2014-07- 02

YaIniT

Yet an almost universal Init Tool

https://github.com/initiate414

WORKING DRAFT

This is the Draft of the Documentation for the Project
YaIniT
and not the state of the development or published works

Author:
Dieter Miosga
Computer Network Specialist
mailto: dieter.miosga@gmx.de

Statistics : Pages: 66 Words: 21578 Characters: 142037

Table of Contents

Foreword...5
....5
1. The principles of the boot and administration Tool YaIniT...........6
1.1 The need of an open and free operating system versus the need of open advertising for free GNU/Linux operating system distributions. . .6
1.2 A Tool is necessary...7
2. Components and Execution Logic (User's Guide).........................8
2.1 INIT...8
2.1.1 Kernel Commandline Parameters.......................................9
2.1.2 INIT Parameters...9
Parameters controlling the console output..........................9
Parameters controlling the debug level............................11
Parameters controlling the customization..........................12
Parameters controlling the system initialization..................13
Parameters specifying the target system...........................20
Parameters in a compressed or abbreviated form....................23
Parameters controlling the target system..........................24
2.1.3 The INIT programmation and logic...............................26
2.2 BUSYBOX..30
2.3 The Linux Kernel..31
2.4 KEXEC..31
2.5 Additional Tools and Features..32
2.5.1 Dialog facilitating features : dialog, xdialog and yad......32
2.5.2 configator...32
3. Detailed Structure (Administrator's Guide)............................33
3.0 General YaIniT configurability...33
3.1 INIT again: Execution Logic and Elements..............................33
3.1.1 Configuration..33
3.1.2 Logical steps and program flow.....................................35
3.1.3 Integration of customizations for hardware and distribution. 40
3.2 BUSYBOX again: setup and configuration................................40
3.2.1 Configuration..40
3.2.2 Predefined Setup and Optimization..................................41
3.3 Linux KERNEL again: setup and configuration..........................41
3.3.1 Configuration..41
3.3.2 Predefined Setup and Optimization..................................42
3.4 KEXEC again: setup and configuration...................................47
3.4.1 Configuration..47
3.4.2 Predefined Setup and Optimization..................................47
3.5 TOOLS again: setup and configuration...................................48
3.5.1 Configuration of configator...48
3.5.2 Use of configator..48
3.5.3 Configuration of dialog...48
3.5.4 Configuration of xdialog...48
3.5.5 Configuration of yad...48
4. APPENDIX...49
....49
4.1 On the usefulness of examples and templates from free software. .49
4.2 The AUR Kernel Repository of Arch-Linux................................49
4.2.1 Linux kernel post-factum patches / pf.natalenko.name........49
4.2.2 Linux kernel RealTime patches / ftp.kernel.org/pub/linux/kernel/projects/rt.......................52

4.2.3 Linux kernel GRSecurity and PaX patches / grsecurity.net....53
4.2.4 Linux kernel l-pa / https://aur.archlinux.org/packages/linux-l-pa....61
4.3 Big Distributions and their useful kernel patches....61
4.3.1 The OpenSuSE Linux kernel patches....61
4.3.2 The Gentoo Linux kernel patches....62
4.3.3 The Mageia Linux kernel patches....62
4.3.4 The Ubuntu Linux kernel patches....62
4.3.5 The Fedora Linux kernel patches....63
4.4 Restrictions on Kernel Configurations by GNU/Linux distributions 63
4.5 Comments on the adaptive changes for YaIniT....64
4.5.1 The more exact Linux kernel configuration, applying the yainit-x86-config.patch....64
4.5.2 Linux kernel configuration with the patching scripts do-make-ash and do-makeg-ash....64
4.5.3 Applying the patchets from bigger distributions....64
5. INDEX....65
5. INDEX....65
6. GLOSSARY....66
6. GLOSSARY....66

Foreword

Almost ten years after the initiation of the GNU/Linux system by starting the public development of Linux in 1991, the presentation of distributions by regular unified surveys started in the way of commercially driven websites.
Since that day, it was observable, that the distributions do want to concur, with the effect, that a common standard for a free and open sourced operating system remains incomplete for the needs of users.
One of these lacks for a common and complete standard is the way, a GNU/Linux distribution will start its initialization, right after the Linux kernel is up and wants to execute the first application, called **`init`**, which has to be located in the root directory of the residing boot file system.

For today, the different configuration setups and initialization logic of distributions remain incompatible.

This small study and more or less administratively development work, wants to show how far the effort to create one unifying and almost universal **`init`** can be driven and how much of various GNU/Linux system creations can be administered by a such a tool,
keeping fulfilled the following criteria:

- With the fewest means necessary else, that all remain fully GPL compatible. All used programs and tools underlie the GPL.
- No overhead or specialist team necessary for to use it: An understanding of the Almquist Shell for programming **`init`** and knowledge of the C programming language to be capable to modify the system tools in case of failure or necessary customization.
- Recursive and Reusable: All written programs can be used on any platform and optimization is applied not only for performance but also for size.
- Widespread use: Can be elaborated and serve as a fundamental for the decisions, that private persons, personnel in public administrations and enterprises will have to take in choosing from the overwhelming number of publications of hundreds of different distributions, all approaches to satisfy the same criteria of investigation. Frees from time expensive, long enduring and recurring work for basic system setup and hardly accessible designed distribution specific structures.

Summer Solstice Release, Spain, 2nd of July, 2014

Dieter Miosga

1. The principles of the boot and administration Tool YaIniT

1.1 The need of an open and free operating system versus the need of open advertising for free GNU/Linux operating system distributions

During the search in the Internet for a handsome, reliable and universal distribution in the years after the beginning of commercial GNU/Linux distribution watchers, the state of the distributions made a lot of work necessary, to reach a successful reduction of administrative time by automatizing all recurring tasks, for maintaining a distribution or operating system fully adapted to personal needs.

After eight years (2003 - 2011) of recurring attention to the results of distribution development, as publicly accessible on the Internet at web sites anonymously, it turned out as not to circumvent to write a separate tool, composed of one or more running Linux kernels, one or more compilations of **`busybox`**, one reliable **`kexec`** mechanism and an **`init`** script, interpretable by the BusyBox Applet ASH, sufficing the complexity necessary for fulfilling the criteria as following:

1. Keep control over kernel security and reliability by allowing a personalized kernel, configured to integrate as many distributions as possible, or make specially configured kernels for distribution targets available.
2. Keep control over the boot process and system initialization. Do not admit automatized system governance such as unwanted network accesses, unwanted hard disk mounts or language and monitor settings as automatized by manufacturers' boot and initialization scripts. What the user specifies, happens or nothing happens.
3. Maintain the system initialization fully user controllable. Do not allow searching for the boot target system in unwanted system bus connections or system storage devices. Allow the specification of all necessary system parameters by the user, as independently from the target system as possible.
4. Keep the administration at a user friendly level. What is usable as Graphical User Interface for the configuration of the Linux kernel and BusyBox, will be available for the entire YaIniT .

1.2 A Tool is necessary

The effort of many distributors, to show something really own, special and new, and a structure, mighty in his degrees of freedom, between the kernel initialization and a fully configured running operating system, make it necessary to apply programmed intelligence, in the form of a complex interaction of some single programs, called a Tool or a Tool Set in the language of the GNU/Linux developers.

The claims for universality, liberty and sustainability obligate to:

1. a use of as few programs or scripts as possible
2. be as user friendly as possible by using only one programming language and one scripting language necessary to understand and verify the entire system logic (minimization of knowledge necessary for own administrative actions, here the C programming language and the ASH Almquist Shell)
3. a choice of programs that all do not restrict the idea of free and open source software, respecting the historical evolution of the concerned associations and providers
4. be usable as independently as possible from the hardware used (modular configurable personal computers, supercomputing centers' machines and small embedded systems should be able to run the same tool for recognition of the initialization target system and to determine and specify the initialization parameters)
5. be freely modifiable, enhanceable and enlargeable by the user who is applying the Tool (special properties of distributions should be respectable by an as flexible as possible programmed logic)

YaIniT is designed to respect the conventions and definitions of its single components as far as possible. Kernel commandline parameters remain in function for the kernel and are distinct from the parameters of YaIniT **`init`**. Unavoidable common use is marked up explicitly in this document.

However, there are restrictions necessary on the set of possible configurations of the Linux kernel and the BusyBox feature, to keep the qualities performance, universality and reusability. Despite of restrictions, YaIniT allows booting kernels with some configurations beyond these restrictions.

2. Components and Execution Logic (User's Guide)

2.1 INIT

The **`init`** program for the YaIniT tool is only plain ASCII text interpretable statements by the Almquist Shell (ASH) from BUSYBOX. It is kept as simple and as efficient as the few experiments performed for optimization have proven.

The init program resides at filesystem root in INITRAMFS. /init is the location, for a plain ASCII text file starting with #! followed with the path and the name of executable to interpret and execute the following lines. A kernel configured with forbidden #! execution needs a link to an executable in binary form, either in ELF format or a.out format. This does not work with the **`init`** of YaIniT.

The **`init`** of YaIniT starts with *`#!/bin/busybox ash`* and needs a working BusyBox executable with a minimal configuration set. With a /bin/sh symbolic link to busybox, configured with the feature to allow /bin/sh point to /bin/busybox, meaning ASH invocation, **`init`** can achieve reuseability or at least a reinvocability by the commonly used Bourne Again Shell, BASH. The busybox ASH is a subset of BASH; all scripts running under ASH do run with BASH but not vice versa.

Some experiments, the Linux kernel parameter recognition logic and a look at a variety of **`init`** mechanisms, let it appear as convenient to create a very flexible commandline input parameter handling structure.

The kernel commandline parameters have to be processed by **`init`** in ASH the following way:

1. The parameters are scanned from the left to the right. The leftmost are called early parameters, the rightmost late parameters.
2. The place in the sequence matters. A certain kernel parameter must be reversible by a contradicting one in the later sequence.
3. The entire set of **`init`** parameters should not deviate from the structure of the Linux kernel parameters, specified in the kernel commandline. If they do, **`init`** will ask for a special customization of its logic, specified explicitly within the environment of **`init`**. If this customization does not exist, **`init`** will bypass its invocation and issue a warning according the verbosity level specified by kernel commandline parameters.
4. Parameters can be unary or binary: A unary parameter is a letter or a set of letters standing alone in the parameter kernel commandline, separated by a blank character from the left or right neighbor. A binary parameter is a letter or a set of letters followed by the equal sign “=”. After the equal sign the subparameters follow. If they are more than one, they are separated by a comma “,”.
5. Due to the restriction in the number of bytes(characters) in the kernel parameter command line, it might become necessary to use an abbreviated form of the specified parameters. An abbreviation convention exists and can be used as following: uniparm=klnmopqrstuvw......, where k,l,n,m,o,p,q,r,s,t,u,v,w,.... are numbers in {0,...,9} and have the signification according kernel commandline parameters explained in section 2.1.2 later on.

2.1.1 Kernel Commandline Parameters

The kernel parameters are documented in the ./Documents directory of any unpacked admissible Linux kernel package. Don't fear to read code to discover undocumented ones.

`init` does not influence specified Linux kernel parameters at the commandline of bootloaders. They are a subset of **`init`** and are interpreted separately from the kernel mechanisms. For example the specification of `noswap` at the kernel commandline can be opposed by mounting explicitly a swap filesystem, if the kernel is configured for it.

2.1.2 INIT Parameters

The kernel parameters alone do, except in a predefined default case, not allow a correct program flow after the kernel initialization. Special **`init`** parameters and specific parameters for the target operating system are necessary.

These are listed here:

Parameters controlling the console output

All these parameters can be preset to a default in a system configuration file, editable before YaIniT generation time. Any specification at the kernel commandline will override the default.

`quiet`

: do not display any system messages at all, same as quiet=7 or verbose=0

independent from kernel messages, specified by loglevel=0,..,9

`quiet=`{0,1,2,3,4,5,6,7,8,9,ask}

: do not display certain system messages

=0 : same as verbose=5
=1 : suppress error messages

=2 : suppress warning message

=3 : 1+2, suppress warning and error messages

=4 : suppress informatory messages

=5 : 1+2+4 suppress warnings, error and informatory messages

=6 : do not enter prompts

=7 : suppress all system messages and do not prompt

`verbose`

: display all system messages, same as quiet=0 and verbose=5

`verbose=`{0,1,2,3,4,5,6,7,8,9,ask}

=0 : same as quiet=7 or quiet

=1 : display error system messages

=2 : display warning system messages

=3 : 1+2, display warnings and errors

=4 : display informatory system messages

=5 : 1+2+4 display warnings, errors and informatory messages

=6 : allow interactive stops in program flow

=7 : display all system messages on console and effectuate interactive stops

=8,9,ask : disregarded, same as 3

`prompt={shell|dialog}`

: specify the nature of system prompt.

=shell : use the busybox ASH statements `read`, `echo`, and the busybox `printf` to perform the dialog.
System fails, when misconfigured and busybox or Linux kernel cannot reach the
system console for both Input and Output. (e.g. missing keyboard driver in initramfs!)

=dialog : use a dialog program to specify the system needs.
This covers the **`dialog`** program with the **`ncurses`** environment, or graphical dialogs
such as **`yad`** or **`xdialog`**, when invoking a complete
GNU/Linux environment with X11 from a loaded rootfilesystem at system boot.
System fails, when misconfigured and initramfs or rootfilesystem do not contain
properly configured the dialog, xdialog or yad programs.

`delay`

: pause the system after each message displayed on the console for a time specified in `wait=`

Every kernel commandline parameter, that contains the **`ask`** possibility, requires a working input/output mechanism to establish the dialog. For example, a missing keyboard driver, either unloadable or not compiled in, will lead to an unbootable system within YaIniT!

Parameters controlling the debug level

All these parameters can be preset to a default in a system configuration file, editable before YaIniT generation time. Any specification at the kernel commandline will override the default.

`debug`

: run in debug mode. Attempt to activate only Linux kernel debugging, but load debugger modules, such as `scsi_debug` or `evbug` if not blacklisted, and do some more necessary system actions. Mounts debugfs and displays kernel debug messages on the console.

`debugfs`

: mount the debugfs at system initialization. Runs independently from `debug` or `initdbg` and does it as a separate and specific action.

`nodebugfs`

: do not mount debugfs.

This reverts the effect of the `debugfs` and `debug` parameters. This should be done, if debugging shall not cover the system parameters accessible by debugfs. For complete debugging, do not use this parameter.

`initdbg`={0,1,2,3,4,5,6,7,8,9}|{init_section0,init_section1,....,init_sectionN,ask}|ask

: run **`init`** in debug mode as specified by level

=0 : don't debug, same as if initdbg is not specified, maintained for the reasons to
be able to switch off preceding definitions

=1 : mount debugfs if compiled into kernel and all parameters in sysfs and procfs are set for its use

=2 : display debug messages from init on the system console as specified

=3 : 1 + 2 combined. Copy the file for init messages together with the output of `dmesg` command into the /var/log directory of target system, if the target system can be mounted in read-write mode.

=4 : check target system specified with root= and rootfs= for compatibility with YaIniT

=5 : 1 + 2 + 4 combined

=6 : check target system specified with root= and rootfs= for compatibility with LSB 4.x

=7 : full debug, 1+2+4+6 combined

=init_sectionN: any text parameters separated by a "," after the number or instead of a number,
specifying the selection for exclusive execution of sections of **`init`**.

=ask: ask in a dialog for the section(s) of the init script to run.

Parameters controlling the customization

All these parameters can be preset to a default in a system configuration file, editable before YaIniT generation time. Any specification at the kernel commandline will override the default.

`distro`=distribution_name

> : specify a reference name which contains special customized **`init`** sections and parameters for the target distribution. This distribution must be accessible by reference to its specifications at the INITRAMFS and rootfilesystem.

`distro_parms`={distro_parm1,distro_parm2,....,distro_parmN,ask}

Parameters controlling the system initialization

Parameters beginning with **no** do strictly forbid to load any modules for the specified system part or system function and will inhibit the access to any devices connected to, as far as possible. After switching to a target system, the control is completely passed to this target system and necessary changes and corrections are overridden. The commandline parameter `nofw` cannot hinder anyway the target boot system to run its own init logic, accessing devices connected to the Firewire Bus. System variables of the YaIniT- **`init`** are reset by the rootfilesystem switch.

Therefore, this must coincide with the `inject` parameter and the possibility to access the target rootfilesystem in read-write mode to store some information on the necessary changes. These must be target system specific and include conditionally depending modules in hierarchical or cross-hierarchical relation. That's why inject parameter programming requires full compliance of certain standards and common conventions, the Linux Standard Base (LSB) among them.

All these parameters are only valid for the initialization during the **`init`** run. If **`init`** is run from INITRAMFS, then the use of the **`busybox`** applets **`switch_root`** or **`chroot`** to invoke the target boot system will annihilate the effectiveness of some of these parameter specifications. Only parameters known by the target system and not modified by it will remain effective, except the target system is overridden by **`init`** parameters `overlay`, `inject` and `sanitize`.

All these parameters can be preset to a default in a system configuration file, editable before YaIniT generation time. Any specification at the kernel commandline will override the default.

`i | interact | interactive`

: do start the **`init`** completely interactive. Ask for every relevant system parameter to be specified within a programmed dialog.

`noauto | na`

: do not automatically search for an adaptively compiled and optimized Linux kernel or BUSYBOX executable in the INITRAMFS or in the target boot system, or load them for execution. Bypass any optimizations for **`busybox`** foreseeable. Do not extract the file containing the processor family adaptations of **`busybox`** and delete this file. Use this for booting with the default generic executable of **`busybox`** or any customized preinstalled version. `noauto` is switched active automatically without being specified, if no specially adapted and selectable executables for **`busybox`** are found in the INITRAMFS or any rootfilesystem used with this function. The user can specify `noauto`, if such is installed before, to circumvent failures or to accelerate booting by omitting the unpacking and selection of the busybox executable. Further, `noauto` switches off automatic kernel module recognition and does load all modules loadable from the present files in INITRAMFS according the predefined system variables for module loading. This is useful, if the kernel hangs by the reason of not or erroneously loaded modules or necessary legacy modules, no longer foreseen in the algorithm of automatic kernel module loading, but it make take time to setup and configure again the whole INITRAMFS.

`nomon |   nomonitor | noscreen`

: do not use the VGA/DVI/HDMI ports, do not access tablet screen, auxiliary screen or USB-touchscreen or load relevant modules for graphical devices (especially for routers or only remotely controllable systems). This excludes the interactivity parameter and any dialog invocation. System messages are all sent to a file in INITRAMFS. Useful for small embedded systems such as WLAN-Routers or NAS-Servers in LAN.

`nousb`

: do not use the Universal Serial Bus and do not load modules for any devices connected to USB ports

`nousb`={0,1,2,3,4,5,6,7,8,9,ask}

=0 : reset nousb, Allow (again) the use of USB and connected devices

=1 : allow USB 1.1 bus only

=2 : allow USB 2.0 bus only

=3 : allow USB 3.0 Bus only

=4 : allow USB 1.1 and 2.0 bus only

=5 : allow USB 1.1 and 3.0 bus only

=6 : allow USB 2.0 and 3.0 bus only

=7 : allow USB 1.1 through 3.0

=8,9 : not set, same as 7

`nousb`=drivername1,...,drivernameN,ask

: do not use USB devices with the specified drivers, do not load this or these drivers

`nopci`

: do not use the PCI System Bus and any devices connected to PCI

`nopci`=drivername1,...,drivernameN,ask

: do not use PCI devices with the specified drivers, do not load this or these drivers

`nopcie`

: do not use the PCI-E System Bus and any devices connected to PCI-E

`nopcie=`drivername1,...,drivernameN,ask

: do not use the PCI-E devices with the specified drivers, do not load this or these drivers

`nopcmcia | nocardbus`

: do not use the PCMCIA/Cardbus connection System Bus and Cardbus/PMCIA devices

`noham | noradio`

: do not use the Amateur Radio System Bus and devices

`nobt | nobtooth | nobluetooth`

: do not use the Bluetooth System Bus and Bluetooth devices `noirda | noinfrared`

: do not use the Infrared Data Associations feature

`nonfc | nonearfield`

: do not use the Near Field Communication devices

`nop9 | noplan9 | no9p`

: do not use the Plan9 resource sharing with 9P2000 protocol

`nocaif`

: do not use the Communication CPU to Application CPU Interface (CAIF)

`nofw | nofirewire`

: do not use the Firewire IEEE-1394 System Bus and devices

`nops2`

: do not use the PS/2 System Bus (no PS/2 keyboard or PS/2-Mouse) and connected devices

`nocan`

: do not use the CAN System Bus and connected devices

`nosnd | nosound`

: do not use any sound card from the System Bus

`nonet`

: do not use any network. This includes noatm, noatalk, nodec, noipx, noether, notr, nowlan, nobat. The TCP/IP, many of it is usually compiled in statically into the kernel binary, is blocked either.

`noatm`

: do not use ATM network or ATM devices

`noatalk`

: do not use Apple-Talk network, the Apple Macintosh protocol

`nodec | nodecnet`

: do not use DEC-Net network, Digital Equipment Corp. protocol `noipx`

: do not use IPX protocol

`nodccp`

: do not use DCCP (Datagram Congestion Control Protocol)

`nosctp`

: do not use SCTP (Streaming Control Transmission Protocol)

`notipc`

: do not use TIPC (Transparent Inter-Process Communication Protocol)

`nol2tp`

: do not use L2TP (Layer Two Tunneling Protocol)

`noqos`

: do not use QoS (Quality of Service) modules and service

`noib | noinfbnd | noinfiniband`

: do not use the Inifiniband feature modules and services

`noether | noeth`

: do not use ethernet network and devices

`notr`

: do not use Token Ring network and devices

`nowlan | nowireless | nowifi`

: do not use wireless IEEE-80211(a|b|g|n|ac) network devices, including Wireless Broadband Devices (WiMAX)

`nobat | nobatman | nomesh`

: do not use B.A.T.M.A.N. Wireless mesh protocol. Do not allow any local PPP connection to peer or WLAN local server, only perform routing through to the remote net by wireless access router.

`noscsi`

: do not use any SCSI interface devices

its use is restricting any scsi device access since kernel 3.0. This excludes loading of drivers for storage devices, such as scsi-optical device, scsi-tape and scsi-disk storage.

If CONFIG_SCSI, CONFIG_CHR_DEV_SG and CONFIG_BLK_DEV_SD are enabled and the IDE/ATA/SATA block devices specified under CONFIG_ATA, then these appear as /dev/sd*. Forbid the use of libata with the `noint` boot parameter. For older kernels or rare old disk drivers, there can be specified CONFIG_IDE and the devices will appear as /dev/hd*. In this case, noscsi will have the effect of letting accessible only /dev/hd* devices.

`noide`

: do not use any IDE devices

deprecated since kernel 3.0, the long term supported hard disk drive modules are all specifiable with the CONFIG_ATA and CONFIG_SCSI sections.

if CONFIG_SCSI, CONFIG_CHR_DEV_SG and CONFIG_BLK_DEV_SD are enabled in the kernel configuration, then only /dev/sd* and /dev/sr* devices will appear, independently of what is configured under CONFIG_IDE.

`nocd | nodvd | nooptical`

: do not use any optical disk devices (CDROM-/DVD-/BlueRay-Disk reader or recorder)

do not allow any optical drive access since kernel 3.0. The restriction includes any storage devices appearing a /dev/sr*   in the device tree. An IDE- CDROM drive appearing as /dev/hd* is blocked for access in the same way.

`noint  | nointernal`

: do not use any internal hard disk devices. This restriction includes any hard disk devices connected to the local internal ISA/EISA/MCA/PCI/PCIE Bus. External devices, connected to a PCMCIA-, PC-CARD-, Firewire-, Serial-, Parallel-, CAN-, or USB-Bus are filtered out and considered as external devices. Only the hard disk (block-) devices connected directly to the internal system bus are excluded from automatic recognition and won't appear in the YaIniT list of bootable devices.

Note: There is no parameter to forbid the use of external hard disk devices against internal ones. Therefore, you must specify explicitly the relevant system bus restriction (nousb, nofw,...).

`nocardmem | noflash`

: do not use any card memory, such SD-Card, Micro-SD, or USB-Memorystick or Pendrive, neither as a boot nor as a root device. Do not load such modules. This restriction includes any disk devices connected to a local internal Bus (ISA/EISA/MCA/PCI/PCIE,....) and to the Firewire- or USB-Bus with drivers for Card or Flash Memory devices.

`novirt | novm`

: do not use any virtualization features, neither running as paravirtualized guest under a virtual host operating system, nor running as a host in kernel-based virtual machine (hypervisor) mode.

`nomod | nomodules`

: do not load any kernel modules, even if there are any accessible. This does exclude the unpacking of kernel modules in tarball archive and the mounting and loading from a squash file.

May also be useful to test kernels with all necessary modules, but no configuration, compiled in

(configuration not retrievable by: → cat /proc/config.gz | gunzip > /DOT.config-`uname -r 2>/dev/null`).

`nomod=`list_of_kernel_module_names_separated_by_comma | ask

: do not load any of the kernel modules specified in the module list. The kernel module names as they appear in the filesystem must be known and specified, not as they appear internally in the Linux system. Do not attach the .ko (kernel object) extension. First, it is not unique because the module file may be compressed (.ko.gz, .ko.bz2, .ko.xz, .ko.lz), second, the YaIniT program is as good as to recognize that automatically. The modules in this list can be specified at system generation time in a configuration file.

list_of_kernel_module_names_separated_by_comma is equal to:

[modulename][:kernel module parameters],....,[modulename][:kernel module parameters]

Note: The specification of any blacklisted modules will overwrite the presetting of any blacklisted and for probing forbidden kernel modules in the YaIniT system files!

`blklst= | blacklist=` list_of_kernel_module_names_separated_by_comma | ask

: same as nomod=

`modlst= | modulelist=`list_of_kernel_module_names_separated_by_comma | ask

: do load all of the kernel modules in the order specified in the module list. The modules in this list can be specified at system generation time in a configuration file.

`whtlst= | whitelist=`list_of_kernel_module_names_separated_by_comma | ask

: same as modlst

Whitelisted modules will be loaded anyway after all automatic selected or preset kernel modules loading and after unloading automatically unused modules.

Note: Modules that appear in both , modlst= and nomod=, or blklst= and whtlst= sections, will be treated by the priority of the last occurrence. The module occurring in the last list, will be handled by YaIniT according to the nature of this list, annihilating all previous occurrences.

`wait`

: stop **`init`** execution, at any place a relevant waiting point is set, for the default wait time in seconds or milliseconds. The default can be set to 0.

`wait=`[0 | WAITMAX]

: stop **`init`** execution at any time relevant halting point for the specified wait time in seconds or milliseconds, until the system maximum is reached. WAITMAX, in its default, is a system variable to be configured at system generation time in a configuration file, valid if `wait=` is not specified. If WAITMAX is greater than 99, the number is valued in milliseconds, else in seconds. A microseconds wait is not implemented in YaIniT. `wait=0` does cancel any system waits, and is to be used with care. Busybox does for now not allow a sleep function in milliseconds values.

`nowait`

: same as `wait=0`. Resets all default and previous `wait=` specifications.

`noprocfs`

: do not mount procfs. This should regularly not be done. The kernel parameters won't be processable by **`init`**! In some cases when booting embedded systems, it might be necessary to use these parameters noprocfs and nosysfs. However, the logic of **`init`** tries to mount once the ***procfs*** and ***sysfs*** to read the kernel command line parameters and system specifications, if the kernel configuration allows this. After having understood these, ***procfs*** and ***sysfs*** will be unmounted again. Kernels, configured without ***procfs*** and ***sysfs***, will obviously be detected with the `initdbg` parameter. Otherwise, they will boot through, regardless the Linux kernel or a system routine crashes, without any warning or other debug possibility by **`init`**, if the target system and the used kernel is not configured correctly. If there is ***procfs*** compiled into the kernel, YaIniT will mount it for short, to recognize its parameters and to load, if allowed, automatically some modules (such as for input devices) . Is all read from ***procfs***, then it is unmounted again.

`nosysfs`

: do not mount sysfs. This should regularly not be done. Essential system information won't be present! Automatic kernel module loading as it is done by YaIniT as well as the target boot system, relies on the presence of sysfs-informations in the “modalias” or “uevent” files.

See noprocfs. ***sysfs*** is handled like ***procfs*** by YaIniT.

`nodebugfs`

: do not mount debugfs.

This reverts the effetcivity of the `debugfs` and `debug` parameters. This should be done, if debugging shall not cover the system parameters accessible by debugfs. For complete debugging, do not use this parameter.

Parameters specifying the target system

The **`init`** logic needs to allow sufficient configuration possibilities for target system properties, to reach a completely user definable system initialization.

All these parameters can be preset to a default in a system configuration file, editable before YaIniT generation time. Any specification at the kernel commandline will override the default.

`root=`[[LABEL=disklabel][=][UUID=uuidcode]] | [PARTUUID=partition-uuid] | [[/dev/]{hd|sd}{a,..,z}{0,..,n}] | [/dev/disk/by-{label|uuid|id}/{LABEL|UUID|ID}] | [RAM] | [ask] [:rootfilesystemtype] [:rootfilesystem with full relative path]

> : do specify the device containing the installed target system or the rootfilesystem to mount or to copy into RAM. If a specification of the disklabel appears to be not unique, than both, UUID and LABEL can be specified:
>
> root=LABEL=MyTravelDisk=UUID=0a10fcgh0
>
> if root=LABEL=MyTravelDisk or root=UUID=0a10fcgh0 are not sufficiently unique
>
> Specifying e.g. PARTUUID=0003ceff-02 should always be unique, if the UUID of the Partition Table from the corresponding disk device is accessible and readable by the **`dd`** applet of **busybox.** Thus, **`init`** will recognize this parameter and try to find this partition, here the second partition (sda2|sdb2|sdc2|...) of the corresponding disk.
>
> Specifying `root=/dev/ram0` or `root=RAM` does obligate YaIniT to boot completely into RAM of the running system. The further booting is executed through the distribution specific settings found in the INITRAMFS or the boot target system. The user is allowed to specify with the boot= parameter compressed squash-filesystem files, that will be completely mounted from RAM or from the specified device.
>
> If the `rootfs=` parameter is specified, then the system searches for the rootfilesystem as specified, and mounts it for switching to it with either **`chroot`** or **`switch_root`** command from BUSYBOX's applets. If no rootfs= parameter, and if not necessary, no rootfstype= parameter is specified, the conventional separator ":" may be used to specify, in fix order, first the filesystem type of the rootfilesystem partition and second the path to the rootfilesystem, which may be specified as a file with its path, containing the rootfilesystem to mount as a loop device, or the directory, where the rootfilesystem resides on the device, specified before the first ":".
>
> In case, there is a file specified, to mount as a loop device, the rootfilesystemtype specification is used to mount this file. The rootdevice will be mounted by autodetection using the autofs4 module, according the specification in the first parameter.
>
> In case, there is a directory path specified, the rootfilesystem will be mounted as specified by the rootdevice parameter and the path will be respected during the switch by `switch_root` or `chroot`. The rootfilesystemtype is the type of the rootdevice and will be used by the mount command. Note that only the filesystems ext2, ext3, ext4, f2fs, xfs, reiserfs, btrfs, initramfs (together with aufs, overlayfs or unionfs) can be used as rootfilesystem in read-write-mode. Squashfilesystems and others, mountable only in read-only mode, must reside on one of these filesystems.
>
> The root= parameter can be specified entirely in lower case letters. Upper case letters will be converted internally to lower case letters during the recognition mechanism. However, upper cases in the system label will be fully respected and are compared as they are on the disk.

`boot=`[[LABEL=disklabel][=][UUID=uuidcode]] | [[/dev/]{hd|sd}{a,..,z}{0,..,n}] | [/dev/disk/by-{label|uuid|id|name}/{label|uuid|id|name}] | [ask]

> : do specify the boot device where the filesystem resides, that will replace INITRAMFS during boot. If boot device is not specified (boot=), or the boot parameter does not appear in the parameter list, then the system files will all be searched in the accessible INITRAMFS.

`swap=`[[LABEL=disklabel][=][UUID=uuidcode]] | [[/dev/]{hd|sd}{a,..,z}{0,..,n}] | [/dev/disk/by-{label|uuid|id}/{label|uuid|id}] | [zramfs[,size in kB]] | [ask]

: do specify the device containing the Linux swap filesystem. Specifying /dev/null as the swap filesystem, will turn off swapping

`rootfs= | rootfsfile=` [[path to systemfiles or root directory]/[rootfs systemfile0][,rootfs systemfile1][,....,rootfs systemfileN] | [ask]

: do specify the path to a target system's rootfilesystem, containing the systemfiles and its filenames.

If left empty, the rootfsfile defaults to / and there will be directly switched with **`busybox`** applets **`chroot`** or **`switch_root`** to the filesystem root on the device specified with root= without loading or mounting any explicit system file. The path and the systemfile refer to the device specified with boot= or ip=, ipx=,atalk=,dec= .

This filesystem is mounted into RAM in read-only mode and overlayed for read-write operations with the union filesystem AUFS or UNIONFS, regularly compiled in into the YaIniT kernel binary. Saving all the changes, accumulating in a read-write section, to a permanent storage device requires special conventions on the user's demand. All system changes and configuration settings can be stored immediately in a separate filesystem on a separate rewritable medium and can be reloaded to overlay at the next boot. Linux filesystems allowed for rootfs are : ext2, ext3, ext4, f2fs, btrfs, reiserfs, xfs, squashfs, ecryptfs. The squashfs is by definition and design a read-only filesystem and requires for changes a time intensive rebuild from the random access memory of the right booted system.

`ip=`[[local IP|empty for dhcp],[remote IP|empty for dhcp],[protocol://DNS-Name of remote server][,path to systemfiles or root directory]] | [ask]

: set up IP protocol parameters and remote location of systemfiles as necessary. Note that only from NFS or CIFS/SMB with protocol=nfs|smb a remote file system access is done for targeting a remote bootable GNU/Linux system. With protocol=http,ftp,... this does not work, while loading squashfilesystems or any other rootfilesystems in a single file via network copy into RAM to use as a TMPFS or RAMFS to boot into, will work. CIFS/NFS partition mounts allow to transfer any system file via network on demand directly, or via execution into the kernel's userspace, while others need to copy the entire rootfilesystem into RAM or onto local disk.

`ipx=` //TODO

: set up IPX protocol parameters and remote location of systemfiles as necessary.

`atalk=` //TODO

: set up Apple Talk protocol parameters and remote location of systemfiles as necessary.

`dec=` //TODO

: set up DECnet protocol parameters and remote location of systemfiles as necessary.

The following are kept for naming convenience and can be specified as well with the modlst= or whtlst= parameter.

`ether=`[modulename][:kernel module parameters],....,[modulename][:kernel module parameters]

: specify ethernet module and parameters

`tr=`[modulename][:kernel module parameters],....,[modulename][:kernel module parameters]

: specify token ring device module and parameters

`atm=`[modulename][:kernel module parameters],....,[modulename][:kernel module parameters]

: specify ATM (Asynchronous Transfer Mode) network device module and parameters `wlan=|wifi=`[modulename][:kernel module parameters],....,[modulename][:kernel module parameters]

: specify wireless device module and parameters

Parameters in a compressed or abbreviated form

For the case, a kernel commandline is overflowing, too many parameters and subspecifications are too long, there is the possibility to specify some of the parameters in a compressed or abbreviated form.

`uniparm=`”{klmnopqrs.........}”

: specify the kernel parameters appearing as unary or binary numbered parameters in a single and unique special parameter. For an unary parameter it is necessary to specify 1, if it is active, or 0 if it is inactive or not present in the commandline. For binary parameters, at the place of the sequence stands a 0, if it is not set and a value from {0,...,9} if set. The only exception is the wait parameter at the end of the sequence: There you can specify a bigger number > 99, meaning the time to wait in milliseconds. Parameters, that have an ordered or unordered sequence of comma separated value words, cannot be used in the compressed form.

The sequence is:

1 2 3 4 5 6 7 8 9 10 11 12 13

interactive quiet verbose debug initdbg nomon nousb nopci nopcie nopcmcia noham nobt nofw

14 15 16 17 18 19 20 21 22 23 24 25 26 27 28

nops2 nocan nosnd nonet noatm noatalk nodec noipx noether notr nowifi nobat noscsi noide novirt

29 30

nomod wait

For example:

uniparm=”0100001”   is equal to :

quiet nousb

So, it does not make sense to abbreviate!

uniparm=”11000100100000” is equal to: uniparm=”1”   interactive and only interactive,

the rest has to be specified the interactive way and is void here! No real abbreviation!

uniparm=” 01005000011110100111101010110 4250” is equal to:

quiet initdbg=5 nopcmcia noham nobt nofw nocan noatm noatalk nodec noipx notr nobat noide novirt wait=4250

The time unit for the last parameter wait is [ms] as there is a number greater than 100.

So, this makes sense to specify in abbreviated form.

Parameters controlling the target system

For the case, a running target GNU/Linux system is accessible and runnable, after an initial debug run or directly, there exist some special kernel parameters, allowing to control the way the target system is accessed.

//TODO: more exact specification of overlay, sanitize and inject

`[{0,1,2,3,4,5,6,S}]`

: a number indicating the system runlevel of the target system.

0 : boot mode

1 : single-user mode for maintenance

2 : multi-user mode without network

3 : multi-user mode with full network support

4 : not set , same as 3

5 : multi-user mode with full network support and GUI-Desktop service

6 : shutdown

S : same as 1, the single user system maintenance mode.

Target systems, that do not make any notice of the runlevel specification and rely on an own programmed system start logic, will boot as they are programmed. In this case, the runlevel specification has no effect.

`rw | readwrite | read_write`

: mount the target system read write when passing control to its `init`.

`ro | readonly | read_only`

: mount the target system read only when passing control to its `init`.

Note: A rootfilesystem, containing a distribution, that needs read-write access to the systems root (/) directory, has to mount it in read-write, at the moment the control is passed to its `init`. If not, then there remains the possibility to specify overlay and /or sanitize for a continued boot. For this case, a separate read-write filesystem is necessary.

`overlay`

: try to overlay as many features as possible already existing in the YaIniT suite over the executables of the target system.   For the overlaying is the OVERLAYFS (from MAGEIA or UBUNTU Kernels) and the AUFS (Another Union File System) or the UNIONFS (www.filesystems.org) used. UNIONFS was not available during the appearance of kernel versions 3.3 through 3.12. The target filesystem is mounted read only and the overlaying files are copied from the YaIniT suite into the read-write space to make them active before the ones of the target system. The target system remains unmodified.

`overlay=`{0,1,2,3,4,5,6,7,8,9,ask}

: specify the level for the overlay mechanism:

=0 : do not overlay, usable to switch off overlaying

=1 : overlay kernel and modules

=2 : overlay existing executables with busybox applets

=3 : 1 + 2 combined

=4 : overlay existing executables in the target system with programs residing in the INITRAMFS of the booting YaIniT suite or specified with boot= parameter for the purpose of a fallback to a more mighty bootshell than BusyBox provides.

=5 :   1+2+4 combined

=6 : correct hindering and wrong entries in the target system's configuration files and overlay them (such as /etc/fstab with a wrong /dev/sd* specification for the target system's root device, monitor and graphic card configuration, network and firewall setup, ….)

=7   :   1+2+4+6 combined

=8,9:   not specified, same as 7

=ask : really start a dialog on the system console to ask for what to do

`inject`

: use the previous experiences with the overlay command to overwrite the files in the target system with the ones of YaIniT. inject and overlay cannot be specified at the same initialization time.

CAUTION: This modifies the target system! And may lead to its unusability, if not tested properly before with possibilities of overlay.

`inject=`{0,1,2,3,4,5,6,7,8,9,ask}

: use the previous experiences with the overlay command to overwrite the files in the target system with the ones of YaIniT according the previously working overlay level.

`sanitize`

: sanitize erroneous or conflicting existing configurations to match for use with the previous experiences by the overlay command. Depends on overlay. Does not sanitize or alleviate errors existing within the logic of the target GNU/Linux system, only relevant ones for YaIniT to be runnable. Includes correction of lacks in configuration or erroneous parameters to let run an YaIniT kernel, configured most close to “make allmodconfig”.

2.1.3 The INIT programmation and logic

The kernel parameters alone do, except for the predefined default case, not suffice the correct program flow after the kernel boot. An **`init`** logic exceeding the Linux kernel commandline parameters is needed for YaIniT's claims in anyway. The **`init`** logic includes conditional prompting for resources not found and user defined configuration possibilities. The execution logic is described as follows:

1. . *self configuration*
   - Parameters passed to the **`init`** script are saved. As the **`init`** script is replicated with another name, they have to be checked for a special first parameter, indicating a different proceeding in the **`init`** script. Note that the Ubuntu kernel has or had in some versions a mechanism to pass kernel commandline parameters directly to **`init`**. This causes no restriction of universality, as long as the special first **`init`** parameter does appear different from Ubuntu's parameters. The distinction to general Linux kernel commandline parameters has to be assured by programming. YaIniT ever examines the entire kernel commandline and assigns the parameters found directly to system variables.
   - Necessary system function are read in. The further proceeding of **`init`** is determined. The output control of system messages is established.
2. . *self analysis*
   - The configurable system parameters are read in, if the file containing them exists.
   - The basic rootfilesystem directory structure for INITRAMFS is created.
   - Linux filesystems procfs and sysfs are mounted.
   - The kernel parameters and kernel commandline parameters are read in and are evaluated as far as necessary.
   - If kernel commandline parameters do forbid the mounting of procfs or sysfs, then according `nosysfs` and `noprocfs`, these filesystems are unmounted again.
   - If kernel commandline parameters do request, other low level filesystems such as debugfs are mounted.
   - The root device is determined. To maintain the difference between booting to RAM with kernel commandline parameter `root=/dev/ram0 | root=RAM | root=LABEL=RAM`, booting to local or by network connection reachable remote block devices or storages, the specification of `root=blockdevice_where_target_rootfilesystem_resides` is necessary to recognize the correct branching and the following correct loading of **`init`** parts. If nothing is specified, than **`init`** will always boot into RAM to allow for any further search or manual specification at the boot prompt. The kernel executable may contain the entire rootfilesystem as well as all the init-scripts.
3. . *self preparation*
   - The architecture and processor family specific configurable parameters are read in, if existent. The machine specifications are analyzed and verified.
   - If precompiled binaries for optimization of **`busybox`** included in a tar-archive (.tar.gz, .tar.xz, .tar.bz2) of the form /bin/busybox-* exist, then the archive is extracted and the matching

executable for the used machine is selected for any future use. This can be inhibited with kernel commandline parameter `noauto.`

- If exists a tar-archive in INITRAMFS, containing kernel modules (and maybe some other files), this archive is extracted if no kernel modules are already resident in INITRAMFS.
- A preconfigurable set of unwanted kernel modules and unwanted kernel version relicts are deleted from INITRAMFS, if existing.
- The device filesystem tree is set up: **`/dev/*`**
  This is still necessary for some kernel configurations, that do not create the device filesystem tree automatically.

4. . *branch to continuation script*

- All symlinks for the **`busybox`** executable are created.
- Delete any disturbing other init : /bin/init, /sbin/init, …., only the root init **`/init`** will persist in INITRAMFS.
- If exist kernel modules for the running kernel, and the kernel modules are present and necessary to load due to the kernel configuration, then the necessary modules for input device (keyboard, mouse, touchscreen) interactivity, block device accessibility and/or network location specification are loaded according the parameters' setting. Blacklisted and whitelisted modules can be overridden from system defaults with `nomod= | mod= | whtlst= | blklst=` respectively.
- The system shell parameter in INITRAMFS is set, such that the symbolic link from /bin/sh or /bin/ash points anyway to /bin/busybox.
- The branch to the appropriate continuation script is performed by copying the YaIniT **`init`** script in INITRAMFS. This copy is invoked by **`init`** at the penultimate statement with a parameter, notifying the **`init`** copy over its internal proceeding: Either for continuing operating from RAM to load a boot or a target rootfilesystem, or prepare more for accessing a block storage device, or for accessing a known and identified block storage device directly.
  For the last one, there is no chance to change anything in the specification of the target system, except with the `i|interactive` feature by the dialog or the `initdbg` feature during one of the drops to interactive system console. Therefore, the target system needs to be mountable for read-write access, and some configurations for the actual boot settings must be changeable manually. Otherwise, the rootfilesystem is directly mounted, the residing `init` mechanism is detected and the system control is passed to the residing active `init` with the **`chroot`** or **`switch_root`** command from **`busybox`**. System checks are not circumvented in anyway, but an installation of kernel modules of the YaIniT kernel is required, if the necessary modules are not all compiled into the kernel.

#-#

Remark:

System Designers and Performance Tuners may ask why in YaIniT is such a recursive and time taking algorithm for the setting of the script parameters, when explicit setting of parameters and reading them in appears much faster.

The answer is: Universality, and therewith Codesize Minimization and Reuseability were prioritized before absolute Performance. The YaIniT system boots only once, when optimizing for the underlying Processor architecture, and a second kernel load is necessary if not preinstalled, while a generic kernel and a generic shell restricts in most of the cases the processor's power for all the time of its use with this kernel and shell. After passing the features of an administratively well prepared YaIniT **`init`** script, high performance and real-time computing possible over the entire Processor architecture. Booting to a preinstalled system is much faster, than booting to RAM for establishing a system configuration with many options. However, as long as the system RAM is faster, so called RAM based Live systems operate much faster than storage based preinstalled ones.

In case of fatal failing of the YaIniT infrastructure, there is the possibility from the Linux kernel design, with a kernel containing the modules necessary for the recognition of the target system device, to boot without the requirement of an **`init`** in INITRAMFS. The Linux kernel allows this by the kernel commandline parameter `root=PARTUUID=partitionuuid-partitionnumber` !

#-#

The continuation scripts for RAM boot perform the following tasks:

1.

   Find the accessible system device, for the specification with the root= | rootfs= | boot= commandline parameters.

- If the root= device does exist, then prove its uniqueness and check its properties (partitions, filesystem types, sizes,...).
  If the root device does not exist and no default device is specified in the YaIniT configuration or the default device does not exist and no other root device location (NAS network based storage, NFS-Server volume,....) is specified by other parameters, then branch to interactive mode and ask the user to specify manually among the list of all accessible devices, if any.
  If the current configuration does not allow to access the target rootfilesystem, then load necessary modules (for filesystems, diskdrives, etc..)
  Mount this partition and make its contents accessible at least in read-only mode.
- If the rootfs= file does exist, then mount the rootfilesystem at a mount point in INITRAMFS.
  Mount failures have to be detected and circumvented as far as possible. The rootfilesystem will be loaded in and mounted from RAM only if enough of it is free. If the rootfilesystem is mountable for read-write operations, then it is mounted to RAM only if enough read-write space is available, as swapfilesystem space, free disk space, or RAM.
  If the rootfs= file is not specified, it defaults to / . This does mean, that the entire mounted partition, specified by root= is taken as rootfilesystem, and the search for an init continues for being able to boot into.
  If the rootfs= file does not exist, then drop into the interactive mode and allow the user to specify manually a file.
- If the boot= partition does exist, then mount the bootfilesystem at a mount point in INITRAMFS.
  Overlay as necessary, using the relevant union-filesystem-mechanism, the contents of the bootfilesystem over the current contents of INITRAMFS, to make them active. If necessary, pass control to specified programs available now. The bootfilesystem feature is designed to contain parts of YaIniT, such as configator, dialog, yad, kexec,...., precompiled Linux kernel versions to select the

optimal one for reboot with KEXEC and the optimization mechanism as described for **`init.`** If the boot device is the same as the root device, it is not necessary to specify it explicitly.

2.

2.2 BUSYBOX

The Almquist Shell (ASH) used above for the init script is just one applet from BUSYBOX.

BUSYBOX is the effort to create all programs, necessary for a running system shell, in a single setup like the Linux kernel configuration is. BUSYBOX is configurable as one standalone static executable with no other references, as one single executable depending on some other libraries (libc, libm, libbb,…..), or as many small executables referring to a single library libbb in the kernel understandable ELF-Format. BUSYBOX is designed to cover many features of the **coreutils** program suite, of the **util-linux** program suite and some more runnables, necessary for a working and mighty command shell of GNU/Linux (with e.g. sysvinit, systemd,....).

For detailed descriptions, code insights and downloads of the BUSYBOX program, see http://www.busybox.net

The other applets are specifiable to compile in at configuration time. It is kept as simple and as performing as the time expensive experiments for optimization have proven.

2.3 The Linux Kernel

The well known place for to download the kernel is at ftp://ftp.kernel.org/pub/linux/kernel

Lots of works have been done to describe the Linux kernel and here is not the place to extend this. For some reasons, the Linux kernel can be called monolithic, restrictive and administratively overheaded. Here it is not intended to advocate for certain conclusions in favor of certain publications, but to show how far the efforts can be elaborated, according certain paradigms and criteria with generalized exigences. We may want not to forget, that the Linux kernel is an exceptional piece of work with rather public accessibility, intrinsic advocacy for all related academic fields, from information science and mathematics over electrical engineering to physics, with a possible usability for entire universities, and a perfect dashboard for the state of software industry as a start-up aid for all entrepreneurial efforts from software engineering to hardware commerce. Why such a project is not realizable without a registered trademark entry, is a question beyond the author's capability for recognition.

The Linux kernel is compiled for YaIniT with its special administrative scripts, all working with the Almquist Shell (ASH) from BUSYBOX. All included configurations contain the possibility for the kernel to be reloaded with KEXEC, without recurring the BIOS or needing a hard reset of the entire system.

Kernel configurations without allowing the KEXEC feature, do allow booting with YaIniT, but any adaptation is excluded. Solely the direct boot with one and single kernel, as invoked from the bootloader, is conceded.

Kernel configurations without elaborated performance tuning work as well as the most exact local machine adapted configuration are tolerated by YaIniT.

However, there are some conditions without not, listed in 3.3,, to be fulfilled for a kernel, to make it boot within YaIniT.

2.4 KEXEC

On the Linux kernel ftp server resides a state of the development of the KEXEC feature.

KEXEC is a C/Assembler program and runs with a few parameters. It can load any in the Linux VFS/FHS residing and accessible and such configured kernel executable (usually vmlinuz*) into RAM above the running kernel (KLOADER subfeature). With another call and parameter for execution, together with the specification of the **`init`** parameters, the new kernel is invoked and the system boots with bypassing the machine BIOS. In the same way, the old kernel can be reloaded and the system may be switched back.

KEXEC is invokable from the Almquist Shell (ASH) of BUSYBOX without big complications. But KEXEC cannot, for now, be compiled as a standalone static executable. The need of some system ELF-libraries is not to avoid.

KEXEC resides on the Linux kernel server at ftp://ftp.kernel.org/pub/

2.5 Additional Tools and Features

The interactive booting with a completely guided dialog appears to be inconvenient when using only the echo and read mechanisms from ASH. However a minimal interactivity is realized within YaIniT. System parameters can be entered interactively from console prompt or from dialog shells or windowing dialog masks.

2.5.1 Dialog facilitating features : dialog, xdialog and yad

An old and simple program for making user's choices and dialogs more friendly and easier to use, is dialog. This was updated by another effort for ease of use, called yad or yet another dialog. We received it at version 1.3 as of 2013-09-25.

`dialog` needs **`ncurses`**, a library to provide the layout for dialogs at console level, **`ncurses`** needs a libc (**`glibc, newlib, eglibc, uclibc,...`**).

`xdialog` (X-Windows Dialog) needs a running X11 environment and of course, a matching libc.

`yad` (yet another dialog) needs a working X11 environment and the necessary GTK+ features, and a matching libc.

2.5.2 configator

This is a specially for YaIniT written configuration analyzer and corrector tool. It is written in ASH.

It is necessary, as the different distributions use a variety of boot logic mechanisms and initialization programs (sysvinit, systemd, bash, perl, python, ...).

Configator uses and invokes, according relevant parameters, the executables of the Linux Standard Base, as of version LSB 4.1 .

3. *Detailed Structure (Administrator's Guide)*

YaIniT is set up in a directory containing its elements: Linux kernel, kexec, BusyBox, the yainit initramdisk, tools (dialog, configator) and eventually a rootfilesystem, containing all what is necessary from GNU/Linux to invoke features like **`xdialog`** or **`yad`**. The directories ***initramdisk*** , ***kernel*** , ***busybox*** , ***tools*** are predefined. A Makefile using the make feature of GNU/Linux does provide the necessary user guidance and can be invoked with the same programs as the Makefile in kernel or busybox (**`make help, make gconfig, make xconfig, make menuconfig`**).

3.0 General YaIniT configurability

3.1 INIT again: Execution Logic and Elements

3.1.1 Configuration

init is usually passed to the kernel on a *`initramdisk`* file (the most common abbreviation is `initrd`), containing the **`init`** executable and the files and structures, that are presented as INITRAMFS immediately after the kernel boot. The name initramdisk is historical. Bootable Kernel binary and a compressed image of INITRAMFS were on two different floppy disks. Today it is a directory within a Linux filesystem, that is compressed with the cpio command to a single file and compressed again with lzma, gzip, or bzip2, such that it is extractable into the INITRAMFS automatically by the kernel in the FHS/VFS root (/). The name of the file is specified at the boot loader prompt or preset in a boot loader configuration file. The file specified with the `init=` kernel commandline parameter denominates the name of the executable right after kernel is up and ready for system execution.

The YaIniT **`init`** needs some customizable files, that will be read in or not according to the flexible program flow of init. They have to reside on a initramdisk or on a rootfilesystem, that is unconditionally accessed by the Linux kernel at system boot (see 3.3. The Linux kernel, 3.3.1 configuration and setup).

The YaIniT initramdisk ships in a compressed version named yainitrd.gz | yainitrd.bz2 | yainitrd.xz | yainitrd.lzm . With the commands gunzip, bunzip2, xz, unlzma, they are decompressed and with the

command ~/YaIniT-initrd # **`cpio -id <./yainitrd`**

an opened initramdisk or better an INITRAMFS in the current directory becomes accessible and customizable.

The different files, all part of the working YaIniT system are described as follows:

init : a file in plain text ASCII format containing the ASH statements of the system **`init`**.

#-#-#-#-#-#-#-#-#-#-#-#-#-#-#-# the following is work in progress #-#

general-functions : a file in plain text ASCII format containing ASH-functions valid for all following ASH scripts over the boot process of YaIniT. Many functions are kept sufficiently universal for use beyond YaIniT.

initvariables : a file in plain text ASCII format containing the definition of the system variables. Here are the names allocated and the variables are set to NULL or “”. Not explicitly listed and initialized system variables may later on fail the **`init`** program when a random system memory content is referenced and not the well defined and preset NULL or “”.

yainitdefaults : a file in plain text ASCII format containing the system variables, such as shell variables, the name of the initramdisk file, the default or minimal kernel version, the names of supported distributions,...... This file is mandatory and **`init`** ends with an error message if not existing or unreadable.

filesys-functions : a file in plain text ASCII format containing ASH-functions useful for all filesystem operations. For now detections for compressed swap partitions in RAM and the unmounting and mounting of system filesystems are implemented in a development state. This file is mandatory and **`init`** ends with an error message if not existing or unreadable.

initsys-functions : a file in plain text ASCII format containing ASH-functions and variables needed by them, necessary for the correct operation of **`init`**. Functions are used whenever similar tasks reoccur often. This file is mandatory and **`init`** ends with an error message if not existing or unreadable.

initrd-functions : a file in plain text ASCII format containing ASH-functions to integrate separable and recurring sections of code, necessary for the correct workflow during the **`init`** process. This file intents to provide a second level of abstraction. The ***initsys-functions*** file is loaded automatically during the execution of the functions from ***initrd-functions***. The ***initsys-functions*** and the ***general-functions*** files have to exist and are mandatory for the entire YaIniT boot. This file is mandatory and **`init`** ends with an error message if not existing or unreadable.

archvarfile : a file in plain text ASCII format containing the architecture specification (ARM, i386, IA64, x86_64, PPC, S390,....) , an identification string, the version of the used kernel, and some more architecture specific settings. The setting of the ARCH variable is unique and can't be ambiguous as the interpreting busybox and the belonging YaIniT Linux kernel must be compiled for this architecture. The user can set here the parameters, that are necessary to adapt at system architecture customization and distribution specification. This file is mandatory and **`init`** ends with an error message if not existing or unreadable.

$ARCH-functions : a file in plain text ASCII format containing ASH-functions and variables needed by them, necessary for the correct operation within the specified architecture in ***archvarfile***. If the user wants to optimize his busybox for a member of a processor family, than there must be a definition for the functions and variables for this optimization at run time. For run time optimization of the **`busybox`** runnable exists one unique mechanism for all processor architecture families within ***initrd-functions***. All that gets to be done is to implement for architecture and processor family the logic for the optimization of the specific processors, compile the different binaries of **`busybox`** with optimizing and processor specific parameters for gcc (see 3.2 BusyBox) and set the variables for it correctly. The default architecture is **i386** like in the Linux kernel's ./Makefile when compiling for x86 machines, and ***i386-functions*** is preprogrammed as example. This file is mandatory and **`init`** ends with an error message if not existing or unreadable.

modprobe-functions : a file in plain text ASCII format containing functions and variables needed by them, necessary for the loading of kernel modules, if kernel modules are present in the INITRAMFS in the filesystem tree at `/lib/modules/$VERSION_OF_RUNNING_KERNEL/` .

***rt.tar.bz2* | *rt.tar.xz* | *rt.tar.gz* | *rt.tar.lzma* |**
rt.tar.* : a file in a compressed tar archive format, with compression by bzip2, xz, lzma or gzip, containing the binaries of **`busybox`** adapted for the different processor families of the architecture.

***libmodules.tar.bz2* | *libmodules.tar.xz* | *libmodules.tar.gz* | *libmodules.tar.lzma* |**
libmodules-$KERNVER.tar.* : a file in a compressed tar archive format, with compression by bzip2, xz, lzma or gzip, containing the kernel modules for the running kernel with the full standard path :

```
/lib/modules/$KERNVER/./kernel/*
/lib/modules/$VERSION_OF_RUNNING_KERNEL/./kernel/*
```

This file will be uncompressed during boot and the modules will be made accessible for **`busybox`** modprobe / **`busybox`** insmod, by issuing the **`busybox`** depmod command.

Until today, **`busybox`** does not allow to specify the format of the modules.dep file, the paths to the modules remain only local paths!

branch-block-functions : a file in plain text ASCII format containing functions and variables needed by them, necessary for branching into the rootfilesystem. This file contains the algorithms necessary for finding and mounting the root blockdevice and switching to the new rootfilesystem.

branch-ram-functions : in old and deprecated shape, not yet sufficiently worked over

branch-net-functions : not yet implemented

branch-loop-functions : not yet implemented

devblockrc : a file in plain text ASCII format containing functions and variables needed by them, to branch to preinstalled rootfilesystem on a block device, a storage device formatted with one of the filesystems ext4, ext3, ext2, f2fs, btrfs, xfs, reiserfs.

devramrc : a file in plain text ASCII format containing functions and variables needed by them, to prepare the system for branching to a rootfilesystem, either loaded in RAM as a squashfilesystem in compressed form or as single file containing one of the allowed filesystems.

devnetrc : not yet implemented

devlooprc : not yet implemented

The use of the last eight scripts is under construction. For now at least two of them are necessary for a successful boot.

3.1.2 Logical steps and program flow

The YaIniT **`init`** follows the following logic:

PHASE 0: *set basic system variables and activate functions*

1. Right after boot, **`init`** sets the run phase parameter to 0 and defines the output prefix for the first message, which is written to a file named in **`init`**. The **`init`** process appears as Linux system process with number 1. If **`init`** is not syntactically correct programmed and not executable, the kernel crashes and the system is inoperable – without further messages or comments. That file, containing the first system messages, will be displayed or not according commandline parameter settings, later, if the files containing the required ASH-Functions are read in. Kernel command line parameters are read in from ***procfs*** (/procfs/cmdline) and the mode of message display is not known (quiet or verbose) before. After a successful boot, these messages can be viewed (debug mode) and saved (debug and rw parameter saves them to system root /).
2. If the file with the necessary functions named ***general-functions*** exists, then it will be read in. This file may not be an executable BusyBox ASH script with an explicit recall of /bin/busybox ash or /bin/bash. It

may contain only system variables, functions and executable statements for insertion at place. If this file does not exist, the system is searching a compressed tar-archive file, containing the system scripts, and tries to unpack it into the system root of the INITRAMFS. If this archive file does not exist, the system stops execution with an unconditional error message at the console. This error message must appear to inform the user about the system state.

The user is at an ordinary system prompt and can search for the ***general-functions*** file in a by hand accessible system prompt, mount a partition with **`busybox`** commands and read it in. With pressing <Ctrl> and <D> simultaneously at the keyboard, the program continues. If the keyboard or system console functionality is not compiled into the kernel, the system stop is final and needs reboot by hard-reset before system reconfiguration will be possible. If the kernel binary has not compiled in the necessary drivers for accessing system storages, there is no possibility to continue and read in the functions from a storage with this kernel. The only remaining way is to reboot with a working distribution and to repair the false setup by remaking the initrd file.

3. The **`init`** parameters are saved in a variable. For the case, the **`init`** script is recalled by itself, it will be replicated and configure itself for execution according these parameters. Not all kernel commandline parameters are accessible by the **`init`** script. "$@" contains all these parameters. In the standard configuration, only user defined parameters appear. The system parameters are accessible via the procfs filesystem, an internal filesystem containing informations on the system state.

   The Ubuntu kernel contains the feature to pass all kernel commandline parameters to **`init`** at invocation time.

   YaIniT provides a high performance feature to read all parameters to variables, without kernel modification if the ***procfs*** functionality is compiled in and they can be read from /proc/cmdline.

PHASE 1: *read in necessary functions and set up internal parameters*

1. **`init`** sets the run phase parameter to 1 and defines the output prefix for the next system messages. At this step, system boot is not continuable and no kernel modules are loadable, if the keyboard driver is not compiled in.
2. If the file, named ***initvariables*,** containing the definitions of basic system variables, exists, then it is read in and the variables are set: A pointer in system memory is created and the system can assign values to the variables during the run of **`init`**. All these variables are initially empty. If not, the system stops with an unconditional error message at the console. At this step, system boot is not continuable and no kernel modules are loadable, if the keyboard driver is not compiled in.
3. If the file with the initrd variables, named ***yainitdefaults*** exists, then it is read in. If not, the system stops execution with an unconditional error message at the console. This error message must appear to inform the user about the system state and an incorrectly configured INITRAMFS. The variables in ***yainitdefaults*** may set specified system variables in ***initvariables***.

   At this step, system boot is not continuable and no kernel modules are loadable, if the keyboard driver is not compiled in. Else, the user is at an ordinary system prompt and can search for the ***yainitdefaults*** file in a by hand accessible system prompt, create it or mount a relevant partition with **`busybox`** commands and read a replacement in. With pressing <Ctrl> and <D> simultaneously at the keyboard, the program continues. This message and all further following messages, this stop and all following stops of the **`init`** script, will be executed by functions from ***initrd-functions***. If the ***initrd-functions*** file does not contain these functions, then the system continues until it crashes or is no more operable. The ***initrd-functions*** file contains an automatic loading of the ***initsys-functions*** file with necessary functions for every routine running in the YaIniT system.
4. That function from ***initrd-functions***, that contains the file system tree specification and can generate the

rootfilesystem in INITRAMFS, is executed. Now should be existent all necessary directories in the actual rootfilesystem. For the generation of the directories, there is no verification. Necessary ones will be checked in the continuation.

5. The PATH variable with all possible directories containing executables is set and exported to other processes.
6. The filesystems procfs and sysfs are mounted, if not already mounted. The rootfilesystem, rootfs at “/” is unconditionally mounted for read-write and remount capability, to make and keep the system operable. If the mount commands are not successful, the system issues an error message and continues until it crashes or by other unforeseen intervention boots successfully.

   Now, after accessibility of `/sys` and `/proc`, it is possible to 1) check how the kernel is configured and determine the further execution of **`init`**, 2) access all system information necessary for the further recognition of hardware. The **`busybox`** command **`lspci`** to examine the PCI-Bus detected hardware does not work without `procfs` and `sysfs`.
7. The kernel version system parameters are set. The mechanism of the kernel version variable recognition requires a with one to three dots dotted string and preferably an extent separated by a '-' character, to specify the adaptation level of the kernel compilation. The top version number is before the first dot, the major version number is between the first and second dot, then after the second dot is the extra version and the YaIniT-specific extent. If not separated by a '-' or a '.', the mechanism is intelligent enough to detect the YaIniT-specific extent at the end of the string after the second dot. The mechanism is not intelligent enough to recognize that you don't have an adapted kernel if a valid YaIniT-extent is at the end of your kernel name. In this case you might need kernel commandline parameter **`noauto`** to avoid the kexec mechanism searching and loading another kernel.
8. The kernel commandline is read and all parameters are set to system variables for being processable immediately after this function call. Where necessary, they need to be translated to system variables and exported to all branches of **`init`** (child processes).
9. If **`init`** is not definitely set by kernel commandline parameter for quiet boot, then the file containing all previous **`init`** system messages is displayed at the system console.

   Note, that during a regular boot with no errors of configuration, the user gets until this point no messages displayed.
10. The commandline parameter **`distro`** is checked and if it is set to a distribution name listed in the variable SUPPORTED_DISTROS from ***yainitdefaults***, then the distribution specific parameters are read in and will be respected during the following program flow.
11. The root device, containing the rootfilesystem, which may be an installed GNU/Linux distribution, is prerecognized and the relevant system variables are set. These are necessary, if a distribution contains really a strange configuration, that needs exceptional interpretation. At this moment, the root device is not touched or verified.
12. If it is recognizable from the `root=` parameter, that the root device is the system RAM, then the YaIniT system branches out of **`init`**. First, **`init`** is replicated to **`initrc`** and then it is executed with keeping all detected parameters for reconfiguration to a so called Live System, running completely in RAM.

PHASE 2: *set machine specific parameters, best adapted executable and device properties*

1. **`init`** sets the run phase parameter to 2 and defines the output prefix for the next system messages.
2. If the file with the specifications of the system architecture ***archvarfile*** exists in the INTRAMFS, then this file is read in. If not, then the system stops and drops the user to the system prompt. A corrective action like in phase 1.2. can be done by hand at the system prompt with the **`busybox`** applet command potential.

3. If the variable ARCH from ***archvarfile*** is specified and set, and another file named ***$ARCH-functions*** exists, then this file is read in. If not, the program continues without being able to optimize and to fulfill any customization on architecture needs at busybox level (see 3.2 BusyBox again). The busybox files in a compressed tar archive will however be extracted and deleted, if exactly that **`busybox`** file, for replacement with the existing one, cannot be found.
4. If the kernel parameter `noauto` is not specified, and the architecture specific system variables and functions are read in, then the special one **`busybox`** compilation for the current executing system architecture will be found and replace the running one.

   Caution!

   If there is a bad compilation, with wrong gcc parameter combinations resulting in an executable causing run time errors, then the system crashes, without warning. The debug parameters `debug` and `initdbg` at the kernel commandline allow to help detecting the exact error, but a reconfigured version of YaIniT needs to be rebuild.
5. If there is a file generated with the squash-filesystem, containing kernel modules for the running kernel version, then this file will be mounted as a loop device and the kernel modules will be made accessible (**`depmod`**, Phase 3). Squashfs files containing kernel modules must be contained in the INITRAMFS and keep a filename of type, specified in ***yainitdefaults***.
6. The Linux device filesystem `/dev/*` is set up with the command **`busybox mdev -s`** , for the case, the kernel feature of automatic `/dev` tree generation is not activated. All device nodes found by compiled in kernel features and **`busybox`** auto detection should be accessible now.
7. Perform an unconditional system process synchronization of all the preceding actions in Phase 2.

PHASE 3: *set machine specific parameters, best adapted executables and device properties*

1. **`init`** sets the run phase parameter to 3 and defines the output prefix for the next system messages.
2. Set the symlinks for the **`busybox`** executable.
3. Delete any init, that exists in INITRAMFS as `/sbin/init, /usr/sbin/init` or `/bin/init, /usr/bin/init`, referring to **`busybox`**, except the /**`init`** in the root directory of INITRAMFS before mounting the target rootfilesystem and branching. This to assure, that during the initial boot only the YaIniT system is used and the appropriate **`init`** of the boot target system will be found and executed for its control, when issuing the **`chroot`** or **`switch_root`** command to invoke the target system as it is.
4. Any possibly overwritten executable for system shell (`/bin/ash,/bin/sh`) is switched back to `/bin/busybox ash` and referred with a symbolic link to it.
5. For the case, that in the rootfilesystem exist kernel modules from kernels with version number less than the default kernel version or different from the running kernel, they can be, specified by parameter and purged from the rootfilesystem to avoid a memory overflow. In this step, all kernel modules residing in INITRAMFS only with outdated versions and different from the running kernel are purged. This happens only in system RAM and it is assured that no unwanted deletion of hard disk contents occur.
6. If a tar archive in the format ***-\`uname -r\`.tar.* ,** is found, this file will be extracted and should fill the standard directory /lib/modules/\``uname -r`\`/kernel with kernel modules. Especially the name ***libmodules-\`uname -r\`.tar.**** is checked. Any other path configuration would fail the system. These kernel modules are available from INITRAMFS for all following conditional use of hardware. This file can contain additional YaIniT system executables, but it is recommended to place them into the rt tarball (rt.tar.xz), containing the different versions of **`busybox`**.

   If you decide to restrict your YaIniT- kernel for the recognition of compressions for only one of the possible compression formats, identified by the file extension, you can do so. The system will deny the extraction and stop some steps further, if not all necessary kernel functions are available and there is need for module loading at initial boot.

7. If a squashfilesystem containing kernel modules is present in the INITRAMFS, this file is mounted in the regular place for the kernel modules. The file must follow the convention

   `*-$TOPKERNVER.$MAJORKERNVER.$MINORKERNVER.$EXTRAKERNVER$SEPARATOR$EXTENT-*.$SQUASHFSEXTENT`.

   where **`$TOPKERNVER.$MAJORKERNVER.$MINORKERNVER.$EXTRAKERNVER$SEPARATOR$EXTENT`** is identic with the output of busyboxes **`` `uname -r` ``** command and $SEPARATOR can be one of { . | - }. The appropriate squashfs file for the running kernel will be selected and mounted

   in the place /lib/modules/$KERNVER/kernel, all other files of the form

   *-***$TOPKERNVER.$MAJORKERNVER.$MINORKERNVER$SEPARATOR$EXTRAKERNVER-*.$SQUASHFSEXT***

   will be purged away. *$SQUASHFSEXTENT* is the last extension of the filename, a free specifiable variable in the ***yainitdefaults*** configuration file and overrideable by the ***archvarfile***. Defaults to “sfs”. The squashfs file is normally generated by the system generation scripts (see **`do-make(g)-ash`**).

   If the configuration of the kernel's squashfs compression does not coincide with compression or the format used by **`mksquashfs`** command from squashfs-tools, there will be no further action, only a warning. Replacement for kernel parameters is searched as in the previous step.

8. Check, if there are kernel modules present in `/lib/modules`. If yes, then read in the ***modprobe-functions*** file, containing the functions for loading modules according to **`YaIniT`**'s needs. For missing and not properly compiled in modules, there will be no warnings, only system stop with advice to reconfigure.

9. Probe for drivers and load them. This is done in the order:
   - load the mainboard, BIOS Chip and CPU drivers to make hardware recognizable
   - load monitor drivers
   - load drivers for AHCI/SATA/PATA Storage Controller Devices
   - load **USB** drivers for external storage and devices
   - bring up **USBHID** and USB devices, Keyboard, Mouse, Touchpad, USB-Controller
   - bring up PCI-ISA-Bridge i8042, PS/2-connector, PS2-Mouse, PS2_Keyboard
   - load **Firewire** drivers, keyboard, mouse, storages
   - load (if necessary) legacy input device driver, serial mouse,keyboard, xt-keyboard,..
   - load required filesystem drivers
   - load network drivers
   - load sound card drivers

     Recent changes in the driver modules for the graphical devices and the video stack of the Linux kernel (since ver. 3.0 and the VGA-switcheroo), made these sequence necessary. There are drivers, that can be loaded only once before others and not be reloaded if these others are temporarily unloaded. Further are there many drivers, that do not work without mandatory access to specific ACPI routines, which is a compromise against the structural ideas of Linux and free software.

10. Apply the **`busybox`** commands **`lspci`** and **`lsmod`** for detection of unused kernel modules and unload them.

11. Probe for system storage, that can contain system files and load the necessary driver.

12. Drop to system prompt before leaving the /**`init`**, if the `initdbg` parameter for debugging is specified.

13. Check whether the system boot target specification is a storage media with a GNU/Linux installation, or the system RAM.

    If it targets system RAM, then the specification of the rootdevice is root=/dev/ram0;   This branch is already done before, if the specification is correct.

    If the rootdevice is an installed system on an accessible block storage device, then the specification looks like root=LABEL=...... | root=UUID=..... | root=PARTUUID=.......
    | root=/dev/disk/by-name|-label|-uuid|-id | /dev/sd* | /dev/hd* | /dev/sr* .

14. Conditionally branch out of `init` here. This is done by replicating init to initrc and invoking it with the parameter for insertion of ***devblockrc***. If the ***branch-direct-functions*** and the ***devblockrc*** script files exist, insert the ***devblockrc*** script file for sophisticated recognition of rootdevices on storage filesystems.

15. Bring up and identify the root device, mount it and find the appropriate command to reinvoke the **`init`** (either **`chroot`** or **`switch_root`**).

16. `init` does allow, according checked parameters, to be replicated in INITRAMFS under another name and to be reexecuted to itself. Other functions and variables can be read in for continuation.

#-#

An installed GNU/Linux system, that resides on a block device, has to be accessed with the ***devblockrc*** script file.

Linux system conception allows the user to boot, with a certain kernel configuration and without the use of an INITRAMDISK file for booting, directly into an installed system on a filesystem and device with compiled in modules (specify   'root=PARTUUID=partitionuuid-partitionnumber' at the kernel commandline).

A target boot system, that resides on a rootfilesystem, as a file system in a single file (formatted as a ext2, ext3, ext4, btrfs, xfs, reiserfs, squashfs), which has to be mounted in the Linux FHS/VFS tree, continues booting into RAM with ***devramrc,*** where the necessary functions will be provided to access the system resources in a more differentiated way. This logic is described in the following section.

#-#

3.1.3 Integration of customizations for hardware and distribution

3.2 BUSYBOX again: setup and configuration

3.2.1 Configuration

The configuration of BusyBox is quite universal.

For the PC hardware standard for desktop and server machines, there was, for now, not found any reasonable need to configure something different than a single static binary for YaIniT, containing all applets accessible as symbolic links to the **`busybox`** binary. Since the first MMX capable processor, the Pentium-MMX with 256 MB

RAM, this ever worked, with intelligent and performant ASH scripts as in YaIniT, at astonishing performance compared to shipped solutions from distributions.

However, the configuration of BusyBox as a static binary does not allow, without extensive modifications of the setup, splitting the applet set to many executables, such that at any stage of a YaIniT boot the needed BusyBox applets are present in different binaries with names different from **`busybox`**. BusyBox does allow binary applets with a shared library in ELF-Format, which makes BusyBox slower.

3.2.2 Predefined Setup and Optimization

3.3 Linux KERNEL again: setup and configuration

The Linux kernel standard model, as downloadable by the Internet, disposes on its own setup mechanism.

3.3.1 Configuration

The YaIniT kernel needs to fulfill several configuration conditions to boot properly with all the scripts in the YaIniT package.

1. Kernel Name convention:

What appears with the **`uname -r`** shell command, cannot be reoriginated exactly anyway to what is written in the kernel's Makefile at the top for the parameters KERNEL_VERSION, MAJOR_KERNEL_VERSION, MINOR_KERNEL_VERSION, EXTRA_KERNEL_VERSION and needs to fulfill a special convention for YaIniT:

1. The field EXTRA_KERNEL_VERSION has to contain a dot before text name : “.”

2. The automatically generated configuration file `.config` with **`make xconfig`** or **`make menuconfig`** may not contain a dot or nothing in the CONFIG_LOCALVERSION field before a name string, it has to be a minus sign : “-” . The following string is used to identify the processor family specific optimization of the kernel binary.

For example, the special configuration of a kernel for SSE3 capable processors, will need something like “-sse3” in the CONFIG_LOCALVERSION field of the .config file. As the realtime patch modifies the local version field by an extra file, it is necessary to modify realtime patch or its produced extra file for local version to maintain naming convention. The kernel extension field is used to identify the kernel according to its special gcc parameters in compilation. The same extension qualifier relates to the kernel parameters and specifies a subset of processors in the architecture and processor family. The configuration for the generation of at boot time selectable busybox executables follows the same convention.

2. Configuration conventions

BusyBox ASH Scripts to control the configuration are part of YaIniT, as well as a set of kernel .config file sections for the different processor families.

The kernel local configuration files appear in YaIniT in the form of DOT.config-* . Whenever a script of YaIniT detects such a file with the right extension after the “minus”, than this configuration file will be copied into the place of .config in the kernel root.

The scripts, as described under 3.3.2, all need a well configured and runnable **`busybox`** binary at `/bin/busybox`.

3.3.2 Predefined Setup and Optimization

Good introductions to Linux kernel administration include a detailed description of the kernel setup and the configuration scripts. One of the properties of Linux is, that the kernel binary is generated by a special use of the linkage editor (ld, gnu-ld, gold) and generative scripts controlling the kernel setup size.

Kernel `Makefile` Modification :

YaIniT disposes on an intelligent kernel `Makefile` modification algorithm, which does the job for modifying the `Makefile`, such that the x86-processor specific gcc-parameters specified in a configuration file and inserted by a patch will be used for the bzImage binary and different, further enhanced gcc-parameters for the kernel modules. For kernel sections requiring the special **`cc`** direct assembly feature of the gcc compiler, such as setup and entry, there is a special modification, keeping the gcc “-march=” setting together with the “-mpreferred -stack-boundary={2|3|4}” code segmentation specification. Since the version 4.8.2, the -m96bit-long-double from SSE2 instructions is fully accessible by “-mpreferred-stack-boundary=3”. Herewith it is guaranteed, that the code of a YaIniT kernel keeps a homogeneous structure. Only for the kernel modules, an extended use of the processor specific SSE command set is possible within the Linux kernel. The kernel binary needs strict soft-math (-msoft-float) or fpu-math (-mhard-float, -mfpmath=387), and no SSE use is possible, without breaking the internal code size restriction and segmentation control. This is specific for x86 processors in the i386 architecture, claiming for a code format with Time Stamp Counting as well as without, and with Physical Address Extensions and Page Allocation Tables as well as without. The MMX and 3DNow! instruction set was found to compile in perfectly. These instructions include the 64 bit register use such that an inadmissible code size variation does not occur. Therefore, Linux kernels for x86-Processors since the Pentium III processor are configured with as many modules as possible for a best use of SSE instruction performance. For the graphical drivers it is important to check whether enabling “-mfpmath=sse”, “-fpmath=387,sse” or only “-fpmath=387” is significant in performance. Since the introduction of the AVX instruction set, the Linux kernel binary can be compiled like the modules with full use of SSE instruction set and 128/256 Bit Floating-Point-Arithmetics included, by release of gcc-4.8.0. Anyway, there is needed an AVX capable machine to benefit from the full SSE integration in the Linux kernel. With “-msse2avx” the gcc-4.8 does convert all SSE-Math into AVX instruction sequences and lines them up to a stack boundary of four 32-bit registers (4 double words) in all code sections, modules and kernel binary. AVX2 with eight 32-bit registers is only implemented since gcc-4.9.0 and the 256-bit register arithmetic is not yet recommended and well approved. For kernels with the use of AVX-Processor extensions, this feature avoids at least a performance loss by switching between the SSE-Registers for modules and the x87-FPU for binary kernel floating point operations. However, this administrative effort was elaborated by the author so far, that most of the performance loss, caused by the so called monolithic standard kernel model design, can be alleviated.

These modifications are done by some scripts, running with the BASH and the ASH in **`busybox`** as well, and are described later.

Since the version of GCC-4.9.0 and higher and the kernel release 3.14.8, the YaIniT shipping standard includes a patch for directly configuring the x86-Processor features. The scripts are worked over to keep the automatic kernel compilation run with all messages tracked in a file, but having no automatic patching. See the appendix at 4.5.1 for more detailed description of the patch.

Restrictions on the Kernel Configuration by YaIniT :

Since the Linux kernel version 3.10, it is possible to switch off by configuration any access to an ASCII plain text format file, containing the magic `#!` sequence as its first bytes, before the path and the executable, to be invoked for the interpretation of the rest of file content. Since some longer time of Linux kernel development, the standalone executables and static libraries can be made non-executable by omitting the a.out format in the same section of .config file.

YaIniT does generally **not** work without the kernel scripting executable feature → CONFIG_BINFMT_SCRIPT=y.

Therefore, the Linux kernel needs to be configured for YaIniT with at least the configuration variables CONFIG_BINFMT_ELF=y CONFIG_BINFMT_SCRIPT=y CONFIG_HAVE_AOUT=y CONFIG_BINFMT_AOUT=y in the section of executable file formats and emulations, and CONFIG_BINFMT_MISC=y/m, if the kernel size is not absolutely the restrictive factor.

Only for final and secure installations in specific production environments, the Linux kernel may be configured without the INITRAMFS (CONFIG_BLK_DEV_RAM=y) to compile in the kernel binary. Embedded systems, without any access of INITRAMFS can normally leave it unused. YaIniT scripts for kernel compilation will operate with higher performance only in system RAM, running the entire compilation of Linux kernels completely in RAM.

For cleanly mounting file systems and having no trouble with missing modules, it is strongly recommended to compile in into the kernel the following modules :

CONFIG_BLK_DEV_LOOP=y , CONFIG_BLK_DEV_RAM=y , CONFIG_AUTOFS4_FS=y

Omitting or insufficiently setting the control group support in the scheduler configuration will make the kernel inoperable on greater distributions such as OpenSuSE since version 12.1. Since the use of `systemd` as init executable is no longer to circumvent, the kernel configurations for these distributions must contain control group scheduling (CONFIG_CGROUPS=y).

To enable module loading is definitely needed. Only for a complete kernel by configuration adaption to an embedded system with well defined interfaces and no other devices possible to connect than the compiled in drivers allow. This is a rare case! But the YaIniT init mechanism is well prepared for that and only a few if decisions can be eliminated as superfluous.

CONFIG_MODULES=y , CONFIG_MODULE_UNLOAD=y, CONFIG_MODVERSIONS=y

Not to circumvent for the general availability of YaIniT is the use of the INITRAMFS, therefore:

CONFIG_BLK_DEV_INITRD=y , CONFIG_INITRAMFS_SOURCE=y
Otherwise it would be impossible to start the execution of `busybox` as the `init` program after the kernel is coming up to execution for userspace. Only the direct boot into a rootfilesystem would be allowed, and there the execution of `busybox` is depending on the CONFIG_BINFMT_SCRIPT mechanism. Any preparation or selection of the boot target during the boot process will be impossible.

If you want to use the YaIniT kernel for general recognition of all possible hardware such as supported by modules in the Linux kernel, there is the need for extracting modules to the /lib/modules directory or to mount a squashfs file, containing the required modules, at the right place. Dealing with modules during `init` time with YaIniT requires the appropriate compression compiled into the kernel:

CONFIG_RD_*=y , CONFIG_HAVE_KERNEL_*=y

The configuration for an embedded system can crash the YaIniT kernel in use with PC standard machines. It is recommended: CONFIG_EMBEDDED=n

For allowing adapted or most universal kernel features CONFIG_EXPERT=y is necessary.

Since a long time, it is necessary to allow general access to SCSI Block Devices, SCSI Generic Devices,

CONFIG_BLK_DEV_BSG=y , and if necessary CONFIG_BLK_DEV_BSGLIB=y

for all disk devices. The generic SCSI Devices are recognizable as /dev/sg entries.   This requires at least a minimal SCSI devices configuration: CONFIG_SCSI=m/y , CONFIG_CHR_DEV_SG=m/y , CONFIG_BLK_DEV_SD=m/y , CONFIG_BLK_DEV_SR=m/y . Without having at least these options modular, there will be no possibility to access disk block devices.

The system devices have to be brought up, as there is no predefined /dev directory tree with device nodes prebuilt. Therefore : CONFIG_DEVTMPFS=y , CONFIG_DEVTMPFS_MOUNT=y ,

Without CONFIG_HOTPLUG_PCI=m/y , the recognition of a processor and the main board may fail.

A kernel without support of TCP/IP is almost impossible. All what is configurable as drivers, may be unloaded: Protocols, Network device drivers, Quality of Service controls, firewalling IPTables. The local network loopback device `lo` cannot be eliminated and is an essential part of the Linux system, needed by many distributions. CONFIG_NET=y , CONFIG_ NETCORE=y   , CONFIG_NETDEVICES=y

The KEXEC feature needs some configuration to run: CONFIG_KEXEC=y , CONFIG_KEXEC_JUMP=y need to be both enabled, otherwise KEXEC does not work or the jump to the loaded new kernel and switching back to the old in case of failure will not be possible. The KEXEC feature to load an other kernel without recurring the BIOS, does not work, if the memory map for firmware is not allocated in the system filesystem sysfs. Therefore:

CONFIG_FIRMWARE_MEMMAP=y

If you do not enable the KEXEC feature, nothing happens, the YaIniT boot procedure exits with a message and the system continues to run with the now running kernel.

With a series of newer motherboards, the booting and module loading mechanism does not work, if the drivers from the X86 Platform Specific Driver Section are not present to expose the system information of DSDT tables to the drivers. There should be at least :

CONFIG_X86_PLATFORM_DEVICES=y , CONFIG_ACPI_WMI=y/m , CONFIG_INTEL_RST=m , CONFIG_INTEL_SMARTCONNECT=m , CONFIG_APPLE_GMUX=m , CONFIG_MXM_WMI=m , CONFIG_ASUS_WMI=m , CONFIG_MSI_WMI=m , CONFIG_HP_WMI=m , CONFIG_DELL_WMI=m , CONFIG_ACER_WMI=m

To allow the necessary module autoloading at system initialization, it is obvious that : CONFIG_DMIID=y , CONFIG_I2C=y/m , CONFIG_I2C_CHARDEV=y/m , CONFIG_I2_HELPER_AUTO=y , CONFIG_PCI_REALLOC_ENABLE_AUTO=y ,

Useful Almquist Shell Scripts to optimize kernel performance on Processor Families :

The YaIniT kernel optimization does not take place in applying some secret patches, with the ultimate performance enhancements, others will dream from for the next ten years. Instead, the profitable optimization is done by allowing the maximum tuneability by setting the parameters of gcc.

`functions-ash-dm`   : contains all ASH functions, that are necessary in the following YaIniT scripts

Needs a working **`/bin/busybox`**

`do-mkramfs`   : to create an INITRAMFS filesystem for setting up and compiling the kernel completely within system RAM. This is much faster!

$1=part of filename of kernel containing file, usually specifying the kernel version. e.g.: `do-mkramfs 3.11` leads to a file `linux-3.11.tar.* ,` which is extracted for following automated patching, auto-configuration and compilation.

$2=directory containing the patches to be applied

$3=another directory, e.g. with configuration files or with the AUFS files

This script creates an INITRAMFS and copies all the files and directories into RAM, necessary for creating kernels within YaIniT. The tar ball must contain the Linux kernel structure, and the second parameter an existing directory with the patches to be applied.

Needs a working **`/bin/busybox`** and the file containing all necessary functions, named `./functions-ash-dm`

`do-rmramfs` : Removes by unmounting an INITRAMFS created by `do-mkramfs`. Obsolete, when rebooting anyway to test the new kernel(s).

Needs a working **`/bin/busybox`**

`apply-patches-ash` : 1. Extract the tarball with the Linux kernel. 2. Apply all the patches, sorted alphabetically, in the first level under the patch directory. The second level to store the obsoleted ones or those to test. 3. Write the protocol of the patching run into the file `patchings.log` and copy this file into the Linux kernel root directory.

$1=kernel version, specifying the Linux kernel directory by the qualifier in ./linux-*

$2=directory with all the patches to apply.

$3=directory with the AUFS filesystem. In this directory may only be AUFS files, that are copied into the kernel directory tree. After this copy, patches in $2 must change the kernel for being to be configured with AUFS.

Needs a working **`/bin/busybox`** and the file containing all necessary functions, named `./functions-ash-dm`

`do-make-ash` : 1. Setup the kernel for optimization run. Read configuration file and create patch for Makefile. Check for a configuration file DOT.config-* with the right optimization extension. If it is not in the kernel root directory, look in ./configs/* . 2. Copy the DOT.config-x file to .config. Run a complete kernel make, with bzImage, modules and modules_install in sequence (does not run a simple **`make`**). Store the gcc messages of compiler output into protocol file. 3. Create a directory ./installed and copy the kernel binary with all modules and the output-protocol therein. 4. Create a squashfs filesystem containing all the kernel modules of this make run and copy it to ./installed. 5. Clean up the kernel directory tree. 6. Repeat the preceding, if more or all kernel configurations are specified.

Parameters:

$1=part of directory name after linux- , usually containing the kernel version

$2=the extension, identifying the optimization with th gcc-parameters specified under the extension name in the file `./makevars-ash.conf`.

$3=another extension or for the next run, or the keyword `resume`, if the kernel make run has been interrupted and can be restarted.

$4=another extension for after next run

$5 - $9 = more extensions for after next runs, $10 is forbidden or interpreted as “'$1'0” !

Needs a parameter file `./makevars-ash.conf`, containing all the necessary information with gcc-parameters and the ASH-Functions to pass the parameters at the right place.

Needs a working **`/bin/busybox`** and the file containing all necessary functions, named `./functions-ash-dm`

`do-makeg-ash` : Does the same as **`do-make-ash`**, but relies on more generic configuration parameters. These are in the file `./makevars-x86generic-ash.conf`

Needs a working **`/bin/busybox`** and the file containing all necessary functions, named `./functions-ash-dm`

The user or administrator may alter the specification files according to his wants. New sections require appropriate entries in the system variables.

3.4 KEXEC again: setup and configuration

3.4.1 Configuration

KEXEC is not very much optimizable like the kernel is. KEXEC contains kernel specific program structures, which respect parts of the kernel such as setup and entries. Therefore, it is recommended to compile KEXEC in a generic way (-march=i386 or -march=i686 , -mtune=generic). For accessing and extracting the initramdisk files by the KEXEC loader, there are shared libraries necessary such as zlib, liblzma, libzo, which must be made known to KEXEC at compile time and therefore be contained in the INITRAMFS file. This makes the YaIniT initrd file bigger. When using the KEXEC mechanism from the rootfilesystem, you can refer to the respective functions by copying manually the file to the rootfilesystem in debug mode of YaIniT from INITRAMFS, when using a union-filesystem from the yainit-initramfs directory in the root directory.

3.4.2 Predefined Setup and Optimization

3.5 TOOLS again: setup and configuration

3.5.1 Configuration of `configator`

3.5.2 Use of `configator`

3.5.3 Configuration of `dialog`

The dialog feature runs only with the VGA screen and the VESA driver, which should be compiled into the kernel. If the

3.5.4 Configuration of `xdialog`

3.5.5 Configuration of `yad`

4. APPENDIX

4.1 On the usefulness of examples and templates from free software

An extensive search with an Internet search engine for Linux kernel tuning and optimization leads to a variety of sites.
One of the first and easy to apply occasions is the RealTime section of the Linux Kernel. We find the current patches at the kernel ftp server ftp://ftp.kernel.org/pub/linux/kernel/projects/rt with some antique appearing writings at the RealTime site, https://rt.wiki.kernel.org/ . A sharper view on this site reveals the annuals from the Real-Time experts meetings, delivering sufficient documentation to stimulate the developers' fantasy for own realtiming.

The Arch-Linux repository of packages, named AUR, is a site, where an astonishing number of different adaptation can be found (https://aur.archlinux.org/packages/kernels) and applied with success.

4.2 The AUR Kernel Repository of Arch-Linux

The AUR Repository of Kernel patches is a very helpful site, where aside from the direct links to the developer sites, the interesting patches, that make void the difficulties, can be downloaded.

My own annotations to the explanations and definitions from the web are marked by this font and typing.

4.2.1 Linux kernel post-factum patches / pf.natalenko.name

The pf-patches are a rather reliable patchset for the optimization of the mainline kernel. In my works, I have found necessary to apply the original patches, when merging with patchsets from the big distributions. Own changes, that take a lot of time until really working, are not to avoid on the way to an optimal solution for a Real-Time Kernel.

The post factum patch consists of a series of distributed patches and is the adapted composition of actual patches

1. The so called CK-patchset from the Australian programmer Con Kolivas with the BFS
2. The Budget-Fair-Queuing I/O-Scheduler BFQ
3. The TUX-On-Ice power management extensions
4. The UKSM, Universal-Kernel-Same-Page-Merging patch for the ease of high workloads, which produce lots of duplicate pages, that are not removed throughout the whole RAM-Memory without this patch.

CK-Patchset:

Brain Fuck Scheduler

From Wikipedia, the free encyclopedia

This article is about the task scheduler. For the similarly named but unrelated programming language, see Brainfuck.

Brain Fuck Scheduler Developer(s)	Con Kolivas
Stable release	0.447 / May 6, 2014[1]
Written in	C
Operating system	Linux
License	GPL (free software)
Website	kernel.kolivas.org

The Brain Fuck Scheduler (BFS) is a process scheduler designed for the Linux kernel in August 2009 as an alternative to the Completely Fair Scheduler and the O(1) scheduler.[2] BFS was created by veteran kernel programmer Con Kolivas.[3]

The objective of BFS, compared to other schedulers, is to provide a scheduler with a simpler algorithm, that does not require adjustment of heuristics or tuning parameters to tailor performance to a specific type of computation workload. The BFS author asserted that these tunable parameters were difficult for the average user to understand, especially in terms of interactions of multiple parameters with each other, and claimed that the use of such tuning parameters could often result in improved performance in a specific targeted type of computation, at the cost of worse performance in the general case.[4] BFS has been reported to improve responsiveness on light-NUMA (non-uniform memory access) Linux mobile devices and desktop computers with fewer than 16 cores.[citation needed]

Shortly following its introduction, the new scheduler made headlines within the Linux community, appearing on Slashdot, with reviews in Linux Magazine and Linux Pro Magazine.[2][5][6] Although there have been varied reviews of improved performance and responsiveness, Con Kolivas does not intend for BFS to be integrated into the mainline kernel.[3]

The CK-Patchset contains further the patch for exacter gcc-adaptation of the x86-kernel architecture specifics, which is found here:

https://github.com/graysky2/kernel_gcc_patch

Kernel_gcc_patch

This kernel patch adds additional CPU options to the Linux kernel accessible under: Processor type and features ---> Processor family --->

Why a specific patch? The kernel uses its own set of CFLAGS, KCFLAGS. For example, see:

- arch/x86/Makefile
- arch/x86/Makefile_32.cpu
- arch/x86/Kconfig.cpu

CPU Family	GCC Optimization
Native optimizations autodetected by GCC	-march=native
AMD K10-family	-march=amdfam10
AMD Family 10h (Barcelona)	-march=barcelona

AMD Family 14h (Bobcat)	-march=btver1
AMD Family 15h (Bulldozer)	-march=bdver1
AMD Piledriver Family 15h (Piledriver)	-march=bdver2
AMD Family 16h (Jaguar)	-march=btver2
Intel 1st Gen Core i3/i5/i7-family (Nehalem)	-march=nehalem
Intel 1.5 Gen Core i3/i5/i7-family (Westmere)	-march=westmere
Intel 2nd Gen Core i3/i5/i7-family (Sandybridge)	-march=sandybridge
Intel 3rd Gen Core i3/i5/i7-family (Ivybridge)	-march=ivybridge
Intel 4th Gen Core i3/i5/i7-family (Haswell)	-march=haswell
Intel 5th Gen Core i3/i5/i7-family (Broadwell)	-march=broadwell

BFQ:

BFQ is a proportional-share storage-I/O scheduler that also supports hierarchical scheduling with a cgroups interface. Here are the main nice features of BFQ.

Low latency for interactive applications

According to our results, whatever the background load is, for interactive tasks the storage device is virtually as responsive as if it was idle. For example, even if one or more of the following background workloads are being served in parallel:

one or more large files are being read or written,

a tree of source files is being compiled,

one or more virtual machines are performing I/O,

a software update is in progress,

indexing daemons are scanning the filesystems and updating their databases,

starting a command/application or loading a file from within an application takes about the same time as if the storage device was idle. As a comparison, with CFQ, NOOP, DEADLINE or SIO, and under the same conditions, applications experience high latencies, or even become unresponsive until the background workload terminates (especially on SSDs).

Low latency for soft real-time applications

Also soft real-time applications, such as audio and video players or audio audio- and video-streaming applications, enjoy about the same latencies regardless of the device load. As a consequence, these applications do not suffer from almost any glitch due to the background workload.

High throughput

BFQ achieves up to 30% higher throughput than CFQ on hard disks with most parallel workloads, and about the same throughput with the rest of the workloads we have considered. BFQ achieves the same throughput as CFQ, NOOP, DEADLINE and SIO on SSDs.

Strong fairness guarantees

As for long-term guarantees, BFQ distributes the throughput as desired to I/O-bound applications (and not just the device time), with any workload and independently of the device parameters..

http://algo.ing.unimo.it/people/paolo/disk_sched/description.php

TuxOnIce:

TuxOnIce is most easily described as the Linux equivalent of Windows' hibernate functionality, but better. It saves the contents of memory to disk and powers down. When the computer is started up again, it reloads the contents and the user can continue from where they left off. No documents need to be reloaded or applications reopened and the process is much faster than a normal shutdown and start up.

TuxOnIce has a long feature list, including the ability to cancel hibernating or resuming by pressing Escape, image compression to save time and space, a versatile plugin architecture, support for machines with Highmem, preemption and SMP.

The TuxOnIce website (this one) and mailing lists provide support for dealing with issues.

The primary author of TuxOnIce is Nigel Cunningham. A huge thanks must also go to Bernard Blackham, Florent Chabaud, Pavel Machek, Gabor Kuti and Michael Frank along with many others who have tested and contributed to the development of TuxOnIce.

http://tuxonice.nigelcunningham.com.au/

UKSM:

Linux kernel has a feature named KSM(Kernel SamePage Merging). it lets the hypervisor system share identical memory pages amongst different processes or virtualized guests. However, it has its limitation such like high CPU usage and slow responses to workload change. So here comes UKSM. With a revolutionary algorithm redesign, UKSM has many advanced features: Full system scan. It automatically scans all user processes’ anonymous VMAs. Before UKSM, a process need to call KSM kernel API to submit its memory areas to KSM for scan. This makes many legacy software other than KVM can NOT benefit from KSM. Now, UKSM scans whole system applications including KVM. All programs benefit from UKSM without even knowing how it works. Super quiet CPU usage. It automatically detects rich areas containing abundant duplicated pages. Rich areas are given a full scan speed. Poor areas are sampled at a reasonable speed with very low CPU consumption usually under 1%. Our benchmarks show that even for CPU intensive workloads, it has a negligible performance impact. Ultra scan speed and CPU efficiency. A new hash algorithm is proposed. As a result, on a machine with Core(TM)2 Quad Q9300 CPU in 32-bit mode and 800MHZ DDR2 main memory, it can scan memory areas that does not contain duplicated pages at speed of 627MB/sec ~ 2445MB/sec and can merge duplicated areas at speed of 477MB/sec ~ 923MB/sec. For a bunch of busy workloads creating lots of duplicated pages, this means, with UKSM, you no long suffer from deadly swapping which is observed in KSM enabled setting. Thrashing area avoidance. If a VM is constantly writing the same data to the duplicated pages, KSM suffer a problem of thrashing, i.e. the pages it merges will soon get copied again. In this situation, the CPU is purely wasted without having much memory saving. UKSM can perfectly avoid this situation by detecting the thrashing areas. In short, if you are managing a Linux server with lots of KVM virtual machines or containers(e.g. OpenVZ, LXC), UKSM is a must have feature! If you are desktop user, you also benefit a lot from it.

(via: http://kerneldedup.org/en/projects/uksm/introduction/).

4.2.2 Linux kernel RealTime patches / ftp.kernel.org/pub/linux/kernel/projects/rt

The regular Realtime Kernel found under

https://www.kernel.org/pub/linux/kernel/projects/rt/

contains the patching for Basic and Full Real-Time Preemption in the Linux Core code.

Note: Except of some small extracted patches becoming part of the mainline kernel, a successful *combination of the Linux Real-Time-Kernel with the other available schedulers and security patchsets requires some fundamental reprogramming of greater kernel sections and is yet unknown. Further unknown are patches for the scheduler management, that allow the coexistence of many different main schedulers (Control-Group-Scheduler, Round-Robin-Scheduler, Brain-Fuck-Scheduler, Fully-Preempting-Real-Time-Scheduler,....), not only the block device I/O schedulers (NoOPerations-Scheduler, Deadline Scheduler, Completely-Fair-Queuing-Scheduler, Budget-Fair-Queuing-Scheduler). Even the modularization of the I/O-Schedulers leads to system crashes with some configurations, when changing the scheduler during runtime (CFQ->BFQ, than BFQ->Deadline with CONFIG_IOSCHED_NOOP=y, CONFIG_DEFAULT_NOOP=y, CONFIG_IOSCHED_DEADLINE=m, CONFIG_IOSCHED_BFQ=m, CONFIG_IOSCHED_CFQ=m).*

4.2.3 Linux kernel GRSecurity and PaX patches / grsecurity.net

The grsecurity is the only patchset, which intends to enhance the security of the Linux kernel against hard attacks such as mentioned in the following.

Grsecurity is an extensive security enhancement to the Linux kernel that defends against a wide range of security threats through intelligent access control, memory corruption-based exploit prevention, and a host of other system hardening that generally require no configuration. It has been actively developed and maintained for the past 13 years. Commercial support for grsecurity is available through Open Source Security, Inc.

http://grsecurity.net/

grsecurity is a set of patches for the Linux kernel with an emphasis on enhancing security. Its typical application is in web servers and systems that accept remote connections from untrusted locations, such as systems offering shell access to its users.

PaX

Resides on the same domain and is a project in tight collaboration with grsecurity.

http://pax.grsecurity.net/

A major component bundled with grsecurity is PaX, which is a patch that, amongst other things, flags data memory, such as that on the stack, as non-executable, and program memory as non-writable. The aim is to prevent executable memory pages from being overwritten with injected machine code, which prevents exploitation of many types of security vulnerabilities, such as buffer overflows. PaX also provides address space layout randomization (ASLR), which randomizes important memory addresses to hinder attacks that rely on such addresses being easily known. PaX is not itself developed by the grsecurity developers, and is also available independently from grsecurity [1].

Role-based Access Control

Another notable component of grsecurity is that it provides a full Role-based access control (RBAC) system. RBAC is intended to restrict access to the system further than what is normally provided by Unix access control lists, with the aim of creating a fully least-privilege system, where users and processes have the absolute minimum privileges to work correctly and nothing more. This way, if the system is compromised, the ability by the attacker to damage or gain sensitive information on the system can be drastically reduced. RBAC works through a collection of "roles". Each role can have individual restrictions on what they can or cannot do, and these roles and restrictions form a "policy" which can be amended as needed.

A list of RBAC features:

Domain support for users and groups

Role transition tables

IP-based roles

Non-root access to special roles

Special roles that require no authentication

Nested subjects

Variable support in configuration

And, or, and difference set operations on variables in configuration

Object mode that controls the creation of setuid and setgid files

Create and delete object modes

Kernel interpretation of inheritance

Real-time regular-expression resolution

Ability to deny ptraces to specific processes

User and group transition checking and enforcement on an inclusive or exclusive basis

/dev/grsec special device for kernel authentication and learning logs

Next-generation code that produces least-privilege policies for the entire system with no configuration

Policy statistics for gradm

Inheritance-based learning

Learning configuration file that allows the administrator to enable inheritance-based learning or disable learning on specific paths

Full pathnames for offending process and parent process

RBAC status function for gradm

/proc/<pid>/ipaddr gives the remote address of the person who started a given process

Secure policy enforcement

Supports read, write, append, execute, view, and read-only ptrace object permissions

Supports hide, protect, and override subject flags

Supports the PaX flags

Shared memory protection feature

Integrated local attack response on all alerts

Subject flag that ensures a process can never execute trojaned code

Full-featured fine-grained auditing

Resource, socket, and capability support

- Protection against exploit bruteforcing
- /proc/pid filedescriptor/memory protection
- Rules can be placed on non-existent files/processes
- Policy regeneration on subjects and objects
- Configurable log suppression
- Configurable process accounting
- Human-readable configuration
- Not filesystem or architecture dependent
- Scales well: supports as many policies as memory can handle with the same performance hit
- No runtime memory allocation
- SMP safe
- O(1) time efficiency for most operations
- Include directive for specifying additional policies
- Enable, disable, reload capabilities
- Option to hide kernel processes

Chroot Restrictions

grsecurity restricts chroot in a variety of ways to prevent a variety of vulnerabilities, privilege escalation attacks, and to add additional checks and balances.

Chroot Modifications:

- No attaching shared memory outside of chroot
- No kill outside of chroot
- No ptrace outside of chroot (architecture independent)
- No capget outside of chroot
- No setpgid outside of chroot
- No getpgid outside of chroot
- No getsid outside of chroot
- No sending of signals by fcntl outside of chroot
- No viewing of any process outside of chroot, even if /proc is mounted
- No mounting or remounting
- No pivot_root
- No double chroot
- No fchdir out of chroot
- Enforced chdir("/") upon chroot
- No (f)chmod +s

- No mknod
- No sysctl writes
- No raising of scheduler priority
- No connecting to abstract Unix domain sockets outside of chroot
- Removal of harmful privileges via capabilities

Miscellaneous Features

grsecurity also adds enhanced auditing to the Linux kernel. It can be configured to audit a specific group of users, audit mounts/unmounts of devices, changes to the system time and date, chdir logging, amongst other things. Some of these other things allow the admin to also log denied resource attempts, failed fork attempts, and exec logging with arguments.

Trusted path execution is another optional feature that can be used to prevent users from executing binaries that are not owned by the root user, or are world-writable. This is useful to prevent users from executing their own malicious binaries or accidentally executing system binaries that could have been modified by a malicious user (being world-writable).

grsecurity also hardens the way chroot "jails" work. A chroot jail can be used to isolate a particular process from the rest of the system, which can be used to minimise the potential for damage should the service be compromised. However, there are ways to "break out" of a chroot jail. grsecurity attempts to prevent this.

There are also other features that increase security and prevent users from gaining unnecessary knowledge about the system, such as restricting the dmesg and netstat commands to the root user [2].

List of additional features and security improvements:

- /proc restrictions that don't leak information about process owners
- Symlink/hardlink restrictions to prevent /tmp races
- Hardlink restrictions to prevent users from hardlinking to files they do not own
- FIFO/Named pipe restrictions
- dmesg(8) restriction
- Enhanced implementation of Trusted Path Execution
- Group-based socket restrictions
- Nearly all options are sysctl-tunable, with a locking mechanism
- All alerts and audits support a feature that logs the IP address of the attacker with the log
- Stream connections across unix domain sockets carry the attacker's IP address with them (on 2.4 kernels only)
- Detection of local connections: copies attacker's IP address to the other task
- Automatic deterrence of exploit bruteforcing
- Pre-defined Low, Medium, High, and Custom security levels

Tunable flood-time and burst for logging

<u>The self definition of the PaX project work, is copied in here:</u>

1. Design

The goal of the PaX project is to research various defense mechanisms against the exploitation of software bugs that give an attacker arbitrary read/write access to the attacked task's address space. This class of bugs contains among others various forms of buffer overflow bugs (be they stack or heap based), user supplied format string bugs, etc.

It is important to realize that our focus is not on the finding and fixing such bugs but rather on prevention and containment of exploit techniques. For our purposes these techniques can affect the attacked task at three different levels:

(1) introduce/execute arbitrary code
(2) execute existing code out of original program order
(3) execute existing code in original program order with arbitrary data

For example the well known shellcode injection technique belongs to (1) while the so-called return-to-libc style technique belongs to (2).

Introducing code into a task's address space is possible by either creating an executable mapping or modifying an already existing writable/executable mapping. The first method can be prevented by controlling what can be mapped into the task and is beyond the PaX project, access control systems are the proper way of handling this. The second method can be prevented by not allowing the creation of writable/executable mappings at all. While this solution breaks some applications that do need such mappings, until they are rewritten to handle such mappings more carefully this is the best we can do. The details of this solution are in a separate document describing NOEXEC.

Executing code (be that introduced by the attacker or already present in the task's address space) requires the ability to change the execution flow using already existing code. Such changes occur when code dereferences a function pointer. An attacker can intervene if such a pointer is stored in writable memory. Although it would seem a good idea to not have function pointers in writable memory at all, it is unfortunately not possible (e.g., saved return addresses from procedures are on the stack), so a different approach is needed. Since the changes need to be in userland and PaX has so far been a kernel oriented project, they will be implemented in the future, see the details in a separate document.

The next category of features PaX employs is a form of diversification: address space layout randomization (ASLR). The generic idea behind this approach is based on the observation that in practice most attacks require advance knowledge of various addresses in the attacked task. If we can introduce entropy into such addresses each time a task is created then we will force the attacker to guess or brute force it which in turn will make the attack attempts quite 'noisy' because any failed attempt will likely crash the target. It will be easy then to watch for and react on such events. The details of this solution are in a separate document describing ASLR.

Before going into the analysis of the above techniques, let's note an often overlooked or misunderstood property of combining defense mechanisms. Some like to look at the individual pieces of a system and arrive at a conclusion regarding the effectivenes of the whole based on that (or worse, dismiss one mechanism because it is not efficient without employing another, and vice versa). In our case this approach can lead to misleading results. Consider that one has a defense mechanism against (1) and (2) such as NOEXEC and the future userland changes in PaX. If only NOEXEC is employed, one could argue that it is pointless since (2) can still be used (in practice this reason has often been used to dismiss non-executable stack approaches, which is not to be confused with NOEXEC however). If one protects against (2) only then one could equally well argue that why bother at all if the attacker can go directly for (1) and then the final conclusion comes saying that none of these defense mechanisms is effective. As hinted at above, this turns out to be the wrong conclusion here, deploying both kinds of defense mechanisms will protect against both (1) and (2) at the same time - where one defense line would fail, the other prevents that (i.e., NOEXEC can be broken by a return-to-libc style attack only and vice versa).

In the following we will assume that both NOEXEC (the non-executable page feature and the mmap/mprotect restrictions) and full ASLR (using ET_DYN executables) are active in the system. Furthermore we also require that there be only PIC ELF libraries on the system and also a crash detection and reaction system be in place that will prevent the execution of the attacked program after a fixed (low) number of crashes. The possible venues of attack against such a system are as follows:

- attack method (3) is possible with 100% reliability if the attacker does not need advance knowledge of addresses in the attacked task.

- attack methods (2) and (3) are possible with 100% reliability if the attacker needs advance knowledge of addresses and can derive them by reading the attacked task's address space (i.e., the target has an information leaking bug).

- attack methods (2) and (3) are possible with a small probability if the attacker needs advance knowledge of addresses but cannot derive them without resorting to guessing or a brute force search ('small' can be further quantified, see the ASLR documentation).

- attack method (1) is possible if the attacker can have the attacked task create, write to and mmap a file. This in turn requires attack method (2), so the analysis of that applies here as well (note that although not part of PaX per se, it is recommended among others, that production systems use an access control system that would prevent this venue of attack).

Based on the above it should come as no surprise that the future direction of PaX will be to prevent or at least reduce the efficiency of method (2) and eliminate or reduce the number of ways method (3) can be done (which will also help counter the other methods of course).

2. Implementation

The main line of development is Linux 2.4 on IA-32 (i386) although most features already exist for alpha, ia64, parisc, ppc, sparc, sparc64 and x86_64 as well and other architectures are coming as hardware becomes available (thanks to the grsecurity and Hardened Gentoo projects). For this reason all implementation documentation is i386 specific (the generic design ideas apply to all architectures though).

The non-executable page feature exists for alpha, i386, ia64, parisc, ppc, sparc, sparc64 and x86_64 while ppc64 can have the same implementation as ppc. The mips and mips64 architectures are hopeless in general as they have a unified TLB (the models with a split one will be supported by PaX). The main document on the non-executable pages and related features is NOEXEC, the two i386 specific approaches are described by PAGEEXEC and SEGMEXEC.

The mmap/mprotect restrictions are mainly architecture independent, only special case handling needs architecture specific code (various pieces of code that need to be executed from writable and therefore non-executable memory, e.g., the stack or the PLT on some architectures). Here the main document is MPROTECT whereas EMUTRAMP and EMUSIGRT describe the i386 specific emulations.

ASLR is also mostly architecture independent, only the randomizable bits of various addresses vary among the architectures. The documents are split based on the randomized region, so RANDKSTACK and RANDUSTACK describe the feature for the kernel and user stack, respectively. RANDMMAP and RANDEXEC are about randomizing the regions used for (among others) ELF libraries and executables, respectively. The infrastructure that makes both SEGMEXEC and RANDEXEC possible is vma mirroring described by the VMMIRROR document.

Since some applications need to do things that PaX prevents (runtime code generation) or make assumptions that are no longer true under PaX (e.g., fixed or at least predictable addresses in the address space), we provide a tool called 'chpax' that gives the end user fine grained control over the various PaX features on a per executable basis.

4.2.4 Linux kernel l-pa / https://aur.archlinux.org/packages/linux-l-pa

An extensive search with an Internet search engine for Linux kernel tuning and optimization leads as well to the Arch-Linux repository of packages, named AUR, where an astonishing number of different adaptation can be found (https://aur.archlinux.org/).
One of these adaptations is a Run-Time Kernel with patching the Makefile and Kconfig files for the use of adapted gcc-parameters: packages / kernels : *linux-l-pa* , hosted by forum member Ninez (or Niñez).
The patch, that should modify the `Makefile` to have compiled the kernel with special gcc-parameters, specified in the configuration file `.config` does not work. If making the bzImage with the verbosity V=1 option (`make bzImage V=1`) in the patched kernel source tree, the gcc output shows the persistent use of the default compiler parameters, such as specified during the compilation of the distribution's gcc (`gcc -v`). This disrupts the kernel code consistency and may lead to random crashes on some processor core machines. Further, the linux-l-pa package contained patches for the semaphore feature, that made the kernel in version 3.10 run instable and crash. Lots of other patches were unusable, resulting in a not running Linux kernel binary.

It is recommended to read the patches before applying them and to use the proposed patch for x86-kernel configuration as contributed by YaIniT and circumscribed under 4.5.1.

4.3 Big Distributions and their useful kernel patches

4.3.1 The OpenSuSE Linux kernel patches

OpenSuSE shows a lot of effort in maintaining a patch list with own modifications for

1. enhanced hard disk support with extensions and modifications to the disk manager and the RAID management together with special SCSI disks management
2. A stack unwind modification of the stack handling
3. The KVM (Kernel Virtual Machine) and XEN Version 3 kernel virtualization, according SuSE usable
4. Rich-ACL support for the VFS-Stack and the EXT4 filesystem
5. Apparmor patch, correlating with the Apparmor-patches from Ubuntu, and enabling the Version 5 of the network feature
6. OVERLAYFS integration as the Layerfs to achieve the use of existing installations in read-only-mode with the possibilities to store all changes aside, but having them effective
7. Some special hardware integrations and adaptations
8. .. ..

4.3.2 The Gentoo Linux kernel patches

Gentoo publishes frequently the triple of patches for any new release of the distribution supported kernels. These versions are called the base, the experimental and the hardened patches. The core features are

1. Special hardware support for the IBM Thinkpads
2. Some tricky corrections to the kernel core functions
3. GRSecurity, SELinux and PAX patches integration
4. BFQ – Budget Fair Queuing I/O-Scheduler integration patches
5. . .. .

4.3.3 The Mageia Linux kernel patches

Mageia adds to any released version of the Mageia (mga) kernel a lot of useful patches.

1. Similar to Ubuntu a separate 3rd-party directory is maintained, with compatibility patches for ndis/ndis-wrapper, AuthenTec AES2501 Fingerprint Sensor Driver support, Acer Netbook keyboard hardware support, a VIA High Speed Serial little kernel module, which enables high speed serial port modes of VIA VT82C686A or VT82C686B southbridge-equipped motherboards
2. Layer-Filesystems, both, AUFS and OVERLAYFS support
3. Patches to keep alive ATI-MACH64 graphic cards
4. The functioning IFWLOG netfilter features
5. Useful patches for the SCSI stack
6. and a lot more of patches to ease hardware support
7. . . ..

Find the actual Mageia publications under

http://mirrors.eu.kernel.org/mageia/distrib/cauldron/

4.3.4 The Ubuntu Linux kernel patches

The Ubuntu Linux kernel patches are numerous and fill many Megabytes, the biggest size of patches among all the distributions.

Find them at

http://mirrors.eu.kernel.org/ubuntu/pool/main/l/linux/

1. A separate Ubuntu kernel section with Layer-Filesystems support, both, AUFS and OVERLAYFS, with a worked over Intel Graphic driver for i915 and some rare hardware support
2. Numerous driver enhancing and correcting patches
3. Own Apparmor features
4. …....

4.3.5 The Fedora Linux kernel patches

With a greater number of patches targeting at the drivers and more of hardware integration the Fedora kernel patchset contains some important changes for the core kernel.

1. Switch off superfluous messages
2. Override the automated support of floppy disks
3. Numerous driver enhancing and correcting patches
4. Fedora specific modifications
5. . . . .

4.4 Restrictions on Kernel Configurations by GNU/Linux distributions

Remarkable and voluntarily, proprietary distributions and with preference the derivatives of bigger and “hardened” distributions, do restrict kernel configurations, necessary for booting into their installed systems, with the intention to augment the effort necessary for concurrent administrators or intelligent users, that want their modifications to spread over as many distros as possible.

This is definitely not accelerating the intelligent evolution of Free and Open Software, resulting in an easy configurable and adaptable operating system with many as user friendly as possible features.

I suggest:

The loss of patent protection for software in the last instance, has to be taken into account from the beginning on of any development of free and open software.

Again and again bigger enterprises demonstrate through the news channels and Online-Newspapers, during the concurrency of market shares by the commercially sold closed code software, a strong claim for patent protection for software.

Distributors, that intent an acceptable sales revenue for their products, need an inexpensive replacement for patent protection without bearing the danger to risk fines for a probable corruption of state employees and illegal competition.

4.5 Comments on the adaptive changes for YaIniT

There are several changes of GNU/Linux original configurations and developments provided with the YaIniT Tool, such as presented under

https://github.com/initiate414/

4.5.1 The more exact Linux kernel configuration, applying the `yainit-x86-config.patch`

4.5.2 Linux kernel configuration with the patching scripts `do-make-ash` and `do-makeg-ash`

4.5.3 Applying the patchets from bigger distributions

5. INDEX

6. GLOSSARY