

---

# Efficient implementation of multiprocessor scheduling algorithms on a simulation testbed



Jorge R. Ramos<sup>\*,†</sup> and Vernon Rego

*Department of Computer Sciences, Purdue University, West Lafayette, IN 47907, U.S.A.*

---

## SUMMARY

**A layered and modular approach to implementing a process-oriented simulator testbed is described. The simulation kernel is supported by a threads library and is, in turn, capable of supporting distinct domains or application areas for various applications. The testbed offers an implementation methodology for testing novel simulation algorithms at the domain level, without interfacing with the kernel. To demonstrate its utility, a novel algorithm for simulating multiprocessing with round-robin scheduling is presented. The algorithm is more complex than the naïve round-robin implementation in use, but offers significant performance improvement. Copyright © 2004 John Wiley & Sons, Ltd.**

KEY WORDS: process-oriented; simulation; round-robin; threads; multiprocessor; scheduling

## INTRODUCTION

Simulation techniques offer a well-accepted methodology for the study of complex stochastic systems, including the empirical derivation of job waiting-times in uni- and multiprocessor systems under various scheduling strategies. While prohibitive expense in building test systems motivates studies of models, such models are often computationally intensive. Thus, there is a need for a framework that enables the implementation of novel and efficient models with minimum effort. Both issues can be addressed with the help of a simulation testbed. The goal is a layered design that hides detailed kernel operations from a domain layer in which interesting algorithms may be implemented,

---

\*Correspondence to: Jorge R. Ramos, Department of Computer Sciences, 250 N. University Street, Purdue University, West Lafayette, IN 47907-2066, U.S.A.

†E-mail: jrramos@cs.purdue.edu

Contract/grant sponsor: DoD; contract/grant numbers: DAAG55-98-1-0246 and PRF-6903235

while simultaneously offering sufficient flexibility for tests. The idea is to enable an analyst to implement, test and compare new algorithms with some ease.

Among various techniques developed over recent decades, process-oriented [1–3] simulation has proven to be a powerful and reliable methodology. In a process-oriented simulation, discrete events instantiate processes which interact; appropriate statements direct the flows and interactions of simulated entities and objects such as jobs, CPUs, queues, etc. An advantage of this technique over others (e.g. event-based simulation) is its high level of abstraction, which enables an analyst to focus on the model instead of implementation detail.

To support a high level of abstraction, process-oriented simulators may be built using three layers: threads, kernel and domain/application. The threads layer creates and manages the processes used by the simulator and supports the kernel layer; the latter implements simulation entities and logic, and also provides a set of primitives to the domain layer. The domain layer is used by analysts to implement simulation models. In this way, application-level users are freed from low-level programming detail.

Examples of process-oriented simulation systems that offer simulation primitives implemented with C-language threads include CSIM [4], Si/SOL [5] and PARASOL [6]. Lightweight threads [7] offer a simple and efficient facility for simulation-process support. The SOL system's modular design has proven useful as a testbed for testing efficient simulation algorithms, just as PARASOL's design has proven useful for implementing parallel simulation processes in shared and distributed memory. In both CSIM and Si/SOL, simulation primitives are restricted to single-server (i.e. uniprocessor) queues or standard multi-server queues, where a single FIFO/LIFO or priority-based queue of jobs is serviced. It is impossible to study new algorithms without versatile kernel-level support and appropriate library primitives in the domain layer.

Consider a single pool (queue) of jobs each of which requires service from a set of processors (servers), as shown in Figure 1. The processors may service distinct jobs in parallel; for simplicity, all job service times are assumed to be the same. If the number of jobs is larger than or equal to the number of processors  $N_{PROC}$ , the first  $N_{PROC}$  jobs in the pool are processed simultaneously. When the number of jobs in the pool is less than  $N_{PROC}$ , however, all jobs are serviced while ( $size\_of\_pool - N_{PROC}$ ) processors idle for a single job quantum. In this paper,  $N$  is used to denote the number of jobs that receive service or the number of processors that service the jobs, so that  $N$  is either  $N_{PROC}$  or  $size\_of\_pool$ , depending on their relative sizes.

The present study is motivated by two objectives. The first is to show how new algorithms may be supported at the domain layer without a redesign of simulation kernel or threads-support layers, if the kernel is sufficiently general. As a second motivation, in support of the first, a non-trivial multiprocessor round-robin algorithm is implemented and tested within the domain layer. Indeed, although this algorithm was chosen as an example, it has turned out to be interesting in its own right because it offers significantly improved performance in both the uniprocessor [5] as well as the multiprocessor cases. Here, the focus is on the multiprocessor version of the algorithm to show two things: first, that the testbed offers a capacity for primitives that easily support new algorithms, and second, that the new algorithm is a significant improvement over a naïve implementation of multiprocessor round-robin scheduling.

In the following sections, the design and implementation of the process-oriented simulator is briefly reviewed as a basis for understanding the rest of the paper. The algorithms section describes how a uniprocessor round-robin algorithm—already an improvement over the naïve round-robin algorithm offered by process-oriented libraries like CSIM—can be generalized to accommodate

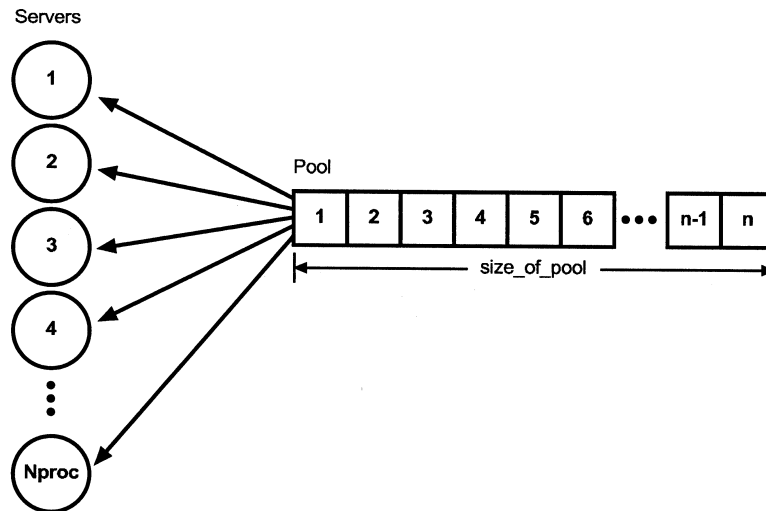


Figure 1. Multiprocessor job-scheduling.

multiple processors. The relative ease with which such an algorithm can be implemented on the SOL testbed serves to demonstrate the utility of a layered, threads-based framework for simulation. Instead of naïvely assigning jobs to processors, the proposed algorithm computes and schedules multiple job-departure times with the aid of appropriate data structures. The benchmark tests show that a significant reduction in simulation execution time can be achieved.

### PROCESS-ORIENTED SIMULATORS: AN OVERVIEW

Consider a brief overview of process-oriented simulation. A discrete-event simulation moves forward in time by updating a clock to the time of the next event and invoking an event handler for the processing of event logic. In a threads-based simulator, event handlers are threads; a handler invocation may involve running a new thread or resuming a suspended thread, where a thread is a simulation process. Threads offer an effective mechanism for supporting interacting processes in simulators. To enable the scheduling of processes in simulation time, each process is bound to an event-activation record which contains, at the very least, a pointer to a thread containing process logic and the time at which this thread must run (i.e. the event time). The event record may contain a number of other items pertinent to the simulation: remaining execution time for a job whose execution has been preempted by a higher priority job, characteristics of each job (more appropriately stored in local variables when processes are threads), simple thread execution statistics, cancellation fields for threads earlier scheduled to run but later must be cancelled, etc.

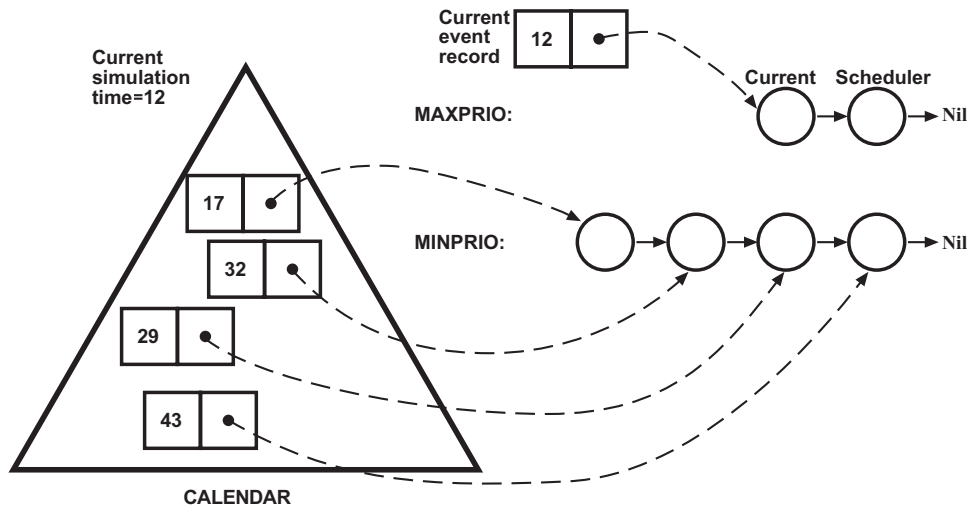


Figure 2. Simulation calendar, event records and process priorities.

When a simulation process is suspended or has run to completion, a special process or function called the simulation scheduler is invoked. The latter examines processes in the future event list (calendar) and elects to run the process with highest priority: the process with the smallest time-stamp value. It relies upon another scheduler—the threads scheduler—to maintain a priority ranking of threads. Indeed, once the simulation scheduler has chosen the next runnable process, it informs the threads scheduler of its choice by manipulating thread priorities.

Each simulation process is given one of two priority values: MINPRIO (minimum priority) or MAXPRIO (maximum priority). The highest priority thread runs until it blocks, and other threads at this priority await their turn in a FIFO queue tied to this priority level. In a process-oriented simulator only the simulation scheduler and the currently running or runnable process may hold the highest priority; all other processes are made to hold priority MINPRIO and, besides waiting in a FIFO threads-system queue at the MINPRIO level, also wait in the simulation calendar until chosen to run.

The calendar is typically a priority queue or similar data structure that offers efficiently retrievable activation records based on the minimum time stamp. An example of a simplified simulation calendar is shown in Figure 2, along with event records and processes. Each event record has two fields, the remaining service time and a pointer to its corresponding process. As shown in the figure, the order of processes in the threads queue is not critical; it is the calendar which ensures that processes run in the right order.

### Process scheduling

A simulation process yields control of the CPU in one of two ways: via an explicit 'delay' request or, when blocked on some condition, an indefinite delay. The process is suspended, and control is

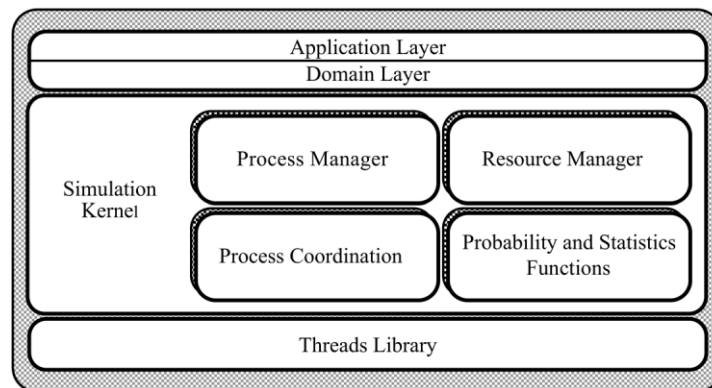


Figure 3. Software layers.

given to the next runnable process. The delay function is a special simulation function, such as the `hold()` primitive in CSIM or the `delay()` primitive in Si; it is invoked with time  $t$  as parameter. The activation record of the running process is loaded with a reactivation time of `clock + t` and saved in the calendar, with `clock` being the current simulation time. When suspension is involuntary—a process is forced to wait in a queue for a busy server, or when it awaits the occurrence of a specific event—the reactivation time is undetermined; the reactivation record is stored in a special event list associated with the condition. Upon occurrence of the condition, the reactivation record is removed and placed in the simulation calendar with current time as the reactivation time.

Transfer of control between processes is effected by the simulation scheduler, requiring thread context switches for each transfer. The scheduler extracts a minimum-timestamp activation record from the calendar, sets the clock to this time and yields control to the process indicated by the activation record. The running process is suspended when it lowers its priority to `MINPRIO`. The scheduler yields control to the activated process after raising its priority to `MAXPRIO`. The latter resumes its execution at the statement following its last point of suspension. Process alternation continues until the calendar is empty or some termination condition is met, causing the simulation to end. While a scheduler may be implemented either as a process or a function, a function-based scheduler is used in SOL because it eliminates one context switch per process transfer, reducing simulation run time at the expense of some complexity. Details on aspects of scheduling can be found in [5].

### Layered architecture

The layered, modular architecture is shown in Figure 3. This approach provides for data abstraction and protection, making it easier to modify and maintain the system. The layered design achieves two desirable goals: encapsulation for the threads library and potential for flexible experimentation with simulation algorithms at the domain layer, free of simulator kernel details. Model code is developed at the application layer, using primitives offered by the domain layer. Portability is readily

achievable because the underlying ARIADNE [7] threads library is portable. Finally, the layering enables easy modification, updates or optimizations to domain libraries so that feature and performance enhancements can be made without affecting the threads or kernel layers below.

### Threads layer

The bottom layer is supported by the ARIADNE threads library; threads run as functions on distinct stacks within a single address space on a host process that supports a virtually unlimited number of threads. Threads have their own stacks, local variables and program counter, with significantly lower creation, context switching and inter-thread communication costs than heavy-weight processes. Further, threads run at user level, with their own priority-based scheduling, requiring no special OS support. Examples of similar systems include the Solaris Thread Library [8] and POSIX Threads [9].

While the Si system was implemented on a hardware multiprocessor, the SOL system exploits ARIADNE's portability; an additional layer of wrapper functions—primitives that encapsulate library-specific functions—allow portability to any threads library. An added advantage is simplicity in system modification: instead of having to access the kernel and domain layer code for specific commands such as `lwp_create()` (thread-create) when making modifications, it suffices to modify the wrapper. The key primitives are easily mapped into various thread functions for different libraries, including:

- `init()`: initialize the threads system;
- `create(tpid)`: create a thread with process id `tpid`;
- `yield(tpid)`: yield control to a thread with process id `tpid`;
- `setpri(tpid, tprio)`: set the priority of thread with process id `tpid` to an integer `tprio`; if `tpid = SELF`, the invoking thread resets its own priority;
- `exit()`: terminate.

### Simulation kernel

To enable the simulation of multiprocessor systems, the simulator must offer sufficient generality so that multiple events can be scheduled simultaneously and processed efficiently. Though such algorithms can be complex, as is evidenced by the test algorithm, the scheduler itself only requires minimal changes to accommodate generality for various algorithms.

#### *Process management: facility for enhancement*

Each simulation process is supported by a thread, and application-level jobs are modeled by processes. Henceforth, the term process and job are used interchangeably. This module contains the functions that manage processes and transfer the control between them, i.e. the `yield(tpid)` primitive, or functions that change priorities. The scheduler function, called `schedule()`, is invoked when an application-level process running at priority `MAXPRIO` terminates or changes its priority to `MINPRIO`; this is done with the help of the `setprio(tpid, tprio)` primitive. The scheduler selects a process with id `tpid` indicated by an activation record with minimum time stamp. It raises `tpid`'s priority to `MAXPRIO` and then gives it control. A basic simulation scheduler dispatches these events through naïve scheduling, involving a context switch between the processes.

In a simulation of multiprocessing, multiple reactivation records with identical reactivation times are routinely scheduled. This suggests multiple and simultaneous departures and, if batch arrivals are allowed, multiple and simultaneous arrivals. For the scheduler this involves context switching between many simultaneously scheduled threads, with high cost and severe inefficiencies. To avoid this, the scheduler can be made to prioritize simultaneous events based on type, so that more efficient queuing algorithms may be implemented in the domain layer.

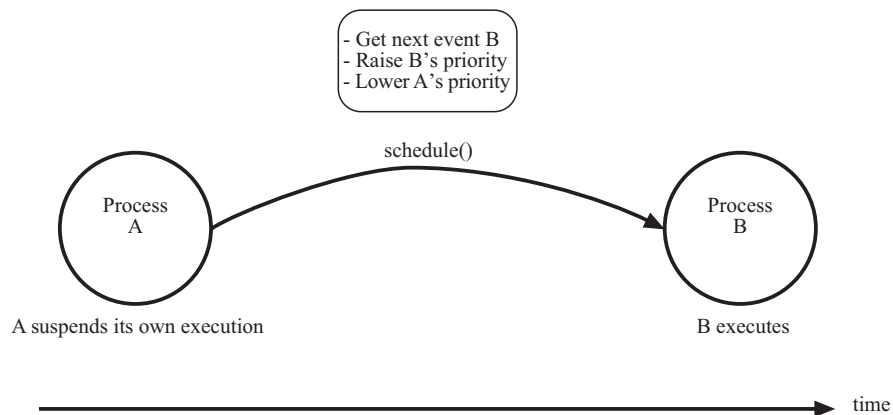
Consider the situation where multiple departures may occur simultaneously—parallel processors completing job processing in parallel—even though job arrivals are single. An efficient algorithm (as detailed in the following section, for example) may require simultaneously scheduled departures to be processed before arrivals, so that the latter do not invalidate the state of the job pool. To accommodate this, the scheduler gives departures priority over arrivals: the kernel is made to recognize the distinct events DEPARTURE, ARRIVAL and OTHER. Upon extracting an event **E** of type ARRIVAL from the calendar, if the next event **nextE** is found to have the same reactivation time as **E** (an operation that can be done with  $O(1)$  cost in a heap), and **nextE** is of type DEPARTURE, **nextE** is scheduled for execution and **E** is returned to the calendar, since **nextE** must be a departure. Transfer of control from the currently running thread (identified by `current_job_pid`) to the thread selected to run (**nextE**) is accomplished when the former raises the latter's priority and decreases its own. With such a modification a standard scheduler is ready to support more efficient scheduling algorithms in the domain layer.

The proposed changes to the scheduler have to be implemented within the kernel layer. These changes are relatively simple, since they involve only a few lines of code. More importantly, existing scheduler implementations can be augmented, preserving backward compatibility with existing simulations that do not need this feature. The pseudo code for such a scheduler, along with a schematic view of its operation, are shown in Figure 4.

The logic of the `hold( $\tau$ )` function, which enables a running process to suspend its own execution for  $\tau$  simulation time units, is shown in Figure 5. A simple optimization is possible when the invoking process is also the next process to run. It simply compares its reactivation time with the minimum timestamp in the calendar, and continues to run if its reactivation time is smaller. This saves on the cost of two context-switches when the scheduler is a thread, and one when the scheduler is a function, in addition to savings with calendar operations. When a process terminates execution, a special function `sim_exit()` is invoked; this function in turn calls `schedule()`, so that the next process in the calendar may run.

#### *Process coordination*

Synchronization between processes may be achieved through user-defined events, based on the `wait_event(e)` and `set_event(e)` functions supported by the kernel. The application declares an event `e` of type `Event` and initializes it by invoking the `create_event()` function. When a process invokes `wait_event(e)`, it is blocked (suspended) on the event, while its event record is held in an event list tied to event `e`. The event is triggered via a `set_event(e)` invocation, at which point all events blocked on `e` are freed, with the current time as reactivation time. Process synchronization may also be done through use of message-passing or mailboxes. Two functions are provided for sending and receiving messages: `mailb_send(mb, msg)` and `mailb_receive(mb, &msg)`, where `mb` is a data structure of type `Mailbox` initialized by the `mailb_create()` function. A process awaiting a message is blocked until it receives the message.



```

schedule() {
    while (future_event_set is not empty) {

        E = extract_next_event from calendar;
        if (E.type == ARRIVAL){
            nextE = look_next_event from calendar;

            if (nextE.clock == E.clock) {

                /* next event in line must be departure */

                tempE = E;
                E = extract_next_event from calendar;
                insert tempE into calendar;
            }
        }
        clock = E.clock;
        setpri(E, MAXPRIO);
        setpri(current_job_pid, MINPRIO);

        /* control transfers to E.pid automatically */
    }
}

```

Figure 4. The schedule () function.

```
hold(t){
    E = extract_next_event from calendar;
    if (t + clock < E.clock) {

        /* imminent execution, don't insert */

        clock += t;
        return;
    }
    else {
        E.pid = current_job_pid;
        E.clock = clock + t;
        insert E into calendar;
        schedule();
    }
}
```

Figure 5. The hold( ) function.

#### *Resource management: domain-level primitives*

Resources are passive system objects with mutually exclusive access. They are used to model components that are held for a time and then released, via `request(r)` and `release(r)` primitives; here `r` denotes a resource created by a `resource_create()` function and typically implemented with a queue. A flag is used to tell if a resource is busy or idle. If a resource is found to be busy, the requesting process is made to wait in the queue. When a process releases a resource it has been using, the first process queued is dequeued and given the resource; its reactivation record is put into the calendar with the current time as its time to run.

To support new and efficient algorithms at the application layer it is necessary to support new application-level primitives in the domain layer. For example, the naïve round-robin algorithm is easily implemented using a queue, where jobs are returned to the queue after a simulated quantum, but at a cost of multiple thread context-switches. A more efficient algorithm [10] utilizes a circular pool to store jobs, implemented as a linked list; the primitives `insert_into_pool(r)` and `delete_from_pool(r)` handle insertion and deletion, respectively. A pointer to the head of the pool is made available to the domain layer.

An important design choice arises at this point. Should logic pertaining to simulation of a new class of models (e.g. multiprocessor simulations) be supported within the resource management functions—by accounting for the number of busy/idle processors in reserve or release operations? Or, should such logic reside within the domain layer? Since the objective is the provision of a general purpose testbed for new simulation algorithms, this functionality is kept within the domain layer. In this way, a variety of models can be tested without interfering with the simulation kernel.

#### **Domain and application layers**

The simulation kernel is available to the domain layer as a library of kernel functions; the idea is to hide its implementation details from the domain layer. When a simulation begins, the kernel initiates an

application-level process called `sim()` which, in turn, invokes domain-layer functions to implement the logic associated with a particular model. An advantage of supporting distinct domain layers is that different application areas (e.g. particle simulations, multiprocessor simulations, network simulations) can be supported in different domains, supporting independence and modularity on a versatile kernel.

In the next section a detailed example is given of multiprocessor scheduling as a domain. In a vehicular traffic system, the domain would provide functionality that enabled vehicles to move within and between objects (e.g. city streets) subject to resource limitations (e.g. space limitation, traffic lights) that lead to queueing behavior; an air-traffic domain would require similar functionality in three-dimensional space, with resource limitations in airports. An agent-based simulation for the study of epidemics, or herding behavior in terror attacks, would require domain functionality that enabled movement within city blocks and buildings, subject to space constraints. Whenever limited resources are involved (competition for space, or access, or service) the resulting contention causes queueing behavior, and much of the functionality from the queueing domain can be imported into the target domain. Performance is solely a function of the degree of contention between a system's processes, as will be made clear in the following section.

## TESTS: MULTIPROCESSOR ROUND-ROBIN QUEUES

With two objectives in view—to demonstrate the use of the testbed, and to show how improved algorithms can easily be developed and tested—a new and efficient algorithm for simulating round-robin multiprocessing is presented. In this model, each arriving job requires a random number of service quanta, and jobs wait in a FIFO queue for a single quantum from the next available processor. In the standard but naïve round-robin implementation, up to  $N$  jobs from the head of the queue are serviced simultaneously by the  $N$  processors during a single quantum. Upon completion of a quantum, jobs that do not require further service quanta leave, while the rest are returned to the rear of the queue. With a threads-based scheduler, if each of  $n$  jobs requires  $k$  quanta of service, a total of  $2 * (nk - 1)$  context switches will ensue. This can be computationally expensive, especially if  $k$  is large.

A more efficient algorithm generalizes the idea presented in [5]. Here, jobs are kept in a circular pool, and scheduling is optimized via use of exit-time computations and cancellations. The net effect is to sharply curtail the number of thread context switches. The pool is implemented using a circular linked list of job records, where a job record must contain at least the job pid and remaining service quanta, as shown in Figure 6. A special pointer points to the next job at the head of the queue.

While the computational algorithms must use the circular pool, the naïve round-robin service can be simulated either by using the pool or by dequeuing and enqueueing the jobs—both methods offer the same performance. However, for ease of understanding and simplified model implementation, it is desirable to use the same resources between the service disciplines. In our experiments, the naïve round-robin discipline is modeled using the circular pool. In practice, a generic round-robin service is used for both algorithms and only one function needs to be changed to switch between different service strategies.

### Round-robin service discipline

In requesting a generic round-robin service, an arriving job invokes the function `round_robin(f, t)` to indicate that it wishes to join a round-robin service center  $f$  and obtain service for time  $t$ . As shown

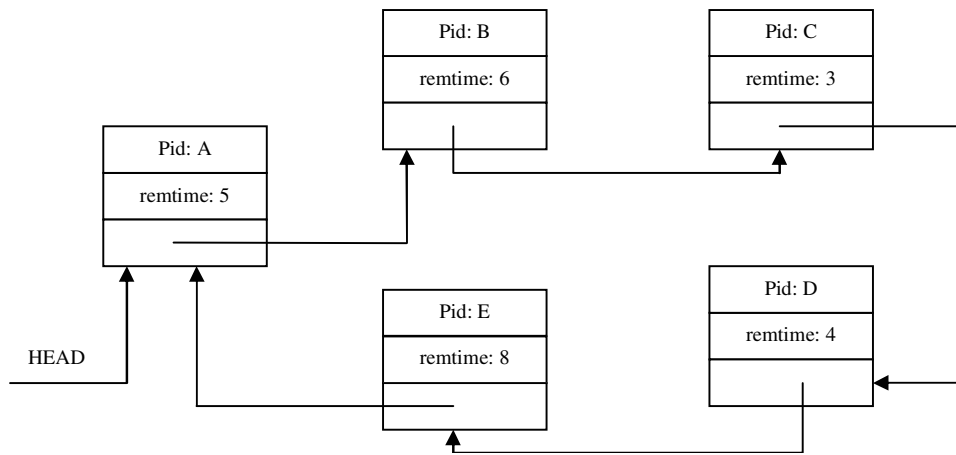


Figure 6. Circular pool for implementing round-robin discipline.

```

round_robin(f, t) {
    insert_into_pool(f, t);
    schedule();
    delete_from_pool(f);
    sim_exit();
}

```

Figure 7. The generic round-robin function.

in Figure 7, when invoked, the `round_robin(f, t)` function places the invoking job's activation record into an existing pool of jobs via `insert_into_pool()`. Because this job is now blocked in a queue awaiting service, the subsequent call to `schedule()` enables the scheduler to take charge and yield control to the appropriate process in the event calendar. When this job's turn to run finally arrives, the job obtains control from the scheduler and invokes the `delete_from_pool()` function, which is an implicit service mechanism. If variations in the service strategy are to be tested (the naive versus the proposed computational algorithm, for example) only this function call needs to be changed—it effectively reduces the job's service requirement by a time slice (`delta`) and performs all the necessary book-keeping. When the remaining service time reaches zero, the job is released and invokes `sim_exit()`, which terminates the process and gives control to the scheduler. If the remaining service time is greater than zero, it is returned to the pool and the next scheduled process runs.

### Pool updates in the computational algorithm

When a process from the pool runs, it performs routine tasks to update the state of the pool over simulation time that has elapsed since the last pool update. Since waiting jobs are serviced in a cyclic

```

adjust_pool(f){
    delta = clock-prev_clock;

    N = NPROC;
    if (size_of_pool(f) < NPROC)
        N = size_of_pool(f);

    pseudo_ticks = delta/q*N;
    sub = pseudo_ticks/size_of_pool(f);
    rem = pseudo_ticks%size_of_pool(f);

    for (ptr = head, i = 0; i < size_of_pool(f); i++) {
        ptr → remtime -= sub;
        ptr = ptr → next;
    }

    for (ptr = head, i = 0; i < rem; i++) {
        ptr → remtime -= 1;
        ptr = ptr → next;
    }

    head = ptr;
}

```

Figure 8. The `adjust_pool()` function.

order, the remaining service time of each has to be decreased accordingly, to reflect service these jobs have received during the elapsed time. Each arriving process invokes the function `adjust_pool(f)` to simulate work done by the multiprocessor between the arrival and the last update.

The `adjust_pool(f)` function calculates elapsed time `delta` by subtracting the time `prev_clock` of the last update from the current time. Next, the value of  $N$  is determined by comparing the number of servers `NPROC` to the size of the pool. Based on these and the quantum  $q$  of service, the number of pseudo-ticks that have elapsed are calculated. This is the number of quanta (ticks) that a single processor would need to bring the pool of waiting jobs to the same state as a multiprocessor in the given elapsed time `delta`. The quantities `sub` and `rem` are calculated to update the pool accordingly, where `sub` is the amount to be subtracted from all jobs in the pool, `rem` is the amount of time to be subtracted from those jobs lying between the current head of the pool and the new head; the new head pointer is made to point to the head of the new pool state. The pseudo-code for function `adjust_pool(f)` is shown in Figure 8.

Consider an example based on a pool of five jobs, with state shown in Figure 6, and examine how an update is made after an elapsed time of two quanta in a three-processor system, using  $q = 1$ . During the first quantum, jobs A, B and C receive service, and during the second quantum, jobs D, E and A receive service. Thus, at the end of two quanta, the new head pointer for the pool points to job B. The adjusted pool is shown in Figure 9.

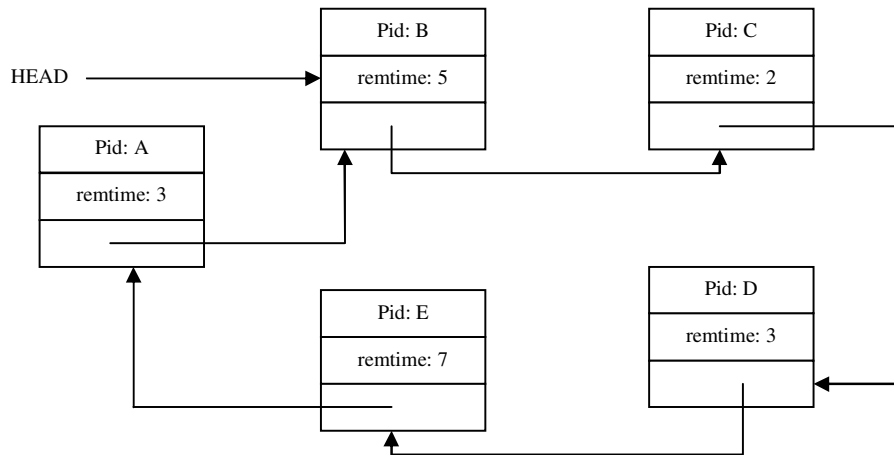


Figure 9. Adjusted pool after two service quanta,  $N = 3$  and  $q = 1$ .

```

insert_into_pool(f, t) {
    adjust_pool(f);
    insert currently running process in pool with retime = t;
    schedule_departures(f)
}

```

Figure 10. Function for handling arrivals.

The algorithm yields the following quantities:  $\delta = 2$ ,  $N = 3$ ,  $q = 1$ , the number of pseudo-ticks at  $2/1 \cdot 3 = 6$ ,  $\text{sub} = 6/5 = 1$ , and  $\text{rem} = 6\%5 = 1$ . Subtracting a single tick from each job, the remaining times for jobs A through E are obtained as 4, 5, 2, 3, and 7, respectively. Finally, since  $\text{rem} = 1$ , the head pointer is moved forward by one step to its new position, subtracting one tick from the service requirement of each job encountered along the way. This causes one tick to be subtracted from job A's service requirement.

### On handling arrival, service and departure

An arriving process invokes function `insert_into_pool(f, t)` to add itself to the job pool. This function performs the pool update and inserts the invoking process at its proper position in the pool for center  $f$ , with requested service time  $t$ . It then proceeds to schedule multiple departures from the pool. The pseudo code for handling arrivals is shown in Figure 10.

With multiple processors, a single quantum may witness the service of up to  $N$  jobs. Each such job that receives service is a potential departure, but may leave the system only if its remaining service time requirement has reached zero. As explained earlier, function `delete_from_pool(f)` simulates

```

delete_from_pool(){
    /* first_process and proc_serviced global variables */

    retain = TRUE;
    while (retain) {

        if (first_process){
            first_process = FALSE;
            adjust_pool(f);
        }

        proc_serviced++;
        E = current job pid;
        retime = E.retime;

        /* update head and schedule new departures
           only after all processes have updated the pool */

        if (proc_serviced == N){
            first_process = TRUE;
            proc_serviced = 0;
            delete jobs from pool with retime < 0;
            schedule_departures(f);
        }

        /* yield control or exit process */
        if (retime > 0)
            schedule();
        else
            retain = FALSE;
    }
}

```

Figure 11. Function for handling service and departures.

service doled out by the processors and also schedules departures. As a result, up to  $N$  processes may be scheduled for departure with the same reactivation time, and expected to run in sequence at that time. The pseudo code for function `delete_from_pool(f)` is shown in Figure 11.

When `delete_from_pool(f)` is invoked by a process, it enters a loop controlled by the `retain` flag, which is initially set to be `TRUE`. It then simulates the passing of service time by invoking `adjust_pool(f)`; the latter simulates the multiprocessing of several jobs. A global flag, `first_process`, is used to ensure that the pool is adjusted only once, by the first invoking process. Another global variable, `proc_serviced`, counts the number of invoking processes. When all scheduled processes have executed the `delete_from_pool(f)` function, i.e. when `proc_serviced` reaches  $N$ , the global variables and flags are reset, jobs with remaining time decremented to zero or less are removed from the pool, and a set of departures is scheduled. Here,  $N$  is a global variable set by the `adjust_pool(f)` function. The jobs are deleted from the pool only

when the last process runs, to ensure that processes that run before it see a system with consistent state. An invoking process exits the loop by setting `retain` to `FALSE`, if its remaining service time requirement is nil; otherwise, it invokes `schedule` to yield control to another process. The loop ensures that processes repeatedly obtain service quanta, and this is achieved in a more efficient manner than in the naïve round-robin strategy.

### Scheduling multiple departures for efficiency

#### *The naïve round-robin discipline*

In the naïve round-robin algorithm, the first  $N$  jobs starting at head of the pool are candidates for departure. The `schedule_departures(f)` function determines the value of  $N$  and then inserts these  $N$  job activation records in the calendar, each with reactivation time equal to  $(\text{clock} + q)$ . If an arrival occurs before the time of the already scheduled departures, the departures are deemed to have been prematurely scheduled (i.e. they did not account for the new arrival in the job pool) and must be canceled.

To circumvent such cancellations and the high cost of multiple calendar insertions/deletions, look-ahead [10] is used; this is a non-disruptive method that produces the same results as cancellations. The time of the next arrival is available to the system in the logic that generates arrivals—typically via a `hold(x(t))` function, which simulates the passage of  $x(t)$  time units between consecutive arrivals, where  $x()$  is a random variate generator. Before scheduling a departure, the algorithm looks ahead to determine the time of the next arrival, i.e.  $\text{clock} + x(t)$ . Some book-keeping is necessary to maintain consistency with the look-ahead motivated use of variate generators. If the scheduled departure time is greater than the arrival time, the departure events are not scheduled. The pseudo code for the naïve round-robin algorithm is shown in Figure 12.

#### *The computational round-robin algorithm*

In the naïve round-robin algorithm, every job (process) runs after a quantum of service, resulting in frequent thread context switches and, consequently, longer simulation time. It is possible to extend techniques introduced in computational approaches [5,10] to accommodate multiple processors. For this, the concept of state introduced in those approaches is used: system state is defined by the remaining service time requirements of all jobs in the pool, the size of the pool and the job next in line to receive service. To sharply reduce the number of events and thus context switches, changes to pool state are recognized only at arrival and departure events.

The naïve and computational views are illustrated in Figure 13. While the former generates multiple context switches at the end of every quantum, the latter exploits tick counting and scheduling to circumvent context switching. Because job departure times are computed by exploiting system state and arrival times, the algorithm is computational. To find the next imminent departure, the algorithm traverses the pool and locates the job with the shortest remaining service time `minrem`. The distance in steps from the pool head to the departing job is stored in variable `steps`.

In a multiprocessor simulation, the next job departure time may be obtained as

$$\text{deptime} = \text{clock} + \text{ceil}((\text{minrem} - 1) \times \text{size\_of\_pool}(f) + \text{steps}) / N \times q$$

```

schedule_departures() {
  /* naïve round-robin discipline */

  /* determine how many departures to schedule */
  N = NPROC;

  if (size_of_pool(f) < NPROC)
    N = size_of_pool(f);

  /* calculate time of next departure */
  deptime ≤ clock + q;

  /* look-ahead to see if departure proceeds */
  if (deptime < time_of_next_arrival) {

    /* put departure events in calendar */
    for(i = 0, ptr = head; i < N; i++) {
      E.pid = ptr → pid;
      E.type = DEPARTURE;
      E.clock = deptime;
      insert E into calendar;
      ptr = ptr → next;
    }
  }
}

```

Figure 12. Naïve round-robin algorithm for multiple departures.

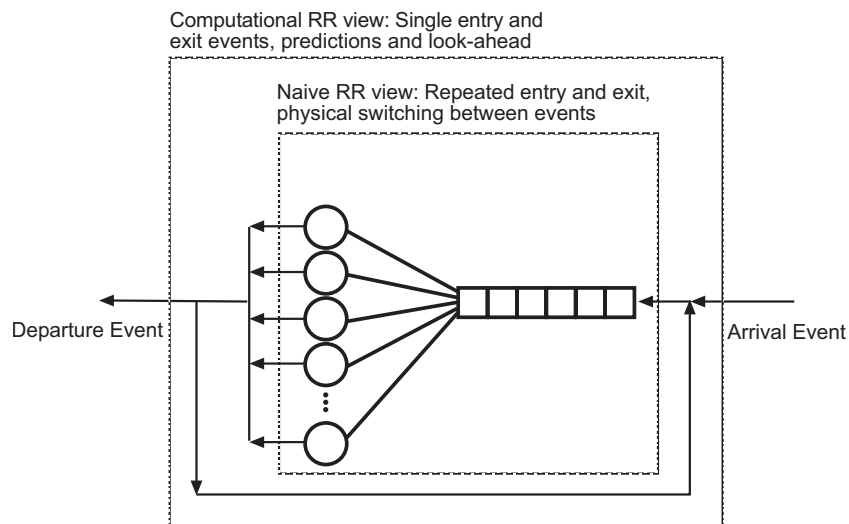


Figure 13. A comparative view of two round-robin algorithms.

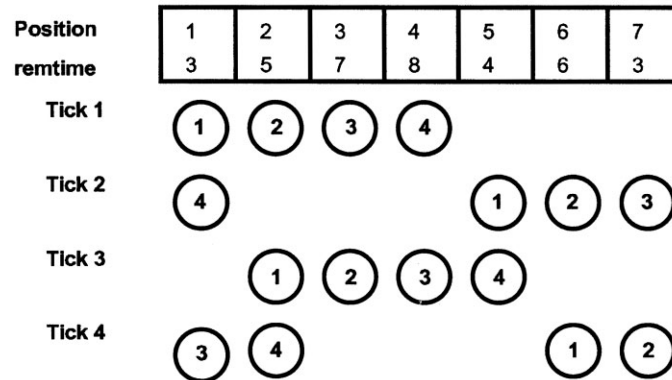


Figure 14. Example of multiprocessor service.

where  $\text{ceil}()$  is the standard ceiling function. Because this time is invalidated if a new arrival occurs, look-ahead is used to avoid overhead arising from cancellations.

A crucial problem in scheduling departures with the multiprocessor version of the computational algorithm lies in identifying the  $N$  candidates for departure on state changes. Consider the simple four-processor system shown in Figure 14, with seven jobs in the pool, and four quanta of service required by the first departure, which is job 1. The head pointer is moved on each tick, and the figure shows which jobs are serviced one quanta on each tick. The candidate jobs for departure in this example are jobs 6, 7, 1, and 2.

Consider how a set of candidates for departure may be identified in a general way. On each departure there are  $N$  candidates. If the jobs are numbered  $1 \dots N$ , the relative position of the job with the  $\text{minrem}$  tick requirement within the set is given by

$$\text{elem} = ((\text{minrem} - 1) \times \text{size\_of\_pool}(f) + \text{steps}) \% N$$

where an  $\text{elem}$  value of zero is interpreted to mean  $\text{elem}$  has value  $N$ . Once the element with minimum tick requirement is identified, its position can be determined, and the other candidates are predecessors in positions  $(1 \text{ to } (\text{elem} - 1))$  and successors in positions  $(\text{elem} + 1) \text{ to } N$ . A direct way to obtain the pointer to the starting position in the candidate set is to use an auxiliary  $N$ -item array  $\text{tt}[]$  of pointers loaded with job records while searching for  $\text{minrem}$ . Once array  $\text{tt}[]$  is available, the starting job is given by

$$\text{start} = \text{steps} - \text{elem} + 1$$

A value of  $\text{start}$  less than or equal to zero is taken to mean that  $\text{start} = + \text{size\_of\_pool}(f)$ . Given this value, the start pointer of the candidate set is  $\text{start\_ptr} = \text{tt}[\text{start}]$ . The candidate processes are the  $N$  jobs obtained by a cyclic traversal of the pool starting from  $\text{start\_ptr}$ . Following the example in Figure 14 and applying the given formulae yields  $\text{minrem} = 3$ ,  $\text{steps} = 1$  for the job in position 1,  $\text{elem} = 3$ , and  $\text{start} = 6$ , with candidate departures being those in positions 6, 7, 1 and 2, as expected.

```

schedule_departures(){
    /* computational algorithm */

    minrem = head → remtime;
    steps = 1;
    tt[1] = head;    /* auxiliary array */

    /* search process with min remaining ticks */
    for (ptr = head → next, i = 2; i ≤ size_of_pool(f); i++){
        tt[i] = ptr;
        if (ptr → remtime < minrem){
            minrem = ptr → remtime;
            steps = i;
        }
        ptr = ptr → next;
    }

    /* how many departures to schedule */
    N = NPROC;
    if (size_of_pool(f) < NPROC)
        N = size_of_pool(f);

    /* calculate time of next departure */
    comp = ((minrem - 1)*size_of_pool(f) + steps)/N;
    deptime = simclock + ceil(comp)*q;

    /* look-ahead to see if departure proceeds */
    if (deptime ≤ time_of_next_arrival){

        /* determine which elements depart -- candidate jobs */
        elem = ((minrem - 1)*size_of_pool(f) + steps)%N;
        if (elem == 0)
            elem = N;
        start = steps - elem + 1;
        if (start ≤ 0)
            start + = size_of_pool(f);
        start_ptr = tt[start];

        /* put departure events in calendar */
        for(ptr = start_ptr, i=0; i < N ;i++){
            E.pid = ptr → pid;
            E.type = DEPARTURE;
            E.clock = deptime;
            insert E into calendar;
            ptr = ptr → next;
        }
    }
}

```

Figure 15. Computational algorithm for multiple departures.

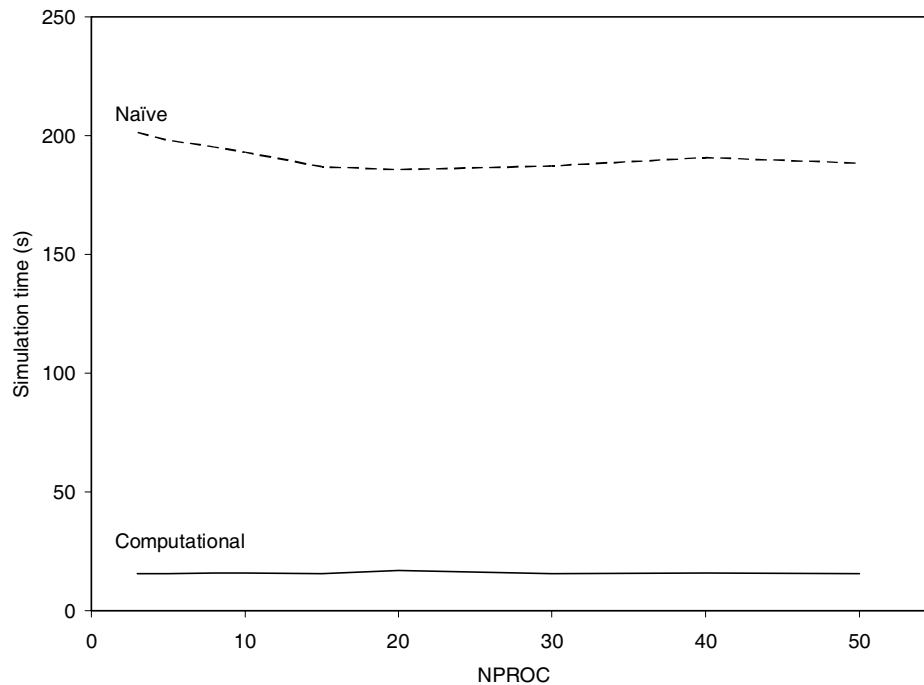


Figure 16. Performance of two multiprocessor round-robin algorithms.

In summary, the computational algorithm locates the next departure, computes the corresponding departure time, identifies the set of candidate jobs for departure, and finally, using look-ahead, schedules the departures accordingly. The pseudo code for the computational algorithm is presented in Figure 15.

### Benchmark measurements

The naïve and computational algorithms for multiprocessor round-robin service were implemented on our testbed for performance comparison. The system was evaluated using exponentially distributed job interarrival and service times, with means  $1/\lambda$  and  $1/\mu$ , respectively, with experimentally determined simulation run times as the output measure.

*Experiment 1.* The intent of this experiment was to measure the performance of both algorithms as a function of the number of processors NPROC. The parameters used were  $1/\lambda = 100$ ,  $1/\mu = 80$ , quantum  $q = 1$ , and  $n = 20\,000$  jobs. The simulation was run for different values of NPROC, ranging from 3 to 50. For each value of NPROC, 20 runs were made using different random number seeds, and the results averaged. A total of 300 runs were made for each round-robin algorithm. The experimental results in Figure 16 show that simulation times are independent of NPROC.

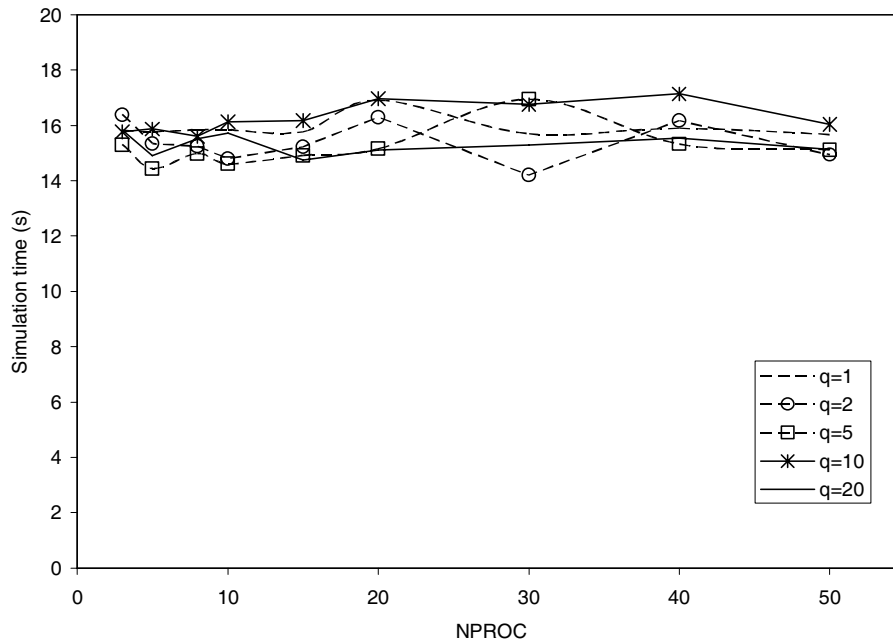


Figure 17. Effect of quantum  $q$  and NPROC on the computational algorithm.

*Experiment 2.* The purpose of this experiment was to observe the effect of the quantum  $q$  on performance. The parameters used were  $1/\lambda = 100$ ,  $1/\mu = 80$ , and  $n = 20\,000$  jobs. The simulation was run for different values of NPROC, ranging from 3 to 50 and  $q = 0.5, 1, 3.5, 5$  and 10. For each value of NPROC, 20 runs were made using different random number seeds, and the results averaged. A total of 120 runs were made for each round-robin algorithm, for each fixed value of NPROC. The experimental results for the computational algorithm are shown in Figure 17. It is clear that the simulation times are independent of the value of  $q$ . The dependence of both algorithms on the value of  $q$  is displayed in Figure 18, for different values of NPROC. The average numbers of context switches incurred by the algorithms are summarized in Table I.

### Interpretation of results

With each algorithm, simulation run time is directly related to the number of events processed, manifesting as overhead in terms of thread context switches, calendar operations and resource operations. The results show that the computational algorithm outperforms the naïve algorithm by sharply curtailing the number of processed events, which in turn drastically reduces the number of context switches. As an example, for  $q = 1$ , the reduction in the number of context switches is close to a factor of 20, as evidenced by the results presented in Table I, while the computation time is

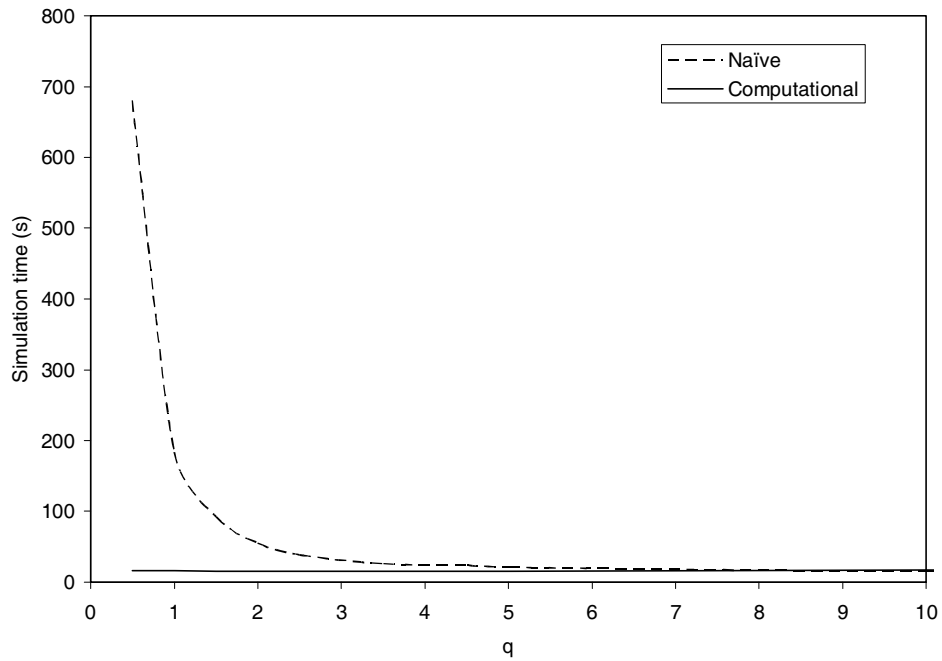


Figure 18. Effect of quantum  $q$  on the multiprocessor round-robin algorithms.

Table I. Average number of context switches for both multiprocessor round-robin algorithms.

Algorithm	$q$	Number of context switches
Naïve	0.5	3 272 007
	1	1 129 992
	3.5	146 583
	5	88 698
	10	46 875
Computational	0.5–10	55 673

reduced by a factor of approximately 10. This suggests that computational overhead associated with the more complex data structures in the computational algorithm reduce the gains given by eliminating unnecessary context switches.

A simple comparative analysis proceeds as follows. Let  $m$  be the number of context switches required by the computational algorithm, with the generic cost of a context switch assumed to be a small constant  $c$ . Also, let  $C$  and  $S$  denote the non-context-switching work done by the computational and naïve algorithms, respectively. If  $T(w)$  is used to denote the processing time required for simulation work  $w$ , speedup in run time is given by

$$\frac{K * m * c + T(S)}{m * c + T(C)} = \alpha * K$$

where  $K$  is the factor of improvement in the number of context switches and  $\alpha * K$  is the factor of improvement in run time obtained by the computational algorithm, where  $0 < \alpha < 1$ . For  $q = 1$ ,  $K \approx 20$  and  $\alpha \approx 0.5$ . Solving for  $T(C)$ ,

$$T(C) = \frac{(1 - \alpha)}{\alpha} * m * c + \frac{T(S)}{\alpha * K}$$

If  $d$  is the average amount of time required to process each of the  $K * m$  events (associated with the  $Km$  context switches) in the non-context-switching portion of the naïve algorithm, then  $T(S) = K * m * d$ , and

$$T(C) = m * \left[ \frac{(1 - \alpha)}{\alpha} * c + \frac{d}{\alpha} \right]$$

from where it can be concluded that each of the  $m$  events (associated with the  $m$  context switches) in the non-context-switching portion of the computational algorithm requires an average cost of  $c * (1 - \alpha) / \alpha + d / \alpha$ . This suggests that the processing of each event may require considerably more time in the computational algorithm, on average. Our experimentally obtained value of  $\alpha \approx 0.6$  suggests that each event in the computational algorithm requires an average processing time of approximately  $c + 2 * d$  units, which is more than twice the average event processing time of the naïve algorithm. Nevertheless, the gains to be had by reducing the number of context switches outweigh these costs by a good measure. Similar conclusions can be drawn for other values of the quantum  $q$ .

The above discussion is valid only for  $0 < \alpha < 1$ , for which case arriving jobs are served in a round-robin fashion. For large values of  $q$ , each arriving job tends to leave after a single service quantum, and this is not round robin. Thus, the analysis does not apply for large  $q$ . In the benchmark experiments, the region of interest occurs for values of  $q < 6$ , as can be seen in Figure 18.

The naïve algorithm requires more simulation time for small values of  $q$ , because the jobs require more events (and context switches) to satisfy their service requirements. The computational algorithm, on the other hand, performs the same work independently of the value of  $q$ , and this behavior is reflected in Figures 17 and 18.

A graphical comparison of the number of events for both algorithms is shown in Figure 19. Observe that each departure event incurs context-switching overhead for up to  $Np$  processes, as shown in Figure 20. It is important to recognize that for the naïve round-robin algorithm, a departure event only means that a job has completed a service quantum and may have to return to the pool, whereas a departure in the computational algorithm implies that at least one job from the candidate set leaves the pool.

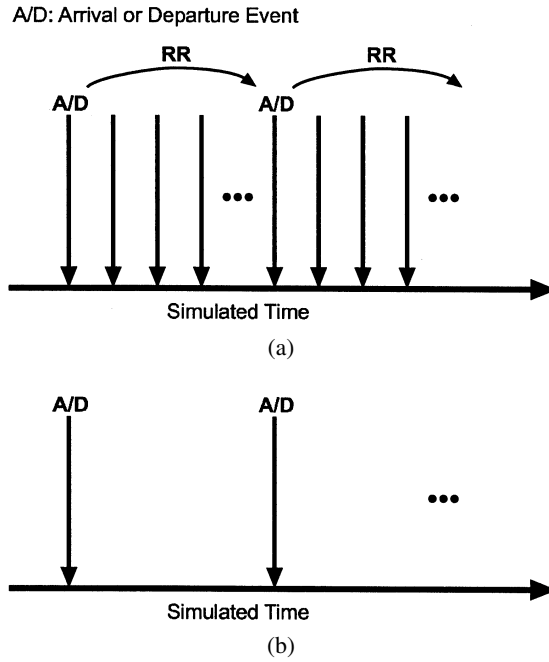


Figure 19. Comparison of events in the round-robin algorithms. (a) The naïve round-robin algorithm for multiple processors; (b) the computational round-robin algorithm for multiple processors.

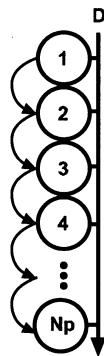


Figure 20. A departure event for the multiprocessor system.

As the number of processors NPROC is increased, the number of jobs handled by each processor is proportionately reduced, i.e. each processor handles  $20\,000/\text{NPROC}$  jobs. This implies that the number of context switches remains roughly constant, as reflected by the roughly constant graphs for both algorithms seen in Figures 16 and 17. The naïve algorithm exhibits a decreasing run-time trend as the number of processors is increased; with fewer processors, there is a larger number of context switches before a departure occurs. The computational algorithm is unaffected by this phenomenon and thus yields a graph that is constant.

## CONCLUSIONS

This study had two objectives—to empirically evaluate a versatile layering scheme for testing algorithmic ideas in process-oriented simulation, and to test a new algorithm with respect to ease of implementation and run-time performance. The study originated from our experiences with thread-based simulation and experimentation with efficient round-robin algorithms for uniprocessor queues. As the study progressed, it was found that it was possible to provide a uniform testbed for a variety of simulation algorithms, and a new multiprocessor round-robin algorithm was an ideal test candidate. As is shown by the results, even though new simulation algorithms can be quite complex, a framework for their experimental study and evaluation is well worth the effort.

We were forced to address many design issues, including leaving much of the code in the domain and application layers. If specific situations or further study reveals advantages in moving functionality into the simulator kernel, appropriate changes are simple to make because of the layering. We found that much generality is possible with a versatile scheduler, and this versatility was obtained with a small modification to the standard scheduler. The result is useful in that it is easy to support multiprocessing simulation algorithms on the old uniprocessor simulation framework.

## REFERENCES

1. Evans JB. *Structures of Discrete Event Simulation*. Ellis Horwood: Chichester, U.K., 1988.
2. Franta WR. *The Process View of Simulation*. North-Holland: Amsterdam, 1977.
3. MacDougall MH. *Simulating Computer Systems: Techniques and Tools*. MIT Press: Cambridge, MA, 1987.
4. Schwetman HD. CSIM: A C-based process-oriented simulation language. *Proceedings of the 1986 Winter Simulation Conference*. IEEE: Piscataway, NJ, 1986; 387–396.
5. Sang J, Chung K, Rego V. A simulation testbed based on lightweight processes. *Software—Practice and Experience* 1994; **24**(5):485–505.
6. Mascarenhas E, Knop F, Rego V. Minimum cost adaptive synchronization: Experiments with the PARASOL system. *Proceedings of the 1997 Winter Simulation Conference*. IEEE: Piscataway, NJ, 1997; 389–396.
7. Mascarenhas E, Rego V. ARIADNE: Architecture of a portable threads system supporting thread migration. *Software—Practice and Experience* 1996; **26**(3):327–357.
8. Sun Microsystems, Inc. *Multithreaded Programming Guide*. Sun Microsystems: Palo Alto, CA, 1997.
9. IEEE. *POSIX 1003.1c API Threads Standard*, 1995.
10. Ramos JR, Sang J, Rego V. An efficient burst-arrival and batch-departure algorithm for round-robin service. *Technical Report*, Purdue-CSD, December 2002.