Design and Implementation of Statistical Rate Monotonic Scheduling in KURT Linux[*]

Alia Atlas[†]
Dept of Internetwork Research
BBN Technologies
Cambridge, MA 02138

Azer Bestavros
Computer Science Department
Boston University
Boston, MA 02215

{akatlas, best}@cs.bu.edu

Abstract

We present the design and implementation of Statistical Rate Monotonic Scheduling (SRMS) [1] on the KURT Linux Operating System [4, 10, 9]. We overview the technical issues we had to address to integrate SRMS into KURT Linux and present the API we have developed for scheduling periodic real-time tasks using SRMS.

1. Introduction and Related Work

SRMS allows the scheduling of periodic tasks with highly variable execution times and statistical QoS requirements. It enforces task isolation (the *firewall* property) so that tasks cannot interfere with each other. SRMS wastes no resource bandwidth on jobs that will miss their deadlines due to overload conditions, resulting from excessive variability in execution times. SRMS is cognizant of the value of various tasks in the system; thus under overload conditions, the deterioration in QoS suffered by various tasks is inversely proportional to their value. The SRMS scheduling algorithm and schedulability analysis are computationally efficient. A detailed description of SRMS is available in [1].

SRMS: The SRMS algorithm consists of two parts: a job admission controller and a scheduler. Like RMS, the SRMS scheduler is a simple, preemptive, fixed-priority scheduler, which assigns the resource to the job with the highest priority that is in need of the resource. The SRMS job admission controller is responsible for maintaining the QoS requirements of the various tasks through admit/reject and priority assignment decisions. In particular, it ensures the important property of *task isolation* (temporal protection), whereby tasks do not infringe upon each other's guaranteed allowances. Job admission control occurs at a job's release time. All admitted jobs are guaranteed to meet their deadlines through a priority assignment that is rate monotonic. Jobs that are not admitted may be either discarded, or allowed to execute at a priority lower than that of all admitted jobs.

OS Support for Real-Time Scheduling: A key problem with priority-based algorithms is that there is no temporal protection; a misbehaving task can damage the performance of other tasks [6]. RMS was implemented in RT-Mach using *procesor capacity reserves* [7] and a policing mechanism. In the Rialto OS, a graph-based scheduling algorithm was implemented to provide guaranteed periodic CPU reservations as well as to guarantee aperiodic tasks [5]. Other efforts have focused on guaranteeing CPU fairness, as defined by a proportional share discipline. In [11], Yau and Lam introduced Adaptive Rate-Controlled Scheduling (RCS) and discussed an implementation in the Solaris UNIX operating system. A similar approach was given by Nieh and Lam [8].

Linux Real-Time Kernels: Linux is an increasingly popular free Unix-clone OS. In [13, 12], Yodaiken and Barabanov introduce Real-Time Linux, which supports hard real-time applications. The implementation inserts a small hard real-time kernel into the system, with the normal Linux kernel running as the lowest priority task. Interrupts are caught by the real-time kernel and are passed to the Linux kernel. This insulates real-time tasks from timing uncertainties caused by the Linux kernel. However, it also means that real-time tasks have no access to any Linux kernel services. A more general approach has been explored by Hill *et al.* in [4, 10, 9]. Their version of real-time Linux (called KURT Linux) supports firm and soft deadlines. The kernel smoothly transitions from (1) standard Linux scheduling to (2) scheduling only real-time tasks or to (3) scheduling all tasks. The scheduling of real-time tasks is done via a table of timed events. KURT Linux allows real-time tasks to access kernel functions and resources, but it provides no support for scheduling, admission control, or QoS specification/negotiation for periodic task sets. As decribed later, we have implemented all of these capabilities for KURT Linux using SRMS.

[*] Partially supported by NSF research grant CCR-9706685.

[†] Research completed while co-author was at Boston University.

2. Supporting SRMS on KURT Linux

KURT Linux is designed for non-hard-deadline (*i.e.* soft or firm deadline) real-time tasks, which may require use of kernel functions. KURT Linux provides microsecond time resolution for event scheduling. It has an API for transitioning into and out of real-time mode. To support real-time tasks, it requires that real-time tasks register and that periodic tasks undergo an admission test. The extant of real-time scheduling in KURT Linux is table-based. A file with a list of events is supplied and used for scheduling; each event consists of the time it should occur and the function which should be called at that time.

2.1. Interrupt Handling in KURT Linux

KURT Linux presents some challenges to an implementation of any real-time scheduling algorithm. While KURT Linux reduces the work done in an interrupt,[1] it does not isolate tasks from the timing uncertainties caused by such an interrupt. This is acceptable, since it is targetted to support soft/firm real-time tasks. In KURT Linux, interrupts are not delayed; they can occur at any point.

When an interrupt occurs, most interrupt service routines set a flag, indicating that work needs to be done; the system must schedule it. This remaining work is known as the *bottom half* of the ISR. In normal Linux, the *bottom halves* are completed every time the scheduler is run. Clearly, intelligent scheduling of the *bottom halves* of interrupts is necessary to minimize priority inversion. This presented a serious implementation challenge that we had to address, as described later in this section.

2.2. Assigning Overhead Costs to Tasks

SRMS as described in [1] does not consider any operating system or scheduling overheads. Operating system overheads are due primarily to the management of interrupts. As discussed above, interrupts consist of two parts—the ISR and the *bottom half.* The overheads for each one of these two parts must be treated in a different manner due to the asynchronous nature of ISR overheads versus the synchronous nature of bottom-half overheads. Scheduling overheads are due to the need of SRMS to determine which task should be scheduled next and to swap that task into the CPU.

Accounting for Scheduling Overhead: First, we considered the scheduling overhead. We assumed that a task cannot voluntarily suspend execution.[2] Each job preempts the CPU exactly once and voluntarily releases it once.

This observation provides a convenient method to upper bound the scheduling overhead of each task. When a task τ_i preempts a lower priority task, which it does once, the task τ_i is charged with the scheduling overhead. Similarly, when the task τ_i releases the CPU voluntarily to a lower priority task, task τ_i is charged with the scheduling overhead. Thus, the time it takes to run the scheduler and to swap processes is always charged to the higher priority process. If a task needs to suspend (waiting for an interrupt) the extra preemption overheads must be considered for calculation of the job's resource requirement.

Accounting for Interrupt Overheads: Certain types of real-time tasks may require that interrupts be used[3], whether it be for disk I/O or network traffic. While it is possible to mask off interrupts, it is not desirable for long periods, since meaningful interrupts may be missed. Therefore, a task will suffer overhead from interrupts. To give some perspective on the size of this overhead, the overhead of having an event timer go off and call the correct process takes over 50 μseconds, while an interrupt takes an average of 7 μseconds. The OS overhead due to interrupts can only be estimated as a function of a given job's expected execution time and the system of tasks' usage of interrupt-driven kernel services.

Dealing with Priority Inversion Due to Scheduling Interrupt Bottom Halves: When an ISR is run, it may set a flag indicating the kernel should complete some specific work, known as the interrupt's *bottom half.* Ideally, the *bottom half* of each interrupt would be run by the task which required the services supplied by that interrupt. We assume that each interrupt will wake up a given task. When the scheduler determines if a task is ready to be scheduled, it can also check if the task is waiting on an interrupt whose ISR has been run, but whose *bottom half* has not been scheduled. If so, then the scheduler could run the appropriate *bottom half,* which would wake up the task. That task would then be charged with the overhead of running the *bottom half.*

Even with this ideal situation, the problems of *priority inversion* would not be eliminated. If a higher priority task's interrupt occurs immediately after the scheduler has swapped in a lower priority task, then the higher priority task must wait for the next scheduling event. One could modify all ISRs such that the scheduler is called if the ISR is associated with a higher priority task. However, this would require modifications to all possible drivers, and would still not eliminate all priority inversion. Priority inversion is inevitable, because the higher priority task is not scheduled while it is awaiting the interrupt.

The above situation would be ideal in that each task would be charged for the execution of the interrupt *bot-*

[1] Interrupt overhead averages 7μseconds.

[2] Not permitting a task to voluntarily suspend is a common requirement in real-time scheduling; at the ready-time, the entire job must be ready.

[3] Every 10 ms a heartbeat event interrupt is scheduled to maintain any kernel services which depend on that timer.

tom halves which it required. Unfortunately, there is no support in the kernel to permit associating interrupts with the tasks which are waiting upon them. Instead, we chose a compromise design as follows.

Normally, the scheduler selects the highest priority task with work to do. To do this, the scheduler checks each task sequentially, from highest priority to lowest. In this check, if the scheduler finds a task which is waiting on an interrupt before it finds a task with work to do, then the scheduler runs the *bottom halves* of the interrupts. If the task which was waiting is awaken, then the scheduler has its selection; otherwise it proceeds. The *bottom halves* are run at most once every scheduling event.

This solution bounds the potential priority inversion to be the length of the shortest period in the system. The time to run the *bottom halves* is considered to be part of the scheduling overhead and is charged to the higher priority task of those swapped out and in. The waiting high priority task will frequently be charged the cost of running the *bottom halves.*

2.3. Task Management and Control

Enforcing Resource Requirements through Policing: An assumption of SRMS is that the scheduler has knowledge of a job's resource requirement as soon as it is released. In an actual operating system, this assumption is usually false and potentially dangerous. Therefore, rather than subtracting the job's execution time from its task's budget when the job is released and admitted, the actual execution time is subtracted from the budget once it is spent. An accurate calculation of a job's execution time will require execution time to complete, and therefore this knowledge will not be available when the job is released. The overhead to calculate the execution time, if known, can be taken into account in calculating the quality of service for the task. Malicious tasks may also lie about a job's expected execution time, in an attempt to acquire more CPU time. To protect against malicious tasks and tasks which cannot accurately compute their execution times, a policing mechanism is necessary.

The policing mechanism we employ consists of setting a scheduler event to occur immediately before the task can spend more time than is available in its budget. The delay from when the task is scheduled to this event is the task's budget minus the one scheduling overhead for releasing the CPU to a lower priority task.

```
/* returns the period of the server */
unsigned long get_server_period(void);

/* returns the utilization of the server */
float get_server_util(void);

/* returns old period if successful (else -1) */
long set_server_period(unsigned long new_period);

/* returns old utilization if successful (else -1) */
float set_server_util(float new_util);
```

Figure 1. API for conventional tasks

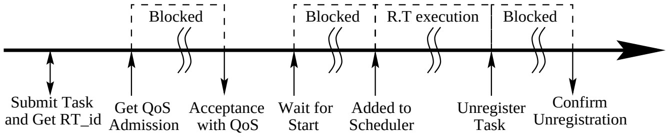

Figure 2. Timeline of a Periodic Task

Scheduler Events: The job of the scheduler is to swap in the chosen task and to set an interrupt for the next time at which the scheduler should be run. It is only necessary to specify the time of the next scheduling event. The time of the next event can be determined simultaneously when deciding which task should be given the processor. This decision is quite simple, namely the highest priority task with work to do is scheduled and the next scheduling event should occur at the earliest release time of any equal- or higher-priority task.

That calculation for the scheduling event does not consider the policing required. The policing mechanism determines the time of the budget-constrained scheduling event, as described above. The earlier of the budget-constrained scheduling event and the release time scheduling event is selected and set to trigger the scheduler at that time, using KURT Linux's built-in event timing mechanism.

Recovery from Missed Deadlines: Some jobs will not be allowed to run to completion under SRMS. This may happen either because the job was rejected or because the job attempted to use more time than was available in its remaining budget. In both cases, the process must be cleaned up so it is ready to run its next job. This clean-up could occur either at the end of the missed job or at the beginning of the next accepted job. If the clean-up occurs at the end of the missed job, then the execution time of the clean-up routine must be known. Moreover, a job may fail due to the need to guarantee the overhead to clean it up, if it fails. Therefore, we choose to consider the time required to clean-up a previously failed job as part of the execution time of the next job.

For a newly-released job, when it is scheduled for the first time, if the previous job failed, the scheduler will send the process a signal. The process will catch this signal and clean up to prepare to run the next job. When the signal handler exits, the process is manipulated so that the signal handler exits to the beginning of its periodic loop, where it will start executing the next job.

3. The SRMS API

KURT Linux provides three different modes—normal Linux, focused real-time, and mixed conventional and real-time. In this API, we assume that the mixed scheduling mode is in effect. All real-time tasks are assumed to be periodic;[4] they may appear (*i.e.* be released) in the system at any time and they may be removed (*i.e.* be terminated) from the system at any time.

To ensure that conventional tasks are not starved, a periodic server is created to service conventional tasks. By default, the server's period is set to five times the maximum real-time task period. The utilization of the server is adjustable. The functions shown in Figure 1 allow access and modification of the period and utilization for the conventional task server.

```
/* Returns granted QoS (0 if below min_QoS) */
float request_QoS_admission(desired_QoS,min_QoS);

/* Returns granted QoS or blocks until at least
   min_QoS is guaranteed. */
float await_QoS_admission(RT_id,desired_QoS,min_QoS);
```

Figure 3. API for task admission with QoS

A real-time task has three basic stages. First, it must register as a real-time task and request admission with a given minimum QoS. Once admitted, the task must execute periodically, as expected. Finally, the task must unregister when it has completed execution. The timeline of a task's existence is illustrated in Figure 2.

Registration: The registration of a task as a real-time task is a straightforward extension of what is supplied by KURT Linux. As seen in Figure 4, the `rtparams` structure is increased to include the task's importance, a pointer to an array of sample execution times, and the number of samples in that array. Once a task has called `set_rtparams`, its information is stored with the kernel.

Admission with QoS: No resources are given to a task until it has been admitted to the system. To request admission, there is a choice of a blocking call and a non-blocking call, as shown in Figure 3. An example using the non-blocking call is shown in Figure 4. The non-blocking call `request_QoS_admission` checks if the task can gain admittance with (at least) the specified minimum QoS. If this cannot be immediately guaranteed, then no resources are allocated to the task, and a QoS of 0 is returned. If a QoS between the minimum and the requested QoS can be guaranteed, then the allowance is set so as to allocate the appropriate resources to the task and the promised QoS is returned. The blocking function `await_QoS_admission` does not return until at least the minimum QoS has been guaranteed. If this is not possible when the function is first called, then the task is blocked. Whenever an admitted periodic task leaves the system (or decreases its QoS), an effort is made to admit such blocked tasks in order of importance. The `await_QoS_admission` function depends on the assumption that some tasks will eventually complete and unregister themselves.

```
void failedJob_handler(int jobnum)
  {
   /* Clean up from failed job and prepare for new one
    * On return from this signal handler, process will
    * wake up after await_scheduling() call. */
  }

int main(int argc, char * argv[])
{
  int num_samples = NUM_SAMPLE_EXEC_TIMES;
  unsigned long sample_execs[NUM_SAMPLE_EXEC_TIMES];
  struct rtparams myRTparams;
  int myRT_id;
  float myQoS;

  /* Fill in sample_execs from a file or memory. */
  myRTparams = {
    /* RT id for this process */ ASSIGN_RT_ID,
    /* rate-monotonic priority */ ASSIGN_RT_PRIORITY,
    /* importance of task (1 - 99) */ 1,
    /* array of sample exec times) */ sample_execs,
    /* length of sample exec array */  num_samples,
    /* period in microseconds */ 33000
  };

  /* Now, register with the kernel as a RT process */
  myRT_id = set_rtparams(
         /* pid, 0 if current process */ 0,
         /* process type */ SCHED_KURT,
         /* RT parameter info */ &myRTparams);
  signal(RT_JOB_FAILED, failedJob_handler);
  myQoS = await_QoS_admission(myRT_id, 80.0, 50.0);
  while (haveWork) {
    jobnum = await_scheduling();
    /* The budget returned has scheduling/OS overheads
     * subtracted for this job of the task. */
    budget = get_RTbudget();
    if (budget > CALCULATE_TIME)
      execTime = myCalculateExec(jobnum);
    /* Allow job admission control to set priority
     * of job properly. admit_RTjob returns 1
     * if the job is admitted and 0 otherwise.  */
    admitted = admit_RTjob(execTime);
    /* If job was rejected, it runs at low priority */
    haveWork = do_work();
  }
  unregisterRT(myRT_id);
  exit(0);
}
```

Figure 4. Example real-time user process

Periodic Execution: Once a task has registered and been admitted, it is ready to be scheduled. To support periodic execution of a task, we designed a function call `await_scheduling` which blocks the task until a new job of that task is released and available for scheduling. The number of the newly released job is returned. This

[4] Dealing with aperiodic real-time tasks is possible by modeling the aperiodic task as a periodic one and terminating it at the end of its first period.

```
while (haveWork) {
   jobnum = await_scheduling();
   haveWork = do_work();
}
```

```
while (haveWork) {
  jobnum = await_scheduling();
  /* The budget returned has scheduling and OS
   * overheads subtracted for this job of the task. */
  budget = get_RTbudget();
  /* myPickAlgorithm selects which algorithm
   * should be used and returns its required
   * execution time.  PICK_TIME is the time
   * needed to run the myPickAlgorithm function. */
  execTime = myPickAlgorithm(budget - PICK_TIME,
                             jobnum, &alg);
  /* Allow job admission control to set priority
   * of job properly */
  admitted = admit_RTjob(execTime);
  /* If job was rejected, the fastest algorithm was
   * picked, so try at the low priority. */
  do_work(alg);
}
```

Figure 5. API use cases: Task ignorant of exec times (top) and Design-to-time task (bottom)

function is used before any jobs are released and between the completion of one job and the release time of the next.

In addition to the proper use of `await_scheduling`, a task must either catch an `RT_JOB_FAILED` signal or use the `void ignore_jobfail_signal(/* int */ TRUE)` function to report to the scheduler that job failures should be ignored and not reported. This option is useful for a task with a *soft deadline*, which needs to complete the work of a job even after its deadline.[5] The signal handler should clean up any remnants of the failed job and restore a pristine state, as expected by a newly released job of that task.

There are three different possible task models which we have designed APIs for. The first API is to support tasks which have no method of determining what the execution time of a job will be. A sample loop for the periodic execution is given in Figure 5 (top). The second API is for a task which has accurate knowledge of its jobs' execution times and of the time it will take to compute those execution times. An example of this *default* API is shown in Figure 4. The third API is for design-to-time tasks; such tasks can select which procedure to use depending upon the time available for the execution [3, 2]. An example task is shown in Figure 5 (bottom).

Unregistration: Once a task has completed its execution, the task must notify the system. To do so, `void unregisterRT(int myRT_id)` is used. It recalculates the allowance of the next higher priority task and waits until it is safe to have that allowance changed. Then it changes the allowance of that next higher priority task and removes the time allocation of the task which is unregistering. Finally, it removes all information about the task. Once `unregisterRT` returns, the task is no longer considered a real-time task and is free to exit or continue executing, as its application demands. A simple example process, illustrating the registrating, QoS admission, signal handling, periodic execution, and unregistering is shown in Figure 4.

[5] The `ignore_jobfail_signal` is also used by the kernel during the `unregisterRT` function.

References

[1] A. K. Atlas and A. Bestavros. Statistical rate monotonic scheduling. In *IEEE Real-Time Systems Symposium*, Dec. 1998.

[2] P. Binns. Incremental rate monotonic scheduling for improved control system performance. In *Real-Time Technology and Applications Symposium*, June 1997.

[3] J.-Y. Chung, J. W. S. Liu, and K.-J. Lin. Scheduling periodic jobs that allow imprecise results. *IEEE Transactions on Computers*, 39(9):1156–1174, Sept. 1990.

[4] R. Hill, B. Srinivasan, S. Pather, and D. Niehaus. Temporal resolution and real-time extensions to linux. Technical Report ITTC-FY98-TR-11510-03, Information and Telecommunication Technology Center, Department of Electrical Engineering and Computer Sciences, University of Kansas, June 1998.

[5] M. B. Jones, D. Rosu, and M.-C. Rosu. Cpu reservations and time constraints: Efficient, predictable scheduling of independent activities. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, Oct. 1997.

[6] C. W. Mercer, R. Rajkumar, and J. Zelenka. Temporal Protection in Real-Time Operating Systems. In *Proceedings of the 11th IEEE Workshop on Real-Time Operating Systems and Software*, pages 79–83, May 1994.

[7] C. W. Mercer, S. Savage, and H. Tokuda. Processor Capacity Reserves for Multimedia Operating Systems. In *Proceedings of the IEEE International Conference on Multimedia Computing and Systems*, May 1994.

[8] J. Nieh and M. S. Lam. The design, implementation and evaluation of smart: A scheduler for multimedia applications. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, Oct. 1997.

[9] B. Srinivasan. A firm real-time system implementation using commercial off-the-shelf hardware and free software. Master's thesis, Department of Electrical Engineering and Computer Science, University of Kansas, June 1998.

[10] B. Srinivasan, S. Pather, R. Hill, F. Ansari, and D. Niehaus. A firm real-time system implementation using commercial off-the-shelf hardware and free software. In *Real-time Technology and Applications Symposium*, pages 112–119, June 1998.

[11] D. K. Y. Yau and S. S. Lam. Adaptive rate-controlled scheduling for multimedia applications. In *ACM Multimedia*, 1996.

[12] V. Yodaiken. The RT-Linux approach to hard realtime. Paper available online on the web at URL: http://rtlinux.cs.nmt.edu/ rtlinux/whitepaper/short.html.

[13] V. Yodaiken and M. Barabanov. A real-time linux. Online at http://rtlinux.cs.nmt.edu/ rtlinx/u.pdf.