Linux Basics 1: Filesystem Intro

To work effectively on a Linux system, it is necessary to understand a number of the basic aspects of Linux:

- **The Filesystem (directories and files)**
  - **the logical filesystem tree (directory structure)**
  - **pathnames**
  - **mounting and mount points**
  - **filenames**
  - **key commands**
  - standard directories
  - regular files vs. special files
  - physical filesystems
  - additional commands
- Users, Groups, and File Permissions (security mechanisms)
- Processes (running programs and their attributes)

Logical Filesystem

The **logical filesystem** is the set of *directories* (*"folders"*) and *files* that are made available by an operating system.

Alternative terms for the logical filesystem include:
filesystem hierarchy and **directory structure**.

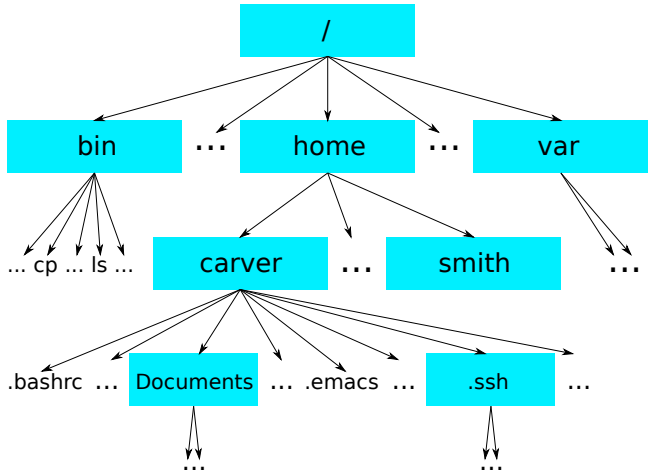
In Linux, the directories and files within the filesystem form a *single* **tree** structure (i.e., a **hierarchical** structure).

In CS, we draw trees with their **roots** at the top.

Thus, the *top-level directory* of the logical filesystem tree is called "**the root directory**."

It is denoted as: / (a single *forward slash*)

Logical Filesystem (contd.)



Pathnames

A particular file or directory within the filesystem structure can be denoted by a **pathname** (**path**) specification.

Note: files and directories are specified in exactly the same way, so one *cannot* tell from a pathname which is being specified.

There are *two types* of pathname specifications:

- **absolute** pathnames
- **relative** pathnames

Pathnames (contd.)

An **absolute pathname/path**:

- completely/absolutely specifies a directory/file
- begins with **/** (denoting *the root directory*)
- includes all subdirectories on path to directory/file
- these path subdirectories are separated by /'s
- e.g., `/home/carver/Documents/test.text`
- gives the *sequence* of directories to traverse in the filesystem tree to reach the directory/file:
  / → `home` → `carver` → `Documents` → `test.text`
- note that containing subdirectory would be:
  `/home/carver/Documents`

Pathnames (contd.)

To understand **relative pathnames**, one must know that every **process** (including each shell command line) has an associated directory known as the **current working directory** (**CWD**) or just **working directory**.

This is the default directory for opening/creating files if only a file/directory *name* is specified (e.g., `test.text`).

The CWD can be explicitly denoted as: **.** ("**dot**").

So a file can be explicitly denoted as being in the CWD as:
`./test.text`

Pathnames (contd.)

A **relative path/pathname**:

- specifies a file/directory *relative to the CWD*
- does *not* start with /
- instead starts with a directory/file in the CWD

Example:

- CWD is: `/home/carver/Documents`
- relative path: `temp/talk.pdf`
- equivalent to absolute path:
  `/home/carver/Documents/temp/talk.pdf`
- ("`/home/carver/Documents`" + "/" + "`temp/talk.pdf`")

Pathnames (contd.)

The **parent directory** (next higher-level directory) of the CWD can be denoted as: **..** ("**dot-dot**" or "**double dot**").

The dot-dot notation can be used in relative path specifications to *ascend* from the CWD, and can be combined as "`../..`"

Example:

- CWD is: `/home/carver/Documents/temp`
- relative path: `../../talks/talk.pdf`
- equivalent to absolute path: `/home/carver/talks/talk.pdf`

Mounting and Mount Points

The single Linux filesystem tree can be constructed from *multiple* **partitions** on *multiple* **storage devices**.

In fact it is standard practice to spread the filesystem among multiple partitions.

One storage partition must be designated as the **root partition**, with its files and directories appearing under `/`.

Every other **storage device** must be **mounted** at some *directory* within the single filesystem tree for its files to be accessible.

The directory at which a device/partition is mounted is called its **mount point**.

Mounting and Mount Points (contd.)

For example, user *home directories* may be stored in a separate disk partition (from `/`), which is normally mounted at `/home`.

E.g., user carver's home directory would be the top-level directory `carver` in the home directories partition.

Once that partition is mounted at `/home`, user carver's home directory would have path `/home/carver`.

Removable devices like CD-ROM drives will have their contents mounted when media are inserted.

E.g., a CD-ROM may be mounted as `/mnt/cdrom`, so a file on the CD-ROM would have a path like: `/mnt/cdrom/fav1.mp3`.

Windows Logical Filesystem

Windows uses a fairly similar scheme for its logical filesystem, but with some key differences:

- each mounted device gets its own separate filesystem tree, identified by a so called "**drive letter**"
- directory separators are "backwards" (backslashes)
- e.g., `C:\WINDOWS\Fonts\Vera.ttf`

Filenames

Linux **filename** scheme:

- much more general than *8.3 scheme* from Windows/DOS
- same naming scheme applies to files, directories, devices, etc.
- cannot tell file's *type* from its filename
  (e.g., `linux.save` could be text file, MP3 file, directory, etc.)
- "**extensions**" do not play a special role with the OS
  (a dot is a character that can appear anywhere in a filename)
- some *applications* do expect certain extensions however
  (e.g., GCC expects C source files to end in "`.c`")

Filenames (contd.)

Linux filename scheme (contd.):

- **extensions** often used very differently than in Windows:
  - files need *not have any extension*
    (e.g., `ls`)
  - extensions often *not 3-letters*
    (e.g., `.html` vs. `.htm`)
  - virtually never use `.exe` for executables
    (files are executable if their **permissions** say so)
  - can have *multiple* "extensions"
    (e.g., `test.text.save.120115`)
- filenames that *begin with a dot* are considered "**hidden files**" and may not be listed by default by some programs
  (e.g., `/home/carver/.bashrc`)

Filenames (contd.)

Linux filename scheme (contd.):

- most hidden files/directories are configuration-related
  (e.g., `/home/carver/.bashrc` and `/home/carver/.ssh`)
- filenames are *case sensitive*:
  - this includes commands, e.g., `ls` ≠ `LS` ≠ `Ls`
  - also applies to extensions, e.g., `foo.mp3` ≠ `foo.MP3`
- POSIX/SUS says to be **portable**, filenames can include only *alphanumerics* plus *dots*, *dashes*, and *underscores*.
- however, most Linux filesystems allow the use of many non-alphanumeric characters, such as spaces, `~`, `?`, `*`, etc.
- the forward slash (`/`) is *never allowed* in filenames, as there could then be ambiguity in interpreting pathnames

Filenames (contd.)

Linux filename scheme: (contd.)

- it is common to use *dashes* or *underscores* to separate "words" in filenames, since if spaces are used, the filenames have to be *quoted* on the command line:

  e.g., `linux-books` or `linux_books`
  vs.
  `"linux books"` or `'linux books'` or `linux\ books`

Filenames (contd.)

Example filenames:

- `test1.text`
- `Test1.text` (completely different file)
- `test1.text.save3`
- `ls` (command/executable)
- `.bashrc` (hidden file)
- `.ssh` (hidden directory)
- `a_long_name`
- `alternative-long-name`
- `a name with spaces . even in extension`

Key Filesystem Commands

cd – change working directory:

- `cd`
- `cd /home/carver/temp`
- `cd temp`
- `cd ..`

pwd – print working directory:

- `pwd`

mkdir – make a new directory:

- `mkdir cs306`
- `mkdir /backup/mp3s`

ls – list directory contents or file(s) info:

- `ls -lA`
- `ls -l *.c`

Key Filesystem Commands (contd.)

mv – move and/or rename file(s)/directory:

- `mv test.c /home/carver/cs306`
- `mv test.c ~/cs306`
- `mv test.c lab1.c`

cp – copy file(s)/directory:

- `cp cs306/*.c ~/cs306-backup`
- `cp -pr cs306 /backup`

rm – remove/delete files or directories:

- `rm *.old`
- `rm -r test-dir`

Linux Basics 2: Filesystem Contd.

To work effectively on a Linux system, it is necessary to understand a number of the basic aspects of Linux:

- **The Filesystem (directories and files)**
  - the logical filesystem tree (directory structure)
  - pathnames
  - file naming scheme
  - key commands
  - **standard directories**
  - **regular files vs. special files**
  - **physical filesystems**
  - **additional commands**
- Users, Groups, and File Permissions (security mechanisms)
- Processes (running programs and their attributes)

Standard Directories

There has been a major effort in recent years to standardize the layout of directories in the Linux filesystem, so most distributions are very similar.

This project is called the Filesystem Hierarchy Standard:
www.pathname.com/fhs

The project specifies a set of directories that must exist, and what their purpose is, so things like system libraries are always in standard locations.

Standard Directories (contd.)

The standard "top-level" directories are:

/bin – (essential) binary/executable files

/boot – bootable kernel files

/dev – device files

/etc – configuration files

/home – user home directories (as /home/username)

/lib – (essential) libraries and kernel modules

/media – mount points for removable media (newer)

/mnt – mount points for removable/temporary media

/opt – optional software

Standard Directories (contd.)

The standard "top-level" directories (contd.):

/proc – pseudo files that display/change OS params

/root – home for the root user

/sbin – system binaries and executables (usually only in root's search path)

/tmp – temporary files (world writable)

/usr – non-host specific, read-only information, secondary hierarchy: e.g., `/usr/bin`, `/usr/lib`, etc.

contains `/usr/local` for purely local/non-system files (cannot be overwritten by standard system software), and yet another hierarchy: e.g., `/usr/local/bin`, etc.

/var – variable files like log files

Regular vs. Special Files

In Linux, many objects that can be opened and read from or written to are treated *largely as if they are disk files*, including:

- **regular files** (normal disk files)
- **directories**
- **symlinks**
- **FIFO**s and **sockets** (IPC mechanisms)
- **devices** (hardware):

All non-regular files are called **special files**.

Regular vs. Special Files (contd.)

Directories are considered *special files* because though they are disk files, they have special format and special API calls:

- they contain a list of *records*
- each record contains an **inode number** and filename
- the first two directory records are always:
  . (dot) a link to the directory itself
  .. (double dot) a link to the parent directory

Physical Filesystems

The term "**filesystem**" can also be used to refer to the *format used to store files on a storage device*, i.e., a **physical filesystem**.

As noted above, the single logical filesystem tree is composed from multiple partitions and/or storage devices, each mounted at some point in the tree.

All of these partitions/storage devices make use of a physical filesystem to organize the data they contain.

There are a wide variety of different physical filesystems in common use, including: FAT, NTFS, ext2/ext3/ext4, JFS, HFS+, ZFS, ISO 9660, UDF, exFAT, JFFS2, LTFS, etc.

Physical Filesystems

A filesystem is designed for a particular type of physical storage device.

The largest set of filesystems are for *"disk systems,"* which allow relatively rapid *random access*.

Disk filesystems include: FAT, NTFS, ext2/ext3/ext4, JFS, HFS+, ZFS, etc.

ISO 9660 and UDF are designed for *optical discs* (CD-ROM, DVD-ROM, BR).

exFAT, JFFS2, etc. are for *flash memory* storage
(note that most consumer devices present a "disk interface," so are used with disk filesystems).

Physical Filesystems (contd.)

Linux has the ability to work with many disk filesystems.

There are several "native" Linux disk filesystems:

- ext2, ext3, ext4, JFS, XFS, ReiserFS, Btrfs
- all of these except ext2 are **journaled**

Linux can also mount Windows filesystems:

- there is full read/write support for FAT32 and NTFS
- this means you can access your Windows files from Linux on a dual-boot machine or from a USB storage device

Linux currently has only read support for journaled HFS+ filesystems used on Macs.

Physical Filesystems (contd.)

Linux physical filesystems store a file's *contents* separate from the file's **metadata** (information about the file).

The metadata is stored in an **inode structure**, which includes the following information:

- inode number
- file owner (a user) and file group
- mode (permissions plus type)
- size in bytes
- last access time (**atime**)
- last modification time (**mtime**)
- last metadata change time (**ctime**)
- link count (explained below)

Physical Filesystems (contd.)

The **link count** attribute of an inode structure warrants further explanation.

We typically think of files as being "contained in" a directory.

What is actually stored inside a Linux directory, however, is a *file record*.

Each file record contains two pieces of information:

- a *name* for the file
- a *link* to the file itself (i.e., to the file's *inode*)

Physical Filesystems (contd.)

Each directory entry (record) is termed a **hard link** to the file.

A file (i.e., the contents) can be *linked multiple times*.

This means that a file can have *multiple names* in the filesystem, in different directories or even in the same directory.

Most regular files will have a link count of one.

Directories have a link count of *at least two*, because each directory contains two special records:

- . (dot) – a link to the directory itself
- .. (double dot) – a link to the parent directory

Physical Filesystems (contd.)

A second type of "file link" is the **symbolic link/soft link/symlink**.

A symlink is a type of *special file* that contains the pathname of another file (as text).

Many commands/functions automatically *"follow symlinks,"* which means they operate on the target file in the symlink rather than on the symlink file itself.

This means that symlinks often function much like hard links in terms of accessing the target file contents.

Symlinks make it easy to see that a file "points to" another file.

Symlinks are much used to provide multiple names for system libraries and the like (e.g., libx.so and libx.so.3 are symlinks to the true library file libx.so.3.2).

Additional Filesystem Commands

ln – create hard/soft links

- `ln existing-name second-name`
- `ln -s existing-name symlink-name`

Linux Basics 3: Users, Groups, & Permissions

To work effectively on a Linux system, it is necessary to understand a number of the basic aspects of Linux:

- The Filesystem (directories and files)
- **Users, Groups, and File Permissions**
  - **users and groups**
  - **usernames vs. UIDs**
  - **group names vs. GIDs**
  - **the root user**
  - **user/group commands**
  - **file access permissions**
  - **directory permissions**
  - **"special" permissions**
  - **permissions commands**
  - **extra: access control lists**
- Processes (running programs and their attributes)

Users and Groups

Linux is a **multiuser OS**, so every user has a **username** and a login **password**.

Actually, each user is ultimately identified by a *non-negative integer* called the **user ID** (**UID**).

In addition to users, there are entities called **groups**, which represent subsets of the users ("group members").

While each group has a *group name*, a group is ultimately identified by a non-negative integer called the **group ID** (**GID**).

Users and Groups (contd.)

Each user must be a member of at least one group, called the user's **primary group**.

A user may be a member of any number of additional groups, called **supplementary groups**.

Groups provide a mechanism for setting access permissions for a subset of users (though they can provide only simple multiuser access control for files).

UIDs, GIDs, and passwords form the basis for much of Linux' *security*, in conjunction with **file access permissions**.

root

The special user **root** is the **administrator** (also referred to as the **superuser**).

root always has UID of 0 (zero).

There is also a **root group** with GID of 0.

Only root has permission to modify most system files (so normal users cannot damage these files).

root has privileges that all other users lack:
access any file, change file ownership, change scheduling/nice priorities, send signals to any process (to terminate it), etc.

Key User/Group Commands

id – print real and effective user and group IDs:

- `id`

groups — print the groups a user is in

- `groups carver`

chown – change file owner and group:

- `chown carver ~/cs306`
- `chown carver:faculty test.c lab1.c`

passwd – change user's password:

- `passwd`

Key User/Group Commands (contd.)

useradd – add a new user:

- `useradd -u 5001 -g faculty carver`

groupadd – add a new group:

- `groupadd -g 1001 gurus`

usermod – modify user information:

- `usermod -aG gurus carver`

File Access Permissions

Every file (regular or special) has an associated *user* (called the file's **owner**) and an associated *group*.

Every file includes **access permissions** for each of *three* **user classes**:

1. **owner**: the file user/owner
2. **group**: members of the file group (excluding owner)
3. **other**: all other users

Only the *most user-specific permissions apply*:
e.g., if a user is the owner, then only the owner permissions apply even if others are less restrictive

File Access Permissions (contd.)

There are *three* **file access permissions** for each of the three user classes: (1) **read**, (2) **write**, (3) **execute**.

For *regular files*, permissions meanings are as expected:

1. **read** means the ability to *view* a file's contents
2. **write** means the ability to *modify* a file's contents (this includes being able to *"truncate"* the file—but note that it does not allow file *deletion*)
3. **execute** means the ability to invoke the file as a "command" and have the kernel run it in a process

Each permission is *Boolean/binary*, so a file's access permissions can be represented using *9 bits*.

Directory Permissions

For *directories*, permissions meanings are slightly different:

1. **read** means the ability to *list out* the files/directories that are contained in the directory
2. **write** means the ability to modify the directory by *creating* new files/directories in it or *deleting* contained files/directories
3. **execute** means the ability to set the directory as the *"current working directory"* or to "pass through" this directory to reach one of its subdirectories

Execute permission is also called **enter** or **search** permission with directories, because of its meaning with directories.

Execute "pass through" permissions requirements for file access are often not fully understood.

Directory Permissions (contd.)

In order to access a file:

- the file must have appropriate permissions to allow the access
- *all directories in the file path* must have execute permission

For example, to modify the file `/home/carver/Documents/test.text`:

- *write permission* will be needed on the *file*
- *execute permission* will be needed on all these *directories*:
  - `/`
  - `/home`
  - `/home/carver`
  - `/home/carver/Documents`

(Appropriate file permissions settings will depend on the file's owner/group and the UID/GID of the accessing program/user.)

"Special" Permissions

There are actually three other "special" permissions bits:

1. **set UID** (**SUID**)
2. **set GID** (**SGID**)
3. **restricted deletion flag** (also known as the **sticky bit**)

The SUID/SGID bits affect the user/group that are associated with an executing program/command:

- SUID permission causes the program's EUID to be set to the file's owner/UID
- SGID permission causes the program's EGID to be set to the file's group/GID

"Special" Permissions (contd.)

The SUID/SGID permissions are used to create programs that run with "**elevated privileges**" (i.e., have access permissions that the user running the program lacks).

For example, the `passwd` command is "**SUID root**."

This allows a normal user to change his password even though he lacks permission to modify the password file.

The **restricted deletion flag** on a *directory* prevents non-owners from deleting files (even if they have write permission for the directory).

This is useful on **world writable** directories like `/tmp`.

Key File Permissions Commands

ls – list file names plus file information:

- `ls -l test.text`

  ⇒ `-rwxrw-r-- 1 carver faculty 297 Mar 16 13:17 test.text`

  (owner carver has r+w+x permissions, faculty group members have r+w, and all other users have only r)

chmod – change file permissions (mode):

- `chmod 760 test.text`
  (760 is *octal*, equals 111 110 000 binary, meaning `rwx rw- ---` for permissions)
- `chmod u=rwx,g=rw,o= test.text`
- `chmod u+x,g-w,o-r test.text`

Extra: Access Control Lists

A key advantage of the basic Linux/UNIX file permissions model is that it is easy to understand, so it is widely used.

By contrast, the permissions model in NTFS on Windows is much more flexible, but also so complex that few users fully understand it, and so few use it effectively.

The key limitation of the basic Linux permissions model is that it is not practical for giving different access permissions to different subsets of users for one file.

Modern Linux filesystems support the use of **access control lists** in conjuntion with file permissions.

With **ACL**s, particular access permissions can be granted to any number of individual users and groups for any file.

Access Control Lists (contd.)

To be backwards compatible with the traditional permissions model, the semantics for Linux ACLs is somewhat complicated.

The two main capabilities ACLs add for file permissions are:

- rwx permissions can be specified directly for any users (without using groups).
- A single **mask permission** can be used by a file's owner to set the *upper bound* for all non-owner permissions.

The commands for getting and setting file ACLs are: `getfacl` and `setfacl`.

ACLs make use of filesystem **extended attributes** (EAs/xattr's), which may have to be enabled when mounting the target filesystem.

Linux Basics 4: Processes

To work effectively on a Linux system, it is necessary to understand a number of the basic aspects of Linux:

- The Filesystem (directories and files)
- Users, Groups, and File Permissions
- **Processes**
  - **what is a process**
  - **process attributes**
  - **process ID (PID)**
  - **environment variables**
  - **exit status**
  - **signals**
  - **process commands**

Processes

The term **process** is used to refer to a program (e.g., command) that is *being executed* (i.e., is running).

Every process has a set of **attributes**, including:

- **process ID** (**PID**) and **parent process ID** (**PPID**)
- **real UID** (**RUID**) and **effective UID** (**EUID**)
- **real GID** (**RGID**) and **effective GID** (**EGID**)
- address space, program counter, run state, etc.
- open files/devices (**file descriptor table**)
- **umask**, **nice** value (priority), resource limits, etc.
- **signal** masks and handlers, alarms/timers, etc.

Processes (contd.)

Every process (except the first) is *created by another process*, which is called its **parent**.

The first process, **init**, is created by the kernel boot process and has PID of 1 (one).

Each process will be in one of several states:

- running/runnable (running or in run queue)
- sleeping/suspended (e.g., waiting for I/O device)
- defunct/zombie (terminated, waiting for collection)
- stopped (job control)

Threads

Each process will have at least one **thread of execution** (one sequence of statements in the program that are being executed).

Linux supports **OS threads**, so each process can in fact contain *multiple threads*.

Most Linux commands/tools show only processes by default; options must be used to show threads.

When viewing threads, one must know these terms:

- **thread ID** (**TID**): unique system-wide ID for an *os thread*
- **thread group**: the set of OS threads in a process
- **thread group ID** (**TGID**): ID for a thread group, is the same as the PID of the containing process

Environment Variables

One important process attribute is the **environment**.

The environment is a *list of variable-value pairs*, which are coded as **C strings** with the format: `"variable=value"`.

These variables are referred to as **environment variables**.

Environment variables are given *uppercase* names by convention (to distinguish them from normal **shell variables**).

The environment is generally *inherited* when a subprocess is created (via `fork()`).

A standard set of environment variables are used to pass basic parameters to all programs.

Environment Variables (contd.)

Key standard environment variables:

PATH – the directories to search for commands

HOME – the current user's home directory

USER,LOGNAME – current username

PWD – current working directory

HOSTNAME – hostname of machine

SHELL – default shell for current user

TERM – terminal type

LANG,LC_* – set of locale related variables.

Exit Status

When a Linux/UNIX process terminates, it returns **exit status** information to the kernel.

This exit status information is supposed to be collected by the process' parent.

One element of the exit status is whether the program had *terminated normally or abnormally*:

- **normal termination**: program called `exit()` (or a related function) or returned from its `main`
- **abnormal termination**: program called `abort()`, received a terminating **signal**, or canceled the thread

Exit Status (contd.)

When a process terminates normally, it must return a *non-negative integer between 0 and 255* to indicate its success/failure status:

- 0 (zero) indicates **success**
- any other value indicates **failure**, with the value possibly representing the reason for failure
- 1 is the default failure return code

Signals

Signals are a mechanism for notifying processes that some *event* has occurred.

They can be considered both as **software interrupts** and as a simple **interprocess communication** (**IPC**) **mechanism**.

Signals can be generated in several ways:

- process/program can cause explicitly (e.g., `kill()`)
- process/program can cause due to error (e.g., **segmentation fault**)
- user can cause by typing *special terminal keys* (e.g., `ctrl-c`)
- user can send using `kill` command

Signals (contd.)

Signals have *default actions* that occur when they are received by a process, which can be one of:

- terminate process
- terminate process and generate **core file**
- ignore signal
- stop process

Processes can also be set to *ignore* or (temporarily) *block* most signals.

However, two signals cannot be ignored/blocked (SIGKILL and SIGSTOP).

Signals (contd.)

Some key signals (for users):

SIGTERM – standard kill signal (but can be ignored/blocked)

SIGKILL – terminate process (cannot be ignored/blocked)

SIGINT – sent by typing **interrupt key** (`ctrl-c`)

SIGQUIT – sent by typing **quit key** (`ctrl-\`)

SIGSEGV – a **segmentation fault** (**segfault**), caused by an illegal memory reference in a program

SIGBUS – a hardware error has occurred

SIGHUP – controlling terminal has disconnected ("hang up")

Key Process Commands

ps – list current process information
- `ps -eF`
- `ps aux`
- `ps -eFT` (show all threads)

pstree – show tree of processes
- `pstree`

top – realtime display of CPU usage by process
- `top`

kill – terminate a process (send it a signal)
- `kill 4521`
- `kill -9 4521` or `kill -SIGKILL 4521`

printenv – print environment variable info
- `printenv PATH`