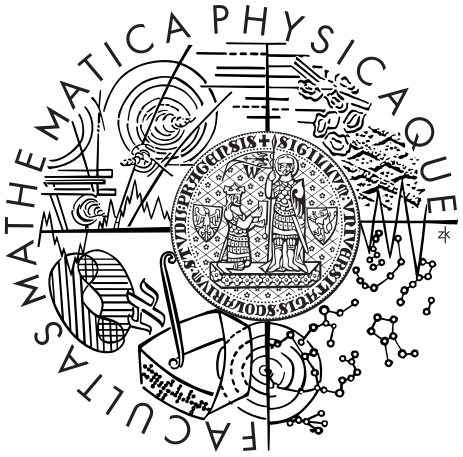
Charles University in Prague
Faculty of Mathematics and Physics

MASTER THESIS



Michal Koutný

System daemon for HelenOS

Department of Distributed and Dependable Systems

Supervisor of the master thesis: Mgr. Martin Děcký, Ph.D.
Study programme: Informatics
Specialization: Software Systems

Prague 2015

I would like to thank Martin Děcký for his help and advice, HelenOS community for their ideas, my parents for support during my studies, my employer for letting me work on the thesis, Jessica McFadden for proofreading and Lennart Poettering for the systemd project.

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In ..... date ..... signature of the author

Název práce: Systémový démon pro HelenOS

Autor: Michal Koutný

Katedra: Katedra distribuovaných a spolehlivých systémů

Vedoucí diplomové práce: Mgr. Martin Děcký, Ph.D., Katedra distribuovaných a spolehlivých systémů

Abstrakt: Přestože je HelenOS operační systém založený na spolupráci nemála služeb, chybí mu jednotné prostředky k jejich ovládání a sledování. Práce nejprve shrnuje přístupy použité pro správu služeb v populárních i mikrokernelových operačních systémech. Dále jsou představeny relevantní detaily v prostředí HelenOSu a následuje rozbor jednotlivých otázek řešených při správě služeb. Získané znalosti jsou pak použity v kontextu HelenOSu a je popsána implementace založená na terminologii systemd. V závěru práce krátce hodnotí výsledek implementace a nastiňuje další myšlenky, které tato implementace otevřela.

Klíčová slova: HelenOS, správa služeb, systemd

Title: System daemon for HelenOS

Author: Michal Koutný

Department: Department of Distributed and Dependable Systems

Supervisor: Mgr. Martin Děcký, Ph.D., Department of Distributed and Dependable Systems

Abstract: HelenOS is an operating system based on a number of cooperating server processes, however it is missing unified means to control and monitor them. The thesis first surveys approaches taken by both popular and microkernel operating systems to the service management. Further it gives a detailed overview of relevant HelenOS environment and continues by the analysis of particular service management issues. The presented knowledge is then applied to HelenOS and the implementation based on the systemd terminology is described. Finally, the thesis shortly assesses the implementation and outlines further ideas that the implementation enabled.

Keywords: HelenOS, service management, systemd

Contents

Introduction

HelenOS environment is missing a system daemon[1] that would give the user a grip on the servers that are running in the system. That begins with control and configuration of the startup process and monitoring state of the servers. Currently, there is only hardcoded starting sequence and no unified notion of a service.

The thesis will survey existing system daemons in various operating systems with emphasis on solution of this problem in microkernel systems.

Then it will explain concepts and mechanisms in the HelenOS operating system that are relevant for the operation of the service manager.

In the next chapter we will analyze needs and problems connected with system daemon more in-depth and we will look how are these particular topics solved by other service managers.

Finally, we will apply the resulting knowledge to design a system daemon for the HelenOS environment and describe its implementation.

Goals

The first goal is to provide a way to capture and resolve dependencies between services and control the booting sequence by that means.

The second goal is to achieve reliable monitoring of the state of the services.

Let's also mention that making the system reliable by automatic restarting of the services is not a goal of this thesis.

[1] System daemon and service manager are used interchangeably in the context of this thesis.

1. Existing service managers

In this chapter we present some existing service managers. The list is not meant to be complete, it should rather illustrate variability of approaches.

1.1 init

There are two major implementations of software that usually go by the name init: BSD style init and System V style init.

1.1.1 BSD init

The operations of the init program in BSD-based operating systems are described here, in particular FreeBSD and OpenBSD were studied and common features are explained (differences were mainly in the interfaces of the programs).

The startup operation is quite simple. It distinguishes between single-user mode, which just starts the superuser's shell connected to the console device, and multi-user mode. When started in multi-user mode it executes the start-up script `/etc/rc` and then maintains terminals for users as defined in the file `/etc/ttys`. This encompasses monitoring for process termination and keeping session database files (`/var/run/wtmp`, `/var/run/utmp`) up to date. [1]

POSIX signals are used to ask init for a system shutdown. When such a request arrives, the shutdown script `/etc/shutdown.rc` is executed and any remaining processes are terminated (first by `SIGTERM` and if they do not terminate within a timeout, `SIGKILL` is sent).

`rc` scripts

The main flow of system initialization is recorded in the `/etc/rc` script. Each service has its control script located in `/etc/rc.d/<service-name>` and can use the common utility functions defined in the `/etc/rc.subs` file.

The order of service starts is hardcoded (in `/etc/rc` in OpenBSD) or it is a topological sort based on partial ordering defined by special comments in service control scripts (FreeBSD). Reverse order is then used when a shutdown is requested. To stop a service, a signal is sent to each process whose execution command matches the given mask (this is all transparently implemented in `/etc/rc.subr`),the same approach is used when checking the state of the service).

History and usage

The BSD style init originates from the Version 6 AT&T UNIX `init` [18] that was released in 1975 [30]. Nowadays, its successors are used in the BSD family of operating systems (FreeBSD, OpenBSD and NetBSD). Since MINIX 3 (Section 1.5) builds upon NetBSD userspace, it uses BSD style init as well.

[1] The process being monitored is `getty` or similar. This feature can also be used to automatically respawn any other process.

1.1.2 System V init

Another flavor of init daemon is the System V init (shortly sysvinit). For reference, we use open source clone (conceived by Miquel van Smoorenburg) which once became widespread across many Linux distributions.

Sysvinit introduces an idea of declarative specification of system services which includes the notion of runlevels.

inittab

Sysvinit stores its configuration in the `/etc/inittab` file that is a line-oriented text file where each line corresponds to a process and various attributes regarding ordering and state control are specified. See sample file (Listing 1) for clarification.

```
# Level to run in
id:2:initdefault:

# Boot-time system configuration/initialization script.
si::sysinit:/etc/init.d/rcS

# What to do in single-user mode.
~:S:wait:/sbin/sulogin

# Control scripts for runlevels
l0:0:wait:/etc/init.d/rc 0
# ...
l6:6:wait:/etc/init.d/rc 6

# Keep virtual consoles in runlevels 2 and 3
1:23:respawn:/sbin/getty 38400 tty1
```

Listing 1: Sample `inittab` configuration

Each record consists of colon-separated columns: identifier, runlevels, action and process. The first field can be ignored for the sake of the explanation, the second field enumerates runlevels for which the record applies and the last field specifies an executable and its potential arguments. The third field *action* is the most interesting, it sets how the process is controlled. Some of the possible values follow:

initdefault It just marks a runlevel as default, process is ignored.

sysinit, boot, bootwait Entries to be executed on system start, i.e. the runlevel field is ignored.

wait, once The process is started when entering specified runlevel(s). With the *wait* variant, sysvinit waits until the process terminates.

respawn Similar to *once*, however, sysvinit restarts the process upon termination.

Symbol	Meaning
`s`, `S`	special runlevel for single-user mode
`0`	runlevel is used before system halt[4]
`1`	runlevel before entering single-user mode
`6`	similar to runlevel `1`, however used before reboot
`2`, `3`, `4`, `5`	main runlevels for administrator's use

Table 1.1: Sysvinit runlevels and their traditional meaning

There exist a variety of other actions which are described in more detail in manual page [17].

Startup sequence and runlevels

The sequence of actions performed by sysvinit on startup corresponds to rows in the `inittab` configuration.

Sysvinit distinguishes two modes of operation (similar to BSD init (Section 1.1.1)): single-user mode and multi-user mode. In single-user mode the only processes started are those with *sysinit* action and `S` runlevel.[2]

When running in multi-user mode, sysvinit also starts processes with *boot* and *bootwait* actions on top of *sysinit*. Consequently, sysvinit enters the default runlevel.

Runlevels represent a logical set of processes (services) for various high-level goals and sysvinit ensures the correct switching between the runlevels. Processes that are present in both the current runlevel and the new one are left untouched, processes not defined for the new runlevel are terminated and processes belonging to the new runlevel only are spawned.

When terminating a process, sysvinit first gives it a chance for graceful termination by sending a `SIGTERM` signal[3] and only after a certain period (5 seconds by default) is an unconditional `SIGKILL` signal sent.

See Table 1.1 for a conventional list of runlevels.

Init scripts

Although sysvinit and its `inittab` configuration provide means for controlled spawning and termination of processes, typical usage involves sysvinit on the coarse level only and actual service management is delegated to scripts.[5]

For each service there is a so called init script that implements start and stop operations for the particular service and may also expose other operations (e.g. restart, configuration reload, state of the service and so on [10]).

In order to specify in what runlevels services should be running, the scripts are referenced by symbolic links from `/etc/rc?.d/` directories (`?` denotes the given runlevel). Names of the links determine a moment how and when a script is run

[2]Sysvinit can even start in single-user mode with very limited configuration without actually reading `inittab` file.

[3]It actually kills whole process group, for groups see Section 3.2.2.

[5]This simple setup is nowadays declining though (in POSIX world). See other systems described in Section 1.2 and Section 1.7.

– a service can be either started or killed when entering the given runlevel and scripts are executed in a given order (see Listing 2).

```
$ ls /etc/rc2.d/
$ K00serviceA    S00serviceB    S10serviceC
```

Listing 2: Example of symbolic links names of init scripts. `serviceA` is not active in runlevel 2, services `serviceB` and `serviceC` are active and `serviceC` should only be started after `serviceB`.

The logic of executing appropriate init scripts is also implemented in a script – `/etc/init.d/rc`. This script (with runlevel as a parameter) is what is stored in the `inittab` entry and what sysvinit controls.

History and usage

Sysvinit was already used in AT&T System V UNIX [28], which dates back to 1983 [29].

Currently, it is used in the Solaris operating system [23] (and its open source derivative Illumos [8]) only in order to launch more advanced SMF (Section 1.2). Sysvinit is also used in the GNU Hurd (Section 1.4.3) where it is a part of the Debian distribution software layer.

Previously, sysvinit was also used by various Linux distributions until it was replaced with another init system (mainly with systemdand for some years also with Upstart)

1.2 Service Management Facility (SMF)

SMF was developed for the Solaris operating system as a successor of sysvinit.

1.2.1 Concepts

SMF formalizes the concept of a service. *Service* is an abstract entity that represents either an indefinitely running application, a device or a group of other services. The actual realization of services are *service instances* that inherit the configuration of a service and it can be overriden for each service instance too.

Various *properties* of a service can be configured, the properties are quite universal and store both persistent and runtime information about a service (instance), furthermore, application-specific data can be stored in dedicated property groups.

One of the service properties are *dependencies*. They can be both positive and negative (`exclude_all`) and the service can depend on another service or a file. Together with dependencies comes a notion of *milestones*, which can be thought of as void services that only specify dependencies.

SMF also defines numerous *states* of service instances (e.g. `offline`, `online`, `disabled`, `degraded`).When we neglect externally caused transitions between states (caused for instance by a service failure), each service is assigned a *restarter*

that is responsible for bringing a service instance to the desired state. The mechanism is extensible since a restarter executes predefined `methods` specified in the configuration. For illustration, the default restarter `svc.startd` requires `start` and `stop` methods and both of them can be configured.

The information about all the services is kept in a *configuration repository* that unifies configuration setting and snapshotting.

1.2.2 Configuration and control

The two main executive processes of SMF are `svc.startd` and `svc.configd`, the main restarter and configuration repository respectively.

From the user perspective, two utilities are of special interest. The first one (`svcadm`) is used to control individual services, typically via executing restarter methods. The second one (`svccfg`) serves to manipulate the configuration repository (e.g. load or archive settings of all services).

1.2.3 History and usage

The first system using SMF by default was Solaris 10 in 2005 [20]. SMF worked well and is still used in Solaris and its open source derivatives.

1.3 Windows services

This section describes what a Windows service is, what the main goals of Service Control Manager (SCM) are and briefly mentions system processes that are related to the services lifecycle.

1.3.1 Structure of Windows service

There exist several types of Windows services: filesystem drivers, (kernel) drivers, and service applications. Metadata about the service such as its type, name, dependencies or path to an executable are stored in the Windows registry.

In the following paragraphs, service applications will be looked at in more detail. The service is not an arbitrary program, it has to stick to a protocol for communication with SCM. Since the process is just a container for multiple services, it must first notify SCM about all the services it can provide, for that purpose the main thread of the process calls `StartServiceCtrlDispatcher` in which it maps service names to their entry points. The library then ensures that the entry point of each service is started in a separate thread. Communication from the SCM to the service is delivered to a handler registered with the `RegisterServiceCtrlHandler` function and in the opposite direction, the service notifies SCM about its state by the `SetServiceStatus` function. The operations that SCM may perform on the service are shown in Table 1.2 and the possible states of a service are shown in Table 1.3.

category	operation	explanation
service lifecycle	pause	service should pause
	continue	paused service should resume
	stop	service should stop
SCM notifications	shutdown	service should clean up, system is shutting down
	paramchange	service should reload its parameters
	interrogate	service should notify its state
network services	...	notifications for network services

Table 1.2: Service operations as specified in handler function documentation [7]

stable active states	started
stable inactive states	stopped
	paused
transitional states	start pending
	stop pending
	continue pending
	pause pending

Table 1.3: Service states as accepted by `SetServiceStatus` function [13]

1.3.2 Service Control Manager (SCM)

Service Control Manager (SCM) is the main component responsible for autostarting, controlling and monitoring Windows services.

It is at startup that SCM has the most work to do, and the following describes what happens. SCM is not the first program started and some drivers are required even before it is started, so the first thing that SCM does is to check whether all the drivers as listed in the registry are started.[6] Consequently, SCM creates a named pipe `\Pipe\Netsvcs` that is used to communicate with the services. Finally, SCM signals its readiness via a `SvcctrlStartEvent_A3752DX` event.

Once SCM is fully operational, it can proceed to start the services marked with an auto-start option. The service can be assigned to an order group, and in the registry `HKLM\System\CurrentControlSet\Control\ServiceGroupOrder` there is a sequence of group names that define the actual order in which the services are started. Services without any group are started last.

Dependency resolution

The dependency of a service can either be another service or the order group, the semantics of the latter is that at least one service from the group must be started.

Dependencies are resolved iteratively in each order group. SCM checks whether the dependencies of a service are satisfied, and if they are, the service is start-

[6]Information both about drivers and application services is stored in the same format under the `HKLM\SYSTEM\CurrentControlSet\Services` registry entry. There are three types of automatic starts: boot-start, system-start and auto-start. Only auto-start services are directly controlled by SCM, the rest is started by I/O Manager and SCM only checks drivers presence in a namespace managed by the system.

ed. If the dependencies refer to a later group (or a service in a later group), it is considered a cyclic dependency and resolution fails. Otherwise, the service is skipped and revisited during the next iteration. Group is considered as resolved when all services were started or cyclic dependency was detected.

Shutdown

The system shutdown is orchestrated by the client/server runtime subsystem (`csrss.exe`), it sends a notification to each process, waits a predefined amount of time for it to exit and if it does not terminate, it is killed. SCM is handled a bit differently – timeout for SCM can be tuned separately and SCM can block the shutdown process. The idea behind this is that a service can negotiate an arbitrary wait time, provided that it regularly responds to SCM – SCM then blocks the shutdown procedure. Analogy to startup dependencies is the order of shutdown notifications stored in the `HKLM\SYSTEM\CurrentControlSet\Control\PreshutdownOrder` registry value.

1.3.3 Other important processes

svchost.exe As already mentioned, multiple Windows services can share a process, `svchost.exe` is a generic container for various services. Each service specifies its group[7] and SCM ensures that only a single `scvhost.exe` is launched and other services are started via a dedicated API to `svchost.exe`.

smss.exe The session manager (`smss.exe`) can be started in multiple instances, one of them is the master session manager – it is the first process started by the kernel after initialization. The master session manager performs some initialization steps (such as checking for disk errors, initializing paging file or preloading some DLLs) and in the end it launches another two intermediate instances of itself (both for interactive and non-interactive session). The main goal of the intermediate session manager is first to start `csrss.exe` and then `winlogon.exe` (for interactive sessions) or `wininit.exe` (for non-interactive sessions).

csrss.exe The client/server runtime subsystem process is vital for the running system. It plays a role in process communication and termination of user sessions, however, it is not that important when considering Windows services.

wininit.exe `wininit.exe` is the main process of the non-interactive session (referred to in the session manager). It also performs some initialization tasks, but what is more relevant is that it starts the service control manager (SCM) together with the local security authority process (LSASS) and the local session manager (LSM).

[7]Group is a parameter of the service image path (e.g. `%SomePath%\svchost.exe -k group-name`), path to the actual library with service implementation is in custom parameters of the service.

winlogon.exe Similarly to `wininit.exe`, `winlogon.exe` is the main process of the interactive session. It manages the interactive login (in cooperation with credential providers and the LSASS) and in the usual case it results in starting a graphical shell (which defaults to `explorer.exe`). Later on, `winlogon.exe` intercepts for the Ctrl+Alt+Del key combination.

services.exe The service control manager (SCM) runs in the `services.exe` process together with other components that support various background tasks.

1.3.4 History and usage

Windows services are a concept of the Windows NT family systems which first appeared in 1993 [25].All derived Windows versions thus use this approach.

1.4 GNU Hurd[8]

GNU Hurd is a microkernel based operating system that is meant to be the GNU project's replacement for UNIX. More precisely, it encompasses the GNU Mach microkernel and a set of servers and protocol specifications which comprise the Hurd itself.

1.4.1 Concepts

The basic entities of the GNU Mach kernel are tasks that are just light envelopes that aggregate resources of a running program (most notably the virtual address space). Tasks communicate by sending messages to each other's ports and Hurd defines the protocols for communication over the ports. For instance, a file is a port that implements the file protocol, typically, such a port would reside on the filesystem server.

In order to achieve universality and extensibility, GNU Hurd makes use of so called translators. Simply put, a translator is a server that consumes one port and provides another with a potentially different protocol. Most prominent are translators bound to files (nodes in the global filesystem namespace).[9] In traditional terms, such translated file nodes can be thought of as mount points. With the distinction that the translator assignment to a file node may be persisted by means of the underlying filesystem (again, in a similar way to traditional symbolic links).[10]

1.4.2 Essential servers

The GNU Hurd environment builds upon a few servers that are necessary for the reasonable operation of the system. These are called essential tasks and they are: root filesystem server, `exec` server, `proc` server and `auth` server.

[8]The content of this section is based on information from the GNU Hurd project website [22], reference manual [21] and source code of the version 0.6.

[9]For instance `ftpfs` that can transparently represent a remote directory.

[10]Interestingly, a translator can be assigned passively to a file node. Then it is only started when it is needed – it is implemented in the `libdiskfs` library.

The root filesystem server uses GNU Hurd's `libdiskfs` library and consequently device drivers that are part of the microkernel.

The `exec` server accomplishes loading of executable files (also scripts with a shebang) and launching them within a new task.

The `proc` server creates abstraction of processes on top of GNU Mach's tasks. It keeps track of processes and their relations (parent-child, process groups).

The `auth` server serves as a trusted mediator when two programs need to authenticate each other.

1.4.3 Startup, monitoring and shutdown

GNU Mach supports multiboot bootloader specifications and thus additional binaries (apart from kernel itself) are present in the memory, ready for being started by the kernel. They consist of the root filesystem server and the `exec` server. The root filesystem plays a principal role in the system startup. This behavior is caused by command line arguments passed by the kernel to the server and it is implemented in the `libdiskfs` library. Since both servers start concurrently, the root filesystem server waits for the `exec`. Afterwards, the root filesystem server executes program `startup`.

The `startup` program continues the startup sequence by starting the `process` and the `auth` server. As already mentioned, those two tasks (together with `exec` and root filesystem) are essential, thus `startup` registers at the kernel to be notified when such tasks terminate. Should this happen unexpectedly, `startup` orchestrates a whole system reboot. In the final stage, `startup` launches a `runsystem.sh` script – in the case of Debian Hurd,[11] this script starts sysvinit (Section 1.1.2) that starts the actual services.

System shutdown is not documented, however, implementation suggests that any task can register at `startup` program to be notified before shutdown. Upon shutdown the `startup` notifies all registered tasks (waits with timeout for a reply to the notification) and only after that are all tasks terminated.

1.5 MINIX 3[12]

Originally, MINIX systems were created for educational purposes. MINIX 3 is the most recent version, which is intended for use outside the academic world. It is based on a microkernel and a set of servers that together comply to POSIX specifications.

1.5.1 Concepts

The architecture of MINIX 3 consists of four layers: kernel, device drivers, server processes and user processes. All parts except the kernel run in the userspace. The kernel ensures scheduling of processes and execution of privileged code. The server processes will be described later, but in brief, they provide the POSIX API

[11] Debian Hurd is a software bundle that allows for the running of the software from the Debian distribution on top of the GNU Hurd infrastructure.

[12] The content of this section is based on the information from "The MINIX book" [26], online documentation in the project wiki [16] and MINIX 3 source code revision 5055c7e.

on top of the microkernel. Device drivers are processes that are similar to server processes, however, they are privileged to interact with I/O. User processes are both user programs and servers in the traditional meaning (e.g. web server).

IPC happens between *endpoints* mainly by the means of message passing.[13] There is one endpoint per process at a time, and the endpoints are identified by their respective process ID combined with generation ID (to prevent PID reuse collisions). The messages of fixed sizes consist of a header and a payload. The payload structure is given at the compile time and is matched according to the header. The messaging API is primarily synchronous and blocking, however, there are also asynchronous and non-blocking variants.

1.5.2 System servers

The process manager (`pm`) ensures process creation and basic monitoring (in terms of POSIX syscalls. They are: `fork`, `waitpid`, `kill` and `exec`). Interestingly, the process manager used to also be partly responsible for the virtual memory,[14] now the burden has moved to the separated `vm` server.

The virtual filesystem server (`vfs`) is another necessary system server. It serves as a proxy to particular filesystem servers and unifies access to them. It also tightly cooperates with the process manager when necessary (e.g. implementation of the `setuid` bit). The role of the `vfs` implies that the root filesystem server (e.g. MINIX filesystem server `mfs`) is also a system server.

The reincarnation server `rs`[15] is of the most concern to this thesis. It monitors other servers and ensures that they are restarted should there be any problem detected. Furthermore, it is also used for manual control of the considered servers. Details about the operation of the `rs` will be described later.

The last important system server is the `init`, which is actually an "ordinary" BSD style init (Section 1.1.1) that oversees the user servers and brings up most of the userspace.

In addition to the aforementioned servers there are several others that are vital for the whole system. They are: `vm` (the server that controls virtual memory and paging), `ds` (the data store server that stores generic data for others), `tty` (for handling input and output to the terminal) and `pfs` (pipe filesystem that implements POSIX pipes).

1.5.3 Startup, monitoring, shutdown

The first program to run is a NetBSD bootloader, which is a multiboot compliant bootloader ([11]) that loads the kernel image and the images of the other programs as well. Some of the programs are part of the kernel (e.g. clock task) and the rest are the system servers (Section 1.5.2). The kernel first starts just the virtual memory server (`vm`) and the resurrection server (`rs`), other system servers are

[13] Larger data transfers are performed via so called memory grants, which is in principle memory sharing between endpoints.

[14] MINIX used continuous areas of the physical memory for the whole virtual memory of a process. The process manager thus had to find an appropriate "hole" to fit the process' requirements.

[15] Sometimes the abbreviation is explained as a resurrection server

loaded, however, they are not scheduled yet. Since the `rs` knows what system servers exist, it passes this information to the `vm` server so that it can prepare appropriate paging structures. Only after that are the remaining system servers scheduled, and the startup proceeds with initialization of the user servers under supervision of the BSD style init (Section 1.1.1).

The resurrection service monitors not only the presence of other services' processes, but also their liveness. System servers register upon start to the resurrection service with a special message (`RS_INIT`) and are then expected to send heartbeat messages in response to the resurrection server's ping requests (this is implemented in the layer called System Events Framework (SEF)). When the resurrection service concludes that the monitored system server is dead (terminates unexpectedly, is stuck in a deadlock), it attempts to restart it. There is a special version of `fork` call (`srv_fork` that is implemented inside the process manager server) that gives the resurrection service more powers when initializing the new process – interestingly, one of them is the usage of an in-memory executable image in case the filesystem is not working.

System shutdown is mostly controlled by the user servers. The shutdown is initiated from the `shutdown` utility that consequently executes `/etc/rc.shutdown` script. The script basically does two important things: a) notifies the resurrection service that the system is about to exit, b) sends `SIGTERM` to the `init` server, which in turn kills all processes. The notification to the resurrection service prevents restarting of killed system servers.

1.6 Upstart

Upstart is quite an exception among service managers. Basically it is an event framework meant to replace the sysvinit. The services are started as a reaction to events and they also emit events for others to react. Interestingly, the framework allows handling compound events, e.g. service starts only when multiple events have been emitted.

More information about Upstart is also in sections 3.4.4 and 3.5.4.

1.6.1 History and usage

Upstart originated as a service manager for Ubuntu distribution, where it was a default service manager until Ubuntu 15.04 when it was superseded by systemd. Chrome OS uses Upstart as of 2015.

1.7 systemd

systemd is an init daemon for Linux operating systems. Its basic concept is a unit, which is a polymorphic object that serves to control and configure the system.

More information about systemd can be found in sections 3.4.5 and 3.5.3 and also in Section 3.2.3 since systemd makes heavy use of cgroups.

1.7.1 systemd units

Here is a description of some of the most important systemd unit types based on [19].

Service Running process or group of processes.

Target Plain unit that only has dependencies.

Mount A mountpoint controlled by systemd.

Socket An endpoint of a service, its independent of the service, can be used for socket activation.

Timer Unit that is activated at specified time/period only.

Scope Basically container for processes of anonymous services.

Slice Wrapper around cgroups so that you it affect multiple services in terms of resource control.

1.7.2 Dependency types

There are multiple types of dependencies between units in systemd.

Requires Unit A is required for a unit B. If B is started, then A is started as well. If A fails to start, B fails as well.

Wants Similar to Requires, however, dependent unit is not affect when dependency fails.

Conflicts Bidirectional dependency, prevents two units to be both active at a time.

After Not a dependency (i.e. doesn't propagate activation) but when both units are to be started, it specifies the order.

2. HelenOS architecture

The goal of this section is to provide insight into HelenOS internals that are necessary to understand when reasoning about service manager. The description is valid for version mainline,2352.

2.1 HelenOS IPC

Inter process communication in HelenOS is based on messaging which has the form of method calls (they can be seen as methods of a server). The two most important types of messages are[1] requests and answers.

Message Every message consists of six integer fields (their width depends on the architecture). The first field is handled distinctively – for requests it represents a method number and for answers it is a return value. The content of the remaining five fields is an application specific payload. Please note that the payload may also be attached to the answer message, which contrasts with traditional methods returning single values.

Before explaining the detailed semantics of the methods, it is necessary to understand phones and answerboxes.

Answerbox An answerbox is a buffer for incoming messages and it is transparently managed by the kernel. Every task has one dedicated answerbox and the userspace retrieves messages from it via system calls.

Phone A phone is a means for sending messages. A correctly initialized phone is connected to a particular answerbox and messages cannot be sent anywhere else but the answerbox. That implies that the cardinality of phone-task relation is multiple phones per task. A phone can be created by special (IPC) methods (Section 2.1.1) and userspace typically refers to phones by their integer descriptors (obtained from the kernel). To call these special methods the process needs to be provided with at least one phone. Each task in HelenOS has such a phone with a well known descriptor (`0`) since birth and it is connected to the naming service.

2.1.1 System IPC methods

In general, semantics of the methods are defined by applications, however there are a couple of methods that convey special meaning to the kernel, which then acts accordingly. These are called system methods (as opposed to user methods) and here follows their desription.

Phone related methods

`IPC_M_CONNECT_ME_TO` This method serves clients to create new phones.

[1] There are also special types of messages used by the kernel to deliver information to userspace. (Section 2.4).

Upon sending such a message, a new phone for the sender is allocated and it is connected to an answerbox of the final receiver. This is a very versatile concept when put together with message routing. Tasks can forward the message according to method arguments without accepting it.

The first three method arguments `arg1`, `arg2`, `arg3` are application specific. The argument `arg4` is used by the async framework (Section 2.1.3) to transfer IPC flags (see Section 4.7.1). The argument `arg5` is modified by the kernel to contain information about the new phone. On the receiver's side, there is a unique system-wide phone identifier (phone hash) and when the message is accepted, the answer contains the phone descriptor in the `arg5`.

`IPC_M_CONNECT_TO_ME` This method serves clients to create reverse (callback) connections.

When the receiver accepts the message a new phone is allocated for them and it is connected to the sender's answerbox.

The first arguments `arg1` to `arg4` are application specific and `arg5` carries information about the created phone (the receiver will see the phone descriptor and the answer to the client will contain a phone hash).

`IPC_M_PHONE_HANGUP` This method notifies the receiver about terminating connection and the sender's phone is deallocated. The answer message is discarded.

Data methods

`IPC_M_DATA_WRITE` By calling this method, the sender's buffer is copied to the receiver's address space.

`arg1` is a pointer to a buffer, respectively in the address space of the sender (request) or in the adress space of the receiver (answer). `arg2` is the size of the data to copy (sender side) or to accept (receiver).

`IPC_M_DATA_READ` The sender of this message presents a buffer that is to be filled by a copy of the receiver's data.

`arg1` is a pointer to a buffer, respectively in the address space of the sender (request) or in the adress space of the receiver (answer). `arg2` is the size of the data to accept (sender side) or to copy (receiver).

`IPC_M_SHARE_OUT` This method works in a similar way to `IPC_M_DATA_WRITE` although data are not copied, they are just mapped into the receiver's address space.

Apart from the buffer specified by the arguments `arg1` and `arg2`, the third argument `arg3` is used to pass the flags set for the mapping (e.g. readonly, executable).

`IPC_M_SHARE_IN` A mapping of receivers memory is created in the sender's address space, analogously to `IPC_M_DATA_READ`.

In a request, `arg1` is the maximum accepted size, `arg2` is a custom argument. In a (received[2]) answer, `arg2` are sharing flags and `arg4` is a pointer to the receiver's address space where data were mapped. (`arg1` and `arg3` of the answer are ignored by the sender since they make no sense in its address space.)

Advanced connection methods

`IPC_M_CONNECTION_CLONE` This method creates a new phone for the receiver that is connected to a specified answerbox – an answerbox that the sender is already connected to through a phone.

The request's `arg1` is a phone descriptor to be cloned, and when the receiver obtains the request it is a descriptor of its cloned phone.

`IPC_M_CLONE_ESTABLISH` This is the first method to be sent through a phone cloned by `IPC_M_CONNECTION_CLONE`. When it is received, the hash of the cloned phone is in `arg5` (compare to `IPC_M_CONNECT_ME_TO`).

Special methods

`IPC_M_STATE_CHANGE_AUTHORIZE` This method is used to notify a third task that two tasks agreed they both have a phone connected to that third task. The request's arguments `arg1` to `arg3` are application specific, `arg4` is unused and the `arg5` is a sender's phone descriptor of a phone connected to the third task. The receiver sends a phone descriptor supposedly connected to the same third task in `arg1`. When both the sender and the receiver refer to the same third task, the third task is notified via the kernel event called `EVENT_TASK_STATE_CHANGE` (see Section 2.4).

`IPC_M_DEBUG` This method is used for calls to the separate answerbox of a task. It is used for debugging.

2.1.2 Low level IPC API

Low level IPC API is a set of wrappers of IPC related system calls. The following section describes the existing low level IPC functions and thus gives an insight into which system calls the kernel provides. [3] It is worth stressing that system calls themselves are designed for asynchronous messaging.

The code can access messages (calls) in two ways: indirectly by referring via call identifier (`ipc_callid_t`) and directly through call structure (`ipc_call_t`), which is the case when manipulating incoming messages (both requests and answers).

[2]Be aware that it is the answer *received by the sender*: sender – request – receiver – answer – sender.

[3]It is not important to distinguish here, however, some of the system calls and related functions exist in two versions (fast and slow) depending on the number of message arguments that are actually used.

Messaging functions

`ipc_call_async(phone, method, arg1, ..., arg5, callback, preempt, private)` This function sends a request through the given phone and registers (optional) callback to be called when an answer arrives (providing it with a private context).

The function call is asynchronous and typically non-blocking, although when number of unprocessed messages through a given phone reaches the kernel limit (`IPC_MAY_ASYNC_CALLS`) it may be delayed (this is implemented in userspace).

`ipc_wait_cycle(call, timeout, flags)` This function retrieves a message from the task's answerbox and ensures that registered answer callbacks are invoked. It then returns the call identifier and fills `call` structure (output parameter).

This is the only spot in IPC API where blocking behavior is supposed[4]

`ipc_answer(callid, retval, arg1, ...)` This function sends an answer to a request message identified by a call identifier, see `ipc_wait_cycle`.

Mixed functions

`ipc_forward(callid, phone, method, arg1, ..., arg5, mode)` This function is used when routing request messages, the message is not explicitly processed by a task, it is sent again through the given phone instead.

In the case of user methods (for definition see Section 2.1.1), the forwarding task can fully modify the message (including the method). The system method requests are restricted regarding modifications, so that the forwarding task can issue methods with at most three arguments (since (system) method of the forwarded call cannot be changed, its number is stored in `arg1` and system methods manipulate with phone hash stored in `arg5`).

`ipc_hangup(phone)` This function disconnects the specified phone from its answerbox and releases the phone.

`ipc_poke()` This method wakes up a thread of the calling task that waits in `ipc_wait_cycle` function (invalid call identifier is then of course returned).

Events and notification functions

`ipc_event[_task]_subscribe(event, method)` This method registers a task to receive notification messages when a kernel event occurs (Section 2.4). Users can specify an arbitrary method to distinguish between individual events.

[4]The call can be non-blocking when proper flags are set. Although it is the only blocking function by design, the `ipc_call_async` function can also block due to limited size of buffers.

`ipc_irq_subscribe(inr, devnr, method, code)` This method is similar to `ipc_event_subscribe`, difference is that it registers listeners for hardware interrupts propagated by the kernel.

It is listed just for completeness, to summarize which subsystems can generate messages for a task.

2.1.3 Async framework

The async framework is a layer built on top of the low level IPC API (Section 2.1.2). From the user's point of view it allows for both sticking to the asynchronous nature of IPC messaging and more conventional synchronous communication.[5]

Fibrils

The key concept behind the async framework are fibrils. Fibrils are execution containers within a task similar to threads. Since the Spartan kernel supports classical threads, the term execution container is used as a more generic term covering both fibrils and threads to prevent possible confusion.

Theoretical advantages of fibrils over threads are as follows:

- They can be implemented in userspace only.
- Smaller consumption of resources (e.g. in scheduler).
- Faster context switch (no system call necessary).

The main disadvantage of fibrils is that they don't support preemptive scheduling and all user code must be written with that in mind.

The effectiveness of HelenOS implementation is disputed,[6] however, in general they are considered a cheaper resource and the design of the async framework relies on this.

Async framework entities

Exchange An exchange is the smallest element of communication from the client's perspective. It can be as simple as a single IPC method call, multiple IPC calls are possible as well. What comprises the exchange depends on the particular protocol between the client and the server.

Session Session represents the client's communication context when two tasks communicate in a client-server fashion. The main purpose of sessions is management of exchanges that is specified upon session creation. There are three possible exchange management styles: atomic (all exchanges are supposed to be single-call only), serialize (to avoid interleaving, execution of concurrent exchanges is serialized) and parallel (concurrent exchanges are carried out using parallelism).

[5] In this context, synchronous means that the caller is blocked until an answer to its request arrives.

[6] `http://trac.helenos.org/ticket/552`

Connection Connection represents the server's communication context when client-server communication takes place. Note that it is rather bound to the client's phone and not the client task (i.e. one client can open multiple distinct connections to the server).

Worker fibril A worker fibril is (automatically) created at the server for each connection. This allows for handling multiple clients at once together with blocking calls within the server's code (they would block only the current fibril).

Worker fibrils are also created for handling IPC notifications (kernel events, interrupt notifications). Fibrils that are created explicitly by users themselves are considered to be worker fibrils as well.

Manager fibril Manager fibrils[7] are the core of the async framework. A manager fibril runs an endless loop in which it intercepts all incoming IPC messages. It creates a new worker fibril when a new connection is initiated (upon receipt of `IPC_M_CONNECT_ME_TO` call, `IPC_M_CLONE_ESTABLISH`) or a when a notification arrives. Other messages are routed to a particular connection's queue (distinguished by phone hash).

The cooperative scheduling works as follows. A worker fibril is running its code and when it would block waiting for an IPC message, it is switched to the manager fibril. The manager fibril then checks for other ready worker fibrils and possibly switches to one. Only when there are no worker fibrils ready, does the manager fibril block the whole thread waiting for an incoming message.

Async framework functions

For the sake of simplicity, it can be said that all low level IPC functions described in Section 2.1.2 have their wrappers in the async framework layer. This is mainly because of control over cooperative scheduling and partly because of different abstraction level (sessions instead of phones). Here follows a description of the additional functions of the async framework.

`async_connect_me_to[_blocking](mgmt, exch, arg1, arg2, arg3` This function[8] creates new sessions and it hides forwarding of the `IPC_M_CONNECT_ME_TO` method by a broker task. As explained in Section 2.1.2 such a call can use at most three arguments and the sender must already be connected to a broker (the call is sent within the exchange `exch`).

`async_exchange_begin(sess)`, `async_exchange_end(exch)` These functions are "brackets" that group individual calls into exchanges.

`async_wait_for(async_call, retval)` This is the function that makes IPC calls behave synchronously. Each request call results in `async_call` structure and this function blocks the current fibril until an answer is received.

[7] There might be multiple manager fibrils when task is running multiple threads.

[8] Actually a pair of functions. The difference is only in implicitly passed IPC flags.

`async_req(exch, method, arg1, ..., arg5, retval, rarg1, ..., rarg5)` This is just a syntactic sugar that compounds sending a request message and waiting for the answer and extraction of answer arguments.

`async_connect_to_me(exch, arg1, ..., arg3, callback, private)` This function invites a server to connect back to a client. As for other connections, a new fibril is spawned and `callback` is executed within it (with `private` context).

2.2 New task creation

2.2.1 Tasks created by kernel

The kernel itself can start tasks whose program image is already present in the memory. The purpose of this is to start first userspace processes after kernel initialization (for details see Section 2.6.3).

2.2.2 Tasks created by userspace

There is the only way that a userspace task can create another task – execute the `SYS_PROGRAM_SPAWN_LOADER` system call. As the name suggests, the kernel creates and starts a new loader task from the well-known program image. The loader image is stored in the kernel since the boot, then the loader is processed differently to other boot time tasks because of the mark in the `PT_INTERP` ELF header. The loader program image is not started, it is stored instead and is only started later via `SYS_PROGRAM_SPAWN_LOADER` system call.

In reality, ordinary tasks do not use the `SYS_PROGRAM_SPAWN_LOADER` system call directly, they just request to be connected to it via the naming service. The naming service then spawns the new loader instance and enables connection to it from the originating task.

The communication between the `task_spawn` caller and the loader goes as follows:

- Task A requests connection to the loader via the naming service.
- The naming service handles such requests specially and spawns the loader.
- The first thing done by the spawned loader is the registration at the naming service as an ordinary server (through its initial phone connected to the naming service, the phone exists until the spawned task terminates). The naming service matches the new instance with the appropriate connection request.
- Task A is connected to the loader instance.
- Once the session with loader is opened, instructions defining spawned task are sent to it (executable filename, opened files and so on). Both task A and the loader communicate in the same way as any other two tasks.
- (At this stage dynamically linked libraries might be loaded.)

- Finally, the task A asks the loader to start the new program (simply by jumping to its entry point) and hangs up its phone to the loader.

2.3 Task monitoring and control

This section describes how can tasks interact between each other without using any explicit client-server protocol. [9]

2.3.1 Task monitoring

A way of task monitoring are the `task_wait*` group of functions. They are used when a task needs to be blocked until another task provides a return value. Note that in the context of HelenOS, returning a value from a task does not imply that the task has terminated – it may just notify the waiting task that a server successfully started and is running until it is killed. On the other hand, a task may terminate even without a return value (e.g. due to an unexpected error), the `task_wait` will return in such a case (it cannot block forever), however, the return value of the task is undefined.

It is worth mentioning that there are two variants: `task_wait_task_id` and `task_wait`. One may be satisfied with the former one only, however, there would be a race condition when you wanted to wait for a task you spawned and it terminated earlier than your `task_wait_task_id` call.[10] That is the purpose of the `task_wait` function that does not take the task ID, but rather a structure initialized by the `task_spawn` call that ensures the correctness.

The waiting mechanism is implemented in cooperation with the naming service. Remember that each task was born as a loader (see Section 2.2) that kept a phone connected to the naming service. The naming service can thus detect task termination by receiving an `IPC_M_HANGUP` call (details in Section 2.1.1). [11]

2.3.2 Task control

A task can control another task in a very limited manner – it can (forcefully) request its termination by `SYS_TASK_KILL` system call (it takes the task ID as an argument). This will cause unconditional termination of all the task's threads and the kernel to release its resources. As for related resources in userspace servers, it is up to their implementation – they will receive the `IPC_M_HANGUP` message.

2.4 Kernel events

Kernel events are a mechanism by which the kernel can notify userspace about various events. They are complementary to system calls and similar to IRQ notifications – they are both delivered as IPC messages. In contrast with IRQ

[9] Some information related to task monitoring is stored in system info, which is a generic database maintained by the kernel and accessible for userspace in read only mode.

[10] Not mentioning that due to task ID reuse the caller may even wait for a totally different task.

[11] This mechanism can yield in false positives when a misbehaving task would circumvent the standard library and would hangup the phone prematurely.

notifications, the kernel events can only be raised by the kernel itself (not by a peripherial).

There are two event namespaces: task namespace and global namespace.

Events from a task namespace are specific for a particular task and only the task itself can subscribe to them. There exists only one task kernel event `EVENT_TASK_STATE_CHANGE`, its semantics are explained in Section 2.1.1.

Global namespace contains events that are self describing, no task context is necessary. For illustration, the names of existing global events are: `EVENT_KIO`, `EVENT_KCONSOLE`, `EVENT_FAULT`, `EVENT_KLOG`. The most important event in the scope of this thesis is `EVENT_FAULT` that occurs when the task has to be terminated by the kernel due to a processor exception (e.g. accessing an invalid address).

Be aware that only a single task can subscribe to a given global event (per type).

2.5 Basic servers

From the Spartan kernel's point of view all tasks are equal. From the IPC point of view, some tasks are servers, some are clients and some are both. And from a functional point of view, some servers are more important than others – they are critical for any advanced system operations (i.e. everything above functions provided by the microkernel). These fundamental servers are described below.

2.5.1 ns (naming service)

The naming service serves two purposes: it is the first brokerfor every task and implements task monitoring API (Section 2.3.1).

The names of servers registered at the naming service are just simple well-known integers so that they can be passed in ordinary IPC messages.

The task monitoring exploits the fact that the naming service is a little bit different from the kernel's perspective – when a new task is spawned (more in Section 2.2) it is created with a phone connected to the naming service.

Interestingly, the naming service is the only HelenOS server that is not implemented using the async framework (Section 2.1.3), it uses low level IPC API (Section 2.1.2) directly. It is mainly for historical purposes and partly due to less resource consumption and better robustness.

The HelenOS standard library provides naming service communication API both for servers (`service_register`) and clients (`service_connect[_blocking]`).

2.5.2 vfs (virtual filesystem server)

This server is an entry point for all filesystem related operations. It manages the filesystem namespace in the form of a rooted tree where subtrees are handled by specialized filesystem servers – this is the result of `mount` operations.

The client code for vfs is implemented in the standard library and it first communicates with the vfs server. According to the given path, the vfs will find the responsible filesystem server (known since the call to the `mount` operation) and data-intensive IPC messages are forwarded from the original client to the filesystem server. The filesystem server then communicates with a block device

server to obtain/manipulate raw data (it depends on filesystem implementation, it is not the case for "virtual" or in memory filesystems.).

The virtual filesystem server is vital for the system operation because without it program executable images are not available.

2.5.3 locsrv (location service)

The purpose of the location service is to allow clients to find and connect to servers. It is similar to what the naming service (Section 2.5.1) does, however, the location service doesn't monitor the tasks in any way and its API is more comfortable.

The location service also has a more advanced model of naming. Each server can register itself under multiple names – services. The names are strings (as opposed to the naming service's integers) and they can be nested within namespaces (the namespaces don't nest, i.e. there is a one-level hierarchy).

The direct naming is complemented by registering services into categories. There can be multiple services in the same category, categories can be enumerated and location service clients can register a handler for category membership updates.

The services registered at the location service can be also accessed as files in a virtual `locfs` filesystem (directory structure matches the namespaces).

2.5.4 rd (RAM disk)

This server creates a block device (which is just a service exposed at the location service supporting specific communication protocol) that operates over a memory range. In this case, it is memory where the data with the root filesystem image were loaded by the bootloader – the kernel makes information about the memory range available to the userspace via sysinfo.

2.5.5 devman (device manager)

The device manager is a central component of the device driver framework, which is described in detail in [27]. Here is a brief description of just the relevant parts only.

The core data structure of the device manager is the device tree. It contains nodes of two types: devices and functions. The meaning is that devices provide functions and functions can be subordinate devices. The leaf functions then represent services for other tasks that are exposed via the standard location service API (this registration is carried out by the device manager, not the driver itself).

The role of the drivers in the device tree terms is that they take in a device node, perform appropriate operations for the given device and output function nodes. Similarly, drivers are used when the tree is disassembled to properly remove function nodes from the devices.

The device manager ensures that the correct drivers are started when they are needed. For this purpose it keeps a database of known drivers which contains information on how to match drivers to devices, and a path to the driver's ex-

ecutable.[12] The database is filled during the initialization of the devman when the configuration files are loaded.

The device manager is also a broker for the driver servers. It allows:

- Communication of a driver to its parent device driver.
- Direct communication of clients with drivers, when clients refer to drivers by the path of the respective device in the device tree.
- (Indirect) communication of clients to drivers via their service names exposed in the location service. In this scenario, the call from a client is routed through both the location service and the device manager to the target driver.

2.5.6 Filesystem servers

Filesystem servers are servers that register themselves at the virtual file system server (Section 2.5.2) and implment the required functions so that the vfs can delegate filesystem operations to them. One filesystem server can handle multiple mount points (with different contents) of its type.[13]

HelenOS provides a library *libfs* to the programmer so that filesystem server implementation is easier. Currently, the following filesystems are supported: cdfs, locfs (see location service, Section 2.5.3), tmpfs (in-memory filesystem), exfat, fat, mfs, ext4fs, udf.

2.5.7 Device drivers

There are two kinds of device drivers due to historical reasons. The newer ones that utilize the device driver framework as described in Section 2.5.5 and older ones that do not. However, the older model drivers can be thought of as standalone servers that expose their API via a broker (mostly the naming service). They have no relation to the device manager.

In a similar way to filesystem servers, HelenOS provides a *libdrv* library that facilitates writing drivers for the device driver framework.

2.6 HelenOS start process

This sections breaks down what happens from the machine start to the moment the HelenOS is usable by a user. It is focused on the IA-32 architecture.

2.6.1 Bootloader

Generally, the Spartan kernel supports any multiboot compatible bootloader [11]. This interface specifies that the bootloader (as well as loading and jumping into

[12] Each driver has a configuration `.ma` file which contains the match information, the name of the configuration file then implicitly specifies the name of the executable as well.

[13] vfs would not even allow to register multiple servers of the same filesystem type. However, you can tag a filesystem server with an instance number and then even the servers of the same filesystem type are considered distinct by the vfs.

the kernel) can also load modules[14] and pass information about them to the kernel.

In case of the GRUB 2 bootloader, the kernel and modules to be loaded are all stored on the boot partition (it can be any filesystem on any device that is supported by GRUB 2). The modules are executables of boot time servers, `init` program and raw image of the root filesystem.

2.6.2 Kernel initialization

The kernel first initializes its internal structures and subsystems (most importantly: kernel console logging, memory management, scheduling) and starts a kernel thread that continues with processing the data from the bootloader.

The last module is supposed to be the RAM disk (Section 2.5.4) and thus the kernel stores its location to the sysinfo database, so that the `rd` server can map the memory and make it accessible to other userspace tasks.

The remaining modules are expected to be ELF binaries that the kernel can load on its own and launch them (they are referred to as boot time tasks). However, not all binaries are handled equally – the binary that is detected to be a loader (thanks to flags in `PT_INTERP` ELF header) is not started, its image is stored instead so that it can be used as a template program for `SYS_SPAWN_LOADER` system call (see Section 2.2).

The program that is first in the modules list is also handled specially by the kernel – all tasks (both boot time and those spawned by `SYS_SPAWN_LOADER` system call) are created with an initial phone that is connected to this first task.

2.6.3 Boot time tasks

In contrast with macrokernel systems where kernel initialization ends with the start of a single userspace program, HelenOS starts multiple tasks concurrently, and their proper synchronization is thus important – it is achieved implicitly by blocking some operations on IPC messaging.

A list of boot time tasks and their roles during system start is below. Note that the group of boot time tasks greatly overlaps with basic servers, so more details can be found in Section 2.5.

Naming service The destination of initial phones of all tasks, together with message routing is vital for establishing communication between boot time tasks. It also synchronizes tasks by holding messages until their recipient is ready to accept them.

Location service A similar principle as the naming service. It is necessary for exposition of the `bd/initrd` service that is needed to mount the root filesystem.

RAM disk The RAM disk server makes available data from the image of the root filesystem.

[14] Not to be confused with any other modules, in this context it is just binary data.

Virtual filesystem server This is the server where root filesystem mounting is requested, it coordinates communication of all parties involved (client, RAM disk server and particular filesystem server). Due to the communication arrangement it also helps to synchronize the boot time tasks execution.

Filesystem server A filesystem server for the appropriate type of filesystem must be running so that data for the root filesystem stored at the RAM disk can be interpreted.

Init program The role of the init program is explained in more detail in Section 2.6.4.

Logger The logger is a server that provides a unified interface for other tasks to log their operations. However, none of the other boot time tasks use this logging[15] and the `logger` server could possibly be omitted from boot time tasks and be started later by the init program.[16]

2.6.4 Init and rest of userspace

The `init` program ensures that all the userspace tasks that are expected by the user are started and also that their dependencies are fulfilled. The mounted filesystem (at least the root filesystem) is a kind of dependency as well and the mount operations are issued by the `init` program.

The order in which the `init` executes the necessary steps is hardcoded and relies on blocking calls when necessary (e.g. mounting the filesystems).

It is worth noting that the same version of the `init` program is used across various HelenOS build configurations – it simply includes the union of all possible services and it skips any errors. The RAM disk building scripts then effectively decide what services will be started by (not)placing them to the RAM disk image.

[15]It is mostly for historical reasons, since the `logger` server's API is younger than the rest of boot time tasks. Furthermore, it would lead to cyclic start dependencies when the `logger` server would need the services of the `vfs` and the `vfs` would require the `logger` as well.

[16]Inclusion of the `logger` among boot time tasks allows setting its command line arguments via the multiboot module command line that can be edited during boot. The `init` program is more rigid and does not permit such modifications.

3. Analysis

In this chapter, we analyze various aspects important for the service manager. And where it is relevant we look on existing implementation in more detail than in Chapter 1.

3.1 System startup

In the scope of this thesis, system startup is a process that begins after the kernel initialization by launching the first userspace code and ends when the machine is ready for its purpose – be it login screens on desktops or a set of various (network) services on servers.

Effectively, it means populating the memory both with proper code (i.e. starting processes) and customizable data (i.e. apply various configurations and settings). This process needs orchestration by one or multiple cooperating programs.

3.1.1 Early userspace

Early userspace is an approach known mainly from Linux environments. Its main idea revolves around the problem of mounting the root filesystem (which is a task the Linux kernel is supposed to do before passing control to userspace, however, we will see that it relates also to the HelenOS startup process, Section 4.2). There are far too many possible ways to mount the root filesystem to be handled by the kernel (such as various filesystems, encrypted device, network filesystem or an atypical device requiring special initialization). To address this issue, the startup process is split into two distinct phases: the early userspace and the ordinary userspace. The early userspace is initialized with a memory-based[1] root filesystem that the kernel can always mount and this root filesystem contains everything necessary to mount the (future) root filesystem of the second phase. The system administrator is provided with utilities[2] that allows them to relatively easily modify the contents of the initial root filesystem – much better than including all that into the kernel.

The to-be root filesystem is mounted in a subdirectory of the early userspace root filesystem and when everything is ready for the second phase, Linux-specific `pivot_root` system call is invoked, which effectively replaces the subtree mounted in `/` with the to-be root filesystem.

It varies among system daemons whether the same `init` program handles both the early userspace and the ordinary userspace or a cascade of multiple programs is used (first `init` is just a shell script and the second fully fledged `init` resides on the actual root filesystem).

[1] Note on terminology – originally Linux used so called *initrd* (RAM disk) which was a block device backed by memory which was mounted with contained image of an ordinary filesystem. Despite nice separation of concerns, it lead to storing the same data in the memory twice (once in the original device and the second copy was in buffers/caches for the mounted filesystem. Thus the concept was enhanced to *initramfs*, which eliminated the block device and data were stored only once in the form of the aforementioned cache.

[2] It depends on particular distribution, however, the system should at least offer `mkinitrd` or `mkinitramfs`.

3.1.2 Metrics watched

System startup is quite a prominent manifestation of system daemon both for users and administrators, and so system daemon qualities are compared during startup using various metrics.

Time The shorter, the better. That holds both for desktops (less annoying) and servers (less downtime).

The startup process consists mainly of loading data into the memory and initialization of hardware, it is therefore more I/O bound rather than CPU bound, i.e. naïve parallelization will not affect overall time that much. However, not all tasks during startup have to be ordered with respect to each other, independent tasks can run concurrently.[3] Some kind of relaxation of the sequential ordering was thus adopted by system daemons: either explicit declaration of dependencies (LSB init scripts, SMF, systemd) or concurrent event framework (Upstart).

An interesting approach is used in Windows systems (Section 1.2) where the start of some services is postponed a certain amount of time (delayed autostart) if they are unnecessary to display the login dialog.

Further efforts to shorten system startup in Linux systems can be illustrated by the usage of `readahead` system call that reduces the time of loading of chosen files related to system start (e.g. configuration files).

Maintenance Well defined extension points are a criteria of easy maintenance (i.e. how can a task be added to the startup sequence). From this point of view, sequential ordering of traditional init scripts is most comfortable due to explicit positions in the sequence. Event framework is on the other end of the spectrum, since actual order is obscured by concurrently running events. Dependency based systems are in between – they provide an idea of dependencies though actual execution is also concurrent.

Wasted PIDs An interesting metric looking at PID of the first process started by a user [12]. The rationale is simple – processes are expensive resources and fewer resources used lead to a swifter startup. This affects especially the traditional init scripts that run multiple subprocesses and favors integrated binaries (trading off some universality).

3.2 Process monitoring

Naturally, the kernel has ultimate knowledge of each existing process, its state and (asynchronous) events that occur during process execution. Not surprisingly, this information is of great use for the system daemon. The following section is an overview of various existing APIs the kernel offers to provide the userspace with this knowledge. kernel the knowledge to the userspace.

[3] Note on terminology: Concurrent tasks can run on a single processor thanks to preemptive scheduling, parallel tasks run physically at the same time.

3.2.1 PID files

Using PID files is the simplest method for service process monitoring. The process identifier of the currently executing service process is written to a well-known file (for the service) and the file is to be deleted when the service terminates.

Although this method is straightforward, PID files have drawbacks too.

Firstly, just simply checking PID files on service startup may lead to race conditions and problems when ensuring only one instance of service is running. Therefore, some other fuse is necessary, for instance checking processes running the same executable or processes running the same command line – naturally this needs support from the underlying operating system.

Secondly, special care has to be taken to always remove the PID file of the terminating service, especially when it is not expected. Stale PID files may prevent (re)start of the service or even worse interfere with unrelated processes (due to PID reuse).

3.2.2 POSIX process groups

The rationale behind the existence of process groups [14] is process multicast (via signals) and access control to a terminal.

The processes are aggregated into *groups* and multiple groups from a *session*. Each group specifies a principal process – *process group leader*.

The session is then associated with zero or one controlling terminal (and controlling terminal has exactly one session). One of the group within a session can be a *foreground group*, signals issued by the controlling terminal are delivered to all processes of the foreground group and the foreground group leader process is allowed to read data from the terminal.

The mechanism of signal delivery to whole group is not necessarily bound to the foreground group only and can be used generically to send signals to arbitrary process group.[4] Similarly to processes, the groups are referenced by their numbers (process group identifiers).

From a service management perspective it is important to know that child processes inherit teh parent's process group, though new groups can be created by `setpgid` system call.[5]

3.2.3 Control groups (cgroups)

Control groups [5] (widely referred to as cgroups) are Linux kernel mechanisms for generic labelling of processes that can be utilized by various kernel subsystems.

The data model of cgroups is quite generic. The top data structure is a *hierarchy* – it represents a tree of cgroups. There are nested cgroups or processes in a cgroup.[6] A process can be only in one cgroup per hierarchy. However, as mentioned at the beginning of the section, cgroups can act as labels, which means

[4]Unless it is restricted by permissions of the user.

[5]The new group is always created in the session of the calling process. Alternatively, a new process group can be created together with a new session by `setsid` system call.

[6]Although the basic entity for control groups is a thread, but for the sake of simplicity only processes can be considered. The cgroups actually allows group operation on threads, thus transparently working with whole processes.

that a process can be a member of multiple cgroups as long as they belong to different hierarchies. The kernel subsystem can then use a particular hierarchy for its purposes (namely resource control).

When a new process is created (by calling `fork`), it inherits all cgroup memberships from its parent.

Controllers

A controller is a kernel subsystem attached to a hierarchy that somehow makes use of the process partitioning and hierarchy.

Block IO controller The controller allows throttling group's IO rate on a particular device or it can set weights for proportional division of available bandwidth.

CPU accounting controller This is a read-only controller that provides various CPU time usage statistics for a cgroup.

CPU sets controller This controller is another interface to the scheduler's features, that allows for specifying on which processors the code should be running. Sets of CPUs and their properties are nicely mapped onto cgroups (and their attribute files).

Process number controller The controller limits the number of processes that may exist in a cgroup. It is enforced by failing `fork` system call when the given count is reached for the caller's cgroup.

Memory resource controller Memory resource controller provides both information about memory usage and knobs to control its management. Most obviously, it can restrict the size of available memory for a cgroup.

Other controllers The list of controllers above is not complete and there are a few other controllers that modify behavior of the networking subsystem or access to devices.

API

The control groups implementation does not add any new system calls, all its functionalities are exposed via special `cgroup` filesystem. There is one mount for each hierarchy and the cgroup controller is specified as a mount option.

Directories under the mountpoint represent cgroups of the hierarchy and the special file `cgroup.procs` lists processes in the given group. New processes are added to the cgroup by writing their PID into that file.

It is also possible to register an executable to be run when cgroup becomes empty (i.e. it has no nested cgroups and no processes anymore). This is configured in `release_agent` file the root cgroup.

Unified hierarchy

The generic model for cgroups described at the beginning of Section 3.2.3 with multiple hierarchies showed to contain too many degrees of freedom in contrast with controllers that can be assigned to a single hierarchy only.

The reworked model thus uses a single hierarchy[7] and controllers are explicitly assigned to a cgroup's direct child cgroups (the controllers still exist as single instances though). Consequently, processes can be in leaf cgroups only, which makes controllers operations more homogenous.

The clumsy `release_agent` notification upon emptying a cgroup were replaced by a more streamlined approach with `cgroup.events` file that can be passively monitored for changes, for instance by `poll` system call.

3.2.4 Contracts

Contracts [24] are means of monitoring groups of processes in the Solaris operating system and they were created to be utilized by SMF (see Section 1.2).

The key idea of contracts are events that are raised in the context of processes from the contract. The events are delivered to the *contract holder*. For illustration, the events are:

- `CT_PR_EV_EMPTY` – last process exited,
- `CT_PR_EV_FORK` – new process added to the contract,
- `CT_PR_EV_EXIT` – a process from the contract exited,
- `CT_PR_EV_CORE` – a process failed and dumped core,
- `CT_PR_EV_SIGNAL` – a process received signal from process different from contract holder,
- `CT_PR_EV_HWERR` – a process was killed because of a hardware error.

When a new process is forked, it is a member of the same contract as its parent, unless the parent sets so called contract template. In that case, the new process is the only member of a newly created contract.

The typical holder of service contracts is the `svc.startd`, which ensures that services have their separate contracts. The separation is because of another feature of contracts – a contract event can be marked as fatal and when such an event occurs all members of the contract are killed – `svc.startd` can react to that for instance by restarting the service.

The API for the contracts is realized via a special `contract` filesystem that is accessed by functions provided by `libcontract`.

3.3 Entities and naming

Despite their name, service managers do not manipulate only services, there are other entities too, and it is interesting to study what relations exist between the entities and how they are named.

[7] It is backward compatible with the classic hierarchy, it only differs with mount options to the `cgroup` filesystem.

3.3.1 Entities

Service The services are the core concept of each service manager,[8] at the abstract level they are just some code that is able to handle other programs'/user's requests.

Process The actual execution takes place in processes, they are thus another important entity worth watching by the service manager (more details in Section 3.2). It is possible to find almost all cardinalities of service-process relation among existing service managers:

$1 : 1$ This is the case of the classical init (Section 1.1.2) where a service is supposed to be a single child process (either directly spawned by init as specified in inittab (Section 1.1.2) or managed through PID file (Section 3.2.1) from initscript). It is also model that uses the reincarnation server to control system servers in the MINIX 3 system (Section 1.5.3).

$1 : n$ Typical representatives of this group are network servers that spawn a child for each connected client. Such a service consists of the listening process and a variable number of worker processes.

$n : 1$ Multiple services running in the same process are used in the Windows operating system (Section 1.3.1).

$m : n$ One could argue that having a process A that would route the IPC to another process B, could be considered the $m : n$ relation (the service is spread to both process A and B and process A may route for multiple analogous services). Fortunately, such a complicated relation does not exist in studied systems (and it is avoided by defining broker services).

Runlevel[9] The set of services that the service manager allows to start or stop at once, effectively switching. See Section 3.5 for details.

Filesystem Details in Section 3.7.

Exposee As will be described in Section 3.3.2 services have names and the names can be used not only to reference to the service but to somehow communicate with it. Also, a service can have multiple names, so for the sake of abstraction it is worth separating names from services, and refering to such names as exposees. An example of an exposee is a UNIX socket (exposed as a file on the filesystem).

Event The notion of events is inevitable because of the dynamic character of service management. Although the event mechanism is hidden both in systemd and SMF implementation, the event as the first class entity is used by the Upstart only (Section 1.6).

[8] Although Upstart (Section 1.6) calls them jobs.

[9] Just one of many names: init has runlevels, SMF has milestones, systemd has targets.

File Sometimes it is desired for a service to depend on a particular file (e.g. a configuration) and in order to capture that, files have to be recognized entities as well. The files as dependencies can be used both in SMF and systemd, and Upstart provides a bridge that emits appropriate events concerning a file.

3.3.2 Naming

All the entities listed in the previous section usually have names. Although naming is not easy,[10] it is necessary for being able to refer to things.

Processes are numbered by PIDs, filesystem and files have their names from a filesystem tree. The most interesting are the names of the services or the exposees themselves.

Sysvinit itself does not use any names for services, however, init scripts are named – just as they are listed in the `/etc/init` directory (flat, without any subdirectories).

Service names in the SMF are called by their *fault management resource identifier* (FMRI), which are names also used by other frameworks of the Solaris operating system and they are hierarchical (for example `svc:/system/cron`).

Both SCM and systemd use simple flat namespace for their respective services. Events in Upstart are named similarly.

D-Bus

D-Bus [6] is an interprocess mechanism that is based on UNIX sockets and used in Linux distributions. It has the form of a virtual bus where individual objects and their methods are exposed under sophisticated hierarchical names.

It can be thus used to expose the whole service as objects on the bus (this mechanism is similar to HelenOS location service, Section 2.5.3).

3.4 Configuration

Configuration is probably the most important aspect of a service manager from the user's point of view (together with smooth start, Section 3.1.2) The section below looks at how studied service managers are configured.

3.4.1 sysvinit

The configuration is stored in a row-oriented textfile and the `init` program supports reloading and applying its content without restarting (which would equal rebooting in the case of PID 1).

The rest of the configuration is stored in a form of initscripts and appropriate symboling links to them from `/etc/rc?.d` directories. There is `insserv` utility that parses LSB headersand takes care of creation of the correct symbolic links.

[10] `http://martinfowler.com/bliki/TwoHardThings.html`

3.4.2 SMF

Configuration for the SMF framework is in a repository managed by a dedicated daemon `svc.configd`. The changes in configuration and runtime operations (e.g. stopping a service) are both achieved via `svcadm` utility. The repository daemon processes the changes of the state and notifies `svc.startd` which then executes the necessary actions.

The configuration from the repository can be serialized into XML and that is the format in which it is stored on the disk. Furthermore, `svc.configd` also parses legacy init scripts.

3.4.3 Windows

Configuration is stored in the Windows registry. It can be modified by `sc` (Service Control) utility.

3.4.4 Upstart

Upstart uses a standalone configuration file for each service (job), they are all stored in `/etc/init` directory. The file is structured into stanzas and it is possible to create an alternative configuration file that overides only selected stanzas (i.e. easy site-local modifications).

Since Upstart's event framework is almost stateless (see compound events in Section 1.6) changes to configuration files are monitored for any changes and the changes are applied on the fly.[11]

Upstart is partly backwards compatible by executing init scripts from directory `/etc/rc?.d`, however, it is not possible to control individual services as they are specified in the init scripts.

3.4.5 systemd

In a similar way to Upstart, systemd also stores its configuration in separate files for each unit. As well as these unit files, systemd has also configuration files that affect systemd behavior (e.g. default timeouts).

systemd exploits properly named directories with symbolic links as a way to store references between units. This is not unlike symbolic links to init scripts from `/etc/rc?.d` directories. `systemctl` utility can create or remove the symbolic links thanks to information in the special section `[Install]` in the unit file that serves this very purpose.

systemd loads its configuration during it initialization or when it is explicitly signalled (e.g. by the `systemctl` utility) and it generates transient unit files from legacy configuration files (mainly init scripts and `/etc/fstab`).

[11] Interestingly there is a bug in the implementation that may cause crash of the init just by touching files in the configuration directory [9].

3.5 Runlevels

In contrast to what the term runlevel may suggest, runlevels are not stages during system startup or somehow stacked configurations. The concept of runlevels originates from sysvinit (Section 1.1.2) and it is a desired state of the userspace in terms of running services.

3.5.1 sysvinit

Each service managed by the sysvinit specifies the runlevels which it should be run in (Section 1.1.2) and runlevels are identified by a digit from 0 to 9 or a single letter for special runlevels (Table 1.1).

Sysvinit ensures that appropriate services are spawned when it enters a given runlevel during boot. Let us denote A_1 and A_2 sets of services specified for runlevel 1 and runlevel 2 respectively. When the system is running runlevel 1 and sysvinit is asked to switch it to runlevel 2, it terminates all processes in $A_1 \setminus A_2$, starts new processes for $A_2 \setminus A_1$ and it does not touch processes in $A_1 \cap A_2$ at all.

In order to have the same sematics of the runlevel switch also for initscripts, sysvinit sets the `PREVLEVEL` environment variable for its spawned children. The `/etc/rc` script (see Init scripts in Section 1.1.2) reads the variable and evaluates what initscripts need to be executed to realize the transition.

3.5.2 Windows

SCM does not support anything like runlevels described in this section, the set of services is just fixed from the boot.

However, there is a somewhat similar method to start system with limited features (and requirements) as well – it is called the *safe mode*. It is not possible to simply switch to the safe mode, it can be entered only upon system start. Effectively it means that the set of services normally started by SCM is intersected with services specified in the `HKLM\SYSTEM\CurrentControlSet\Control\SafeBoot` registry entry.

3.5.3 SMF and systemd

Both SMF and systemd expand the idea of runlevels and decompose them into more fine-grained logical sets of services. Those sets can have dependencies similar to ordinary services. SMF calls the sets *milestones* and systemd *targets*.

SMF has compatibility milestones for the traditional runlevels and since SMF runs as a child of sysvinit, it is affected by the traditionalswitching of runlevels by sysvinit (Section 3.5.1) and on top of that it ensures the services for the given milestone are running.

systemd works with targets only (though there are backward compatible targets for traditional runlevels) and provides `isolate` operation that results in running services from the target only and no others.

3.5.4 Upstart

Traditional runlevels are transparently mapped to Upstart event mechanism. Upstart defines the `runlevel` event that has the runlevel's name as a parameter. Services present in the runlevel are declared to start on that event and services not active in the runlevel stop on the event. Switching the runlevel is then just emission of the `runlevel` event with an appropriate parameter.

3.6 Resource control

Resource control comprises limitation of usage of various system resources (CPU time, memory, I/O bandwidth). In order to be enforced and not only honored, there must be support in the operating system kernel. As such it is a generic mechanism that need not be exclusively bound to service processes only.

All studied implementations of resource control come with a container for processes[12] and various resources can be limited or allocated to individual containers.

SMF, systemd and Upstart natively support confining the service processes into the resource control containers. Despite the existence of job objects, SCM is not aware of them.

3.7 Mounting filesystems

Filesystems can be thought of as special services that are accessible via the interface of files and directories.

Systems such as Solaris and Linux store list of filesystems with parameters to be mounted during startup in well-known configuration files (`/etc/fstab` in Linux, `/etc/vfstab` in Solaris). The respective service manager than specifies a service that runs early during the startup and mounts the filesystems as declared in the configuration file. The filesystems are thus handled quite coarsely, systemd creates a mount unitfor each mountpoint, although it is also parsed from the `/etc/fstab` file.

3.8 Lazy service activation

The lazy activation is a mechanism that is coupled with service's interface. It means that a service must have well defined interface in order to determine where activating mechanism should be inserted. Following section describes some examples of lazy service activation.

3.8.1 Superserver

Superserver is a network application that waits for clients and when a client sends message to superserver's specific endpoint,[13] the superserver starts another server

[12] *Projects* in Solaris, *job objects* in Windows and *control groups* (see Section 3.2.3) in Linux.
[13] We use the generic term, however, in practice its mostly a TCP port.

and passes the client communication to it. Following clients can then connect directly to the spawned child server or new instances can be spawned repeatedly.

The implementation relies on the POSIX `accept` system call that creates a fresh socket for each connected client and this socket is then inherited by the forked child.

Well known superserver in Linux distribution is `xinetd` (extensible internet services daemon) that provides rich configuration options.

3.8.2 Socket activation

Since the POSIX API for sockets is polymorphic over various socket types, the principle of superserver (in the previous section 3.8.1) can be extended for use with UNIX sockets represented by files in the filesystem.

These sockets are the interface for many services and so the lazy activation approach was used in practice by service manager launchd in the MacOS operating system [15]. Later it was adopted also by the systemd.

3.8.3 D-Bus

D-Bus supports also lazy activation as on top of its naming capabilities (see Section 3.3.2). One can specify an executable for arbitrary object name that should be accessed via D-Bus and the D-Bus daemon ensures that when a client looks for the object, underlying process is started so that the object name will be backed by the real implementation. This is stored in configuration files of the D-Bus daemon and the format assumes that one process can register multiple D-Bus objects.

3.9 Service restarting

Usually we need to restart services when they act up or when we need to reinitialize them, e.g. because of a configuration change or an update. Secondly, services could be restarted when they terminate (either fatally or simply because they are single-use only).

3.9.1 Non-critical services

For non-critical services restarting is not a big issue, if all dependencies are known (and there are no cycles) the services are stopped in reverse topological ordering. If the dependencies are not obvious and cannot be honored, some clients may fail when their service is down. However, they should be allow for unavailable service since we are talking about non-critical services.

The second aspect of restarting, i.e. keeping the service always running by monitoring its state is implemented almost on all studied service managers.

Even the original sysvinit is able to restart services when they terminate (it is used for `getty` running on terminals). Other service managers (SMF, Upstart, SCM and systemd) allow tuning the restarting behavior (i.e. maximum frequency of restarting, maximum number of failures tolerated).

3.9.2 Critical services

The critical services are required for the very fundamental functions of the system and their unavailability is practically equivalent to unavailability of the kernel itself.

Macrokernel operating systems

Although many of the critical tasks are done by the kernel itself, there is always at least one userspace process that is critical too – the service manager – if that fails system is in unspecified state. In Windows, processes explicitly marked as critical cause the system to reboot. Linux ends up in kernel panic when the init process terminates.

The POSIX systems can also avoid whole system reboots when the init process needs to be restarted (typically after update), thanks to the `exec` system call that keeps process's envelope (e.g. open sockets) and executes a new program. However, if the re-executed init process has some internal state it has to save it to a persistent location and restore it after re-execution (this is what systemd does).

Microkernel operating systems

Microkernel architectures have more critical tasks placed into userspace and they are usually split into multiple processes. That is a potential advantage since it doesn't need to fail all at once. However, the processes are heavily interconnected and the failure thus easily propagates to other critical processes.

The GNU Hurd system monitors such a situation and should it happen, it initiates whole system reboot (see Section 1.4.3).

The problem with continuous operation is that the (critical) servers are stateful and their simple restart won't conserve the operations. The MINIX 3 operating system solves this by storing and recovering servers' state from the data store server and keeping a copy of the critical executable images (see Section 1.5.3) in memory of the reincarnation server.

4. Design and implementation

4.1 Overview of the architecture

The responsibilities of the service manager can be divided into three groups.

Broker The broker is where the exposees (Section 3.3.1) are exposed (obviously), it provides API for (de)registration of exposees and most importantly it mediates connections from clients to servers. In the HelenOS context, it could be said that if the server uses `ipc_forward` function then it is a broker.

The primary entity of broker is an exposee.

Restarter The restarter is a component that is able to start and stop services and it also monitors the activity state of the service, keeping always its current state.

The primary entity of restarter is a task.

Resolver The resolver is responsible for resolving dependencies between service, it communicates both with the restarter (so that it knows the dependencies are fulfilled) and with the broker (so that it potentially knows what the dependencies are).

The primary entity of resolver is a unit.

Merging functionality of all three – broker, restarter and resolver into a single program would simplify communication of the individual parties, however, it wouldn't fit HelenOS environment. Firstly, HelenOS design favors decomposition into smaller servers and secondly there already exist some of those components and the merge would be counterproductive.

The brokers are described in more detail in Section 4.4.

The restarter was partially present within the naming service, however, the relevant part was extracted into newly created server taskman, more about that is in Section 4.6.

So the only missing component is the resolver. For that purpose sysman server was created and this chapter describes various aspects of its design. Its similarity to systemd (Section 1.7) is partly because of inspiration, partly because of convergence to the same ideas and partly as an attempt to reuse its terminology so that it'd be more easily understandable for uninitiated.

4.2 Multi stage boot

When the userspace is being started there is a trade-off between size of initially loaded data and number of steps until the userspace environment is ready – in this case the environment is mostly the root filesystem. One extreme is to have only the necessary servers loaded (as described in Section 2.6.3), the second extreme is to bundle all servers and other data into the initial filesystem image. The solution to this is to split the starting process into multiple stages. Since having the root

filesystem is the key event, the stages are two: early userspace that prepares the root filesystem (more details in Section 3.1.1) and the proper userspace.

The goal of the two stage boot was to strip content of the initial RAM disk (Section 2.5.4) to necessary minimum. It reduces memory usage, it fits the single responsibility principle (prepare root filesystem only) and it also makes the RAM disk content dependent on smaller amount of code, i.e. potentially reducing the need of its rebuilds on a production system.

A slight disadvantage of the two stage boot is that it requires support in the vfs (Section 2.5.2) for changing the root filesystem after the actual root filesystem is initialized, so that all applications can access it transparently. The initial filesystem is mounted under `/` and the main filesystem is mounted under `/root` and applications were modified accordingly.

Originally, HelenOS build system produced a single filesystem image that was linked into the RAM disk server. This was changed and now it produces one image for the RAM disk and second one for the actual filesystem (`boot/roothd.img`).[1] The image `boot/roothd.img` is attached by the `tools/ew.py` script as a hard drive of the testing virtual machine.

Details about execution of the stages are described in Section 4.7.

4.3 Configuration

One of the requirements on the system daemon is to give a user means to easily define and tune its behavior. Not only because the system daemon controls system start, this user definitions should be made persistent.

The idea of having a separate configuration server (similar to SMF, Section 3.4.2) was rejected due to the complexity (communication, synchronization) that couldn't be justified by the single responsibility per server, since there was expected very little to be done by such a server – in the end it was all implemented as part of the sysman.

4.3.1 Format

The criteria regarding the format of the configuration were: binary or text format, syntax and division into files. Multiple text files with INI syntax[2] were chosen as the solution. Text files are easy to read and edit. INI syntax is simple yet versatile and splitting into files makes configuration more structured and readable.

Since some other HelenOS applications might make use[3] of the configuration parser, the functions were implemented as a `libconf` library.

4.3.2 Processing

The configuration files are subject to the split due to multi stage boot (early userspace contains files necessary to prepare the actual root filesystem only) and thus sysman was designed so that it can load the configuration repeatedly. The

[1] New configuration options `CONFIG_RAMDISK_ONLY` and `HDFMT` were added to this end.

[2] Author is not aware of any specification of a canonical INI format. Fortunately, the syntax is quite simple and there is no space for serious ambiguities.

[3] devman currently uses its own INI-like format, with very simple built-in parser.

current implementation supports extensions (i.e. adding new units) only, not modification of the existing units.

Because there can be references to other (not yet processed) units in the configuration files, the sysman performs two-pass processing of the configuration to resolve those references. Also the loaded data are marked and only if the both passes succeed, the marked data are unmarked and used by the sysman routines. Loading configuration from multiple files is thus made in transactional manner.

4.4 Entities

First, we need to identify what entities (as defined in Section 3.3.1) are there in the HelenOS environment.

There are servers (Section 2.5) that expose services via the naming service (Section 2.5.1. There are servers that expose their services via the location service (Section 2.5.3), such as window compositor or networking servers. Then there are device drivers (Section 2.5.7) that expose their services (devices) indirectly through the device manager and the location service or directly through the device manager (see Section 2.5.5 for explanation). Finally, there are the filesystem servers (Section 2.5.6) that expose their functions through mountpoints they are handling.

With the summary of the existing entities, let's map them to the trinity scheme described in Section 4.1. The naming service and the location service are clearly brokers. The device manager and vfs (Section 2.5.2) are also brokers for their respective specialized servers. The exposees are thus the names registered at these brokers.[4]

Once we clarified what the brokers are, let's think about possible unit types.

4.4.1 Unit types

From the resolver's perspective, all the servers that have some exposees registered (regardless the broker) are the same, i.e. they are service units.

When designing other types of units, we realize there is a partial duality between units and exposees. There can be a unit u with multiple exposees $e_1, \ldots, e_n$ or the same situation can be modelled as units $u, v_1, \ldots, v_n$ where each unit v_i has a single exposee e_i and is dependent on the unit u.

Let's see examples how the duality affected design of unit types.

A filesystem server is a unit that has an exposee at vfs (the registered filesystem type). Further, we can look at mountpoints (handled by the filesystem server) as other exposees. The dual view, is that there is the filesystem server unit as previously and on top of that there are other units of the new type mountpoint, each having a single exposee (the mountpoint).

The second example are device drivers (Section 2.5.7). Each driver (being a service unit) can have an exposee per device or alternatively there can be device units that depend on the driver service.

In the case of mountpoints the dual model was because having mountpoints as units makes them available to the dependency graph and more importantly,

[4]Filesystem servers are exposed under their respective filesystem type name.

user can configure behavior of mountpoints since the unit bears its settings.

On the other hand, the devices are just exposees of a driver service. There is nothing to be configured for them and dependency on a device is realized by registering a handler at the location service (see Section 2.5.3).

We decribed *service* units and *mountpoint* units so far. There are two more unit types necessary: *targets* and *configuration* units. The target units play the same role as in systemd (description in Section 1.7.1).

A configuration unit represents a fragment of configuration and when it is started, sysman loads the given configuration. The configuration units were created for the purposes of the multi stage startup (details in Section 4.7) so that the configuration is loaded in appropriate moment.

4.4.2 Units and tasks

Tasks are relevant for the service units only.

Idea of creating a specialized service API (similar to Windows, Section 1.3.1) was rejected in favor of expanding already existing HelenOS mechanisms of generic inter-task communication (see Section 4.6.2). That also affected used cardinality of the unit to task relation. The designed model is $1 : n$, however, implemented is the model $1 : 1$. For the explanation of the cardinality, see Process entity in Section 3.3.1 and for details about implementation of the $1 : 1$ model see Section 4.6.

The cardinality model $n : 1$ was deemed suboptimal because it mixes multiple concerns into one task, doesn't provide actual separation (common address space) and makes control of a service via task API inappropriate. In this context, it is worth noting, that service units are not considered a part of their brokers (it would only complicate matters, the model would be then $m : n$).

4.5 Dependency resolver

Dependency resolver is a part of the sysman server that ensures that states of all units and operations upon them honor the defined dependencies between units.

Units and dependencies between them form an oriented graph. The units are vertices and oriented edges represent a dependencies between the adjacent units. For the purposes of the thesis only single type of dependency was implemented – if there is an edge $A \rightarrow B$, then the unit B must be successfully started before the unit A is started. Inverse relation applies to stop operation.

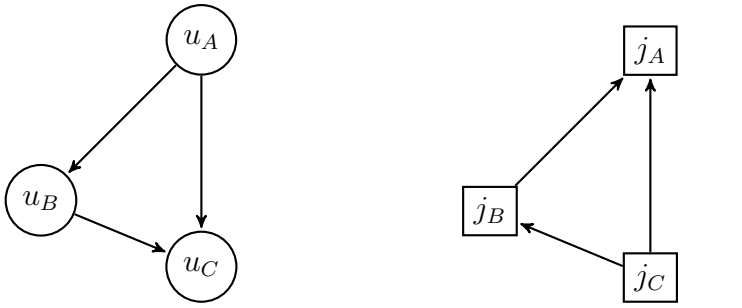
If the operations with the units were synchronous and non-blocking, the units would be topologically sorted and the operations applied one by one. Unfortunately, the operations are either blocking or asynchronous and we strive to avoid blocking of the sysman [5] and for asynchronous operations we must keep units in transitional states (starting, stopping). Furthermore, we'd need to keep a desired state in each unit, dependencies would have to be taken into account and it'd became even more complicated if there was a request to stop a unit while it would be in a transitional state (starting). This is just an illustration that we need new

[5] This is just an example, with dedicated fibrils, it'd be possible to perform blocking unit operations and not block the sysman server totally.

abstraction that would facilitate asynchronous operations with units. And this abstraction are jobs – a job is a context of an ongoing unit's operation.

4.5.1 Job closures

Two data structures are used when jobs are processed: job closure and job queue.[6] Let's explain on an example how these work. Suppose there is a request to start a unit A that is in the dependency graph shown in Figure 4.1a.



(a) Dependency graph (b) Job closure of start A

Figure 4.1: Creating job closure from dependency graph

Breadth-first traversal in the dependency graph is started in the node u_A and for each reached unit an appropriate job is created (in this case starting of dependency units). The type of edge that connects two unit nodes determines whether there will be a constraint between the respective jobs. The implemented dependency requires strict ordering and so blocking constraints are added between the jobs. After the traversal terminates, the set of jobs and constraints is called a *job closure* (with respect to the operation on the unit A), see the simple example in Figure 4.1b.

The created job closure is static, the jobs are not executed. In order to execute a job, it is added to the job queue together with the constraints from the job closure. Only the jobs that aren't blocked by others are dequeued and executed.

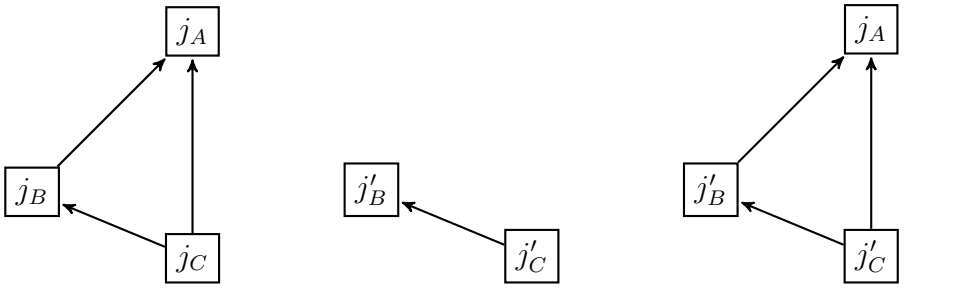
The implementation doesn't allow multiple jobs for a single unit (not even queued) and so adding some jobs to the queue can fail. However, if we the added job is of the same type as the one already present, the jobs are merged.

In order to avoid unexpected unit behavior, none job from a job closure can have a conflicting job in the queue (see Figure 4.2 for successfully added job closure). If that happens, the job closure is rejected and job queue is reverted to original state (jobs are unmerged).

4.6 Task monitoring

Originally, the naming service served both as a broker and a server implementing all task creation and monitoring API (see Section 2.5.1). It was decided

[6]Job queue is implemented as linked list. Inherent limitation of HelenOS linked lists is that typical structure can only in one list at the moment. New abstract data structure – dynamic array – was implemented to serve purposes of job closure (in contrast with linked lists, job can be in several dynamic arrays at once).



(a) Job closure (b) Job queue (c) Job queue after queuing and merging the job closure

Figure 4.2: Enqueuing a job closure (Jobs j_B and j'_B were successfully merged (same for C) and job j_A was enqueued.)

to extract task related functionality from the naming service into a new server (taskman), thus properly separating concerns and preparing room for advanced task monitoring (group-based approaches as shown in Section 3.2).

Currently, taskman allows to register a callback (`task_register_event_handler()`) from other applications so that they can be notified about retvals or exit values (see Section 4.6.2) of other tasks. This is mostly useful for the sysman server.

4.6.1 New task creation modification

Because the naming service is not handling task spawning anymore, some changes were made so that newly spawned tasks are not connected to the naming service but to the taskman instead (more precisely they are connected to the caller of the `SYS_PROGRAM_SPAWN_LOADER` system call,[7] which might be possibly exploited for a nice recursion later).

The goal of the initial loader handshake (Section 2.2.2) is only to introduce the new task to taskman and obtain a connection to the naming service (however, that is done lazily so that the naming service itself can start as any other process).

4.6.2 Task termination

The original detection of task termination as described in Section 2.3.1 was not reliable. Because of that a new kernel event `EVENT_EXIT` was added.[8] This allows to distinguish when a task just returned its *retval* and when it really terminated (exit value). See Table 4.1 for the semantics of retval and exit value.

Also the `task_wait` API was slightly modified, user can specify by flags in `task_wait_t` structure whether they want to wait for retval or exit value and `task_wait` would unset appropriate flags to indicate what event actually occurred.

[7]Except for boot time tasks (Section 2.6.3), they are connected to taskman as well, however, it is achieved by putting taskman on the first place among boot time tasks.

[8]For kernel events see Section 2.4. Already existing `EVENT_FAULT` is newly processed similarly to `EVENT_EXIT` (Section 4.6) thus breaking functionality of `taskdump` application.

retval	exit	meaning
none	running	task is running
none	exited	task unexpectedly terminated
set	running	task daemonized itself
set	exited	task terminated (with or without an error)

Table 4.1: Possible combination of task retval and exit values

4.7 System startup

The whole userspace startup process (compare with Section 2.6.4) is orchestrated by sysman according to the configuration files. Because of the chicken-egg problem, little portion of the configuration is hardcoded into sysman. It is the following sequence of three units that are started one after another by the sysman:

- `initrd.tgt` Hardcoded target that ensures load of first configuration from RAM disk.
  - `init.cfg` Hardcoded unit loads configuration from RAM disk (from `/cfg/sysman` directory).
    - `initrd.mnt` Hardcoded unit mounts the RAM disk filesystem. It doesn't depend on anything since boot time tasks (Section 2.6.3) are all it needs.
- `rootfs.tgt` Target that ensures loading and preparation of the root filesystem. This unit is loaded from configuration file `/cfg/sysman/rootfs.tgt`.
- `default.tgt` Target that brings the rest of userspace up. It is loaded from `/root/cfg/sysman/default.tgt`, however, this path is already configurable in `/cfg/sysman/rootfs.cfg`.

4.7.1 IPC autostart

IPC autostart is a feature that reduces number of dependencies that must be explicitly specified in configuration files. Async framework reserves `arg4` of `IPC_M_CONNECT_ME_TO` method for flags that specifies how brokers should handle connection requests. `IPC_FLAG_BLOCKING` causes that broker doesn't forward the connection request until the particular server registered (instead of failing immediately).

A new flag macro `IPC_AUTOSTART` was defined, the behavior is similar to `IPC_FLAG_BLOCKING`, however the broker asks sysman to start the appropriate unit so that it shouldn't block indefinitely. This new flag was implemented into the location service and the vfs, since in their case it is quite straightforward how to translate the exposee name to the unit name.

5. Conclusion

The stated goals of the thesis were somehow fulfilled. The system is able to boot, go through the cascade of individual stages of two stage boot and the services to start are declared in text configuration files. This is more configurable than the original hardcoded version (although it brings the burden of keeping configurations for mutliple platforms, which was not addressed in the thesis). The booting time in comparison with the original solution increased quite significantly. The reason for that is that the sysman has quite verbose output during the start, and since the root filesystem is stored on an external device, the system has to wait for it a little longer.

The second goal, service state monitoring, was achieved by restructuring task creation procedure and initroducing a dedicated task to process task-related kernel events.

Although the thesis was mainly concentrated on the service manager issue [4], it also moved forward booting from persistent filesystems [3]. It prepared environment to cope with service connection timeouts [1] and almost enabled graceful shutdown [2].

Future work

There are still many things that'd need polishing and things that would be just nice to have with the sysman infrastructure existing.

Unordered dependencies

Current dependencies doesn't allow start of a unit until its dependencies start. This is not always necessary and it prolongs startup process. The required changes are small, limited to job closure creation (Section 4.5.1).

Notification API and graceful service stop

Async framework (Section 2.1.3) was extended with functionality to be stopped (`async_manager_stop`). Tasks could register to taskman to receive notifications that could be used for graceful stopping of a service. In the end, graceful shutdown of the system (thanks to dependency resolver and knowledge of dependencies from IPC autostart) could be also realized.

Restore taskdump application

Originally, taskmon listened for the kernel event `EVENT_FAULT` in started taskdump utility to collect the failed job. Now, `EVENT_FAULT` is listened to by taskman, some other task should probably register to the taskman for propagation of this event.

Nested brokers

Devman exposes driver exposees in a subnamespace of the location service. This prevents effective use of sysman broker API, there should be explicit support for

nested brokers.

Multiple tasks per service

Extend taskman with task group management and use it in sysman.

Replace naming service with location service

The location and naming service responsibilities overlap and robust location service could be used instead of the naming service, thus reducing complexity of the system.

Recursive boots

Since the taskman is exclusively responsible for its spawned tasks (now all), some hierarchical model could be used to achieve recursive boots or light userspace virtualization.

Bibliography

[1] #184 (Support for ipc_connect_me_to_timeout() would be useful) - HelenOS. http://trac.helenos.org/ticket/184.

[2] #414 (Graceful system shutdown) - HelenOS. http://trac.helenos.org/ticket/414.

[3] #447 (Boot from persistent file system) - HelenOS. http://trac.helenos.org/ticket/447.

[4] #525 (Service manager) - HelenOS. http://trac.helenos.org/ticket/525.

[5] CGROUPS. https://www.kernel.org/doc/Documentation/cgroups/cgroups.txt.

[6] dbus. http://www.freedesktop.org/wiki/Software/dbus/.

[7] Handler callback function (Windows). https://msdn.microsoft.com/en-us/library/windows/desktop/ms683240%28v=vs.85%29.aspx.

[8] illumos-gate/init.1m. https://github.com/illumos/illumos-gate/blob/0a1278f26ea4b7c8c0285d4f2d6c5b680904aa01/usr/src/man/man1m/init.1m.

[9] Kill init by touching a bunch of files. http://rachelbythebay.com/w/2014/11/24/touch/.

[10] Linux Standard Base Specification. http://refspecs.linux-foundation.org/LSB_3.0.0/LSB-Core-generic/LSB-Core-generic/generic-lsb.html.

[11] Multiboot Specification version 0.6.96. http://www.gnu.org/software/grub/manual/multiboot/multiboot.html.

[12] Rethinking PID 1. http://0pointer.net/blog/projects/systemd.html.

[13] SERVICE_STATUS structure (Windows). https://msdn.microsoft.com/en-us/library/windows/desktop/ms685996%28v=vs.85%29.aspx.

[14] setpgid (The Open Group Base Specifications Issue 6). http://pubs.opengroup.org/onlinepubs/009695399/functions/setpgid.html.

[15] systemd for Developers I. http://0pointer.de/blog/projects/socket-activation.html.

[16] www:documentation:start [Wiki]. http://wiki.minix3.org/doku.php?id=www:documentation:start.

[17] *init(8) Linux System Administrator's Manual*, July 2004.

[18] *init(8) BSD System Manager's Manual*, March 2012.

[19] *systemd(1) system and service manager*, July 2015.

[20] Jonathan Adams, David Bustos, Stephen Hahn, David Powell, and Liane Praza. Solaris service management facility: Modern system startup and administration. In *LISA*, pages 225–236, 2005.

[21] Thomas BSG Bushnell and Gordon Matzigkeit. *The GNU Hurd Reference Manual*, September 2013.

[22] Inc. Free Software Foundation. GNU Hurd. `https://www.gnu.org/software/hurd/index.html`.

[23] Oracle. *init (1M) System Administration Commands*, July 2014. `http://docs.oracle.com/cd/E36784_01/html/E36871/init-1m.html`.

[24] Oracle. *process (4) File Formats*, July 2014. `http://docs.oracle.com/cd/E36784_01/html/E36882/process-4.html`.

[25] Mark Russinovich and David A. Solomon. *Windows Internals, Sixth Edition, Part 1*. Microsoft Press, 6th edition, 2012.

[26] Andrew S. Tanenbaum and Albert S. Woodhull. *Operating Systems Design and Implementation*. Prentice Hall, 3rd edition, 2006.

[27] Lenka Trochtová. Device drivers interface in HelenOS system. Master's thesis, Charles University in Prague, 2010.

[28] Wikipedia. Init — Wikipedia, The Free Encyclopedia, 2015. `https://en.wikipedia.org/w/index.php?title=Init&oldid=664324284`.

[29] Wikipedia. UNIX System V — Wikipedia, The Free Encyclopedia, 2015. `https://en.wikipedia.org/w/index.php?title=UNIX_System_V&oldid=665737279`.

[30] Wikipedia. Version 6 Unix — Wikipedia, The Free Encyclopedia, 2015. `https://en.wikipedia.org/w/index.php?title=Version_6_Unix&oldid=666656940`.

List of Abbreviations

API	application programming interface
CPU	central processing unit
DLL	dynamically linked library
ELF	Executable and Linking Format
IPC	interprocess communication
LSB	Linux Standard Base
PID	process identifier
POSIX	Portable Operating System Interface
SCM	Service Control Manager
SMF	Service Management Facility
sysinfo	readonly general purpose database of the Spartan kernel
sysvinit	System V init

A. User documentation

Compile and run

Compile source using standard HelenOS toolchain as described on the official website.[1]

- Enable multi stage boot by setting `CONFIG_RAMDISK_ONLY` to false. [2]
- Disable task debugging by setting `CONFIG_UDEBUG` to false. (Otherwise, there'll be no task that'd reap failed tasks.)

Use standard `tools/ew.py` script to properly setup the virtual machine.

During boot phase, you will see plenty of debug output even after the graphical interface is rendering. sysman enables kernel console during boot, you have to escape it by typing "continue".

Control utility

Fingerprint of the system daemon to the user is very little. You can interact with it through `sysctl` utility. It can list the units and their states and manipulate their state. Start `sysctl` without any arguments too see its syntax.

Configuration files

```
[Configuration]
Path = /path/to/configuration/directory

[Mount]
What = devices/path\to\device\node
Where = /path/to/mount/directory
Type = fat

[Service]
ExecStart = /path/to/executable arg1 arg2

[Unit]
After = dependency1.svc dependency2.tgt
```

Listing 3: Supported sections in respective unit files (`[Unit]` is universal, target does have nothing)

Configuration unit files end with `.cfg`, mounts with `.mnt`, services with `.svc` and targets with `tgt`.

[1] `http://trac.helenos.org/wiki/UsersGuide/CompilingFromSource`

[2] Single stage boot should in theory work, however it was not tested and proper unit files are missing.

B. Source code overview

Source code with implemented changes can be obtained from the repository below and a snapshot is on the attached CD.

```
bzr branch lp:~werkov/helenos/system-daemon
```

Directory structure

Here is a list of substantially modified directories and files.

```
boot/
    initrd.img          RAM disk image
    Makefile.common     definitions of RAM disk files
    roothd.img          root FS image
uspace/
    app/
        sysctl/         sysman control utility
    cfg/
        sysman/         unit file repository
    dist/               RAM disk content
    disthd/             root FS content
    lib/
        conf/           configuration parsing library
        sysman/         sysman API
    srv/
        sysman/         sysman source
            test/       sysman PCUT tests
            units/      unit implementatins
        taskman/        taskman source
```