Parallels

Managing UBC Resources in Parallels Virtuozzo Containers 4.0

ISBN: N/A
Parallels
13755 Sunrise Valley Drive
Suite 600
Herndon, VA 20171
USA
Tel: +1 (703) 815 5670
Fax: +1 (703) 815 5675

Contents

Preface 5

About This Guide 5
Who Should Read This Guide 5
Organization of This Guide 6
Documentation Conventions....6
Typographical Conventions....7
Shell Prompts in Command Examples 7
General Conventions 8
Feedback....8

Basic Management of System Resources 9

Basics of Resource Control....9
Resource Control and System Security 10
Resource Control Principles 11
Other Existing Resource Control Mechanisms....11
Summary 13
Resource Management Parameters Overview 13
Configuring Resource Management Parameters for Containers....15
Using Existing Configurations 16
Scaling Configuration....17
Creating Configuration for Fraction on Server....18
Validating Configuration....19
Checking Resources Configuration and Real Usage 21
Summary of Configuration Creation Methods 23
Adjusting Individual Resource Management Parameters....24
Controlling CPU Time Consumption 24
Types of Virtuozzo CPU Time Consumption Control 25
Configuring and Inspecting CPU Time Consumption Control Settings....26

Advanced Management of System Resources 29

System Resource Control Parameters in Detail 29
Overview 31
Primary Parameters 33
Secondary Parameters....36
Auxiliary Parameters 42
Inspecting Resource Control Settings....45
Verifying and Troubleshooting Resource Control Configuration....47
Checking Resource Control Settings for Applications 48
Checking Configuration Consistency of Single Container....53
Checking Usage and Distribution of System Resources and Validating Settings of All Containers55
Checking Resource Usage and Settings With Advanced Utilities....61
Monitoring Resources 65

Changes From Previous Releases 68

Changes Between Virtuozzo 2.6 and 3.0 68
Changes Between Virtuozzo 2.5.1 and 2.6 68
Changes Between Virtuozzo 2.5 and 2.5.1 68
Changes Between 2.0.2 and 2.5 69

Examples 71

Examples of System Resources Parameter Settings 71
Examples Explanation 73
Not Specified Values 74

Index 75

CHAPTER 1

Preface

In This Chapter

About This Guide 5
Who Should Read This Guide 5
Organization of This Guide 6
Documentation Conventions 6
Feedback 8

About This Guide

This document discusses management of system resources in Parallels Virtuozzo installations: the control of the usage of system resources by Containers and the administrative procedures related to this control. System resources include different kinds of memory and operating system resources (such as number of processes). Control of CPU time consumption is also briefly considered in this document.

Who Should Read This Guide

The primary audience for this guide is anyone looking for additional ways to customize memory limits for and to provide memory guarantees to Containers. To efficiently use this document, the reader should:

- understand basic Virtuozzo concepts such as Container;
- be familiar with Linux command line tools (such as `ps(l)`, `cat(l)`, `grep(l)`) and some shell (preferably, `bash(l)`);
- be familiar with the `man(l)` command, system of Linux manual pages, and be able to navigate in it;
- be familiar with Red Hat Linux layout of system files and directories (however, the actual system being configured doesn't need to be Red Hat Linux).

Familiarity with other resource control mechanisms (e.g. `sysctl (2)`) and experience in using debugging tools (e.g. `strace(D)`) will help, but is not mandatory.

Organization of This Guide

Chapter 2, Basics Management of System Resources, discusses the following points:

- The Basics of Resource Controls section deals with the resource control in general: what resources are accounted, whom they are attributed, and how resource consumption is limited or controlled. A brief description of standard Linux resource control mechanisms is also provided. Those standard mechanisms may be optionally used as supplementary to Virtuozzo resource control.
- The Resource Management Parameters Overview section introduces the notion of resource management parameter and lists all the parameters.
- The Configuring Resource Management Parameters for Containers section explains different methods of configuring resource management parameters for newly created Containers, explains how to make sure the resource configuration is valid and emphasizes that only as many Containers as the system is capable to support should be run on the system.
- The Controlling CPU Time Consumption section briefly discusses control of CPU time consumption by Containers.

Chapter 3, Advanced Management of System Resources, covers the following points:

- The System Resource Control Parameters in Detail section considers in more detail parameters controlling system resources. This chapter is intended for Virtuozzo administrators who want to understand Virtuozzo resource control deeper. Parameters controlling system resources are combined into 3 groups - primary, secondary, and auxiliary. Meanings of the parameters are explained, and tips how to configure each parameter individually are provided. It also explains how to view the configuration of the parameters and get information about the resource usage.
- The Verifying and Troubleshooting Resource Control Configuration section covers advanced topics and is intended for developers of new scripts and programs managing and monitoring Virtuozzo system and experienced system administrators troubleshooting application problems in Virtuozzo systems or willing to fine-tune or optimize resource control configurations.

Appendix A outlines the difference between the current and previous Virtuozzo releases with respect to resource control.

Appendix B contains examples of complete resource control configurations and comments about what these configurations are good for.

Documentation Conventions

Before you start using this guide, it is important to understand the documentation conventions used in it. For information on specialized terms used in the documentation, see the Glossary at the end of this document.

Typographical Conventions

The following kinds of formatting in the text identify special information.

Formatting convention	Type of Information	Example
Triangular Bullet(➢)	Step-by-step procedures. You can follow the instructions below to complete a specific task.	*To create a Container:*
Special Bold	Items you must select, such as menu options, command buttons, or items in a list.	Go to the Resources tab.
	Titles of chapters, sections, and subsections.	Read the Basic Administration chapter.
Italics	Used to emphasize the importance of a point, to introduce a term or to designate a command line placeholder, which is to be replaced with a real name or value.	These are the so-called *EZ templates*. To destroy a Conainer, type `vzctl destroy` *`ctid`*.
`Monospace`	The names of commands, files, and directories.	Use `vzctl start` to start a Container.
`Preformatted`	On-screen computer output in your command-line sessions; source code in XML, C++, or other programming languages.	`Saved parameters for Container 101`
`Monospace Bold`	What you type, contrasted with on-screen computer output.	**`# rpm –V virtuozzo-release`**
CAPITALS	Names of keys on the keyboard.	SHIFT, CTRL, ALT
KEY+KEY	Key combinations for which the user must press and hold down one key and then press another.	CTRL+P, ALT+F4

Shell Prompts in Command Examples

Command line examples throughout this guide presume that you are using the Bourne-again shell (bash). Whenever a command can be run as a regular user, we will display it with a dollar sign prompt. When a command is meant to be run as root, we will display it with a hash mark prompt:

Bourne-again shell prompt	$
Bourne-again shell root prompt	#

General Conventions

Be aware of the following conventions used in this book.

- Chapters in this guide are divided into sections, which, in turn, are subdivided into subsections. For example, Documentation Conventions is a section, and General Conventions is a subsection.
- When following steps or using examples, be sure to type double-quotes ("), left single-quotes (`), and right single-quotes (') exactly as shown.
- The key referred to as RETURN is labeled ENTER on some keyboards.

The root path usually includes the `/bin`, `/sbin`, `/usr/bin` and `/usr/sbin` directories, so the steps in this book show the commands in these directories without absolute path names. Steps that use commands in other, less common, directories show the absolute paths in the examples.

Feedback

If you spot a typo in this guide, or if you have thought of a way to make this guide better, we would love to hear from you!

The SWsoft documentation forum is the ideal place for your comments and suggestions. It is regularly monitored by the members of the SWsoft technical documentation department, so it is likely that you will receive a reply to your post before long.

Note that new users will be asked to fill in a short registration form before being able to post. Registering will allow you to participate not only in the documentation forum discussions, but in all the other SWsoft forums as well.

CHAPTER 2

Basic Management of System Resources

In This Chapter

Basics of Resource Control 9
Resource Management Parameters Overview 13
Configuring Resource Management Parameters for Containers 15
Controlling CPU Time Consumption 24

Basics of Resource Control

The main goals of resource control are:

- prevention of hangs of the server or other serious impacts of resource over-consumption by one of the Containers, both if the over-consumption is accidental or malicious;
- enforcement of certain "quality" of service for different Containers, including enforcement of fairness of resource usage among the environments and, if necessary, preferred quality of service for selected environments.

Virtuozzo provides additional capabilities to control memory and operating system resources in comparison with standard Linux. This additional resource control module is historically called "User Beancounter Patch". It accounts how much resources each Container consumes and allows administrators to control availability of resources to the environments.

In addition to the control of memory and system resources, Virtuozzo provides:

- CPU usage control (detailed information on this resource is given in the Controlling CPU Time Consumption section (on page 24));
- disk quota (please see the Managing Resources chapter of Parallels Virtuozzo Containers User's Guide);
- network traffic accounting and shaping (please see the Managing Resources chapter of Parallels Virtuozzo Containers User's Guide).

Resource Control and System Security

Virtuozzo resource control is an important security mechanism. It provides protection from accidental or deliberate over-usage of resources by Containers affecting the service level of other Containers. Thus, the resource control provides:

- *fault isolation*, i.e. makes sure that failures (or misbehavior) of applications belonging to one Container do not affect other Containers and
- *performance isolation*, i.e. makes sure that each Container has certain performance, and that activity in one Container can't cause severe performance degradation for other Containers.

Virtuozzo resource control protects Containers from denial-of-service attacks by other Containers hosted on the same physical server. The example attacks prevented by Virtuozzo resource control are:

- CPU overload (an attack by spawning a big number of processes each consuming CPU in a loop), prevented by Hierarchical Fair CPU Scheduling,
- "fork bomb" (an attack by just spawning a huge number of processes), prevented by the `numproc` and `kmemsize` resource limits,
- "malloc bomb" (an attack by allocating a lot of memory in a loop), prevented by the `privvmpages` resource limits,
- "socket/pipe bomb" (an attack by creating a lot of sockets or pipes and writing but not reading data), prevented by the `othersockbuf` and `kmemsize` resource limits).

There are many other attacks that can be prevented by Virtuozzo resource control.

Another possible attack is the attack on the disk bandwidth, attempting to leave as little bandwidth to other Containers as possible. This attack is possible theoretically, but is difficult to be conducted efficiently The attacker's abilities in practice are limited, in particular, by the disk quota. In any case, the other Containers and the host system will continue to function under this attack (although, with bad responsiveness of the applications). Such attacks may be traced by ps(l) or other means and stop by administrative measures. Future Virtuozzo versions will provide full automatic protection from disk bandwidth attacks, too.

For the protection from denial-of-service attacks to be in effect, administrators must be sure that resource control settings are safe and one Containers can't eat up all the system resources. The validation of the resource control settings in this respect is considered in the Checking Resources Configuration and Real Usage (on page 21) and Checking Usage and Distribution of System Resources and Validating Settings for all Containers (on page 55) sections.

Resource Control Principles

Virtuozzo resource management deals with 4 groups of resources:

- system resources (extensively covered in the given guide);
- CPU resources (covered in the Controlling CPU Time Consumption section (on page 24));
- disk resources, and
- network resources (covered in Parallels Virtuozzo Containers User's Guide).

For each group of resources, Virtuozzo provides new management functionality in comparison with standard Linux systems. This section explains this new functionality and difference between Virtuozzo management of system resources and traditional resource control mechanisms existing in all Linux and Unix-like systems.

Virtuozzo accounts and controls resources used by Containers. Inside each Container like in a stand-alone system its administrator can use the traditional resource control mechanisms such as

- system-wide limits on certain resources, such as IPC SysV shared memory segments; those system-wide limits apply to the Container where they are configured, see the System-Wide Limits subsection (on page 12);
- per-process limits configured through the `setrlimit (2)` interface, see the Per-Process Limits subsection (on page 12);
- per-UID limits such as disk quota or limit on the number of processes.

Note: Theoretically, system administrators can set up resource control for arbitrary groups of processes formed by inheritance. The example of such alternative setup is resource control for all processes of a particular user of a Unix-like system. That is, in Virtuozzo, Containers are "users" for the purposes of resource control.

Resource management in Virtuozzo controls physical resources (such as RAM), operating system resources (such as number of processes) and some artificial counters, called in general "parameters". All the parameters of system resource control are discussed in detail in the System Resource Control Parameters in Detail section (on page 29).

Depending on the parameter, the resource control is implemented as

- limits, i.e. upper boundaries on what this Container can consume, and
- guarantees, i.e. mechanisms ensuring that this Container can get the assigned "resources" regardless of the activity and the amount of resources required by other Containers.

Other Existing Resource Control Mechanisms

The main mechanisms controlling system resources and existing in Linux installations without Virtuozzo (and other Unix-like systems) are

- system-wide limits and
- per-process limits (the `setrlimit(2)` interface).

System-Wide Limits

The first mechanism limits the number of some operating system objects and, for some objects, their sizes, such as:

- `shmmni`: the maximum number of shared memory segments in the system;
- `shmall`: the maximum total size of all shared memory segments in the system;
- `rtsig-max`: the maximum number of real-time (queued) signals in the system;
- `tcp_mem`: the maximum total size of buffers of all sockets in the system.

Some of the system-wide limits (such as the maximum number of processes) may be not available for tuning by `sysctl(8)`, but are compiled into the kernel. Other limits are exposed to system administrators as `sysctl` parameters and can be tuned runtime through the `/proc/sys` interface. 4 limits listed above are examples from the list of about a hundred of the `sysctl(8)` parameters that can be tuned in a typical Linux system.

Administrators of Containers can tune some limits, such as `shmmni`, inside their Containers. The configured limit will apply to this Container only.

Other limits, such as `rtsig-max` and `tcp_mem`, in Virtuozzo systems apply only to processes of the host system, outside any Containers. They do not affect processes inside Containers and cannot be changed from inside them.

Per-Process Limits

The `setrlimit (2)` interface allows system administrators and individual users to set certain limits on each process. Additionally, the same interface can set limits on the total number of processes one user of a Unix-like system can run. Individual users are allowed to lower but not to raise the limits imposed by the administrator. The processes may be limited in

- the size of their data segment;
- the size of their stack;
- the amount of memory locked by `mlock` by this process;
- the maximum CPU time this process can run;
- the maximum size of files the process can create;
- the maximum number of files the process can open and some other parameters.

More information is available in the `setrlimit(2)` manual pages and the help page for `ulimit` built-in of the `bash` shell. The main purpose of these limits is to cause misbehaving processes to fail earlier, before they consume too much resources and the system starts to experience resource shortage and slows down.

Summary

Resource control in Virtuozzo systems is multi-level. Virtuozzo administrators manage physical resources of the system and control resources available to Containers. Administrators inside Containers can use standard Linux resource control mechanisms.

Virtuozzo resource control (i.e. the control of resources available to Containers) is performed in different terms than the standard Linux one. Virtuozzo resource control manages resources for new entities - Containers, and provides real protection from Denial-of-Service attacks or accidental application misbehavior inside Containers. The `setrlimit(2)` interface has a not very practical set of controlled resources and because it controls the resources on a per-process basis, being unable to control the resources consumed by a group of processes as a whole.

Resource Management Parameters Overview

Administrators control the resources available to the Container through a set of its resource management parameters. These parameters are presented in the following table:

Name	Type	Description
Main parameters		
`numproc`	`system`	Number of processes and threads.
`numtcpsock`	`system`	Number of TCP sockets.
`numothersock`	`system`	Number of sockets other than TCP.
`vmguarpages`	`system`	Memory allocation guarantee.
`cpuunits`	`cpu`	CPU power.
`diskspace`	`disk`	Disk space quota.
`rate`	`network`	Network bandwidth.
`ratebound`	`network`	Whether network bandwidth is limited.
Additional parameters		
`kmemsize`	`system`	Size of unswappable kernel memory allocated for processes in this Container.
`tcpsndbuf`	`system`	Total size of TCP send buffers.
`tcprcvbuf`	`system`	Total size of TCP receive buffers.
`othersockbuf`	`system`	Total size of UNIX-domain socket buffers, UDP and other datagram protocol send buffers.
`dgramrcvbuf`	`system`	Receive buffers of UDP and other datagram protocols.
`oomguarpages`	`system`	The guaranteed amount of memory for the case the memory is "over-booked" (out-of-memory kill guarantee).
`privvmpages`	`system`	Memory allocation limit.

lockedpages	system	Process pages not allowed to be swapped out (pages locked by mlock(2)).
shmpages	system	Total size of shared memory (IPC, shared anonymous mappings and tmpfs objects), in pages.
physpages	system	Total number of RAM pages used by processes.
numfile	system	Number of open files.
numflock	system	Number of file locks.
numpty	system	Number of pseudo-terminals.
numsiginfo	system	Number of siginfo structures.
dcachesize	system	Total size of dentry and inode structures locked in memory.
numiptent	system	Number of NETFILTER (IP packet filtering) entries.
cpulimit	cpu	Limit on CPU consumption.
diskinodes	disk	Disk inode quota.
quotatime	disk	Disk quota grace period.
quotaugidlimit	disk	Limit on the number of uid and gid accounting entries.

Main parameters are the ones most important for functionality, performance, and availability of the Container.

Note: Availability characterizes the property of Containers or any other server environment to stay operative and serve clients. It is usually measured as the ratio of operative time to total time. High availability is achieved by rare failures and low downtime.

The Type column shows the type of resource controlled by the parameter: system, cpu, disk, or network. This guide is focused on system resources and discusses their management. CPU resources are briefly discussed in the Controlling CPU Time Consumption section (on page 24).

Management of disk space and network resources is discussed in Parallels Virtuozzo Containers User's Guide. The corresponding parameters are shown in the table for completeness.

Configuring Resource Management Parameters for Containers

Usually, each Container needs all 28 resource management parameters to be set.

Note: If disk quota is turned off in the Virtuozzo global configuration file (`/etc/vz/vz.conf`), the disk quota-related parameters of Containers don't need to be set.

The best and easiest way to configure resource management parameters for a new Container is to produce the configuration at once, using commands operating with the configuration as a whole rather than to set each of the parameters individually. A complete configuration can be produced by

- taking a sample configuration packaged with Virtuozzo;
- taking a custom sample configuration;
- taking the configuration of another already configured Container;
- scaling a sample or some Container configuration.

The corresponding procedures of producing the configuration are considered below.

Note: After producing a configuration by any means other than simple copying, the configuration needs to be validated for correctness as discussed in the Validating Configuration subsection (on page 19).

Using Existing Configurations

An existing sample configuration of resource management parameters can be used at the moment of creation of a new Container:

```
# vzctl create CT_ID –config sample_name
```

where *sample_name* is the name of a sample configuration, either pre-packaged Virtuozzo sample or a custom one. Creation of custom configuration samples is considered in the following subsections.

Another way to use an existing sample configuration is manual copying by cp(l):

```
# cd /etc/vz/conf
# cp ve_sample_name.conf-sample CT_ID.conf
```

Similarly, a configuration of an existing Container may be used:

```
# cd /etc/vz/conf
# cp  orig_CT_ID.conf  new_CT_ID.conf
```

Please keep in mind that copying a configuration of an existing Container using cp(l) also copies other attributes of the Container, such as its hostname, IP addresses, information about the operating system and application templates and so on. Administrators should review the copied configuration and set those other attributes to the desired values by vzctl(8). Sample configurations do not contain the "unnecessary" information about hostnames, IP addresses, and so on. Also, in configurations produced by scaling (see the next subsection) the unnecessary information can be removed automatically.

Scaling Configuration

Administrators can produce new configurations of resource management parameters by scaling existing Container or sample configurations. Scaling produces configurations "heavier" or "lighter" than the original one in the given number of times. Heavier configurations are produced by multiplying the values of all parameters by a number greater than 1. Lighter configurations are produced by multiplying by a number greater than 0 but less than 1.

Note: Lighter configurations produced by multiplying by a number less than 1 always need validation, as discussed in the Validating Configuration subsection (on page 19).

Scaling of configurations can be performed by `vzcfgscale(8)`. A sample configuration can be scaled for a new Container as follows:

```
# cd /etc/vz/conf
# vzcfgscale -a factor -o CT_ID.conf ve-sample_name.conf-sample
```

where *`factor`* is the scaling factor.

Configuration of another Container can be scaled as follows:

```
# cd /etc/vz/conf
# vzcfgscale -a factor -r -o new_CT_ID.conf orig_CT_ID.conf
```

Note that invoking the `vzcfgscale(8)` utility with the `-r` option removes the unnecessary information about host names, IP addresses and similar from the original configuration file, so the resulting configuration file contains only information related to resource management.

Scaling an existing configuration is the simplest way of producing custom sample configurations. A new sample configuration may be produced from another sample:

```
# cd /etc/vz/conf
# vzcfgscale -a factor -r -o ve-new_sample_name.conf-sample \
             ve-sample_name.conf-sample
```

or from the configuration of some Container:

```
# cd /etc/vz/conf
# vzcfgscale -a factor -r -o ve-new_sample_name.conf-sample CT_ID.conf
```

Please keep in mind that scaling a configuration by `vzcfgscale(8)` utility with `factor 1` and the `-r` option is a simple way to copy configurations and strip them from unnecessary information. Factor equal to 1 and the corresponding `-a` option can be omitted in this case:

```
# cd /etc/vz/conf
# vzcfgscale -r -o ve-new_sample_name.conf-sample CT_ID.conf
```

Creating Configuration for Fraction on Server

It is possible to create a configuration of resource management parameters that roughly represents a given fraction of the server. It can be done by `vzsplit(8)` utility:

```
# vzsplit -n number -f new_sample_name
```

This command will produce a configuration for $\frac{1}{number}$ of the server. In other words, the configuration parameters are computed by `vzsplit(8)` to allow to run up to *number* of Containers based on this configuration on this server. The configuration produced by `vzsplit(8)` is saved as a sample configuration that can be used afterwards for creating new Containers. The configuration file is named `ve-`*new_sample_name*`.conf-sample` and placed in the Container configuration directory (`/etc/vz/conf`).

Note that the configurations produced by `vzsplit(8)` heavily depend on the hardware configuration of the server where `vzsplit(8)` was invoked. Thus, these configurations will not always represent the requested fraction of the server on which they are stored. If the hardware is upgraded or the Container is moved to another server, the Container configuration will not represent the requested fraction of the upgraded or new server.

Also note that the version of `vzsplit(8)` shipped with the current Virtuozzo version computes the settings for system and CPU parameters, but doesn't compute the settings for disk and network ones. Future Virtuozzo versions are going to handle all resource types in `vzsplit(8)`.

The creation of configurations by `vzsplit(8)` also requires the configuration validation after creation (please see the next subsection).

Validating Configuration

Resource control parameters are interdependent. So, arbitrary numbers assigned to the parameters do not produce valid configurations. Attempts to start a Container with an invalid resource configuration may cause

- failed start;
- successful start of the Container, but failed start of applications inside it;
- successful start of the Container, but failing, dying often or hanging applications.

So, it is important to be sure that the Container configuration is valid.

In general, every time the resource configuration of a Container is modified, it needs to be validated.

Automatic check of resource control parameter dependencies and the validation of a single Container configuration rely on this configuration only. That means that the configuration doesn't need to be validated before each start of the Container or after the Container migration to a different server. The validation of parameter settings is necessary only when the configuration is modified.

However, the configuration validation is not the only step that needs to be performed by the administrator to make sure that Containers work safely and reliably. Single system has limited resources and can run limited number of Containers. Overloading the system, i.e. placing more Containers than the system can support, hurts performance and reliability of the hosting even if the configurations of each individual Container are valid and consistent. Checking system utilization, planning and assigning reasonable number of Containers to each system is another important part of system management. Simple procedures and tools for checking system utilization and evaluating what share of the server resources the Container uses are considered in the Checking Resources Configuration and Real Usage subsection (on page 21). More information for experienced Virtuozzo administrators is provided in the Verifying and Troubleshooting Resource Control Configuration section (on page 47).

Validation Procedure

The validation of a Container resource configuration may be performed by `vzcfgvalidate(8)`. This utility can catch typical mistakes in the configuration and provide recommendations and hints about its adjustments. It can be invoked as follows:

```
# cd /etc/vz/conf
# vzcfgvalidate  config_file
```

where *`config_file`* is the name of Container or sample configuration file.

This validation utility checks the configuration according to the rules listed in the Verifying and Troubleshooting Resource Control Configuration section (on page 47) and shows which constraints are not satisfied. The example output is shown below.

```
# vzcfgvalidate lOl.conf
Error: barrier should be equal limit for numfile (currently,
1280:5000)
Error: tcpsndbuf.lim-tcpsndbuf.bar should be > 512000 (currently,
204800)
Warning: tcprcvbuf.lim-tcprcvbuf.bar should be > 512000 (currently,
204800)
Recommendation: dgramrcvbuf.bar should be > 132096 (currently, 65536)
Recommendation: othersockbuf.bar should be > 132096 (currently,
122880)
# vzcfgvalidate ve-basic.conf-sample
Recommendation: dgramrcvbuf.bar should be > 132096 (currently, 65536)
Recommendation: othersockbuf.bar should be > 132096 (currently,
122880)
Validation completed: success
```

Each message indicates the severity of the found problem:

- `Error` means that an important constraint is not satisfied and the configuration is invalid; applications in a Container with such invalid configuration have increased chances to fail unexpectedly, be terminated or hang;
- `Warning` means that some constraint is not satisfied and the configuration is invalid; applications in a Container with such invalid configuration may have suboptimal performance or fail in a not graceful way;
- `Recommendation` means that the configuration is valid in general, however, if the system has enough memory, it's better to increase the settings as advised.

Note that the `vzcfgvalidate(8)` utility checks that the Container resource settings are valid and self-consistent, but does not attempt to verify that the applications installed in the Container will be able to run under these settings, or that the application configurations (such as `httpd.conf`) match the resource settings. Applications inside the Container will be fully operational and work reliably only when the application configuration matches the resource parameter settings of the Container. Set of modules, number of processes the application is allowed spawn, number of clients the application can handle and all memory settings need to be reviewed in application configuration. This topic is discussed in the Checking That Settings Allow Applications to Function subsection (on page 48) in detail.

Automatic Validation

To be sure that the Container resource configuration is always valid, Virtuozzo administrators can make `vzctl(8)` to automatically perform configuration validation before each start of Containers. This option is controlled by the `VE_VALIDATE_ACTTON` variable in the Virtuozzo global configuration file (`/etc/vz/vz.conf`). Depending on the setting of this variable, `vzctl(8)` can issue a warning, exit with an error, or fix the configuration to meet all constraints on the parameter values. In the current Virtuozzo version, the default for this option is a non-automatic validation at the Container start.

Checking Resources Configuration and Real Usage

The given subsection discusses the usage of general memory resources. The utility and the procedures explained in this subsection will help you answer the following questions:

- how many such Containers this system can run;
- what share of the total resources of the system this Container may consume and what share it consumes currently;
- whether this Container significantly under-use the resources allowed for it in its resource configuration;
- whether this Container uses significantly more than what was planned according to its resource configuration (because the system currently has spare resources and allows this Container to use them).

Information about inspecting the usage of individual resources is provided in the Inspecting Resource Control Settings subsection (on page 45).

Checking Resource Usage of Single Container

The information about resource usage by one Container is provided by the `vzcalc(8)` utility The example of its output is shown below:

```
# vzcalc 101
Resource    Current(%)  Promised(%)  Max(%)
Memory      0.60        5.25         6.98
```

In the current Virtuozzo version, this utility shows information about memory usage only In the future Virtuozzo versions, it will provide the information about other resources (such as CPU) as well.

In this example, Container 101 is reported to consume 0.60% of the memory resources currently. According to its configuration, it is promised to be able to use at least 5.25% of the memory resources and is limited by 6.98% of memory.

At the moment of the `vzcalc(8)` execution in the example, the Web server running in Container 101 was not accessed. It explains the fact that the Container used less than it was promised (0.60% against 5.25%).

Containers are allowed to consume more than they have been promised if the system has spare resources. The column labelled "Max" specifies the limitation on the resource consumption even if there are spare resources.

The number of Containers a system can run is determined by the configuration of the hardware, the configuration of the Containers, and the load (such as Web page hit rate) each Container handles. The `vzcalc(8)` utility helps understand how many Containers similar to the given one the system can run.

The numbers in the example show that

- the system is capable of running up to 19 such Containers (the number 19 is computed as $\frac{100\%}{5.25\%}$) without overcommitment;
- but with the current (zero) load of the Container it is possible to squeeze up to 166 such Containers on the system (the number 166 is computed as $\frac{100\%}{0.60\%}$). However, squeezing 166 such environments will lead to application failures when the load starts.

Note that the number of Containers a system can run may significantly vary, depending on the Container configuration and the applications inside it. Appendix B shows configurations that allow to run from 1 to 400 Containers on a server with 2GB of RAM. Servers with large memory and several CPUs can run up to 2,500 Containers with light Web servers.

The `-v` option of `vzcalc(8)` provides more information about the usage and limits of different types of resources, as discussed in the Advanced Utilities for Checking Resource Usage and Settings subsection (on page 61).

Note: The types are "low" memory, total RAM, memory plus swap space, allocation limits, number of processes.

Running the `vzcalc(8)` utility is especially recommended after the creation of a new resource configuration or starting to provide new application. In these cases `vzcalc(8)` should be run when the Container and all its applications are started to get information about the real usage of resources (the Current column).

Utilities for Checking Resource Configuration and Usage

A complete list of utilities for checking resource configuration and usage is presented in the table below. These utilities are discussed in the Advanced Utilities for Checking Resource Usage and Settings subsection (on page 61).

Name	Description
`vzcalc`	Shows the amount of resources (in percent) configured for and really used by one Container; with `-v` - for each resource type.
`vzmemcheck`	Shows the amount of memory resources (in percent and absolute values) for each resource type for the whole system; with `-v` - for each Container individually.
`vzcheckovr`	Checks if the system is overcommitted (i.e. if the Containers are promised more resources than the system) and compares the commitment level (the ratio between promised and existing resources) with the values configured as alert levels by the Virtuozzo administrator.
`vzcpucheck`	Shows CPU power promised to the Container and available on the server (please see Parallels Virtuozzo Containers User's Guide).
`vznetstat`	Shows the network resource usage statistics (please see Parallels Virtuozzo Containers User's Guide).

Summary of Configuration Creation Methods

Summarizing, the methods of the configuration creation of resource management parameters are the following:

- specifying the configuration sample by the `--config` option at the moment of the Container creation;
- scaling some other configuration by `vzcfgscale(8)`;
- copying configuration by `vzcfgscale(8)` with the factor equal to 1;
- copying configuration by `cp(l)`;
- using `vzsplit(8)`.

Specifying the configuration sample at the Container creation time and using `vzcfgscale(8)` are the preferable methods.

Resource management parameters can also be configured using Parallels Management Console (see Parallels Management Console Online Help).

Adjusting Individual Resource Management Parameters

Settings of resource management parameters of a Container can be adjusted by experienced Virtuozzo administrators for each parameter individually. Constraints and dependencies (listed in the Checking Configuration Consistency of Single Container subsection (on page 53)) need to be taken into account for such adjustments. After adjustments are made, the resulting configuration should be validated as discussed above.

For Virtuozzo administrators not very familiar with resource management parameter interdependencies and the ways the resource parameter settings affect the applications in the Container, the preferred way to change the configuration is using the `vzcfgscale(8)` utility (please see the Scaling Configuration subsection (on page 17)).

Adjustment of the settings of individual parameters is performed by the `vzctl(8)` utility as follows:

```
# vzctl set  CT_ID --param_name value –save
```

where *`CT_ID`* is the numeric identifier of the Container, *`param_name`* is the name of the parameter (a list of available parameters is given in the Resource Management Parameters Overview section (on page 13)), and *`value`* is the setting (a number or a pair of the barrier and the limit; see the `vzctl(8)` manual pages).

Controlling CPU Time Consumption

In addition to memory and operating system resources, Virtuozzo also controls the consumption of CPU time by Containers. This chapter briefly describes the features of this control.

Similarly to other components of the Virtuozzo resource control, CPU time consumption control has 2 purposes:

- prevention of denial-of-service attacks and making sure that each Container gets its fair share of CPU time, and
- allowing administrators to differentiate Container service levels, specifying how much CPU time each Container can get.

Types of Virtuozzo CPU Time Consumption Control

Virtuozzo provides 3 types of CPU time consumption control. It allows you:

- to guarantee certain CPU power for each Container;
- to specify the relative quality of service for Containers, i.e. to specify that one Container should be provided more CPU time in the given amount of times than to another Container, and
- to specify the absolute limit on the percentage of CPU time the Container gets.

Guaranteed CPU power. Guaranteed CPU power for each Container is assigned using the `cpuunits` parameter. Each Container is guaranteed to always be able to use the assigned power if the system is configured without overcommitment, i.e. if the CPU power assigned to all Containers and the host system processes does not exceed the power of the server.

How the guarantee works can be understood from the following example.

Let's consider the case when the sum of all assigned CPU power is equal to the power of the server, i.e. the server is fully "loaded". In this case the CPU power of each Container is exactly the share of CPU time that this Container will get if all Containers request CPU at some moment. If some Containers do not need CPU, other Containers will get the bigger CPU share than the power assigned to them. It means that the assigned CPU power is the guaranteed CPU time, and this guarantee does not depend on what server hosts the Container and the hardware of this server. If the sum of all assigned CPU power is less than the power of the server, Containers may get more than their CPU guarantee. So, the CPU guarantee is maintained if the server is under loaded or fully loaded. Certainly, if the server is overloaded, CPU guarantees can't be maintained.

Relative CPU power. The CPU power assigned to the Container using the `cpuunits` parameter also defines the relative amount of CPU time available to Containers. When 2 Containers have running processes, the one with bigger CPU power assignment will get more CPU time, pro rata to their CPU power.

For example, if the power of the system is 100, 000 and there are only 2 Containers requiring CPU at the moment with CPU power of 2, 000 and 6, 000, then the first will get 25% of CPU time ($\frac{2000}{2000+6000}$ x 100%) and the second one will get 75% ($\frac{6000}{2000+6000}$ x 100%).

Note that the shares of CPU time each Container gets in the example above is considerably bigger than the guaranteed shares. The first Container in the example was guaranteed 2% of CPU time ($\frac{2000}{100000}$ x 100%) of the system and the second one 6%. It is common that Containers are guaranteed considerably smaller share of CPU time than what they are able to get when they need CPU. Containers running on the same system often need CPU at different moments. So, they can do their job and have reasonable performance even if their guarantees are small.

Limiting CPU time consumption. In some cases it might be desirable to limit CPU time consumption by a Container even when the system has spare CPU time. Such restrictions do not improve security (because of guarantees and relative CPU power rules described above), but may be used for reduction of the quality of service for some Containers and for reduction of variation of peak performance when a Container is moved between systems with different CPU power.

CPU time consumption limit is specified in percents of 1 CPU of the system. In systems with more than 1 CPU limits of 100% and more may be used. CPU time consumption limit can also be not set, in which case only CPU power setting will be taken into account for giving CPU time to the Container.

Scheduling of individual processes inside Containers. Inside Containers, processes are scheduled by the standard Linux scheduler in a regular way, i.e. they are scheduled according to their priority (see `nice(1)`).

Note: It is possible to allow administrators of Containers to build scheduling hierarchies inside the Containers and use weight-based scheduler inside, but this setup is not the regular one.

Please keep in mind that processes with the lowest priority (+19) should not be considered as "background" ones. Those processes will get the smallest share of CPU time in the Container where they are executed, but may consume considerable amount of CPU time in comparison with processes in other Containers with lower CPU power setting. The same also applies to "reniced" processes in the host system.

Configuring and Inspecting CPU Time Consumption Control Settings

Configuring CPU Time Consumption Control Parameters

The guaranteed CPU power is assigned with the `--cpuunits` option of the `vzctl(8) set` command

```
# vzctl set CT_ID --cpuunits number --save
```

where number is the desired CPU power. This CPU power is measured in certain units which are hardware-independent.

Normally, the sum of CPU power assigned to all Containers and the host system shouldn't exceed the CPU power of the server (which can be viewed by the `vzcpucheck(8)` utility). A server with one 1 GHz PHI CPU has CPU power of about 50, 000 units. The power of a server with multiple CPUs is the sum of the powers of each CPU.

The limit on the CPU time consumption is set with the `--cpulimit` option of the `vzctl(8) set` command

```
# vzctl set CT_ID --cpulimit number --save
```

where *number* is the desired limit on CPU time consumption in percents of 1 CPU of the system. On systems with more than 1 CPU limits of more than 100% are allowed.

Assigning CPU Power to Host System Processes

Host system processes (such as Secure Shell daemon (`sshd`), system logging daemon (`syslogd`) and similar) also need CPU time. The share of CPU time for the host system processes can be changed on the fly by executing the following command:

```
# vzctl set 0 --cpuunits number
```

where *`number`* is the desired number of CPU units.

The host system CPU share is permanently stored in the `/etc/sysconfig/vz` configuration file. Detailed information on this file is provided in Parallels Virtuozzo Containers Reference Guide.

Note that the `--save` option doesn't save the `cpuunits` parameter assignment for the host system, unlike the assignments for Containers.

Checking Settings

CPU power settings can be checked by the `vzcpucheck(8)` utility as shown below:

```
# vzcpucheck
Current CPU utilization: 112681
Power of the node: 112721
```

Without additional options, it shows the total CPU power (in units) assigned to currently running Containers and the host system processes in the `Current CPU utilization` line and the CPU power of the server in the `Power of the node` line. If the server has more than 1 CPU and SMP kernel is booted, the power of all CPUs will be included in that line.

If the total CPU power assigned to running Containers and the host system processes exceeds the power of the server, a warning is shown as illustrated below:

```
# vzctl start 100
# vzcpucheck
Current CPU utilization: 113681
Power of the node: 112721
Warning: Hardware Node is overcommitted
```

Invoked with the `-v` option, `vzcpucheck(8)` shows also CPU power assignment of each Container:

```
# vzcpucheck -v
ctid  units
-----------
0     5681
100   1000
1001  4000
1003  4000
1004  4000
...
Current CPU utilization: 113681
Power of the node: 112721
Warning: Hardware Node is overcommitted
```

The information about CPU time consumption settings can also be obtained from the Container configuration file (detailed information on this file is provided in Parallels Virtuozzo Containers Reference Guide) and from the `/proc/fairsched` file. Note that the format of the `/proc/fairsched` file is not a part of the Virtuozzo API and may be changed in future Virtuozzo versions.

CHAPTER 3

Advanced Management of System Resources

In This Chapter

System Resource Control Parameters in Detail 29
Verifying and Troubleshooting Resource Control Configuration..... 47

System Resource Control Parameters in Detail

The UBC parameters can be subdivided into the following categories: primary, secondary, and auxiliary parameters. The primary parameters are the start point for creating a Container configuration from scratch. The secondary parameters are dependent on the primary ones and are calculated from them according to a set of constraints. The auxiliary parameters help improve fault isolation among applications in one and the same Container and the way applications handle errors and consume resources. They also help enforce administrative policies on Containers by limiting the resources required by an application and preventing the application to run in the Container.

Listed below are all the system resource control parameters. The parameters starting with "num" are measured in integers. The parameters ending in "buf" or "size" are measured in bytes. The parameters containing "pages" in their names are measured in 4096-byte pages. The File column indicates that all the system parameters are defined in the corresponding Container configuration files (V).

Name	Description
Primary parameters	
`numproc`	The maximal number of processes and threads the Container may create.
`avnumproc`	The average number of processes and threads.
`numtcpsock`	The number of TCP sockets (PF_INET family, SOCK_STREAM type). This parameter limits the number of TCP connections and, thus, the number of clients the server application can handle in parallel.
`numothersock`	The number of sockets other than TCP ones. Local (UNIX-domain) sockets are used for communications inside the system. UDP sockets are used, for example, for Domain Name Service (DNS) queries. UDP and other sockets may also be used in some very specialized applications (SNMP agents and others).

`vmguarpages` — The memory allocation guarantee, in pages (one page is 4 Kb). Container applications are guaranteed to be able to allocate additional memory so long as the amount of memory accounted as `privvmpages` (see the auxiliary parameters) does not exceed the configured barrier of the `vmguarpages` parameter. Above the barrier, additional memory allocation is not guaranteed and may fail in case of overall memory shortage.

Secondary parameters

`kmemsize` — The size of unswappable kernel memory allocated for the internal kernel structures for the processes of a particular Container.

`tcpsndbuf` — The total size of send buffers for TCP sockets, i.e. the amount of kernel memory allocated for the data sent from an application to a TCP socket, but not acknowledged by the remote side yet.

`tcprcvbuf` — The total size of receive buffers for TCP sockets, i.e. the amount of kernel memory allocated for the data received from the remote side, but not read by the local application yet.

`othersockbuf` — The total size of UNIX-domain socket buffers, UDP, and other datagram protocol send buffers.

`dgramrcvbuf` — The total size of receive buffers of UDP and other datagram protocols.

`oomguarpages` — The out-of-memory guarantee, in pages (one page is 4 Kb). Any Container process will not be killed even in case of heavy memory shortage if the current memory consumption (including both physical memory and swap) does not reach the `oomguarpages` barrier.

`privvmpages` — The size of private (or potentially private) memory allocated by an application. The memory that is always shared among different applications is not included in this resource parameter.

Auxiliary parameters

`lockedpages` — The memory not allowed to be swapped out (locked with the `mlock()` system call), in pages.

`shmpages` — The total size of shared memory (including IPC, shared anonymous mappings and `tmpfs` objects) allocated by the processes of a particular Container, in pages.

`physpages` — The total size of RAM used by the Container processes. This is an accounting-only parameter currently. It shows the usage of RAM by the Container. For the memory pages used by several different Containers (mappings of shared libraries, for example), only the corresponding fraction of a page is charged to each Container. The sum of the `physpages` usage for all Containers corresponds to the total number of pages used in the system by all the accounted users.

`numfile` — The number of files opened by all Container processes.

`numflock` — The number of file locks created by all Container processes.

`numpty` — The number of pseudo-terminals, such as an `ssh` session, the `screen` or `xterm` applications, etc.

`numsiginfo` — The number of `siginfo` structures (essentially, this parameter limits the size of the signal delivery queue).

`dcachesize` — The total size of `dentry` and `inode` structures locked in the memory.

`numiptent` — The number of IP packet filtering entries.

More detailed description of the parameters and the resource control mechanisms governed by these parameters is provided in the following subsections.

Overview

All resource control parameters have a number of common properties and a number of differences:

1. Most parameters provide both accounting of some system resource and allow controlling its consumption. The exceptions are `physpages` (accounting only) and `vmguarpages` (no accounting, control only) explained below.
2. Each parameter has 2 configuration variables called "barrier" and "limit" (please see the Controlling Resource Information in proc subsection (on page 46)).

   Although both 2 variables are changeable, for some parameters only one or none of the variables may be effectively used for the resource control. For example, `physpages` is an accounting-only parameter and its both configuration variables are not effectively used in the current Virtuozzo version. The description of each parameter explains the meaning of the barrier and the limit and to what value they should be set if they are not effectively used.

   In general, for all parameters the barrier should not be greater than the limit.
3. As discussed in the Resource Control Principles subsection (on page 11), the parameters control how the resources are distributed between Containers in terms of:
   - limits, i.e. upper boundaries on what this Container can consume, and
   - guarantees, i.e. mechanisms ensuring that this Container can get the assigned "resources" regardless of the activity and the amount of resources required by other Containers.

   The parameters containing `guar` in their names, e.g. `vmguarpages` and `oomguarpages`, are Container guarantees. They guarantee availability of resources and certain service level. However, these parameters do not impose usage restrictions. These guarantees are discussed in more detail in the paragraphs describing the corresponding parameters.

   The limit of `vmguarpages` and `oomguarpages` should be set to the maximal value (2,147,483,647 on 32-bit Intel-family processors).
4. For some resource limiting parameters (such as `kmemsize`) both the barrier and limit settings are effectively used. If the resource usage exceeds the barrier but doesn't exceed the limit, vital operations (such as process stack expansion) are still allowed to allocate new resources, and other ones are not allowed. A gap between the barrier and the limit gives applications better chances to handle resource shortage gracefully.

   For other resource limiting parameters (such as `numproc`) the barrier and the limit should be set to the same value.

5 Each parameter has "natural units of measurement" - the units of measurement of values shown via the `/proc/user_beancounters` interface (please see the Controlling Resource Information in proc subsection (on page 46)) and accepted by `vzctl(8)`. Values related to parameters with names starting with num are measured in pieces. Values related to parameters with names ending with pages are measured in memory pages (4KB on 32-bit Intel-family processors). The remaining values (parameters ending with `size` and `buf`) are measured in bytes.

The discussed above properties of the parameters are summarized in table below:

Name	Type	Account	Barrier	Limit	Units
Primary parameters					
`numproc`	limiting	yes	no	yes	pcs
`numtcpsock`	limiting	yes	no	yes	pcs
`numothersock`	limiting	yes	no	yes	pcs
`vmguarpages`	guarantee	no	guarantee	no	pages
Secondary parameters					
`kmemsize`	limiting	yes	yes	yes	bytes
`tcpsndbuf`	limiting	yes	yes	yes	bytes
`tcprcvbuf`	limiting	yes	yes	yes	bytes
`othersockbuf`	limiting	yes	yes	yes	bytes
`dgramrcvbuf`	limiting	yes	yes	yes	bytes
`oomguarpages`	guarantee	yes	guarantee	no	pages
`privvmpages`	limiting	yes	yes	yes	pages
Auxiliary parameters					
`lockedpages`	limiting	yes	yes	yes	pages
`shmpages`	limiting	yes	no	yes	pages
`physpages`	accounting	yes	no	no	pages
`numfile`	limiting	yes	no	yes	pcs
`numflock`	limiting	yes	yes	yes	pcs
`numpty`	limiting	yes	no	yes	pcs
`numsiginfo`	limiting	yes	no	yes	pcs
`dcachesize`	limiting	yes	yes	yes	bytes
`numiptent`	limiting	yes	no	yes	pcs

Primary Parameters

The most important parameters determining the resources available to the Container are explained below. The meaning of the parameters is illustrated assuming that the Container runs some network server applications.

The description of each parameter is followed by the information about the recommended gap between the barrier and the limit and then by the discussion about the limitations on the totals of the settings of this parameter for all Containers in a system.

numproc

`numproc`: the maximum number of processes and kernel-level threads allowed for this Container.

Many server applications (like Apache Web server, FTP and mail servers) spawn a process to handle each client, so the limit on the number of processes defines how many clients the application will be able to handle in parallel. However, the number of processes doesn't limit how "heavy" the application is and whether the server will be able to serve heavy requests from clients.

Configuring resource control system, it is important to estimate both the maximum number of processes and the average number of processes (referred to as `avnumproc` later). Other (dependent) resource control parameters may depend both on the limit and the average number (please see the Checking Configuration Consistency of Single Container subsection (on page 53)).

The barrier of the `numproc` parameter doesn't provide additional control and should be set equal to the limit.

There is a theoretical restriction on the total number of processes in the system. Currently, it is about 32,000. Usually, the kernel imposes a lower limit on the number of processes. This limit can be viewed and tuned through the `/proc/sys/kernel/threads-max` entry, but it should not be set to a value greater than the theoretical limit 32,000.

In practice, more than about 16,000 processes start to cause poor responsiveness of the system worsening when the number grows. Total number of processes exceeding 32,000 is very likely to cause hang of the system. Also, each process consumes some memory, and the available total and "low" (please see the Low Memory subsection (on page 56)) limit the number of processes to lower values. With typical processes, it is normal to be able to run only up to 8,000 processes in a system.

numtcpsock

`numtcpsock`: the maximum number of TCP sockets. This parameter limits the number of TCP connections and, thus, the number of clients the server application can handle in parallel.

The barrier of this parameter should be set equal to the limit.

There are no theoretical restrictions directly limiting the total number of TCP sockets in Virtuozzo system. However, the number of sockets needs to be controlled because each socket needs certain amount of memory for receive and transmit buffers (see the descriptions of `tcpsndbuf` and `tcprcvbuf` below), and the memory is a limited resource. So, the number of sockets is limited indirectly by memory constraints.

numothersock

`numothersock`: the maximum number of non-TCP sockets (local sockets, UDP, and other types of sockets).

Local (UNIX-domain) sockets are used for communications inside the system. Multi-tier applications (for example, a Web application with a database server as a back-end) may need one or multiple local sockets to serve each client. Straightforward applications (for example, most mail servers) do not use local sockets.

UDP sockets are used for Domain Name Service (DNS) queries, but the number of such sockets opened simultaneously is low. UDP and other sockets may also be used in some very special applications (SNMP agents and others).

The barrier of this parameter should be set equal to the limit.

Similarly to TCP sockets, there are no theoretical restrictions directly limiting the total number of non-TCP sockets in Virtuozzo systems. The number of non-TCP sockets needs to be controlled because each socket needs certain amount of memory for its buffers, and the memory is a limited resource.

vmguarpages

`vmguarpages`: memory allocation guarantee.

This parameter controls how much memory is available to the Container (i.e. how much memory its applications can allocate by `malloc(3)` or other standard Linux memory allocation mechanisms). The more clients are served or the more "heavy" the application is, the more memory the Container needs.

The amount of memory that Container's applications are guaranteed to be able to allocate is specified as the barrier of the `vmguarpages` parameter. The `vmguarpages` parameter does not have its own accounting. The current amount of allocated memory space is accounted into another parameter - `privvmpages` described in the Secondary Parameters section (on page 36). The meaning of the limit of the `vmguarpages` parameter is unspecified in the current version and should be set to the maximal allowed value (2,147,483,647 on 32-bit Intel-family processors).

Memory allocation requests made by applications are granted or denied basing on the following rules. If the current amount of the allocated memory does not exceed the guaranteed amount (the barrier of `vmguarpages`), the memory allocation requests always succeed. If the current amount exceeds the guarantee but stays below the barrier of the `privvmpages` parameter, allocations may or may not succeed, depending on the total amount of available memory in the system. Starting from the barrier of the `privvmpages` parameter, normal priority allocations and, starting from the limit, all memory allocations made by the applications fail.

As it can be seen, these rules involve 2 resource control parameters. The memory allocation guarantee (`vmguarpages`) is the primary tool for controlling the memory available to Containers. It allows administrators to provide Service Level Agreements — agreements guaranteeing certain quality of service, certain amount of resources and general availability of the service. `privvmpages` is a helper parameter, limiting memory allocations even when the system has spare resources.

The unit of measurement of `vmguarpages` values is memory pages (4KB on 32-bit Intel-family processors).

The total memory allocation guarantees given to Containers are limited by the physical resources of the server - the size of RAM and the swap space - as discussed in the Allocated Memory subsection (on page 59).

Secondary Parameters

Dependant (secondary) parameters are directly connected to the primary ones and can't be configured arbitrarily.

Units. The `oomguarpages` and `privvmpages` values are measured in memory pages (4KB on 32-bit Intel-family processors). For the remaining secondary parameters, the values are measured in bytes.

System-wide limits. All secondary parameters are related to memory. The sum of the limits of memory-related parameters of all Containers running in a system must not exceed the physical resources of the server. The restrictions on the configuration of memory-related parameters are listed in the Checking Usage and Distribution of System Resources and Validating Settings of All Containers subsection (on page 55). Those restrictions are very important since their violation may allow any Container cause the whole system to hang.

kmemsize

`kmemsize`: the size of unswappable memory allocated by the operating system kernel. It includes all the kernel internal data structures associated with the Container processes, except the network buffers discussed below. These data structures reside in the first gigabyte of the server RAM, the so called "low" memory (please see the Low Memory subsection (on page 56)).

The `kmemsize` parameter is related to the number of processes (`numproc`). Each process consumes certain amount of kernel memory - 24 kilobytes at minimum, 30-60 KB typically. Very large processes may consume more than that.

It is important to have a certain safety gap between the barrier and the limit of the `kmemsize` parameter (for instance, 10%, as in examples in Appendix B). Equal barrier and limit of the `kmemsize` parameter may lead to the situation where the kernel will need to kill Container's applications to keep the `kmemsize` usage under the limit.

`kmemsize` limits can't be set arbitrarily high. The total amount of memory accounted into the `kmemsize` parameter plus the socket buffer space (discussed below) is limited by the hardware resources of the system. This total limit is discussed in the Low Memory subsection (on page 56).

tcpsndbuf

`tcpsndbuf`: the total size of buffers used to send data over TCP network connections. These socket buffers reside in "low" memory (please see the Low Memory subsection (on page 56)).

The `tcpsndbuf` parameter settings depend on the number of TCP sockets (the `numtcpsock` parameter) and should allow for some minimal amount of socket buffer memory for each socket (please see the Checking Configuration Consistency of Single Container subsection (on page 53) for the full list of dependencies):

$$\texttt{tcpsndbuf}_{\texttt{lim}} - \texttt{tcpsndbuf}_{\texttt{bar}} \geq 2.5\texttt{KB} \cdot \texttt{numtcpsock}$$

If this restriction is not satisfied, some network connections may silently hang being unable to transmit data.

Setting high values for the `tcpsndbuf` parameter may, but doesn't necessarily, increase the performance of network communications. Note that, unlike most other parameters, hitting the `tcpsndbuf` limits and failed socket buffer allocations do not have strong negative effect on the applications, but just reduce the performance of network communications.

The `tcpsndbuf` limits can't be set arbitrarily high. The total size of send buffers of TCP sockets consumable by all Containers plus the sizes of socket buffers of other types plus the size of the memory accounted into the kmemsize parameter is limited by the hardware resources of the system. This total limit is discussed in the Low Memory subsection (on page 56).

tcprcvbuf

`tcprcvbuf`: the total size of buffers used to temporary store the data coming from TCP network connections. These socket buffers also reside in 'low'' memory (please see the Low Memory subsection (on page 56)).

The `tcprcvbuf` parameter settings depend on the number of TCP sockets (the `numtcpsock` parameter) and should allow for some minimal amount of socket buffer memory for each socket (please see the Checking Configuration Consistency of Single Container subsection (on page 53) for the complete list of dependencies):

$$\texttt{tcprcvbuf}_{\texttt{lim}} - \texttt{tcprcvbuf}_{\texttt{bar}} \geq 2.5\texttt{KB} \cdot \texttt{numtcpsock}.$$

If this restriction is not satisfied, some network connections may stall, being unable to receive data, and will be terminated after a couple of minutes.

Similarly to `tcpsndbuf`, setting high values for the `tcprcvbuf` parameter may, but doesn't necessarily, increase the performance of network communications. Hitting the `tcprcvbuf` limits and temporary failures of socket buffer allocations do not have strong negative effect on the applications, but just reduce the performance of network communications. However, staying above the barrier of the `tcprcvbuf` parameter for a long time is less harmless than for `tcpsndbuf`. Long periods of exceeding the barrier may cause termination of some connections. These subtle differences between send and receive buffers are summarized in the table below:

	tcpsndbuf	tcprcvbuf
The actual size of the buffers is determined by	the link speed of the client receiving data from an application in the Container	the responsiveness of the application in the Container.
The consequences of the insufficient gap between the barrier and the limit are	severe, network connections and the application may silently hang	not so severe, network connections may become slow or, if too much data stays in buffers for too long time, one or more network connections can terminate by timeout.

The consequences of the insufficient value of the barrier	not significant, network connections may become slow	more significant, network connections may become slow or, if too much data stays in buffers for too long time, one or more network connections can terminate by timeout.

The `tcprcvbuf` limits can't be set arbitrarily high. The total size of receive buffers of TCP sockets consumable by all Containers plus the sizes of socket buffers of other types plus the size of the memory accounted into the kmemsize parameter is limited by the hardware resources of the system. This total limit is discussed in the Low Memory subsection (on page 56).

othersockbuf

`othersockbuf`: the total size of buffers used by local (UNIX-domain) connections between processes inside the system (such as connections to a local database server) and send buffers of UDP and other datagram protocols.

The `othersockbuf` parameter settings depend on the number of non-TCP sockets (the `numothersock` parameter) and should allow for some minimal amount of socket buffer memory for each socket

$$\texttt{othersockbuf}_{\texttt{lim}} - \texttt{othersockbuf}_{\texttt{bar}} \geq 2.5\text{KB} \cdot \texttt{numothersock}$$

(see the Checking Configuration Consistency of Single Container subsection (on page 53) for the complete list of dependencies).

Increased limit for `othersockbuf` is necessary for high performance of communications through local (UNIX-domain) sockets. However, similarly to `tcpsndbuf`, hitting the `othersockbuf` barrier affects the communication performance only and does not affect the functionality.

The `othersockbuf` limits can't be set arbitrarily high. The total size of UNIX-domain and datagram socket buffers consumable by all Containers plus the sizes of socket buffers of other types plus the size of the memory accounted into the `kmemsize` parameter is limited by the hardware resources of the system. This total limit is discussed in the Low Memory subsection (on page 56).

dgramrcvbuf

`dgramrcvbuf`: the total size of buffers used to temporary store the incoming packets of UDP and other datagram protocols. The `dgramrcvbuf` parameter settings depend on the number of non-TCP sockets (the `numothersock` parameter).

The `dgramrcvbuf` limits usually don't need to be high. Only if the Container needs to send and receive very large datagrams, the barriers for both `othersockbuf` and `dgramrcvbuf` parameters should be raised. Hitting `dgramrcvbuf` means that some datagrams are dropped, which may or may not be important for application functionality. UDP is a protocol with not guaranteed delivery, so even if the buffer size limits permit, the datagrams may be as well dropped later on any stage of the processing, and applications should be able to handle packet drops.

Unlike other socket buffer parameters, for `dgramrcvbuf` the barrier should be set equal to the limit.

The `dgramrcvbuf` limits can't be set arbitrarily high. The total size of receive buffers of datagram sockets consumable by all Containers plus the sizes of socket buffers of other types plus the size of the memory accounted into the `kmemsize` parameter is limited by the hardware resources of the system. This total limit is discussed in the Low Memory subsection (on page 56).

oomguarpages

oomguarpages: the guaranteed amount of memory for the case the memory is "over-booked" (out-of-memory kill guarantee).

The oomguarpages parameter is related to vmguarpages. If applications start to consume more memory than the server has, the system faces an out-of-memory condition. In this case the operating system will start to kill Container's processes to free some memory and prevent the total death of the system. Although it happens very rarely in typical system loads, killing processes in out-of-memory situations is a normal reaction of the system, and it is built into every Linux kernel.

Note: Out-of-memory situations may happen because of a number of reasons, mostly related to server overload or not safe resource configurations. One possible reason of out-of-memory situations is the excess of total vmguarpages guarantees the available physical resources. Another possible reason is high memory consumption by host system processes running outside of any Container. Also, the kernel might allow some Containers to allocate memory above their vmguarpages guarantees when the system had a lot of free memory, and later, when other Containers claim their guarantees, the system will experience the memory shortage.

Out-of-memory situation may also happen only in the "low" memory zone even if the system has enough memory in other zones. "Low" memory is discussed in the Low Memory subsection (on page 56).

The oomguarpages parameter accounts the total amount of memory and swap space used by the processes of a particular Container. The barrier of the oomguarpages parameter is the out-of-memory guarantee.

If the current usage of memory and swap space (the current value accounted into the oomguarpages parameter) plus the amount of used kernel memory (kmemsize) and socket buffers is below the oomguarpages barrier, processes in this Container are guaranteed not to be killed in out-of-memory situations. If the system is in the out-of-memory situation and there are several Containers with oomguarpages excess, applications in the Container with the biggest excess will be killed first.

Note that the oomguarpages parameter has a different meaning of the barrier than most other parameters. Not only the barrier is a guaranteed amount of memory rather than a limitation, the value of the barrier is compared with the sum of the current memory usage accounted into the oomguarpages parameter and the memory usage accounted into other parameters. The failcnt counter of the oomguarpages parameter increases when a process in this Container is killed because of an out-of-memory situation, but not when the barrier is reached.

If the administrator needs to make sure that some application won't be forcedly killed regardless of the application's behavior, setting the privvmpages limit to a value not greater than the oomguarpages guarantee significantly reduces the likelihood of the application being killed, and setting it to a half of the oomguarpages guarantee completely prevents it. Such configurations are not popular because they significantly reduce the utilization of the hardware.

The meaning of the limit of the oomguarpages parameter is unspecified in the current version.

The total out-of-memory guarantees given to the Containers should not exceed the physical capacity of the server, as discussed in the Memory and Swap Space subsection (on page 58). If guarantees are given for more than the system has, applications in Containers with a guaranteed level of service and system daemons (including `sshd`) may be killed in out-of-memory situations.

privvmpages

`privvmpages`: memory allocation limit.

The `privvmpages` parameter allows controlling the amount of memory allocated by applications.

The barrier and the limit of the `privvmpages` parameter control the upper boundary of the total size of allocated memory. Note that this upper boundary doesn't guarantee that the Container will be able to allocate that much memory, neither does it guarantee that other Containers will be able to allocate their fair share of memory.

The `privvmpages` parameter accounts allocated (but, possibly, not used yet) memory. The accounted value is an estimation how much memory will be really consumed when the Container applications start to use the allocated memory. Actually consumed at this moment memory is accounted into the `oomguarpages` parameter.

There should be a safety gap between the barrier and the limit for the `privvmpages` parameter to reduce the number of memory allocation failures that the application is unable to handle. This gap will be used for "high-priority" memory allocations, such as process stack expansion. Normal priority allocations will fail when the barrier is reached.

Since the memory accounted into the `privvmpages` parameter may not be actually used, the sum of current `privvmpages` values for all Containers may exceed the RAM and swap size of the server. However, systems with such excess may cause applications to work less reliably (see the Allocated Memory subsection (on page 59) for the information what excess may be considered safe).

The total `privvmpages` limits set for all Containers should reflect the physical resources of the server. Also, it is important not to allow any single Container to allocate a significant portion of all system RAM to avoid serious service level degradation for other environments. Both these configuration requirements are discussed in the Allocated Memory subsection (on page 59).

Auxiliary Parameters

Configuration of primary and secondary resource control parameters is important for security and stability of the whole system. Auxiliary parameters differ much from the primary and secondary parameters in this respect.

The main functions of auxiliary parameters are the following:

- These parameters help applications to handle resource consumption limitations.

  Without these auxiliary parameters, possible bugs in applications (such as forgetting to unlock locked files or forgetting to collect signals) will cause slowdown and, after some time, termination of the applications because of memory exhaustion. In presence of these parameters, applications will notice the problem (because, for example, attempts to create new file locks start to fail), show an appropriate message helping to understand the problem and either continue operations or exit gracefully.
- These parameters improve fault isolation between applications in the same Container. Failures or misbehavior of one application inside a Container is more likely to cause hitting a limit on some auxiliary parameter and normal termination of this misbehaving application, rather than abnormal termination of some other long-running application inside the same environment.
- These parameters may be used to impose some administrative limits on the Container (for example, to not allow the user to run database servers by setting low limits for the `shmpages` parameter, or limiting the number of simultaneous shell sessions through the `numpty` parameter).

So, auxiliary parameters play a role similar to the limitations configurable through the `setrlimit(2)` interface and `sysctl(8)` in standard Linux installations (please see the Other Existing Resource Control Mechanisms subsection (on page 11)).

Because of this helper role in resource control, system management software may show auxiliary parameters in advanced mode for experienced administrators only and hide them in "basic" management modes.

lockedpages

`lockedpages`: the process pages not allowed to be swapped out (pages locked by `mlock(2)`). The size of these pages is also accounted into `kmemsize`. The barrier may be set equal to the limit or may allow some gap between the barrier and the limit, depending on the nature of applications using memory locking features. Note that typical server applications like Web, FTP, mail servers do not use memory locking features. The configuration of this parameter doesn't affect security and stability of the whole system or isolation between Containers. Its configuration affects functionality and resource shortage reaction of applications in the given Container only.

shmpages

`shmpages`: the total size of shared memory (IPC, shared anonymous mappings and `tmpfs` objects). These pages are also accounted into `privvmpages`.

The barrier should be set equal to the limit. The configuration of this parameter doesn't affect security and stability of the whole system or isolation between Containers. Its configuration affects functionality and resource shortage reaction of applications in the given Container only.

physpages

`physpages`: the total number of RAM pages used by processes in this Container.

Memory pages used by several different Containers (mappings of shared libraries, for example) are accounted for each Container, but the memory usage of each of them is increased only by a fraction of a page, depending on the total number of environments using this memory page. This accounting method is considerably more practical than just summing the memory "usage" information reported by `ps(l)` or `top(l)`. Unlike other accounting methods, the sum of the `physpages` usage for all Containers yields to the total number of pages used in the system by all Containers.

`physpages` is an accounting-only parameter currently. In future Virtuozzo releases, this parameter will allow to provide guaranteed amount of application memory residing in RAM and not swappable. For compatibility with future versions, the barrier of this parameter should be set to 0 and the limit to the maximal allowed value (2,147,483,647 on 32-bit Intel-family processors).

numfile, numflock, numpty, and numsiginfo

`numfile`: the number of "files" in use including real files, sockets, and pipes. The barrier should be set equal to the limit. The configuration of this parameter doesn't affect security and stability of the whole system or isolation between Containers. Its configuration affects functionality and resource shortage reaction of applications in the given Container only.

`numflock`: the number of file locks. The configuration of this parameter should have a gap between the barrier and the limit (for instance, about 10%, as illustrated in Appendix B). Very high `numflock` limits and the big number of file locks in the system may cause certain (not fatal) slowdown of the whole system. So, the `numflock` limits should be reasonable and match to the real requirements of the applications.

`numpty`: the number of pseudo-terminals. This parameter is usually used to limit the number of simultaneous shell sessions. The barrier should be set equal to the limit. The configuration of this parameter doesn't affect security and stability of the whole system or isolation between Containers. Its configuration affects functionality and resource shortage reaction of applications in the given Container only. However, in Virtuozzo systems, the actual number of pseudo-terminals allowed for one Container is limited to 256 BSD-style pseudo-terminals plus 256 UNIX98-style terminals.

`numsiginfo`: the number of `siginfo` structures. The size of the structure is also accounted into the `kmemsize` parameter. The default installations of stand-alone Linux systems limit this number to 1024 for the whole system. In Virtuozzo installations, the `numsiginfo` limit applies to each Container individually.

The barrier should be set equal to the limit. Very high settings of the `numsiginfo` limit may reduce responsiveness of the system. It is unlikely that any Container will need the limit greater than the Linux default - 1024.

dcachesize

`dcachesize`: the total size of `dentry` and `inode` structures locked in memory.

The `dcachesize` parameter controls filesystem-related caches, such as directory entry (`dentry`) and `inode` caches. The value accounted into the `dcachesize` parameter is also included into `kmemsize`. `dcachesize` exists as a separate parameter to impose a limit allowing file operations to sense memory shortage and return an error to applications, protecting applications from memory shortages during critical operations that shouldn't fail.

The configuration of this parameter should have a gap between the barrier and the limit (about 10%, as illustrated in Appendix B). The configuration of this parameter doesn't affect security and stability of the whole system or isolation between Containers. Its configuration affects functionality and resource shortage reaction of applications in the given Container only.

numiptent

`numiptent`: number of `NETFILTER` (IP packet filtering) entries.

The barrier should be set equal to the limit. There is a restriction on the total number of IP packet filtering entries in the system. It depends on the amount of other allocations in so called the `vmalloc` memory area and constitutes about 250,000 entries. Violation of this restriction may cause failures of operations with IP packet filter tables (execution of `iptables(8)`) in any Container or the host system, or failures of Container starts. Also, high `numiptent` settings cause considerable slowdown of processing of network packets. It is not recommended to set the `numiptent` limit to more than 200-300.

Inspecting Resource Control Settings

Resource control settings for each Container are stored in its configuration file - `ve.conf(5)`. On the start of the Container, the settings are loaded from the configuration file and applied to the Container (The settings may also be loaded and applied on some other events such as execution of `vzctl exec`, so administrators shouldn't rely on the fact that the Container configuration file is read at the start only.). When a Container is running, its current resource control settings are also exposed via proc (5) filesystem together with resource accounting and statistics information.

The ways to inspect the resource control settings are:

- view the Container configuration file by any viewer, such as `more(l)` or `less(D)` - see the `ve.conf(5)` manual pages;
- view the `/proc/user_beancounters` file - see the next subsection;

Note: The current configuration shown via `/proc/user_beancounters` can theoretically be different from the configuration stored in the `ve.conf(5)` file. However, if adjustments to resource control settings of a running Container are made by invoking `vzctl set` with the `--save` flag (see the `vzctl(8)` manual page and Parallels Virtuozzo Containers Reference Guide, the contents of `/proc/user_beancounters` and `ve.conf(5)` will match.

- view the configuration by Parallels Management Console - see Parallels Management Console Online Help;
- request the resource control settings from Parallels Agent - see the Parallels Agent documentation.

Controlling Resource Information in proc

The `/proc/user_beancounters` file in the `proc(5)` filesystem shows the resource control information about running Containers. An example content of the `/proc/user_beancounters` file is shown below:

```
# cat /proc/user_beancounters
Version: 2.4
 uid resource     held    maxheld  barrier  limit      failcnt
101: kmemsize     275976  337392   1884160  2097152    0
     lockedpages  0       0        32       32         0
     privvmpages  2675    3264     25600    28160      0
     shmpages     0       0        512      512        0
     dummy        0       0        0        0          0
     numproc      8       10       100      100        0
     physpages    650     652      0        2147483647 0
     vmguarpages  0       0        1024     2147483647 0
     oomguarpages 650     652      0        2147483647 0
     numtcpsock   0       0        80       80         0
     numflock     0       1        100      110        0
     numpty       0       0        16       16         0
     numsiginfo   0       2        256      256        0
     tcpsndbuf    0       0        262144   262144     0
     tcprcvbuf    0       0        524288   524288     0
     othersockbuf 2224    3264     131072   131072     0
     dgramrcvbuf  0       0        131072   131072     0
     numothersock 1       5        30       30         0
     dcachesize   30277   35618    524288   557056     0
     numfile      71      80       1024     1024       0
     dummy        0       0        0        0          0
     dummy        0       0        0        0          0
     dummy        0       0        0        0          0
     numiptent    4       4        5        5          0
```

The `uid` field is the numeric identifier of the Container (the 'user" for the purpose of resource control).

For accountable parameters, the `held` field shows the current counter for the Container (resource "usage"), and the `maxheld` field shows the counter maximum for the last accounting period. The accounting period is the lifetime of the Container, unless a statistics collection agent is running. A statistics collection agent (such as Parallels Agent - see the Parallels Agent documentation) may set other accounting periods. The `failcnt` field shows the number of refused "resource allocations" for the whole lifetime of the Container (regardless whether Parallels Agent is running). The `failcnt` counter is increased only for accounting parameters (please see the Overview subsection (on page 31)) and has a special meaning for the `oomguarpages` parameter explained in the section describing this parameter.

Note: The lifetime of a Container is the period from the start of the Container till the time all processes in the Container exited and all resources used by these processes are freed. Usually the lifetime is just the time between the start and the stop of the Container.

The barrier and limit fields are resource control settings. For some parameters, only one of them may be used, for some parameters - both, as shown in the Overview subsection (on page 31). These fields may specify limits or guarantees, and the exact meaning of them is parameter-specific. The description of each parameter (see the Primary Parameters, Secondary Parameters, and Auxiliary Parameters subsections above) contains the information about the difference between the barrier and the limit of this parameter.

General notes about parameters and their barrier and limit settings are provided in the Overview subsection (on page 31). Parameters for which the limit can't be used have the maximal allowed value (2,147,483,647 on 32-bit Intel-family processors) in the `limit` field.

As also discussed in the Overview subsection (on page 31), all values related to parameters with names starting with `num` are measured in pieces. Values related to parameters with names ending with pages are measured in memory pages (4KB on 32-bit Intel-family processors). Values related to other parameters are measured in bytes.

The `dummy` entries are placeholders for obsolete or future resource control parameters and should be ignored.

Verifying and Troubleshooting Resource Control Configuration

The Configuring Resource Management Parameters for Containers section (on page 15) has discussed the basic procedures for the creation of resource control parameter configurations for Containers. Settings of resource control parameters for all Containers should be:

- functional, i.e. allow users and applications to work, not creating for them unnecessary obstacles; and at the same time
- safe, i.e. prevent denial-of-service attacks by resource overconsumption, provide fault and performance isolation between processes in different Containers, and make sure that Containers are able to get the configured share of the system resources.

Note: Fault isolation means that faults of applications in one Container do not affect other Containers. Performance isolation means that one Container can't cause significant performance degradation for others. Isolation and security topics are briefly discussed in the Resource Control and System Security subsection (on page 10).

Both of these points (i.e. functionality and safety) need to be verified by the administrator. This verification is the topic of this chapter.

After the configuration is created, administrators usually need to perform the following steps:

1. check that the applications can perform their tasks and work well with the given resource settings;
2. make sure that the configuration of the resource control parameters of the Containers is valid;
3. compare the power of the server and the total resources of the system with the resource control settings of the Containers running on it;
4. make sure that some periodic monitor of the health of the Container and the whole system is operating.

When the administrator deals with an application with not priorly known resource requirements, it often requires more than one iteration of adjusting the resource control parameters and checking how well applications work, before the optimal configuration of resource control parameters is found. If the system has a lot of spare resources, it is more easy to develop a fully functional configuration of the system, because the configuration doesn't need to be very optimized. If the system resources are just enough for the desired number of Containers, it may require several iterations of test runs and configuration adjustments to find a suitable configuration.

This chapter discusses 4 steps described above. Its material is intended for

- Virtuozzo administrators not using Parallels Management Console and skillful enough to perform application troubleshooting and
- developers of custom Virtuozzo management and monitoring software.

Checking Resource Control Settings for Applications

This section is focused on the procedures of verification that applications providing some networking services (Web, FTP, mail and similar servers) are able to function with the given resource control settings.

Checking that the server application is functional includes the tests that:

- it is able to start;
- it answers to requests;
- it works and shows reasonable performance under the desired load.

Checking Application Start

Most server applications print error messages on `stderr` if they are unable to start, as shown below. If the application starts at the start of the Container automatically, recent Red Hat Linux distributions store the messages from `stderr` in `/var/log/boot.log`:

```
# /etc/rc.d/init.d/httpd start
/etc/rc.d/init.d/httpd: fork: Cannot allocate memory

# /etc/rc.d/init.d/httpd start
/etc/rc.d/init.d/httpd: fork: Cannot allocate memory
/etc/rc.d/init.d/httpd: fork: Cannot allocate memory
Starting httpd: /etc/rc.d/init.d/httpd: fork: Cannot allocate memory
fork: Cannot allocate memory
                                                           [FAILED]

# /etc/rc.d/init.d/httpd start
Starting httpd: Ouch! ap_mm_create(1048576, "/var/run/httpd.mm.10483")
failed Error: MM: mm:core: failed to acquire shared memory segment
(No space left on device): OS: No such file or directory
                                                           [FAILED]
```

Successful start of the application may be verified by the `ps(1)` command, by a status option of the application control script (if it's provided), by verifying that the application listens on its TCP port by the `netstat(8)` command, as shown in the outputs below:

- The start has failed:

```
# /etc/rc.d/init.d/httpd start
Starting httpd:         [0K ]
# ps ax|grep httpd
10536 ttypO  R  0:00  grep httpd
# /etc/rc.d/init.d/httpd status
httpd dead but pid file exists
# netstat -ant|grep LISTEN
tcp  0  0  0.0.0.0:22  0.0.0.0:*  LISTEN
tcp  0  0  0.0.0.0:25  0.0.0.0:*  LISTEN
# tail /var/log/httpd/error_log
[Mon Jun 3 17::52:05 2002] [error] mod_ssl: Cannot allocate shared
memory:mm:core: failed to acquire shared memory segment (No space left
on device)
```

- The start is successful:

```
# /etc/rc.d/init.d/httpd start
Starting httpd          [OK ]
# ps ax|grep httpd
925  ?     S  0:00  /usr/sbin/httpd  -DHAVE_ACCESS  -DHAVE_PR0XY
928  ?     S  0:00  /usr/sbin/httpd  -DHAVE_ACCESS  -DHAVE_PR0XY
929  ?     S  0:00  /usr/sbin/httpd  -DHAVE_ACCESS  -DHAVE_PR0XY
930  ?     S  0:00  /usr/sbin/httpd  -DHAVE_ACCESS  -DHAVE_PR0XY
931  ?     S  0:00  /usr/sbin/httpd  -DHAVE_ACCESS  -DHAVE_PR0XY
932  ?     S  0:00  /usr/sbin/httpd  -DHAVE_ACCESS  -DHAVE_PR0XY
933  ?     S  0:00  /usr/sbin/httpd  -DHAVE_ACCESS  -DHAVE_PR0XY
934  ?     S  0:00  /usr/sbin/httpd  -DHAVE_ACCESS  -DHAVE_PR0XY
935  ?     S  0:00  /usr/sbin/httpd  -DHAVE_ACCESS  -DHAVE_PR0XY
937  ttypO S  0:00  grep httpd
# netstat -ant Igrep LISTEN
tcp   0     0 0.0.0.0:80      0.0.0.0:*   LISTEN
tcp   0     0 0.0.0.0:22      0.0.0.0:*   LISTEN
tcp   0     0 0.0.0.0:25      0.0.0.0:*   LISTEN
```

```
tcp   0     0 0.0.0.0:443     0.0.0.0:*   LISTEN
```

If the application fails to start and its `stderr` doesn't provide anything useful, more information may appear in application log files (such as `/var/log/httpd/error_log`, as shown above) or system log files (such as `/var/log/messages` and `/var/log/daemon`). In some circumstances, there may be kernel messages (available through the `dmesg(8)` command, in `/var/log/kern` or `/var/log/messages`) related to the application start failure.

Applications may fail to start not only because of insufficient resources, but by some other reasons. If application's messages do not provide enough information to understand why its start failed, the most reliable way to get the information is to use `strace(l)`. In the example below, the reason of the failed `httpd` start appears to be unsuccessful attempt to allocate an IPC shared memory segment (the `shmget` call) of the size of 532480:

Note: It has been observed that `strace` invoked from under `vzctl enter` works unreliably. To use `strace`, it is better to log into the Container through `ssh` or run `strace` with the `-p` option in the host system.

```
# strace -f -s200 -q -o log /etc/rc.d/init.d/httpd start
Starting httpd:                        [FAILED]
# tail -15  log
close(4)                               = 0
munmap(0x40018000, 4096)               = 0
brk(0x81b2000)                         = 0x81b2000
brk(0x81b3000)                         = 0x81b3000
brk(0x81b4000)                         = 0x81b4000
brk(0x81b6000)                         = 0x81b6000
shmget (IPC_PRIVATE,    532480, IPC.      .CREAT10x18010600) = -1
EN0SPC (No space  left on device)
close(O)                               = 0
unlink("/var/cache/ 'ssl_gcache_data. sem ") = -1 EN0ENT (No such file
or directory)
time(NULL)                             = 1023175149
getpidQ                                = 28712
time(NULL)                             = 1023175149
write(15, "[Tue Jun 4 11:19:09 2002]      [error] mod_ssl: Cannot
allocate shared
memory: mm:core: failed to ;  acquire     shared memory segment   (No
space left  on
device)\n", 150) = 150
munmap(0x40017000, 4096)               = 0
_exit(l)                               = ?
```

A good hint to whether the application failed to start because of reaching some resource limit is provided by the `failcnt` counter shown in the `/proc/ user_beancounters` file (see the Controlling Resource Information in proc subsection) (on page 46). If any of the `failcnt` counters increased during the application start resulted in a failure, the limit on corresponding resource control parameter may be related to that failure. If the `failcnt` counters do not increase (for example, just remain zero), resource control limitations are unlikely to be related to the application start failures.

Troubleshooting Application Failures Related to Resource Limitations

Application messages often provide enough information to understand the reasons of the application start failure. Messages about failed fork are related to reaching the limit on `numproc` or its derived parameter `kmemsize`.

Messages about shared memory may be related to `shmpages` or `privvmpages` limits. Messages about sockets may be related to `numtcpsock`, `numothersock`, `dcachesize` or `kmemsize` parameters.

The other way to find which resource control limits are exceeded on application's start is checking the `/proc/user_beancounters` file. The parameter with an increased `failcnt` counter is likely to be the one, limit of which didn't allow the application to start. In the following example, `httpd` failed to start because of the `numproc` limit:

```
# /etc/rc.d/init .d/httpd start
/etc/rc.d/init.d/httpd: fork : Cannot allocate memory
# cat /proc/user. _beancounters
Version: 2.4
uid    resource     held   maxheld barrier limit     failcnt
10026: kmemsize     263555 430517  798720  851968    0
       lockedpages  0      0       4       4         0
       privvmpages  773    1118    3072    3450      0
       shmpages     0      0       512     512       0
       dummy        0      0       0       0         0
       numproc      7      11      11      11        12
       physpages    638    969     0       2147483647 0
       vmguarpages  0      0       1725    2147483647 0
       oomguarpages 638    969     1725    2147483647 0
       numtcpsock   0      1       40      40        0
       numflock     0      1       50      60        0
       numpty       0      0       4       4         0
       numsiginfo   0      1       256     256       0
       tcpsndbuf    0      0       159744 262144     0
       tcprcvbuf    0      0       159744 262144     0
       othersockbuf 2224   9504    61440  163840     0
       dgramrcvbuf  0      4272    32768  32768      0
       numothersock 2      17      40     40         0
       dcachesize   38874  55176   184320 196608     0
       numfile      113    218     512    512        0
```

Also, a hint which limit prevents the application from starting and working can be obtained from the `strace(1)` logs as discussed in the previous section.

This and the previous subsection has discussed how to find the resource control parameter limitations on which caused the application to fail. The other important question is how to estimate how much resources the application needs.

The easiest way to determine the limits that allow the application to start and function is to raise the limits to some very high values and check the `maxheld` field in the usage statistics in the `/proc/user_beancounters` file after the application has started. That will show the "maximal" usage of the resources accounted into the resource control parameters. To effectively use the `maxheld` field in `/proc/user_beancounters`, Parallels Agent should be stopped. If stopping Parallels Agent is not desired, its API or Parallels Management Console should be used instead to obtain the information about the resource usage, see the note in the Controlling Resource Information in proc subsection (on page 46).

Note: If an application is unable to work, it doesn't necessarily mean that the system is misconfigured and the resource limits need to be increased. Server resources are limited, and it is impossible to allow each Container to start an infinite number of applications. So, before increasing the limits, it should be considered whether the application requires reasonable amount of resources (according to the existing usage and/or billing policy). The `vzcalc(8)` utility considered in the Checking Resource Configuration and Real Usage subsection (on page 21) can help understand how much system resources is already given to the Container and how many such Containers the system will be able to run.

Checking Application Correct Operation

To make sure that some application will work reliably for a long time, administrator needs to perform the following tests.

1. A simple functionality test. For example, for a Web server, accessibility of some Web pages may be checked. The functionality test is performed by appropriate client software, Web browser for HTTP server, FTP client for FTP server and so on.

   If something doesn't work as expected, the investigations may be done in a manner similar to tracing application start problems, explained in the previous sections. The sources of the information for the investigation are the same:

   - the `/proc/user_beancounters` file;
   - application `stderr` or `/var/log/boot.log`;
   - application log files (such as `/var/log/messages`);
   - system log files (`/var/log/messages` and `/var/log/daemon`);
   - kernel logs (`dmesg(8)`, `/var/log/kern`);
   - logs collected by `strace(l)`.

2. Verification that the application configuration matches the resource control settings.

   Most applications allow to configure some application-level limits limiting the resource consumption. For example, Apache Web server has MinSpareServers, MaxSpareServers and MaxClients parameters, controlling the number of Apache processes and, thus, the amount of the consumed memory and other resources. Some applications, such as Java machines, allow configuring the amount of memory that can be used by the machine directly.

   Certainly, the settings of the application-level parameters should agree with the system resource control parameters. Otherwise, the application may react inadequately or terminate operations when it hits the resource limits, instead of managing its load and resource usage by its own built-in mechanisms.

3. Tests under high load.

   Administrators should make real-life tests of the application under the highest expected load. For Web servers, the load can be created by

   - `http_load`, `http://www.acme.com/software/http_load/`;
   - `wget(l)`;
   - `httperf`, `ftp://ftp.hpl.hp.com/pub/httperf/`;

   or other tools.

With `http_load`, the following tests can be made:

- general measurement of maximum throughput;
- run with the `parallel` parameter equal or greater than the `MaxClients` parameter from the Apache configuration file, to test that the Web server works with the maximal allowed number of parallel connections,
- run with the `rate` parameter specifying the maximal expected page hit rate of the Web server.

The tests under the maximum expected load are important because the application may happen to start and work well under low load, but start to fail unexpectedly under higher load.

Checking Configuration Consistency of Single Container

System resource control parameters have certain interdependencies, so they can't be configured arbitrarily Configuration of resource control parameters for a Container is invalid if the constraints listed below are not satisfied.

Validation of the configuration should be performed together with the configuration creation, as discussed in the Configuring Resource Management Parameters for Containers section (on page 15), and each time the configuration is changed. The configuration validation can be performed by the `vzcfgvalidate(8)` tool (as explained in the Validating Configuration subsection (on page 19)), by Parallels Management Console (see Parallels Management Console Online Help), or through Parallels Agent (see the Parallels Agent documentation). Also, `vzctl(8)` can perform validation automatically at the Container start (see the Automatic Validation subsection (on page 21)).

This section contains detailed information about consistency checking for the cases where the diagnostics provided by `vzcfgvalidate(8)` is insufficient. This information can be used by experienced Virtuozzo administrators wishing to fine-tune the resource control settings and for developers of custom validation scripts, custom system monitors or in-house automatic hosting provision system.

The resource control configuration constraints are listed below. Indexes `bar` and `lim` in the formulae below mean the barrier and the limit of the parameters, respectively:

1 `kmemsize` should be enough for the expected number of processes:

$$\texttt{kmemsize}_{\texttt{bar}} \geq 40\text{KB} \cdot \texttt{avnumproc} + \texttt{dcachesize}_{\texttt{lim}} \qquad (1)$$

(`avnumproc` stands for the expected average number of processes). This constraint is important for reliable work of applications in the Container. If it is not satisfied, applications will start to fail at the middle of operations instead of failing at the moment of spawning more processes, and the application abilities to handle resource shortage will be very limited.

2 Memory allocation limits should not be less than the guarantee.

$$\texttt{privvmpages}_{\texttt{bar}} \geq \texttt{vmguarpages}_{\texttt{bar}} \qquad (2)$$

If this constraint is not satisfied, `vmguarpages` guarantee will not work.

3 Send buffers should have enough space for all sockets.

$$\texttt{tcpsndbuf}_{\texttt{lim}} - \texttt{tcpsndbuf}_{\texttt{bar}} \geq 2.5\texttt{KB} \cdot \texttt{numtcpsock} \quad (3)$$

$$\texttt{othersockbuf}_{\texttt{lim}} - \texttt{othersockbuf}_{\texttt{bar}} \geq 2.5\texttt{KB} \cdot \texttt{numothersock} \quad (4)$$

These constraints are also important. If they are not satisfied, transmission of data over the sockets may hang in some circumstances.

4 Other TCP socket buffers should be big enough:

$$\texttt{tcprcvbuf}_{\texttt{lim}} - \texttt{tcprcvbuf}_{\texttt{bar}} \geq 2.5\texttt{KB} \cdot \texttt{numtcpsock} \quad (5)$$

$$\texttt{tcprcvbuf}_{\texttt{bar}} \geq 64\texttt{KB} \quad (6)$$

$$\texttt{tcpsndbuf}_{\texttt{bar}} \geq 64\texttt{KB} \quad (7)$$

Selecting the left side equal to the right side in the inequalities above ensures minimal performance of network communications. Increasing the left side will increase the performance to a certain extent.

5 Buffers of non-TCP sockets should not be excessively small:

$$\texttt{dgramrcvbuf}_{\texttt{bar}} \geq 32\texttt{KB} \quad (8)$$

$$\texttt{othersockbuf}_{\texttt{bar}} \geq 32\texttt{KB} \quad (9)$$

6 If the system has enough memory, UDP socket buffers should be bigger:

$$\texttt{dgramrcvbuf}_{\texttt{bar}} \geq 129\texttt{KB} \quad (10)$$

$$\texttt{othersockbuf}_{\texttt{bar}} \geq 129\texttt{KB} \quad (11)$$

These constraints are desired, but not essential. Big enough buffers for UDP sockets improve reliability of datagram delivery. However, note that if the UDP traffic is so bursty that it needs larger buffers, the datagrams will likely be lost not because of resource control limits, but because of other memory and performance limitations.

7 Number of file limit should be adequate for the expected number of processes:

$$\texttt{numfile} \geq \texttt{avnumproc} \cdot 32 \quad (12)$$

Note that each process after the `execve(2)` system call requires a file for each loaded shared library. Too low numfile limit will increase the chances of failures during the `execve(2)` call with diagnostics not clear for the users.

8 Since sockets and devices are also considered as files, the limit on the number of files should not be less than the total limit on the number of sockets and pseudo-terminals:

$$\texttt{numfile} \geq \texttt{numtcpsock} + \texttt{numothersock} + \texttt{numpty} \quad (13)$$

Lower numfile limits usually do not make sense, not allowing Containers to use all the configured sockets.

9 The limit on the total size of `dentry` and `inode` structures locked in memory should be adequate for allowed number of files:

$$\texttt{dcachesize}_{\texttt{bar}} \geq \texttt{numfile} \cdot 384 \text{ bytes} \quad (14)$$

Too low `dcachesize` limit will increase the chances of file operation refusals not expected by applications.

10 In addition to the conditions listed above

$$\mathtt{barrier} \leq \mathtt{limit} \qquad (15)$$

should be maintained for each parameter.

All the interdependencies discussed above and their importance are summarized in the following table:

1	$\mathtt{kmemsize}_{\mathtt{bar}} \geq 40\mathrm{KB} \cdot \mathtt{avnumproc} + \mathtt{dcachesize}_{\mathtt{lim}}$	mandatory
2	$\mathtt{privvmpages}_{\mathtt{bar}} \geq \mathtt{vmguarpages}_{\mathtt{bar}}$	recommended
3	$\mathtt{tcpsndbuf}_{\mathtt{lim}} - \mathtt{tcpsndbuf}_{\mathtt{bar}} \geq 2.5\mathrm{KB} \cdot \mathtt{numtcpsock}$	mandatory
4	$\mathtt{othersockbuf}_{\mathtt{lim}} - \mathtt{othersockbuf}_{\mathtt{bar}} \geq 2.5\mathrm{KB} \cdot \mathtt{numothersock}$	mandatory
5	$\mathtt{tcprcvbuf}_{\mathtt{lim}} - \mathtt{tcprcvbuf}_{\mathtt{bar}} \geq 2.5\mathrm{KB} \cdot \mathtt{numtcpsock}$	recommended
6	$\mathtt{tcprcvbuf}_{\mathtt{bar}} \geq 64\mathrm{KB}$	recommended
7	$\mathtt{tcpsndbuf}_{\mathtt{bar}} \geq 64\mathrm{KB}$	recommended
8	$\mathtt{dgramrcvbuf}_{\mathtt{bar}} \geq 32\mathrm{KB}$	recommended
9	$\mathtt{othersockbuf}_{\mathtt{bar}} \geq 32\mathrm{KB}$	recommended
10	$\mathtt{dgramrcvbuf}_{\mathtt{bar}} \geq 129\mathrm{KB}$	optional
11	$\mathtt{othersockbuf}_{\mathtt{bar}} \geq 129\mathrm{KB}$	optional
12	$\mathtt{numfile} \geq \mathtt{avnumproc} \cdot 32$	recommended
13	$\mathtt{numfile} \geq \mathtt{numtcpsock} + \mathtt{numothersock} + \mathtt{numpty}$	recommended
14	$\mathtt{dcachesize}_{\mathtt{bar}} \geq \mathtt{numfile} \cdot 384$ bytes	recommended
15	$\mathtt{barrier} \leq \mathtt{limit}$	mandatory

The configured limits can be checked through

- Container configuration files (see the `ve.conf(5)` manual pages and Parallels Virtuozzo Containers Reference Guide) in `/etc/vz/conf`;
- the `/proc/user_beancounters` interface (see the Controlling Resource Information in proc section (on page 46));
- Parallels Agent (see the Parallels Agent documentation).

Inspecting resource control parameter configurations in general is discussed in the Inspecting Resource Control Settings subsection (on page 45).

Checking Usage and Distribution of System Resources and Validating Settings of All Containers

The previous section discussed the validation of the resource control configuration of a single Container. This section discusses checks that the configuration of the whole system is valid.

Configurations where resources allowed for Containers exceed system capacity (more precisely, configurations with excessive overcommitment as explained below) are invalid and dangerous from the stability point of view. They may result in abnormal termination of the applications, bad responsiveness of the system and, sometimes, system hangs. Whereas the configuration validation discussed in the previous section addressed application functionality, the validation considered in this section is aimed at security and stability of the whole system.

Resource Utilization and Commitment Level

Several resources of the whole system (such as RAM) are discussed below in terms of utilization and commitment level.

Utilization shows the amount of resources consumed by all Containers at the given time. In general, low utilization values mean that the system is under-utilized. Often, it means that the system is capable of supporting more Containers if the existing environments continue to maintain the same load and resource consumption level. High utilization values (in general, more than 1) mean the the system is overloaded and the service level of the Containers is degraded.

Commitment level shows how much resources are "promised" to the existing Containers. Low commitment levels mean that the system is capable of supporting more Containers. Commitment levels more than 1 mean that the Containers are promised more resources than the system has, and the system is said to be overcommitted. If the system runs a lot of Containers, it is usually acceptable to have some overcommitment, because it is unlikely that all Containers will request resources at the same time. However, very high commitment levels (as discussed for each resource individually below) will cause Containers to fail to allocate and use the resources promised to them and may hurt system stability.

Summarizing written in the previous paragraph, overcommitment is a trade-off between efficiency and utilization of the hardware on one hand and the ability to guarantee resources and system stability on the one hand.

Low Memory

Because of specifics of architecture of Intel's processors, the RAM of the server can't be used uniformly The most important memory area is so called "low" memory, a part of memory residing at lower addresses and directly accessible by the kernel.

In Virtuozzo 2.5.1 release, the size of the "low" memory area was limited by 832MB. In Virtuozzo 2.5.1 Service Pack 1 and later versions including Virtuozzo 4.0, the "low" memory area size is limited by 832MB in UP (uniprocessor) version of the kernel and by 3.6GB in SMP and Enterprise versions of the kernel. If the total size of server RAM is less than the limit (832MB or 3.6GB, respectively), then the actual size of the "low" memory area is equal to the total memory size. This actual size is shown in the `LowTotal` line in `/proc/meminfo`.

Utilization. The lower bound estimation of "low" memory utilization is

$$\frac{\sum_{all\ VE} \left(\mathtt{kmemsize}_{\mathtt{cur}} + \mathtt{allsocketbuf}_{\mathtt{cur}}\right)}{0.4 \cdot low\ area\ size},$$

where

$$\begin{aligned}\mathtt{allsocketbuf} = {} & \mathtt{tcprcvbuf} + \mathtt{tcpsndbuf} + \\ & \mathtt{dgramrcvbuf} + \mathtt{othersockbuf}\end{aligned}$$

Utilization of "low" memory below 1 is normal. Utilization above 1 is not safe, and utilization above 2 is dangerous and is likely to cause bad system responsiveness, application stalls for seconds or more and termination of some applications.

Commitment level. The commitment level can be computed as

$$\frac{\sum_{all\ VE}\left(\texttt{kmemsize}_{\texttt{lim}} + \texttt{allsocketbuf}_{\texttt{lim}}\right)}{0.4 \cdot \min(RAM\ size, low\ area\ size)}$$

Commitment levels below 1 are normal. Levels between 1 and 1.2 are usually acceptable for systems with about 100 Containers. Systems with more Containers may have the commitment level increased. to about 1.5-2 for 400 environments.

If the Containers actually claim the "low" memory allowed for them and the utilization closes to 1, the performance and stability of the whole system will dramatically drop. So, commitment levels exceeding the suggested ones should be considered unsafe and are not recommended.

Total Memory

This subsection considers utilization of the whole RAM. Usage of swap space and the sum of used RAM and swap space are discussed in the Memory and Swap Space subsection (on page 58).

Current Virtuozzo version can't guarantee availability of certain amount of memory (in opposite to the sum of memory and swap space), so the commitment level is not applicable to RAM. RAM guarantees will be implemented in future versions.

Utilization. The amount of RAM consumed by all Containers can be computed as

$$\sum_{all\ VE}\left(\texttt{physpages}_{\texttt{cur}} \cdot 4096 + \texttt{kmemsize}_{\texttt{cur}} + \texttt{allsocketbuf}_{\texttt{cur}}\right)$$

all Containers

The difference between memory usage shown by `free(l)` or `/proc/meminfo` and the total amount of RAM consumed by Containers is the memory used by system daemons and different caches. The memory utilization can be computed as

$$\frac{\sum_{all\ VE}\left(\texttt{physpages}_{\texttt{cur}} \cdot 4096 + \texttt{kmemsize}_{\texttt{cur}} + \texttt{allsocketbuf}_{\texttt{cur}}\right)}{RAM\ size}$$

Utilization levels from 0.8 to 1 are normal. Lower utilization means that the system is under-utilized, and, if other system resources and their commitment levels permit, the system can host more Containers. By the nature of the accounting of `physpages` and other parameters, total RAM utilization can't be bigger than 1.

Memory and Swap Space

The main resource of the server determining the amount of memory available for applications is the union of RAM and swap space. If the total size of memory used by applications exceeds the RAM size, Linux kernel moves some data to swap and loads it back when the application needs it. More frequently used data tends to stay in RAM, less frequently used data spends more time in swap.

Swap-in and swap-out activity reduces the system performance to some extent. However, if this activity is not excessive, the performance decrease is not very noticeable. On the other hand, the benefits of using swap space are quite big, allowing to increase the number of Containers in the system about 2 times. Standard Linux monitoring tool vmstat(8) shows the swap-in and swap-out activity. The activity below a hundred kilobytes a second usually does not significantly affect the responsiveness of network server applications such as Web, FTP and mail servers.

Swap space is essential for handling system load bursts. System with enough swap space just slows down at high load bursts, whereas the system without swap space reacts to high load bursts by refusing memory allocations (causing applications to refuse to accept clients or terminate) and by direct killing of some applications. Additionally, the presence of swap space helps the system to better balance memory and move data between "low" memory and the rest of the RAM.

In all Virtuozzo installations it is strongly recommended to have swap space of size not less than the RAM size.

Also, it is not recommended to create swap space of the size of more than 4 times RAM size because of performance degradation related to swap-in and swap-out activity. That is, the system should be configured so that

$$RAM\ size \le swap\ size \le 4 \cdot RAM\ size$$

The optimal configuration is the swap space size twice bigger than the RAM size.

Utilization.

$$\frac{\sum_{all\ VE} \left(\texttt{oomguarpages}_{\texttt{cur}} \cdot 4096 + \texttt{kmemsize}_{\texttt{cur}} + \texttt{allsocketbuf}_{\texttt{cur}}\right)}{RAM\ size + swap\ size}$$

The normal utilization of memory and swap space ranges between

$$\frac{RAM\ size}{RAM\ size + swap\ size} \quad \text{and} \quad \frac{RAM\ size + \frac{1}{2} swap\ size}{RAM\ size + swap\ size}$$

Lower utilization means that the system memory is under-utilized at the moment of checking the utilization. Higher utilization is likely to cause gradual performance degradation because of swap-in and swap-out activity and is a sign of overloading the system.

Commitment level.

$$\frac{\sum_{all\ VE} \left(\texttt{oomguarpages}_{\texttt{bar}} \cdot 4096 + \texttt{kmemsize}_{\texttt{lim}} + \texttt{allsocketbuf}_{\texttt{lim}}\right)}{RAM\ size + swap\ size}$$

The normal commitment level is about 0.8-1.

Commitment levels more than 1 means that the Containers are guaranteed more memory than the system has. Such overcommitment is strongly not recommended. If memory and swap space are overcommitted and all the memory is consumed, random applications, including the ones belonging to the host system, may be killed, and the system may become inaccessible by `ssh(l)` and lose other important functionality.

It is better to guarantee Containers less memory and have less commitment levels than to accidently overcommit the system in memory and swap space. If the system has spare memory and swap, Containers will transparently be able to use the memory and swap above their guarantees. Guarantees given to Containers should not be big, and it is normal if memory and swap space usage for some environments stays above their guarantee. It is also normal to give guarantees only to Containers with preferred service. But administrators should not guarantee Container more than the system actually has.

Allocated Memory

This subsection considers standard memory allocations made by applications in the Container. The allocations inside each Container are controlled by two parameters: `vmguarpages` and `privvmpages` explained in the Primary Parameters (on page 33) and Secondary Parameters (on page 36) subsections.

Allocated memory is a more "virtual" system resource than the RAM or RAM and swap space. Applications can allocate memory but start to use it only later, and the amount of free memory will decrease only at the moment of the use. The sum of the sizes of memory allocated in all Containers is the estimation of how much physical memory will be used when (and if) all applications claim the allocated memory.

Utilization.

$$\frac{\sum_{all\ VE}\left(\texttt{privvmpages}_{\texttt{cur}}\cdot 4096+\texttt{kmemsize}_{\texttt{cur}}+\texttt{allsocketbuf}_{\texttt{cur}}\right)}{RAM\ size+swap\ size}$$

This utilization level is the ratio of the amount of allocated memory to the capacity of the system.

Low utilization level means that the system can support more Containers, if other resources permit. High utilization levels may, but doesn't necessarily, mean that the system is overloaded. As it was explained above, not all applications use all the allocated memory, so this utilization level may exceed 1.

This utilization level can be used together with the commitment level and the level of memory allocation restrictions (discussed below) to configure memory allocation restrictions for the Containers.

Commitment level. Allocated memory commitment level

$$\frac{\sum_{all\ VE}\left(\texttt{vmguarpages}_{\texttt{bar}}\cdot 4096+\texttt{kmemsize}_{\texttt{lim}}+\texttt{allsocketbuf}_{\texttt{lim}}\right)}{RAM\ size+swap\ size}$$

is the ratio of the memory size guaranteed to be available for allocations to the capacity of the system. Similarly to the commitment level of memory and swap space (as discussed in the Memory and Swap Space subsection (on page 58)), this level should be kept below 1. If the level is above 1, it significantly increases the chances of applications to be killed instead of be notified in case of a memory shortage. It's better to provide lower guarantees than to accidently guarantee more than the system has, because Containers are allowed to allocated memory above their guarantee if the system has spare memory. It is also normal to give guarantees only to Containers with preferred service.

Limiting memory allocations. In addition to providing allocation guarantees, it is possible to impose restrictions on the amount of memory allocated by applications inside a Container.

If a system has multiple Containers, it is important to make sure that for each Container

$$\texttt{privvmpages}_{\texttt{lim}} \cdot 4096 \leq 0.6 \cdot RAM\ size.$$

If this condition is not satisfied, a single Container may easily cause an excessive swap-out and very bad performance of the whole system. Usually, for each Container `privvmpages` limitations are set to values much less than the size of the RAM.

The resource control parameters should be configured in a way, so that in case of memory shortage applications are given chance to notice the shortage and exit gracefully, instead of being terminated by the kernel. For this purpose, it is recommended to maintain reasonable total level of memory allocation restrictions, computed as

$$\frac{\sum_{all\ VE} \left(\texttt{privvmpages}_{\texttt{lim}} \cdot 4096 + \texttt{kmemsize}_{\texttt{lim}} + \texttt{allsocketbuf}_{\texttt{lim}}\right)}{RAM\ size + swap\ size}$$

This number shows how much memory applications are allowed to allocate in comparison with the capacity of the system.

In practice, a lot of applications do not use the memory very efficiently and, sometimes, allocated memory will never be used later. For example, Apache Web server at the start time allocates about 20-30%% more memory that it will ever use. Some multi-threaded applications are especially bad at using their memory, and their rate of allocated to used memory may happen to be 1000%.

The bigger the level of memory allocation restrictions is, the more chances are that applications will be killed instead of getting an error on the next memory allocation in case of a memory shortage. The levels of memory allocation restrictions between 1.5 and 4 are usually acceptable. Administrators can find experimentally the optimal settings for their load, basing on the frequency of messages "Out of Memory: killing process" in system logs, saved by `klogd(8)` and `syslogd(8)`. However, for stability-critical applications, it's better to keep the level not exceeding 1.

Checking Resource Usage and Settings With Advanced Utilities

Virtuozzo provides several utilities computing utilization and commitment levels according to the formulae discussed above. These utilities help to understand

- the general utilization of the system;
- the amount of resources "promised" to Containers (i.e. the commitment);
- the relative shares of the total system resources used by and promised to each Container, allowing to find the Containers with biggest resource consumption, the environments consuming more and consuming less than their share according to the configuration;
- the safety of the resource control settings.

Checking Resource Utilization and Settings of Single Container

The utility for checking resource utilization and settings for a single Container is `vzcalc(8)`. An example of the `vzcalc(8)` output is shown below:

```
# vzcalc -v 8011
Resource   Current(%)    Promised(%)   Max(%)
Low Mem    0.10          1.16          1.16
Total RAM  0.06          n/a           n/a
Mem + Swap 0.03          0.56          n/a
Alloc. Mem 0.06          0.56          2.01
Num. Proc  0.05          n/a           0.49
----------------------------------------
Memory     0.10          1.16          2.01
```

If the Container is running, in the `Current` column `vzcalc(8)` shows the current utilization of "low" memory, total RAM, RAM plus swap space, total size of memory allocations and number of processes. `Current` and all other values are shown as percents of the system capacity.

The `Promised` column shows how much the Container will be able to allocate without a denial. The `Max` column shows the upper limit on what the Container can allocate. If the resource requirements of the Container are between the `Promised` and `Max` values, resource allocations may be granted or denied depending on the amount of spare resources the system has.

The summary line (`Memory`) summarizes utilization and limits on the resource consumption. If `vzcalc(8)` is invoked without the `-v` option, only summary is shown. This summary can be used for a rough estimation of how many such Containers can be run on this system, as described in the Checking Resource Usage of Single Container subsection (on page 22).

Explaining vzcalc(8) Output

The exact description of the value in the `Promised` and `Max` columns is given below.

In the `Low Mem` line, `Promised` and `Max` show the same value, computed as

$$\frac{\texttt{kmemsize}_{\texttt{lim}} + \texttt{allsocketbuf}_{\texttt{lim}}}{0.4 \cdot \min(RAM\ size, low\ area\ size)} \cdot 100\%,$$

which is equal to the contribution of this Container to "low" memory commitment level (see the Low Memory subsection (on page 56)).

In the `Mem + Swap` line, `Promised` shows memory plus swap space guarantee, computed as

$$\frac{\texttt{oomguarpages}_{\texttt{bar}} \cdot 4096 + \texttt{kmemsize}_{\texttt{lim}} + \texttt{allsocketbuf}_{\texttt{lim}}}{RAM\ size + swap\ size} \cdot 100\%,$$

which is equal to the contribution of this Container to memory plus swap commitment level (see the Memory and Swap Space subsection (on page 58)).

In the `Alloc. Mem` line, `Promised` shows allocated memory guarantee (see the Allocated Memory subsection (on page 59)), computed as

$$\frac{\texttt{vmguarpages}_{\texttt{bar}} \cdot 4096 + \texttt{kmemsize}_{\texttt{lim}} + \texttt{allsocketbuf}_{\texttt{lim}}}{RAM\ size + swap\ size} \cdot 100\%,$$

and `Max` shows memory allocation restriction, computed as

$$\frac{\texttt{privvmpages}_{\texttt{lim}} \cdot 4096 + \texttt{kmemsize}_{\texttt{lim}} + \texttt{allsocketbuf}_{\texttt{lim}}}{RAM\ size + swap\ size} \cdot 100\%.$$

Checking Resource Utilization and Settings for Whole System

The utility for checking resource utilization and settings for the whole system is `vzmemcheck(8)`. An example of the `vzmemcheck(8)` output is shown below:

```
# vzmemcheck -v
Output values in •/.
ctid    LowMem  LowMem   RAM  MemSwap   MemSwap  Alloc   Alloc   Alloc
        util    commit  util     util    commit   util  commit   limit
10003   0.10    1.16    0.19     0.05    0.56     0.09    0.56    2.01
109     0.15    1.16    0.31     0.10    0.56     0.23    0.56    2.01
111     0.15    1.16    0.30     0.10    0.56     0.22    0.56    2.01
104     0.01    1.16    0.01     0.00    0.56     0.00    0.56    2.01
5006    0.11    1.16    0.13     0.05    0.56     0.09    0.56    2.01
119     0.11    12.41   0.12     0.05    9.28     0.09    9.28   23.86
...
1       0.33    5.25    8.21      2.28    2.93    2.77    0.83   23.02
------------------------------------------------------------------------
---
Summary: 5.45  127.08  27.08      7.60    84.74   36.22   82.65
225.81
```

This utility computes in percents the "low" memory utilization and commitment levels (see the Low Memory subsection (on page 56)), total RAM utilization (see the Total Memory subsection (on page 57)), memory plus swap space utilization and commitment levels (see the Memory and Swap Space subsection (on page 58)), allocated memory utilization, commitment and allocation restriction levels (see the Allocated Memory subsection (on page 59)). Note that these values are computed for currently running Containers and do not include the Container placed on this system but stopped at this moment.

With the `-v` option, `vzmemcheck(8)` shows the utilization and commitment levels broken down to each Container.

The `vzmemcheck(8)` output can be used for evaluation of

- whether the system is overcommitted and the current overcommitment level (thus, checking how safe the configuration is)
- total utilization of the resources of the system
- whether more Containers can be placed on the system
- whether the system is overloaded and either some Containers should be moved out or more memory should be added to the system.

In the example above, the system commitment levels are within the suggested ranges. However, the physical resources of the system are utilized by 27.08% (the biggest of "low" memory, total RAM and memory plus swap utilization). The difference between utilization and commitment levels is that big because the `vzmemcheck(8)` snapshot was taken at "quite" hours when the load on Containers was low.

If the maximal daily utilization stays below 100% Virtuozzo administrator may decide to place more Containers on the system. He may either decide to take the risk of higher commitment levels (first of all, "low" memory commitment in this example) and lower stability or lower limits on "low" memory consumption by Containers (especially, Container #119 in this example). Lower limits may cause application failures in this Container if the memory consumption by it suddenly increases, but the lower limits will save the whole system from performance degradation and possible hangs if almost all "low" memory is consumed.

Checking System Overcommitment

The utility for checking current system overcommitment and safety of the total resource control settings is `vzcheckovr (8)`. An example of the output of `vzcheckovr(8)` is shown below:

```
# vzcheckovr
Low Memory commitment 139.49% exceeds warning level (120.00%)
Warning: node configuration is unsafe
# vzcheckovr -v
                  Current commitments(%)  Warning level(%)
Low Memory                       139.49             120.00
Memory + Swap                     84.02             100.00
Alocated Memory                   81.93             100.00
                  Current limits(%) Warning level(%)
Total Alloc Limit                292.97             400.00
Max Alloc Limit                   38.62              50.00
Warning: node configuration is unsafe
```

This utility computes commitment levels as described in the Checking Usage and Distribution of System Resources and Validating Settings for All Containers (on page 55) section and compares them with the values chosen by the Virtuozzo administrator. Virtuozzo administrator decides which commitment levels are acceptable for him from the point of utilization-safety trade-off, as discussed in the Resource Utilization and Commitment Level (see page 56) subsection and stores the values in the Virtuozzo global configuration file (`/etc/sysconf ig/vz`). Then `vzcheckovr(8)` will produce a warning if these configured values are exceeded. Similarly to `vzmemcheck(8)`, `vzcheckovr(8)` takes into account only currently running Containers and ignores the ones placed on this system but stopped at this moment.

The brief `vzcheckovr(8)` output just shows if (and which) commitment level exceeds the warning level. Detailed `vzcheckovr(8)` output (with the `-v` option) shows all current commitment levels and corresponding warning levels. All values are shown and kept in the configuration in percents.

'Low Memory", "Memory + Swap" and "Allocated Memory" commitment levels in the `vzcheckovr(8)` output reflect the commitment levels described in the Checking Usage and Distribution of System Resources and Validating Settings for All Containers section (on page 55). "Total Alloc Limit" is the total level of memory allocation restrictions described in the Allocated Memory subsection (on page 59), i.e. the sum of memory allocation restrictions of all Containers divided by the sum of RAM and swap sizes. For stability reasons, memory allocation restriction for each Container individually should not be greater than the RAM size. To check if this condition is satisfied, `vzcheckovr(8)` shows and checks "Max Alloc Limit" - the biggest memory allocation restriction among all running Containers divided by the RAM size.

Automatic Checks

The overcommitment check described in the previous subsection can be performed automatically at each Container start. This option is controlled by the `OVERCOMMITMENT_ACTION` variable in the Virtuozzo global configuration file (`/etc/vz/vz.conf`). Depending on the setting of this variable, `vzctl(8)` can issue a warning or refuse to start the Container if after the start the overcommitment levels would exceed the warning levels. In the current version of Virtuozzo, the default for this option is no automatic check at the Container start.

Monitoring Resources

Implementation of monitoring is an important system administration task. Properly implemented, monitoring increases quality of service for the users and availability of the system.

Virtuozzo provides monitoring tools allowing:

- checking the current load, responsiveness, used and free resources and other system "health" parameters (the `vzstat(8)` utility);
- remote checking of availability, load, and responsiveness of Virtuozzo systems (the `vzrmon` package) and sending alerts by email, ICQ or SMS;
- producing server load and availability reports and diagrams (the `vzrmon` package).

To implement custom monitoring solutions, Virtuozzo administrators may also use

- general system and application health monitoring systems, such as Big Brother (http://bb4.com/);
- the `CUST0M_ACTI0N` variable in the `vzrmond(8)` configuration;
- custom scripts using text (`-t`) output of the `vzstat(8)` utility;
- custom programs using Parallels Agent interface (see the Parallels Agent documentation):
- custom programs using low-level interfaces and tools, such as
  - **a** `/proc/user_beancounters` (see the Controlling Resource Information in proc subsection (on page 46)), paying special attention to the `failcnt` and `maxheld` fields,
  - **b** kernel logs, available from the `dmesg(8)` utility and also usually stored in `/var/log/messages` by the `klogd(8)` and `syslogd(8)` daemons,
  - **c** tools like `ping(8)`, `wget(l)`, `http_load` (http://www.acme.com/software/http_load/) to check accessibility of network applications in the Containers.

Caution! It is important to make sure that the monitoring is not excessive, i.e. that the monitoring itself doesn't create the load bigger than (or comparable with) the main functionality of the server. Note that running multiple Containers on a single server doesn't resemble multiple servers in this respect. Accidental increase of the load of each Container by just 0.5% will bump the utilization of the server to 100% if the server runs 200 environments. Especially dangerous are the scripts and monitoring tools that not only monitor the state of the servers and applications, but take actions (such as application restart) if they "feel" that something goes wrong.

View System Information With vzstat

The `vzstat(8)` utility provides Virtuozzo administrators with the information about the current:

- load of the system, i.e. how "work" the system does;
- responsiveness of the system, i.e. how quickly the system responses to incoming HTTP and other requests, shell commands and so on,
- utilization of system resources and the amount of spare resources.

`vzstat(8)` also makes basic analysis of the figures being shown and highlights the figures that need administrator's attention and may be signs of overload or other problems. An example of the `vzstat(8)` output is shown below:

```
  7:27am, up 4 days, 20:53, 12 users, load average: 2.01, 1.74, 1.50
VENum 59, procs 785: running 2, sleeping 781, unint 2, zombie 0, stopped 0
CPU [ OK ]: VEs   1%, VE0  98%, user  39%, sys  61%, idle   0%, lat(ms) 67/0
Mem [ OK ]: total 1007MB, free 12MB/1MB (low/high), lat(ms) 0/0
  ZONE0 (DMA): size 16MB, act 5MB, inact 1MB, free 2MB (0/0/0)
  ZONE1 (Normal): size 816MB, act 107MB, inact 126MB, free 9MB (0/1/2)
  ZONE2 (HighMem): size 191MB, act 52MB, inact 129MB, free 1MB (0/1/2)
  Mem lat (ms): A0 0, K0 0, U0 0, K1 0, U1 0
  Slab pages: 530MB/530MB (ino 436MB, de 40MB, bh 35MB, pb 1MB)
Swap [ OK ]: tot 3961MB, free 3806MB, in 0.000MB/s, out 0.000MB/s
Net [ OK ]: tot in 0.001MB/s 24pkt/s, out 0.001MB/s 21pkt/s
  lo: in 0.000MB/s 0pkt/s, out 0.000MB/s 0pkt/s
  eth0: in 0.001MB/s 24pkt/s, out 0.001MB/s 21pkt/s
  eth1: in 0.000MB/s 0pkt/s, out 0.000MB/s 0pkt/s
Disks [ OK ]: in 0.000MB/s, out 0.000MB/s

  VEID ST    %VM     %KM         PROC    CPU     SOCK FCNT MLAT IP
     1 OK 6.9/102 0.3/1.8    0/69/256 0.0/0.0 207/1256    0    0 10.101.60.79
...
```

Using `vzstat(8)` Virtuozzo administrators may

- check whether the users receive reasonable quality of service;
- evaluate whether the system has spare resources and more Containers can be placed on the system without degradation of the quality of service;
- find out which Containers consume more resources than others;
- perform troubleshooting of quality of service and performance issues.

Output screen. The output screen of `vzstat(8)` consists of 2 parts: information about system as a whole (top part) and information about individual Containers.

The top part contains 5 sections for CPU, memory, swap space, network and disk resources. Each section displays the state ("OK" or "FAIL") according to whether `vzstat(8)` noted any problem concerning this resource. The verbosity level of the information in each section can be changed by the administrator (see the `vzstat(8)` manual pages).

The bottom part shows information about "top" Containers according to one of the sort criteria: CPU usage, memory usage, total number of processes, number of running processes, number of resource allocation refusals (see the `failcnt` description in the Controlling Resource Information in proc subsection (on page 46)).

Information about system load. The most important figures in the `vzstat(8)` output related to the system load are:

- number of running Containers;
- total number of processes, number of running and "unint" (waiting for I/O completion) processes;
- the percentage of CPU idle time (the less idle time is, the more CPU-intensive work the system does);
- used network bandwidth;
- disk I/O rate.

Information about system responsiveness. The figures indicating system responsiveness are:

- scheduling latency (last figures in the `CPU` line);
- memory allocation latency;
- swap input/output rate;
- disk I/O rate.

Information about used and spare resources. The information about used and spare resources is present in CPU, memory, swap and disk sections.

CPU resource may be considered as fully used if the percentage of idle time is close to 0 and the scheduling latency increases to hundreds milliseconds.

Memory resource may be considered as fully used if the amount of free memory in normal or high zones is 1MB or less and either memory allocation latency is high or swapin/swapout rates are high.

Network resource is fully used if the used bandwidth becomes close to the capabilities of the link.

APPENDIX A

Changes From Previous Releases

In This Appendix

Changes Between Virtuozzo 2.6 and 3.0 68
Changes Between Virtuozzo 2.5.1 and 2.6 68
Changes Between Virtuozzo 2.5 and 2.5.1 68
Changes Between 2.0.2 and 2.5 69

Changes Between Virtuozzo 2.6 and 3.0

No changes.

Changes Between Virtuozzo 2.5.1 and 2.6

Computations in the `vzsplit(8)` utility were refined.

Changes Between Virtuozzo 2.5 and 2.5.1

- Computations in configuration validation utilities (`vzcfgvalidate(8)`, `vzcalc(8)`, `vzmemcheck(8)`) were refined.
- The `vzcfgscale(8)` utility was made safe to be run with the same input and output file.
- A new utility - `vzcheckovr(8)` - for checking the resource configuration for the whole system and overcommitment levels (see the Checking System Overcommitment subsection (on page 64)) was introduced.
- `vzctl(8)` can perform automatic validation of the Container's and system configurations (see the Automatic Validation (on page 21) and Automatic Checks (on page 64) subsections).

Absolute CPU time consumption limits (the `--cpulimit` option of `vzctl (8)`) were implemented in addition to CPU time guarantee and relative power (see the Types of Virtuozzo CPU Time Consumption Control subsection (on page 25)).

Changes Between 2.0.2 and 2.5

This section is a brief description of the difference in the management of system resources between Virtuozzo versions 2.0.2 and 2.5.

The `vzctl` package provides compatibility with the previous release. The `vzctl(8)` utility can operate with Container configurations, start the Containers and update the configurations with Virtuozzo 2.0.x resource parameter sets. Also, Container configurations can be converted into Virtuozzo 2.5x format by `the vzcfgconvert(8)` utility from the `vzctl` package.

- The changes in the system resource control parameters are the following:
  - new parameters: `privvmpages`, `dcachesize`, `numfile`
  - obsolete parameters: `anonshpages`, `totvmpages`
  - some parameters changed their names and meanings:

    `-numtcpsock`: former `numsock`, but doesn't count non-`TCP PF_INET` sockets (UDP and other) now;

    `-numothersock`: former `numunixsock`, but it's started to count UDP and other sockets now;

    `-tcpsndbuf`: former `tcpsendbuf`, now it ensures certain fairness of buffer use between sockets and requires a gap between the barrier and the limit;

    `-tcprcvbuf`: the same name as before, now it ensures certain fairness of buffer use between sockets and requires a gap between the barrier and the limit;

    `-othersockbuf`: former `unixsockbuf`, now it ensures certain fairness of buffer use between sockets and requires a gap between the barrier and the limit;

    `-dgramrcvbuf`: former `sockrcvbuf`, renamed to eliminate ambiguity in which socket type this parameter controls;

    `-shmpages`: former `ipcshmpages`; but counts anonymous shared pages (former `anonshpages`) and `tmpfs` objects now;

    `-physpages`: former `rsspages`, but includes in-use `shmpages` and doesn't have swap-out guarantee now;

    `-vmguarpages`: former `vmspaceguar`, a guarantee for `privvmpages` now;

    `-oomguarpages`: former `oomguar`, has its own accounting and includes swap space and `shmpages` now.
- Changes in the `/proc/user_beancounters` file format:
  - version string has been added;
  - `maxheld` field has changed the name from "max";
  - `failcnt` field has been added.

- The rules of configuration consistency (see the Checking Configuration Consistency of Single Container (on page 53) subsection) and the configuration examples have also been changed.
- New utilities helping create, validate and review Containers configurations were introduced. The utilities are: `vzcfgscale(8)`, `vzsplit(8)`, `vzcfgvalidate(8)`, `vzcalc(8)`, `vzmemcheck(8)`.

Compatibility with the previous Virtuozzo releases is provided by the `vzctl` package.

Changes in resource control affecting compatibility will be minimized in future versions of Virtuozzo, and `vzctl` in those future versions will also maintain compatibility with previous releases.

APPENDIX B

Examples

In This Appendix

Examples of System Resources Parameter Settings 71
Examples Explanation..... 73
Not Specified Values 74

Examples of System Resources Parameter Settings

The table below contains example settings of the system resources parameters. There are 4 example configurations - A, B, C, and D, in the order of increasing Container power.

Power represents the power of the Container. It is shown as the RAM size of the server and the number of Containers of this type that can be run on such a server separated by the "/" sign. The examples assume that the system has swap space twice bigger than the RAM.

Helper values are intermediate values, produced during computation of the resource control parameter limits. These values help to understand the process of computing the limits and to verify the result. total mem represents the total amount of RAM allowed to be used by each Container, kernel mem is its kernel fraction (consisting of kmemsize and all socket buffers) and user mem is the memory allowed to be allocated by processes. avnumproc is the expected average number of processes, as used in recommendations in the Checking Configuration Consistency of Single Container subsection (on page 53).

For parameters having distinctive barrier and limit, the values of the barrier and the limit are shown on two lines separated by the "/" sign.

All values are given in their "natural units of measurement" (except were the units are explicitly specified). For parameters with names ending in "pages" the natural units of measurement are pages. For other memory parameters (kmemsize, dcachesize, and all socket buffers) the units are bytes. For the remaining parameters (such as numproc) the units are items. The natural units of measurement are the units in which the values are accepted by the vzctl utility and stored in the Container configuration file.

Name	A	B	C	D
Power	2GB/400	2GB/120	2GB/8	2GB/1
Helper values				
total mem	5056KB	16256KB	248MB	1984MB
kernel mem	1606KB	3968KB	42304KB	300MB

user mem	3450KB	12288KB	200MB	1684MB
avnumproc	15	40	200	
Primary parameters				
numproc	40	65	400	32,000
numtcpsock	40	80	500	2,147,483,647
numothersock	40	80	500	2,147,483,647
vmguarpages	1,725	6,144	102,400	0
Secondary parameters				
kmemsize	870,400/ 923,648	2,457,600/ 2,621,440	16,384,000/ 18,022,400	2,147,483,647/ 2,147,483,647
tcpsndbuf	159,744/ 262,144	319,488/ 524,288	5,365,760/ 10,485,760	402,653,184/ 402,653,184
tcprcvbuf	159,744/ 262,144	319,488/ 524,288	5,365,760/ 10,485,760	402,653,184/ 402,653,184
othersockbuf	61,440/ 163,840	122,880/ 327,680	1,503,232/ 4,063,232	2,147,483,647/ 2,147,483,647
dgramrcvbuf	32,768	65,536	262,144	2,147,483,647
oomguarpages	1,725	6,144	102,400	0
Auxiliary parameters				
lockedpages	4	32	4,096	2,147,483,647
shmpages	512	8,192	131,072	2,147,483,647
privvmpages	3,840/4,224	22,528/24,576	262,144/ 292,912	2,147,483,647/ 2,147,483,647
numfile	512	1,280	8,192	2,147,483,647
numflock	50/60	100/110	200/220	1,000/ 2,147,483,647
numpty	4	16	64	2,147,483,647
numsiginfo	256	256	512	1,024
dcachesize	196,608/ 202,752	524,288/ 548,864	4,194,304/ 4,317,184	2,147,483,647/ 2,147,483,647

Examples Explanation

Example A is the configuration of the most "light" Container. It can have 15 processes on average and up to 40 network connections. This configuration allows running a simple Web server, handling static and dynamic pages produced by simple scripts. This Container is also accessible over `ssh` and `ftp`. Configurations of apache and FTP server software in this Container must be adjusted to reduce the number of spawned processes and the memory consumption.

Note: Not all operating system templates are suitable for such light Containers. Containers based on Red Hat Linux 7.3 template work well with these resource assignments, but Red Hat Linux 8.0 and SuSE Linux 8.0 are not suitable for these light Containers. Applications in 8.0 Red Hat and SuSE distributions require considerably more resources than in Red Hat Linux 7.3 and earlier - for example, Apache Web server in the default configuration from the Virtuozzo template needs about 6MB of memory on Red Hat Linux 7.3 and about 15MB on Red Hat Linux 8.0.

A server with 2GB of RAM can run up to 400 of Containers with the Example A resource configuration.

Example B is a configuration for a not "heavy" and not very loaded server. It can be a dynamic Web server, mail, FTP or DNS server (but not a combination of them). The configuration assumes 40 processes on average and up to 80 network connections.

A server with 2GB of RAM can run up to 120 of such Containers.

Example C is a configuration for a "heavy" server: Web application server with a database back-end or any other server consuming a considerable amount of memory and other resources. Mail and FTP servers having up to 200 simultaneous clients can also work with this configuration. The configuration is designed for 200 processes on average, up to 500 network connections and about 250MB of RAM for each Container.

Example D is a configuration for 1 Container on a server. This configuration makes a Container very close to a stand-alone Linux system from the point of view of resource control. Most resource limitations introduced by Virtuozzo resource control are inhibited in this resource configuration.

Caution! It is not a safe configuration. Like a stand-alone Linux system, a system with such Container can hang if applications consume too much memory. Usually, it isn't a security problem, because it is a configuration for only 1 Container on a server. However, to make the configuration more robust and protect the system from silent hangs, `numproc`, `kmemsize`, and other parameters should be limited to lower values. For example, a configuration produced by scaling the configuration of *Example C* with the factor of 4 is a safe configuration.

Not Specified Values

For compatibility with future versions, the limit components of the `vmguarpages`, `oomguarpages`, and `physpages` parameters should be set to 2,147,483,647. The barrier of the `physpages` parameter should be set to 0.

Index

A

About This Guide • 5
Adjusting Individual Resource Management Parameters • 24
Advanced Management of System Resources • 29
Allocated Memory • 59
Assigning CPU Power to Host System Processes • 27
Automatic Checks • 64
Automatic Validation • 21
Auxiliary Parameters • 42

B

Basic Management of System Resources • 9
Basics of Resource Control • 9

C

Changes Between 2.0.2 and 2.5 • 69
Changes Between Virtuozzo 2.5 and 2.5.1 • 68
Changes Between Virtuozzo 2.5.1 and 2.6 • 68
Changes Between Virtuozzo 2.6 and 3.0 • 68
Changes From Previous Releases • 68
Checking Application Correct Operation • 52
Checking Application Start • 49
Checking Configuration Consistency of Single Container • 53
Checking Resource Control Settings for Applications • 48
Checking Resource Usage and Settings With Advanced Utilities • 61
Checking Resource Usage of Single Container • 22
Checking Resource Utilization and Settings for Whole System • 63
Checking Resource Utilization and Settings of Single Container • 61
Checking Resources Configuration and Real Usage • 21
Checking Settings • 28
Checking System Overcommitment • 64
Checking Usage and Distribution of System Resources and Validating Settings of All Containers • 55
Configuring and Inspecting CPU Time Consumption Control Settings • 26
Configuring CPU Time Consumption Control Parameters • 26
Configuring Resource Management Parameters for Containers • 15
Controlling CPU Time Consumption • 24
Controlling Resource Information in proc • 46
Creating Configuration for Fraction on Server • 18

D

dcachesize • 44
dgramrcvbuf • 39
Documentation Conventions • 6

E

Examples • 71
Examples Explanation • 73
Examples of System Resources Parameter Settings • 71
Explaining vzcalc(8) Output • 61

F

Feedback • 8

G

General Conventions • 8

I

Inspecting Resource Control Settings • 45

K

kmemsize • 36

L

lockedpages • 42
Low Memory • 56

M

Memory and Swap Space • 58
Monitoring Resources • 65

N

Not Specified Values • 74
numfile, numflock, numpty, and numsiginfo • 44
numiptent • 45

numothersock • 34
numproc • 33
numtcpsock • 34

O

oomguarpages • 40
Organization of This Guide • 6
Other Existing Resource Control Mechanisms • 11
othersockbuf • 38
Overview • 31

P

Per-Process Limits • 12
physpages • 43
Preface • 5
Primary Parameters • 33
privvmpages • 41

R

Resource Control and System Security • 10
Resource Control Principles • 11
Resource Management Parameters Overview • 13
Resource Utilization and Commitment Level • 56

S

Scaling Configuration • 17
Secondary Parameters • 36
Shell Prompts in Command Examples • 7
shmpages • 43
Summary • 13
Summary of Configuration Creation Methods • 23
System Resource Control Parameters in Detail • 29
System-Wide Limits • 12

T

tcprcvbuf • 37
tcpsndbuf • 36
Total Memory • 57
Troubleshooting Application Failures Related to Resource Limitations • 51
Types of Virtuozzo CPU Time Consumption Control • 25
Typographical Conventions • 7

U

Using Existing Configurations • 16
Utilities for Checking Resource Configuration and Usage • 23

V

Validating Configuration • 19
Validation Procedure • 20
Verifying and Troubleshooting Resource Control Configuration • 47
View System Information With vzstat • 66
vmguarpages • 35

W

Who Should Read This Guide • 5