- Slax -
- handbook -

most recent version of this documentation can be found at
http://www.slax.org/documentation.php

Contents

What is Slax

Slax is a Live operating system based on Linux. Live means it runs from an external media without any need for permanent installation. Slax boots from USB mass storage devices such as Flash Drive keys as well as from regular hard drives and CD/DVD discs. Simply plug your device in and boot from it. Entire Slax operating system resides in a single directory `/slax/` on your device, making it easier to organize with your other data.

Slax provides KDE4 desktop environment already preconfigured for the most common tasks. Included are applications and tools for data recovery, multimedia playback, instant messaging, web and more. Slax has zram support activated by default, which is a new technology for automatic RAM compression. Thanks to it, Slax runs on computers with as low as 48MB of RAM (in text mode). You can put Slax on wide range of different filesystems, including EXT (ext2,ext3,ext4), btrfs, and even FAT and NTFS.

When Slax is started from a read-only media such as CD/DVD, it keeps all system modifications in memory only, and all the modifications are lost when you reboot. On the other hand, if you run Slax from a writable device such as USB Flash Drive, it stores all changes there, so all your configurations and modifications are restored next time you boot, even if it is on a different computer. This feature is known as Persistent Changes and you can read more about it in a separate chapter.

Choosing optimal architecture (32bit / 64bit)

You probably noticed that Slax is available for 32bit and 64bit processor architectures. The 32bit version is designed to run on very old computers (as old as Intel 486; that dates back to 1989). It will run properly on brand new computers too, but it is limited in the amount of RAM it can see . max 3GB of RAM will be accessible, even if your computer has more. On the other hand, the 64bit version has no such limits, but will not run on computers older than few years - your CPU must understand 64bit instructions in order to run it. So in general, if you plan to use Slax on very old computer, or you seek for a system with maximum possible compatibility, then choose 32bit. If your intention is to use Slax on brand new machine, you'll better go for 64bit. And if you don't know your target hardware, go for 32bit to be sure that it will run everywhere, keeping in mind the fact it won't see more than 3GB of RAM (which is sufficient in all common cases).

System requirements to run Slax

	Slax 32bit version	**Slax 64bit version**
Processor:	i486 or newer CPU, all Intel processors and AMD processors will work	An x86_64 CPU, like AMD Athlon 64, Opteron, Sempron, Intel Core 2/i3/i5/i7, and others
Memory:	48 MB of RAM for text-mode 256 MB of RAM for KDE desktop	64 MB of RAM for text-mode 256 MB of RAM for KDE desktop
Peripherals:	CD or USB drive to boot from	CD or USB drive to boot from
Optionally:	network card, sound card	network card, sound card

Source code and license of Slax

Slax is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License (GPL) as published by the Free Software Foundation. Slax is distributed in the hope that it will be useful, but without any warranty; use at your own risk. The GNU GPL license requires that all source codes are published so others could reuse it, modify or learn from it. You can trust me, this is very good idea - without it, there would be no Slax at all. Slax source code is publicly available for download. Furthermore, Slax itself shares a lot of code with Slackware, whose source code is as well publicly available.

Slax on CD/DVD

If you plan to run Slax from a CD or DVD disc then you need to download Slax as an ISO file. In fact, the ISO file is a complete image of a CD, so what you need to do is to burn it to a CD/DVD media. Actually it doesn't matter if you choose CD or DVD, both will just work. The most important part is that you can't burn it as a regular file. That wouldn't work. Instead, you have to burn it as a disc image. In Windows 7 for example, just right-click the ISO file and select `Burn disc image` from the context menu. On older windowses, you'll need some special software for the task, for example you can try Free ISO Burner. When done, put Slax CD/DVD disc to your CD/DVD drive and reboot. You may need to press some key to show a boot menu while your computer starts and select to boot from CD/DVD. That magic key which shows you the boot menu is usually `F11`, `F9` or `Esc`, consult your BIOS documentation or watch onscreen instructions when your computer reboots to make sure.

Slax on USB device or hard disk

In order to run Slax from USB device or from hard drive, you should download Slax as a ZIP archive. When downloaded, extract the ZIP archive directly to your disk's root directory. It will create a single folder `/slax/` on your device. When done, one more step is required in order to make the drive bootable: navigate to `/slax/boot/` directory on your USB device or hard disk and locate `bootinst.bat` file there. This file contains boot installer program, so just run it by double clicking at it, it will make all the necessary changes to your device's master boot record so your computer's BIOS could actually understand how to boot Slax from your disk.

Next follow the same procedure like if you were booting from CD - reboot your computer and choose to boot from the USB drive or hard disk in your computer's boot menu. Again, you may need to consult your BIOS documentation to find out how to boot an operating system on your computer from your desired device.

Slax boot options

Before Slax itself starts loading, you can see a big clover image in the middle of your screen. This is the boot logo. It is displayed for a short while, and you have exactly four seconds to press `Esc` key during that time in order to fine tune the way how Slax is going to boot. Pressing `Esc` will invoke a simple boot menu like the following:

```
        Run Slax
        [*] Persistent changes
        [*] Graphical desktop
        [ ] Copy to RAM
```

You may use this menu to disable persistent changes, make Slax run in text-mode only, or copy Slax data to RAM during startup. Use arrow keys to navigate and `Enter` key to select (toggle) any option. When done, press `Enter` on the "Run Slax" item to continue.

Persistent changes

By default, Slax detects if you run it from a writable device. If yes, then all the changes you make to the operating system itself are saved and restored next time you boot. All file modifications to Slax itself are saved into a special file `changes.dat`, which is created on your boot device in `/slax/changes/` directory, and grows automatically in size up to 4GB. If you, for any reason, do not like this feature, simply uncheck the appropriate boot option and your Slax will start using the default 'fresh' configuration. It may be useful also in cases you'd like to test something system-wide, since you can always revert to the default state by simple reboot (in case things screw up).

The file `changes.dat` is designed to work even on FAT filesystems, which are commonly used on most USB flash drives. Unfortunately FAT is limited to 4GB file size; for that reason, persistent changes can't grow more. In case you need to save more, please format your storage drive with some Linux filesystem such as EXT4 or BTRFS and install Slax to it. Slax will be able to save changes natively and will be limited only by the actual capacity of your device. Persistent Changes functionality does not (of course) affect files on hard drives in your computer. If you modify these files, they will always be modified regardless of your persistent changes settings.

Shutting down Slax safely

When Slax is running, it reads system data from the device it booted from. If you're using Persistent Changes then Slax even writes data to your boot device. Unplugging or ejecting it would make the operating system crash. Due to that, you can unplug the boot device only after your computer is switched off or reboots to other operating system. Similarly, if you access your computer's hard drives while running Slax, those will stay mounted and will be marked as 'in use'. Be sure to always shutdown Slax properly, either from KDE menu or using `poweroff` or `reboot` commands, and always wait until the system ends. Especially if you have Windows installed on your computer, those will detect improper shutdown of your hard drives and may warn you about that.

Running Slax from memory

There may be situations though when you need to unplug the boot device as soon as possible while keeping Slax running. This is indeed possible; it requires your computer to load (copy) all Slax data to RAM memory during startup, so it is accessible even after your boot device is no longer plugged in. In order to put this "Copy to RAM" feature into action, make sure the boot option in Slax boot menu is checked (to copy data to RAM), and the option Persistent Changes is unchecked (to skip writing your chages to the boot device). The time needed to start Slax will increase, since it will need to copy the entire `/slax` directory from CD or USB to your computer's memory, but then it will run Slax from there, letting you disconnect your boot device. Your computer will need at least 512 MB of RAM to hold all Slax data while still having enough free RAM for the operating system itself. Remember that even if you run Slax from memory, you have to properly shut it down when needed in order to safely unmount your hard drives (if any).

Cheatcodes for Slax

Boot parameters (also known as cheatcodes) are used to affect the boot process of Slax. Some of them are common for all Linuxes, others are specific for Slax only. You can use them to disable desired kind of hardware detection, to start Slax from hard drive, etc. To use cheatcodes, press `Esc` key to activate boot menu during Slax startup as usual, and when you see the boot menu, press `Tab`. A command line will appear at the bottom of the screen, which you can edit or add new boot parameters at the end:

Cheatcode	Meaning	Example
acpi=off	Disable ACPI	acpi=off
nohotplug	Disable all hardware auto detection	nohotplug

from=	Load Slax data from specified directory
	or even from an ISO file
		from=/slax7/
		from=/Downloads/slax.iso

nosound	Mute sound on startup	nosound
toram	Activate Copy to RAM feature	toram
perch	Activate Persistent Changes feature	perch
debug	Enable Slax startup debugging	debug

Separate commands by space. See manual pages `man bootparam` for more common cheatcodes.

Slax modules

Slax modules (also known as bundles) extend Slax by additional software or data. Slax modules differ significantly from other Linux packages you might know, so it is important to explain that difference. Contrary to other packaging systems, Slax modules do not need unpacking. They are used in the packed form. Instead of installing, Slax modules are activated. In technical terms, that means mounted and added to aufs union as a new branch. But don't worry if these technical details mean nothing to you, you don't need to care about it at all in order to use Slax modules.

Obtaining modules for Slax

There are several ways how to obtain Slax modules. Probably the easiest is to search at Slax website in the Modules section and download your desired module from there. Modules are divided into several categories for easier navigation. Each item has links leading to 32bit or 64bit version, so make sure that the architecture version you're downloading matches your Slax. If unsure what architecture is your Slax using, see `/etc/slax-version` while running Slax to find that out.

Activating (installing)

Offline module activation is performed when Slax is not running. You can activate module (a file with `.sb` extension) by copying it to `/slax/modules/` directory on your boot device. All modules copied there will be automatically activated during Slax startup. Removing the module (the file with .sb extension) from `/slax/modules/` directory will uninstall it so it is not a part of Slax any longer.

Online module activation is performed when Slax is running, directly within Slax system. You can either use Software Center to activate and deactivate modules on the fly (currently in development) or you can use special commands, as explained in the next section.

Command line interface

You can use command line tools to manage modules while running Slax. Run `konsole` or simply login into the text mode prompt first. From there, use the below described commands to activate a previously downloaded module as well as to deactivate any module which is no longer in use. You can also search for modules in the repository and you can even activate a module which is not downloaded yet.

slax search

To search in Slax repository for your desired software, use:

```
slax search [ keyword ]
```

where [keyword] is the name of the software you're looking for, or just a keyword describing it. You may even use several keywords enclosed by quotes, such as "keyword1 keyword2 keyword3". After you hit `Enter`, you'll get response from Slax server with a list of modules matching your search criteria. As an example, if you search for "editor" keyword, you may get an output similar to the following:

```
root@slax:~# slax search "editor"
elvis:Elvis is a text editor
jed:programmer's editor
joe:Joe text editor
jove:Jonathan's Own Version of Emacs
nano:Nano's ANOther editor, an enhanced free Pico clone
vim:Vi IMproved
```

As you can see, the first word before semicolon is a module name, and the rest is a short description of the given module.

slax info

If you wish to see some more detailed information for any module, use the following command:

```
slax info [ name ]
```

where [name] is the name of the module you're interested in. Lets assume, for example, that you wish to get some more details about the 'vim' module. Using `slax info vim`command, you will get some similar output like the following:

```
root@slax:~# slax info vim
name: vim
version: 7.3.645
required bundles:
maintainer: Tomas M
last update: 2012-12-22 21:46:15
categories: console, editors
size: 6544 KB
description: Vi IMproved ...
```

As you can see, there are several details about the module, like name, version, other modules required to run this one (none for vim at the moment), maintainer name and email address, last update date, category where the module belongs to, size of the module (approximate) and some longer description (which has been truncated in this example).

slax download

When you find a module you were searching for, you can download it by using the following command:

```
slax download [ name ] [ dir ]
```

where [name] is (again) the name of the module you're downloading, and the last parameter [dir] is the directory where you wish to save the module file. It may be, for example, `/tmp` or `.` (current directory).

Beware that the module will be saved under a filename prefixed by ID, such as /tmp/124-name.sb. It is wise to preserve the numerical prefix since Slax Software Center relies on that in order to recognize what modules are currently activated. After the modules is downloaded, the current filename (including the numerical prefix) is printed on the screen.

slax activate

Use the following command to activate Slax module:

```
slax activate [ file or name ]
```

If the given parameter [file] exists and is a readable file, it will be activated. If you provide [name] instead while the [name] is not any existing file, the module of the given name will be firstly downloaded and activated afterwards (if available in the repository, of course). Technical details of module activation are explained in Slax Internals section. Yet there is one important detail which needs to be noted here.

It is not technically possible to activate a module while it resides inside the active AUFS root filesystem (which is the effective root filesystem in Slax). Due to that, all module files need to be moved outside the AUFS root. But don't worry, Slax does everything for you automatically during the activate operation. This is mostly noticable if you download a module using the web browser in Slax, it gets stored in your Downloads folder, and the activation command will move it to `/mnt/live/memory/modules` or to `/mnt/live/data/slax/` when it is writable.

Build Scripts

Slax modules are made mostly by Slax users themselves. To ensure quality and transparency, the only way to contribute a module is to submit so called build script. The build script is a short bash script which handles the entire module creation, starting by downloading appropriate source codes from the Internet, unpacking them, configuring, compiling and packaging. As soon as a buildscript is submitted to Slax server, it gets processed on the server and binary modules are created for all supported Slax architectures automatically. All build scripts are available for download.

If you just wish to compile some software for Slax for your own use, then you are free to make it by hand, not using any build scripts at all. But if you like to share your software with other Slax users, you are required to write a proper build script. Binary modules are not accepted to the repository.

Each buildscript has to contain several variables defined in order to let the automatic build system understand what software is it going to compile, what is its version, what category should it assign to, where to download source files from, what other Slax modules should be activated in order for the software to compile, and so on.

Build script template

To make it easier for users to create build scripts, there is a template available in Slax. It contains description of each required variable and further instructions. The template is stored in `/usr/share/slax/template.SlaxBuild`. To make sure you are always using the most recent build script, it is recommended to download its newest version by issuing the following command:

```
slax buildscript download template
```

The newest version of template buildscript will be downloaded to `template.SlaxBuild` in current directory.

Downloading existing build scripts

You can download current build script for any module available in the repository. That way, you can easily discover how is the software configured or compiled. You can also examine existing buildscripts to learn some tricks or examples how to configure or compile software on Slax, but remember that some of the methods used in someone else's build scripts may be not recommended or may be even discouraged. The following command is used to download build script for existing module:

```
slax buildscript download [ name ]
```

where [name] is the name of Slax module for which you're downloading the buildscript. The script will be saved to `name.SlaxBuild` in current directory.

Uploading new build scripts

After your build script is ready to be shared with others, feel free to upload it to Slax server. Remember that the entire script will become publicly available. All submitted build scripts are subject to GNU GPL license. Upload your build scripts using:

```
slax buildscript upload [ file ]
```

where [file] is path to your build script's file. The real file name on disk doesn't matter since the build script will be always stored under the name specified in `SLAX_BUNDLE_NAME` variable, which is defined inside the build script. When you upload your build script, you'll be prompted for password, and all further write operations with this build script will require you to enter the password again. This ensures that nobody else will be able to modify your build script on server.

Updating existing build scripts

You can update or delete your existing build scripts using one of the following commands:

```
slax buildscript update [ file ]
slax buildscript delete [ name ]
```

where [file] is the filename of your buildscript you are updating and [name] is the buildscript name you would like to delete on Slax server.

Slax internals

All Slax data files are located on the boot media in a single directory. It is no surprise that the name of that directory is 'slax'. All the magic happens inside. Here is an overview of simplified directory structure; directories are red, some interesting files are mentioned as well, using *italic*:

```
slax
├─── boot
│    ├─── isolinux.bin
│    ├─── syslinux.cfg
│    ├─── initrfs.img
│    ├─── vmlinuz
│    └─── ...
├─── changes
├─── rootcopy
├─── 01-core.sb
├─── 02-xorg.sb
├─── 03-kdeps.sb
├─── 04-kde.sb
└─── ...
```

Booting the Linux kernel

When your computer's BIOS boots Slax, it actually just runs SYSLINUX boot loader. The boot loader itself is stored either in file isolinux.bin or ldlinux.sys, depending on your boot media - CD/DVD uses isolinux.bin, USB disk or hard drive uses ldlinux.sys.

As soon as the SYSLINUX boot loader is executed, it learns what to do next from its configuration file (you guessed it) syslinux.cfg. In Slax, this configuration file contains instructions to show some cool boot logo and optionally provide boot menu if the user hits a key before timeout. When the timeout counter reaches zero or the user exited boot menu, SYSLINUX boot loader loads two files into memory: vmlinuz (Linux kernel) and initrfs.img (base root filesystem). The progress is indicated by continuous stream of dots printed on screen. Once the files are loaded, the vmlinuz binary is executed to start the Linux kernel.

Pre-init

Under normal conditions (when a standard Linux distribution is starting from a hard drive), the Linux kernel would mount root filesystem from the hard drive and /sbin/init would be executed as the main process which takes care of system startup. In Slax, the situation is different - there is no hard drive to mount the root filesystem from, yet the kernel surely needs some init process to be started. For that purpose, Slax carries a base filesystem in initrfs.img file - it is a compressed CPIO archive with some directories and files inside, including core Linux tools (commands) and the desired init.

So after the Linux kernel has successfully initialized and has a full control of your computer, its last task is to find the mentioned CPIO archive in memory (it was loaded there from file initrfs.img by syslinux boot loader as you surely remember), extract it (into a memory area which acts as a temporary root filesystem, called initramfs) and execute temporary /init process from there.

Escaping initramfs

At this moment, we have a fully initialized Linux Kernel running, initramfs area in memory is populated by a temporary root filesystem with just the most basic Linux commands, and temporary init just started.

Having the temporary root filesystem in initramfs is not ideal, since it doesn't support pivot_root system call - an important operation which will be used later in the boot up process. We need to switch from initramfs to something else. To do that, the temporary init firstly mounts a tmpfs filesystem over /m, moves all files and directories in there including the init script itself, and uses switch_root to make this tmpfs /m the new root and to restart the init itself from there too. Blue star denotes the directory which is moved.

initramfs as root: ->

```
(initramfs)
/
├─── bin
│     ├─── sh
│     ├─── mount
│     └─── ...
├─── dev
├─── lib
│     ├─── cleanup
│     └─── ...
├─── mnt
├─── m (tmpfs) ★
├─── memory
├─── proc
├─── sys
└─── init
```

initramfs after move to tmpfs: ->

```
(initramfs)
/
└─── m (tmpfs) ★
      ├─── bin
      │     ├───sh
      │     ├───mount
      │     └─── ...
      ├─── dev
      ├─── lib
      │     ├─── cleanup
      │     └─── ...
      ├─── mnt
      ├─── memory
      ├─── proc
      ├─── sys
      └─── init
```

tmpfs after switch_root:

```
(tmpfs) ★
/
├─── bin
│     ├─── sh
│     ├─── mount
│     └─── ...
├─── dev
├─── lib
│     ├─── cleanup
│     └─── ...
├─── mnt
├─── memory
├─── proc
├─── sys
└─── init
```

No matter how strange this whole action looks like (we've ended up with the very same directory structure like before, it seems like no improvement at all), the change is significant. Since now, the temporary root filesystem is on tmpfs instead of initramfs, and thus pivot_root operation will be available when needed in the future.

You may be wondering why is the current root filesystem still labeled as temporary. It's because we're still at the very beginning of the construction phase and we'll just build the real root filesystem later, as explained below, by combining Slax compressed data images to AUFS union.

Slax data lookup

Before the init process could start to search for Slax data on available devices, it needs to setup the working environment. The proc and sysfs filesystems are mounted over /proc and /sys respectively. Some important kernel drivers such as aufs, squashfs and loop are loaded using modprobe, and device files are created in /dev directory by mdev command. Path to the mdev binary is also echoed to /proc/sys/kernel/hotplug, to make sure that the Linux Kernel automatically creates new device files in /dev for all the newly discovered disk partitions as soon as they become available later. That is specially needed for partitions on USB devices, since those can take few seconds to fully initialize.

As soon as storage devices are accessible through device files in /dev, blkid command is used to filter out only those which can be mounted (which contain a filesystem known to the running kernel). The devices are examined (mounted read-only over /memory/data/) one after another, until a valid Slax data directory is found. Then, all files with .sb extension (Slax Bundles) are processed - for each Slax Bundle, a directory is created in /memory/bundles/ and the bundle (which in fact is a squashfs compressed image file) is loop-mounted over it. In the diagram below, squashfs content is green.

```
(tmpfs)
/
├─── bin
├─── dev
├─── ...
├─── memory
│    ├─── bundles
│    │    ├─── 01-core.sb (squasfhs mount) < ───┐
│    │    │    ├─── bin                         │
│    │    │    ├─── etc                         │
│    │    │    ├─── sbin                        │
│    │    │    └─── ...                         │
│    │    ├─── 02-xorg.sb ......................│ ...
│    │    │    ├─── etc                         │   :
│    │    │    ├─── usr                         │   :
│    │    │    └─── ...                         │   :
│    │    ├─── 04-kde.sb .......................│ ..:..
│    │    │    ├─── usr                         │   :  :
│    │    │    └─── ...                         │   :  :
│    │    └─── ...                              │   :  :  loop
│    ├─── data (slax device mounted here)       │   :  :  mounts
│    │    └─── slax                             │   :  :
│    │         ├─── boot                        │   :  :
│    │         ├─── changes                     │   :  :
│    │         ├─── rootcopy                    │   :  :
│    │         ├─── 01-core.sb ──── > ──── > ───┘   :  :
│    │         ├─── 02-xorg.sb .....................:  :
│    │         ├─── 04-kde.sb .........................:
│    │         └─── ...
│    ├─── changes (empty yet)
│    └─── union (empty yet)
├─── proc (procfs mounted here)
├─── sys (sysfs mounted here)
└─── init
```

Putting it together with AUFS

Various parts of the final root filesystem are now mounted read-only under separated folders in /memory/bundles. Core Linux system utilities and libraries such as /bin/bash or /lib/ld-linux.so are located in /memory/bundles/01-core.sb/, files related to KDE desktop environment can be found in /memory/bundles/04-kde.sb/, and so on. Combining them into a single root filesystem, which is even writable, is only possible thanks to AUFS - an union-like filesystem developed by Mr. Junjiro Okajima. AUFS is capable of taking several folders (so called branches) and combining them together to a single directory. That is exactly what happens next, the separated parts are combined, together with a directory /memory/changes/, to AUFS union, which gets mounted at /memory/union.

```
(tmpfs)
/
├─── ...
└─── memory
     ├─── bundles
     │    ├─── 01-core.sb ───────── > ──────┐
     │    ├─── 02-xorg.sb ..................│.......
     │    ├─── 04-kde.sb ...................│.......:........
     │    └─── ...                          │       :       :
     ├─── changes ───────── > ───┐          │       :       :
     ├─── ...                    v          v       :       :
     └─── union < ═══════ < ═══════ < ───── < ──────┘  < ───┘
          ├─── bin          AUFS
          ├─── etc          mount
          ├─── mnt
          ├─── root
          ├─── sbin
          ├─── usr
          ├─── ...
          └─── var
```

Changes

The empty directory /memory/changes is writable, thus the entire AUFS mount in /memory/union happens to be writable as well. All new or changed files inside the union are copied-up to this empty directory before the system creates or modifies them. Since the directory for changes resides on tmpfs (that is in RAM), all new and modified files are stored in RAM and thus are lost on reboot.

Yet if Slax is started from a writable media such as USB device or hard disk, it recognizes that and mounts the writable drive over /memory/changes before it is joined with the other branches in union, which effectively means that changed and new files will be stored on the boot device rather than in RAM, and reboot won't erase them. This feature is called Persistent Changes and can be turned on or off by a boot menu setting.

Switching to the real root

At this point, fully writable final root filesystem has been built in /memory/union. The temporary init's life is coming to its end. It uses pivot_root and chroot system calls to make /memory/union the new root, transferring the temporary tmpfs root to /mnt/live/ in the new root. Blue and green stars at the diagram below denote what moves where. Finally, the real /sbin/init from aufs root filesystem is executed. The booting of Slax operating system just begins.

tmpfs root before pivot_root syscall:

```
(tmpfs) ★
/
├─── ...
└─── memory
     └─── union (aufs) ★
          ├─── bin
          ├─── etc
          ├─── mnt
          │    └─── live (empty)
          ├─── root
          ├─── sbin
          │    ├─── ...
          │    └───init
          ├─── usr
          ├─── ...
          └─── var
```

aufs as new root:

```
(aufs) ★
/
├─── bin
├─── etc
├─── mnt
│    └─── live (tmpfs) ★
│         ├─── ...
│         └─── memory
│              └─── union (empty)
├─── root
├─── sbin
│    ├─── ...
│    └─── init <- gets executed
├─── usr
├─── ...
└─── var
```

Adding modules to Slax on the fly

The root filesystem is writable and we could install new software packages while running Slax just the usual way, by unpacking them. Yet there is another possibility to add new files and directories to Slax on the fly without installing any packages. Thanks to the fact Slax is running with AUFS as root, we can take some other squashfs compressed filesystem, loop-mount it over a directory which resides outside of the aufs tree (for example over /mnt/live/memory/bundles/name.sb/ which is on tmpfs), and then issue a remount command which adds the newly mounted directory to aufs as a new branch.

All the files and directories from the new squashfs module will instantly appear as like if they were installed in the system from the beginning, while decompression is done on the fly, only for the files which are actually accessed.

Similarly, we can remove a previously added AUFS branch (mounted squashfs) from the aufs root by another remount command. The files which were part of the branch will instantly disappear from the system, which effectively uninstalls the package.

Slax shutdown

When Slax is shutting down either for reboot or for system power off, it performs all the standard tasks as every other Linux would do, like unmounting all partitions mounted by the user, terminating all processes, and so on. But since the boot device may be still mounted and used for persistent changes at the very end, some more steps need to be done before the real power off is issued, to ensure the boot device is cleanly unmounted.

Instead of turning the system off at the moment when init thinks it should do that, Slax executes a cleanup script which resides in /mnt/live/lib/cleanup. This script calls pivot_root and chroot as we know it in order to change the root filesystem back to the tmpfs again. That is exactly the opposite of what was described in the sections called Switching to the real root.

When root filesystem is switched back to tmpfs, 'telinit u' is executed to inform the current init process to terminate (it would otherwise block the union from unmounting) and re-execute the cleanup script as a new init. The union is no longer blocked since there are no open files and no running processes on it any longer. It is properly unmounted, all loop-mounts disconnected and the boot device is cleanly ejected. At the end, the computer reboots or shuts down, depending on what the user intended to do.