Singularity
Design Note
10

Resource Management and Scheduling

Design principles, decisions, and rationale for Singularity's Resource Management and Scheduling facilities

This design note describes the abstractions and interfaces by which Singularity performs real-time resource management and scheduling. Resource management and scheduling are defined in a general way such that they can apply to all resources that affect the timeliness of a soft real-time applications' execution.

1. Introduction

1.1. Motivation and Goals

One of the primary goals of Singularity is to make it possible for soft real-time applications to operate correctly. For instance, for an application such as a video recording application, if video frames or audio samples to be recorded are dropped, then the user won't say "oh well, it's only a soft real-time application" – they'll say "this application is broken". We want our users to consider that Singularity's soft real-time applications work reliably – the opposite of "broken".

We want to be able to have multiple real-time applications coexist, with each meeting their deadlines, or if not possible, with the system informing the application(s) up front that insufficient resources are available for all deadlines to be reliably met. Even single applications, such as real-time conferencing applications or multimedia games are likely to need to support multiple independent real-time activities.

Listing them, the primary goals behind the design and interfaces presented in this document are to:

- Make it possible to develop independent real-time applications independently on Singularity, while enabling their predictable concurrent execution, both with each other and with non-real-time applications.
- Enable practical management of multiple resources by applications, including accurate usage accounting and the ability to reserve resources.
- Support scheduling and resource management for non-real-time interactive applications as well.

Note that scheduling and resource management are cross-cutting issues, affecting many aspects of the system, such as memory management, GC, I/O, etc. Initial resources we are considering support for are CPU, I/O bandwidth, network bandwidth, and power/energy.

1.2. Non-Goals

It is explicitly outside our chosen design scope to support safety-critical hard-real-time applications.

1.3. Driving Scenarios

Some of the usage scenarios the Singularity scheduling and resource management facilities are intended to enable are:

- Simple servers that mostly receive requests, work on them until completion, and then reply to them.
- Servers that are themselves resource schedulers that receive requests and may queue or reorder them based on the reservations for the resource they are scheduling and based on the specific characteristics of the resource, like disk sector order.
- Non-RPC style servers that receive more complex streams of requests and/or generate streams of replies.
- Clients that use ongoing resource reservations but declare no tasks with specific deadlines.
- Clients that declare tasks with specified deadlines for specific pieces of work during their execution.
- Clients that use no reservations or tasks.
- Clients that use both reservations and tasks.
- Clients that want to operate synchronously with respect to the execution of services that they request via channels.
- Clients that want to operate asynchronously with respect to the execution of services that they request, and in particular, that want to perform asynchronous I/O operations.

2. Definitions

Resource: A facility or service available to programs where the quantity of the facility is limited and must/may be managed. Examples include CPU Time, Disk Bandwidth, Network Bandwidth, and Memory.

Resource Container: We have chosen to use this term, introduced in [Banga et al. 99], for the abstraction to which resources are reserved and from which they are consumed. This is the same notion as Rialto's [Jones et al. 97] "Activity".

Task: An individual application step or sequence of actions that the program desires to be completed by a specified deadline. The application specifies estimates of the resources needed to carry out the task at task inception. When the task finishes the system informs the application of the actual amounts of resources consumed performing the task. A task is like Rialto's "Time Constraint", but generalized to also apply to resources other than just CPU time.

Resource Reservation: A guaranteed minimum amount of a specified resource to be provided to the target of the reservation within a specified timeframe. Reservations come in two forms: recurring and one-shot. Recurring reservations specify that at least the reserved quantity of the resource will be available to runnable threads in the resource container on an ongoing basis. One-shot reservations specify that at least the reserved quantity of the resource will be made available to threads running within the task the reservation was made for by the tasks deadline.

Scheduling Inheritance: The notion that if thread B is doing work for thread A, that B inherits A's scheduling attributes and is scheduled with at least the same urgency as A would have been. This is a generalization of the notion of "priority inheritance", with the difference being that all scheduling attributes, and not just priorities, are inherited between threads.

Activity: This term has meant different things to different people during the Singularity discussions. It has meant what we are now referring to as "Resource Containers" to many people but has also meant what we're now referring to as "Tasks" to other people or in other contexts. We have opted not to use this term within the Singularity system.

3. Key Decisions and Rationale

Scheduling decisions will be made using deadlines and resource reservations – not priorities. There is no plan to introduce any notion of priorities. The rationale behind this decision is that the correct use of priorities requires global knowledge of all tasks being executed in the system – something that is impossible in an open system. Whereas having programs make statements about their own resource needs and deadlines conveys useful, locally-determinable information that is true and actionable.

Multiple resource schedulers will be implemented – not just a scheduler for the CPU resource. The rationale for this decision is that the critical resource many time-dependent tasks is not the CPU, but can be I/O activities such as disk and/or network I/O.

Each thread in the system will be created to run within a resource container specified at thread creation time. Resource containers may have multiple threads associated with them, potentially spanning multiple processes. The rationale behind this decision is that resource containers will provide a useful abstraction that reservations are made for and that work is charged against.

Threads will also always be working on behalf of a specified task or their resource container, if the thread is not currently executing for a specific task. It is likely that each resource container will have a "default task", such that we will be able to always account for threads as if they are working on a task. The rationale behind this decision is that we want it to always be clear where to charge work against.

Scheduling inheritance will be implemented so that when a task's timely execution is dependent upon services provided by other threads, that those other threads will be scheduled as if they are also working on the task. This is to be true for all resource schedulers, including those scheduling non-CPU resources such as network and disk I/O. Scheduling inheritance will be applied both to channel communications where "start of request" and "last reply" messages can be determined and to local thread synchronization abstractions such as mutexes. The rational behind this decision is that we want to prevent situations where problems analogous to priority inversions would otherwise occur. By implementing scheduling inheritance, if A is waiting for B, then B is scheduled with at least the same urgency as A, preventing such inversions.

We can make due with explicit Receive calls provided that the thread that will do the Receive on any particularly channel is declared to the system before any messages could arrive on the channel. The rationale behind the need to be able to identify the receiving thread at the time a message is sent is so that scheduling inheritance can be performed from the sender to the thread that will receive the message.

Furthermore, if messages are queued on a channel that the thread bound to that channel will need to inherit the scheduling attributes of all the queued messages in the channel. This is also to prevent problems analogous to priority inversion.

Clients can do an arbitrary number of message Send calls without necessarily relinquishing the CPU. The rationale behind this decision is that only the client knows when it needs to block on a reply and when it needs to it will do so explicitly.

If a contract can be used in an RPC style the normal code generated will specify that scheduling inheritance be done on the first send and that the inheritance will end upon sending the last reply. The rationale behind this decision is that we want to be able to perform scheduling inheritance even for contracts where requests and replies may consist of multiple messages, and not just a single message in each direction.

Resource amounts are resource-specific quantities, such as cycles for CPU and I/O transfer amounts for disk and network. Whereas deadlines are always expressed in absolute wallclock times using DateTime values. The rationale behind this decision is that no common unit exists for expressing quantities of the different resources, therefore each resource must specify its own units and means of describing the quantities that can be reserved and amounts of work performed.

We decided to definitely not do a grand all-knowing-all-seeing multi-resource scheduler, but instead, to have each resource schedule itself using a laxity-based scheduler to try to do as well in multiple resource schedules as possible. The rationale behind this decision was that we considered the non-modularity and complexity of trying to schedule all the diverse possible resources in a single piece of code to be untenable.

We will try to use a form of Borrowed Virtual Time (BVT) scheduling [Duda & Cheriton 99] for the second chance task queue. The rationale behind this decision is that it should maximize the chance of multi-resource scheduling working, even in the presence of blocking.

We may do periodic scheduling as a sequence of automatically generated tasks. The rationale behind this possibility is that it would provide a unified scheduling paradigm for both one-shot and recurring scheduling needs.

We're happy with the resource management and task APIs we've discussed because they will admit a wide range of possible schedulers under the hood without changing applications, letting us easily prototype different schedulers and measure the tradeoffs between them. For instance, a Rialto-style scheduler could be plugged in and used without changes to applications.

3.1. Options Considered and Rejected

This section documents possible decisions considered and rejected by the group.

We considered designs where threads would be working for multiple sets of tasks at once and where we would schedule using Earliest Deadline First (EDF) of all the task deadlines. We realized that the problem with this is that EDF doesn't take into account situations where multiple resources need to be scheduled sequentially.

We realized that a Rialto-style pre-computed schedule likely won't work well in a multi-resource environment. The problem is that if a thread is waiting for a non-CPU resource that it can miss its pre-computed CPU scheduling slots. This led us to decide to use BVT on the second chance queue with a laxity based scheduler as the primary algorithm.

We considered having applications provide estimates not just of the amounts of resources needed for their tasks, but also the order that the resources would be needed in and which resources overlap how. This seemed just too complicated for application writers ever to get it right and so we rejected it.

4. Interfaces

This section gives pseudocode APIs for the resource management and scheduling interfaces provided by the Singularity system.

4.1. Resource Container Interfaces

```
// Creates a new Resource Container object
resourceContainer = new ResourceContainer();

// Returns the Resource Container associated with the running thread
resourceContainer = CurrentResourceContainer();

// Returns the default task object used by threads with a
// Resource Container when not working on any program-specified task
task = resourceContainer.DefaultTask();
```

4.2. Thread Interfaces

```
// Starts a new thread running that is associated with the specified
// Resource Container.  If resouceContainer is null,
// CurrentResourceContainer() is used.
thread = new Thread(resourceContainer, entryPoint);

// Returns the thread currently executing
thread = CurrentThread();
```

4.3. Task Interfaces

```
// Creates a new task for the current thread, specifying estimated
// resource amounts needed to complete the task and its deadline
// in absolute wallclock time.  Typical values for parentTask are
// CurrentTask(), which says that the new task is a subtask of the
// thread's current task, and
// CurrentResourceContainer().DefaultTask(), which specifies that
// the new task is an independent task within the thread's
// resource container.  The succeeded value indicates whether all
// the resources specified in the set of estimates were
// successfully reserved for the task.  (Resource accounting for
// the task will be done even if the reservations did not
// succeed.)
(task,   succeeded)   =   new   Task(parentTask,   resourceEstimates[],
deadline);

// Returns the Task the calling thread is presently working on
task = CurrentTask();

// Indicates that the current thread is now working on the
// specified task.  Causes both scheduling inheritance and
// resource accounting for the task.
task.WorkingOn();

// Indicates that the current thread is no longer working on the
// specified task.  This causes the thead‘s current task to be
// set to the task’s parent task.
task.NotWorkingOn();

// Tells the system that the specified task has completed and
// returns the actual amounts of resources used to accomplish the
// task.
actualAmounts[] = task.Finished();

// Tells the system explicitly that the specified task is blocked
// waiting on the thread, explicitly causing scheduling
// inheritance.  (This also happens implicitly in some cases.)
// Causes scheduling inheritance to the thread but not resource
// accounting for the task.
task.BlockedOn(thread);

// Tells the system that the task is no longer blocked on the thread
task.NotBlockedOn(thread);
```

4.4. Composite Task Interfaces for Common Cases

```
// Typical case shorthand for new Task(...) followed by
// task.WorkingOn()
(task,   succeeded)   =   BeginTask(parentTask,   resourceEstimates[],
deadline);

// Typical case shorthand for task.NotWorkingOn() followed by
// task.Finished()
actualAmounts[] = task.End();
```

4.5. CPU Resource Interfaces

```
// Convert CPU cycles to a resource amount.
// Can be used in task resource estimate arrays.
cpuResourceAmount = CPUAmount(cycles);

// Convert CPU resource amount to cycle count
cycles = CPUCycles(cpuResourceAmount);

// Return amount of CPU available for reservation
cycles = AvailableCPU(resourceContainer, period);

// Reserve CPU on an ongoing basis for a resource container.
// Replaces existing reservation.  Use 0 cycles to end reservation.
success = ReserveCPU(resourceContainer, cycles, period);
```

4.6. Other Possible Interfaces

We may support a MostUrgentTasks() call, returning an ordered list of the most urgent N tasks needing to be scheduled by a thread/process/resource container, where N would typically be the number of physical processors on the machine. This would be intended to facilitate multi-resource scheduling.

5. Example Code Sequences

This section gives examples of how the resource management interfaces are expected to be used.

5.1. Normal client Task usage

```
// Compute resourceEstimates based upon past usage.
// Determine deadline based upon program’s real-time requirements.
(task,   succeeded)   =   BeginTask(CurrentTask(),   resouceEstimates,
deadline);
if (succeeded)
    // Do normal work for task...
else
    // Do less work if possible, since task is unlikely to receive its
    // needed resources
actualUsage = task.EndTask();
```

5.2. Client Task performing asynchronous operations

```
// Compute resourceEstimates based upon past usage.
// Determine deadline based upon program’s real-time requirements.
(task,   succeeded)   =   BeginTask(CurrentTask(),   resourceEstimates,
deadline);
// Causes scheduling inheritance to receiving thread
Send(Async_Request);
task.NotWorkingOn();

... then the client later does...

(msg, task) = Receive(); // Receives the reply from the async request.
Also does an implicit task.WorkingOn() call.
actualUsage = task.EndTask();
```

5.3. Simple server code

```
(msg, task) = Receive(); // Does an implicit task.WorkingOn()
// ... do work to service request ...
msg.SendReply(replyMsg); // Does an implicit task.NotWorkingOn()
```

5.4. Smart server code queuing and reordering requests

```
(msg, task) = Receive(); // Does an implicit task.WorkingOn()
...
workQueue.Queue(msg, task)
...
pick worker thread w to associate with the task
task.BlockedOn(w);
task.NotWorkingOn();
// Likely implied by the NotWorkingOn() call…
task.NotBlockedOn(CurrentThread());
```

5.5. Smart server code dequeueing and servicing requests

```
... (code running in worker thread W) ...
Choose work item on workQueue to service
(msg, task) = workQueue.Dequeue();
task.WorkingOn();
// ... do work to service request...
msg.SendReply(replyMsg);  //  Does  an  implicit  task.NotWorkingOn()  and
an implicit task.NotBlockedOn(CurrentThread())
```

6. Dependencies Upon Other Pieces of Singularity

6.1. Receive() Call Semantics

We are depending upon having the following variants of Receive() available to applications:

- Receive a message from any channel that this thread is bound to
- Receive a message from any of a set of channels specified in the call (each of which the thread must be bound to)

In both of these calls, the scheduler gets to determine which message to deliver if multiple messages are available.

From the Task object passed in via Receive() you can retrieve:

- resource estimates
- resource usage for the task so far
- the parent task
- quantities needed to compute the task laxity

7. Open Issues

7.1. Admission Control Algorithms/Admittance Test

Treating all the resources as if they were CPU is way too conservative but other possible algorithms may not be conservative enough. As a first cut, we might admit assuming that the threads won't block and if they block then they may lose portions of their task reservation, although this is unsatisfying. For admission control purposes, we could assume that resource usage on multiple resources occurs serially, rather than sometimes in parallel.

7.2. Semantics of Nested Tasks

Should task.WorkingOn() keep a stack of tasks being worked on by a thread and does task.NotWorkingOn() pop the stack or should task.WorkingOn() replace the thread's task without keeping any history? I'm assuming the stack model because it allows lexical scoping like the following to work without the inner scope (which may be in a library) needing to know anything about the enclosing task:

```
BeginTask(...)
        BeginTask(...)
        EndTask()
EndTask()
```

7.3. Interactions Between Borrowed Virtual Time (BVT) Scheduling and Laxity

We are planning to use a minimum-laxity-first scheduler as the primary scheduling algorithm for all resources since laxity values are meaningful across resources. We are planning to use a Borrowed Virtual Time (BVT) scheduler on the CPU second chance queue. It's an open question how to factor task laxity into the BVT-based scheduling decisions, and likewise, to factor the amount that a task is "borrowed" in the BVT calculation into the primarily laxity-based scheduling decisions.

A related issue is how to have all resource schedulers favor scheduling tasks where other resources have smaller laxity. For instance, the disk scheduler could favor requests with low CPU laxity.

7.4. How to Account for Garbage Collection Time

How do we find time to GC? Can we charge the task for the GC behavior that it causes so we can reserve time for it?

8. Possible Research Outcomes and Questions to Investigate

Building scheduling inheritance will allow us to measure its effects and costs. By turning it off we will be able to measure the amounts of scheduling inversions and quantify their effects.

We should be able to compare the effectiveness of our laxity/BVT scheduler versus a Rialto pre-computed schedule by supporting both kinds of schedulers.

We will want to quantify the garbage generation rate associated with our scheduling algorithms.

9. References

[Banga et al. 99] Gaurav Banga, Peter Druschel, and Jeffrey C. Mogul. Resource Containers: A New Facility for Resource Management in Server Systems. In *Proceedings of Third USENIX/ACM Symposium on Operating Systems Design and Implementation* (OSDI). New Orleans, LA, February 1999.

[Duda & Cheriton 99] Kenneth J. Duda and David R. Cheriton. Borrowed-virtual-time (BVT) scheduling: supporting latency-sensitive threads in a general-purpose scheduler. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles* (SOSP). Kiawah Island, SC, December 1999.

[Jones et al. 97] Michael B. Jones, Daniela Rosu, and Marcel-Catalin Rosu. CPU Reservations and Time Constraints: Efficient, Predictable Scheduling of Independent Activities. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles* (SOSP). Saint-Malo, France, October 1997.