

PSIM3 REFERENCE MANUAL

Object-Oriented Discrete Event Simulation C++ Package

José M. Garrido

Department of Computer Science and Information Systems

January, 2008

College of Science and Mathematics
Kennesaw State University

© 2008, J. M. Garrido

Contents

1	Introduction	4
2	The Psim3 Library	4
2.1	The Simulation Time	5
2.2	Defining Active Objects	5
2.3	Running a Simulation	6
2.4	Features in Class <i>process</i>	7
2.4.1	Name of an Active Object	7
2.4.2	The Simulation Clock	8
2.4.3	Priority of an Active Object	8
2.4.4	States of an Active Object	9
2.5	Scheduling a Process	9
2.6	Suspending a Process	10
2.7	Interrupting a Process	10
2.8	Terminating a Process	11
3	The Queue Library	12
3.1	General Description	12
3.2	Features of Class <i>squeue</i>	12
3.3	Features of Class <i>pqueue</i>	14
4	The Resources	18
4.1	General Description	18
4.2	Relevant Features of the <i>res</i> Class	19
4.3	Features in Class <i>bin</i>	20
5	The <i>waitq</i> Class	21
6	The <i>condq</i> Class	23
7	Random Numbers	25
7.1	Class <i>randint</i>	25
7.2	Class <i>erand</i>	27
7.3	Class <i>normal</i>	28
7.4	Class <i>poisson</i>	29
7.5	Class <i>urand</i>	30

<i>Psim3 Reference Manual</i>	3
-------------------------------	---

8 The Simulation Package on the Web Site	31
8.1 Files Available	31
8.2 Brief Instructions for Compiling and Linking	31

1 Introduction

This report presents a brief reference manual for the Psim3 simulation library. The library, Psim3, is used to implement simulation models with the C++ programming language.

The Psim3 library is freely available for educational and research purposes, are copyrighted (©J Garrido), and can be downloaded with examples from the Psim Web page:

`science.kennesaw.edu/~jgarrido/psim.html`

2 The Psim3 Library

The Psim3 simulation library is a set of C++ classes that implements the mechanisms necessary to carry out simulation runs using the object-oriented approach with C++ and the Pthreads library. For constructing simulation models, these library classes provide the following facilities:

- The definition of active objects, which are instantiations of subclasses of class *process*.
- The starting of a simulation run that will execute for a predetermined period, called the simulation period.
- The generation of random numbers, each from specified probability distribution.

Every user program that implements a simulation model with the Psim3 library needs to include the header file, `proc.h` as follows:

```
#include "proc.h"
```

This gives the simulation model access to all the class specifications in the basic Psim3 library. The most important class in the library is class *process*. An active object in a simulation model is an object of a user-defined subclass of class *process*. Note that in Psim3, an active object is also referred to as a *process*.

2.1 The Simulation Time

Simulation time is treated as a dense number; the type chosen for these values are of type `double` in C++. In all implementations using Psim3, variables that refer to time need to be of type `double`. For example, to assign a variable `current_time` the time of the system clock, the following syntax is used:

```
double current_time;
...
current_time = get_clock(); // time of system clock
```

2.2 Defining Active Objects

A simulation model in Psim3 must define one or more subclasses of class of the Psim3 class *process*. The active objects are then created as instances of these subclasses. To define an active object in a simulation model implemented in C++, the program must define a class with an appropriate name and inherit the *process* class from the Psim3 library. For example:

```
class server: public process {
    // private features
    ...
public:
    server( char *serv_name);    // constructor
    Main_body(void);
    // other public features
};
```

The code definition in a simulation model defines a subclass, called *server*, that inherits the *process* class from the Psim3 library. The public qualifier used to inherit the library class allows access to all the features of the *process* class. As any other class, class *server* includes definitions of its private features (data members and member functions) and its public features. All instances of the defined class *server* will behave as active objects.

Two of the public features defined above are very important and must be included in every class that represents an active object in the simulation model:

1. The constructor (*server*), which is an operation that is executed when an object of the class is created. The only parameter required is a character string for the name of the active object.

2. The function, *Main_body*, which is present in every subclass of class *process* in the simulation model. This function defines all the activities that are to be carried out by the active objects of this subclass. The name of this function (*Main_body*) is pre-defined for all subclasses of class *process*.

2.3 Running a Simulation

To define a simulation, a C++ program that implements a simulation model needs to declare and create an object of class *simulation* in its main function. The instantiation of this class requires a title for the simulation model. This is another Psim3 library class that a model needs to access. The following is an example of an instantiation of the simulation class:

```
simulation srun("Simple Batch System");
```

The C++ statement declares and creates an object called *srun*. The object created includes the title of the model, or the object can be used as a title for the simulation run (i.e., every simulation run can have a different title).

To start a simulation run and to let it execute for a specified period, the member function, *start_sim*, must be invoked from the object of class *simulation* created above. For example:

```
double simperiod = 450.75;
...
srun.start_sim(simperiod);
```

The first line declares a variable, *simperiod*, of type *double*. This variable represents the simulation period for the simulation run. Recall that all variables and objects that refer to the time dimension must be declared with this type. The *simperiod* variable is used in the second C++ statement above as a parameter when invoking the *start_sim* member function that belongs to *srun* (which is an object of the simulation class). In this case, the simulation runs for 450.75 time units.

Instead of an object of class *simulation*, a pointer of class *simulation* can be declared and initialized by creating an object of class *simulation*. For example:

```
simulation *srun;
. . .
```

```
srun = new simulation("Simple Batch System");  
...  
srun->start_sim(simperiod);
```

After the simulation run completes, the C++ program normally computes the summary statistics as final results and displays these results.

2.4 Features in Class *process*

When a user defines a subclass of class *process* in a simulation model, the subclass will have access to all the features inherited from the Psim3 *process* class. These features are specified in the `proc.h` header file. Some of these features are inherited from upper-level base classes. An active object is an instantiation of a user-defined subclass of the Psim3 class *process*.

Only the public features are discussed here. The following features are briefly discussed with examples. The user should refer to the `proc.h` header file.

2.4.1 Name of an Active Object

The name of an active object is assigned when this object is created using the constructor. The only parameter required in the constructor is the name. The *t_name* data member is a pointer to a character string with the name of the active object. For example, consider class *server*, a subclass of the Psim3 class *process*, and the following code displays the name of an active object of class *server*:

```
server * mach1;  
...  
mach1 = new server("machine_A"); // create new server  
...  
cout << "Name of process: " << mach1->t_name << endl;
```

In the code, *mach1* is the reference to an object of type *server*. A new object is created and assigned to this reference. In the last line, the name of the object (referenced by *mach1*) is displayed on the screen. To display the name of an active object, use *t_name* as follows:

```
cout << "Name of this active object: " << t_name << "\n";
```

2.4.2 The Simulation Clock

As stated above, the time dimension uses type `double` in C++. The simulation clock has the current time of the simulation. To access the simulation clock, the `get_clock` Psim3 library function must be used. For example:

```
double arrival_time;
...
arrival_time = get_clock();
```

The `arrival_time` variable is declared of type `double`, and it gets the value returned by the function `get_clock`.

2.4.3 Priority of an Active Object

The priority of an active object is an integer value and represents its relative importance in relation to other processes. The highest priority value is 0. To access the priority of an active process, the member function `get_prio` must be invoked. For example, to access the priority of another active object, invoke the function in the following manner:

```
server * mach1;
...
int s_prio;
s_prio = mach1->get_prio();
```

The first line declares a pointer to an active object of class `server`. The next line declares an integer variable to store the priority of the object. The third line invokes the `get_prio` member function in the object referenced by `mach1`, and assigns it to the integer variable `s_prio`.

To set the priority of an active object, the object must invoke the `set_prio` member function with an integer parameter. For example, suppose an active object needs to set its priority to 1. The following C++ statement sets the active object's priority to 1:

```
set_prio(1);
```

Now suppose that an active object needs to increase its priority by 1. The following statements accomplish this:

```
int my_priority;
...
my_priority = get_prio(); // get current priority
my_priority++;           // increment current priority
set_prio(my_priority);   // set new priority
```

2.4.4 States of an Active Object

There are three predefined states for an active object: IDLE, RUNNING, and TERMINATED. The state of an active object cannot be directly set since the simulation executive directly controls the scheduling of active objects. However, the state of an active object can be accessed by the *rdstate* member function. When invoked, this function returns any of the previously mentioned values. The function is normally called (or invoked) to check the state of an active object (process). For example, the following statement checks if active object *mach1* is in the RUNNING state:

```
server * mach1;
if ( mach1->rdstate() == RUNNING) {
    // carry out some activities if true
    ...
}
```

Note that in Psim3, an active object is also referred to as a process. To facilitate the implementation of processes, another member function, *idle*, is also provided in the Psim3 library. This function tests if the process is idle and returns true or false. The easiest way to check if a process is idle is:

```
if (! mach1->idle()) {
    // activities if process is not idle
    ...
}
```

Another similar member function provided in the Psim3 library is the function *pending*. This function returns true if the process is not terminated, otherwise it returns false.

2.5 Scheduling a Process

The dynamic behavior of a system is dependent on the scheduling of processes during an observation period, or the simulation period. To schedule a process means to place it in an event list with a time stamp so that the simulation executive will execute its appropriate phase when the system clock reaches that time.

The *delay* member function is used for this type of scheduling in Psim3. For example, suppose a process is to be scheduled 5.65 time units from the current clock time. The following line of code implements this scheduling:

```
delay(5.65);
```

The delay member function will normally place the process in the event list to be executed at 5.65 time units from the current simulation time. The simulation executive will resume execution of this process after 5.65 time units from the current clock time.

The delay operation is applied to simulate an activity that has a certain time duration, or to simulate a wait activity. This function is also used to reactivate a process that has been suspended. For example, assume that a process referenced by *mach1* has been suspended; another process can reactivate it using a delay with a value of 0.0. For example:

```
server *mach1;  
...  
mach1->delay(0.0); // reactivate barber
```

Process *mach1* of type *server* will now be reactivated, which means schedule the process now. This process will change its state from IDLE to RUNNING.

2.6 Suspending a Process

When a process has to suspend itself or another process, it invokes the *suspend* member function. This function takes the process away from the event list (if it has been scheduled) and changes its state from RUNNING to IDLE. In many practical cases, a suspended process is placed in a queue. For example, the following C++ code suspends the process *mach1* (defined above):

```
mach1->suspend(); // suspend barber
```

In Psim3 there are two other member functions of class *process* that can also suspend a process: *deactivate* and *sleep*. Refer to the header file `proc.h`.

2.7 Interrupting a Process

A process can interrupt another process that is carrying out some activity. The interrupting process uses an interrupt level (an integer). The interrupted process senses the interrupt and examines the *interrupted* data member, which has the value of the interrupt level. The function member

p_interrupt interrupts the process and returns the remaining time left for the current interrupted activity. The following C++ code shows how to interrupt process barber with interrupt level 2.

```
int int_lev = 2; // interrupt level
double rem_time;
rem_time = mach1->p_interrupt(int_lev);
```

As before, the type for the remaining time variable, *rem_time*, is `double`. This value may be useful to control the interval that the interrupted process will continue to execute when it is allowed to resume its activities.

The interrupted process will decide what instructions to execute when it receives the interrupt; the decision is based on the value of the interrupt level by invoking the *int_level* member function. The C++ code that follows implements the interrupt activities for the interrupted process.

```
int int_lev; // interrupt level
...
delay(act_period);
int_lev = int_level(); // get the interrupt level
if (interrupted == 2) { // interrupted with level 2
    int_time = get_clock(); // carry out some task
    . . .
}
else { // otherwise
    ... // carry out another task
}
```

The value of 0 for the *interrupted* data member means that there was no interruption during the activities of the process, which has a normal duration period, *act_period*.

2.8 Terminating a Process

A process can terminate itself or terminate another process. The function member *terminate* is invoked to normally terminate a process. In the following line of code, a process terminates itself.

```
terminate();
```

3 The Queue Library

3.1 General Description

The queue library provides two queue classes and is designed to be used with the basic Psim3 library. A simulation model that requires simple queues or priority queues must access these queue classes. The queue library provides two classes:

1. The *squeue* class for simple queues.
2. The *pqueue* class for priority queues.

All simulation models that use queues must include the `queue.h` header file at the top of the program. This gives the program implementing the simulation model access to the queue classes. For example:

```
#include "queue.h"
```

3.2 Features of Class *squeue*

The most relevant features of the class *squeue* for simple queues are the following:

- The constructor for creating a simple queue requires the name of the queue and the queue size. This last parameter is optional; if not included, the assigned default size is 1,000. For example, the following C++ statements declare and create a simple queue, *serv_queue* with an assigned name “Server Queue” and size of 15 jobs:

```
squeue *serv_queue;    // declare simple queue
...
// now create simple queue
serv_queue = new squeue("Server Queue", 15);
```

- To get the current size of a simple queue, the *length* function is provided. This function returns an integer value that corresponds to the current length of the queue. For example, to get the current length of *serv_queue* (defined above), a process uses the following statements:

```
int serv_length;           // length of server queue
serv_length = serv_queue->length(); // get queue length
```

- To check for the general states of the queue, empty or full, the *full* and *empty* functions are provided. These functions return a boolean value (true or false). For example, to check if *serv_queue* (defined above) is full, a process uses the following C++ statements:

```

if( serv_queue->full()) {
    // if queue is full execute instructions here
    ...
}
else
    // queue is not full
    ...

```

In a similar manner the *empty* function checks if a queue is empty.

- The function *into* is used to insert a process into a simple queue. The only parameter required by this function is the process to insert; this process becomes the new process at the tail of the queue. The size of the queue is increased by one. For example to insert process *serv_obj* into the simple queue, *serv_queue*, another process uses the following C++ statements:

```

server *serv_obj; // reference to a server process
...
// enqueue server process
serv_queue->into(serv_obj);

```

- The function *out* removes the process from the head of a simple queue. This function requires no parameters. The size of the queue is reduced by one, as a result of this operation. For example, to remove a *server* process from *serv_queue* and assign a reference *serv_obj* to it, a process uses the following statement:

```

serv_obj = (server*) serv_queue->out();

```

The casting for pointer to *server* is necessary to specify the type of the process dequeued.

For special scheduling that may be needed in some simulation models, the following functions are provided. These functions do not follow the conventional rules for queue processing.

- The function *last* removes the last process inserted to the queue (i.e., the process at the tail of the queue). This is an abnormal operation; the usual operation is to remove the process at the head of the queue. This function requires no parameters. For example, to remove a *server* process from the tail of *serv_queue* (the last process inserted) and assign a reference *serv_obj* to it, a process uses the following statement:

```
serv_obj = (server*) serv_queue->last();
```

- The function *back* inserts a process to the head of a queue and requires a reference to the process as the only parameter. This is another abnormal operation. For example, to insert process *serv_obj* to the head of a simple queue, *serv_queue*, a process uses the following C++ statement:

```
// put back server process into queue
serv_queue->back(serv_obj);
```

- The function *remov* removes a specified process from the queue. This function requires the reference to the process as the only parameter. For example, to remove the process referenced by *serv_obj* from queue *serv_queue*, a process uses the following statement:

```
serv_queue->remov(serv_obj); // remove process
```

3.3 Features of Class *pqueue*

The most relevant features of class *pqueue* used for priority queues are the following:

- The constructor for creating a priority queue requires the name of the queue, the number of different priorities, and the queue size for all priorities. The last two parameters are optional; if not included, the number of priorities is assigned the value 10 and the assigned default queue size is 1,000 for every priority. For example, the following C++ statements declare and create a priority queue, *serv_queue* with an assigned name “Server Queue,” using 10 different priorities and size of 15 server processes for every priority:

```
pqueue *serv_queue    // declare priority queue
...
```

```
// create priority queue
serv_queue = new pqueue("Server Queue", 10, 15);
```

- To get the current size (total number of processes) of a priority queue, the function *length* is provided. This function returns an integer value that corresponds to the current length of the queue. For example, to get the current length of *serv_queue* defined above, a process uses the following statements:

```
serv_queue->length(); // get queue length
```

- To get the number of processes of a specified priority in a queue, the function *plength* is provided. The only parameter required is the priority. The function returns an integer that corresponds to the number of processes found with the specified priority. For example, to get the current number of processes in *serv_queue*, with priority *l_prio*, a process uses the following statements:

```
// number of processes with priority
int num_proc_prio;
int l_prio; // priority
. . .
// get queue length
num_proc_prio = serv_queue->plength(l_prio);
```

- To test if the number of processes of a specified priority is zero, the function *pempty* is provided. As before, this function needs only one parameter, the priority. The function returns a Boolean value (true or false). For example, to test if the number of processes with priority *l_prio* in *serv_queue* is zero, a process uses the following statements:

```
if (serv_queue->pempty(l_prio)) {
    // execute if no processes with priority l_prio
}
```

- To test if the number of processes of a specified priority has reached its limit (upper bound), the function *pfull* is provided. As before, this function needs only one parameter, the priority. The function returns a Boolean value (true or false). For example, to test if the number of processes with priority *l_prio* in queue *serv_queue* has reached its limit, a process uses the following statements:

```

if (serv_queue->pfull(l_prio)) {
    // execute if queue with priority l_prio is full
}

```

- To test if the complete priority queue is empty, the function *empty* is provided. This function requires no parameters and returns a Boolean value. For example, to check if *serv_queue* is empty (for all priorities), a process uses the following statements:

```

if (serv_queue->empty()) {
    // execute if the queue is empty
    ...
}

```

- The function *into* is used to insert a process into a priority queue. The only parameter required by this function is the process to insert; its priority is implicit. This process becomes the new process at the tail of the priority group, for its implicit priority, in the queue. The size of the corresponding priority group is increased by one. For example to insert process *serv_obj* into the priority queue, *serv_queue*, a process uses the following C++ statements:

```

// reference to a server process
server *serv_obj;
...
// enqueue server process
serv_queue->into(serv_obj);

```

- The function *out* removes the highest priority process from the priority queue. If there are several processes with the same high priority, the process at the head of the queue is removed. This function requires no parameters. For example, to remove the highest priority process from *serv_queue* and assign it a reference *serv_obj*, a process uses the following statement:

```

serv_obj = (server*) serv_queue->out();

```

The casting for pointer to class *server* is necessary to specify the type of process dequeued.

- The function *pout* removes the next process with a specified priority from the queue. The process removed is the one at the head of its priority group. The only required priority for this function is the priority of the process. For example, to remove the next *server* process with priority *m_prio* from *serv_queue* and assign it to reference *serv_obj*, a process uses the following statement:

```
serv_obj = (server*) serv_queue->pout(m_prio);
```

As in simple queues, for special scheduling that may be needed in some simulation models, the following functions are provided. These functions do not follow the conventional rules for queue processing.

- The function *plast* removes the last process inserted into the queue with a specified priority (i.e., the process at the tail of the priority group in the queue). This is another abnormal operation. This function requires the priority as the only parameter. For example, to remove the last *server* process that was inserted into *serv_queue* with priority *l_prio* and assign it to reference *serv_obj*, a process uses the following statement:

```
serv_obj = (server*) serv_queue->plast(l_prio);
```

- The function *llast* removes the last process inserted to the queue with the lowest priority in the queue (i.e., the process at the tail of the lowest priority group in the queue). This is another abnormal operation. This function requires no parameters. For example, to remove the last *server* process with the lowest priority that was inserted into *serv_queue* and assign to reference *serv_obj*, a process uses the following statement:

```
serv_obj = (server*) serv_queue->llast();
```

- The function *pllast* removes the last process inserted to the queue with the lowest priority lower than the specified priority (i.e., the process at the tail of the lowest priority group with a priority lower than the specified priority). This is another abnormal operation. This function requires the specified priority as the only parameter. For example, to remove the last *server* process with the lowest priority lower than *ll_prio* that was inserted into the *serv_queue* and assign it to reference *serv_obj*; a process uses the following statement:

```
serv_obj = (server*) serv_queue->pllast(ll_prio);
```

- The function *pback* inserts a process to the head of its priority group in a queue and requires a reference to the process as the only parameter. This function uses the implicit priority of the process. This is another abnormal operation. For example, to insert process *serv_obj* into the head of *serv_queue*, a process uses the following C++ statement:

```
// put back server into serv_queue
serv_queue->pback(serv_obj);
```

- The function *remov* removes a specified process from the priority queue. This function requires the reference to the process as the only parameter. For example, to remove the process referenced by *serv_obj* from *serv_queue*, a process uses the following statement:

```
serv_queue->remov(serv_obj); // remove process serv_obj
```

4 The Resources

4.1 General Description

Psim3 includes two types of resources: standard resources and detachable resources. Two classes are provided for the synchronization of processes that use resources. This library provides two classes:

1. The *res* class for standard resources that allows processes to access these resources in a mutually exclusive manner.
2. The *bin* class for detachable resources that allows producer consumer process synchronization.

All simulation models that use the first resource class must include the `res.h` header file at the top of the program. This gives the program implementing the simulation model access to the first resource class. For example:

```
#include "res.h"
```

All simulation models that use the second resource class must include the `bin.h` header file at the top of the program. This gives the program implementing the simulation model access to the second resource class. For example:

```
#include "bin.h"
```

4.2 Relevant Features of the *res* Class

The most relevant features of the resource class, *res*, are the following:

- The constructor for creating a resource pool of this class requires the name of the resource pool and the pool size. This second parameter defines the initial number of available resource items in the resource pool. For example, the following C++ statements declare and create a resource pool, *serv_res*, with an assigned name “Server Resource” and a size of 15 resource items:

```
res *serv_res;           // declare resource pool
...
// create resource pool
serv_res = new res("Server Resource", 15);
```

A third parameter, which is optional, specifies the number of priorities to use. This is useful for processes with different priorities that reference objects of class *res*. The default value of this third parameter is 10.

- To get the number of available resource items in a resource pool, the function *num_avail* is provided. This function requires no parameters and returns an integer value that corresponds to the number of available resource items found in the resource pool. For example, to get the current number of available resource items in the resource pool *serv_res*, which was defined above, a process uses the following C++ statements:

```
int num_res;
...
num_res = serv_res->num_avail();
```

- For resource allocation to a process, the *acquire* function is provided. This function allows a process to acquire a specified number of resource items from a resource pool only when there are sufficient resources available. The number of requested resources is specified as the only parameter. When the number of resources is not sufficient, the process is suspended. For example, if process needs to acquire five resource items from the *serv_res* resource pool (defined above), the process uses the following statements:

```
serv_res->acquire(5))
```

If the process is suspended because of insufficient resources, it will be restarted by another process that releases resource items.

- When a process needs to release resources, it invokes the *release* function, which de-allocates a specified number of resource items that the process holds. This function requires a parameter that corresponds to the number of resource items to release. The function has no return value. For example if a process is to release three resources from the *serv_res* resource pool (defined above), it uses the following statement:

```
serv_res->release(3);
```

4.3 Features in Class *bin*

The most relevant features of the resource class *bin* are the following:

- The constructor for creating a *bin* resource container requires the name of the resource container and the initial container size. This second parameter defines the initial number of available resources in the resource container. For example, the following C++ statements declare and create a *bin* resource container, *serv_cont*, with an assigned name “Server Bin” and a size of 15 resource items:

```
bin *serv_cont; // declare resource container
...
// create a resource container (bin pool)
serv_cont = new bin("Server Bin", 15);
```

A third parameter, which is optional, specifies the number of priorities to use. This is useful for processes with different priorities that reference objects of class *bin*. The default value of this third parameter is 10.

- To get the number of available resource items in a *bin* resource container, the function *num_avail* is provided. This function requires no parameters and returns an integer value that corresponds to the number of available resource items found in the *bin* resource container. For example, to get the current number of available resource items in the resource container *serv_cont*, which was defined above, a process uses the following C++ statements:

```
int num_cont;
...
num_cont = serv_cont->num_avail();
```

- For a process to take a number of resources from a *bin* resource container, it must use the function *take*. If the number of resources in the *bin* container is sufficient, function *take* returns and the process can proceed, otherwise the process is suspended. If the operation succeeds, the number of resources in the *bin* container is decreased by the number of resource items taken by the process. For example, if a process needs to take five resource items from the *serv_cont* container (defined above), the following statements can be used:

```
serv_cont->take(5); // take 5 items from serv_cont
// execute there are sufficient resources
...
```

- When a process needs to place (or give) resource items to a *bin* container, it invokes the *give* function, which places a specified number of container items into the container. This function requires a parameter that corresponds to the number of resource items to place in the *bin* container. The function has no return value. For example, if a process is to place three resource items in the *serv_cont* container (defined above), it uses the following statement:

```
serv_cont->give(3); // place 3 items into serv_cont
```

5 The *waitq* Class

Class *waitq* manipulates the cooperation of processes. Objects of this class are used as a synchronization mechanism for the process cooperation. This synchronization allows one process to dominate and is treated as the master process; the other processes are treated as passive slaves.

All simulation models that need to use the *waitq* class must include the header file *waitq.h*, as in the following statement:

```
#include "waitq.h"
```

The relevant operations are as follows:

- The constructor for creating a synchronization object for cooperation requires a name and an optional number of priorities. This last parameter is needed for the hidden priority queues, one for the master processes and one for the slave processes. The default value for this parameter is 10. For example, to create a synchronization object with name “Cooperating Server”, and with 10 priorities (the default number), a simulation model uses the following statements:

```
waitq *coopt_serv; // declaring a ref of class waitq
...
coopt_serv = new waitq("Cooperating Server");
```

- To get the length of the *slave queue* (i.e., the number of slave processes waiting on the synchronization object), the function *length* is provided. This function returns an integer value with the number of *slave* processes waiting. For example, to get the number of slave processes waiting in the *coopt_serv* object defined above, a process uses the following statements:

```
int num_slaves;
...
num_slaves = coopt_serv->length();
```

- To get the length of the *master queue* (i.e., the number of *master* processes waiting on the synchronization object), the function *lengthm* is provided. This function returns an integer value with the number of master processes waiting. For example, to get the number of master processes waiting in the *coopt_serv* object defined above, a process uses the following statements:

```
int num_master;
...
num_master = coopt_serv->lengthm();
```

- A *slave* process that needs to cooperate with a master process must invoke the *wait* function. This function returns no value and does not require a parameter. If there is not a master process available, the slave process invoking this function is suspended. For example, a slave process attempting cooperation with a master process using the synchronization object *coopt_serv* (defined above) uses the following statements:

```

    coopt_serv->wait(); // wait for master process
    ...

```

- A master process that needs to cooperate with slave processes must invoke the *coopt* function. This function returns a reference (or a pointer) to the slave process retrieved from the slave queue. The function requires no parameters. If there is no slave process available in the slave queue, the master process invoking this function is suspended and placed in the master queue. For example, a master process that needs to cooperate with a slave process using the *coopt_serv* synchronization object uses the following statements:

```

    coopt_serv->coopt()
    // execute, a slave process was found
    ...

```

6 The *condq* Class

Class *condq* handles the waiting of processes for a specified condition (also called conditional waiting). Objects of this class are used as a synchronization mechanism for the processes to evaluate the specified condition and wait if the condition is not true. This synchronization allows processes to wait in a conditional queue by priority.

All simulation models that need to use class *condq* must include the header file *condq.h*, as in the following statement:

```
#include "condq.h"
```

The relevant operations are as follows:

- The constructor for creating a synchronization object for conditional waiting requires a name and an optional number of priorities. This last parameter is needed for the hidden priority queue of processes. For example, to create a synchronization object with name “Cond-Serv” and with six priorities, a simulation model uses the following statements:

```

    condq *cond_serv;
    ...
    cond_serv = new condq("CondServ", 6);

```

- The member function *waituntil* evaluates the specified condition and suspends the process if the condition is not true. The suspended process is placed in priority queue of the *condq* object. The function has only one parameter, a Boolean value of the condition. The function returns a Boolean value, which is false if the condition evaluates to false, otherwise the function returns true. The condition specified in the parameter is a Boolean function with no parameter. For example, a process that uses a condition *cond_a* and synchronizes with *condq* object, *cond_serv* (created above), uses the following statements:

```

bool test_val;
bool cond_a;
. . .
test_val = false;
while (! Test_val)
    cond_serv->waituntil(cond_a);
// execute if cond_a is true

```

- The member function *signal* reactivates the processes at the head of the priority queue in the *condq* object. The reactivated processes will evaluate their condition and may be suspended again. This function requires no parameter and returns no value. For example, a process that signals the *condq* object *cond_serv* (created above) uses the following statement:

```

cond_serv->signal();

```

- The member function *setall* sets the internal flag for the evaluation mode in the *condq* object. If this flag is set to true, the *signal* function allows all waiting processes to test their condition. This function requires one parameter, the Boolean value for setting the flag. The function returns no parameter. For example, a process that needs to set the flag so that all processes will be allowed to test their condition on the *cond_serv* object (defined above) uses the following statement:

```

cond_serv->setall(true);

```

- The function member *length* returns the number of processes in the conditional queue of a *condq* object. This function returns an integer value that corresponds to the conditional queue length. The function requires no parameter. For example, a process that needs to get the

number of processes in the conditional queue in the *cond_serv* object (defined above) uses the following statements:

```
int cond_num;
cond_num = cond_serv->length();
```

7 Random Numbers

Psim3 has a collection of classes for the generation of random numbers from different probability distributions. A random variable is *discrete* if its cumulative distribution function only changes value at certain points x_1, x_2, \dots and remains constant between these points. The function $F(X)$ has values p_1, p_2, \dots at these points, and $p_1 + p_2 + \dots = 1$. A random variable is *continuous* if its cumulative distribution function is continuous everywhere.

Psim3 provides several C++ classes for random number generators that use different probability distributions. The following classes are the most common ones that provide the random number generators:

- *randint*, for uniformly distributed random numbers between 0.0 and 1.0.
- *erand*, for the generation of random numbers from an exponential distribution.
- *poisson*, for the generation of random numbers from a Poisson distribution.
- *normal*, for the generation of random numbers from a normal distribution.
- *urand*, for the generation of random numbers within a specified range and using a uniform probability distribution.

7.1 Class *randint*

Class *randint* provides the most basic type of random number generation. The class uses a primitive uniform distribution. The following description gives some details on the member functions in class *randint*.

- `randint(unsigned int seed=0)`. This is the constructor for random number generator object of class *randint*. The parameter *seed* is used to set the random number stream.

This constructor is used to create a random number generator object. For example, the following C++ code declares and creates a random number generator object with 3 as the value of the seed.

```
// declare reference to random generator
randint *ran_gen1;
...
ran_gen1 = new randint(3); // create rand gen
...
```

If the seed argument is not specified when invoking the constructor, the random generator object uses a seed derived from the processor timer. This implies that there will be a different random stream of numbers for every simulation run.

- `void seed(unsigned int useed)`. This function sets the seed for the random number stream. The parameter *useed* is the value for the seed. This member function is used when the random generator object has already been created and there is to set a value for the seed. For example, the following C++ code sets the seed to 13 for generator object *ran_gen1* defined above.

```
ran_gen1->seed(7); // set seed to 7
...
```

- `double fdraw()`. This member function generates a random number between 0.0 to 1.0, using a generator object already created. For example, the following C++ code generates a random number from the generator object *ran_gen1* defined above, and stores the number in variable *ran_value*.

```
double ran_value;
...
ran_value = ran_gen1->fdraw(); // generate random number
...
```

- `long draw()`. This member function generates a random number between 0 to *RAND_MAX*, using a generator object already created. For example, the following C++ code generates a random number from the generator object *ran_gen2*, and stores the number in variable *ran2_value*.

```

long ran2_value;
...
ran2_value = ran_gen2->draw(); // generate random number
...

```

Every time the simulation model needs a random number, it calls the *draw* or the *fdraw* function that returns a 'random' number.

7.2 Class *erand*

Class *erand* provides random number generation using a negative exponential distribution. The following description gives some details of the member functions in class *erand*.

- `erand(double emean, unsigned int eseed=0)`. This is the constructor for initializing the exponential random number generator. The parameters are *emean*, the mean value of the distribution and *essed*, the seed value for selecting a random stream. `erand(long emean, unsigned int eseed=0)` is the overloaded constructor.

These constructors are used to create a random number generator object, the arguments to be supplied are the mean value of the samples, and the seed (optional argument). For example, the following C++ code declares and creates a random number generator object *ran_gen* with mean of 34.5 and 3 as the value of the seed.

```

// declare reference to random generator
erand *ran_gen;
...
ran_gen = new erand(34.5, 3); // create generator
...

```

- `double fdraw()`. This member function generates a random number using the exponential distribution with the mean value given in the constructor. For example, the following C++ code generates a random number of type *double* using random generator *ran_gen* (created above) and stores the random number in variable *ran_value*.

```

double ran_value;
...
ran_value = ran_gen->fdraw();
...

```

If the seed argument is not specified when invoking the constructor, the random generator object uses the processor timer for the seed. This implies that there will be a different random stream of numbers for every simulation run.

7.3 Class *normal*

Class *normal* provides random number generation using a normal distribution between two given values. The following description presents details of the member functions in class *normal*.

- `normal(long mean, long stdev, unsigned int seed=0)`. This is the constructor for random number generator object using the Normal distribution. The parameters are: *mean*, the mean value of the distribution, *stdev*, the value of the standard deviation, and *seed*, the value to set the random stream. The overloaded constructor is:

```
public Normal(double rmean, double rstd, int seed=0)
```

The constructor is used to create and initialize a random number generator object using a Normal distribution with the given value of the mean, the standard deviation, and the value of the seed (optional argument). For example, the following C++ code declares and creates a random number generator object *norm_gen* with mean 22.5 and standard deviation of 1.3.

```
normal *norm_gen; // reference to random generator
...
norm_gen = new normal(22.5, 1.3); // create generator
...
```

- `double fdraw()`. This member function generates a random number using the normal distribution with the mean value and the standard deviation given in the constructor. For example, the following C++ code generates a random number of type *double* using random generator *norm_gen* (created above) and stores the random number in variable *ran_value*.

```
double ran_value;
...
ran_value = norm_gen->fdraw(); // generate random number
...
```

If the seed argument is not specified when invoking the constructor, the random generator object uses the processor timer for the seed. This implies that there will be a different random stream of numbers for every simulation run.

7.4 Class *poisson*

Class *poisson* provides random number generation using a Poisson distribution. The following description presents details of the member functions in class *poisson*.

- The constructor used to initialize a random number generator using a Poisson distribution is:

```
poisson(double prate, double pperiod, int seed=0)
```

The parameters of the constructor are: *prate*, the rate of arrival (arrivals per unit time), *ppperiod*, the period considered for arrivals, and *seed*, the seed for selecting an appropriate random stream. The constructor is used to create and initialize a random number generator object with the given value of the arrival rate, the period for arrivals, and the value of the seed (optional argument). For example, the following C++ code declares and creates a random number generator object *poisson_gen* with mean 18.45 (arrivals per time unit) and arrivals period of 10.5 time units.

```
normal *poisson_gen; // reference to random generator
...
// create rand generator
poisson_gen = new poisson(18.45, 10.5);
...
```

- `int fdraw()`. This member function generates a random number using the Poisson distribution with the values of the arrival rate, the period for the arrivals, and the seed, given in the constructor. For example, the following C++ code generates a random number of type *int* using random generator *poisson_gen* (created above) and stores the random number in variable *ran_value*.

```
int ran_value;
```

```

...
ran_value = poisson_gen->fdraw();
...

```

If the seed argument is not specified when invoking the constructor, the random generator object uses the processor timer for the seed. This implies that there will be a different random stream of numbers for every simulation run.

7.5 Class *urand*

Class *urand* provides random number generation using a uniform distribution between the values given. The following description presents details of the member functions in class *urand*.

- `urand(long low, long high, unsigned int useed=0)`. This is the constructor for the random number generator with Uniform distribution. The parameters are: *low*, the low bound for the range of values, *high*, the upper bound for the range of values, and *useed*, the seed to set the random stream.

The constructor for random number generator with uniform distribution using values of type *double* is the following:

```
urand(double flow, double fhigh, int useed=0);
```

The parameters are: *flow*, low bound for the range of values, *fhigh*, the upper bound for the range of values, and *useed*. The constructor is used to create and initialize a random number generator object with the given value of the low bound, high bound, and the value of the seed (optional argument). For example, the following C++ code declares and creates a random number generator object *unif_gen* with low bound of 6.35 and high bound of 35.55.

```

rand *unif_gen; // reference random generator
...
unif_gen = new urand(6.35, 35.55); // create generator
...

```

- `double fdraw()`. This member function generates a random number using the uniform distribution with the low and high limits given in the

constructor. For example, the following C++ code generates a random number of type *double* using random generator *unif_gen* (created above) and stores the random number in variable *ran_value*.

```
double ran_value;
...
ran_value = unif_gen->fdraw();
...
```

If the seed argument is not specified when invoking the constructor, the random generator object uses the processor timer for the seed. This implies that there will be a different random stream of numbers for every simulation run.

8 The Simulation Package on the Web Site

8.1 Files Available

- The following are the precompiled libraries and additional files for using the Psim3 package.
 - `VCCpsim3.zip`, the Psim3 library compiled with Visual C++ 6.0. This compressed files includes the *Pthreads* library and the various header files.
 - `MGWpsim3.zip`, the Psim3 library built with the GNU C++ (MinGW) compiler for Windows. This compressed files includes the *Pthreads* library and the various header files.
 - `linus_psim3.tar.gz`, the Psim3 library built with the GNU C++ compiler on Linux Red Hat Enterprise 4. This compressed archive file includes the various header files.
- Several C++ files for sample simulation models.

8.2 Brief Instructions for Compiling and Linking

For compiling and linking the source files of the simulation models, note that the two external libraries necessary are the *Psim3* and the *Pthreads* libraries. With the development environment and compiler used, the compilation should be carried out with the necessary source files including the header files `proc.h` and `pthread.h`.

For the linking step, the appropriate version of the Psim3 and Pthreads library needs to be included in the project file or the make file. For example, the DOS command for compiling and linking a source C++ program (e.g., `batch.cpp`) on Windows with the GNU C++ (MinGW) compiler, is the following:

```
g++ batch.cpp libpsim3.a libpthreadGC.a
```

This simple single command line appears in the batch file called `psim3.bat`. Assuming that the current directory is `psim3\models` and the program in file `batfcfs.cpp` is to be compiled and linked, then the command line at the DOS prompt is:

```
c:\psim3\models> psim3 batfcfs.cpp
```