Project no. IST-033576

XtreemOS

Integrated Project
BUILDING AND PROMOTING A LINUX-BASED OPERATING SYSTEM TO SUPPORT VIRTUAL ORGANIZATIONS FOR NEXT GENERATION GRIDS

Design and Implementation of a Customizable Scheduler
D2.2.6

Due date of deliverable: November 30^{th}, 2007
Actual submission date: November 30^{th}, 2007

Start date of project: June 1^{st} 2006

Type: Deliverable
WP number: WP2.2
Task number: T2.2.6

Responsible institution: XLAB
Editor & and editor's address: Marko Novak
XLAB d.o.o.
Teslova 30
1000 Ljubljana
Slovenia

Version 1.0 / Last edited by Marko Novak / November 30^{th}, 2007

Project co-funded by the European Commission within the Sixth Framework Programme		
Dissemination Level		
PU	Public	√
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	

Revision history:

Version	Date	Authors	Institution	Section affected, comments
0.1	28.10.07	Marko Novak	XLAB	First draft for internal review
0.2	29.10.07	Marko Novak	XLAB	Fixed layout of source code.
0.3	8.11.07	Marko Novak	XLAB	Integrated comments of internal reviewer (Toni Cortes, BSC).
0.4	16.11.07	Marko Novak	XLAB	Inserted data about internal reviewers and T2.2.6 task.
0.5	21.11.07	Marko Novak	XLAB	Integrated comments of internal reviewer (Carsten Franke, SAP).
1.0	30.11.07	Marko Novak	XLAB	Final version of the deliverable ready.

Reviewers:
Toni Cortes (BSC), Carsten Franke (SAP)

Tasks related to this deliverable:

Task No.	Task description	Partners involved°
T2.2.6	Design and implementation of a customizable scheduler	XLAB*, INRIA

°This task list may not be equivalent to the list of partners contributing as authors to the deliverable
*Task leader

Abstract

This document presents the work that was done on the implementation of customizable scheduler for LinuxSSI-XOS (shorter, LinuxSSI) operating system in the first implementation phase of XtreemOS project (from month 0 to month 18). Scheduler is a component of LinuxSSI which is in charge of placing processes to different cluster nodes. Its goals are, among others, balancing load across the cluster, optimization of resource utilization, etc. By migrating processes from one cluster node to another it takes care of transferring load from overloaded nodes to less busy ones.

During the first implementation phase, we have successfully implemented dynamical loading of probes and scheduling policies. We designed special framework which we named "Pluggable Probes and Scheduling Policies Framework" (PlugProPol). PlugProPol is an infrastructure which enables user to write his own probes and scheduling policies and add them to the system in runtime, without the need to restart the whole cluster. In this document, we describe main features and architecture of PlugProPol. We also introduce the three entities that form PlugProPol: probe, scheduling policy and filter.

The document also contains detailed instructions on how to install and configure LinuxSSI as well as PlugProPol framework and a PlugProPol user's guide, where all the PlugProPol commands are described along with their examples of usage. The user's guide is followed by the developer's guide in which we explain how to write probes, scheduling policies and filters. The developer's guide also contains a reference guide for the PlugProPol main functions.

Contents

Chapter 1

Introduction

1.1 Terminology

In this section, you can find the descriptions of the various terms, which are used throughout the deliverable. Understanding of the meaning of these terms is very important for the understanding of the whole deliverable.

Probe: probe is an entity for measuring different resource properties (e.g. CPU load, CPU speed, total memory, free memory). User can implement these probes as separate Linux kernel modules and inserts them dynamically into kernel (the insertion is possible only if the user has root permissions). By doing this, it extends set of resource properties that are being measured. Probes should be implemented in a way that each probe monitors resource properties of a single resource.

Scheduling policy: scheduling policy is an implementation of a job scheduling algorithm. In LinuxSSI, scheduling policy is in charge of selecting a proper node for a particular process. An example of scheduling policy is a load balancing policy. This policy takes resource properties from one or more probes as an input and when it senses local node is very heavily used, it is allowed to migrate certain number of processes to remote nodes.

End port: a part of the scheduling policy which takes care of acquiring resource measurement value from remote probe. Each end port of a scheduling policy can be connected only to one probe attribute. Additionally, the data type of end port must match the data type of the probe attribute in order to be able to connect them together.

Scheduler object: in PlugProPol terminology a scheduler object is a ConfigFS entity which contains scheduling policy (see the description above) and a process set. A process set is used for storing PID's of all the processes to which a particular scheduling policy applies to. So scheduler object is basically used for assigning a scheduling policies to particular processes.

Filter: filter is an intermediate entity which is used for caching and processing of probes' data. A particular filter can be connected to a probe or to another filter (i.e. filters can be chained). The filters are most frequently used for filtering probes'

measurement data on its way from probes towards scheduling policies based on various thresholds.

Process placement policy: process placement policy takes care of selecting a node on which a particular process will be placed. This can be used in various scenarios (e.g., for load balancing, etc.). The process placement policy is used only once for each process, at the process creation time (i.e. during the "fork" system call).

For a practical illustration of the terms above, see the Section 2.3. This section contains example entities of the different PlugProPol entities.

1.2 LinuxSSI

LinuxSSI is an adaptation of Linux operating system for computer clusters. Its main goal is to provide Single System Image of a whole computer cluster. According to [3], Single System Image (SSI) is a property of a system to hide the heterogeneous and distributed nature of the available resources and present them to users and applications as a single unified computing resource. In other words, an SSI cluster of 10 uniprocessor computers is seen by the user as a single 10-processor computer. The advantages of SSI clusters are ease of use (i.e., the whole cluster is used as a single machine), high availability (i.e., the system continues to operate even if some failures occurred), automatic load balancing and others.

LinuxSSI is based on Kerrighed project which is a community project led by a company called Kerlabs (`http://www.kerlabs.com`). Kerrighed's, as well as LinuxSSI's, main targeted features are:

- cluster-wide process management (cluster-wide PID's, fork, kill, etc.),
- support for cluster-wide shared memory,
- cluster file system,
- process checkpointing,
- process migration,
- mechanisms for high availability,
- customizable cluster-wide scheduler.

1.3 Customizable scheduler for LinuxSSI

In LinuxSSI, a scheduler is a component which selects a cluster node, to which a particular process will be placed. There are many types of schedulers, each of them having unique goal (some schedulers try to minimize average response time, others try to optimize resource utilization, etc.). In the first implementation phase of

XtreemOS project, we were dealing with load balancing schedulers. These schedulers take care of migrating processes from one cluster node to another and thus transferring load from overloaded nodes to less busy ones. This way, there are less idle nodes which means that computing power of the whole cluster is better utilized. As a consequence, a cluster with a global scheduler is able to process more tasks in the same mount of time as a cluster without it. The LinuxSSI scheduler also takes care of dispatching user-defined jobs to different nodes of the cluster and enables user to submit jobs, monitor their state and control their execution.

In XtreemOS Description of Work [4] and later in D2.2.1 deliverable [6] we promised to implement customizable scheduler for LinuxSSI which would enable user to tune it according to his needs. A user would be able to implement his own probes and scheduling algorithms, plug them to the scheduler and connect them in runtime. This way, we would improve the configurability of the old LinuxSSI scheduler that was inherited from Kerrighed.

Moreover, the whole scheduler should adapt itself to the load in the cluster. For example, when cluster was heavily loaded, the scheduler could have automatically prolonged probing period of all its probes in order to induce less overhead due to measurements. When the load was normal again, it would have set the probing period back to the original value. Furthermore, a complete scheduling algorithm could be switched, based on the load of the cluster. This way, we could have separate scheduling policies for times when cluster is heavily loaded, for normal load and for situations where load is very light [8, 9].

During the first implementation phase, we have successfully implemented dynamical loading of probes and scheduling policies. We designed special framework which we named “Pluggable Probes and Scheduling Policies Framework” (PlugProPol). This framework takes care of loading user-implemented resource measurement probes and scheduling policies into the Linux kernel and enabling/disabling them in runtime. The framework can also be utilized by the user to fine-tune the probes’ and policies’ parameters and thus adapt the scheduler to the current state of the cluster.

As a part of PlugProPol framework, we also implemented infrastructure for dynamic selection of process placement policies (see the definition of process placement policy in the next chapter). The old version of the scheduler provided only one hard-coded process placement policy, which made it non-adaptable to the changing state of the cluster. To improve this, we decided to implement the infrastructure that would enable users to choose proper policy in the runtime.

Our contribution is mostly related to the LinuxSSI-XOS foundation (i.e., cluster version of XtreemOS-F foundation) of the XtreemOS operating system. On the other hand, by implementing PlugProPol, we enabled XtreemOS-G grid scheduler to have a better control over the clusters since from now on, it can trigger loading/unloading of probes and scheduling policies.

The deliverable is organized as follows. The 2nd chapter describes main features and architecture of PlugProPol, a framework for loading probes and scheduling policies in runtime. In the 3rd chapter, we give detailed information on how

to install and configure LinuxSSI operating system and PlugProPol after that. In 4th chapter, we list all the commands that can be used with PlugProPol. We also explain its syntax and give its description and examples of use. In 5th chapter, we describe the PlugProPol's API and demonstrate its use by inserting source code of a simple probe and scheduling policy. Finally, in 6th chapter we conclude and state our future work.

Chapter 2

Pluggable Probes and Scheduling Policies Framework (PlugProPol)

Pluggable Probes and Scheduling Policies Framework (PlugProPol) is an infrastructure which enables user to write his own probes and scheduling policies and add them to the system in runtime (without the need to restart the whole cluster). If a user wants, for example, to monitor disk usage on his local machine, he only implements a proper probe and plugs it to PlugProPol in runtime. This makes the scheduling much more configurable since no reboot is needed.

All the probes and scheduling policies are implemented as Linux kernel modules, they run in kernel space and are able to access kernel data structures directly. No system calls are needed, which means such infrastructure induces less overhead than the one where probes and scheduling policies would be implemented in user space.

2.1 Main features

One of the main goals for our framework was to make it highly configurable despite the fact that it runs in kernel space. The reason we decided to put it in kernel space, was the fact that this approach has a couple of important advantages over execution in user space:

1. measuring resource properties from user space is more complicated than measuring from kernel space, sometimes even impossible. For example: in Linux, we have to do a lot of parsing of different proc files if we are trying to get a number of runnable processes from user space. On the other hand, the same task can be accomplished very easily from kernel space by invoking a few proper kernel functions,

2. there is much less overhead induced if we perform measuring in kernel space: monitoring probe can directly access resource properties in kernel space, rather than having to use system calls to transfer them to user space.

This saves us some CPU time. Since some resources are measured very frequently, the save is not insignificant.

Concerning configurability, we want the users to be able to plug measurement probes at any time and immediately start measuring resource properties. No reboot should be necessary. For example: if a user wanted to monitor disk usage on his local machine, he would only have to implement a proper probe and plug it to the framework at runtime. After that, he would be able to start collecting data about disk usage immediately. This approach enables user to easily change the group of locally measured resources and adapt it to his current needs, local system load, etc.

We also want to be able to connect scheduling algorithms to all the necessary probes in runtime. This way we are able to detach probes from the rest of scheduling infrastructure which introduces modularity to our implementation and thus makes it more robust. Furthermore, users are able to dynamically choose the probes that will provide data to their scheduling algorithms. Whenever a user wants to connect an end port (i.e. a part of scheduling policy that is used for acquiring a value of a single probe attribute) of a scheduling policy to some probe attribute, the PlugProPol framework itself first checks if their data types match. If a data type that a particular end port collects doesn't match the probe attribute's data type, they cannot be connected. This prevents a user to connect incompatible entities by mistake. Besides dynamically connecting scheduling policies to the probes we also want the user to be able to specify in what way he would like to receive the data from that probe. User can choose one of the two ways as specified in the Open Grid Forum (OGF) specifications [13]:

1. query-response: here, implementation of scheduling algorithm periodically "pulls" data from probe.

2. publish-subscribe: here, implementation of scheduling algorithm subscribes to probe's data and probe "pushes" data to port every time data is changed.

The framework also takes care of executing all its commands cluster-wide: whenever a particular command (e.g. command for loading a particular probe) is executed on some node in the LinuxSSI cluster, it immediately gets propagated to the rest of the nodes in the cluster. This relieves the user of having to manually load a particular probe on every node in the cluster. The framework itself assures that a particular command does not succeed unless it has executed successfully on all the nodes in the cluster.

The security issues are taken care of Linux itself since by default, only root user can load kernel modules. This prevents malicious users to load probes or scheduling policies that could crash the system. Furthermore, PlugProPol uses Linux module infrastructure to monitor dependencies among the kernel modules. As a consequence, a user cannot remove a probe if it is still used by some scheduling policy.

2.2 Design and implementation

When designing PlugProPol framework we decided that a user should be able to implement each probe, as well as scheduling policy, as a separate Linux kernel module. This would enable him to load and unload different probes and scheduling policies independently of each other which would increase modularity of the scheduler. Because probes and scheduling policies are kernel modules, they would be running in kernel space which would enable them to access resource properties with less overhead, as mentioned in previous section.

In order to achieve high configurability of the framework (i.e. a user should be able to load probes and scheduling policies from user space in runtime) despite the fact that it runs in kernel space, we decided to base it on "ConfigFS" pseudo file system [1], a virtual file system designed for configuring kernel modules from user space. It is very similar to "sysfs" pseudo file system, since both use directories as the way of representing objects and files as the way of representing object attributes. The fundamental difference between them is that ConfigFS objects can be controlled from user space by any user with sufficient permissions, whereas sysfs objects have to be controlled from kernel modules in kernel space. This makes ConfigFS more suitable for our framework. If you look at the PlugProPol source code, you will see that majority of code is related to ConfigFS: initialization of ConfigFS data structures, registration of ConfigFS objects, implementation of all the necessary callbacks, etc.

The ConfigFS file system enables our framework to react to different user commands: mkdir, rmdir, ln, etc. to load and unload probes, to link scheduling algorithms with probes, etc. ConfigFS has been a part of Linux kernel since version 2.6.17.13. In ConfigFS, probes and scheduling policies are represented as separate directories.

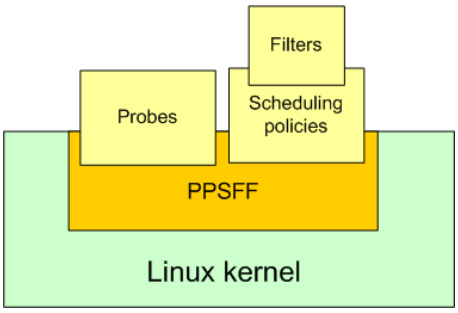

Figure 2.1: Architecture of PlugProPol framework.

The PlugProPol framework (figure 2.1) consists of three entities (see their descriptions in Section 1.1):

- probes
- scheduling policies

- filters

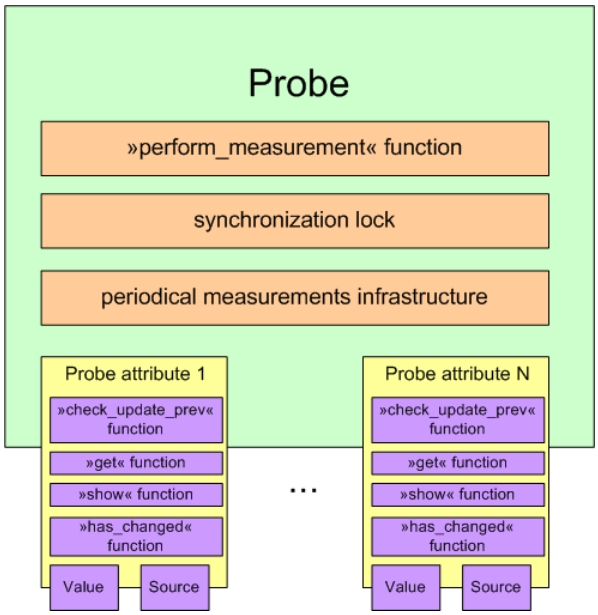

Figure 2.2: Architecture of probes.

Figure 2.2 shows architecture of probes. Every probe contains user-defined attributes (i.e. user adds each attribute programmatically at the implementation time by calling the "krg_probe_value_create" function of PlugProPol framework) which are used for storing resource measurements. These attributes are represented as subdirectories of the probe's main directory in order to enable symbolic link creation between a probe and a scheduling policy (in ConfigFS, a symbolic link can only be created if target is a directory). Each probe has to implements its own resource measurement function (i.e. the "perform_measurement" function), functions for retrieving attribute values (i.e. separate "get" function for each probe attribute), for showing attribute values (i.e. separate "show" function for each probe attribute) and for determining if a particular attribute has changed (i.e. separate "has_changed" function for each probe attribute). Furthermore, it has to allocate memory for storing the measurement data. When a probe registers with the PlugProPol framework, the framework itself takes care of reconfiguring it and performing resource measurements.

The purpose of scheduler object (figure 2.3, see definition in Section 1.1) in PlugProPol framework is basically to assign a particular scheduling policy only to a subset of processes. As opposed to probes and scheduling policies, a user can choose arbitrary name to create scheduler object. The most important part of scheduler object is scheduling policy.

PlugProPol scheduling policy (figure 2.4) is basically implementation of scheduling algorithm. It takes care of deciding when the process migration has to be in-

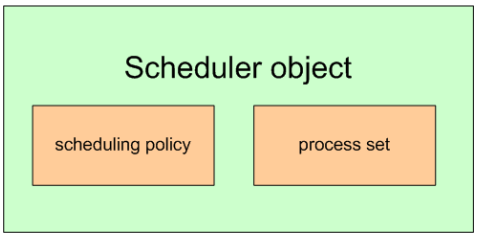

Figure 2.3: Architecture of scheduler object.

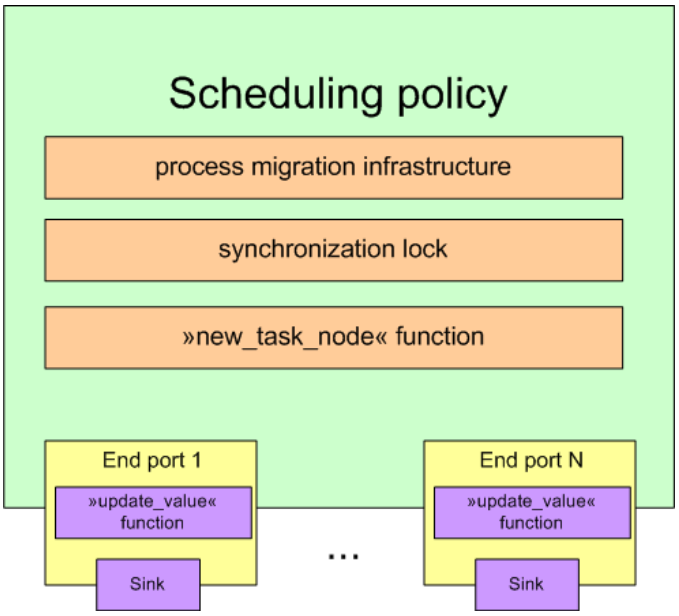

Figure 2.4: Architecture of scheduling policies.

voked. It also chooses a process to migrate and a node on which it will bi migrated. Additionally, scheduling policy also takes care of selecting a node for a newly created process.

When scheduling algorithms are connected to probes, they can collect their data via end ports using one of the two methods mentioned above: query-response and publish-subscribe. Since both methods don't need any user space functionality there is very little overhead when transferring data between probes and scheduling algorithm implementations. The framework takes care of the whole transferring procedure, each probe only has to implement "update_value" function for each end port, which takes care of updating that property's value. The "update_value" functions can also contain all the logic for checking if process migration should be initiated.

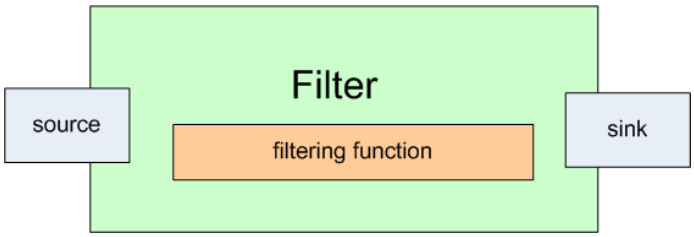

Figure 2.5: Architecture of filters.

The last entities to describe are the filters (figure 2.5). They are used for processing and caching the probes' measurement data and are therefore positioned between the probes and scheduling policies. Each filter has a source endpoint on which other entities (filters or scheduling policies) connect in order to be able to acquire data from the filter. It also has a sink endpoint through which it receives data from other entities (filters or probes). As already mentioned above, filters can also be chained (a particular filter's sink can be connected to the other sink's source) which means we can apply multiple filtering conditions to the measurement data of particular probe.

2.3 Example PlugProPol entities

In order to better illustrate the entities presented in the previous section, this section contains a sample use case of the PlugProPol framework. Let's say that a user wants to implement a scheduling algorithm which makes sure that cluster nodes' CPUs are as equally balanced as possible. We will call this algorithm "CPU load balancing algorithm". The PlugProPol entities that need to be implemented in order to enable this algorithm are shown in Figure 2.6.

First, a user has to implement a CPU probe. This probe is in charge of measuring the load of each CPU of a particular cluster node (note that a particular probe instance measures only CPU load on local cluster node). So if a user wants

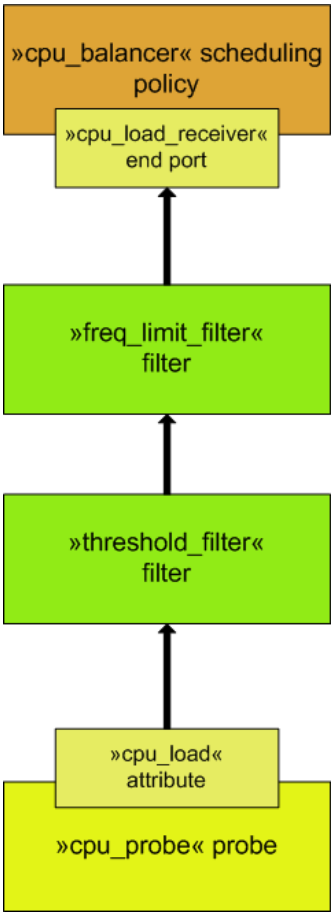

Figure 2.6: The PlugProPol entities of CPU load balancing algorithm.

to find out how loaded the CPUs of a particular node are, he only has to read the the “cpu_load” file in ConfigFS file system. This file is located in “cpu_probe” directory which represents CPU probe in ConfigFS file system.

After the CPU probe has been implemented, a user has to implement an actual load balancing algorithm. This algorithm is in charge of migrating processes from a local node to the remote node with the lowest CPU load. In PlugProPol framework, the CPU load balancing algorithms is implemented in a form of a “cpu_balancer” scheduling policy. The “cpu_balancer” policy defines a special “cpu_load_receiver” end port, which is used for connecting it with “cpu_load” attribute of the “cpu_probe” probe. This way, the CPU load balancing algorithm collects CPU load measurements. When the load is too high, the process migration procedure is initiated.

To prevent load balancing algorithm to be executed too frequently and thus induce large overhead, the PlugProPol framework enables user to implement special filters. These filters receive probe measurement data as their input and process it in various ways. In this example, we define a “threshold_filter” filter, which collects CPU load measurements from “cpu_probe” and passes them through only if their value is greater than a given threshold. This way, we make sure that the load balancing algorithm will be invoked only if the CPU load has risen over a given threshold.

The second filter in our example is “freq_limit_filter”. This filter makes sure that a specified time period has passed between subsequent migrations. This way, we want to prevent two nodes from passing a given process between each other (for example, node A detects high load and sends process P to node B, which is less loaded. By doing this, its load decreases substantially. At the same time, the CPU load on node B rises over a given threshold and load balancing policy decides to send P back to node A. This procedure can repeat many times.) and thus creating high overhead. The “freq_limit_filter” filter receives data from “threshold_filter” and passes it through only if a given time interval has passed since the last migration.

PlugProPol framework also enables us to assign the CPU load balancing algorithm to only a subset of the running processes. This can be done via PlugProPol scheduler object’s process set: we assign it a list of processes, which a particular policy should be in charge of. If we want a given scheduling policy to handle all the running processes on a given node, we have to set the “handle_all” attribute of its process set to 1.

Chapter 3

Installation and Configuration

PlugProPol is distributed as a part of LinuxSSI operating system so in order to enable it, we have to install LinuxSSI first. The LinuxSSI installation procedure is described in detail in XtreemOS D2.2.7 deliverable [7]. At the time of writing, PlugProPol was still under development so certain parts may change in the future.

As mentioned above, the installation of scheduler-enabled LinuxSSI is exactly the same as in XtreemOS D2.2.7 deliverable [7]. However, a user has to do some additional things, mostly related to ConfigFS file system, which is used for controlling LinuxSSI scheduler.

First of all, a user has to create a "config" directory (i.e., a target directory to which ConfigFS is mounted) in the root directory of a diskless nodes' system image. In XtreemOS D2.2.7 deliverable [7], this root directory is located in "/NFSROOT/LinuxSSI" directory:

```
# chroot /NFSROOT/LinuxSSI
# mkdir /config
```

The next step is to check if the following features are enabled in LinuxSSI kernel configuration, after we have finished it (if you do not know how to configure a Linux kernel, consult the proper documentation [12, 15]).

The easiest way to check this is to open the kernel configuration file from the main directory of LinuxSSI source code (for example, by invoking the **"vi <LinuxSSI_src>/kernel/.config**" command) and check if the corresponding options are set to "y":

- the ConfigFS file system (CONFIG_CONFIGFS_FS=y)
- automatic kernel module loading (CONFIG_KMOD=y), this feature is needed to enable PlugProPol framework to load kernel modules

The final step is to edit the **/etc/fstab** file (note that this is actually the "/NFSROOT/LinuxSSI/etc/fstab" file, but since we changed root node to "/NFSROOT/LinuxSSI", we must disregard that part of the path). This file has to contain the mount parameters for "configfs" file system, as shown below:

```
# the line below mounts ConfigFS file system which
# is needed by the PlugProPol framework
none /config configfs defaults 0 0
```

Chapter 4

User's guide

Since PlugProPol framework is based on ConfigFS file system, all its operations such as probes/scheduling policies loading and unloading, linking can be performed from user space using standard Linux commands such as mkdir, rm, ln. This chapter contains a list of Linux commands which can be used with PlugProPol framework, along with their descriptions and examples of usage. To guarantee security and stability of the system, these commands can be executed only by a user with root permissions. As already mentioned in Section 2.1, all the commands are executed cluster-wide.

4.0.1 mkdir

Syntax: *mkdir <module_name | process_PID>*

Arguments:

- *<module_name | process_PID>*: name of the probe/scheduling policy module to load. This name must be equal to the name of the module file (the one with the ".ko" extension) which implements given probe/scheduling policy. When working with process sets, a PID of a process which we want to add to process set.

Description: by invoking this command a user requests PlugProPol framework to load a given probe or scheduling policy. In order to initiate probe loading, user has to invoke this command in */config/probes* directory. Similarly, he has to invoke the command in */config/schedulers* directory if he wants to load scheduling policy. Alternative, this command can also be used to add PID's to process sets. Every time the "make" command is used in the ConfigFS file system, a user-defined callback function is invoked. In PlugProPol framework, this callback takes care of all the activities necessary for loading.

Probes and scheduling policies should be implemented as Linux kernel modules [11] and should register themselves with PlugProPol framework in their ini-

tialization function. For an example of how to write a proper probe or scheduling policy see "mem_probe.c" and "echo_policy.c" in the Appendix. Once the probe/scheduling policy is loaded with PlugProPol the framework itself takes care of performing measurements (for the probes) and collecting them (for scheduling policies).

In order for a user to be able to load a particular probe or scheduling policy module with PlugProPol, he must first register it. This is done by inserting full path to the module file (the one with the ".ko" extension) to the **/lib/modules/<kerrighed_kernel_version>/modules.dep** file, followed by a colon and full path to "kerrighed.ko" module file. Below you can see example entries for "mem_probe" and "echo_policy" (note that both path to the probe/scheduling module file and path to the "kerrighed.ko" file must be written in the same line for each entry):

```
/lib/modules/2.6.20-krgLinuxSSI/extra/mem\_probe.ko:_
  /lib/modules/2.6.20-krgLinuxSSI/extra/kerrighed.ko
/lib/modules/2.6.20-krgLinuxSSI/extra/echo\_policy.ko:_
  /lib/modules/2.6.20-krgLinuxSSI/extra/kerrighed.ko
```

Example:

```
♯ load mem_probe probe
mkdir /config/krg_scheduler/probes/mem_probe

♯ load echo_policy scheduling policy
mkdir /config/krg_scheduler/schedulers/test-scheduler
mkdir /config/krg_scheduler/schedulers/test-scheduler/echo_policy
♯ adding PID to process set
mkdir /config/krg_scheduler/schedulers/test-scheduler/process_set/33465
```

4.0.2 rmdir

Syntax:

- *rmdir <module_name>*
- *rm -rf <module_name>*

Arguments:

- *<module_name>*: name of the probe/scheduling policy to unload. This name must be equal to the name of the module file (the one with the ".ko" extension) which implements given probe/scheduling policy. When working with process sets, a PID of a process which we want to remove from process set.

Description: this command is used for unloading probe or scheduling policy module that was loaded with mkdir command. Similarly to mkdir, the rmdir command has to be invoked either in */config/probes* directory for probes or in */config/schedulers* directory for scheduling policies. Alternative, this command can also be used to remove PID's to process sets. When this command is invoked with a probe, it will successfully remove it only if there are no scheduling policies that are using it. Like with "mkdir", the "rmdir" command triggers its own callback function if invoked in ConfigFS file system. In PlugProPol framework, this function takes care of all the thing related to unloading.

Example:

```
♯ unload mem_probe probe
rmdir /config/krg_scheduler/probes/mem_probe

♯ unload echo_policy scheduling policy
rmdir /config/krg_scheduler/schedulers/test-scheduler/echo_policy
♯ removing PID from process set
rmdir /config/krg_scheduler/schedulers/test-scheduler/process_set/33465
```

4.0.3 echo

Syntax: *echo* <*value*> > <*attribute_name*>

Arguments:

- <*value*>: the value which we want to assign to a given attribute. It can be given either in textual or in numerical form. If the value contains spaces it has to be surrounded by double quotes (").
- <*attribute_name*>: the name of the attribute to which we want to assign the value. The name must be given as an absolute path to the file which represents the attribute.

Description: this command assigns a value to the attribute specified by the attribute_name filename. Each attribute is represented by a separated file which are located in of the subdirectories in "/config/probes" or "/config/schedulers" directory. The command enables user to set probe and scheduling policy parameters in runtime and thus dynamically tune the scheduler.

Example:

```
♯ set probing period of mem_probe probe to 1 second.
echo 1000 > /config/krg_scheduler/probes/mem_probe/probe_period
```

4.0.4 cat

Syntax: *cat* <*attribute_name*>

Arguments:

- <*attribute_name*>: the name of the attribute whose value we want to retrieve. The name must be given as an absolute path to the file which represents the attribute.

Description: this command retrieves a value of the attribute specified by the attribute_name filename. Each attribute is represented by a separated file which are located in of the subdirectories in "/config/probes" or "/config/schedulers" directory. The command enables user to get probe and scheduling policy parameters and is mostly used for reading properties of resources monitored by the probes.

Example:

```
♯ get total memory from mem_probe probe
cat /config/krg_scheduler/probes/mem_probe/ram_total/value

♯ get total memory from echo_policy scheduling policy
cat /config/krg_scheduler/schedulers/test-scheduler/echo_policy/_
port_mem_total/value
```

4.0.5 ln -s

source and sink must be directories (cannot link attributes).
Syntax: *ln -s* <*data_source*> <*data_sink*>

Arguments:

- <*data_source*>: source endpoint of the link which will provide us the data. This must be a directory (i.e. we cannot make symbolic links to attributes).
- <*data_sink*>: sink endpoint of the link which will consume the data. Like data_source, this must also be a directory (i.e. we cannot make symbolic links to attributes).

Description: by invoking the "ls -s" command, a user can connect one PlugProPol entity to another. By doing this he enables the "sink" entity to collect data from the "source" entity.

Only the connections that are defined below are possible:

- scheduling policy to probe,

- filter to probe,
- filter to another filter,
- scheduling policy to filter.

As soon as particular data source and sink are connected, PlugProPol framework takes care of data transfer between them.

Example:

```
♯ link port_mem_total endpoint of echo_policy scheduling policy
♯ to ram_total endpoint of mem_probe probe
ln -s /config/krg_scheduler/probes/mem_probe/ram_total _
/config/krg_scheduler/schedulers/test_scheduler/echo_policy/port_mem_total

♯ link port_mem_free endpoint of echo_policy scheduling policy
♯ to ram_free endpoint of mem_probe probe
ln -s /config/krg_scheduler/probes/mem_probe/ram_free _
/config/krg_scheduler/schedulers/test_scheduler/echo_policy/port_mem_free
```

To further demonstrate the use of PlugProPol commands, we have added contents of the "startup_mosix_scheduler.sh" Bash script in appendix. This script shows how to enable various probes, filters and special "Mosix Load Balancer" scheduling policy that user can enable in LinuxSSI. "Mosix Load Balancer" is an implementation of the same algorithm for load balancing that is present in Mosix operating system [2].

Chapter 5

Developer's guide

In this chapter, we describe API of the PlugProPol framework. Each entity (probe, scheduling policy, filter) has its own set of function for registering it with PlugProPol. An example source code for a probe, scheduling policy and filter can be found in the appendix.

5.1 Probes

When implementing a probe, a user has to first reserve memory for storing measurement data and define all the necessary functions: a measurement function, a callback function for getting measurement data, a callback function for showing measurement data and a callback function for determining if measurement variables have changed (see listing 5.1).

Listing 5.1: Macros for defining probe attributes and their callback functions.

```
/**
 * Convenience macro for defining a typed "get" callback function
 * with arguments. This callback is used by the filter/scheduling
 * policy to retrieve values from the probe.
 *
 * @param name          name of the probe attribute to which this callback
 *                      will be assigned.
 * @param type          type of the data a callback returns (eg. int).
 * @param ptr           name of the type *arg of the method
 * @param nr            name of the array length parameter of the method
 * @param in_type       type of the parameters of the method (eg. int)
 * @param in_ptr        name of the in_type *arg of the method
 * @param in_nr name of the parameters array length arg of the
 *                                                  method
 */
#define DEFINE_PROBE_VALUE_GET_WITH_INPUT(name, type, ptr, nr, in_type,
        in_ptr, in_nr)

/**
 * Convenience macro for defining a typed "get" callback with no
 * arguments. This callback is used by the filter/scheduling
 * policy to retrieve values from the probe.
 *
```

```
 * @param name          name of the probe attribute to which this callback
 *                      will be assigned.
 * @param type          type of the data a callback returns (eg. int)
 * @param ptr           name of the type *arg of the method
 * @param nr            name of the array length parameter of the method
 */
#define DEFINE_PROBE_VALUE_GET(name, type, ptr, nr)

/**
 * Convenience macro for defining a callback function which checks if
 * value of particular probe attribute has changed. This callback is
 * used by the publish-subscribe infrastructure to determine when a
 * "value changed" event needs to be triggered.
 *
 * @param name          name of the probe attribute to which this callback
 *                      will be assigned.
 */
#define DEFINE_PROBE_VALUE_HAS_CHANGED(name)

/**
 * Convenience macro which states that a particular probe attribute
 * doesn't have any callback function for checking if its value has
 * changed.
 *
 * @param name          name of the probe attribute to which this callback
 *                      will be assigned.
 */
#define DEFINE_PROBE_VALUE_HAS_CHANGED_NULL(name)

/**
 * Convenience macro for defining a "show" callback function.
 * This callback is used for displaying the value of a probe attribute
 * with the "cat" command.
 *
 * @param name          name of the probe attribute to which this callback
 *                      will be assigned.
 * @param page          name of the buffer arg of the method
 */
#define DEFINE_PROBE_VALUE_SHOW(name, page)

/**
 * Convenience macro which states that a particular probe attribute
 * doesn't have any callback function for showing its value.
 *
 * @param name          name of the probe attribute to which this callback
 *                      will be assigned.
 */
#define DEFINE_PROBE_VALUE_SHOW_NULL(name)

/**
 * Mandatory macro for defining a probe attribute whose "get"
 * callback function requires input arguments. Probe attributes are
 * used for exporting measurement values to the filters and
 * scheduling policies.
 *
 * The "get", "has_changed", and "show" callback functions must be
 * defined with DEFINE_PROBE_VALUE_* macros.
 *
 * @param name          name of the probe attribute
 * @param attrs         not used yet
 * @param value_type    type of the probe attribute (eg. unsigned int)
 * @param get_param_type        type of the arguments for the "get"
```

```
 *                                                              callback
 */
#define PROBE_VALUE_TYPE_WITH_INPUT(name, attrs, value_type, get_param_type)

/**
 * Mandatory macro to defining a probe attribute whose "get" callback
 * function doesn't accept arguments.
 *
 * The "get", "has_changed", and "show" callback functions must be
 * defined with DEFINE_PROBE_VALUE_* macros. Probe attributes are
 * used for exporting measurement values to the filters and
 * scheduling policies.
 *
 * @param name          name of the probe attribute
 * @param attrs         not used yet
 * @param value_type    type of the probe attribute (eg. unsigned int)
 */
#define PROBE_VALUE_TYPE(name, attrs, value_type)

/**
 * Mandatory macro to define a probe_type. Probe type contains a
 * callback function for performing measurement of particular
 * resource along with probe attributes which expose measurement
 * values to filters/scheduling policies.
 *
 * @param name          name of the variable containing the probe type.
 * @param attrs currently unused
 * @param _perform_measurement  probe measurement function. This
 *              function takes care of periodic measurings and subscribers
 *              refreshment. This function can be NULL.
 */
#define PROBE_TYPE(name, attrs, _perform_measurement)
```

After the probe type, all the probe attributes and their callback functions have been defined, a developer can start writing kernel module initialization function (a.k.a. the “init_module” function). In it he first instantiates all the probe attributes by invoking “krg_probe_value_create” function for each of them. After that, he creates probe (by invoking the “krg_probe_create” function), performs first measuring to initialize measurement data and registers probe with PlugProPol framework (by invoking the “krg_probe_register” function). For the definitions of these function, see listing 5.2.

Listing 5.2: Functions for creating and registering probes.

```
/**
 * This function allocates memory and initializes a probe value.
 *
 * @param type          Type describing the probe value, defined with
 *                      PROBE_VALUE_TYPE
 * @param name          Name of the value's subdirectory in the probe's
 *                      directory. Must be unique for a given a probe.
 *
 * @return              pointer to the created probe_value, or NULL if
 *                                  error occured
 */
struct probe_value * krg_probe_value_create(
        struct probe_value_type *type, const char *name);

/**
```

```
 * This function frees all the memory taken by the probe value.
 *
 * @param value          pointer to a probe value which memory we
 *                                                    want to free.
 */
void krg_probe_value_free(struct probe_value *value);

/**
 * Function that a probe value should call when the value changes
 * and the probe does not have a perform_measurement() method.
 * Does nothing if the probe provides a perform_measurement() method.
 *
 * @param value          Value having been updated
 */
void krg_probe_value_notify_update(struct probe_value *value);

/**
 * This function is used for registering probe. This function has to
 * be called at the end of "init_module" function for each probe's
 * module.
 *
 * @param probe          pointer to the probe we wish to register.
 *
 * @return               0, if probe was successfully registered.
 *                       -EEXIST, if probe with same name is already
 *                                registered.
 */
int krg_probe_register(struct probe *probe);

/**
 * This function is used for removing probe registration. This
 * function can only be called at module unloading (from the
 * "cleanup_module" function).
 *
 * @param probe          pointer to the probe we wish to unregister.
 */
void krg_probe_unregister(struct probe *probe);
```

Besides module initialization function, a developer also has to write a module cleanup function (a.k.a. “cleanup_module” function) in which it unregisters a probe and frees all the memory taken by it and its attributes.

Source code of an example probe can be found in the appendix (see the page 44).

5.2 Scheduling policies

As explained earlier in the document, PlugProPol scheduling policies are in charge of invoking process migration. They select a process to be migrated and a destination node to which that process will be transferred.

The implementation is very similar to the one of the probes. A developer first defines all the scheduling policy end ports ((using “SCHEDULER_END_PORT_TYPE” function)) along with the callbacks for updating their value. End ports are used for connecting scheduling policies with probe/filter attributes. From these probes/filters, the scheduling probes later collect the measurement data. After all the end

ports have been implemented the developer also defines a process placement function. This function is in charge of selecting the cluster node on which a process will be created at its "fork" time (see listing 5.3).

Listing 5.3: Macros and data structures for defining scheduling policy attributes, operation and end ports.

```
/**
 * Mandatory macro for defining an end port type when
 * "scheduler_end_port_get_value" doesn't require any arguments.
 *
 * @param name          variable name of the port type
 * @param update_value  callback to notify source's value updates
 * @param value_type    type of values which this port can retrieve
 *                      (eg. unsigned int)
 */
#define SCHEDULER_END_PORT_TYPE(name, update_value, value_type)

/**
 * Mandatory macro for defining an end port type when
 * "scheduler_end_port_get_value" takes input arguments.
 *
 * @param name          variable name of the port type
 * @param update_value  callback to notify source's value updates
 * @param value_type    type of values which this port can retrieve
 *                      (eg. unsigned int)
 * @param get_param_type        type of input arguments for
                                "scheduler_end_port_get_value" function
 */
#define SCHEDULER_END_PORT_TYPE_WITH_INPUT(name, update_value,
        value_type, get_param_type)

/**
 * This struct is used for representing scheduling policies'
 * attributes. It contains attribute-specific functions for reading
 * attribute and storing value.
 */
struct sched_policy_attribute {
        struct configfs_attribute attr;

        /** function for reading attribute's value. */
        ssize_t (*show)(struct sched_policy *, char *);
        /** function for storing attribute's value. */
        ssize_t (*store)(struct sched_policy *, const char *, size_t);
};

/**
 * Struct which contains each policy's operations.
 */
struct sched_policy_operations {
        /* Sched policy constructor. Creates new instance of scheduling
         * policy. */
        struct sched_policy * (*new)(const char *name);
        /* Sched policy destructor. Removes an instance of scheduling
         * policy. */
        void (*destroy)(struct sched_policy *policy);
        /* Process placement function. Called when a task attached to
         * this policy creates a new task */
        kerrighed_node_t (*new_task_node)(struct sched_policy *policy,
                                          struct task_struct *parent);
```

```
};

/**
 * Mandatory macro for defining a scheduling policy type. Can be
 * used through the SCHED_POLICY_TYPE macro.
 *
 * @param owner          module defining the sched_policy type
 * @param _name          unique name for the sched_policy type
 * @param _ops           sched_policy_operations for this type
 * @param _attrs         NULL-terminated array of custom
 *                       sched_policy_attribute, or NULL
 */
#define SCHED_POLICY_TYPE_INIT(owner, _name, _ops, _attrs)

/**
 * Convenience macro for defining a scheduling policy type.
 *
 * @param var            name of the variable containing the type
 * @param name           unique name of the sched_policy type
 * @param ops            sched_policy_operations for this policy type
 * @param attrs          NULL-terminated array of custom
 *                       sched_policy_attribute, or NULL
 */
#define SCHED_POLICY_TYPE(var, name, ops, attrs)
```

Besides the process placement function, the developer also has to implement the functions for creating and destroying instances of particular scheduling policies. A particular scheduling policy type can have multiple instances, each belonging to a different scheduler object (e.g. we can have a separate instances of CPU load balancing policy for I/O-bound processes and for CPU-bound processes). All instances of a given scheduling policy share the same scheduling algorithm, however they can have different parameters. The functions for scheduling policy creation/deletion reserve and free memory for each instance of a given scheduling policy (see listing 5.4 for a definition of an end port creation and removal functions).

Listing 5.4: Definition of an end port creation and removal functions

```
/**
 * This function initializes a new scheduling policy. Must be
 * called by sched_policy constructors.
 *
 * @param policy         pointer to the sched_policy to init.
 * @param name           name of the scheduling policy. This name
 *                       must be the one provided as argument to the
 *                       constructor.
 * @param type           type of the sched_policy
 * @param def_groups     NULL-terminated array of subdirs of the
 *                       sched_policy directory, or NULL
 *
 * @return               0 if successul,
 *                       -ENODEV is module is unloading (should not happen!),
 *                       -ENOMEM if not sufficient memory could be allocated.
 */
int krg_sched_policy_init(struct sched_policy *policy, const char *name,
        struct sched_policy_type *type, struct config_group *def_groups[]);

/**
```

```
 * This function frees all the memory taken by a scheduling policy.
 * Must be called by the sched_policy destructor.
 *
 * @param policy        pointer to sched_policy whose memory we want to
 *                      free.
 */
void krg_sched_policy_cleanup(struct sched_policy *policy);

/**
 * Create and initialize a new end port that can be included as a
 *      subdir (see scheduler_end_port_config_group).
 * The end port can be connected to a filter (using mkdir in its
 * subdir) or directly to a probe value (making a symlink from an
 * entry to its subdir).
 *
 * @param name          name of the new end port
 * @param update_value  callback to notify an update of the end
 *                      port's source
 * @param private       any value that the user wishes. It will be passed
 *                      unchanged to the update_value callback.
 *
 * @return              pointer to the new end port, or
 *                      NULL if error
 */
struct scheduler_end_port *
scheduler_end_port_create(const char *name,
                          struct scheduler_end_port_type *type,
                          void *private);
/**
 * Free an end port.
 *
 * @param port          pointer to end port to free
 */
void scheduler_end_port_free(struct scheduler_end_port *port);
```

After all the necessary functions have been defined, a developer implements a scheduling policy initialization and cleanup functions. The initialization function takes care of initializing all the end ports and registering scheduling policy with PlugProPol framework (when a particular scheduling policy is registered with PlugProPol framework, the administrator can assign it to different scheduler objects. This way, he puts it in charge of all the processes that belong to that scheduling object). The cleanup function is used for unregistering scheduling policy and for releasing all the memory taken by it (see listing 5.5).

Listing 5.5: Functions for initializing end port types, registering and unregistering scheduling policies.

```
/**
 * Complete initialization of an end port type.
 * Must be called before any port creation, typically at module init
 *
 * @param type          port type to init
 *
 * @return              0 if successful, or
 *                      negative error code
 */
int scheduler_end_port_type_init(
        struct scheduler_end_port_type *type);
```

```
/**
 * Free all resources allocated for a port type at initiliazation.
 * Must be called after all port destructions, typically at module exit
 *
 * @param type          port type to cleanup
 */
void scheduler_end_port_type_cleanup(
        struct scheduler_end_port_type *type);

/**
 * This function is used for registering newly added scheduling
 * policy types. Once a type is registered, new scheduling policies
 * of this type can be created when user does mkdir with the type
 * name.
 *
 * @param type          pointer to the scheduling policy type to register.
 *
 * @return              0 if successful,
 *                      -EEXIST if scheduling policy type with the same
 *                      name is already registered.
 */
int krg_sched_policy_type_register(struct sched_policy_type *type);

/**
 * This function is used for removing scheduling policy
 * registrations. This function can only be called at module
 * unloading.
 *
 * @param type          pointer to the scheduling policy type to
 *                      unregister.
 */
void krg_sched_policy_type_unregister(struct sched_policy_type *type);
```

Source code of an example scheduling policy can be found in the appendix (see the page 46).

5.3 Filters

The process of implementing filters is very similar to the one of probes as well as scheduling policies. First, a developer has to define filter attributes and their “get” (), “show” and “update_value” callback functions:

- get: this callback function is used by a scheduling policy/another filter to retrieve attribute value from a filter,
- show: this callback function is used for displaying the filter attribute value via the “cat” command,
- update_value: this callback function is used by a probe/another filter to push data to a filter.

The filter attributes are used for connecting filters with probes/scheduling policies/other filters. This way, a given filter can acquire measurements data from probes/other filters and forward it to scheduling policies/other filters.

After the filter attribute has been defined, functions for creating and destroying instances of a given filter type. As with scheduling policies, a given filter can have multiple instances, each belonging to a different scheduler object. Instances oaf a given filter share the same implementation, they only have different user-defined parameters. This way, we can use an instance of "cpu_thresh" filter (i.e., the filter that filters CPU load values if they are smaller than a given threshold) with CPU threshold set to 0.7 for one group of processes and an instance with CPU threshold set to 0.9 for the other group.

Next, a developer combines the filter attributes and filter instance constructors/destructors into a filter type by invoking the "SCHEDULER_FILTER_TYPE" function. A list of macros for defining filters, their attributes and callback functions can be found in listing 5.6.

Listing 5.6: Convenience macros and data structures for defining filters, their attributes and callback functions.

```
/* Structure representing a filter attribute in a filter directory */
struct scheduler_filter_attribute {
        /** configfs super-class */
        struct configfs_attribute config_attr;
        /** callback to show the attribute */
        ssize_t (*show)(struct scheduler_filter *, char *);
        /** callback to modify the attribute from userspace */
        ssize_t (*store)(struct scheduler_filter *, const char *, size_t);
};

/**
 * Convenience macro for defining a typed "get" callback function which
 * doesn't accept any arguments. This callback is used by the
 * scheduling policy/other filter to retrieve values from the filter.
 *
 * @param name          name of the filter type
 * @param filter        name of the struct filter *arg of the method
 * @param type          type of the values output by the filter (eg. int)
 * @param ptr           name of the type *arg of the method
 * @param nr            name of the array length arg of the method
 */
#define DEFINE_SCHEDULER_FILTER_GET_VALUE(name, filter, type, ptr, nr)

/**
 * Convenience macro for defining a typed "get" callback function which
 * requires input arguments. This callback is used by the
 * scheduling policy/other filter to retrieve values from the filter.
 *
 * @param name          name of the filter type
 * @param filter        name of the struct filter *arg of the method
 * @param type          type of the values output by the filter (eg. int)
 * @param ptr           name of the type *arg of the method
 * @param nr            name of the array length parameter of the method
 * @param in_type       type of the parameters of the method (eg. int)
 * @param in_ptr        name of the in_type *arg of the method
 * @param in_nr name of the parameters array length arg of the method
 */
#define DEFINE_SCHEDULER_FILTER_GET_VALUE_WITH_INPUT(name, filter,
        type, ptr, nr, in_type, in_ptr, in_nr)

/**
```

```
 * Convenience macro for defining a typed "show" callback function.
 * This callback is used for displaying the value of a filter attribute
 * with the "cat" command.
 *
 * @param name          name of the filter type
 * @param filter        name of the struct filter *arg of the method
 * @param page          name of the buffer arg of the method
 */
#define DEFINE_SCHEDULER_FILTER_SHOW_VALUE(name, filter, page)

/**
 * Convenience macro which developer uses when a filter doesn't
 * implement any "show" callback function.
 *
 * @param name          name of the filter type
 */
#define DEFINE_SCHEDULER_FILTER_SHOW_VALUE_NULL(name)

/**
 * Convenience macro for defining a typed "update_value" callback
 * function. this callback function is used by a probe/another
 * filter to push data to a filter.
 *
 * @param name          name of the filter type
 * @param filter        name of the struct filter *arg of the method
 */
#define DEFINE_SCHEDULER_FILTER_UPDATE_VALUE(name, filter)

/**
 * Convenience macro which developer uses when a filter doesn't
 * implement any "update_value" callback function.
 *
 * @param name          name of the filter type
 */
#define DEFINE_SCHEDULER_FILTER_UPDATE_VALUE_NULL(name)


/**
 * Convenience macro for defining a typed "remote_get_value"
 * callback function which requires input arguments. The
 * "remote_get_value" callback is used for retrieving value
 * from remote probe/filter.
 *
 * @param name          name of the filter type
 * @param filter        name of the struct filter *arg of the method
 * @param node          name of the node arg of the method
 * @param type          type of the values output by the filter (eg. int)
 * @param ptr           name of the type *arg of the method
 * @param nr            name of the array length parameter of the method
 * @param in_type       type of the parameters of the method (eg. int)
 * @param in_ptr        name of the in_type *arg of the method
 * @param in_nr         name of the parameters array length arg of the
 *                      method
 */
#define DEFINE_SCHEDULER_FILTER_GET_REMOTE_VALUE(name, filter, node, type,
        ptr, nr, in_type, in_ptr, in_nr)

/**
 * Convenience macro which developer uses when a filter doesn't implement
 * any "remote_get_value" callback function.
 *
 * @param name          name of the filter type
```

```
 */
#define DEFINE_SCHEDULER_FILTER_GET_REMOTE_VALUE_NULL(name)

/**
 * Convenience macro to define a trivial get_value method.
 * The method is not typed.
 *
 * @param name          name of the filter type
 */
#define DEFINE_SCHEDULER_FILTER_GET_VALUE_SIMPLE(name)

/**
 * Convenience macro to define a trivial show_value method.
 *
 * @param name          name of the filter type
 */
#define DEFINE_SCHEDULER_FILTER_SHOW_VALUE_SIMPLE(name)

/**
 * Convenience macro to define a trivial update_value method.
 *
 * @param name          name of the filter type
 */
#define DEFINE_SCHEDULER_FILTER_UPDATE_VALUE_SIMPLE(name)

/**
 * Convenience macro to define a trivial get_remote_value method.
 *
 * @param name          name of the filter type
 */
#define DEFINE_SCHEDULER_FILTER_GET_REMOTE_VALUE_SIMPLE(name)

/**
 * Convenience macro for defining a filter attribute. The filter
 * attributes are used for connecting filters with probes/scheduling
 * policies/other filters.
 *
 * @param var_name      name of the scheduler_filter_attribute variable
 * @param name          name of the attribute entry in the filter directory
 * @param mode          access mode of the attribute entry
 * @param show          show callback of the attribute
 * @param store         store callback of the attribute
 */
#define SCHEDULER_FILTER_ATTRIBUTE(var_name, name, mode, show, store)

/**
 * Convenience macro for defining a filter type whose "get" callback
 * function accepts no arguments. The filter type combines the filter
 * attributes and filter instance constructors/destructors.
 *
 * The "get", "update_value", and "show" callback functions must be
 * defined using DEFINE_SCHEDULER_FILTER_* macros.
 *
 * @param name          unique name among schuduler_port_type names. This
 *                      name is used for both the variable and the
 *                      user-visible type name.
 * @param new           constructor for this filter type
 * @param destroy       destructor for this filter type
 * @param src_value_type
 *                      type of filter's values (eg. unsigned int)
 * @param snk_value_type
 *                      type of lower source's values (eg. unsigned int)
```

```
 * @param attrs          NULL-terminated array of custom
 *                       scheduler_filter_attributes, or NULL
 */
#define SCHEDULER_FILTER_TYPE(name, new, destroy, src_value_type,
        snk_value_type, attrs)

/**
 * Convenience macro for defining a filter type whose "get" callback
 * function requires input arguments. The filter type combines the
 * filter attributes and filter instance constructors/destructors.
 *
 * The "get", "update_value", and "show" callback functions must be
 * defined using DEFINE_SCHEDULER_FILTER_* macros.
 *
 * @param name           unique name among schuduler_port_type names. This
 *                       name is used for both the variable and the user-visible
 *                       type name.
 * @param new            constructor for this filter type
 * @param destroy        destructor for this filter type
 * @param src_value_type         type of filter's values (eg. unsigned int)
 * @param src_get_param_type     type of the parameters for the get_value
 *                               method
 * @param snk_value_type         type of lower source's values (eg. unsigned
 *                               int)
 * @param snk_get_param_type     type of the parameters used in
 *                               filter's get_value calls
 * @param attrs          NULL-terminated array of custom
 *                       scheduler_filter_attributes, or NULL
 */
#define SCHEDULER_FILTER_TYPE_WITH_INPUT(name, new, destroy,
\
        src_value_type, src_get_param_type, snk_value_type,
        snk_get_param_type, attrs)

/**
 * Convenience macro to define a filter type using whose "get"
 * callback accepts arguments only when querying lower source.
 * The filter type combines the filter attributes and filter
 * instance constructors/destructors.
 *
 * The "get", "update_value", and "show" callback functions must be
 * defined using DEFINE_SCHEDULER_FILTER_* macros.
 *
 * @param name           unique name among schuduler_port_type names. This
 *                       name is used for both the variable and the user-visible
 *                       type name.
 * @param new            constructor for this filter type
 * @param destroy        destructor for this filter type
 * @param src_value_type         type of filter's values (eg. unsigned int)
 * @param snk_value_type         type of lower source's values (eg. unsigned
 *                               int)
 * @param snk_get_param_type     type of the parameters used in
 *                               filter's get_value calls
 * @param attrs          NULL-terminated array of custom
 *                       scheduler_filter_attributes, or NULL
 */
#define SCHEDULER_FILTER_TYPE_WITH_SINK_INPUT(name, new, destroy,
        src_value_type, snk_value_type, snk_get_param_type, attrs)

/**
 * Convenience macro to define a filter type using whose "get"
 * callback accepts arguments only when queried.
```

```
 * The filter type combines the filter attributes and filter
 * instance constructors/destructors.
 *
 * The "get", "update_value", and "show" callback functions must
 * be defined using DEFINE_SCHEDULER_FILTER_* macros.
 *
 * @param name          unique name among schuduler_port_type names. This
 *                      name is used for both the variable and the user-visible
 *                      type name.
 * @param new           constructor for this filter type
 * @param destroy       destructor for this filter type
 * @param src_value_type        type of filter's values (eg. unsigned int)
 * @param src_get_param_type    type of the parameters for the
 *                              get_value method
 * @param snk_value_type        type of lower source's values (eg. unsigned
 *                              int)
 * @param attrs         NULL-terminated array of custom
 *                      scheduler_filter_attributes, or NULL
 */
#define SCHEDULER_FILTER_TYPE_WITH_SOURCE_INPUT(name, new, destroy,
        src_value_type, src_get_param_type, snk_value_type, attrs)
```

After the filter and its functions have been implemented, a developer only has to define module initialization and cleanup functions. These functions are very simple for filters: they are used only for registering (or unregistering) filters with PlugProPol framework. When a particular filter is registered with PlugProPol framework, the administrator can add it to filter chains of different scheduling objects so it enforces a particular filtering policy to the resource measurement data. The cleanup function is used for unregistering filter and for releasing all the memory taken by it (see listing 5.7).

Listing 5.7: Definition of functions for registering and unregistering filters.

```
/**
 * Register a new filter type.
 *
 * @param type          type initialized with SCHEDULER_FILTER_TYPE[_INIT]
 *                      to register
 *
 * @return              0 is successful,
 *                      -EINVAL if the type is not complete,
 *                      -ENOMEM if not sufficient memory could be allocated,
 *                      -EEXIST if a filter type of the same name is already
 *                              registered
 */
int scheduler_filter_type_register(struct scheduler_filter_type *type);

/**
 * Unregister a filter type. Must can only be called at module
 * unloading.
 *
 * @param type          The filter type to unregister
 */
void scheduler_filter_type_unregister(struct scheduler_filter_type *type);
```

Chapter 6

Conclusion and future work

In this deliverable, we stated the current progress we made on implementation of customizable scheduler. We described the structure of PlugProPol framework for dynamic loading, unloading and connection of probes and scheduling policies from user space. We also listed PlugProPol's main features, gave a detailed instruction on how to install and use it and provided syntax, description and examples of all the commands that can be used to control it. Finally, we documented PlugProPol's API for all the developers that would like to write their own probes and scheduling policies for it.

As a part of our future work, we are first planning to finish an implementation of DRMAA [14] interface to the scheduler, which is used for the submission and control of jobs to one or more Distributed Resource Management (DRM) systems. This would enable all the applications that support this standard (e.g. all the applications that are written for the Torque and PBS resource management systems) to be executed on LinuxSSI. Of course, we will make sure that the SAGA (i.e. Simple API Grid Application) interface [10, 5] will also be supported, at least the part of it that is related to job submission. Regarding PlugProPol framework, we are also planning to implement various probes (e.g. I/O probes for monitoring hard disk and network traffic) and scheduling policies (e.g. load policies that are able to handle inter-process dependencies such as IPC or shared memory). Among others, we are also planning to implement scheduling policies, which will enable applications to be migrated only to the nodes with certain properties and the ones that would make sure that some processes are never located on the same cluster node.

Finally, we are planning on extending LinuxSSI scheduler with feedback loops for self-adaptability just as described in Section 1.3 (the probes should adapt their probing period and complete scheduling policies could be switched based on the load in the cluster).

Bibliography

[1] Configfs pseudo file system. `http://lwn.net/Articles/148973/`, `http://lwn.net/Articles/130342/`, `http://lwn.net/Articles/148987/`.

[2] A. Barak and A. Shiloh. A distributed load-balancing policy for a multicomputer. *Softw. Pract. Exper.*, 15(9):901–913, September 1985.

[3] R. Buyya, T. Cortes, and H. Jin. Single system image. *Int. J. High Perform. Comput. Appl.*, 15(2):124–135, 2001.

[4] XtreemOS consortium. Annex 1 - description of work. Integrated Project, April 2006.

[5] XtreemOS consortium. First draft specification of programming interfaces. Deliverable D3.1.1, November 2006.

[6] XtreemOS consortium. Specification of federation resource management mechanisms. Deliverable D2.1.1, November 2006.

[7] XtreemOS consortium. Prototype of the basic version of linuxssi. Deliverable D2.2.7, November 2007.

[8] C. Franke, F. Hoffmann, J. Lepping, and U. Schwiegelshohn. Development of scheduling strategies with genetic fuzzy systems. *Applied Soft Computing Journal*, 8(1):706Ű–721, January 2008.

[9] C. Franke, J. Lepping, and U. Schwiegelshohn. Genetic fuzzy systems applied to online job scheduling. In *Proceedings of the 2007 IEEE International Conference on Fuzzy Systems*, pages 1573–1578, London, June 2007. IEEE, IEEE Press.

[10] T. Goodale, S. Jha, T. Kielmann, A. Merzky, J. Shalf, and C. Smith. A Simple API for Grid Applications (SAGA). Grid Forum Working Draft, Open Grid Forum, 2006.

[11] O. Pomerantz P. Salzman, M. Burian. The linux kernel module programming guide. `http://www.tldp.org/LDP/lkmpg/2.6/html/book1.htm`, 2001.

[12] Linux Reviews. Kernel configuration. `http://linuxreviews.org/sysadmin/kernel-configuration/`, 2008.

[13] B. Tierney, R. Aydt, D. Gunter, W. Smith, V. Taylor, R. Wolski, and M. Swany. A grid monitoring architecture. Technical Report GWD-PERF-16-2, Global Grid Forum (OGF), January 2002.

[14] P. Tröger, H. Rajic, A. Haas, and P. Domagalski. Standardization of an api for distributed resource management systems. In *Proceedings of the Seventh IEEE International Symposium on Cluster Computing and the Grid (CCGrid 2007)*, pages 619–626, Rio de Janeiro, Brazil, May 14-17 2007.

[15] A. Vasudevan. The linux kernel howto. `http://linuxreviews.org/sysadmin/kernel-configuration/`, 2008.

Appendix

startup_mosix_scheduler.sh

```
#!/bin/sh

# name of the test scheduler.
SCHEDULER_NAME=test-scheduler
# 2 seconds between each migration.
MIGRATION_INTERVAL=2000000000

# configfs path for loading Mosix load balancer.
POLICY_PATH="schedulers/${SCHEDULER_NAME}/configfs-mosix-load-balancer"
# configfs path for loading Mosix probe for monitoring CPU load.
MOSIX_PROBE_PATH=probes/configfs-mosix-probe
# configfs path for loading migration probe (probe which monitors
# if migration is in progress).
MIGRATION_PROBE_PATH=probes/migration_probe

# go to the ConfigFS directory of LinuxSSI scheduler.
cd /config/krg_scheduler

# load Mosix probe.
mkdir $MOSIX_PROBE_PATH
# load migration probe.
mkdir $MIGRATION_PROBE_PATH
# create "test-scheduler" scheduler and load Mosix load balancing
# policy.
mkdir -p $POLICY_PATH

# create a filter that limits frequency of migrations and attach
# it to Mosix load balancer.
mkdir ${POLICY_PATH}/local_load/freq_limit_filter
# set a period between subsequent migrations to be at least
# MIGRATION_INTERVAL.
echo $MIGRATION_INTERVAL > _
${POLICY_PATH}/local_load/freq_limit_filter/min_interval

# create a filter that doesn't perform migration unless CPU load
# is greater than a given threshold. At the same time, attach it
# to "freq_limit_filter" filter.
mkdir ${POLICY_PATH}/local_load/freq_limit_filter/threshold_filter

# link "last_event" attribute of "freq_limit_filter" filter to
# "last_migration" attribute of migration probe.
ln -s ${MIGRATION_PROBE_PATH}/last_migration _
${POLICY_PATH}/local_load/freq_limit_filter/last_event/migration
# link "events_on_going" attribute of "freq_limit_filter" filter
# to "migration_on_going" attribute of migration probe.
```

```
ln -s ${MIGRATION_PROBE_PATH}/migration_on_going _
${POLICY_PATH}/local_load/freq_limit_filter/events_on_going/migration
# link "process_load" attribute of Mosix load balancer to
# "process_load" attribute of Mosix probe.
ln -s ${MOSIX_PROBE_PATH}/process_load _
${POLICY_PATH}/process_load/mosix
# link "norm_upper_load" attribute of Mosix load balancer to
# "norm_upper_load" attribute of Mosix probe.
ln -s ${MOSIX_PROBE_PATH}/norm_upper_load _
${POLICY_PATH}/remote_load/mosix_upper
# link "single_process_load" attribute of Mosix load balancer to
# "norm_single_process_load" attribute of Mosix probe.
ln -s ${MOSIX_PROBE_PATH}/norm_single_process_load _
${POLICY_PATH}/single_process_load/mosix
# link "mosix_mean" attribute of Mosix "threshold_filter" filter to
# "norm_mean_load" attribute of Mosix probe.
ln -s ${MOSIX_PROBE_PATH}/norm_mean_load _
${POLICY_PATH}/local_load/freq_limit_filter/threshold_filter/mosix_mean
```

mem_probe.c

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/mm.h>

#include <scheduler/configfs-probe.h>
#include <scheduler/configfs-const.h>
#include "mem_probe.h"

#define K(x) ((x) << (PAGE_SHIFT -10))

static mem_probe_data_t probe_data;
static mem_probe_data_t probe_data_prev;

static struct probe *mem_probe;

// define "get" callback function for "mem-free" attribute
DEFINE_PROBE_VALUE_GET(mem_free, unsigned long, value_p, nr)
{
        *value_p = K(probe_data.ram_free);

        return 1;
}

// define "show" callback function for "mem-free" attribute
DEFINE_PROBE_VALUE_SHOW(mem_free, page)
{
        ssize_t ret;

        ret = sprintf(page, "%lu\n", K(probe_data.ram_free));

        return ret;
}

// define "get" callback function for "mem-total" attribute
DEFINE_PROBE_VALUE_GET(mem_total, unsigned long, value_p, nr)
{
        *value_p = K(probe_data.ram_total);

        return 1;
}
```

```
// define "show" callback function for "mem-total" attribute
DEFINE_PROBE_VALUE_SHOW(mem_total, page)
{
        ssize_t ret;

        ret = sprintf(page, "%lu\n", K(probe_data.ram_total));

        return ret;
}

// define probe measurement function
static void measure_mem(void)
{
        struct sysinfo meminfo;

        si_meminfo(&meminfo);

        probe_data.ram_free = meminfo.freeram;
        probe_data.ram_total = meminfo.totalram;
}

// define "value_changed" callback function for "mem-total" attribute
DEFINE_PROBE_VALUE_HAS_CHANGED(mem_total)
{
        int isChanged = 0;

        if (probe_data.ram_total != probe_data_prev.ram_total) {
                isChanged = 1;

                probe_data_prev.ram_total = probe_data.ram_total;
        }

        return isChanged;
}

// define "mem-total" probe attribute.
static PROBE_VALUE_TYPE(mem_total, NULL, unsigned long);

// define "value_changed" callback function for "mem-free" attribute
DEFINE_PROBE_VALUE_HAS_CHANGED(mem_free)
{
        int isChanged = 0;

        if (probe_data.ram_free != probe_data_prev.ram_free) {
                // the value has changed
                isChanged = 1;
                // update the previous value with current value.
                probe_data_prev.ram_free = probe_data.ram_free;
        }

        return isChanged;
}

// define "mem-free" probe attribute
static PROBE_VALUE_TYPE(mem_free, NULL, unsigned long);

static struct probe_value *mem_probe_values[3];

// define memory probe
static PROBE_TYPE(mem_probe_type, NULL, measure_mem);
```

```
int init_module()
{
        int err = -ENOMEM;

   // create probe attributes
        mem_probe_values[0] = krg_probe_value_create(&mem_free,
                "mem-free");
        mem_probe_values[1] = krg_probe_value_create(&mem_total,
                "mem-total");
        mem_probe_values[2] = NULL;

        if (mem_probe_values[0]==NULL || mem_probe_values[1]==NULL) {
                printk(KERN_ERR "error: cannot initialize mem_probe "
                        "attributes\n");
                goto out_kmalloc;
        }

        // create probe
        mem_probe = krg_probe_create(&mem_probe_type, MEM_PROBE_NAME,
                mem_probe_values);
        if (mem_probe == NULL) {
                printk(KERN_ERR "error: mem_probe creation failed!\n");
                goto out_kmalloc;
        }

        // perform first measurement
        measure_mem();
        probe_data_prev = probe_data;

        // register probe with PlugProPol framework
        err = krg_probe_register(mem_probe);
        if (err)
                goto err_register;

        return 0;

err_register:
        krg_probe_free(mem_probe);
out_kmalloc:
        if (mem_probe_values[0] != NULL)
                krg_probe_value_free(mem_probe_values[0]);
        if (mem_probe_values[1] != NULL)
                krg_probe_value_free(mem_probe_values[1]);

        return err;
}

void cleanup_module()
{
        int i;
        PDEBUG(KERN_INFO "mem_probe cleanup function called!\n");
        // unregister probe
        krg_probe_unregister(mem_probe);
        // destroy probe
        krg_probe_free(mem_probe);
        for (i = 0; mem_probe_values[i] != NULL; i++)
                krg_probe_value_free(mem_probe_values[i]);
}
```

echo_policy.c

```
#include <linux/kernel.h>
#include <linux/module.h>

#include <scheduler/configfs-sched-policy.h>
#include <scheduler/configfs-end_port.h>
#include "echo_policy.h"

struct echo_policy {
        struct sched_policy policy;
        struct scheduler_end_port *port_mem_free;
        struct scheduler_end_port *port_mem_total;
};

// define "show" callback function for "echo-attr" attribute
static ssize_t sched_policy_attr_echo_show(struct sched_policy *item,
        char *page) {
        ssize_t ret = 0;

        ret = sprintf(page, "calling sched_policy_attr_echo_show!\n");

        return ret;
}

// define "echo-attr" attribute
static struct sched_policy_attribute echo_attr = {
        .attr = {
                .ca_owner = THIS_MODULE,
                .ca_name = "echo-attr",
                .ca_mode = S_IRUGO,
        },
        .show = sched_policy_attr_echo_show,
};

// define a list of policy's attributes.
static struct sched_policy_attribute *echo_policy_attrs[] = {
        &echo_attr,
        NULL,
};

// define "update_value" callback function for "port_mem_free"
// end port of "echo-policy"
static void update_mem_free(struct scheduler_end_port *port, void *private)
{
        unsigned long mem_free;

        if (SCHEDULER_END_PORT_GET_VALUE(port,
                        &mem_free, 1) > 0) {
                printk(KERN_INFO "echo_policy: mem_free=%lu\n", mem_free);
        }
        else {
                printk(KERN_ERR "echo_policy: cannot read mem_free\n");
        }
}

// define "port_mem_free" end port
static SCHEDULER_END_PORT_TYPE(port_mem_free_type, NULL,
        unsigned long);

// define "update_value" callback function for "port_mem_total"
// end port of "echo-policy"
static void update_mem_total(struct scheduler_end_port *port, void *private)
{
```

```
        unsigned long mem_total;

        if (SCHEDULER_END_PORT_GET_VALUE(port,
                                         &mem_total, 1) > 0) {
                printk(KERN_INFO "echo_policy: mem_total=%lu\n", mem_total);
        }
        else {
                printk(KERN_ERR "echo_policy: cannot read mem_total\n");
        }
}

// define "port_mem_free" end port
static SCHEDULER_END_PORT_TYPE(port_mem_total_type, update_mem_total,
        unsigned long);

static struct sched_policy * echo_policy_new(const char *name);
static void echo_policy_destroy(struct sched_policy *policy);

// define constructor and destructor for "echo_policy" scheduling
// policy
static struct sched_policy_operations echo_policy_ops = {
        .new = echo_policy_new,
        .destroy = echo_policy_destroy,
};

// define "echo_policy" scheduling policy
static SCHED_POLICY_TYPE(echo_policy_type, "echo_policy",
       &echo_policy_ops, echo_policy_attrs);

// define constructor for "echo_policy" scheduling policy
static struct sched_policy * echo_policy_new(const char *name)
{
        struct echo_policy *p;
        struct config_group *def_groups[3];
        int err;

        p = kmalloc(sizeof(*p), GFP_KERNEL);
        if (!p)
                goto err_echo_policy;

        // create end ports of "echo_policy" scheduling policy
        p->port_mem_free = scheduler_end_port_create("port_mem_free",
                                                     &port_mem_free_type,
                                                     p);
        p->port_mem_total = scheduler_end_port_create("port_mem_total",
                                                      &port_mem_total_type,
                                                      p);
        if (p->port_mem_free==NULL || p->port_mem_total==NULL)
                goto out_def_groups;

        // initialize default memory groups.
        def_groups[0] = scheduler_end_port_config_group(p->port_mem_free);
        def_groups[1] = scheduler_end_port_config_group(p->port_mem_total);
        def_groups[2] = NULL;

        // initialize scheduling policy
        err = krg_sched_policy_init(&p->policy, name, &echo_policy_type,
                                    def_groups);
        if (err)
                goto err_policy;

        return &p->policy;
```

```
err_policy:
out_def_groups:
        if (p->port_mem_free != NULL)
                scheduler_end_port_free(p->port_mem_free);
        if (p->port_mem_total != NULL)
                scheduler_end_port_free(p->port_mem_total);
err_echo_policy:
        printk(KERN_ERR "error: echo_policy creation failed!\n");
        return NULL;
}

// define destructor for "echo_policy" scheduling policy
static void echo_policy_destroy(struct sched_policy *policy)
{
        struct echo_policy *p =
                container_of(policy, struct echo_policy, policy);
        // destroy scheduling policy
        krg_sched_policy_cleanup(policy);
        // destroy scheduling policy endports
        scheduler_end_port_free(p->port_mem_free);
        scheduler_end_port_free(p->port_mem_total);
        kfree(p);
}

int init_module(void)
{
        int err;

        // initialize scheduling policy end ports
        err = scheduler_end_port_type_init(&port_mem_free_type);
        if (err)
                goto err_mem_free;
        err = scheduler_end_port_type_init(&port_mem_total_type);
        if (err)
                goto err_mem_total;

        // register scheduling policy with PlugProPol framework
        err = krg_sched_policy_type_register(&echo_policy_type);
        if (err)
                goto err_register;
 out:
        return err;

 err_register:
        scheduler_end_port_type_cleanup(&port_mem_total_type);
 err_mem_total:
        scheduler_end_port_type_cleanup(&port_mem_free_type);
 err_mem_free:
        goto out;
}

void cleanup_module(void)
{
        // unregister scheduling policy
        krg_sched_policy_type_unregister(&echo_policy_type);
        // destroy scheduling policy end ports
        scheduler_end_port_type_cleanup(&port_mem_total_type);
        scheduler_end_port_type_cleanup(&port_mem_free_type);
}
```

threshold_filter.c

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/types.h>

#include <scheduler/configfs-filter.h>

struct threshold_filter {
        struct scheduler_filter filter;
        unsigned long threshold;
};

#define to_threshold_filter(_filter) \
        container_of((_filter), struct threshold_filter, filter)

// define the "show" callback function of the "threshold" filter
// attribute
static ssize_t threshold_attribute_show(
        struct scheduler_filter *filter, char *page)
{
        struct threshold_filter *f = to_threshold_filter(filter);
        return sprintf(page, "%lu", f->threshold);
}

// define the "store" callback function of the "threshold" filter
// attribute
static ssize_t threshold_attribute_store(
        struct scheduler_filter *filter,
        const char *page, size_t count)
{
        struct threshold_filter *f = to_threshold_filter(filter);
        unsigned long new_value;
        char *last_read;

        new_value = simple_strtoul(page, &last_read, 0);
        f->threshold = new_value;

        printk("count=%u read=%d\n", count, last_read - page);
        return count;
}

// define a "threshold" filter attribute
static SCHEDULER_FILTER_ATTRIBUTE(threshold_attr, "threshold", 0666,
        threshold_attribute_show,
        threshold_attribute_store);

// define a list of filter attributes
static struct scheduler_filter_attribute *threshold_attrs[] = {
        &threshold_attr,
        NULL
};

// defince the "get" callback function for the filter
DEFINE_SCHEDULER_FILTER_GET_VALUE_SIMPLE(threshold_filter);
// defince the "show" callback function for the filter
DEFINE_SCHEDULER_FILTER_SHOW_VALUE_SIMPLE(threshold_filter);
// defince the "update_value" callback function for the filter
DEFINE_SCHEDULER_FILTER_UPDATE_VALUE(threshold_filter, filter)
{
        struct threshold_filter *f = to_threshold_filter(filter);
```

```
        unsigned long value;
        ssize_t ret;

        ret = SCHEDULER_FILTER_SIMPLE_GET_VALUE(filter, &value, 1);
        if (ret > 0 && value >= f->threshold)
                scheduler_filter_simple_update_value(filter);
}
// defince the "remote_get_value" callback function for the filter
DEFINE_SCHEDULER_FILTER_GET_REMOTE_VALUE_SIMPLE(threshold_filter);

static struct scheduler_filter * threshold_new(const char *name);
static void threshold_destroy(struct scheduler_filter *port);

// define "threshold_filter" filter
static SCHEDULER_FILTER_TYPE(threshold_filter, threshold_new,
        threshold_destroy,bunsigned long, unsigned long, threshold_attrs);

// define constructor for "threshold_filter" filter
static struct scheduler_filter * threshold_new(const char *name)
{
        struct threshold_filter *f = kmalloc(sizeof(*f), GFP_KERNEL);
        int err;

        if (!f)
                goto err_f;
        err = scheduler_filter_init(&f->filter, name, &threshold_filter);
        if (err)
                goto err_filter;
        f->threshold = 0;

        return &f->filter;

 err_filter:
        kfree(f);
 err_f:
        return NULL;
}

// define destructor for "threshold_filter" filter
static void threshold_destroy(struct scheduler_filter *filter)
{
        struct threshold_filter *f = to_threshold_filter(filter);
        scheduler_filter_cleanup(filter);
        kfree(f);
}

static int init_module(void)
{
        // register filter with PlugProPol framework
        return scheduler_filter_type_register(&threshold_filter);
}

static void cleanup_module(void)
{
        // unregister filter
        scheduler_filter_type_unregister(&threshold_filter);
}
```