A Forth-Simulator of Real-Time Multi-Task Applications

Sergey Baranov
St. Petersburg Institute for Informatics and Automation of the Russian Academy of Sciences (SPIIRAS), ITMO University[1]
SNBaranov@gmail.com

Introduction. Software applications for real-time systems (RTS) are usually built as cooperative complexes of communicating *tasks* τ_1, τ_2, … , τ_n, which share common computational and informational resources and whose behavior is impacted by *system events*, occurring from time to time according to a particular *scenario*. Each task is a sequential program, closed in itself with respect to control flow, and is activated in response to external events within some timing intervals not less than some value called its period and is expected to elaborate some response as a result of its activation and the following run. A j^{th} activation of the task τ_i ($1 \le i \le n$) means generation of a j^{th} *instance* of this task τ_i; i.e., a respective *job* denoted as ${}_j\tau_i$ for subsequent execution. When this execution terminates, it means that a respective result has been provided in response to the system event which caused activation of this particular job.

A characteristic feature of an RTS is the requirement for *on-time execution*, usually expressed as a requirement that for each task τ_i the longevity $r({}_j\tau_i)$ of any of its jobs ${}_j\tau_i$ shall not exceed some pre-defined deadline value D_i: $\forall i,j\ r({}_j\tau_i) \le D_i$. With the notion of the task *response* time $R_i = \max\{r({}_1\tau_i), r({}_2\tau_i), ...\}$ this may be reformulated as $\forall i\ R_i \le D_i$ with any allowable scenario of system events and is often interpreted as the property of *feasibility* of the given multi-task application. To check application feasibility, various *structural models* of its tasks are built and analyzed to provide reliable estimates for the response times of the application tasks, taking into account all impacting factors.

Software simulation is an acknowledged method to check feasibility of real-time multi-task applications. This paper describes an experience of constructing such simulator in Forth with the VFX Forth for Windows [1] as a development platform. Forth was selected as the implementations language due to the flexibility it provides for implementing programming solutions. The simulator employs a simple model of a multi-task application under study which may use several *scheduling modes* with various task priorities for allocation of the processor computational resource and several *access protocols* to access shared informational resources. The simulator helps to study multi-task application behavior and check whether a given combination of the scheduling mode and access protocol guarantees application feasibility under the given processor performance and system event scenarios. It may also identify the minimal processor performance which still ensures application feasibility under the given conditions.

By now, the nomenclature of scheduling modes and access protocols implemented in the simulator consists of two classical scheduling modes – RM (*rate monotonic*) and EDF (*earliest deadline first*) – and three access protocols – NI (*no inheritance*), BI (*basic inheritance*), and PI (*priority inheritance*). However, it may be further extended to simulate systems with other scheduling modes on a multi-processor and/or multi-core platform and other protocols of access to shared informational resources [2].

Source Data. Simulation is based on components of four kinds: *resources*, *tasks*, *jobs*, and *events*. Resources and tasks are entities of the application under study; jobs and events are entities created and operated on by the simulator. Resources and tasks are also represented within the simulator with respective entities. The application is assumed to run on a single processor platform with a certain processor performance *P* in terms of "the number of standard operations per second", which a particular scaling factor determining the actual processor speed is related to. Each application task τ_i is characterized by its timing period T_i – the minimal timing interval

[1] This work was partially financially supported by Government of the Russian Federation, Grant 074-U01.

between two consecutive activations of τ_i determined by the current scenario of system events, its priority $Prio_i$ – which descends with increase of i, its weight W_i – the amount of processor time needed to accomplish this task, its deadline D_i – the maximal time period for the task to be completed, and its phase Ph_i – the offset of the first activation of this task from the simulation starting moment (by default Ph_i=0). Like the processor performance P, the task weight W_i is specified in the number of standard operations, and may be converted into seconds: $C_i=W_i/P$. Obviously, $\forall i\ C_i \leq T_i$. The values T_i, D_i, and Ph_i are specified in absolute timing units (e.g., seconds) and do not depend on the processor performance P.

Application tasks may access shared informational resources identified with their unique ID numbers; however, at any moment of time a shared resource may be accessed by only one task. Tasks which do not share any informational resources are considered to be *independent* with respect to each other. To prevent simultaneous access of 2 or more tasks to a shared resource, *critical intervals* within the task code are established and guarded with special constructs of the *mutex* type, which is a particular case of Dijkstra semaphores.

With this in mind, the structure of each task τ_i is represented in the simulator as a finite series of $k(i)$ segments, each segment performing some computation within a certain period of time S_j>0 and terminating with one of the following system events: "*Lock m*", "*Unlock m*", or "*End*", *m* being the resource ID number. The duration of processing a system event is assumed to be negligibly small. A correct application should neither unlock a resource not locked by this task earlier, nor lock it again without preceding unlocking it, nor leave it locked upon task termination, and each task should terminate with the segment "*End*". Obviously, the task weight W_i equals to the sum of time periods of all its segments: $W_i = \Sigma_{j=1..k(i)} S_j$.

Task τ_1

```
<task name=‟t_1″prio=‟1″
      phase=‟5″ period=‟15″>
  <segment length=‟1″
           interface=‟m_1″
           op_type=‟lock″/>
  <segment length=‟1″
           interface=‟m_1″
           op_type=‟unlock″/>
  <segment length=‟1″
           op_type=‟end″/>
</task>
```

Task τ_2

```
<task name=‟t_2″   prio=‟2″
         phase=‟5″ period=‟35″>
  <segment length=‟9″
           op_type=‟end″/>
</task>
```

Task τ_3

```
<task name=‟t_3″prio=‟3″
      phase=‟3″ period=‟25″>
  <segment length=‟1″
           interface=‟m_1″
           op_type=‟lock″/>
  <segment length=‟2″
           interface=‟m_2″
           op_type=‟lock″/>
  <segment length=‟1″
           interface=‟m_2″
           op_type=‟unlock″/>
  <segment length=‟1″
           interface=‟m_1″
           op_type=‟unlock″/>
  <segment length=‟1″
           op_type=‟end″/>
</task>
```

Task τ_4

```
<task name=‟t_4″prio=‟4″>
                period=‟45″>
  <segment length=‟2″
           interface=‟m_2″
           op_type=‟lock″/>
  <segment length=‟4″
           interface=‟m_2″
           op_type=‟unlock″/>
  <segment length=‟1″
           op_type=‟end″/>
</task>
```

Fig. 1. Four tasks sharing 2 resources

An example of an application description in an XML-type fashion [3] is provided in Fig. 1. Here are 4 tasks τ_1, τ_2, τ_3, and τ_4, which share 2 informational resources m_1 and m_2. The code of the highest priority task τ_1 consists of 3 segments of 1 time unit each. Its first segment ends with the operation lock for resource m_1; the next segment ends with unlocking this resource and the third segment terminates the task. The code of the task τ_2 consists of the only segment of 9 time units while task τ_3 consists of 5 segments with two critical intervals to access the resources m_1 and m_2, the intervals being embedded in one another. The least priority task τ_4 consists of 3 segments and accesses only the resource m_2.

Task periods T_1, T_2, T_3, and T_4 for task activations are 15, 35, 25, and 45 time units respectively with the phase shifts 15, 35, 25, and 45; deadlines are assumed to be equal to task periods: $D_i=T_i$. Tasks and resources are rendered by objects of the type *task* and *resource* respectively and are created by respective Forth words during simulator initialization when reading an input file with the task descriptions:

```
: CreateTask ( -- task-addr)
: CreateResource ( n -- resource-addr)
```

Output Data. For each task τ_i the derivative characteristics are defined: its utility load $U_i=C_i/T_i$ and its hardness $H_i=T_i/D_i$ which characterize tasks execution. If $H_i \leq 1$ then the existence intervals of consecutive jobs ${}_j\tau_i$ and ${}_{j+1}\tau_i$ created from two consecutive activations of the task τ_i do not intersect. The reverse condition $H_i > 1$ means that they may intersect. An important metric – the density of the whole application: $Dens=max_P (\Sigma_{i=1..n} U_i)$ – may be calculated too, in order to compare different application structures and implementations on their efficiency [4].

The ultimate purpose of simulation is to obtain data on efficiency of various combinations of scheduling modes and access protocols in various scenarios of system events. In particular, the dual problem to calculating the application density – to determine the minimal processor performance which still ensures the feasibility of the application (i.e., that $\forall i\ R_i \leq D_i$) under given conditions – may be solved as well.

To calculate the application density, the initial interval [a,b] for selecting the scaling factor $f \in$[a,b] for the task weights and processor performance is established. Prior to the simulator run, the source values of task segment durations S_j (and therefore, the task weights W_i) in task descriptions and the processor performance P are multiplied by this factor. Obviously, if the inequality $R_i \leq D_i$ is violated for some i at the end-values a and b of the interval, it is violated for all intermediate values. However, for $f=a=0$ (which means an infinitely high processor performance) these inequalities do hold for all i. Therefore, the initial values are set to $a=0$ and $b=\Sigma_{i=1..n} U_i$ with the standard processor performance $P=10^6$ standard operations per second. Then the first simulation iteration is performed with the scaling factor $f=(b–a)/2$. If no violations of $R_i \leq D_i$ occurred, then a is set to f, otherwise b is set to f and simulation is reiterated until the scaling interval shrinks to just one value [a, a+1] in which case the scaling factor equals to this found value a, the application density is calculated accordingly, and the minimal processor performance P which still ensures the application feasibility is $P=a\times10^6$ operations per second. It usually takes from 5 to 15 simulations to reach the resulting values.

Data Structures. The simulator uses ordered chained lists whose elements consist of 3 cells: the link to the next list element or NULL, the ordering value and the data specific to the list. Elements in a list are ordered with respect to the ordering value, starting with the smallest one. Lists are defined with the defining word `List`:

```
: List ( list-element-size, max-list-length -- )
```

and use respective “methods” to add and retrieve elements in lists created by this word:

```
: >List ( new-elem-addr, list-addr -- )
        Place a new element into the ordered list
: List@ ( list-addr-- elem-addr)
        Get the first (heading) element of the list
: List> ( list-addr-- elem-addr)
        Delete the first element from the list
: List>> ( ordering-value, list-addr--)
        Find and delete a list element with this ordering value
```

Static objects (tasks and resources) are created at the simulator initialization from the task description file and are modified during simulation.

A resource is rendered with an object of 4 cells: its ID number, its priority (reserved for future use), its status (either `NULL` if the resource is currently unlocked, or a reference to the job description, which currently owns this resource and locked it), and a possibly empty ordered list of job descriptions, currently waiting for this resource to become unlocked. Resources are stored in a special pool which allows to easily enumerate them and to add a new one.

Tasks are represented with objects of various length which depends on the number of task segments. It starts with 10 cells followed by a series of 4 cells for each task segment. The initial 10 cells contain: task unique ID number *i*, task period T_i, task weight in the number of standard operations W_i, task weight in seconds C_i (depends on the scaling factor *f*), task priority $Prio_i$, task response time R_i (is calculated during simulation), task deadline D_i, task phase Ph_i, the number of executed task activations , and the number of task segments. The 4 cells for each task segment are: segment type (*Lock*, *Unlock*, or *End*), segment parameter (the resource ID for *Lock*/*Unlock* and zero for *End*), segment weight in the number of standard operations S_j, and the segment time in seconds (recalculated while scaling the task data with the scaling factor *f*).

Dynamic objects (jobs and events) are created during simulation sessions as needed with the words `CreateJob` and `CreateEvent` :

```
: CreateJob ( task-addr--job-addr)
: CreateEvent
( resource-addr, job-addr, task-addr, event-type, event-time --
  event-addr)
```

The job object is represented with 10 cells: the job unique ID, its current priority (it may change with the priority inheritance scheduling mode), current segment number which specifies the segment begin executed, current segment expected termination time, current segment start time, current segment used time, current segment time yet to be used, reference to the respective task, number of references to the job description, and a reference to a resource which this job is waiting for or `NULL` if the job is not waiting for a resource. Jobs waiting for the processor form a chained list `JobList` in the order of their current priorities. The first job in this list owns the processor and is considered as the current one. When this list is empty, the processors stays idle.

System events are characterized by the time when they occur. Events with the same timing form a group of time-sake events. Four types of system events are considered: *to activate* a task (i.e., to form a job for this task and add it to the list `JobList` of active jobs waiting for the processor), *to terminate* the current job (and pass the processor to another job in list `JobList`, if any), *to lock* a resource, or *to unlock* a resource – and these activities are performed with respective Forth words:

```
: TaskActivate ( task-addr--)
: JobTerminate ( job-addr--)
: ResourceLock ( resource-addr, job-addr--)
: ResourceUnlock ( resource-addr, job-addr--)
```

The event object which represents a system event consists of 6 cells: the event unique ID, the scheduled time for this event to occur, the type of the event (*Activate*, *Lock*/*Unlock*, or *End*), a reference to the task object to be activated or `NULL`, a reference to the job object to be ended or `NULL`, and a reference to the resource object to be locked/unlocked or `NULL`. The chained list `EventList` of system events ordered with respect to their time moments when they scheduled to occur is maintained by the simulator.

The Simulator. Simulator initialization consists in selecting the desired combination of the scheduling mode and access protocol, setting the respective simulator constraints, reading the task description file, and forming the respective resource and task objects. Then the initial list of system events `EventList` is formed which consists in activation of the all tasks at the moments of system time defined by their phase shifts. Counts for their maximal response times are set to zero and all resources are set to be unlocked.

The major simulator loop does the following. The first group of time-sake events in the `EventList` is considered, the simulator system time is set to this time moment and all system events from this first group are processed one-by-one. Processing depends on the event type: activate a task, terminate a job, or lock/unlock a shared resource.

Activating a task. A new job is created from this task referred to by the event with its planned starting time equal to the current system time and is added to the `JobList` with its priority, while a new event is added to the `EventList` – to activated the next copy of this task at the moment of time not less than the current time plus the task period T_i.

Terminating a job. The response time of the task referred to by the respective job object is updated: the difference between the current system time and the moment when this job was created and added to `JobList` (the response time which consists of the time when the job owned the processor plus the time it waited for it) is calculated and the maximum of this value and the response time already stored in the task referred to is stored as the new value of the task response time. If this exceeds the task deadline D_i, then a violation of the task feasibility is registered. The considered job is deleted from the `JobList`.

Locking a resource. If the resource is unlocked, then it becomes locked by this task; otherwise, the job is moved from the `JobList` to the ordered list of jobs waiting for unlocking of this resource.

Unlocking a resource. If the ordered list of jobs waiting for unlocking of this resource is not empty, then the first job form this list is moved from it back to the `JobList` according to its priority and the resource becomes locked by this job; otherwise, the resource becomes unlocked.

Upon completion of the event processing, the considered event is deleted from the `EventList`. After all time-sake events have been processed, the `JobList`, which may have changed as a result of previous event processing, is considered unless it is empty.

If the `JobList` is not empty then the first job from it (which currently owns the processor) is selected and the residue of the processor time not yet consumed by its current segment is considered. This value determines the moment of the segment termination. If this value is greater than the time of the next time-sake group of system events in the `EventList` then this residue is decremented by the remaining time till this event group; otherwise, a new event corresponding to this segment is added to the `EventList` for this moment of segment termination and the next job segment if any becomes its current segment.

Emptiness of the `JobList` means that the processor is idle from this moment till the next time-sake event group in the `EventList`. Upon completion of processing the first job of `JobList` (if any) the major loop is reiterated. The loop terminates upon exhausting the time limit of the simulation session or when a specified number of created jobs is reached (which of these conditions occurs earlier, if both limits are specified).

```
TimeLimit=25 JobLimit=0 ViolationLimit=1
SchedulingMode=RM InheritanceMode=NI
Configuration file name: c:\MPE\App_4t2r.txt
Time=0 Proc=0 for 0 A 4.1
Time=2 Proc=4.1 for 2 L 4.1 of 2
Time=3 Proc=4.1 for 1 A 3.2
Time=4 Proc=3.2 for 1 L 3.2 of 1
Time=5 Proc=3.2 for 1 A 1.3 A 2.4
Time=6 Proc=1.3 for 1 W 1.3 of 1
Time=15 Proc=2.4 for 9 E 2.4
Time=16 Proc=3.2 for 1 W 3.2 of 2
Time=19 Proc=4.1 for 3 U 4.1 of 2 L 3.2 of 2
Time=20 Proc=3.2 for 1 U 3.2 of 2
Time=21 Proc=3.2 for 1 U 3.2 of 1 L 1.3 of 1
Time=22 Proc=1.3 for 1 U 1.3 of 1
Time=23 Proc=1.3 for 1 E 1.3
Time=24 Proc=3.2 for 1 E 3.2
```

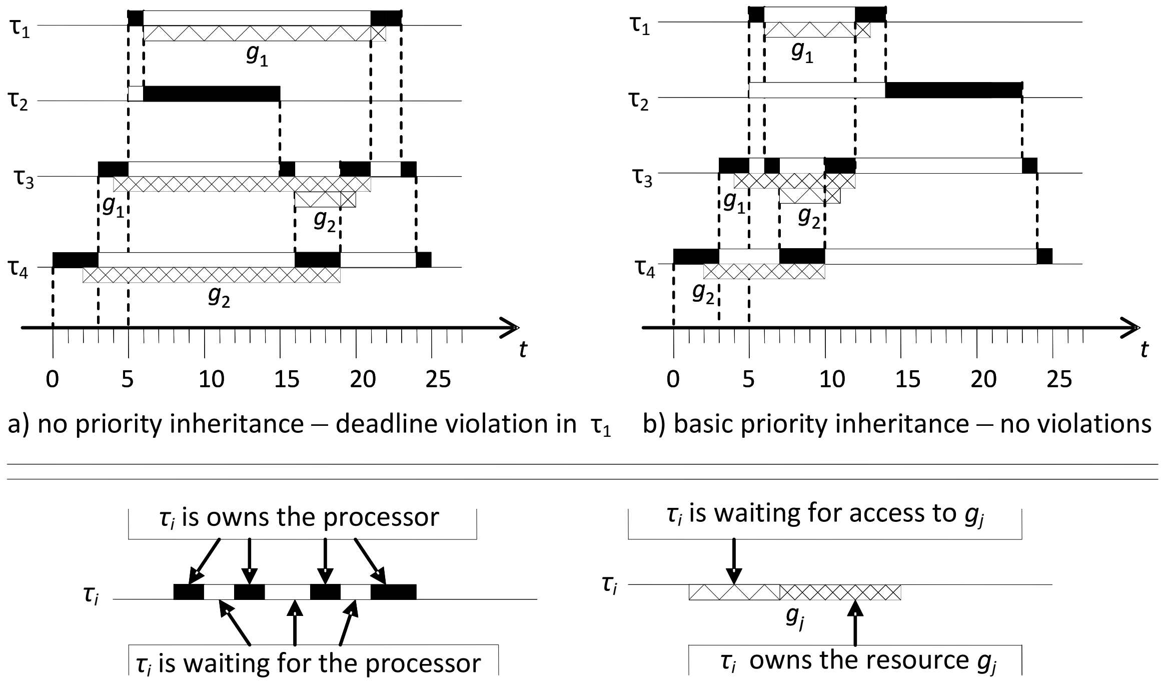
```
TimeLimit=25 JobLimit=0 ViolationLimit=1
SchedulingMode=RM InheritanceMode=BI
Configuration file name: c:\MPE\App_4t2r.txt
Time=0 Proc=0 for 0 A 4.1
Time=2 Proc=4.1 for 2 L 4.1 of 2
Time=3 Proc=4.1 for 1 A 3.2
Time=4 Proc=3.2 for 1 L 3.2 of 1
Time=5 Proc=3.2 for 1 A 1.3 A 2.4
Time=6 Proc=1.3 for 1 W 1.3 of 1
Time=7 Proc=3.2 for 1 W 3.2 of 2
Time=10 Proc=4.1 for 3 U 4.1 of 2 L 3.2 of 2
Time=11 Proc=3.2 for 1 U 3.2 of 2
Time=12 Proc=3.2 for 1 U 3.2 of 1 L 1.3 of 1
Time=13 Proc=1.3 for 1 U 1.3 of 1
Time=14 Proc=1.3 for 1 E 1.3
Time=23 Proc=2.4 for 9 E 2.4
Time=24 Proc=3.2 for 1 E 3.2
```

```
Time=25 Proc=4.1 for 1 E 4.1                      Time=25 Proc=4.1 for 1 E 4.1
Time=25 Hardness=1,0000  1/Hardness=1,0000        Time=25 Hardness=1,0000  1/Hardness=1,0000
Density=0,6056  ScalingFactor=1,0000              Density=0,6056  ScalingFactor=1,0000 ok
ERROR: Deadline violation in Task 1  ok
```

Fig. 2. Logs of two simulation sessions as they are output by the simulator

The results of simulation – task maximal response time, number of deadline violations, the application density, and other statistics data are displayed. A simulation log may also be displayed. When any system event is processed, the respective time and other accompanying data are printed-out. All these data may be easily copied into MS Excel for a graphical representation of the obtained results and execution log.

There are the two logs of simulator runs in Fig. 2 – for two different protocols of access to shared resources: NI (no inheritance) and BI (basic priority inheritance) as they are recorded by the simulator. The number after "Time=" is the time of an occurring system event denoted by one of the letters: A – activate, E – end, L – lock, U – unlock, or W – wait to lock an already locked resource, followed by the event parameter. The job ID is displayed as two numbers (the task number and the unique job number separated with a period). The section "of" is followed by the resource number to be locked or unlocked, while a number after "for" is the activity duration terminated with this event. Same logs are presented in Fig. 3 in a more readable graphic form.

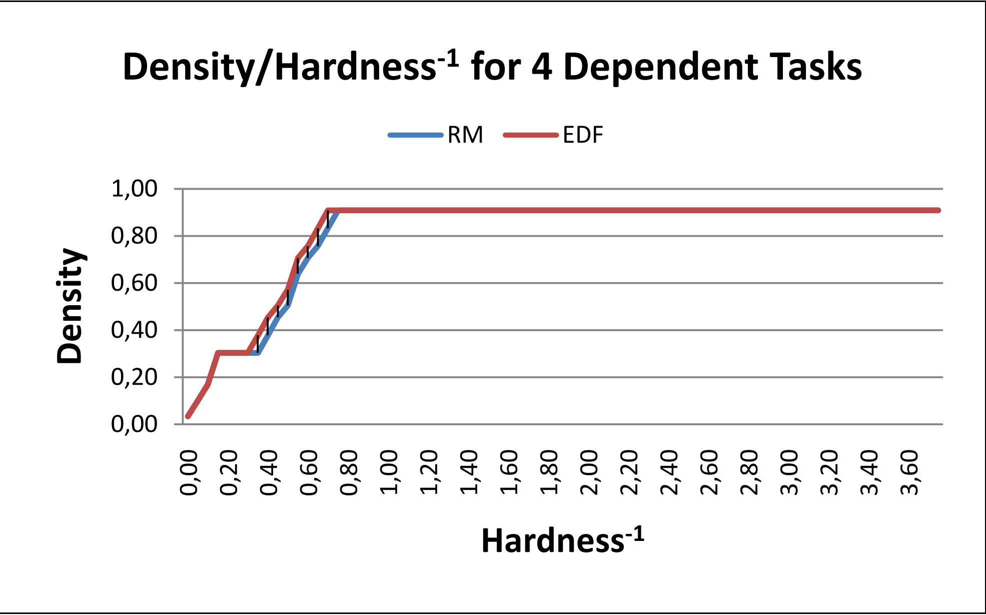


τ_1
τ_2
τ_3
τ_4
g_1
g_2
t
0
5
10
15
20
25
a) no priority inheritance – deadline violation in τ_1
b) basic priority inheritance – no violations
τ_i is owns the processor
τ_i is waiting for the processor
τ_i is waiting for access to g_j
τ_i owns the resource g_j
τ_i
g_j

Fig. 3. Simulated execution of 4 tasks sharing 2 resources with different access protocols

This application, when simulated twice with different access protocols, demonstrates two different behaviors: a violation of the specified deadline 15 for the highest priority task τ_1 under the protocol NI – Fig. 3a, and correct work with no violations under the protocol BI – Fig. 3b.

Fig. 4 compares two scheduling modes for the same application of 4 tasks and 2 shared resources defined in Fig. 1. The output simulation data were copied into an Excel file to obtain these charts. Data for application hardness and respective density values for the two scheduling modes are in the right columns of the chart. As one can see, there's no big difference in the application density between the two scheduling modes RM and EDF for this application. Density as a function of *hardness*$^{-1}$ grows nearly linearly with two plateaus and then the growth stops after *hardness*$^{-1}$=0.75. As one can see, this application cannot reach 100% density – its maximum is 0.9083 with the application *hardness*=1/0.75= 1.33 and it does not change with further decrease of hardness (i.e., increase of hardness^{-1}), which means that the processor would be inevitably idle for at least ≈10% of time while executing this application.



1/H	RM	EDF
0.00	0.0336	0.0336
0.05	0.1009	0.1009
0.10	0.1682	0.1682
0.15	0.3028	0.3028
0.20	0.3028	0.3028
0.25	0.3028	0.3028
0.30	0.3028	0.3028
0.35	0.3028	0.3784
0.40	0.3784	0.4541
0.45	0.4541	0.5046
0.50	0.5046	0.5719
0.55	0.6392	0.7065
0.60	0.7065	0.7568
0.65	0.7568	0.8326
0.70	0.8326	0.9083
0.75	0.9083	0.9083

Fig. 4. RM vs. EDF for same application of 4 tasks with 2 resources

Four Dining Philosophers. This classical puzzle, first proposed by E.Dijkstra as "Five Dining Philosophers" [5], demonstrates the situation of mutual blocking under certain scenarios of dependent task behavior with any number $n \geq 2$ of the respective processes. Let's consider 4 iterative processes, each with two alternate activities called "think" and "eat", the latter assuming simultaneous access to 2 of 4 shared resources (called the left and the right fork for this philosopher) for a certain period of time. Access to the resources is performed via critical intervals guarded with respective mutexes.

With the proposed technique this may represented as 4 tasks τ_1, τ_2, τ_3, and τ_4 (the philosophers), which share 4 informational resources r_1, r_2, r_3, and r_4 (the forks). Task phases are 10, 7, 4, and 1 respectively; 2 units after its start the task τ_1 locks the resource r_1 and after 4 units more it locks the resource r_2. Then after 20 time units it unlocks r_1 and in 68 units more it unlocks r_2. After 1000 time units or more since its start, the task τ_1 reiterates. Other tasks behave similarly with 73, 79, and 85 time units rather than 68 for unlocking their second resource (left fork). In the formalism of Fig.1 the behavior of task τ_1 may be specified as (others are similar):

```
<task name="t_1" phase="10" period="1000">
   <segment length=2 interface="r_1" op_type="lock"/>
   <segment length=4 interface="r_2" op_type="lock"/>
   <segment length=20 interface="r_1" op_type="unlock"/>
   <segment length=68 interface="r_2" op_type="unlock"/>
   <segment length=2 op_type="end"/> </task>
```

With the specified phases and timings for locking/unlocking resources, a clinch occurs at time=25, as Fig.5 displays this with the log obtained by the simulator.

System Log	Interpretation/Comments

TimeLimit=1000000 JobLimit=0 ViolationLimit=0
SchedulingMode=RM InheritanceMode=PI
Configuration file name: c:\MPE\App_4PhD.txt
	Rate Monotonic with Priority Inheritance

Time=1 Proc=0 for 1 A 4.1	Task 4 (job 4.1) is activated at time=1
Time=3 Proc=4.1 for 2 L 4.1 of 4	Task 4 (job 4.1) locks resource 4 at time=3
Time=4 Proc=4.1 for 1 A 3.2	Task 3 (job 3.2) is activated at time=4
Time=6 Proc=3.2 for 2 L 3.2 of 3	Task 3 (job 3.2) locks resource 3 at time=6

System Log	Interpretation/Comments
Time=7 Proc=3.2 for 1 A 2.3	Task 2 (job 2.3) is activated at time=7
Time=9 Proc=2.3 for 2 L 2.3 of 2	Task 2 (job 2.3) locks resource 2 at time=9
Time=10 Proc=2.3 for 1 A 1.4	Task 1 (job 1.4) is activated at time=10
Time=12 Proc=1.4 for 2 L 1.4 of 1	Task 1 (job 1.4) locks resource 1 at time=12
Time=16 Proc=1.4 for 4 W 1.4 of 2	Task 1 (job 1.4) waits for resource 2 at time=16
Time=19 Proc=2.3 for 3 W 2.3 of 3	Task 2 (job 2.3) waits for resource 3 at time=19
Time=22 Proc=3.2 for 3 W 3.2 of 4	Task 3 (job 3.2) waits for resource 4 at time=22
Time=25 Proc=4.1 for 3 **Mutual clinch for job 4.1 on resource 1** ok	Clinch detected for task 4 (job 4.1) when it tried to lock resource 1 at time=25

Fig. 5. System log for the 4 philosophers puzzle

The resource status displayed by the word `.resources` confirms this clinch. As one can see there's a vicious circle of locked resources with mutually waiting jobs:

```
Resource_1 Prio=0 Status=Job 1.4 JobsWaiting=NULL
Resource_2 Prio=0 Status=Job 2.3 JobsWaiting=Job 1.4
Resource_3 Prio=0 Status=Job 3.2 JobsWaiting=Job 2.3
Resource_4 Prio=0 Status=Job 4.1 JobsWaiting=Job 3.2
```

Conclusions. The simulator was written in Forth with VFX Forth for Windows, version 4.70, provided to the author at the courtesy of MPE [6], and is just 985 lines of code under the respective coding standards. It uses only fixed-point arithmetic and works remarkably fast on a PC. To avoid memory overflow, the simulator uses its own simple subsystem for memory allocation and reuse for chained list elements, jobs and events. Further work will be focused on improving the user interface, extending the nomenclature of scheduling modes and access protocols of this simulator, and transition to simulation of multi-core and multiprocessor platforms, as well as running more experiments with models of real-time multi-task applications.

References.

1. VFX Forth for Windows. User manual. Manual revision 4.70, 19 August 2014. – Southampton: MicroProcessor Engineering Limited, 2014. – 429 p.
2. Andersson B., Baruah S., Jonsson J. Static-Priority Scheduling on Multiprocessors // Proc. of 22nd IEEE Real-Time Systems Symposium. – London, 2001. – P.193-202.
3. Nikiforov V.V., Shkirtil V.I. Specification of interfaces in real-time software applications by XML forms. // SPIIRAS Proceedings, 2009, issue 11. – P. 159-175. (In Russian.)
4. Baranov S.N., Nikiforov V.V. Density of Multi-Task Real-Time Applications // Proceedings of the 17th Conference of Open Innovations Association FRUCT, Yaroslavl, Russia, 20-24 April 2015. – P.9-15.
5. Dijkstra E.W. Hierarchical ordering of sequential processes. Acta Informatica 1(2), 1971. – P.115-138.
6. MicroProcessor Engineering Limited. Company site http://www.mpeforth.com .

About the Author. Sergey N. Baranov graduated with honor the Leningrad State University in 1972, worked at this University, at SPIIRAS, Motorola, St.Petersburg State Polytechnic University; PhD since 1978, Doc.Sci since 1991, Professor since 1993. Currently he works at SPIIRAS as a Chief Research Associate and a lecturer at 3 major St. Petersburg Universities. His major scientific interests are software engineering, compilers, analysis and verification of software specifications, formal methods, symbolic computations, and Forth. He is an active member of the international Forth community after publishing in 1988 the monograph "The Programming Language Forth and its Implementations", the first one on Forth to appear in Russian.