1992

Si: A Lightweight-Process Interaction Simulation System

Janche Sang

Ke-hsiung Chung

Vernon Rego
Purdue University, rego@cs.purdue.edu

Report Number:
92-086

SI: A LIGHTWEIGHT-PROCESS INTERACTION SIMULATION SYSTEM

Janche Sang
Ke-Hsiung Chung
Vernon Rego

CSD-TR-92-086
November 8, 1992
(Revised January 12, 1993)

Si : A Lightweight-Process Interaction Simulation System

Janche Sang
Ke-hsiung Chung
Vernon Rego*
Department of Computer Sciences
Purdue University
West Lafayette, IN 47907

January 12, 1993

Abstract

The *Si* lightweight-process based system for simulating process interactions is an enhancement to the C programming language in the form of library primitives with sets of predefined data structures. The *Si* system encapsulates an existing lightweight-process library to provide a discrete-event simulation environment supporting the process view. It was developed as a research testbed for investigating methods which support simulations efficiently. Easy extensions and modifications to the *Si* system are a major design objective, accomplished through modularity and layering. This paper describes the system, our experiences with its implementation, and its applicability to simulation modeling. We report on performance measurements of different implementations of the simulation scheduler, and of different algorithms for simulating service disciplines.

* Research supported in part by NSF award CCR-9102331, NATO award 900108 and the Mathematical Sciences Section of Oak Ridge National Laboratory under contract contract DE-AC05-84OR21400 with Martin Marietta Energy Systems, Inc.

1 Introduction

The process view of simulation, developed and refined over the past two decades [6, 8], is a view than enables an analyst to model a discrete-event system in terms of interacting processes. A key advantage of the process-interaction approach to simulation modeling is that model description via processes and their interactions makes the entire modeling activity, from model design to code debugging and execution, require relatively little effort. This is in comparison to the level of effort required in alternate techniques, such as the event-scheduling simulation method, which offers a lower level of abstraction than the process-interaction method.

In the process view of simulation, processes employ a variety of statements to define the flow of entities (transactions, customers, jobs, etc.) through the system. Two or more processes may compete for resources and exchange messages with one another for the purpose of synchronization. The relatively high level of abstraction enjoyed by process-oriented models make model construction relatively effortless. Another benefit is the added flexibility given by application-level processes, and the ease with which process-based models can be scaled to implement simulations of large systems [14]. In this work we describe the design of a process-interaction system based on lightweight processes. $\mathcal{S}i$, a system for $\mathcal{S}$imulating process $i$nteractions, is a process-oriented discrete-event simulation system. It has been designed to enhance the capabilities of the C programming language through a set of primitives which provide a quasi-parallel programming environment.

The primary goal of this work is to develop a research vehicle for conducting experiments and obtaining measurements in empirical evaluations of algorithms used in simulation systems. Such a research vehicle helps in identifying principles of good design and alternatives which contribute toward efficient, accurate, and reliable simulation software. A secondary goal is to present an interface that is simple and straightforward, but sophisticated enough to meet the needs of a wide range of applications. The primitives provided in the $\mathcal{S}i$ skeleton are sufficient for a variety of tasks, including process manipulation and synchronization, random number generation, and statistics collection - all of which enable the description of an application to the maximum extent possible without application-specific details.

There are two main approaches to designing process-oriented simulation software. One

approach is to design a specific simulation language. Several existing languages such as SIMULA[1], GPSS[10], HSL[20], etc. belong to this category. The other approach is to construct and place simulation primitives on top of an existing language. Examples of this approach can be found in [2, 12, 22, 24], where Ada, Extended Pascal with coroutines, C, and Modula II have been used as target languages. We decided to take the second approach with *Si* for the following reasons. First, it requires less effort to extend and modify a library than to design and implement a new language. Second, users tend to be more comfortable with using a familiar and trusted language instead of having to learn yet another new language [7].

One apparently reasonable approach, at least at first glance, toward constructing a process-oriented environment is via traditional UNIX-like processes, each consisting of a single address space. Unfortunately, this approach will inevitably suffer the tremendous overheads associated with process creation, context-switching between address spaces, etc. This overhead-associated deficiency of conventional processes has led to the use of so-called lightweight processes which operate within a single address space and consequently enjoy significantly reduced overheads. Such an approach was taken in the design of CSIM [22], an elegant design which utilizes a built-in lightweight-process library for process manipulation. We decided to base *Si* on the efficient Sun Lightweight-Process library (LWP) [26]. We chose this as a kernel because of its availability in our research environment, its capacity for reliable context-switching and allocation of protected stacks, and its continued development. Further, ease of portability of the system, due to potential conformation with POSIX standards makes the library attractive.

The remainder of the paper is organized as follows. In Section 2 we briefly introduce the Sun Lightweight-Process library. Some design issues are discussed in Section 3, and Section 4 details the implementation of the four major modules in *Si*. Section 4.1 contains a description of the process management, process coordination, resource management and statistics functions, and Section 4.2 contains a simple example of a vacationing-server system implemented in *Si*. In Section 5 we illustrate some of the strengths of the *Si* system, with early measures of performance given in Section 5.1, a simple but remarkable performance enhancement, easily implementable in *Si*, described in Section 5.2, and a brief comparison of different algorithms for simulating service disciplines given in Section 5.3. A short conclusion is presented in Section 6.

2 The Lightweight-Process Library

Lightweight processes are threads of control existing within a single host process, and consequently sharing a single address space. In fundamental structure, a lightweight process is no different from a process; each has its own stack, local variables, and program counter. However, as compared to a process, a lightweight process is lighter in terms of overheads associated with creation, context-switching, interprocess communication, and other routine functions.

The Sun Lightweight-Process (abbreviated as LWP) Library is currently supported at the user level. Because lightweight process operations require no intervention from the operating system (OS) kernel, the LWP library has its own scheduling discipline which is transparent to the OS. Currently, Sun's LWP library supports priority-based scheduling in that the process with the highest priority always has the right to run. The scheduling discipline is non-preemptive, i.e., processes are executed in first-in-first-out (FIFO) order within a priority class. The following LWP library primitives are used in the *Si* system:

- `lwp_create(tid,func,prio,flags,stack,nargs, arg1,···,argn)`
  create a new process with identity `tid`. The process `tid` begins by executing the function `func` with `nargs` arguments `arg1`, ···, `argn` on the `stack`. The priority `prio` and the option `flags` describe the initial state of the new process.

- `lwp_destroy(tid)`
  terminate the process `tid`. If `tid` is the currently executing process, another process with the highest priority will be selected to run.

- `lwp_yield(tid)`
  allow the currently executing process to relinquish control to the process `tid`. The process `tid` should have the same scheduling priority, otherwise the transfer fails.

- `lwp_setpri(tid,prio)`
  alter the scheduling priority of the specified process `tid`. If the updated priority of the process `tid` is greater than the priority of the currently executing process, control is transferred to the process `tid`.

3 Design Issues

The design of the $\mathcal{S}i$ system was motivated by three primary objectives, namely,

- to provide a simple and effective interface which encapsulates the LWP library,
- to provide modularity that supports easy extensions and modifications, and
- to achieve efficient simulation executions.

The $\mathcal{S}i$ system employs the LWP library as its kernel. A layered design enables applications to be created with direct utilization of LWP functions. Therefore, any changes to the LWP library is transparent to the application level. $\mathcal{S}i$ was designed with a modular structure to simplify the replacement and modification of specific parts of the simulator. Modularity is well-recognized as an important concept in system design because it provides for convenient system maintenance. This property is invaluable in software systems like $\mathcal{S}i$ which are used as research testbeds, frequently undergoing modifications to improve existing algorithms or to add new features which improve system functions and performance. For example, two different approaches are provided in $\mathcal{S}i$ to simulate the round-robin service discipline. The modular structure allow us to implement these two algorithms without modifying system structure.

The performance of executing simulations is a major concern in our design. Since simulations are usually time-consuming, the need for significantly reducing the execution time of simulation programs is a critical one. One source of overhead which exhibits the potential to contribute to long execution times is due to context-switching between lightweight processes. Although a lightweight process's context-switching overhead is cheaper than that of its heavier UNIX counterpart, it nevertheless can still contribute in a significant way towards execution time, particularly when the number of context-switches is large. Another source of overhead may be attributed to $\mathcal{S}i$'s layered design, which results in a series of function calls. This is unavoidable, unless the layers are broken up and combined. But the overhead can be minimized if only a thin layering of sparse code is used in $\mathcal{S}i$ to encapsulate LWP primitives.

In general, our objectives in designing and building the $\mathcal{S}i$ system are to build a layered, modular, and efficient simulation system for process-interaction based simulations. Our eventual goal is the integration of the $\mathcal{S}i$ and EcliPSe systems [17, 27]. The latter system provides

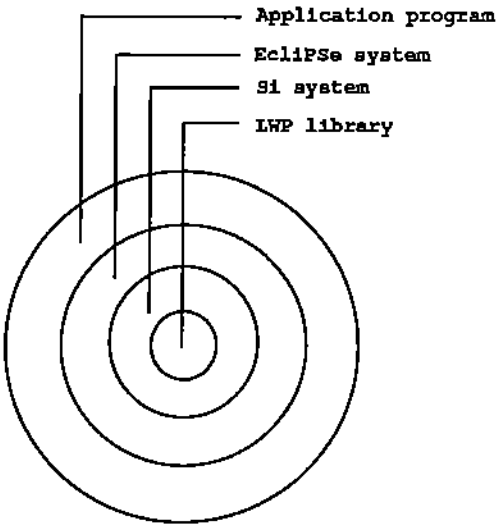


Figure 1: System Layers

high level simulation primitives which enable simulation tasks to be executed in parallel. With such support, the applications developed in *Si* can be conveniently executed on networks of heterogeneous workstations. The requisite layering for such a system is shown in Figure 1.

Scheduler Design

In contrast to the event-scheduling approach [13], where the occurrence of each event is explicitly associated with a corresponding event-handler, in process-oriented simulations event occurrences are viewed as being implicitly generated by processes. In other words, simulation time is advanced through a series of one or more process (control) switches. Such switches in control occur when an executing process either explicitly requests a delay and voluntarily undergoes one, or awaits the triggering of a specific condition and involuntarily subjects itself to a delay in awaiting the occurrence of the condition. In either case, the executing process must be temporarily suspended.

When a process initiates its own suspension, a reactivation record of the process is created and saved. If the suspension is voluntary, such as through a request for a delay of time `t`, the reactivation record of the process is marked with a reactivation time of `clock + t`, and the record containing at least the process's identifier and reactivation time is saved in a special event list known as the simulation calendar. The variable `clock` represents the current value of the simulation clock, and the simulation calendar is either a priority queue or equivalent

structure which yields its contents, in units of process reactivation records, in order of increasing reactivation instants. If the suspension is involuntary, which may occur if a process awaits the occurrence of a certain condition (i.e., event), the reactivation instant is undetermined. Such a process is forced to wait in a special event list associated with the condition, the occurrence of which causes a reactivation record marked with reactivation time `clock` to be created and saved in the simulation calendar.

Regardless of the kind of process suspension that occurs, a context-switch takes place when an executing process undergoes suspension, and control is transferred to a new process. This new process corresponds to that process whose reactivation record in the simulation calendar has the smallest reactivation time. A process that has been reactivated in this way will resume its execution at the statement following its point of suspension.

The task of scheduling processes can be managed by a process scheduler which is in itself a process. When a process is to be suspended, control is transferred to the scheduler. This special process determines the identity of the process which is be to executed next, utilizing the simulation calendar to retrieve the necessary information. Control is then transferred to this new process, and the simulation continues. Such an approach is used, for example, in [12]. An alternate approach is to implement the process scheduler through function invocations. In this way, management of process suspensions and resumption is done by the processes themselves, without resorting to use of an additional process for the task of scheduling. Such an approach is used, for example, in [2]. Naturally, these two approaches have intrinsic differences, leading to different implementations, different application interfaces, and different performance characteristics. For the purpose of comparison, we have implemented both scheduling mechanisms iin $\mathcal{S}i$. Our experiments with both mechanisms, and our conclusions are described in the following sections.

4 Implementation Issues

The $\mathcal{S}i$ system consists of four major components, namely, a process management component, a process coordination component, a resource management component, and a probability and statistics library. Figure 2 depicts layout of components and examples of functions provided within each.

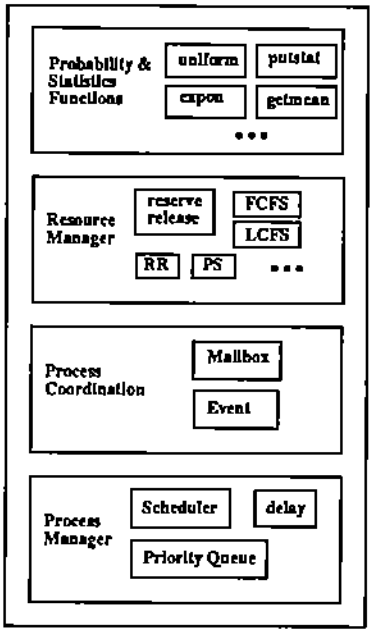


Figure 2: The modules in $\mathcal{S}i$

4.1 Module Descriptions

The process creation task in $\mathcal{S}i$ is achieved through an invocation of the function `si_create(func, nargs, arg1,···, argn)` which encapsulates the LWP function `lwp_create()`. Following the creation of a process, another function invocation `si_insert(E)` is used to ensure that the specified process with reactivation record `E` is reactivated at time `E.clock` by saving the record in the simulation calendar.

Process Management

Transfer of control between specific processes is most easily done with the aid of the `lwp_yield()` function. However, a problem arises because the LWP library provides its own process scheduler which schedules processes based on LWP scheduling priority; the scheduling rule always selects a process with the highest priority to run. If more than one such process exists, the scheduler uses a FIFO policy within the priority class. In building $\mathcal{S}i$ on top of the LWP library, a potential problem arises when an executing process terminates. At this point, the LWP scheduler will transfer control to the highest LWP priority process, one which is not necessarily the process whose execution is imminent in simulation logic, i.e., one whose activa-

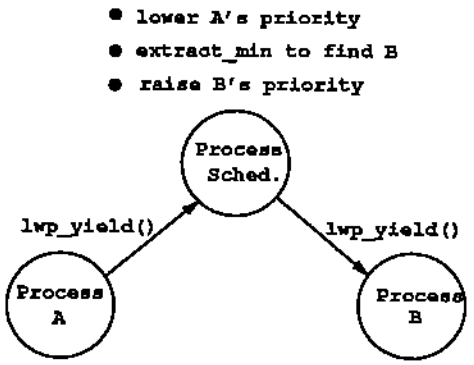
tion record in the simulation calendar has the smallest reactivation time. While an apparent solution is to require a mapping between priorities in $\mathcal{S}i$ and priorities in LWP, the integer priority formats used in LWP preclude the use of this option because $\mathcal{S}i$'s priorities are double precision numbers, representing reactivation (simulation) times. Therefore it is necessary to determine an alternate scheme for transferring control between a terminating process and $\mathcal{S}i$'s scheduler.

Our strategy is to force the LWP scheduler to schedule only a limited number of processes in $\mathcal{S}i$. To achieve this, each process is assigned an LWP priority, either a MINPRIO (minimum poriority) value or a MAXPRIO (maximum priority) value. This principle is stated as follows:

Principle: At any given instant, at most two processes exist with the highest priority value MAXPRIO. One of these is $\mathcal{S}i$'s process scheduler, and the other is a currently executing application-level process.

With this principle, when a currently executing process terminates, the process scheduler obtains control under the LWP library's own scheduling policy since it is the unique process left with priority value MAXPRIO. This solves the control transfer problem. Figure 3 shows the pseudo-code required for the process scheduer. As implied by the principle, an executing process scheduler selects the highest priority process, say process `pid`, from the simulation calendar, using smallest reactivation time as a measure of priority. It raises the LWP priority of process `pid` to MAXPRIO and subsequently transfers control to process `pid`. When the process scheduler is invoked due to process suspension, the scheduler follows exactly the opposite routine. It reduces the LWP priority of the suspended process, say process `pid`, from a value MAXPRIO to a value MINPRIO, prior to saving its reactivation record in the simulation calendar, if necessary. Observe that two distinct processes are created within the `main()` routine shown in Figure 3. The first process executes a `si()` function which the application code treats as a main routine. The second process is the process scheduler whose function it is to execute the `schedule()` routine. Both are created with the highest priority value MAXPRIO. The notation $\mathcal{S}i_p$ is used to emphasize the fact that the scheduler is implemented as a process here.

As indicated in the preceding section, an alternate approach toward implementing transfer




```
schedule()
{
    while(future_event_set is not empty) {
        if previously executing process is alive /* but suspended */
            lwp_setpri(ppid,MINPRIO);
        E = extract_min(future_event_set);
        clock = E.clock;
        lwp_setpri(E.pid,MAXPRIO);
        /* Now only E.pid and the scheduler have highest priority.*/
        si_yield(E.pid) /* resume execution of process pid */
    }
}
main()
{
    int si();
    ···initialization···
    lwp_create(si_process,si,MAXPRIO,···);
    lwp_create(scheduler,schedule,MAXPRIO,···);
}
```

Figure 3: Process Scheduling in Si_p

of control between processes is through function invocations. To demonstrate this point, we implement a different version of the process scheduler, through a small modification of the function `schedule()`. In the remainder of the text, we use the notation Si_f to emphasize the use of function invocations in the implementation of the scheduler. The function `schedule()` in Si_f uses a strategy similar to that used in Si_p to switch control between processes. Since the scheduler in Si_f is no longer a process, an executing application-level process is the only process with the highest priority value MAXPRIO. First, in function `schedule()`, the selection of the next process to execute, say `pid`, is made by accessing the simulation calendar. Following this, process `pid`'s priority is raised to value MAXPRIO, and the currently executing process (i.e., the process seeking suspension) reduces its own priority by invoking function `lwp_setpri()`. Through this priority reduction scheme, control is transferred to the single remaining highest priority process, i.e., `pid`.

The scheme described above allows for transfer of control between a process requiring suspension and a process whose execution is imminent. Unfortunately, a problem arises when an executing process terminates naturally, without either voluntarily or involuntarily requesting suspension. Upon its termination, there is no unique highest-priority process to take control. Left to its own devices the LWP scheduler would give control to the first process in line within the single priority class shared by all other processes. Needless to say, this would result in incorrect simulation logic. To get around this problem, we use an additional function called `si_exit()` which a process must invoke just prior to its termination. The function `si_exit()` invokes function `schedule()` which uses function `lwp_destroy()` to eliminate the currently executing process, effectively a suicide operation. As before, `schedule()` also selects a process whose execution is imminent, so that the invocation of function `lwp_destroy()` causes control to switch to the new high-priority process. These actions are summarized in the pseudo-code shown in Figure 4.

It is worth mentioning that there is yet another way to solve the "natural termination" problem. By modifying the return address of a process, control can be transfered to some function specified in the address when the process terminates naturally. This can be accomplished either by modifying `lwp_create()` to set the address of the function `schedule()` as the return address, or by implementing a new thread creation routine which achieves the same effect [9].

Since our design goals emphasize high level abstractions and layered design, we chose not to adopt this option.

There is another important function in Si known as the `delay()` function. When an executing process decides to suspend itself for `t` units of simulated time, it invokes the function `delay(t)`. This function inserts the invoking process's reactivation record, including its reactivation instant `clock + t`, into simulation calendar. Since the invoking process must undergo suspension, the process scheduler selects as the next process to execute that process in the simulation calendar with the lowest reactivation time. In the Si_p design, transfer of control to the process scheduler is done via the `lwp_yield(scheduler)` action, while in the Si_f design, the scheduler is invoked directly through a call to function `schedule()`. This is described in the pseudo-code shown in Figure 5.

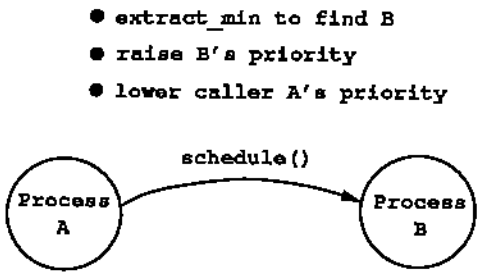
Process Coordination

The Si system provides two distinct coordination mechanisms to support synchronize between processes. One mechanism is through user-declared events, effected by calling `wait_event()` and `set_event()` primitives. A process is suspended if it invokes function `wait_event(e)` because it is forced to wait until event `e` occurs. Event `e` is said to occur when some other process invokes function `set_event(e)`. At this point, all processes waiting for event `e` are reactivated simultaneously. In Si, an event `e` is declared to be of type `Event` and initialized by the `create_event()` function.

The other mechanism for process synchronization is through message-passing. There is a predefined data structure called `Mailbox` which is created by function `create_mailbox`. Messages can be sent and received through the mailbox by using functions `send()` and `receive()`, respectively. The function `send(mb,msg)` deposits the message `msg` in the mailbox `mb`. If there is a process awaiting the arrival of this message, `send()` enables the process to access the message and consequently be reactivated. The reverse function `receive(mb,&msg)` obtains the message from the mailbox. If no message is available, the invoking process has to be suspended until a message arrives. For simplicity, the size of a message is limited to one word (i.e., the size of an integer or pointer). This is patterned after the design of the XINU system [3].

Resource Management

In contrast to processes which are used to model active components of a system, resources



Process switching procedure in Si_f

```
schedule()
{
    while(future_event_set is not empty) {
        E = extract_min(future_event_set);
        clock = E.clock;
        lwp_setpri(E.pid,MAXPRIO);
        if the current process is dying
            lwp_destroy(cpid); /* suicides*/
            /* control transfers to E.pid automatically.  */
        else
            lwp_setpri(cpid,MINPRIO);
            /* control transfers to E.pid automatically.  */
    }
}
main()
{
    int si();
    ···initialization···
    lwp_create(si_thread,si,MAXPRIO,···);
}
```

Figure 4: Process Scheduling in Si_f

```
delay(t)
{
    E.pid = CurrentPID;
    E.clock = clock + t;
    si_insert(E);
    lwp_yield(scheduler); /* or schedule(), for Si_f version */
}
```

Figure 5: The function `delay()`

are used to model passive system objects with mutually exclusive access. In other words, processes request access to resources, use these resources for a certain length of time, and finally release them and proceed with different activities. The *Si* system supports two basic functions for resource access, called the `request(r)` and `release(r)` functions (see Figure 6). The resource object is declared to be of type `Resource` and initialized by a `create_resource()` function. When a resource `r` is occupied, other requesting processes must wait in a queue associated with resource `r`. When resource `r` is released, a suspended process in the front of the queue is given permission to resume, with access to `r`.

In a real application, a variety of queueing disciplines is possible, including first-in first-out (FIFO), round-robin (RR), processor-sharing (PS), etc.. Some, such as the the latter two, are more complicated than others. These complex disciplines utilize different rules in selecting the next process to execute, when faced with a choice. To give analysts a common interface to these disciplines, the *Si* system employs a function `use(r,t)` (patterned after CSIM [22]) to allow a process to utilize a resource `r` for a given length of time `t`. The queueing discipline is specified as a parameter when resource `r` is initialized. For example, the statement

```
r = create_resource(ps)
```

binds resource `r` with the `ps` function which is predefined in *Si* to simulate the PS discipline. This simple interface also provides a level of modularity which lets analysts develop their own queueing discipline in a fairly effortless manner.

Probability and Statistics Functions

The *Si* system provides some necessary random number generation functions such as `uniform()` for generating deviates from a uniform distribution, and `expon(u)` for generating deviates from an exponential distribution with mean u, etc.. In addition, *Si* supports two

```
request(r)
{
    if (resource r is free)
        flag r as occupied;
    else {
        compute required statistics;
        insert current process's pid at tail of queue.
        lwp_yield(scheduler); /* or schedule() in si_f */
    }
}
release(r)
{
    if (waiting queue for resource r is not empty) {
        remove process pid from head of queue;
        compute required statistics;
        si_insert(E); /* with E.pid = pid, E.clock = clock */
    }
    else {
        compute required statistics;
        flag resource r as free;
    }
}
fcfs(r,t)
{
    request(r);
        delay(t);
    release(r);
}
ps(r,t)
{
    ...
}
...
create_resource(fp)
{
    ···initialization···;
    r->fp = fp ; /* fp is a scheduling function pointer */
    ...
}
use(r,t)
{
    (*(r->fp))(r,t);
}
```

Figure 6: Resource Management in $\mathcal{S}i$

types of statistics. The first type is a prefined type, involving data that is implicitly associated with a resource to be automatically collected, sunmmarized and reported. For example, applications can use functions such as `util(r)` and `qlen(r)` to obtain the utilization of and queue-size at resource `r`. The second type is a user-defined type, requiring user-tables for statistics collection. For example, an explicit invocation of the function `putstat(t,x)` inserts datum `x` into a user-defined table `t` which is declared with the type `Table`. The sample mean and variance can be obtained by calls to functions `getmean(t)` and `getvar(t)`, respectively.

At present, the *Si* system also supports functions `reset_resource(r)` and `reset_table(t)` to clear the statistic-collection fields in resource `r` and table `t`, respectively. There are two advantages to using these functions. The first advantage is that these functions can be used to eliminate the effects of the start-up transient in simulations. The second is that they can be used in the regenerative simulation method, where independent samples are obtained from independent cycles of a simulated system [4].

4.2 An Example

To illustrate some of the features of the *Si* system discussed in the preceding section, we present a simple example. The model is that of a single-server queueing system with a vacationing server [11]. In contrast to a stationary server, a vacationing server leaves the service-station to go off on a vacation whenever the queue is found to be empty. If the server returns from a vacation only to find an empty queue, he leaves for another vacation. Whenever the server finds the queue not empty, the server functions just like the server in a stationary-server queue. In our model, server vacation times, customer interarrival times and customer service times are all exponential random variables, and the queueing discipline is FIFO.

In the example shown in Figure 4.2, a total of 10^4 customers is simulated. Customer interarrival time means are 5.0 (IM), service time means are 4.0 (SM), and vacation time means are 2.0 (VM). As mentioned previously, the process `si()` is the first to execute. After initialization of a resource, a mailbox, two events, and two processes `gen_cust()` and `server()` are created. Process `gen_cust()` generates and sends a sequence of customers into the system, using the `expon()` function and the `delay()` primitive to space them apart in simulated time. Process `server()` emulates the vacationing server. If some customer process awaits the resource, i.e.

the status of resource r is BUSY, the server notes the requested service time from a message sent to it by the customer through the mailbox. It ensures that the customer has access to the resource for the requested time, and notifies the customer by setting an event when done. Otherwise, no customer process is found wanting the resource, and the server goes off on a vacation. The functions `request(r)` and `release(r)` maintain a first-come-first-served ordering in giving customers access to the resource. When finished, the last customer sets the event `alldone`, reactivating the process `si` so that it prints out simulation results and terminates.

5 Performance Measurements

In this section we first present some performance measurements obtained from the $\mathcal{S}i$ system, next a simple performance enhancement which can significantly reduce simulation execution time, and finally some specific algorithms for service disciplines. An important reason for pursuing modular design and layering with the $\mathcal{S}i$ system is potential for easy modification, and thus use of $\mathcal{S}i$ as a testbed for new ideas in simulation. In the following subsections we present two different examples of how this approach has proven beneficial. The first example is motivated by recognition of the simple fact that simulation execution performance can be significantly enhanced if the scheduler's interaction with the simulation calendar is slightly modified. The second example demonstrates that new, computation-based algorithms for simulating service disciplines more efficiently than direct process-mapped algorithms, are easy to incorporate in a modular $\mathcal{S}i$ system. The configuration used in our measurements was a Sun SPARC IPC workstation (15.7 MIPS, 8 MB memory, 64KB cache) running SunOS 4.1.

The measured times presented here are averages, and it should be emphasized that the measured averages are not intended to represent absolute performance but rather relative performance for a particular parameter configuration. Thus the comparison of average times is of more interest than a comparison of raw numerical data.

Cost of Operations

In Table 5 is shown a set of measured system overheads for the tasks of process creation and context switching in both the $\mathcal{S}i_p$ and $\mathcal{S}i_f$ subsystems. Also included, for the purpose of comparison, are the corresponding overheads in the LWP library. Not surprisingly, the $\mathcal{S}i_f$ subsystem exhibits less context-switching overhead than its $\mathcal{S}i_p$ counterpart, primarily due to

```
#include <si.h>
#define NMAX 10000 /* no.  of simulated customers */
#define SM 4.0 /* mean service time */
#define IM 5.0 /* mean cust.  interarrival time */
#define VM 2.0 /* mean vacation time */
Resource r;
Mailbox mb;
Event done, alldone;
si()
{
    r = create_resource();
    mb = create_mailbox();
    done = create_event(); alldone = create_event();
    si_create(gen_cust,0);
    si_create(server,0);
    wait_event(alldone);
    printf("%f, %f\n", qlen(f), util(f));
}
gen_cust()
{ int k;
    for(k=0; k < NMAX; k++) {
        delay(expon(IM));
        si_create(cust,1,k);
    }
}
server()
{ float *time;
    while(1) {
        if(status(r) == BUSY) {
            receive(mb,&time);
            delay(*time); /* in service */
            set_event(done);
        } else
            delay(expon(VM)); /* on vacation */
    }
}
cust(k)
{ float *time;
    request(r);
        time = expon(SM);
        send(&time);
        wait_event(done);
    release(r);
    if(k==NMAX)
        set_event(alldone);
}
```

Figure 7: A Vacationing Server Model

Operations	LWP	Si_p	Si_f
Creation + Deletion	660	960	860
Process Switches	70	220	170

Table 1: Latency of Operations (in microseconds)

its use of a function, instead of an additional process, for process scheduling.

Benchmark Measurements

A pragmatic approach toward evaluating the performance of the Si system is through the use of benchmark models, using both ease of model development and execution times as indicators of performance. Though our primary interest is in the Si_p and Si_f subsystems, we have included the CSIM system as a basis of comparison. We chose CSIM because it is a sound C-based simulation system that has been gaining an increasing amount of popularity in the modeling of complex computer systems [23]. In addition, using the same language (i.e., C) to realize models makes the coding of equivalent definitions considerably easier. In this subsection, we develop models for a single-server queueing system, and a multiqueueing system for a token ring local area network. Both models have previously been used as benchmarks in comparing simulation systems [18, 24].

Benchmark I: A Single Server Model

The first model is simple, describing a single-server queueing station. The customer arrival process is Poisson, and customer service times are independent, exponentially distributed random variables. We assume that customers are served in their order of arrival (i.e., FIFO discipline).

In the first experiment, we execute the benchmark program to measure average execution times versus a varying number of simulated customers. Each execution is repeated several times, using different random number seeds in each case, and the average execution time is computed. The traffic intensity ρ of the system is set to be 0.8, ensuring a stable system.

Benchmark II: A Multiple Queue Model

The second model is a little more elaborate, utilizing a multiqueue system with roving server to emulate a token-ring protocol [25]. The multiqueue system represents N independent

computer stations situated on a ring, and the roving server represents the token. Messages made up of packets are generated by each station for transmission to other stations on the ring. A single token is passed unidirectionally from one station to its successor on the ring, to provide stations with a mechanism for conflict-free access to the ring for packet transmissions. A station which acquires the token and has queued packets is allowed to complete transmission of a single packet before relinquishing control of the token to the succeeding station on the ring. It is of some interest to determine queueing characteristics of packets at different stations as a function of ring parameters and station traffic.

The parameters of the model include message interarrival time distributions, and packet transmission time distributions at the different stations on the ring. For convenience, we assume that all interarrival time distributions are identical, each being exponential with mean $1/\lambda$. Also, for convenience, assume that all packet transmission time distributions are identical, each being exponential with mean $1/\mu$. Finally, assume that the token-passing time between consecutive stations on the ring is a small constant, a function of ring delay.

As in the first experiment, we measure the execution time of the simulation for varying numbers of simulated customers. The number of stations N in the system is set to be 10 and each station is configured with $1/\lambda = 1000/6$ and $1/\mu = 10$, which yields a system traffic intensity ρ of 0.6. The time for token-passing is taken to be the constant 1. A similar setup which uses CSIM to model the token ring network can be found in [5].

Empirical Results and Interpretation

The results of both experiments, given in Tables 2 and 3, suggest that the two $\mathcal{S}i$ subsystems are competitive with CSIM in terms of performance. In particular, the $\mathcal{S}i_f$ system outperforms CSIM by up to 20%. We emphasize that this does not imply the $\mathcal{S}i$ system is better than CSIM in all respects. Indeed CSIM is a very stable system, developed and used over several years, while $\mathcal{S}i$ is still an experimental and evolving system.

It is interesting to observe that the $\mathcal{S}i_f$ subsystem consistently outperforms the $\mathcal{S}i_p$ subsystem, even attaining a 40% improvement in execution time for Benchmark II. This is simply due to the fact that $\mathcal{S}i_f$ employs functions to schedule processes instead of using a dedicated process for this task. Consequently, $\mathcal{S}i_f$ suffers significantly less context-switching overhead. Based on the empirical results, we can claim that we have achieved a certain level of efficiency,

	Simulated Customers (×100)				
	10	50	100	200	500
Si_p	1.5	7.4	14.8	29.7	73.8
Si_f	1.2	6.1	12.1	24.0	60.8
CSIM	1.5	7.6	15.1	30.4	75.9

Table 2: Execution time (in seconds) for Benchmark I

	Simulated Customers (×100)				
	10	50	100	200	500
Si_p	3.5	16.7	33.5	67.3	168.5
Si_f	2.1	10.0	19.9	39.7	100.3
CSIM	2.6	13.1	26.1	51.9	130.8

Table 3: Execution time (in seconds) for Benchmark II

one of the main considerations in our original set of objectives.

5.1 A Scheduling Enhancement

Upon examining the code given for the function `delay()` (see Figure 5), it will be clear there is some likelihood that a process reactivation record which has just been inserted into the simulation calendar may very well represent the process whose execution is imminent. In such a case, the insertion of this activation record will immediately be followed by its deletion from the simulation calendar. Clearly, the cost of insertion and subsequent deletion can be avoided if the process in question is recognized to be the process whose execution is imminent. Apart from insertion and deletion savings, unnecessary context-switches between such a process, i.e. one undergoing a potential suspension, and the scheduler may be avoided. Recognition of such a situation entails a comparison operation in which the scheduler determines if the simulation priority of the process in hand is greater than the simulation priority of the highest priority process in the simulation calendar. Because this comparison operation must now be done for each process scheduled for execution, there is a trade-off between the new scheme (in terms of the additional comparison cost) and the old scheme (where there is no comparison cost, but avoidable insertion-deletion pairs of operations and context-switches).

```
delay(t)
{
    E = findmin(future_event_set);
    if( t+clock < E.clock) {
        /* no need to insert; process continues to execute */
        clock += t;
        return;
    }
    else {
    ··· original code in delay() ···
    }
}
```

Figure 8: Add an extra checking to `delay()`

In order to obtain a rough assessment of the frequency of such unnecessary insertion-deletion actions of process reactivation records, we measure the ratio of such occurrences to the total number of times that function `delay()` is invoked. Recall that invocation of function `delay()` initiates a process's suspension by a control-switch from the process to the scheduler, and a subsequent control-switch to a new, or the same, process. Table 4 contains percentages of such unnecessary actions for both Benchmark programs. Surprisingly, the avoidable cost can be seen to be as high as 78% for the multiqueue model.

This simple idea is incorporated in Si through a small modification of the original code in the function `delay(t)`. An additional function, `findmin()`, is required for actually performing the comparison. It determines if the highest priority process in the simulation calendar, with reactivation record `E` and reactivation time `E.clock` is to be given control before or after the function that invoked `delay()` and requires control at simulation time `clock + t`. Clearly, if the quantity `E.clock` is smaller, then the currently executing process must be suspended; otherwise, the savings will include an insertion, a deletion, and two context-switching actions in the case of Si_p. In the latter situation, the process continues execution, upon an immediate return from function `delay()`. A succinct description of this scheme is given by the pseudo-code shown in Figure 8.

Using the modified `delay(t)` function, we repeat the experiments described above to obtain average execution times for the two benchmark models. The results, shown in Table 5, indicate

a significant improvement in performance as compared to the previous results (see Tables 2 and 3). The impact of this modified piece of code on the performance of the $\mathcal{S}i_p$ subsystem is larger than its impact on the $\mathcal{S}i_f$ subsystem. This is largely attributable to the fact that both, process switching overheads and simulation calendar overheads are reduced in $\mathcal{S}i_p$, while only simulation calendar overheads are reduced in $\mathcal{S}i_f$. Though the difference in performance between the $\mathcal{S}i_p$ and $\mathcal{S}i_f$ subsystems decreases with the comparison modification, the $\mathcal{S}i_f$ subsystem is still a consistently better performer than the $\mathcal{S}i_p$ subsystem.

The performance improvement to be had from the additional comparison operation depends on the frequency of the desired property (i.e., the currently executing process must continue execution without incurring calendar and context-switching overheads) relative to the actual cost of performing the comparison for every process that invokes the `delay()` function. It should be apparent that the more frequent the condition supporting the property, the larger will be the savings. Also, the larger the ratio of this count to the number of times the `delay` function is invoked, the more gain is to be had by adding this comparison test. Using f to denote the frequency with which the condition is true, C_{test} and $C_{overhead}$ to represent the costs of comparison and overhead, respectively, and r the ratio $C_{test}/C_{overhead}$, the enhancement is beneficial whenever

$$C_{overhead} > C_{test} + (1 - f) \times C_{overhead} \tag{1}$$

or equivalently, whenever

$$f > \frac{C_{test}}{C_{overhead}} = r. \tag{2}$$

In support of this modification, it is known that when scheduling distributions (which dictate how reactivation-times are dispersed in the simulation calendar) are mixture distributions, reactivation-times tend to pile up towards the beginning of the simulation calendar [15]. Because such scheduling distributions are realistic, the modified scheme is likely to almost certainly yield reduced execution times for most applications. A detailed analysis of the savings given by this method can be found in [19].

5.2 Round-Robin and Processor-Sharing Algorithms

Service disciplines such as first-in-first-out (FIFO), round-robin (RR), processor-sharing (PS), etc. are functional parameters of service stations in queue-based simulations. As systems that

	saved/total
Benchmark I	$27802/100000 \approx 28\%$
Benchmark II	$337954/433437 \approx 78\%$

Table 4: Ratio

		Simulated Customers (×100)				
		10	50	100	200	500
Benchmark I	Si_p	1.4	6.9	13.6	27.2	67.9
	Si_f	1.2	5.9	11.8	23.5	59.2
Benchmark II	Si_p	1.9	9.5	18.9	37.6	94.2
	Si_f	1.7	8.6	16.9	34.0	84.7

Table 5: Execution time (in seconds) for Model I and Model II (improved)

are designed and built become increasingly complicated both in functionality and description, we are faced with a choice between hypothetically weak (in the sense of model assumptions) analytic models and conclusively weak (in the sense of execution data) simulation models. In most instances of practical interest we usually have little choice but to rely on good simulation models to answer questions related to system performance. Hence efficient techniques for implementing simulation algorithms, including algorithms for service disciplines, are useful as simulation execution enhancements.

In the round-robin (RR) service discipline, a job is serviced for a single quantum q at a time, sharing the service resource with other jobs undergoing the same service allocation. If the remaining service time required by a job exceeds the quantum size q, the job's processing is interrupted at the end of its quantum and it is returned to the rear of the queue, awaiting the service quantum it will receive in the next round. Instead of taking a naive approach which simulates the round-robin discipline by physically switching control from one process to another, we propose a computational scheme which is based on predicting departure instants of serviced jobs leaving the pool of queued jobs. The computed departure instant of a particular job may be invalidated by one or more newly arriving jobs, i.e., one or more arriving after the corresponding departure event is scheduled, but before the departure event can occur. This is

because the server must now attend to one or more previously unaccounted for job arrivals, and decrease the amount of attention it planned on giving to jobs already in the system prior to the arrival of the new job(s). Consequently, a scheduled departure event that has been invalidated in this manner must be cancelled, and an updated departure instant for the same or another job scheduled in its place. In addition, it is necessary that the remaining service time requirements of each job in the system be adjusted whenever an arrival event or a departure event occurs.

The processor-sharing (PS) discipline schedules jobs as if the server were processing all the jobs in the facility queue simultaneously. That is, each job receives service for a time which is inversely proportional to the number of competing jobs in the pool. A naive algorithm for simulating the PS discipline is based on a computation and prediction method which doles out an equal amount of service time to all jobs in the pool. This approach suffers in that it is computationally demanding, especially when there are frequent updates of the remaining service time requirements for each job in the pool. To alleviate the computational requirement to some extent, we propose a lazy-update algorithm which accumulates the requisite amount of updating required in a special variable, instead of directly performing the update on all jobs exhaustively. When the departure time of the earliest job to leave the pool is to to be computed, the value resident in the special variable is be subtracted from the remaining service time of the next job to depart, reflecting the true remaining service time. Because of this modification, the only required operations for handling the pool are $INSERT$ and $EXTRACTMIN$ primitives. By combining such infrequent updates with the use of an efficient priority queue data structure, the lazy-update algorithm can be shown to reduce the $O(n^2)$ complexity of the naive algorithm to $O(n \log n)$

We conduct experiments to measure the execution performance of the proposed RR and PS algorithms, respectively, in comparison to the naive algorithms. We use the single server model for this experiment. In Tables 6 and 7 it can be seen that the proposed algorithms perform well compared to the naive algorithms. The RR discipline exhibits larger performance differences between the naive and computational approaches because of the large number of context-switches required by the former. The differences are not particularly significant for the PS discipline because both approaches exploit computation. A detailed description of these

	Simulated Customers (×100)				
	10	50	100	200	500
Naive	3.5	17.2	34.1	67.8	169.5
Computational	1.7	8.2	16.3	32.8	83.0

Table 6: Comparison of Execution time (in seconds) for RR algorithms

	Simulated Customers (×100)				
	10	50	100	200	500
Naive	1.8	8.5	16.5	34.2	86.0
Lazy-Update	1.5	7.4	14.9	29.3	73.2

Table 7: Comparison of Execution time (in seconds) for PS algorithms

algorithms and related experiments can be found in [21].

6 Conclusions

Our experiences with the the design and implementation of the $\mathcal{S}i$ system have been amply rewarding. The support of a very reliable lightweight process library has greatly reduced the effort required in building an efficient, experimental simulation test-bed. During the early stages of design and implementation, we faced several different design choices which were at times not altogether consistent with our design principles and objectives. For example, the process scheduler could be implemented by either a dedicated process or by function invocation, with each method leading to different versions of $\mathcal{S}i$. While the former suffers overheads typically associated with process switching and control, the latter suffers in terms of application-interface inelegance, in that an extra function call is required of a terminating process. Our experiences suggest that, despite the use of lightweight processes, context-switching costs are not insignificant. This is clearly seen in the use of the simple comparison check which eliminates certain unnecessary context switches during simulation execution, improving the performance of both the $\mathcal{S}i_f$ and $\mathcal{S}i_p$ subsystems.

With the $\mathcal{S}i$ system we have been successful in developing a software infrastructure which provides an ideal experimental environment. Central to this capability is the modular design

philosophy, which unambiguously defines interfaces to the system's functional components. New algorithms can therefore be implemented, tested, and experimented with, almost effortlessly, requiring only the simple need to match interfaces. We have implemented new algorithms in $\mathcal{S}i$ to simulate the round-robin and processor-sharing disciplines, and both the ease of algorithm incorporation in $\mathcal{S}i$, and the performance of $\mathcal{S}i$ with the new algorithms has been excellent.

A current disadvantage of the $\mathcal{S}i$ system is its lack of portability. This is due to its implementation on a lightweight process library which is machine dependent. However, since $\mathcal{S}i$ adopts a layered design, and because almost all lightweight process libraries support the universal functions of process creation, switching (i.e., yielding), and deletion, the effort required in porting $\mathcal{S}i$ to rest on top of another lightweight process library is minimal. This effort will need to focus only on the process manipulation layer, which is the innermost layer in the $\mathcal{S}i$ system. Therefore, such a modification will be transparent to the application-layer. With the advent of the standard POSIX threads package (e.g., [16]), we believe that the $\mathcal{S}i$ system will be easily portable to any POSIX supporting machine.

References

[1] G. Birtwistle, O. Dahl, B. Myrhaug, and K. Nygaard. *Simula Begin.* Van Nostrand Reinhold, New York, 1979.

[2] G. Bruno. Using Ada for discrete event simulation. *Software-Practice and Experience*, 14:685–695, 1984.

[3] D. Comer. *Operating System Design The XINU Approach.* Prentice-Hall, 1984.

[4] M. A. Crane and A. J. Lemoine. *An Introduction to the Regenerative Method for Simulation Analysis, Lecture Notes in Control and Information Sciences.* Springer-Verlag, New York, 1977.

[5] G. Edwards and R. Sankar. Modeling and simulation of networks using CSIM. *Simulation*, 58:2:131–136, February 1992.

[6] J. B. Evans. *Structures of Discrete Event Simulation.* Ellis Horwood Limited, Market Cross House, England, 1988.

[7] D. Ferrari. *Computer Systems Performance Evaluation.* Prentice-Hall, 1978.

[8] W. R. Franta. *The Process View of Simulation.* North-Holland, Amsterdam, 1977.

[9] P. Gautron. Porting and extending the C++ task system with the support of lightweight processes. In *Proceedings of USENIX C++ Conference*, pages 135–146, April 1991.

[10] J. D. Henriksen and R. C. Crain. *GPSS/H User's Manual*, 1983.

[11] L. Kleinrock. *Queueing Systems, Vol I: Theory.* John Wiley & Sons, New York, 1975.

[12] J. Kriz and H. Sandmayr. Extension of Pascal by coroutines and its application to quasi-parallel programming and simulation. *Software-Practice and Experience*, 10:773–789, 1980.

[13] A. M. Law and W. D. Kelton. *Simulation Modeling and Analysis.* McGraw-Hill, 1982.

[14] M. H. MacDougall. *Simulating Computer Systems: Techniques and Tools.* The MIT Press, Cambridge, Massachusetts, 1987.

[15] W. McCormack and R. G. Sargent. Analysis of future event set algorithms for discrete event simulation. *Communications of the ACM*, 24(12):801–812, December 1981.

[16] F. Mueller. A library implementation of POSIX threads under UNIX. In *Proceedings of the Winter USENIX Conference*, 1993.

[17] H. Nakanishi, V. Rego, and V. Sunderam. Superconcurrent simulation of polymer chains on heterogeneous networks. *1992 Gordon Bell Prize Paper, Proceedings of the Fifth High-Performance Computing and Communications Conference: Supercomputing '92*, November 1992.

[18] V. Rego. A note on two simulation benchmarks. *Simulation Digest*, 20(3):26–35, 1990.

[19] V. Rego and J. Sang. A remarkable simulation scheduling enhancement. Technical report, Computer Sciences Department, Purdue University, 1992.

[20] D. Sanderson, R. R. R. Sharma, and S. Treu. The hierarchical simulation language HSL: A versatile tool for process-oriented simulation. *ACM Trans. on Modeling and Computer Simulation*, 1(2):115–153, April 1991.

[21] J. Sang, K. Chung, and V. Rego. Efficient techniques for simulating service disciplines. Technical report, Computer Sciences Department, Purdue University, June 1992.

[22] H. D. Schwetman. CSIM: A C-based process-oriented simulation language. In *Proceedings of the 1986 Winter Simulation Conference*, pages 387–396, 1986.

[23] H. D. Schwetman. Using CSIM to model complex systems. In *Proceedings of the Winter Simulation Conference*, pages 246–253, 1988.

[24] R. Sharma and L. L. Rose. Modular design for simulation. *Software-Practice and Experience*, 18:945–966, 1988.

[25] J. Spragins. *Telecommunications Protocols and Design.* Addison Wesley, New York, N.Y, 1991.

[26] Sun Microsystems, Inc. *SunOS Programming Utilities and Libries: Lightweight Processes*, March 1990.

[27] V. S. Sunderam and V. Rego. Eclipse: A system for high performance concurrent simulation. *Software-Practice and Experience*, 21(11):1189–1219, 1991.