SIMON: A SIMULATOR OF MULTICOMPUTER NETWORKS

Richard M. Fujimoto

Computer Science Division
Electrical Engineering and Computer Sciences
University of California, Berkeley, CA 94720
August 31, 1983

ABSTRACT

This document describes a simulator for modelling execution of parallel programs on a multiprocessor. The simulator executes a set of programs as if each were run on a separate processor, and compiles statistics for the entire run. A portion of the simulator, known as the "switch model", simulates the exchange of messages among the processors through an interconnection network. The simulator was developed to allow different types of interconnection hardware (e. g. packet switches, crossbars, etc.) to be modelled by simply "plugging in" the appropriate switch model.

Applications are programmed as a set of communicating tasks (processes). The interface seen by the applications programmer is discussed, and in particular, the communications mechanism is described in detail. Examples are given. Finally, the implementation of Simon is described.

1. INTRODUCTION

This document describes a simulation program called Simon (simulator of multicoputer networks). It is assumed that the reader has at least a reading knowledge of the C programming language [Kern78]. All applications programs writen for Simon should be written in C.

1.1. What is Simon?

Simon is a deterministic event driven simulation program which models the execution of a parallel application program on a multiprocessor computing system. The application program, provided by the user, consists of a number of sequential programs which execute concurrently. Simon executes each sequential program as if it were run on a seperate processor. Message transmissions between the processors are simulated, and statistics concerning the program's dynamic behavior (e.g. execution time, time spent waiting for data, etc.) are reported.

Simon consists of three distinct modules:

(1) The application program.

(2) The simulator base.

(3) The switch model.

Each of these will now be described in turn.

The first component, the application program, consists of a number of user defined "tasks" written in C. A task is defined as a sequential program, and the data it uses, executing on a processor. The tasks execute asynchronously, and communicate by exchanging messages. Intertask communication is made possible by routines provided by Simon for sending and receiving messages. No shared memory is allowed, i.e. no two tasks can directly access the same memory location. A task is conceptually identical to a process, however this term is reserved to refer to a Simon program executing on a host computer.

The second component, the simulator base, time multiplexes execution of the tasks on the host computer, in this case a VAX. The base also collects statistics on the application program and outputs them when execution completes.

The third component, the switch model, simulates the transmission of messages among the processors. The switch model might be, to mention a few, a crossbar, an Ethernet, or a store-and-forward communications network. By seperating the model for the interconnection switch into a separate module, alternate switching structures can be compared by just "plugging in" the appropriate switch models.

1.2. Who Should Use Simon?

Simon was written with two types of users in mind:

(1) Persons interested in developing and analyzing parallel application programs.

(2) Persons interested in analyzing the performance of various interconnection schemes under loads generated by real application programs.

In the first case, applications such as circuit simulation and voice recognition are evisioned, although Simon is by no means restricted to these applications. In the second case, Simon provides an alternative to simulation models based on loads created artificially by random number generators.

1.3. Restrictions

In order to reduce the complexity and overhead of running Simon, several restrictions have been made. First, this implementation assumes that each task runs on its own processor. Thus, two distinct task may not execute on the same processor. The maximum number of processors (tasks) however, is virtually unlimited, subject only to memory constraints on the computer running Simon. Further, it is assumed that tasks and communications among tasks are statically defined, i. e. all tasks must be created before any can begin execution, and each task must initially specify all other tasks with which it may send or receive messages.

1.4. About this Document

This document is intended for those who are interested in writing parallel application programs for Simon, or for those interested in understanding its internal structure and implementation. For the former, mechanisms for creating tasks are defined, as well as intertask communications facilities. For the latter, an overview of the current implementation of Simon for a VAX-11/780 will be discussed.

The remainder of this document consists of six sections, and a number of appendices. The next two sections describe what the user needs to know to write application programs. Section 2 describes the routines provided for creating and executing tasks, and section 3 describes routines for exchanging messages. Section 4 gives some examples using the routines defined in the previous two sections. Section 5 describes the timing model which is used for gathering statistics on the task. Section 6 gives some guidelines for writing programs for Simon. These guidelines are provided to avoid some rather subtle bugs in applications programs which Simon cannot detect. Finally, section 7 describes the internal organization of the Simon program.

A summary of all of the routines discussed in this document is given in Appendix I. Other appendices describe the mechanics of using Simon and the special C compiler necessary for the timing model.

2. TASKS

An application program can view SIMON as a set of subroutines for performing various functions, much like a C program can view an operating system as a set of subroutines for, e. g., printing results or reading data from a file. This section and the next describe the functions provided by SIMON as well as the routines which implement them.

The application program executing on the multiprocessor is a set of concurrently executing "tasks" (or equivalently, processes). A task can be thought of as an instance (i. e. a loaded and executing core image) of a C program. A program, like that in a conventional uniprocessor, consists of a set of routines which call each other in some arbitrary fashion. One routine in the program for a task acts as the "main procedure", and is called when the task first begins execution. This routine should *not* be called "main".

Several tasks may be created from a single program. To the user, this is equivalent to generating multiple copies of the program, and creating a single task for each one. In reality however, SIMON shares a single copy of the code among the tasks, and creates a separate data area for each.

This section describes the routines provided by SIMON for creating and executing tasks. First, task creation via the "mktask()" routine is described. A task cannot be created however, unless there already exists another task to create it, presenting a "chicken and egg" problem. To solve this problem, a routine called "bootstrap()" is provided for getting the simulator started. This is discussed in the following subsection. If multiple tasks are created from a single program and that program uses static or global variables, a routine, called "init()", must be used. This is the subject of the third subsection.

2.1. Creating Tasks: The Mktask() Routine

SIMON provides a routine called mktask() for creating tasks. Mktask() performs several functions. In addition to "creating" the task, mktask() assigns the task a user provided name and task "id". Every task has a unique id which is used internally to distinguish it from other tasks (note that task names need not be unique). This id may also be used in conjunction with some switch models to assign certain tasks to execute on specific processors. Space for maintaining information necessary for the task to execute (e. g. state information) is also allocated, as well as space for collecting statistics. Mktask() is also used to identify the task's main procedure. When the task first begins execution, this procedure is called via an ordinary subroutine call. Finally, mktask() allows the user to specify parameters which are passed to the created task when it begins execution.

Mktask() takes five parameters, and is defined as follows:

mktask (name, cdptr, id, parm, lngth);

where:

name	= a character string specifying the name of the task.
cdptr	= a pointer to the main procedure for the task.
id	= a positive integer specifying the id to be assigned to the task.
parm	= a pointer to a parameter list.
lngth	= the length of this parameter list in bytes.

It is an error to assign two tasks to the same id, or to assign a task to id 0. It will be seen later that task id 0 is reserved for the "bootstrap" task.

The last two parameters are used to copy the parameter list into an internal buffer. A pointer to this buffer is passed to the task when it is first called. If the size of the parameter list is specified as zero, mktask() assumes no parameters are to be passed to the task. The parameter pointer must be NULL in this case, or an error will result.

Thus, the main procedure of each task must have at most one parameter, a pointer to parameter(s) used by that task. If a task requires more than one parameter, a structure should be created, and a pointer to this structure passed to mktask().

The parameter list passed to a task should not contain pointers, i. e. all parameters passed to the task should physically reside in the block of memory specified by the last two parameters passed to mktask(). Mktask() copies the specified data without interpretation, so any data stored at the address specified by the pointer may have been changed by the time the task begins to execute. Passing pointers in the parameter list can lead to rather obscure bugs. An example of this will be seen later.

To illustrate the use of mktask(), a few examples will now be given. The simplest form of mktask() is shown below:

```
bootstrap()
{
int foo();
mktask ("foo", foo, 5, NULL, 0);
}


foo()
{
code for task foo
}
```

This creates a task called "foo" which is assigned to task id 5. The main procedure of the task is also called foo. Foo is not passed any parameters. Bootstrap() is a task already in existence, and will be discussed in the next subsection.

A second example demonstrating the passing of a single parameter to a task is given below:

```
bootstrap()
{
int fie();
int i;

i = 20;
mktask ("fie", fie, 6, &i, sizeof (int));
}


fie (p)
int *p;
{
printf ("Parameter passed is %d\n", *p);
}
```

Here, task fie is created and assigned to task id 6. When executed, the line "Parameter passed is 20" will be displayed on the terminal screen. Note the use of the sizeof() routine to get the size, in bytes, of a C object.

Finally, a third example demonstrates the passing of several parameters to two different task via structures.

```
struct parm     /* format of parameters */
{
int  a[10];   /* Pass an array */
int  n;       /* number of elements */
};


bootstrap()
{
int T();
struct parm p;
int i, id;

id = 1;        /* id assigned to tasks */

p.n = 10;
for (i=0; i<p.n; i++) p.a[i] = i;
mktask ("T", T, id++, &p, sizeof (struct parm));

for (i=0; i<p.n; i++) p.a[p.n-1-i] = i;
mktask ("T", T, id++, &p, sizeof (struct parm));
}


T (p)
struct parm *p;
{
int i;

for (i=0; i< p->n; i++)
        printf ("%d\n", p->a[i]);
}
```

Two tasks are created from the program T. The first is passed an array of integers in ascending order, and the second is passed an array in descending order. Note that the structure parm contains no pointers. Suppose we decided to pass the tasks a pointer to the data rather than the data itself. Bootstrap() might then be defined as follows:

```
struct parm     /* format of parameters */
{
int  *a;      /* Pass a pointer!! */
int  n;       /* number of elements */
};


bootstrap()
{
```

```
int T();
struct parm p;
int i, id, buf[10];

id = 1;
p.n = 10;
p.a = buf;

for (i=0; i<p.n; i++) p.a[i] = i;
mktask ("T", T, id++, &p, sizeof (struct parm));

for (i=0; i<p.n; i++) p.a[p.n-1-i] = i;
mktask ("T", T, id++, &p, sizeof (struct parm));
}
```

There are two reasons why this program malfunctions. First, buf is declared as an automatic (i.e. non-static local) variable, and is thus kept on the runtime stack for bootstrap(). The pointer passed to the tasks is thus a pointer into the runtime stack! When bootstrap() returns, its stack is returned to the system, and subsequently overwritten by other procedures. Thus, the created tasks receive garbled data. However, even if buf were declared as a global variable (and thus, not kept on the stack), there is still another problem. The first task is passed only a pointer to the data, and not the data itself. The second "for loop" in bootstrap() modifies this data, and so the result is *both* tasks receive the array in descending order. In the previous (correct) example, the actual data was passed via mktask() to the newly created task. Since mktask() creates a separate copy of this data, it cannot be changed by any user program. For this reason, it is recommended that structures passed to tasks as parameters contain no pointers fields.

2.2. Getting Started: The Bootstrap() Routine

In order for the user to create and execute tasks, he must execute mktask(). In order to execute mktask(), he must have created and executed a task. To end this cycle, SIMON assumes the preexistence of a user-defined task called bootstrap(). Bootstrap() is passed no parameters, and is arbitrarily assigned to task id 0.

Bootstrap() uses mktask() to create the user tasks, and then returns control back to the simulator, never to be executed again. It's sole purpose is to create the set of tasks the user wishes to execute. Bootstrap() is not allowed to communicate with any other task. All tasks, including bootstrap(), are created and begin execution, at time 0, regardless of when mktask() was executed. Thus, in this respect, bootstrap is a somewhat artificial task used only to get the simulation started. Within SIMON, bootstrap() is treated as any ordinary task however.

The first implementation of SIMON assumes that tasks cannot be created dynamically. All tasks must be created by bootstrap(). Thus, it is an error for any task other than bootstrap() to execute the mktask() routine.

2.3. Multiple Instances of Tasks: The Init() Routine

In sequential programming, one often writes a subroutine to perform some function independently of the specific data it is to operate on. This function is then called from various points in the program, each time specifying the data via parameters. Thus, the algorithm remains the same, but the data varies from call to call. Each call to the subroutine creates an "instance" of that subroutine. Local variables are created, actual parameters (those specified in the function call) are mapped to formal parameters (those specified in the subroutine), etc. Since execution is sequential, at most one instance of the subroutine exists at any given time, ignoring recursion.

Similarly, it is useful to write one program and then create many tasks from this program which only differ in the data being operated on. Here however, the various tasks execute concurrently, and thus exist simultaneously. Just as one parameterizes instances of a subroutine, one must also parameterize tasks (as described above in the mktask() routine). This is useful for example, in array operations where each task performs the same computation, but on different parts of the array.

When multiple instances of a task are created from a program using non-automatic (i. e. static or global) variables, the simulator defined routine init() must be used. Init() tells SIMON where these variables reside. SIMON must have this information so that these variables can be saved and restored when execution switches among tasks created from the same program. Automatic variables are stored on the runtime stack of the executing program, and are saved and restored by SIMON when necessary, transparent to the user.

Non-automatic variables should all be stored in a single block of memory which will be referred to here as the "global data area". Init() has two parameters, and is defined as follows:

init (glob, lngth);

where:

glob	= a pointer to the start of the global data area.
lngth	= the length of this area in bytes.

Init() need only be used in a task if multiple instances are created. It must be executed exactly once by such tasks, and must execute before any other simulator defined routine is executed.

To use init(), the user should:

(1) place ALL of the non-automatic variables for the task into a structure.

(2) call init() using a pointer to this structure, and a size gotten from sizeof().

In addition, it is recommended that arrays be created dynamically via one of the storage allocation routines (e. g. calloc()) provided in C. This will reduce the execution time of the simulator, since variables created by (say) calloc() need not be saved and restored when execution switches from one task to another. This is because each task (even those created from the same program) dynamically creates it's own data area to which it has sole access. In other instances, this recommendation becomes a requirement, since each task is allocated a fixed size area for saving its stack. If an attempt is made to save more data than this area can hold, an execution error results. Data areas created by calloc() are not kept on the stack.

In the discussion above, it is assumed that non-automatic variables are used to share data between procedures of the same task. Different tasks must not use global variables to exchange information, as this compromises the simulation by performing communications between tasks without using the switch model.

The following is an example of a program from which several tasks are created. Note the use of calloc() for creating arrays, and the manner in which non-automatic variables are declared so their location can be passed to SIMON via init().

```
static struct gstruct   /* Global data area */
{
int a, b, c;           /* Some scalars */
int *g_arr;            /* A global array */
} glob;


foo ()                 /* Program for a task */
{
int i, j;              /* Automatic scalar */
int *a_arr;            /* Automatic array */
char *calloc();        /* Storage allocator */

/* Define global data area */
init (&glob, sizeof (struct gstruct));

/* Create automatic variable array */
a_arr = (int *) calloc (100, sizeof (int));

/* Create global array */
glob.g_arr = (int *) calloc (20, sizeof (char));

/* Reference globals and automatic locals */
i = 20; j = 30;        /* Automatic scalar */
a_arr[j] = 30;         /* Automatic array */
glob.a = 20;           /* Global scalar */
glob.g_arr[i] = 5;     /* Global array */
        .
        .
        .
}
```

2.4. Other Routines

Finally, two other routines pertaining to tasks will be mentioned. These are task(), which returns a task's id, and taskname(), which returns a pointer to the task's name (the character string passed to mktask() when the task was created). The character string returned must not be overwritten, as it is SIMON's only copy.

The task() routine takes no parameters, and returns an integer value. Taskname() takes a single parameter, the id of the task in question, and returns a pointer to a character string. Thus, a task can determine it's own name by calling:

```
taskname (task());
```

The information passed through a task's name or its id may be used for further parameterization of the task.

3. INTERTASK COMMUNICATIONS

This section describes the mechanisms for allowing tasks to send/receive messages to/from other tasks. Here, two types of communications mechanisms are defined:

(1) specification of which tasks communicate with which other tasks.

(2) sending and receiving messages.

The first mechanism uses "fifo's" whose names are globally known throughout the multiprocessor. Each task has a set of "export" and a set of "import" fifo's for sending and receiving messages respectively. A task sends a message by putting it into one of his globally known export fifo's. A message is received by taking it out of an import fifo. The details of how the data is transported from an export to an import fifo is left up to the switch model, and will not be discussed here.

The next subsection discusses the basic communications mechanism in greater detail. Following this, the subroutines implementing each of the two types of communications mechanisms defined above will be described. A concise definition of the routines defined in this section is made in Appendix I.

3.1. Overview of the Communications Mechanisms

Data may be transmitted to or received from other tasks via fifo's. Each task creates a set of fifo's which act as the interface between it and any tasks it communicates with.

Fifo's hold messages which are being sent to or have arrived from other tasks. Thus, elements (i. e. messages) of the same fifo may vary in size. The contents of messages are not interpreted by the simulator.

Each fifo has a user defined name (an arbitrary character string) which is globally known by all other tasks. Furthermore, fifo's are classified as being either "export" or "import". When a task creates an export fifo of some name (say X) it is given the capability to "export" data (i. e. send messages) to all tasks which have an import fifo called X.

A task may have both an export and an import fifo called X, but it cannot have more than one import or export fifo of the same name. Several tasks may each have an import fifo called X, allowing one to easily broadcast data to several tasks. Similarly, several tasks may have export fifo's of the same name, allowing several tasks to send data to one (or more) import fifo(s). Thus, there are no restrictions on creating fifo's other than creating more than one import/export fifo of the same name within a single task.

Thus, communication paths among tasks are defined by the names of the fifo's created within each task. When a task wants to send a message to another task, it puts the data into one of its export fifo's. The switch model removes the data from the export fifo, replicates it as many times as necessary, and places a copy in each import fifo of the same name as the export fifo the data was originally put in. All of this is transparent to the programmer. The receiver now only needs to get the data out of its import fifo. Two tasks must have a commonly named fifo to exchange data directly.

Sequentiality is preserved between any pair of export and import fifo's of the same name. If two messages are placed in export fifo "X", one after the other, the simulator will guarantee that these messages arrive in all import fifo's named "X" in the same order in which they were sent. In network terminology, every pair of identically named export and import fifo's form a "virtual circuit".

The next subsection describes creation of export and import fifo's. The following subsection describes the routines provided for putting (getting) data into export (from import) fifo's.

3.2. Fifo Creation

This subsection describes the simulator defined routines for creating fifo's. Export() and import() are the basic routines used for creating export and import fifo's respectively. Also, arrays of fifo's can be created using the exparr() and imparr() routines. Two additional routines called cnvarr() and cnvarrsv() are provided to "index" into an array of fifo's.

When a task begins executing for the first time, it must create all of the fifo's it will need for the entire simulation run. No dynamic fifo creation is allowed. After this initial execution period (which ends when the task gives up the cpu to allow other tasks to execute) the task must not attempt to create any new fifo's. Any attempt to do so will be flagged as an error.

3.2.1. Export()

The export() function creates a single export fifo. It takes a single parameter, a character string (or more accurately, a pointer to a character string) specifying the name of the fifo. When export() is called, the simulator creates an export fifo of this name, allowing the task to send data to all other tasks which have an import fifo of the same name. Thus, the statement:

```
export ("X");
```

creates an export fifo called X.

3.2.2. Import()

Import() is very similar to export(). Import() however, takes two parameters. In addition to the name parameter, it requires a parameter specifying the maximum number of messages the import fifo can hold. If a 0 is specified, the length of the fifo is not bounded (this is the most commonly used case). Fifo's with a maximum length specified are called bounded fifo's, and are useful for applications where a task receiving data is not interested in all the data sent to it, but only the last (say) 3 values.

When import() is called, the simulator creates an import fifo of the specified name. This allows the task to receive data sent by other tasks with export fifo's of the same name. Data arriving into a bounded fifo which is full causes the oldest data (at the front of the fifo) to be discarded, the new data to be placed at the end of the fifo, and a flag (readable by overfl() defined below) to be set. Note that only import fifo's have bounded length. For example,

```
import ("Y", 0);
import ("Z", 3);
```

creates two import fifo's. The fifo called Y is of unlimited length, but the fifo called Z will only hold up to 3 messages.

3.2.3. Exparr(), Imparr(), Cnvarr(), and Cnvarrsv()

These routines are used for creating and accessing arrays of fifo's. An "array of fifo's" is simply a set of fifo's, all of which are either export or import fifo's, which follow a certain naming convention. The convention for naming individual fifo's in an array is to concatenate a "base name" with a subscript enclosed by brackets ([]). Thus the name of a fifo which is part of an array looks like a C array variable. Subscripts start at 0.

Exparr() creates an array of export, and imparr() an array of import fifo's. Exparr() takes two parameters, one specifying the base name to be used in naming the fifo's, and the other specifying the length of the array. Imparr() takes these two parameters plus a third giving the maximum length of the fifo's, or 0 if their length is not limited. All import fifo's of an array must have the same maximum length. Exparr() and imparr() use export() and import() respectively to create fifo's. They could easily be written by the user, but are provided as a convenience and to form a convention for naming individual fifo's in a FIFO array.

For example,

```
exparr ("X", 4);
```

creates four export fifo's (with base name X) called "X[0]", "X[1]", "X[2]", and "X[3]". Note that no blanks are automatically inserted into the fifo name. Remember that fifo's which are members of arrays are identical to ordinary fifo's. The only difference is that they follow this naming convention. A task could create a single fifo corresponding to one element of the array by simply calling export() (or import()), as for example:

```
export ("X[2]");
```

In practice, one would like to access the ith fifo, where i is an integer variable. To do this, a mapping function called cnvarr() is provided. Cnvarr() takes two arguments, a character string specifying a base name, and an integer variable specifying a subscript. Thus, cnvarr ("X", 2);

returns the character string "X[2]". No checking is done to see if the resulting name corresponds to any fifo existing in the system. An error will result however, if an attempt is made to use an invalid fifo name.

The value returned by cnvarr() should never be stored into another variable. It should only be used as the argument passed to a simulator defined subroutine (e. g. put() or get(), defined below). The character string returned is only valid until the next time the task executes cnvarr(). This is because each task uses a single buffer for holding character strings generated by cnvarr(). Cnvarr() always returns a pointer to the start of this buffer. Thus, if two successive calls to cnvarr() are made, the result of the second will overwrite the first.

If it is necessary to assign the pointer returned by cnvarr() to another variable, the cnvarrsv() routine should be used. This is identical to cnvarr(), except the storage for the resulting string is allocated by calloc(), and thus remains indefinitely. Calloc() is not used on calls to cnvarr() to conserve storage.

3.3. Sending and Receiving Messages

This subsection describes routines provided by the simulator for sending and receiving messages. In addition, routines for interrogating the status of fifo's are also discussed.

Three functions, put(), puts(), and get() are available for putting data into export fifo's and getting data from import fifo's. A routine called waitf() allows a task to wait for data to arrive in any of several import fifo's. Routines called size() and qlength() are available for determining the size of the next message and the number of messages in an import fifo. Finally, a routine called overfl() is used to detect overflow of import fifo's which have a limited length (bounded fifo's). The following is a detailed description of each of these routines.

3.3.1. Put() and Puts()

These two routines are used to send messages, i. e. put data into an export fifo. The put() routine takes three parameters. The first is a character string specifying the name of the export fifo data is to be put into. The second is a pointer to the first byte of data to be sent. The third is the number of bytes to be sent. When called, a contiguous block of data L bytes long (where L is the third parameter) beginning at the memory location specified by the second parameter is copied into one of the simulator's internal buffers (forming a single message), and added to the export fifo specified by the first parameter. Thus each element of the fifo is a block of data (i. e. a message), which for a given fifo, may vary in length from element to element. Thus, the statement

```
put ("X", &i, sizeof(i));
```

sends a message consisting of the integer (assume i is declared as an integer) which is the current value of the variable i.

If a task declares an export and import fifo of the same name, it is usually not desired that a put() cause the task to send data to itself. Thus, put() only causes data to be sent to other tasks. In some situations however, it may be convenient to allow a task to receive its own messages. For example, suppose a task needs to process data which is sometimes created locally within that task, but other times remotely in some other task. Allowing a task to send data to itself allows the receiver to treat the two cases as one. For this purpose, the function puts() (put to self) is defined. Puts() uses the same parameters as put(). The first parameter should be the name of a fifo which is both an export and an import for that task. When a task executes puts(), it sends data to itself as well as to all other import fifo's of the same name. If a task executes puts() on an export fifo for which it does not also have an import fifo of the same name, an error will result.

3.3.2. Get()

Get() takes two parameters, a character string specifying the name of some import fifo created by the task, and a pointer to where the received data is to be stored. When called, the next message of the specified import fifo (in the general case, a block of data of arbitrary length) is removed from the fifo and copied into contiguous memory locations starting at the address specified by the second parameter. If the specified import fifo is empty when get() is called, the task waits for data to arrive. The waiting is of course, transparent to the programmer. Thus, the statement

```
get ("X", &j);
```

might be used to receive the value of i sent by the task in the put() example above. When the get() completes execution, j will hold whatever data was sent. Note that the above implies that the receiver knows what type of data is being sent. It is up to the user to ensure that the receiver correctly interprets any data sent to it. If the next message in fifo X held (say) a floating point number, the simulator would not complain, but the result would, in general, be unpredictable if the receiver were expecting an integer.

3.3.3. Waitf()

Waitf() takes two parameters. The first is an array of character strings (i. e. pointers to character strings), each of which is the name of an import fifo created by that task. The second parameter is an integer specifying the length of this array. When executed, the simulator checks each of the specified import fifo's, and if all are empty, the task blocks. When a message arrives in one of these fifo's, the task is restarted at the instruction following the waitf(). If one or more of the specified fifo's is not empty when the waitf() is executed, then the waitf() acts like a no-op.

The waitf() function allows a task to wait for data to arrive in any of a set of import fifo's. When a message arrives and is placed into one of these import fifo's, the task must determine which fifo the message arrived in. Clearly, get() cannot be used for this purpose, since a get() on an empty fifo will cause the task to block. Two routines which can perform this function are size() and qlength(). These will be discussed next.

3.3.4. Size() and Qlength()

The size() function takes a single parameter specifying an import fifo created by the task. When called, it returns the size (in bytes) of the next message in this fifo. If the fifo is empty, size() returns 0. This routine is useful for receiving variable length data.

Qlength() also takes a single parameter specifying the name of an import fifo. This routine returns the number of messages currently in that fifo. If the fifo is empty, qlength() returns 0. It, like size(), can be used for polling fifo's to see if they have any data (see discussion of the waitf() routine defined above). For efficiency reasons however, it is recommended that the user use size() for this purpose.

3.3.5. Overfl()

Overfl() takes a single parameter specifying an import fifo created by the task. Each import fifo has a flag associated with it which is set when a message arrives in a bounded fifo (i. e. one whose length is limited to a fixed number of messages) which is full, causing the oldest message in the fifo (the front message) to be lost. Overfl() returns the value of this flag, and causes the flag to be reset. Thus, it is similar to the overrun bit in a UART.

4. Examples

To clarify the ideas presented so far, this section presents several examples of the use of the functions defined above. The first is a simple use of export and import fifo's to transport an array from one task to another. The second uses the size() and waitf() functions to implement a "server" task. Finally, the third example demonstrates the creation of multiple tasks from a single program to compute the dot product of two vectors.

4.1. Exporting an Array

This example demonstrates how one can export an array of length N from one task to another. Two tasks, T1 and T2, respectively send and receive the array A. The sending task, T1, is defined as:

```
#define N 100              /* size of array */

T1()                    /* task to send array */
{
int A[N];                   /* array to be sent */

export ("A");               /* create export fifo A */
getarray (A);               /* routine to input array A */
put ("A", A, N*sizeof(int));   /* send array */
}

getarray(A)                 /* routine to input array */
int A[];

{
int i;
for (i=0; i<N; i++) {         /* input array */
      printf ("enter data:");
      scanf ("%d", A + i);
      }
}
```

The receiving task, T2, is defined as:

```
T2()                    /* task to receive array */
{
int B[N];                   /* array to hold result */

import ("A",0);              /* create import fifo A */
get ("A", B);              /* get array */
}
```

The bootstrap routine for this pair of tasks is defined as:

```
#include <stdio.h>

bootstrap()

{
int T1(), T2();              /* names of the tasks */

mktask ("T1", T1, 1, NULL, 0);
mktask ("T2", T2, 2, NULL, 0);
}
```

4.2. A Server to Square Variables

A "server" is implemented which will square and return any floating point value sent to it. It is assumed that each task which uses the server has an export and an import fifo whose name is "A[i]", where i is some integer assigned to that task. To use the server, a task sends a floating point number on its export fifo, and waits for the result to arrive on its import fifo.

In this example, the server uses the waitf() routine to wait for data to arrive in one of N fifo's. It then uses size() to poll the fifo's to determine which one received the data. One could have used qlength() instead of size(), however size() is executed more efficiently by the simulator. Thus, size() is recommended when testing to see if a fifo is empty.

The server first sets up an array of character strings (actually, an array of pointers to character strings) which lists the set of fifo's it will waitf() on. This is followed by an infinite loop which consists of two actions:

(1) wait for data to arrive in one of its fifo's

(2) poll its fifo's, and return a result for any data that has arrived.

Thus, the server task SQUARE is defined as:

```
#include <stdio.h>
#define N 10

SQUARE()
{
int i;
char *p[N];              /* pointers to fifo names */
char *cnvarrsv(), *cnvarr();
float x;
```

```
    exparr ("A", N);            /* create arrays of fifo's */
    imparr ("A", N, 0);

    /* set up array of fifo names for waitf() procedure */

    for (i=0; i<N; i++)
            p[i] = cnvarrsv ("A", i); /* fifo name */

    /* At this point, p[0] points to the character string
       "A[0]", p[1] to "A[1]", etc. */

    for ( ; ; ) {               /* loop forever */
            waitf (p, N);          /* wait for data */

      /* Execute the following when data arrives.
         Find fifo with data and return result. */

            for (i=0 ; i<N ; i++)
                  if (size (cnvarr("A", i)) != 0) {
                        get (cnvarr("A", i), &x);
                        x *= x;
                        put (cnvarr("A", i), &x,
                        sizeof(float));
                        }
            }     /* Infinite loop */
    }
```

Note that cnvarrsv() is used instead of cnvarr() whenever the value returned is assigned to a variable. A task using the server might be defined as (in file USER.c):

```
    USER(x)
    char x[];

    {
    float z;

    export (x);
    import (x, 0);
            .
            .
            .
    put (x, &z, sizeof(float));
    get (x, &z);
            .
            .
            .
    }
```

The bootstrap routine must pass a parameter to the user task specifying the name of the fifo it is to use. Suppose we want to create N tasks from the USER program defined above. To do this, bootstrap() might look like:

```
bootstrap()

{
int i, id, strlen();
char *cnvarr();

id = 1;          /* id of created tasks */

for (i=0; i<N; i++)
      mktask ("USER", USER, id++, cnvarr ("A", i),
           strlen(cnvarr ("A", i)) + 1);

      mktask ("SQUARE", SQUARE, id++, NULL, 0);
}
```

The above code uses the predefined C routine strlen() to get the length of the string s (in bytes). A one is added to strlen because of the C convention of appending a null (\0) character to mark the end of character strings.

Finally, note that even though many tasks are created from the USER program, the init() routine need not be used (SQUARE does not use init() because multiple instances are not created). This is because USER does not have any global or static variables. The next example will demonstrate the use of init().

4.2.1. Dot Product

This example computes the dot product of a pair of vectors, A and B. Three programs are used. The first (called T) multiplies two array elements together. For arrays of length N, there are N tasks generated from this program. T uses global variables so that use of the init() routine can be demonstrated. The second, called TDISTR, distributes the data to these N tasks, and the third, TRESULT, collects the results and adds them up to form the dot product in the variable sum. One task is created for each of TDISTR and TRESULT.

The parameter and global data structures are defined first:

```
struct parm            /* format of parameters */
{
char x[10];            /* input fifo name */
char y[10];
};

struct                 /* globals for task */
{
float a, b, c;         /* work variables */
} glob;
```

Task T is defined as follows:

```
T (p)                       /* multiply numbers */
struct parm *p;                 /* x and y are parameters
                            specifying import fifo's */

{
init (&glob, sizeof (glob));
import (p->x, 0);           /* create fifo's */
import (p->y, 0);
export ("C");               /* result fifo */

get(p->x, &glob.a);         /* get operands */
get(p->y, &glob.b);
glob.c = glob.a * glob.b;
put ("C", &glob.c, sizeof (float)); /* result */
}

#define N       100

TDISTR ()
{
float A[N], B[N];           /* vector operands */
int i;
exparr ("A", N);            /* arrays of fifo's */
exparr ("B", N);
      .
      .
      .
  <<input data into A[] and B[]>>
      .
      .
      .

for (i=0 ; i<N ; i++) {  /* loop to send data */
      put (cnvarr ("A", i), A+i, sizeof(float));
      put (cnvarr ("B", i), B+i, sizeof(float));
      }
}
```

Note that the put() routine takes a pointer to the data as an argument, and not the data itself. Thus, "A+i" and "B+i" are used rather than "A[i]" and "B[i]".

And finally, the task which collects the multiplied operands and generates the result, TRESULT, is:

```
TRESULT ()
{
float sum, temp;
int i;
import ("C", 0);          /* import fifo's */

sum = 0.0;                /* add results from T */
for (i=0 ; i<N ; i++) {
      get ("C", &temp);
      sum += temp;
      }
}
```

Note that all the values of A[i] * B[i] collect in fifo "C" of TRESULT. The order in which the data arrives is arbitrary, but this is of no consequence here.

The bootstrap() routine for dot product would look like:

```
#include <stdio.h>

bootstrap()

{
int i, id;
int T(), TDISTR(), TRESULT();
char *cnvarr();
struct parm p;

id = 1;

/* create tasks from program T */

for (i=0; i<N; i++) {
      strcpy (p.x, cnvarr ("A", i));
      strcpy (p.y, cnvarr ("B", i));
      mktask (cnvarr("T", i), T, id++,
          &p, sizeof (p));
      }

/* create other tasks */

mktask ("TDISTR", TDISTR, id++, NULL, 0);
mktask ("TRESULT", TRESULT, id++, NULL, 0);
}
```

Note that the parameters passed to the tasks in the calls to mktask() do not contain pointers. Storage for the strings x and y is allocated within the parm structure itself, rather than using pointers to character strings. The predefined routine strcpy() is used to copy characters into the parameter list. Finally, notice the use of cnvarr() in mktask() to create different names for the tasks defined using the program T.

5. TIMING MODEL

A timing model is clearly necessary if the simulator is to be used for any kind of performance evaluation, whether of communications networks or execution of parallel algorithms. This section will describe the timing model provided by the simulator as well as a high level view of how it works.

Each processor has a clock which keeps track of time for the task running on that processor (recall that each processor is assumed to execute at most one task). In the real multiprocessor, all of the clocks would advance simultaneously with real time, and thus would (at least ideally) indicate the same time. Here however, execution of tasks is time multiplexed on the host computer running the simulator. Thus, at any moment in the simulation run, clocks on different tasks will usually differ. The clock for each task will reflect how far that task has progressed in its computations.

When the simulator begins executing a task, the clock for that task is started. At some later time, when execution is switched to another task, the original task's clock is stopped. A task's clock also advances if the task becomes idle, e. g. when the task must wait for data to arrive from some other processor. All tasks initially begin execution with their clocks set at "0".

Using this clock, the simulator can estimate when events occur in the real system, such as when tasks send messages, when messages arrive (actually, the switch model determines this), etc. The simulator also uses the timing model to accumulate statistics on the task, such as how long it spent executing, and how long it was waiting for data. Such statistics are printed out at the end of each simulation run.

Implementation of the timing model requires that the user compile his programs with a modified version of the C compiler (cc). Let us call this modified version simcc. Simcc is like cc except that in addition to generating VAX assembler instructions for the code being compiled (which are automatically piped into the assembler program), it inserts additional instructions which will increment the clock for the task as it executes. Thus, at least conceptually, execution of each assembler instruction generated by cc is followed by the execution of an inserted instruction which causes the clock for that task to advance by an amount of time equal to the execution time of that instruction. An option to simcc allows the user to assign an execution time to each VAX instruction (i. e. each opcode). Use of this option is described in Appendix IV. The current implementation assigns a default execution time of one microsecond to each instruction if no such assignment is made.

At present, there is one routine available to the user pertaining to the timing model. This is the getclk() routine, which allows a task to read it's own clock. Getclk() takes a single parameter, the id of the executing task. It returns an integer which is the number of microseconds that have elapsed since the task was created (recall that all tasks are assumed to be created at time 0).

6. THINGS TO LOOK OUT FOR

Unfortunately, the simulator could not be designed in a manner that would detect all possible errors the user might commit. Because of this, some rather subtle bugs may be arise. This section outlines some "easily made" mistakes the user might encounter in writing programs for the simulator.

6.1. Global and Static Variables

As pointed out earlier, if multiple instances of a task are created, and if the program for these tasks uses global or static variables, the init() function MUST be used. Since the simulator cannot determine which tasks fall under this category, it cannot detect those which neglect this detail.

Tasks neglecting to use init() when they should may find that variables mysteriously change without being assigned to! This is because the simulator will not save or restore the values of these variables when execution switches from one task to another. Thus, in effect, all of the tasks created from that program use the same sets of variables, allowing one task to change another's. Needless to say, the results are generally unpredictable.

6.2. Use of Cnvarr()

The user should never assign the value returned by cnvarr() to a variable. Subsequent calls to cnvarr() will change the value of this variable, again without the use of an assignment statement. As noted earlier, this is because the result generated by cnvarr() is always stored in a single buffer (actually, each task has its own buffer). Thus, each call overwrites the value generated by the previous call.

If the value is to be assigned to a variable, cnvarrsv() should be used. The result is then stored in a data area created via the calloc() function. It is left to the user to release this storage (via cfree()) when he is done using it.

6.3. Variable Sharing

Two tasks should never access the same variables. Doing so effectively allows tasks to communicate without using the switch model, thus invalidating any statistics gathered during the simulation run.

6.4. Fifo Names

Blanks imbedded within fifo names are significant. Thus, "T ", " T", and "T" are three distict fifo's. Thus, a message placed in export fifo "T" will not be sent to import fifo's " T" or "T ".

7. IMPLEMENTATION

This section describes the implementation of Simon. It is intended for a programmer who is planning to modify or augment Simon, or for someone interested in developing new switch models. A high level description of the operation of Simon is described, as well as details of the manner in which the program is physically partitioned into modules (i.e. files) and the interfaces and interactions between modules. Readers interested in low level details (e.g. the order and types of parameters in subroutine calls) should refer directly to the source code for Simon. It is assumed that the reader is familiar with the user's interface to Simon (seen by the applications programmer) described earlier. A general knowledge of the internal workings of compilers, operating systems, and event driven simulation programs would also be helpful, though not strictly necessary.

At the highest level of abstraction, Simon models the multicomputer system as a sequence of *state transitions*. In Simon, internal variables keep track of the current state of the real system at any instant in time. Transitions are modeled as *events*. Just as the real system moves from state to state following certain well-defined state transitions, Simon's internal variables move from value to value following certain well-defined events. The model used by Simon to represent the state of the multicomputer system, and the events modeling transitions between states are defined here.

Simon is implemented as a discrete time, event driven simulation program. It is *discrete time* (in contrast to continuous time) because the set of times at which state transitions, i.e. events, can occur is a countable (and finite because the precision of the host computer is limited) set. It is *event driven* because Simon executes by processing events, and stops when there are no more events to process. Each event is timestamped to indicate the time at which the event occurs in the real system. Events are processed in time increasing order, and generally causes some modification in the state of the system and generate, or *schedule*, additional events. Thus the simulator always has a backlog of events waiting to be processed. It continually tries to clear this backlog by processing events, one after another, however in doing so, keeps adding to the backlog! When the backlog is finally cleared, the simulation run is complete.

The operation of Simon consists of setting up some initial state, and then processing events to model the state transitions of the real system. Roughly speaking, Simon is composed of the following components:

(1) Events.

(2) Tasks.

(3) Fifos and Virtual Circuits.

(4) Messages.

(5) Timing Model.

A high level understanding of the internal workings of Simon can be obtained by understanding the events defined by Simon, and how they modify the state of the system. These events will be discussed first. The remaining components model the state of the multicomputer system, and will be discussed in subsequent sections.

First let us say something about the overall organization of the Simon program. The source code is distributed across several files, with each file containing the code for performing some logical function. The code for managing tasks for example, reside in the file "task.c". The different files and the functions they perform are listed in Appendix II. Interactions between files is, with only a few exceptions, through procedure calls. There are very few global variables. Although this results in some performance penalty since procedure calls are more expensive than accesses to global variables, it improves the modularity and readability of the resulting code. The emphasis in developing Simon was towards modularity to promote readability and to reduce the difficulty of making changes to the code. Modifications to improve performance have not been attempted at the time this document was prepared, but is certainly an area of future work.

Throughout this document, the convention will be used that when the name of a routine is referred to, the file in which it resides will follow in bold-face print. For example, save() (**task.c**) refers to the save routine, which resides in the file task.c. Routines residing in the switch model will be assumed to reside in a file called swmod.c (the simplest switch model), unless specified otherwise.

7.1. Events

State transitions in the real system are modeled in Simon by events. A data structure called the *event queue* keeps a time sorted list of events waiting to be processed. The main() (**mainsim.c**) program of Simon repeatedly executes the following actions:

(1) Take the next event off of the event queue.

(2) Processes the event by modifying some of Simon's internal variables and/or scheduling other events.

Execution terminates when the event queue becomes empty.

The real multicomputer system consists of a number of autonomous computers, each executing some part of a parallel application program. How does one go about selecting the events for Simon? Removing an event from the event queue, processing it, and then scheduling other events is a fairly time consuming process, so it is unreasonable to go through this scenario each time the application program modifies one of it's variables. Instead, Simon allows each task to execute, modifying it's own variables. Each task executes directly on the host computer. Thus, under this scheme, only one type of event is required: an event which initiates execution of a task. When the simulation begins, the event queue is initialized to hold one such event for each task defined by the application program, and each task is executed to completion, one after the other. In Simon, such an event which marks the initialization of a task is called an "init-task event", and as just described, one is scheduled for each task created by the "bootstrap" program as part of the initialization process.

After a little thought however, it is easy to see that this simple scheme will not work if tasks interact with each other, e.g. by exchanging messages. Clearly a task cannot execute to completion if uses data generated by another task which has not yet begun execution. Thus, other events are necessary to simulate interactions, and to force one task to temporarily stop executing while other tasks are allowed to "catch up". Since all interactions between processors occur via messages, let us define two new events:

(1) The "send event" denotes a processor sending a message.

(2) The "arrive event" denotes a message arriving at a processor.

The send event is generated when a processor executes the put() (**lib.c**) routine. Since a send event denotes a message entering the communication switch, it must be processed by the switch model. As a consequence of processing the send event, the switch model eventually schedules an arrive event indicating that the message has reached it's final destination.

Getting back to our original scheme, we see that the two additional events described above are still not sufficient, since we still need a mechanism to temporarily stop execution of tasks on the host computer so that others can execute and generate the data they need. One more event, called the "get event", is defined for this purpose. When a task needs data from another task, it calls a routine defined by Simon, e.g. the get() (**lib.c**) routine. Simon temporarily stops execution of the task, and schedules a get event to note the fact that this task has been stopped. Simon then goes back to the event queue and processes other events. Eventually, when data arrives for this task, it will be restarted, and allowed to continue execution.

Consider the sequence of events that occur when two tasks are created, A and B. Suppose task A sends a single message to task B. Because of timing variations, the scenario which will now be described is not the only one possible, however it illustrates the interactions of the events described above. Initially, the event queue consists of two events, namely the inittask events for A and B. Suppose that the inittask event for task B appears ahead of that for A in the event queue. Task B begins execution, and request to receive the message by executing the get() (**lib.c**) routine. Simon schedules a get event, and blocks the task. The next event is the inittask event for A, so A begins execution. Task A sends the message by executing the put() (**lib.c**) routine. This causes a send event to be scheduled into the event queue. Task A continues to execute, since as will be seen later, there is no reason to stop it now. Let us assume that A completes execution, returning control back to Simon. Returning to the event queue, we see there are two events: the get event denoting B's request to receive the message, and the send event denoting the message being sent. Let us assume the send event precedes the get event in the queue. Since this event signals a message entering the switch, the switch model is called upon to simulate the transmission of the message. In doing so, the switch model can perform arbitrarily complex (or arbitrarily simple) operations, perhaps scheduling some of its own events. The final result is that it schedules an arrive event which indicates that the message has reached it's destination. Now the queue holds an arrive event and B's get event. The two possible ordering of these events in the queue correspond to the two situations which can occur in the real multicomputer: either the message arrives before B asks for it, implying B does *not* have to wait for it, or the message arrives after B asks for it, and B must become idle, waiting for data.

Consider the first case. Here, the arrive event is ahead of the get event. Processing of this event merely corresponds to Simon noting that the message has arrived. it places the message in the appropriate import fifo. The get event is the processed. The message is removed from the fifo and passed to task B. The task is restarted, and completes execution. Since the event queue is now empty, the simulation is complete.

Consider the second case. Here, the get event precedes the arrive event, indicating the task asked for the data before it had arrived. In the real multicomputer, this corresponds to the task blocking, waiting for data to arrive. Here, Simon marks the task as "blocked", and goes back to process the next event in the event queue. The only remaining event is the arrive event. Simon notices that task B is blocked, waiting for the arrive event, so it unblocks the task, passes it the message, and allows it to continue execution. Again, B completes execution, and the simulation is complete. It is important to distinguish between the two types of "blocking" performed by Simon. In the first, the task was blocked to allow another task to "catch up". This "blocking" is just an artifact of the simulation technique, and does not correspond to a task being blocked in the real system. The second blocking corresponds to a task waiting for data in the real system. Note that in Simon, the first type of blocking is marked by a get event in the event queue, while the second type of block does not have any such events scheduled. The task is later restarted by an arrive event.

With the scheme described above, each task has its own clock which indicates how much time has elapsed since the task began execution. It is important to realize that at any given time in the simulation, different tasks will usually have different values on their respective clocks. This points out another important function of the event queue: namely to ensure that interactions among tasks are simulated in the proper time sequence. Since some task's clocks are ahead of others, it is important not to let a task get "too far" ahead. For example, suppose tasks "A" and "B" each send a message to task "C" using some fifo, say "foo". Clearly, in the real system, either the message from A will arrive at C first, or the message from B will arrive first. Suppose the one from A arrives first. Suppose B was allowed to execute, scheduling a send, and eventually an arrive event for it's message. Now suppose C executed. It executes a get() (lib.c), and erroneously receives the message from B. Only later is it discovered that A generated a message which arrives before the message from B, and that the simulation has been compromised! Happily, the scenario described above cannot occur in Simon. The simulation cannot be compromised if it executes interactions between tasks in the proper time sequence. However, since all interactions go through the time sorted event queue, events residing in the queue cannot be simulated in the wrong order. The only way events can be simulated out of sequence is if an event is scheduled with a timestamp preceding that of the event now being processed. However, this implies an event causes another event which occured earlier in time than the first, e.g. a message arriving before it was sent! Since this cannot happen in the real system, it cannot happen in the simulator, so long as events correspond to occurances in the real system.

From this perspective, the purpose of the event queue takes on a new light. The event queue is no longer simply a "warehouse" of events waiting to be processed. In addition, it acts as a sequencer which ensures that events modeling the real system are processed in the correct time sequence. Thus, it is easy to see when Simon must temporarily block a task, and when it may allow the task to continue executing: when a task's behavior may be affected by an interaction with another task, Simon must block the task and schedule an event on the event queue. Otherwise, the task may be allowed to continue executing. Using this rule, it is now clear that we do not need to block a task executing a put() (**lib.c**), and similarly, we do not need to block a task executing a get if the corresponding import fifo holds at least one message. The latter results from the fact that Simon places data into each import fifo in correct time sequence (since only arrive events cause additions to import fifo's, and all arrive events are guaranteed to be sequenced correctly), so no new message will be placed in front of another message already in the fifo.

Finally, one other event is defined. This is the "initsw" event. This event occurs exactly only once, at the beginning of every simulation run, and is used to initialize the switch model.

In summary, five different types of events have been defined:

(1) inittask: initiate execution of a task.

(2) send: send message into communication network.

(3) arrive: message arrives from the communication network.

(4) get: task pauses because it interacts with another task.

(5) initsw: initialize switch model.

Each of these will now be described in greater detail. When these events are removed from the event queue, the main program calls a particular routine to process that event. The name of this routine and the functions it performs will be discussed.

7.1.1. The Inittask Event

Simon initially places an inittask event in the event queue for each task created in the bootstrap() routine. This event is processed by the routine inittaskev() (**evhand.c**). It calls the routine sttask() (**mainsim.c**)) which sets some flags indicating that the task is to begin execution. Control is then returned to the main program. Note that sttask() does *not* begin execution of the task directly, but only sets some flags representing a request to start the task. After processing each event, the main program checks to see if the event it has just finished processing requested to start a task. If so, it begins executing the task via an ordinary procedure call. Restarting tasks (after being stopped because of interactions with other tasks) also follow this same procedure, i.e. the event handler sets flags to request the restart, and the task is later restarted by the main program. By using this indirect scheme for starting/restarting the execution of tasks, *all* tasks begin execution from the same point in the main program. This ensures that the base of the runtime stack always occupies the same memory location, and it simplifies the task switching operation because all returns from the task (either to stop it temporarily or permanently) resume execution at the same place. After a task stops execution, the next function performed is the removal and processing of the next event in the event queue.

7.1.2. The Send Event

The send event is scheduled by the put() and puts() (**lib.c**) routines, and is processed by the sendev() (**swmod.c**) routine. The functions performed by this routine will be described later in the discussion on switch models.

7.1.3. The Arrive Event

The arrive event is scheduled directly, or indirectly through the sendev() (**swmod.c**) routine, and is processed by arriveev() (**evhand.c**). The event handler must add the message to the destination import fifo. Also, if a task is blocked, waiting for this message to arrive, the sttask() (**task.c**) routine is called to request that the task be restarted.

7.1.4. The Get Event

A get event may be scheduled by any routine which has an outcome depending on some interaction with another task. Currently, the routines which fall into this category are: get(), overfl(), qlength(), size(), and waitf() (**lib.c**). These routines may or may not cause a get event to be scheduled. If it can be determined that the result of this routine is not affected by anything any other task can do, then a get event need not be scheduled. For example, if the get() routine is executed on a fifo which already holds at least one message, then this routine will always return the first message in the fifo, regardless of any other messages that may arrive. The get(), size(), and waitf() routines do not schedule a get event if the import fifo specified by the routine (or in the case of waitf(), at least one fifo) has a message. Otherwise, a get event is scheduled. The overfl() and qlength() routines always schedule a get event because they must wait until all other tasks have "caught up" with the current one in order to determine the result returned by the routine.

Let us now consider the processing of the get event when it appears at the head of the event queue. Five different types of get events exist, distinguished by the routine which caused the event. All types are processed by the getev() (**evhand.c**) routine. If the event was scheduled by the get() routine, the task is restarted if the fifo the get was performed on now has a message. The restart request is performed by calling sttask() (**task.c**), as described above. In this case, the message arrived before the task needed it. Otherwise, the task is marked "blocked" via the blktsk() (**task.c**) routine signifying that the task is idle, waiting for data in the real system. The "waitf" get event is handled in a similar fashion. The remaining get event types all cause the task to be restarted, since they only ask for information which can now be determined with complete certainty (e.g. the number of messages in a fifo).

7.1.5. Other Event Related Routines

A number of other routines are defined relating to the event queue. These routines are defined in the file evnt.c. Each type of event has a separate routine for scheduling an event, e.g. the scsend() routine schedules a send event. All call the schedule() routine which inserts the event into the time ordered event queue. Routines are provided to remove events from the event queue, and to return parameters for the different event types. The initevq() routine initializes the event queue, notevqempty() checks to see if the queue is empty, and the prevq() routine prints the contents of the event queue onto the standard output. This last routine is provided for debugging purposes. Finally, a number of other routines are provided for internal storage management purposes.

7.2. Tasks

Simon maintains information on each task to allow them to time multiplex their execution on the host computer, and to collect statistics. In many respects, the management of tasks on Simon resembles the management of processes in a uniprocessor operating system. First, Simon must be able to create and begin the execution of a task when an inittask event occurs. When a get event occurs, the task must stop execution, and another task must be restarted. Finally, when the task completes execution, it must be destroyed.

7.2.1. Starting Tasks

Creating and beginning the execution of a task is a relatively straightforward operation. When the user-defined bootstrap() routine creates a task by executing the mktask() (**lib.c**) routine, Simon creates and initializes a data structure for the task called a "task control block", or tcb. The information maintained in the tcb includes the name of the task (a user defined character string), the status of the task (e.g. running, or blocked and waiting for data - if the task is blocked, then this means it is blocked in the real system), pointers to saved state information, a pointer to the code for the task, and various statistics on the execution of the task.

The mktask() (**lib.c**) routine specifies, among other things, an integer called the "id" of the task being created. A task's id is unique, and is used internally by Simon to refer to the task. Since task's cannot be created dynamically, task id's are never reused. Once the tcb is set up, the task is "officially" created.

After mktask() creates the tcb by calling taskcr() (**task.c**), an inittask event is scheduled so that the task will eventually begin execution. When the inittask event is processed, the task begins execution through an ordinary procedure call. The inittaskev() (**evhand.c**) handles the event. The mechanics of performing this function were described earlier in the section in inittask events.

7.2.2. Stopping / Restarting Tasks

Stopping and restarting the execution of a task requires some fancy "footwork" on the part of Simon. This is because a certain amount of information must be saved when the task stops executing, and restored when the task is restarted. The information which must be handled this way depends on the host computer, and the program used to compile application programs. Here, we will restrict our discussion to the C compiler (used at Berkeley) and the VAX host computer. In the discussion which follows, it is important that the reader distinguish between a *task* defined by Simon, and a *process* running under the host computer. Simon may contain several tasks, however to the host computer and operating system, it is run as a single user process.

A C program executing on the VAX uses the VAX's runtime stack to hold local variables, as well as information indicating the dynamic structure of procedure calls (so the correct procedure can be executed when the current one returns). A program executing as a single process uses a single runtime stack. Thus, the various tasks running under Simon all share the same runtime stack. When a task stops execution, the current runtime stack must be saved, since this information will be overwritten by the next task. Similarly, when a task is restarted, it's runtime stack must be restored before execution can be resumed.

In addition to the runtime stack, other information must be saved/restored if multiple instances of a task are created from the same program. Besides local variables, each task has a number of global variables. The C compiler assigns each global variable to a private memory location. As long as only one instance of a procedure is created, the memory locations assigned to the globals remain unshared, so it is not necessary to save/restore them when tasks are stopped/restarted. If several tasks are created from the same program however, all will use the same memory locations for their global variables, so it is necessary to save/restore them. Unfortunately, Simon cannot easily determine where these variables are located. Thus, it is up to the user (via the init() (**lib.c**) routine) to tell Simon where its global variables are located if several tasks are created from the same program. It might be noted that dynamic variables created by e.g. the calloc() program need not be saved/restored, even if multiple instances of a task are created. This is because a private copy is created for each task after the task begins execution. Since these memory locations are not shared by other tasks, no saving/restoring is necessary.

Thus, stopping the execution of a task causes the task's runtime stack, and perhaps its global variables, to be saved in the task's tcb (actually a pointer to the saved area is kept in the tcb). This state must be restored before the task can be restarted. Let us consider the situation in which a task does a get on an empty fifo, and must temporarily stop executing. The get() (**lib.c**) schedules a get event, and then calls the save() (**task.c**) routine. Save() saves the runtime stack, and global variables if necessary. Now, we would like to return to the main program at the point where it starts/restarts tasks, so that we can go on to process the next event in the event queue. Save() cannot simply execute a return however, since this will take us back into the get() routine. In order to accomplish this, the return address information in the runtime stack is overwritten, so that instead of returning to save, we pop off all of the stack frames for the task, and return to the main program (recall that all tasks are started/restarted from the same point in the main program). The resume() (**task.c**) performs this covert deed. When it returns, the main program is now executing, rather than the program which called it, save().

When it is time to resume execution of the task, the restore() (**task.c**) routine is called. All starting/restarting of tasks are performed by this routine. To restart the task which did the get(), the saved runtime stack is loaded back into the real runtime stack, and globals are restored if necessary. This is a dangerous business, because in restoring the stack, the stack frame of the restore() routine is overwritten, destroying its local variables! Nevertheless, a return is now executed. Since the stack now reflects the state of the stack at the time at which save() was saving the stack, the return executes as if save did the return, i.e. we are now suddenly back in the get() routine at the point just after the call to save. From the viewpoint of the task executing the get() routine, it simply called save(), and sometime later execution returned from save. Thus, the save routine acts as a no-op to the program which executes it. Of course, in the time between the call to save() and its return, thousands of other events may have been processed.

From the description presented above, it is clear that the saving/restoring of a task is a tricky business using certain "unconventional" coding practices. Modifying the save(), resume(), and restore(), routines should only be done after a thorough understanding of its workings has been achieved. Seemingly harmless changes, e.g. altering the order in which variables are declared in the resume() routine, can have disastrous consequences.

7.2.3. Terminating Execution of Tasks

Terminating the execution of a task is straightforward. Excluding execution errors which terminate the entire simulation run, a task may finish execution by either returning from its main program, or by executing the stop() (**lib.c**) routine. Returning to the main program causes the runtime stack to be popped by normal procedure returns, so control is returned to Simon as a consequence of returning from the procedure call which initially caused the task to begin execution. Execution of the stop() (**lib.c**) routine causes the stack to be popped in a manner similar to that used in saving the stack, except the stack's contents are not saved since the task will not be restarted. In any case, execution resumes as if a return was executed from the initial call which began execution.

7.2.4. Other Task Management Routines

The remaining routines for task management (**task.c**) provide various bookkeeping functions, e.g. keeping track of which tasks are blocked on which fifos, or provide information about tasks, e.g. the name of a task given its id. Finally, the prtask() routine prints out summary information of the execution of all tasks. This routine is normally called only once, at the end of the simulation run.

7.3. Fifos and Virtual Circuits

The fifos represent the interface which carries messages between tasks and the switch model. Each import/export fifo pair of the same name corresponds to a virtual circuit. The fifo.c file contains routines to create import and export fifos, to add (remove) elements to (from) fifos, to test the status of fifos (full, empty, etc.), and to gather end-to-end traffic statistics. The implementation of the fifo management routines is described fully in [John81].

7.4. Messages

The file task.c creates the notion of tasks, and defines certain operations, e.g. saving state, which can be performed on them. Each task is identified with a unique id. Other modules need not know *how* the various functions are performed within the task module, but only need to be concerned with the interface it provides. Similarly, msg.c is a module defining how messages are dealt with.

Each message is identified by a unique id, called the message id. Unlike task ids however, a message id remains unique only as long as the message remains in existence. Once the message arrives at its final destination, the message id may be reused. This is necessary for efficiency reasons, since unlike tasks, messages come and go at a relatively high rate.

When one processor sends a message to another, a message must be created, and placed into an export fifo (actually only the message id is placed into the buffer since this is the interface provided by the message module). The data portion of the message must be copied into a separate buffer. A simple pointer to the task's variable cannot be used because the task might change this variable before the message arrives at it's destination, causing the new value to be transmitted rather than the value which existed when the message was sent.

When the message reaches its destination, the message id is placed into an import fifo. When the receiving task reads the message (via the get() (**lib.c**) routine), the contents of the message is copied into one of receiver's variables, and the message is destroyed. The id is now free to be used by newly created messages.

The simple scheme described above is complicated somewhat by the fact that Simon allows broadcast, i.e. multiple destination communications. If a message is broadcast to several destinations, it is wasteful to create a separate copy and message id for each destination, so only one copy is created. A count is associated with the number of copies existing in the real system. When the message is created, the count field of the created message is set to equal the number of destination processors. When a copy of the message arrives at its destination, the count is decremented. The message is destroyed and its id released when the count becomes zero. Note that this scheme allows different physical messages in the network to have the same message id, however this does not lead to any inconsistencies within Simon.

Each message currently in existence has a message descriptor associated with it. This descriptor is analogous to the tcb allocated to each task. Message descriptors include such information as the size of the message, location of its buffer, time at which it was created, number of copies, etc.

Messages are created and their contents copied into a buffer through the mkmsg() (**msg.c**) routine. Rmmsg() (**msg.c**) destroys messages, and cpmsg() (**msg.c**) copies the contents of the buffer for the message into the receiver's data area. Other routines are provided to extract information about a particular message, e.g. its size.

7.5. Timing Model

The timing model consists of two components. The first is the simcc compiler which inserts instructions into the application program to continuously update a clock as the program executes. The second is the portion of Simon responsible for maintaining the current clock of each task. The implementation of these two components will now be discussed.

7.5.1. Simcc

First, the simcc program is the modified version of the cc program, the front end for the C compiler. The cc program calls the compiler to compile the program. The output generated by the C compiler is an assembly language version of the program being compiled. The assembler then creates an object file. The simcc program inserts a filter between the C compiler and the assembler. This filter is called the ccsf (C Compiler Statistics Filter) program. The output generated by ccsf is the assembly language program input to it with instructions which increment a global variable called "clock_" Conceptually, an add instruction could be inserted before each assembly language instruction to increment clock_ by an amount equal to the time to execute the instruction which follows. With a little thought however, it is easy to see that this scheme can be improved upon: a block of assignment statements in which the only entry is into the first statement, and the only exit is from the last statement may be preceded by an add instruction which increments the clock by an amount equal to the execution time of the block of statements. This latter scheme reduces the amount of overhead required to increment clock_.

This is the approach taken by ccsf. The ccsf program also allows the user to specify the execution time of each VAX instruction, and computes the execution time of a block of instructions by simply computing the arithmetic sum of the execution times of the individual instructions. It turns out however, that one cannot arbitrarily insert add instructions into the instruction stream, since doing so may cause the condition codes on the host computer to change. The ccsf program, which is VAX specific, inserts instructions which first save the condition codes on the runtime stack, then inserts an add instruction, and finally inserts instructions which restore the condition codes. It is interesting to note that the final instruction which restores the condition codes is the "return from interrupt" instruction.

7.5.2. Timing Model Within Simon

The interface between Simon and the task's timing model (as set up by simcc) is the variable clock_. When a task first begins execution, clock_is set to 0. As the task executes, clock_is incremented via the inserted instructions. When the task stops executing (e.g. because it must wait on a get), the value of clock_is save in the task's tcb. Clock_is restored to this value when the task is restarted.

Two timing models exist. In order to avoid anomolous effects caused by floating point arithmetic, clock_is implemented as an unsigned integer (32 bits on the VAX) variable. This means it can grow as large as approximately 4 billion time units. If the time unit is nanoseconds, this allows each task's clock to run up to approximately 4 seconds. With a microsecond time unit, clocks may reach values exceeding 4,000 seconds. The nanosecond timing model is in the file timen.c, while the microsecond model is in timeu.c. Only one of these two files is used on each simulation run.

Except for the switch model, the interpretation of time units is arbitrary, i.e. since the only real time in Simon is that expired by the tasks, no inconsistencies result. In the switch model however, delays refer to some time unit, so this module must know whether the time unit it is to compute for (say) delay is in microseconds or nanoseconds. To handle this situation, each timing module provides a number of conversion routines so to convert microseconds/nanoseconds to whatever time unit the rest of the simulator is using. This gives the switch models the flexibility to use whatever time unit they prefer, while remaining consistent with the rest of the simulator in a transparent fashion.

In addition to conversion routines, the timing modules provide a number of other routines for initializing, setting, starting, stopping, and reading a task's clock. Care must be taken in manipulating these clocks to ensure that time does not flow backwards! Should this occur, the simulator will be compromised, and unpredicatable behavior results.

7.6. Switch Model Interface

Now that the events have been defined, the interface provided to the switch model can be explained. From the discussion above, it is clear that only three event — the send, arrive and initsw events — pertain to the switch model. Minimally, the switch model must provide three routines:

(1) initsw() (**swmod.c**).

(2) sendev() (**swmod.c**).

(3) switchev() (**swmod.c**).

The initsw() (**swmod.c**) routine is called once at the begining of the simulation run. It indicates such parameters as the maximum number of tasks that will ever be created, maximum number of import and export fifos, etc. It is responsible for initializing any data structures used by the switch model. Also, it is here that the switch can input any information about the communication network, e.g. the speed of communication links, topology of the network, etc. An input file pointer is passed to the routine. This pointer points to a file descriptor for the file specified in the command line which invoked Simon if the -s option was used. Otherwise, the standard input is used. The -s flag should *always* be used to input switch model information, since application programs often use the standard input to input their own data.

The sendev() (**swmod.c**) routine is called to process a send event which was scheduled by a task executing either the put() (**lib.c**) or puts() (**lib.c**) routines. The routine is passed information indicating the time at which the message was sent, the task sending the message, a list of tasks which are to receive the message, a message identifier (to be discussed later), and the size of the message in bytes. Note that a particular message may have several destinations since broadcast communications are allowed. Since sending a message corresponds to adding an entry to an export fifo, sendev() (**swmod.c**) must remove the message from the fifo, and schedule an arrive event for each destination. Each arrive event is timestamped to indicate the time at which the message arrived at its destination. The difference between this time the time at which the send event occured indicates the time required by the interconnection switch to transmit the message.

It might be noted that the sendev() (**swmod.c**) routine need not directly perform the functions described above. It is the switch model as a whole which is responsible for doing this. The sendev() (**swmod.c**) could schedule its own internally defined events which lead to removing the message from the export fifo and scheduling the associated arrive event(s). In a store-and-forward network for example, intermediate events might indicate messages going from hop to hop through the network.

The switch model is allowed to schedule and process it's own, internally defined events. When such an event appears at the head of the event queue, the switchev() (**swmod.c**) routine is called. In general, the main program calls switchev() (**swmod.c**) whenever it removes an event from the event queue which it does not recognize (the five events listed earlier are the only events recognized by the main program).

Finally, one other restriction is placed on the switch model: messages sent on the same virtual circuit from the same task *must* be scheduled to arrive in the same order in which they were sent. This is because the user interface specifies that sequentiality is preserved on each virtual circuit. Violation of this rule will result in incorrect computations for application programs which depend on this feature.

7.7. Notes on Porting Simon: Machine Dependencies

Although work is under way to port Simon to other machines, at present, no working version is known to exist on any other machine than a VAX. There are (at least) two portions of Simon which must be modified if it is to be ported to another machine. These are the simcc compiler, and the task swapping mechanisms.

Since the ccsf program scans through assembly language programs and assigns execution times to blocks of VAX instructions, it is clearly VAX specific. A new ccsf program is required for each host computer Simon is run on. The function performed by ccsf remains the same, however different instruction opcodes will have to be recognized, and different instruction sequences will have to be inserted to ensure that the instructions can be inserted without disturbing the original code. All of this assumes that the compiler on the new host generates an assembly language program as an intermediate step. Without this, incorporating a timing model which can be used easily by Simon is much more difficult.

As discussed earlier, the code in task.c for saving and restoring the runtime stack of application programs depends on the format of the stack frame. This depends on the compiler and host machine. If Simon is ported to another machine, these routines will have to be modified to use whatever format that machine uses.

Finally, a third point of difficult may arise if an attempt is made to move Simon to a machine which does not support 32 bit integer arithmetic. This is the timing model. Without 32 bit arithmetic, the granularity of each time unit may become too corse. For example, if 16 bit arithmetic is used, clocks can only reach a maximum value of 65,000. With the low-precision microsecond time unit, clocks may only reaches values of 65 milliseconds. Corser time units are required to reach higher values, reducing the precision of the simulator even further.

REFERENCES

[John81] L. Johnson, "A FIFO Based Communication Scheme for a Multiprocessor Simulator," Master's Report, UC Berkeley (Sept. 1981).

[Kern78] B. W. Kernighan and D. M. Ritchie, "The C Programming Language," , Prentice-Hall Inc., Englewood Cliffs, New Jersey (1978).

APPENDIX 1

This appendix outlines the routines defined within the simulator which are available for use by the programmer.

char *cnvarr (base, i)

char *base; int i;

Convert character string p to an array reference using base name "base" and index value "i".

char *cnvarrsv (base, i)

char *base; int i;

Same as cnvarr() except storage for the result is allocated via ealloc().

exparr (base, n)

char *base; int n;

Create an array of "n" fifo's with base name "base".

export (name)

char *name;

Create a single export fifo called of name "name".

get (name, datap)

char *name, *datap;

Get next message from fifo "name", and store its data where "datap" points.

int getclk (id)

int id;

Returns time on clock (microseconds) for task whose id is "id".

imparr (base, n, maxl)

char *base; int n, maxl;

Create an array of "n" import fifo's with base name "base", each holding up to "maxl" messages (infinite if "maxl" is 0).

import (name, maxl)

char *name; int maxl;

Create a single import fifo with name "name", capable of holding up to "maxl" messages (infinite if "maxl" is 0).

init (glob, lngth)

char *glob; int lngth;

Use when multiple tasks are created from a single program using global or static variables which occupy "lngth" bytes starting at location "glob".

mktask (name, cdptr, id, parm, lngth)

char *name, *parm; int (*cdptr)(), id, lngth;

Create a task called "name", and assign it to task id "id". The block of memory "lngth" bytes long starting at "parmp" is passed to the procedure "cdptr" when the task starts executing.

int overfl (name)

char *name;

Return and reset overflow flag for import fifo "name".

put (name, datap, lngth)

char *name, *datap; int lngth;

Put "lngth" bytes of data starting at location "datap" into export fifo "name".

puts (name, datap, lngth)

char *name, *datap; int lngth;

Same as put(), but send message to self as well.

int qlength (name)

char *name;

Return number of messages currently in import fifo "name".

int size (name)

char *name;

Return size of first message in import fifo "name", or 0 if fifo is empty.

stop()

Terminate execution of task.

int task()

Returns the calling task's id.

char *taskname (id)

int id;

Returns a pointer to the name of the task whose id is "id".

waitf (names, n)

char **names; int n;

Wait for data to arrive in one of the "n" import fifo's specified by "names".

APPENDIX II

This appendix describes the mechanics of using the simulator. All object files may be found in the directory (on ESVAX and CSVAX) ~fujimoto/S. Example user programs are in the directory ~fujimoto/SIMEX. All test programs should be compiled with simcc (~fujimoto/BIN/simcc) rather than cc.

The "S" directory contains a set of object files. To use the simulator, link these object files to your test programs. The simulator's object files and the functions they perform are listed below.

object file	function
evhand.o	event handler routines (called by mainsim)
evnt.o	event queue primitives
fifo.o	fifo management routines
lib.o	user callable routines
mainsim.o	main program (initialization and main loop)
msg.o	message management routines
task.o	task management (saving/restoring state, etc.)
time.o	task timing model
util.o	miscellaneous utility routines
swmod.o	null switch model (zero latency transmissions)
ether.o	ethernet switch model

Note that the last two object files are switch models. Only one of these should be loaded on each simulation run.

APPENDIX III

This appendix describes the parameters which can be specified in the command line when invoking the simulator. The parameters the simulator will accept are listed below.

flag	**type of parameter**	**default**	**meaning**
-nt	integer	200	max nmbr of tasks
-nm	integer	2000	max nmbr of msgs at one time
-nn	integer	1000	max nmbr of different fifo names
-ni	integer	1000	max nmbr of import fifo's
-ne	integer	1000	max nmbr of export fifo's
-t	<none>	off	generate traffic statistics
-o	file	stdout	output file
-s	file	stdin	input file to switch model

For example, to increase the maximum number of tasks to 400 while diverting output to the file foo, enter:

```
%sim -o foo -nt 400
```

APPENDIX IV

This appendix describes how to use the modified version of the C compiler. The simulator's timing model requires that all user programs be compiled with this compiler, called simcc (~fujimoto/BIN/simcc). In addition to all of the options provided in cc, simcc provides a -T option for specifying the execution times of VAX assembler instructions. When specified, the -T is followed by the name of a file which lists these execution times in the format described below. Thus, if your execution times are in the file "time", you would say:

```
%simcc -T time ...
```

to compile your program, where "..." is other options and names of files being compiled. Note that the blank after "-T" must be present. If the "-T" option is not specified, the default execution times will be used (all instructions execute in one microsecond).

The file "time" consists of a list of pairs, with each element of the pair seperated by blank(s) and/or tab(s), and successive pairs seperated by blank(s), tab(s), and/or newline(s) (elements of a pair must be on the same line). The first element of a pair is the mneumonic for a VAX assembler instruction. The second element is an integer specifying the execution time of that instruction in NANOSECONDS. An example "time" file is:

```
pushl   200
calls   200
addl3   200
muld3   200
movd    200
```

which specifies that the five instructions above execute in 200 nanoseconds. Note that mneumonics should all be in small letters, and the execution time is an integer with NO decimal point. All instructions besides these five will be assigned the default execution time (which is still set at 1000, or 1 microsecond).