). The argument *type* specifies which of the three to activate and must have one of the following three values: LENGTH, TIME, or UTIL. The length of a queue is the number of activities in it, including those being served if the queue is a resource. The time in the queue statistic is a measure of how long an activity spends in the queue, including time in the server for resource queues. Utilization statistics can only be collected on resources. They measure how many of the servers are busy over time. Statistics records of type LENGTH and UTIL are interval statistics records, and ones of type TIME are point statistics records. The last five arguments characterize the statistics record and are the same as for all statistics records. These arguments and the difference between point and interval statistics records are explained in the chapter on statistics records.

void QueueResetStats(qptr)
*RESOURCE *qptr;*

This operation resets all of the statistics records associated with the queue pointed to by *qptr* that have been previously activated. This reset operation should be used instead of StatrecReset(), or the initial sample point for the statistics records will missed.

*STATREC *QueueStatPtr(qptr, type)*
*RESOURCE *qptr;*
int type;

This operation returns a pointer to a statistics record associated with the queue pointed to by *qptr*. Chapter 4 describes the various operations (e.g., printing a report) that can be performed on a statistics record once you have a pointer to it. *Type* must be either LENGTH, TIME, or UTIL (for resources only) to select one of the three statistics records associated with the queue.

4. STATISTICS RECORDS & RANDOM NUMBERS

Statistics records are the simulation objects used to collect and process data produced by a simulation. They operate on sequences of weighted numbers called *samples*. Several statistics plus histograms can be computed on these sequences.

Implementation of the Unix random number generators vary from system to system, so that a simulation driven by a random number generator can give different results on different systems. To avoid this, YACSIM provides a random number generator that generates the same sequences, given the same seed, for all systems on which YACSIM has been ported.

4.1. POINT AND INTERVAL STATISTICS RECORDS

There are two types of statistics records: *point statistics records* and *interval statistics records*. They have the same set of operations and features. The only difference between them is in the way they define weights. Samples for a point statistics record are ordered pairs (x,y) where x is a value and y is the weight. Samples for interval statistics records are ordered pairs (x,y) where x is the value and y is used to define the weight in the following way. The weight for value x_i is defined as $w_i = (y_{i+1} - y_i)$ where (x_i,y_i) denotes the *i-th* sample in the sequence. In the remaining discussion of statistics records, we will not make a distinction between point and interval statistics records. We will describe their action on pairs of the form (v,w) where v is a value and w is a weight. For point statistics records, w is the second element of the sample pair. For interval statistics records, the weight w of a sample is obtained from intervals as described above.

For a sequence of weighted values $(v_0,w_0), (v_1,w_1), \ldots , (v_{n-1},w_{n-1})$, statistics records can calculate the following statistics:

Samples The number of samples.

$$\textbf{Samples} = n$$

Max The maximum unweighted value in the sequence of samples.

$$\textbf{Max} = \max\{ v_i \mid 0 \le i < n \}$$

Min The minimum unweighted value in the sequence of samples.

$$\textbf{Min} = \min\{ v_i \mid 0 \le i < n \}$$

Interval The sampling interval. See the description of operations EndInterval() and Interval() below for a definition of this interval.

Rate The average sampling rate.

$$\textbf{Rate} = \textbf{Samples/Interval}$$

Mean The weighted mean.

$$\textbf{Mean} = \frac{\sum_{i=0}^{n-1} v_i \times w_i}{\textbf{Samples}}$$

StdDev The weighted standard deviation.

$$\textbf{StdDev} = \sqrt{\frac{\sum_{i=0}^{n-1} w_i (v_i - \bar{v})^2}{\left(\sum_{i=0}^{n-1} w_i\right) - 1}}$$

A statistics record can also accumulate a histogram of its input sequence. The user specifies a number of bins *n* and low and high values. The histogram will have *n* bins of equal size between the low and high values. It will have two additional bins, one for all values less than the low value and the other for all values greater than or equal to the high value. For each sample *(v,w)*, the bin that corresponds to *v* is incremented by *w*.

Operations:

*STATREC *NewStatrec(sname, type, meanflg, histflg, nbins, lowbin, highbin)*
int type;
*char *sname;*
int meanflg;
int histflg;
int nbins;
double lowbin;
double highbin;

> This operation creates a statistics record with name *sname*. *Type* is either POINT or INTERVAL to specify whether the statistics record will be a point or an interval statistics record. It will always compute the statistics **Samples**, **Max**, **Min**, **Interval**, and **Rate**. *Meanflg* indicates whether or not this statistics record will compute the mean and standard deviation. There are two possible values. MEANS indicates that the mean and standard deviation are computed, and NOMEANS suppresses their computation. *Histflg* indicates whether or not histograms will be computed. Its two possible values are NOHIST and HIST. NOHIST turns off the collection of histogram data. HIST activates the collection of a full histogram with *nbin* bins between *lowbin* and *highbin*, plus two overflow bins as described above.

void StatrecSetHistSz(sz)
int sz;

> When a new statistics record is created with histogram collection activated, a block of memory is allocated to accumulate the histogram data. The size of this block must be big enough to hold all of the bins of the histogram. To minimize calls to malloc(), YACSIM maintains a pool of histogram memory blocks and does its own allocation whenever possible. There is a fixed default size for these blocks which is used whenever a histogram of that size or less is needed. If a histogram larger than the default size is required, malloc() is called to allocate its memory. The operation StatrecSetHistSz() can be used to change the default histogram size to *sz*. It must be used before the first call to NewStatrec(). Otherwise, the default size is left unchanged and a warning message is generated.

void StatrecReset(srptr)
STATREC srptr;

This operation resets the statistics record pointed to by *srptr* to the state it was in when it was created. It also sets the start of the sampling interval to the value of simulation time when it is executed. This operation should not be used to reset any of the three statistics records associated with queues. Use QueueResetStats() instead.

void StatrecUpdate(srptr, x, y)
*STATREC *srptr;*
double x;
double y;

This is the operation used to add another sample to the statistics record pointed to by *srptr*. *X* is the value of the sample. If the record is a point statistics record, *y* is the weight of the sample. If it is an interval statistics record, *y* is used to define an interval weight as explained above.

void StatrecReport(srptr)
*STATREC *srptr;*

This operation prints all of the statistics computed by the statistics record pointed to by *srptr* in a standard format. It also prints a graphical representation of the histogram on the user's display, if one was computed by the statistics record.

int StatrecBins(srptr)
*STATREC *srptr;*

double StatrecLowBin(srptr)
*STATREC *srptr;*

double StatrecHighBin(srptr)
*STATREC *srptr;*

double StatrecBinSize(srptr)
*STATREC *srptr;*

double StatrecHist(srptr, i)
*STATREC *srptr;*
int i;

These five functions access the histogram data of the statistics record pointed to by *srptr*. StatrecBins() returns the number of bins, not counting the two overflow bins. StatrecLowBin() returns the low limit of the bin immediately above the low overflow bin. StatrecHighBin() returns the low limit of the high overflow bin. StatrecBinSize() returns the bin size. StatrecHist(srptr,i) returns the value of the i-th bin of the histogram pointed to by *srptr*.

int StatrecSamples(srptr)
*STATREC *srptr;*

double StatrecMinVal(srptr)
*STATREC *srptr;*

double StatrecMaxVal(srptr)
*STATREC *srptr;*

double StatrecMean(srptr)
*STATREC *srptr;*

double StatrecSdv(srptr)
*STATREC *srptr;*

These five operations return statistics computed by the statistics record. StatrecSamples() returns the statistic **Samples**. StatrecMinVal() and StatrecMaxVal() return the statistics **Min** and **Max.** StatrecMean() returns **Mean** and StatrecSdv() returns **StdDev**.

void StatrecEndInterval(srptr)
*STATREC *srptr;*

double StatrecInterval(srptr)
*STATREC *srptr;*

double StatrecRate(srptr)
*STATREC *srptr;*

Creating a new statistics record sets the beginning time of the sampling interval to the value of simulation time when it is created. If you want to start the sampling interval at a later time, you can use the operation StatrecReset() described previously. The ending time of the sampling interval is initially the same as the starting time and is reset to the current simulation time each time a new sample is entered. This defines the sampling interval to be the period from the time the statistics record is created, or the last time it was reset, to the time of the last sample. The operation StatrecEndInterval() sets the ending time of the interval to the value of simulation time when it is executed. This enables the user to end the sampling interval at someplace other than a sampling point. StatrecInterval() returns the statistic **Interval** and StatrecRate() returns the statistic **Rate**.

Examples

srptr1 = NewStatrec("Stat1", POINT, NOMEANS, NOHIST, 0, 0.0, 0.0);

This creates a point statistics record named "Stat1" that will only compute the statistics Samples, Max, Min, Interval, and Rate. The last three arguments are ignored.

srptr2 = NewStatrec("Stat2", INTERVAL, MEANS, NOHIST, 0, 0.0, 0.0);

This creates an interval statistics record named "Stat2." It will compute all seven statistics, but no histogram. The last three arguments are ignored.

srptr3 = NewStatrec("Stat3", POINT, MEANS, HIST, 5, 1.0, 11.0);

The point statistics record created here will compute all seven statistics plus a histogram with five equally spaced bins between 1.0 and 11.0 and two overflow bins. The bin size will be 2.0.

StatrecUpdate(srptr1, 3.5, 1.0);

Since *srptr1* points to a point statistics record, this statement will add a sample with value 3.5 and weight 1.0.

StatrecUpdate(srptr3, 4.0, 25.6)

This adds a sample with value 4 and weight 25.6 to the point statistics record pointed to by *srptr3*.

StatrecUpdate(srptr2, 2.0, 1.0);
StatrecUpdate(srptr2, 3.0, 3.0);
StatrecUpdate(srptr2, 3.0, 4.5);
StatrecUpdate(srptr2, 1.5, 5.0);

This sequence of updates on the statistics record pointed to by *srptr2* will result in a sample with value 2.0 and weight 3.0 - 1.0 = 2.0, a sample with value 3.0 and weight 1.5, and a sample with value 3.0 and weight 0.5.

StatrecReset(srptr3); *(executed at time t1)*
StatrecEndInterval(srptr3); *(executed at time t2)*

These two operations will define the sampling interval to be from t1 to t2.

Comments:

If you invoke one of the operations that returns a statistic or histogram value and that statistic or histogram is not computed by the statistics record, then the simulator will print a warning message. If histograms are computed, but the values are all zero, StatrecReport() will print a warning message instead of the histogram graph. If a negative weight is computed for an interval statistic due to a later value having an earlier time, StatrecReport() will compute and print the statistics and the histogram, but will print the warning message *Invalid statistics; interval error*.

4.2. RANDOM NUMBER GENERATION

There is an internal YACSIM random number generator is called YacRand. It uses a multiplicative congruential random number generator suggested by Shedler in Lavenberg's "Computer Performance Modeling Handbook." It is the same generator used in the IBM System/360 (save for the difference in 360 and IEEE 754 floating point arithmetic). The generator is specified as

$$X_{n+1} = aX_n \bmod m,$$

where $a = 16807$ ($= 7^5$) and $m = 2147483647$ ($= 2^{31} - 1$)

A call to YacRand actually produces a uniformly distributed random number x, such that $0 \le x < 1$, by dividing the (integer) X_{n+1} by the modulus m and returning this value.

Operations:

double YacRand()

This operation is used to access the YACSIM random number generator. It generates and returns the next random number in the sequence.

void YacSeed(seed)
double seed;

This operation seeds the random number generator, which otherwise uses a seed of 0.5. In order to maintain a correspondence between YacRand and drand48, the seed value, which is a double, should be greater than 0 and less than 1. It is multiplied inside YacSeed by the modulus to produce an integer seed between 1 and the modulus. Calling YacSeed() with a seed of 0 causes it to use the default value of 0.5.

5. THE SIMULATION DRIVER

5.1. THE DRIVER

The driver is that part of a simulator that controls the sequencing of activities. It usually does this by means of a linked list called an *event list*. When a user schedules an activity to take place in the future, the driver inserts that activity into the event list. The driver also keeps the list ordered by inserting new activities in such a way that all activities scheduled to occur before the new activity are ahead of it, and all activities scheduled to occur after the new activity are behind it in the list. In other words, the event list is kept ordered by the scheduled times of its activities. The driver initiates activities in the order they appear in the list by always taking the activity at the head of the list to initiate next. Each time the driver removes an activity that has a scheduled time greater than the current simulation time, it increases simulation time to the time of that activity.

Operations:

void DriverReset()

> This operation resets simulation time to 0.0, clears the event list, clears all simulation queues, and destroys all existing simulation objects. That is, it takes the simulator back to the state it was in when it first started execution and before any objects (queues, activities, and statistics records) were created. You must recreate all objects after executing DriverReset() before you can run another simulation.

int DriverRun(timeinc)
double timeinc;

> This operation starts or restarts a simulation. Once the user invokes this operation, the simulator will run for *timeinc* units of simulation time, until its event list is empty, or until it is interrupted, and then return control to the user. The user can continue a simulation by invoking DriverRun() again with a new time increment *t*. The simulator will pick up where it left off and run for *t* more units of time. Invoking DriverReset() before another run will start the simulation over at time 0.0. If DriverRun() is invoked with *timeinc* less than or equal to 0.0, the simulator will run until the event list is empty or the simulation is interrupted. The return value is 0 if the simulation runs the full *timeinc* units of time or exhausts its event list. Due to the operation DriverInterrupt() described below, it is possible for the simulation to terminate before that time. In that case, DriverRun() will return a non-zero value as explained for the DriverInterrupt() operation.

void DriverInterrupt(retval)
int retval;

> This operation suspends a simulation run before the requested simulation time increment is completed. The user can invoke it from within an event or a process, and it will cause the interruption to occur when that activity terminates or suspends. The argument *retval* is the value returned by DriverRun(). A *retval* of 0 will generate an error termination of the simulation as that value is reserved to signify a simulation that has run to completion.

double GetSimTime()

This operation returns the current simulation time. You can invoke it from anywhere in the simulation code.

Examples:

DriverRun(10.0);

This statement transfers control to the driver for 10.0 units of simulation time.

DriverRun(0.0);

This statement also transfers control to the driver, but it will not return until the event queue is empty or an interrupt occurs.

i = DriverRun(100.0);

This statement transfers control to the driver for 100 units of simulation time. The return value i will be 0 if the simulation runs for the full 100 time steps or the event list is exhausted, and will be non-zero if it is interrupted.

DriverRun(5.5);
DriverRun(5.0);
DriverReset();
DriverRun(10.5);

The first statement will run the simulator for 5.5. units of simulation time. The second statement will then execute 5.0 more units of simulation time. The third statement resets the simulator and the fourth initiates a simulation run of 10.5 time units. The effect of this sequence is to perform two simulations of 10.5 units of simulation time each.

DriverInterrupt(5);

This interrupts the simulator and returns the value 5 to the user.

5.2. THE EVENT LIST

Recall that the activities on the event list are ordered by their scheduled time of occurrence. Each time a new activity is added to the event list, it is inserted in the proper order. If the event list is large, the insertion of activities can account for a significant amount of the time to perform a simulation. Therefore, several techniques for reducing the insertion time of event list have been developed. YACSIM uses one called a *calendar queue*.

Calendar queues break up the event list into bins in such a way that the proper bin for an activity can be quickly determined. The details of this approach can be found in the paper "Calendar Queues: A Fast O(1) Priority Queue Implementation for the Simulation Event Set Problem," *Communications of the ACM*, Vol. 31, No. 10, Oct. 1988, pp. 1220-1227. The calendar queue insertion algorithm includes a method of resizing the number and size of the bins as the queue grows and shrinks in order to optimize its performance. While this technique works well for most distributions of times if the queue is large, its performance can deteriorate for certain distributions and for simulations with small event lists. Therefore a method for the user to control the number of bins and their size has been implemented.

The parameters of the calendar queue bins can be specified with command line arguments to the simulation. If a simulation is invoked with the argument *+bi*, where $i \geq 1$, the simulator will use a calendar queue with exactly *i* bins. A simulation invoked with the argument *+wj*, where *j* is a floating point number, will use a calendar queue with all bins of size *j*. In both these cases the automatic resizing of the bins will be turned off. The argument *+l* or the special case *+b1* will cause the simulator to use a simple sequential search for insertion of new activities. This is a much simpler implementation than the calendar queue, and it can be faster if the event list never grows very large during a simulation. The *+bi*,*+wj* and *+l* arguments must precede any arguments that are to be passed to UserMain() as explained in Section 1.3. The operation EventListSelect() described below provides another way to specify the type and parameters of the event list.

Unfortunately, there does not appear to be any good way to predict in advance the optimal size for the bins, or whether or not fixed or variable bins will perform better. Several statistics described in the following section can be collected on the event list's behavior and may be helpful in choosing optimal bin parameters.

The default event list implementation is the calendar queue with automatic bin sizing activated.

Operations:

void EventListSelect(type, bins, bwidth)
int type;
int bins;
double bwidth;

> This operations provides the user with a means of selecting which type of event list implementation will be used. It has the same effect as the command line arguments *+bi*, *+wi*, and *+l*, but it can be called from with the user's program. *Type* is either CALQUE to specify the calendar queue implementation or LINQUE to specify the linear linked list implementation. If the type is CALQUE, then *bins* specifies a fixed number of bins and *bwidth* a fixed bin width. If *bins* is less than 2 and *bwidth* less than or equal to 0.0, the automatic bin sizing algorithm is used. **CAUTION: This operation must be called in UserMain() at the very beginning of a simulation and before any other simulation operations are called. No warning is generated if it is called later.**

int EventListSize()

> This operation returns the number of activities currently in the event list.

void EventListCollectStats(type, meanflg, histflg, nbin, low, high)
int type;
int meanflg;
int histflg;
int nbin;
double low;
double high;

> Calling this operation activates statistics collection for the event list. It does this by creating statistics records (see Chapter 5) that are updated automatically whenever there is any change in the event list. The following four different statistics can be collected as specified by the argument *type* .

type	Statistic Record Update Value
LENGTH	The size of the list (i.e., the number of activities in the list).
BINS	The number of bins in a calendar queue (ignored for the linear list)

BINWIDTH The size of a calendar queue's bins (ignored for the linear list)

EMPTYBINS The number of empty calendar queue bins (ignored for the linear list)

The last five arguments characterize the statistics record and are the same as for all statistics records (see Chapter 5). All four of the statistics records associated with the event list are point statistics records that are updated with a weight of 1.0 every time an activity is added to or deleted from the event list.

void EventListResetStats()

This operation resets all of the statistics records associated with the event list that have been previously activated. This reset operation should be used instead of StatrecReset().

*STATREC *EventListStatPtr(type)*
int type;

This operation returns a pointer to one of the statistics records associated with the event list. Chapter 4 describes the various operations, such as printing a summary report, that can be performed on a statistics records through its pointer. *Type* must be one of LENGTH, BINS, BINWIDTH, or EMPTYBINS to select one of the four statistics record types. If the selected type has not been activated, a NULL pointer is returned.

Examples:

EventListSelect(LINQUE, 0, 0.0);

This statement selects the linear linked list implementation of the event list. The last two arguments are ignored.

EventListSelect(CALQUE, 0, 0.0);

This statement selects the calendar queue implementation of the event list with automatic bin sizing.

EventListSelect(CALQUE, 4, 2.05);

This statement selects the calendar queue implementation of the event list with four bins of width 2.5.

EventListCollectStats(BINWIDTH, MEANS, HIST, 10, 0.0, 10.0);

This statement activates the collection of statistics on the bin widths and specifies that both means and histograms will be collected.

6. DEBUGGING

In addition to the standard C debugging tools, the YACSIM simulator provides a tracing capability and numerous error and warning messages. Although we have attempted to make the error and warning messages self-explanatory, it is difficult to provide much detail in a single line of text. Therefore, the next two sections list all of these messages and provide additional information on their causes. The difference between warning and error messages is that conditions that generate error messages will also terminated the simulation, while conditions that generate warning messages will not. The third section explains the tracing capability built into the simulators.

6.1. WARNING MESSAGES

All histogram entries = 0:

The statistics record operation StatrecReport() generates this warning when it attempts to print a histogram graph and finds all of the bin entries zero. It prints this warning instead of the graph.

Can only set the default histogram size before calling NewStatrec:

The operation StatrecSetHistSz() was called after one or more statistics records have been created. In this case, the default size for histograms is not changed.

Can only set the default stack size before calling NewProcess:

The operation ProcessSetStkSz() was called after one or more processes have been created. In this case, the default size for process stacks is not changed.

Invalid event list statistics type; statistics not collected:

EventListCollectStat() has been called with an invalid *type* argument. It must be either LENGTH, BINS, BINWIDTH, or EMPTYBINS. Statistics collection is not activated.

Invalid statistics; interval error:

The operation StatrecReport() of an interval statistics record generates this warning if it encounters a negative interval due to two samples where the later one has an earlier time value than the first one. Since this is usually an error, a warning is generated, but the report is still printed in case it is not.

Invalid statistic type for queues, can't collect:

QueueCollectStat() has been called with an invalid *type* argument. It must be either LENGTH, TIME, or UTIL. Statistics collection is not activated.

Invalid statistic type for queue:

QueueStatPtr() has been called with an invalid *type* argument. It must be either LENGTH, TIME, or UTIL. The operation returns the null pointer instead of a valid statistics record pointer.

Not all message bytes received:

The operation ProcessReceiveMsg() does not receive all of the characters in a message if the *bytes* argument is less than the message size. The remaining characters can be obtained with additional calls to ProcessReceiveMsg().

Preemptive Resume resources will only use one server:

The operation that creates resources generates this warning when a value different from 1 is entered for the number of servers and the queuing discipline is one of the preemptive resume disciplines. The resource will be created with only one server no matter how many are requested.

Process statistics collection already set:

This message is generated when the operation ProcessCollectStat() is applied to the same process more than once.

Processor sharing resources will only use one serer:

The operation that creates resources generates this warning when a value different from 1 is entered for the number of servers and the queuing discipline is processor sharing. The resource will be created with only one server no matter how many are requested.

Queue length statistics collection already set:

This message is generated when the operation ResourceCollectStat() activating the collection of queue length statisics is applied to the same resource more than once.

Queue time statistics collection already set:

This message is generated when the operation ResourceCollectStat() activating the collection of queue time statisics is applied to the same resource more than once.

Queue utilization statistics collection already set:

This message is generated when the operation ResourceCollectStat() activating the collection of queue utilization statistics is applied to the same resource more than once.

Setting a set flag:

The flag operation FlagSet() generates this warning when it is applied to a flag that is already set. A warning is given because two or more successive applications of FlagSet() are redundant.

Setting a set semaphore:

The semaphore operation SemaphoreSet() generates this warning when it is applied to a semaphore that is already set. A warning is given because two or more successive applications of SemaphoreSet() are redundant.

Setting an IVAR that is not linked to a condition :

This usually means that a condition has been created before one of the state variables it depends on has been created.

Setting an FVAR that is not linked to a condition:

This usually means that a condition has been created before one of the state variables it depends on has been created.

Statistics not collected; cannot print report:

The operation ActivityStatRept() was called for some activity in order to print a report of that activity's statistics, but the collection of statistics had not been previously activated.

Time slice only used with round robin queue discipline:

The operation that creates a new resource generates this warning when a time slice value other than the default value is entered for any resource other than Round Robin or Round Robin Preemptive Resume With Priorities.

Trying to receive a negative number of bytes:

The operation ProcessReceiveMsg() was called with a negative "bytes" argument. In this case the operation will not receive any of the messages bytes, but they are retained by the system and can be read by a subsequent call to ProcessReceiveMsg().

YacRand seed out of range; default seed used

The argument to YacSeed() must be > 0.0 and < 1.0. Otherwise the argument is ignored and the default seed value is used.

6.2. ERROR MESSAGES

ActivityGetParPtr() must be called from within an activity:

Since only activities can have a parent, ActivityGetParPtr() can only be called from within an activity's body.

Can not call DriverReset() from within a process or an event:

The driver can only be reset after returning from DriverRun(), that is, while the simulation is not active.

Can not delay for a negative time:

The operation ProcessDelay() generates this error message if it is called with a negative time increment.

Can not request negative service time from a resource:

An attempt to schedule (or reschedule) an activity for a negative amount of service time from a resource generates this message.

Can not reschedule a deleting event:

Since a deleting event will always be destroyed after it occurs, it makes no sense for it to reschedule itself.

Can not reschedule a pending event:

A pending event is one that has been scheduled, but has not yet occurred. An attempt to schedule such an event is an error.

Can not schedule an activity to occur in the past:

This error results from an attempt to schedule an activity to happen at a time prior to the current simulation time.

Changing Delete Flag of a scheduled event:

Once an event has been scheduled, you can not change its delete flag until after it occurs.

EventGetState() must be called from within an active event:

The state of an event can only be accessed by the user from within the body of that event.

EventReschedule() can only be invoked from within an event body:

EventReschedule() is provided only for an event to reschedule itself.

Events can not be scheduled in the past:

An attempt to schedule an event with a negative time increment generates this message.

EventSetDelFlag() not called from within an event:

The delete flag of an event can only be set from within the body of that event.

EventSetState() has NULL pointer, but not called from an event:

This operation must be called from within an event's body or the argurment must point to an event.

Fset only works with FVAR's:

Trying to apply the operation Fset() to an integer state variable will generate this message.

Histograms not calculated for this statistics record:

This error results when there is an attempt to access the histogram information in a statistics record that does not compute histograms.

Invalid statistics record type, use POINT or INTERVAL:

The type argument to the NewStatrec() operations must be either POINT or INTERVAL.

Invalid histogram flag, use HIST or NOHIST:

The histflg argument to the NewStatrec() operations must be either HIST or NOHIST.

Invalid statistics type passed to EventListStatPtr():

The type passed must be LENGTH, BINS, BINWIDTH, or EMPTYBINS.

Iset only works with IVAR's:

Trying to apply the operation Iset() to a floating point state variable will generate this message.

Malloc fails in ... :

This error message indicates that the dynamic memory allocation function has run out of memory.

Means not calculated for this statistics record:

The statistics record operation StatrecMean() generates this error if it is applied to a statistics record that was created with argument *meanflg* = NOMEANS.

Null Activity Referenced:

A null pointer has been passed to an operation that expects a pointer to an activity.

Null queue element pointer passed to ... :

A null pointer to a queue element has been passed to an operation.

Null queue pointer passed to ... :

A null pointer to a queue has been passed to an operation.

Only processes can perform forking or blocking schedules:

An attempt to execute one of the activity forking operations from any place except the body of a process generates this error.

Only processes can send blocking messages:

ProcessSendMsg() can only be called from within the body of a process it the *blkflg* argument is BLOCK.

Only processes can wait at barriers:

The operation BarrierSync() has been called from some place other than from within the body of a process.

Processes can only be scheduled once:

You can not schedule a process once it has been scheduled. You can create multiple processes with the same body and even the same name, but each of them can only be scheduled once.

ProcessCheckMsg() can not be invoked from within a process body:

An attempt to execute the operation ProcessCheckMsg() from someplace other than the body of a process generates this error.

ProcessDelay() can only be invoked from within a process body:

An attempt to execute the operation ProcessDelay() from any place except the body of a process generates this error.

ProcessJoin() can only be invoked from within a process body:

An attempt to execute the operation ProcessJoin() from any place except the body of a process generates this error.

ProcessReceiveMsg() can only be invoked from within a process body:

Calling the operation ProcessReceiveMsg() from anyplace but within the body of a process will generate this error message.

ProcessReceiveMsg() has a null receive buffer:

The operation ProcessReceiveMsg() was called with its "buf" argument NULL, but the message was not empty.

ProcessSendMsg() must be called from within an activity:

Only processes and events can call ProcessSendMsg().

ProcessSleep() can only be invoked from within a process body:

An attempt to execute the operation ProcessSleep() from any place except the body of a process generates this error.

ResourceUse() must be called from a processess' body:

Only Processes can call ResourceUse().

Returning unallocated object to pool:

There has been an attempt to return something to a pool that was not obtained from a pool.

Simulator interrupted with 0 stopflag:

This error occurs when the user invokes DriverInterrupt(i) with $i = 0$. Since i becomes the return value for DriverRun() and a return value of 0 indicates a normal termination, not an interrupt, the argument i must be non-zero.

Std. Dev. not calculated for this statistics record:

The statistics record operation StatrecSdv() generates this error if it is applied to a statistics record that was created with the argument *meanflg* = NOMEANS.

Unimplemented queuing discipline:

The first argument to NewResource() specifies the type of queuing discipline the resource will use. If that does not match with one of the implemented disciplines, this error is generated.

6.3. TRACING

YACSIM has extensive tracing capability. The user can turn it on and off from any place within a simulation and can set it to one of six levels to generate different amounts of trace data. The global variable **TraceLevel** controls the trace output. Setting **TraceLevel** to 0 turns all tracing off. Setting it to 1, 2, 3, 4, or 5 sets the tracing level to that value. Tracing level *i* contains all the information of level *j*, for $j < i$. Level 1 tracing only prints warning messages and the final simulation time at the end of the simulation. Level 2 gives a coarse trace of when the simulation program calls and returns from the driver and when it switches processes. Level 3 tracing gives much more information about the invocation of all simulation operations available to the user. Level 4 adds information about the status of each queue every time it changes. Level 5 tracing give very low level traces including event list operations.

In addition to altering the variable **TraceLevel**, the user can set the trace level at the beginning of a simulation with a command line argument of the form *+ti,* where $0 \leq i \leq 5$. This sets the trace level to *i*. In this way the trace level can be changed without recompiling the simulation program. The *+ti* argument must precede any command line arguments that are to be passed to UserMain(), as explained in Section 1.3.

Most of the trace output should be self-explanatory. Every time a simulation object is mentioned in the trace output, it is referred to by its name. This is the name the user assigns to a simulation object when it is created. This name is arbitrary and need not be unique. Each object in the trace is also referenced by a unique ID number assigned it automatically when it is created. The form of the reference is *name[ID]* where *name* is the object's name and *ID* is its unique ID number. This form can be changed with the global variable **TraceIDs**. If it is set to 0, all the ID numbers in the trace are forced to 0. passeing the command line argument *-i* has the same effect.

APPENDIX 1: DEFINED SYMBOLS

This appendix list the symbols defined in the file sim.h. They are used extensively as arguments to many of the YACSIM operations. They are described in the discussions of the operations that use them and are only listed here.

Types of Activity Scheduling:

INDEPENDENT	0
BLOCK	1
FORK	2

Event Characteristics:

DELETE	1
NODELETE	0

Message Parameters:

ANYTYPE	-1
ANYSENDER	0
NOBLOCK	0
BLOCK	1

Statistics Record Characteristics:

NOMEANS	0
MEANS	1
HIST	2
NOHIST	3
POINT	4
INTERVAL	5

Argument and Buffer Size:

UNKNOWN	-1
DEFAULTSTK	0

Types of Queue Disciplines:

FCFS	1
FCFSPRWP	2
LCFSPR	3
PROCSHAR	4
RR	5
RAND	6
LCFSPRWP	7
SJN	8
RRPRWP	9
LCFS	10

Queue Statistics:

TIME	1
UTIL	2
LENGTH	3
BINS	4
BINWIDTH	5
EMPTYBINS	6

Event List Types:

CALQUE	0
LINQUE	1

Miscellaneous

ME	0

APPENDIX 2: SUMMARY OF OPERATIONS

ACTIVITY Operations

```
ActivityArgSize(aptr)                          /* Returns the size of an argument                  */
ActivityCollectStats(aptr)                     /* Activates statistics collection for an activity  */
ActivityGetArg(aptr)                           /* Returns the argument pointer of an activity      */
ActivityGetMyPtr()                             /* Returns a pointer to the active activity         */
ActivityGetParPtr()                            /* Returns a pointer to the active activity's parent */
ActivitySchedCond(aptr, condptr, blkflg)       /* Schedules an activity to wait for a condition    */
ActivitySchedFlag(aptr, flgptr, blkflg)        /* Schedules an activity to wait for a flag         */
ActivitySchedRes(aptr, rptr, timeinc, blkflg)  /* Schedules an activity to wait use a resource     */
ActivitySchedSema(aptr, semptr, blkflg)        /* Schedules an activity to wait for a semaphore    */
ActivitySchedTime(aptr, timeinc, blkflg)       /* Schedules an activity to start in the future     */
ActivitySetArg(aptr,argptr,argsize)            /* Sets the argument pointer of an activity         */
ActivityStatPtr(aptr)                          /* Returns a pointer to an activity's stat record   */
ActivityStatRept(aptr)                         /* Prints a report of an activity's statistics      */
```

PROCESS Operations

```
NewProcess(pname, bodyname, stksz)                  /* Creates & return a pointer to a new sim. process  */
ProcessCheckMsg(type, sender)                       /* Checks for a message of a given type and sender   */
ProcessDelay(timeinc)                               /* Suspends the current process for a time period    */
ProcessJoin()                                       /* Suspend until all forked child activities terminate */
ProcessReceiveMsg(buf, bytes, blkflg, type, sender) /* Copies received data into buf and returns its size */
ProcessSendMsg(dest, buf, bytes, blkflg, type)      /* Sends a message to a process                      */
ProcessSetPriority(procptr, p)                      /* Sets the priority of a process                    */
ProcessSetStkSz(stksz)                              /* Set the default stack size                        */
ProcessSleep()                                      /* Suspends the current process for an indefinite time */
```

EVENT Operations

```
NewEvent(ename, bodyname, delflg, etype)  /* Creates and returns a pointer to a new event  */
EventGetDelFlag(eptr)                     /* Returns DELETE (1) or NODELETE (0)            */
EventGetState()                           /* Returns the state of an event                 */
EventGetType(eptr)                        /* Returns the events type                       */
EventReschedCond(condptr, stval)          /* Reschedules an event to wait for a condition  */
EventReschedFlag(flgptr, stval)           /* Reschedules an event to wait for a flag       */
EventReschedRes(resptr, timeinc, stval)   /* Reschedules an event to use a resource        */
EventReschedSema(semptr, stval)           /* Reschedules an event to wait for a semaphore  */
EventReschedTime(timeinc, stval)          /* Reschedules an event to occur in the future   */
EventSetDelFlag()                         /* Makes an event deleting                       */
EventSetState(stvat)                      /* Sets state used to designate a return point   */
EventSetType(eptr, etype)                 /* Sets the event's type                         */
```

QUEUE Operations

```
QueueCollectStats(qptr, type, meanflg, histflg, nbin, low, high)
                                          /* Initiates statistics collection for a queue   */
QueueResetStats(qptr)                     /* Resets statistics collectin for a queue       */
QueueStatPtr(qptr, type)                  /* Returns a pointer to a queue's statistics     */
```

SEMAPHORE Operations

```
NewSemaphore(sname, i)                    /* Creates & returns a pointer to a new semaphore  */
SemaphoreDecr(sptr)                       /* Decrement the sem. value & return the new value */
```

```
SemaphoreInit(sptr, i)                  /* If queue is empty its value is set to i              */
SemaphoreSet(sptr)                      /* Set sem. to 1 if empty, or release an activity       */
SemaphoreSignal(sptr)                   /* Signals the semaphore                                */
SemaphoreValue(sptr)                    /* Returns the value of the semaphore                   */
SemaphoreWait(sptr)                     /* Wait on a semaphore                                  */
SemaphoreWaiting(sptr)                  /* Returns the # of activities in the queue             */
```

BARRIER Operations

```
NewBarrier(bname, i)                    /* Creates and returns a pointer to a new barrier       */
BarrierInit(bptr, i)                    /* If a barrier's queue is empty, sets its value to i   */
BarrierNeeded(bptr)                     /* Returns # of additional syncs needed to free barrier */
BarrierSync(bptr)                       /* Waits at a barrier synchronization point             */
BarrierWaiting(bptr)                    /* Returns the # of processes waiting at the barrier    */
```

FLAG Operations

```
NewFlag(fname)                          /* Creates and returns a pointer to a new flag          */
FlagRelease(fptr)                       /* Releases activities waiting at a flag                */
FlagSet(fptr)                           /* Sets a flag                                          */
FlagWait(fptr)                          /* Waits for a flag to be set or released               */
FlagWaiting(fptr)                       /* Returns the # of activities in the queue             */
```

STVAR Operations

```
NewFvar(fvname, x)                      /* Creates and returns a pointer to a new Fvar          */
NewIvar(ivname, i)                      /* Creates and returns a pointer to a new Ivar          */
Fset(fvptr, x)                          /* Sets the value of an Fvar                            */
Fval(fvptr)                             /* Returns th value of an Fvar                          */
Iset(ivptr, i)                          /* Sets the value of an Ivar                            */
Ival(ivptr)                             /* Returns the value of an Ivar                         */
```

CONDITION Operations

```
NewCondition(cname, exname, sv1, ..., svn, NULL, argptr, argsize)
                                        /* Creates and returns a pointer to a new condition     */
ConditionSetArg(cptr, aptr, asize)      /* Sets the argument pointer of a condition             */
ConditionState(cptr)                    /* Returns the state of a condition                     */
ConditionWait(cptr)                     /* Waits until a condition holds                        */
ConditionWaiting(cptr)                  /* Returns the # of activities in the queue             */
```

RESOURCE Operations

```
NewResource(rname, qdisc, nserv, slice) /* Creates and returns a pointer to a new resource      */
ResourceServicing(rptr)                 /* Returns the # of activities getting service          */
ResourceUse(rptr, timeinc)              /* Requests service from a resource                     */
ResourceWaiting(rptr)                   /* Returns the # of processes in the queue              */
```

STATREC Operations

```
NewStatrec(sname, type, meanflg, histflg, nbins, lowbin, highbin)
                                        /* Creates and returns a pointer to a new statrec       */
StatrecBins(srptr)                      /* Returns the number of bins                           */
StatrecBinSize(srptr)                   /* Returns the bin size                                 */
StatrecEndInterval(srptr)               /* Terminates a sampling interval                       */
StatrecHighBin(srptr)                   /* Returns the high bin lower bound                     */
StatrecHist(srptr, i)                   /* Returns the value of the ith histogram element       */
StatrecInterval(srptr)                  /* Returns the sampling interval                        */
StatrecLowBin(srptr)                    /* Returns the low bin upper bound                      */
StatrecMaxVal(srptr)                    /* Returns the maximum sample value                     */
StatrecMean(srptr)                      /* Returns the mean                                     */
```

```
StatrecMinVal(srptr)                    /* Returns the minimum sample value                    */
StatrecRate(srptr)                      /* Returns the sampling rate                           */
StatrecReport(srptr)                    /* Generates and displays a statrec report             */
StatrecReset(srptr)                     /* Resets the statrec                                  */
StatrecSamples(srptr)                   /* Returns the number of samples                       */
StatrecSdv(srptr)                       /* Returns the standard deviation                      */
StatrecSetHistSz(sz)                    /* Set the default histogram size                      */
StatrecUpdate(srptr, x, y)              /* Updates the statrec                                 */
YacRand()                               /* Random number generator                             */
YacSeed(seed)                           /* Set the seed for yacrand                            */
```

DRIVER Operations

```
DriverInterrupt(retval)                 /* Interrupts the driver and returns retval to the user */
DriverReset()                           /* Resets the driver (Sets YS__Simtime to 0)           */
DriverRun(timeinc)                      /* Activates the simulation driver returns a value     */
EventListCollectStats(type, meanflg, histflg, nbin, low, high)   /* Activates automatic stats collection   */
EventListResetStats()                   /* Resest a statistics record of a queue               */
EventListSelect(type, bins, bwidth)     /* Selects the type of event list to use               */
EventListSize()                         /* Returns the event list size                         */
EventListStatPtr(type)                  /* Returns ptr to evlst's statrec                      */
GetSimTime()                            /* Returns the current simulation time                 */
```

APPENDIX 3: ALPHABETICAL OPERATION LIST