A Computational Multiprocessor Round-Robin Algorithm

Jorge R. Ramos and Vernon Rego
Department of Computer Sciences
Purdue University
West Lafayette, IN 47906

Abstract

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.

1 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 also motivates studies of models, such models are often computationally intensive. Thus, there is a need for a framework that enables the implementation of interesting and efficient models with minimum effort. Among various techniques developed over recent decades, process-oriented[1,2,3] 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, for example, jobs, CPUs, queues, etc. 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. 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.

Consider a single queue (pool) 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, we assume that all job service times are the same. If the number of jobs is larger than or equal to the number of processors `NPROC`, the first `NPROC` jobs in the pool are processed simultaneously. When the number of jobs in the pool is less than `NPROC`, however, all jobs are serviced while (`size_of_pool` − `NPROC`) processors idle for a single job quantum. We will use N to denote the number of jobs that receive service or the number of processors that service the jobs, so that N is either `NPROC` or `size_of_pool`, depending on their relative sizes.

Our 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, we implement and test a nontrivial multiprocessor round-robin algorithm within the domain layer, a layer available to application-level code. 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[8] as well as the multiprocessor cases. Here, we focus 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.

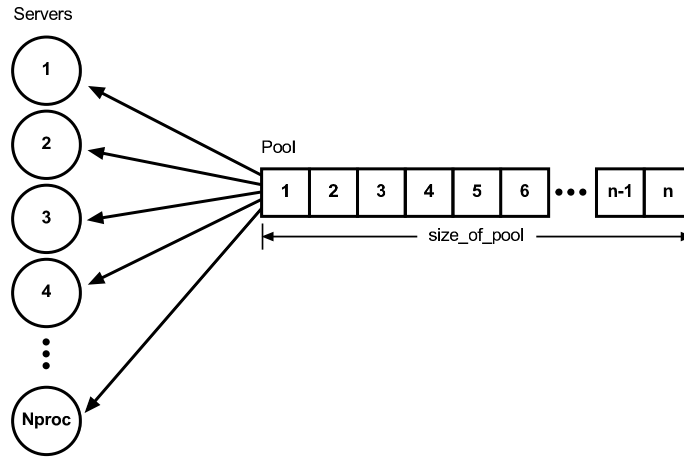

Figure 1: Multiprocessor Job-scheduling

The rest of the paper is organized as follows. The design and implementation of the process-oriented simulator is briefly reviewed in Section 2. In Section 3 we show how a uniprocessor round-robin algorithm can be generalized to accommodate multiple processors. 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. We conclude briefly in Section 4.

2 Process-oriented simulators: An Overview

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. 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, etc. 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 smallest time-stamp value.

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. The calendar is typically a priority queue or similar data structure that offers efficiently retrievable activation records based on minimum time-stamp.

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 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.

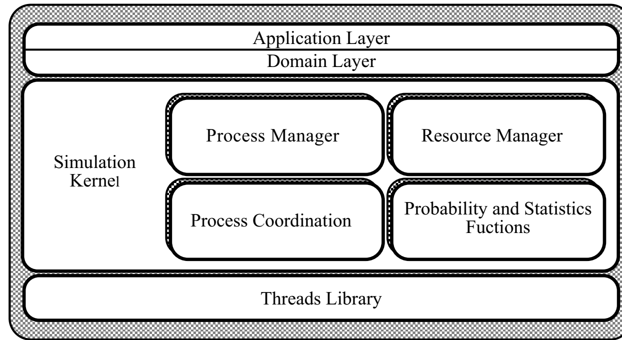

Figure 2: Software Layers

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. A scheduler may be implemented either as a process or a function, we chose to go with a function-based scheduler in Sol because it eliminates one context-switch per process transfer, reducing simulation run-time.

Layered Architecture

The layered, modular architecture is shown in Figure 2. 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 domain layer. Portability is readily had 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[7] threads library; threads run as functions on distinct stacks within a single address space on a host process that support a virtually unlimited number of threads. Examples of similar systems include the Solaris Thread Library[9] and POSIX Threads[10]. For portability, we use key primitives wrappers that are easily mapped into various threads functions for different libraries.

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 our test algorithm, the scheduler itself only requires minimal changes to accommodate generality for various algorithms. Further details of the kernel functions can be found in [11].

Process management

Each simulation process is supported by a thread, and application-level jobs are modeled by processes. Henceforth, we use the term process and job interchangeably. This module contains the functions that enable transfer of control between processes, `schedule()`, which invokes the simulation scheduler, `hold(t)`, which enables a running process to suspend its own executin for `t` simulation time units and `sim_exit()`, which explicitly calls the scheduler when a process terminates execution.

A basic simulation scheduler dispatches these events through naïve scheduling, though the high-cost of context-switching between multiple, simultaneously scheduled threads leads to severe inefficiencies. To

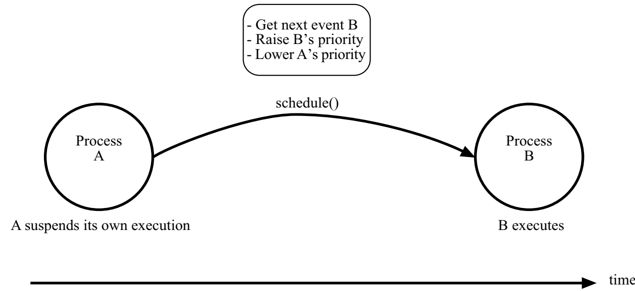

Figure 3: The scheduler

enhance process management the scheduler can be made to prioritize simultaneous events based on type. 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 may require simultaneously scheduled departures to be processed before arrivals. To accommodate this, the scheduler gives departures priority over arrivals, when both events have the same reactivation time, an operation which can be done with O(1) cost in a heap. The schematic view of the `schedule`() function is shown in Figure 3, where is understood that B was chosen using the described priority scheme.

Process coordination

Synchronization between processes may be achieved through user-defined events or through the use of message-passing or mailboxes. A process awaiting a message is blocked until it receives an event or message.

Resource management: domain-level primitives

Resources are passive system objects with mutually exclusive access. They are used to model components which are held for a time and then released and are typically implemented with a queue and control flags. 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, an efficient algorithm[8] for round-robin discipline 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 our objective is the provision of a general purpose testbed for new simulation algorithms, we choose to keep this functionality 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 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 on different domains, keeping distinct domains independent and modular, while sharing a versatile kernel.

3 Test algorithms: Multiprocessor round-robin queues

We present a new and efficient algorithm for simulating round-robin multiprocessing. In this model, each arriving job requires a random number of service quanta, and jobs wait in a FIFO queue for a single quantum

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

Figure 4: The generic round-robin function

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, n jobs which each require k quanta of service will generate $2 * (nk - 1)$ context-switches. This can be computationally expensive, especially if k is large.

A more efficient algorithm generalizes the idea presented in[8]: jobs are kept in a circular pool, and scheduling is optimized via use of exit-time computations and cancellations. As an example, a circular pool with event-record (`Pid,remtime`) could be: $(A, 5) \rightarrow (B, 6) \rightarrow (C, 3) \rightarrow (D, 4) \rightarrow (E, 8)$. E points to A a special pointer HEAD points to the next job at the head of the queue, in this case HEAD points to A.

For reasons of space constraints, we present only a brief description of the algorithms, however more details and the pseudo-code can be found in [11].

Round-Robin Service Discipline

In requesting generic round-robin serviced, 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 in Figure 4, 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 (i.e., the naïve 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 necessary book-keeping. When 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 order, the remaining service time of each has to be decreased accordingly, to reflect service these jobs have received during the elapsed time. A process invokes the function `adjust_pool(f)` to simulate work done by the multiprocessor between the arrival and the last update.

Consider the previous example based on a pool of five jobs and examine how an update will be made after an elapsed time of two quanta in a three-processors system. During the first quantum, jobs A, B and C would receive service, and during the second quantum, jobs D, E and A would receive service. Thus, at the end of two quanta, the new head pointer for the pool will point to job B and the adjusted pool will be (`Pid,remtime`): $(A, 3) \rightarrow (B, 5) \rightarrow (C, 2) \rightarrow (D, 3) \rightarrow (E, 7)$, and E points to A. The algorithm performs this adjustment for any given pool and any number of processors in the system.

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.

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. The function `delete_from_pool(f)` simulates 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.

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, 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. 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 (`clock + q`). If an arrival occurs before the time of the already scheduled departures, the departures are canceled. To circumvent such cancellations and the high cost of multiple calendar-insertions/deletions, we resort to look-ahead[8]; this is 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. Before scheduling a departure, the algorithm looks ahead to determine the time of the next arrival and proceeds with the scheduling only if the departure occurs before the next arrival.

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[5,8] to accommodate multiple processors. To do this we use the concept of state, which 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. 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 shortest remaining service-time `minrem`. The distance in steps from the pool head to departing job is stored in variable `steps`. In a multiprocessor simulation, the next job departure time may be obtained as

$$\texttt{deptime} = \texttt{clock} + \texttt{ceil(}\ \texttt{((minrem} - \texttt{1)} \times \texttt{size_of_pool(f)} + \texttt{steps)}\ /\ N\ \texttt{)} \times q$$

where `ceil()` is the standard ceiling function. As before, we resort to look-ahead 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

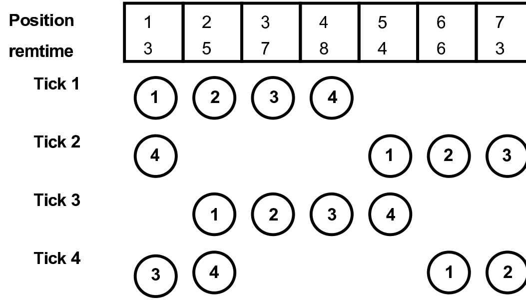

Figure 5: Example of multiprocessor service

system shown in Figure 5, with 7 jobs in the pool, and 4 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 will be N candidates. If the jobs are numbered from $1\ldots N$, the relative position of the job with the `minrem` tick requirement within the set is given by:

$$\texttt{elem = ((minrem } -1) \times \texttt{ size_of_pool(f) + steps) \% } N$$

where an `elem` value of zero is interpreted to mean `elem` has value N. Once the element with minimum tick requirement is identified, its position can be determined, and the other candidates will be predecessors in positions (1 to (`elem`–1)) and successors in positions ((`elem` + 1) 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 `tt[ ]` of pointers loaded with job records while searching for `minrem`. Once array `tt[ ]` is available, the starting job is given by

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

A value of `start` less than or equal to zero is taken to mean that `start + = size_of pool(f)`. Given this value, the start pointer of the candidate set is `start_ptr = tt[start]`. The candidate processes are the N jobs obtained by a cyclic traversal of the pool starting from `start_ptr`. Following the example in Figure 5 and applying the given formulae yields `minrem` = 3, `steps` = 1 for the job in position 1, `elem` = 3, and `start` = 6, with candidate departures being those in positions 6, 7, 1 and 2, as expected.

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.

Benchmark measurements

The naïve and computational algorithms for multiprocessor round-robin service were implemented on our testbed for performance comparison. The intent was to measure the performance of both algorithms as a function of the number of processors NPROC. 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.

The parameters used were $1/\lambda = 100$, $1/\mu = 80$, quantum $q = 1$, and $N = 20000$ jobs. The simulation was run for different values of NPROC, ranging from 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 are shown in Figure 6 and in Table 1.

Table 1: Comparison of number of average context switches for multiple-processor RR

	Number of context-switches
Naïve	1,129,992
Computational	55,673

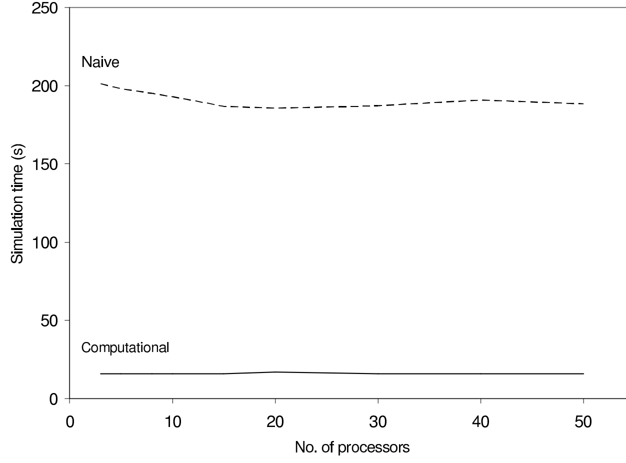

Figure 6: Performance of two multiprocessor round-robin algorithms

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. The reduction in the number of context-switches is close to a factor of 20, as evidenced by the results presented in Table 1, while the computation time is reduced by a factor of approximately 10. This suggests that computational overhead associated with the more complex data structures in the computational algorithm reduce by approximately one-half 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

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

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 $0 < \alpha < 1$. In our experiment, we obtain $K \approx 20$ and $\alpha \approx 0.5$. Solving for $T(C)$, we obtain

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

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 we can conclude 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 $c*(1-\alpha)/\alpha + d/\alpha$ which suggests that the processing of each event may require considerably more time in the computational algorithm, on average. Our experimentally obtained value of $\alpha \approx 0.5$ 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.

As the number of processors n is increased, the number of jobs handled by each processor is proportionately reduced, i.e., each processor handles $20000/n$ jobs. This implies that the number of context switches remains roughly constant, as reflected by the roughly constant graphs obtained in Figure 6 for both algorithms. 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.

4 Conclusion

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. Our study originated with our experiences with threads-based simulation and experimentation with efficient round-robin algorithms for uniprocessor queues. As the study progressed, we 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 our 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 of 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. J.B. Evans. *Structures of Discrete Event Simulation*, Ellis Horwood, Chichester, England, 1988.
2. W.R. Franta. *The Process View of Simulation*, North-Holland, Amsterdam, 1977.
3. M.H. MacDougall. Simulating Computer Systems: Techniques and Tools. *The MIT Press*, Cambridge, Massachusetts, 1987
4. H.D. Schwetman. CSIM: a C-Based Process-Oriented Simulation Language, *Proceedings of the 1986 Winter Simulation Conference*, 1986, pp. 387–396
5. J. Sang, K. Chung, and V. Rego. A simulation Testbed Based on Lightweight Processes. *Software-Practice and Experience* Vol. 24(5), pp. 485-505, May 1994.
6. E. Mascarenhas, F. Knop and V. Rego. Minimum Cost Adaptive Synchronization: Experiments with the ParaSol system. *Winter Simulation Conference*, pp. 389–396, Dec. 11–14, 1997.
7. E. Mascarenhas and V.Rego. ARIADNE: Architecture of a Portable Threads System Supporting Thread Migration. *Software – Practice & Experience*, Vol. 26(3), pp. 327–357, March 1996.
8. J.R. Ramos, J.Sang, and V.Rego. An Efficient Burst-Arrival and Burst-Departure Algorithm for Round-Robin Service, Technical Report, Purdue-CSD, December 2002.
9. Sun Microsystems, Inc. *Multithreaded Programming Guide*, Sun Microsystems, Palo Alto, California, 1997.
10. IEEE, POSIX 1003.1c API Threads Standard, 1995.
11. J.R. Ramos and V.Rego. Algorithms for Multiprocessor Scheduling on a Simulation Testbed , Technical Report, Purdue-CSD, March 2002.