Introduction to Linux

Part II

https://goo.gl/cypvZ1

Agenda

1. Linux resources you can use
2. Standard input/output and its redirection
3. UNIX pipelines
4. Background processes
5. Durable sessions with `screen`
6. File ownership and permissions
7. Shell scripting basics
8. (Customizing your environment)

What Linux resources can I use?

To do development and run light workloads:

- Your own computer may already run Linux.
- You can install Linux in a virtual machine.
  I recommend Virtualbox.
- If you're running Windows 10, you can install
  Windows Subsystem for Linux

What if it's not enough?

To create persistent services:

- Ask your group's sysadmin for servers/VM resources.
- UniBe Informatikdienste offers virtual machines.
- Cloud resources: SWITCHengines, other cloud services.

To run heavy calculations:

- UBELIX Linux cluster.
- Your group may have in-house infrastructure.
- Again, cloud services.

Prepare for the tutorial

You should be either running Linux, or connected to a Linux system.

Log in to UBELIX, or inform me that you need access to another system.

Execute the following to set up training (if you haven't already):

```
$ wget https://scits.math.unibe.ch/script -O - | /bin/bash
```

Processes and their input/output

When you're connected to a Linux system, what you normally see is the shell prompt, awaiting input:

```
user@host:~ $
```

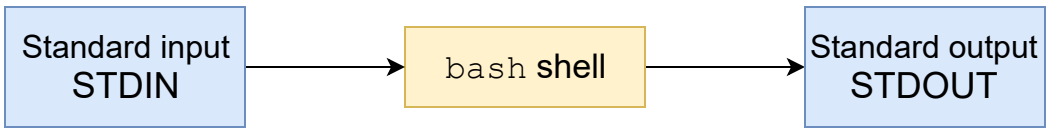
A **process** called shell is responsible for input/output at this moment.

By default, the input is either a keyboard connected to the system, or your keypresses being relayed over the network.

The output, by default, is the screen connected to the system, or text being relayed for display over the network.

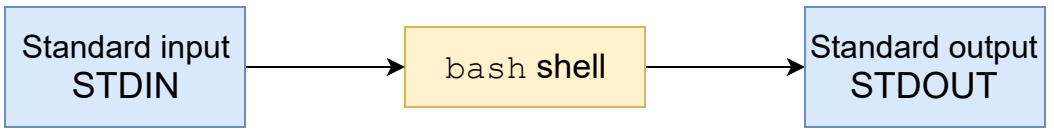
Processes and their input/output

Those are called **standard input** and **standard output**, or **STDIN/STDOUT**.

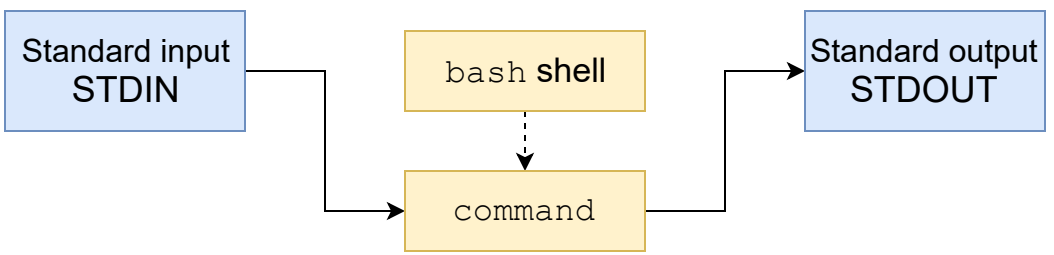


Processes and their input/output

Those are called **standard input** and **standard output**, or **STDIN/STDOUT**.



If the user issues a command that calls another program, the shell creates a **child process** and attaches the input/output to it.



The shell will wait until the program terminates, after which STDIN/STDOUT get reattached and a prompt is displayed.

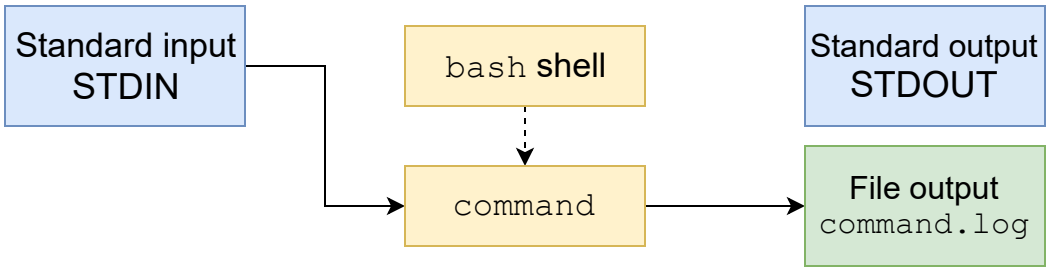
Redirecting output

Sometimes, we want to capture what a command is outputting to the screen.

For example, suppose that we want to save into a file the directory structure returned by `ls -R`:

```
user@host:~ $ ls -R ~/scits-training
scits-training:
animals  moving  numbers  io  scripts
[...]
user@host:~ $
```

We can instruct the shell to **redirect** the standard output:



Redirecting output

To redirect the output, we add **`> FILE`** to the command:

```
user@host:~ $ cd ~/scits-training/io
user@host:~/scits-training/io $ ls -R ~/scits-training > listing
user@host:~/scits-training/io $ cat listing
scits-training:
animals  moving  numbers  io  scripts
[...]
```

This will overwrite the contents of `FILE` (if any) with the output of the command.

The file will be created, if it does not exist yet.

Appending output

Sometimes we don't want to overwrite (sometimes called "clobber") the file with new contents and add them to the end instead.

To do that, use `>> FILE` instead of `> FILE`.

Exercise:

1. Try running the command `date` to see what it outputs.
2. Run `date` 3 times, appending the output to a file `date.log`.
3. Verify with `cat` that the file contains 3 records.

Error output

Try saving the output of a command with errors and you'll see that it still outputs to the screen:

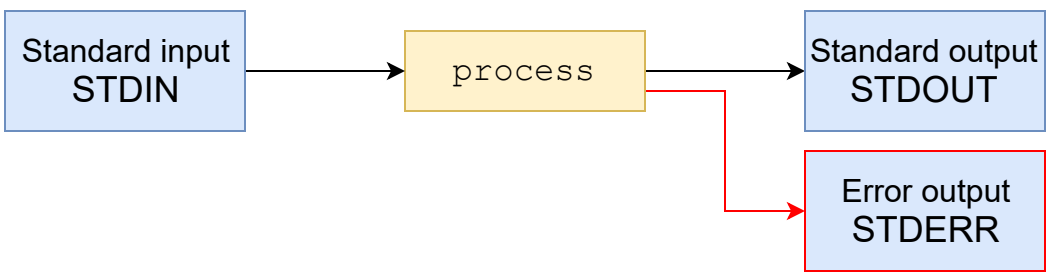
```
user@host:~/scits-training/io $ ls , > listing
ls: cannot access ,: No such file or directory
user@host:~/scits-training/io $
```

Error output

Try saving the output of a command with errors and you'll see that it still outputs to the screen:

```
user@host:~/scits-training/io $ ls , > listing
ls: cannot access ,: No such file or directory
user@host:~/scits-training/io $
```

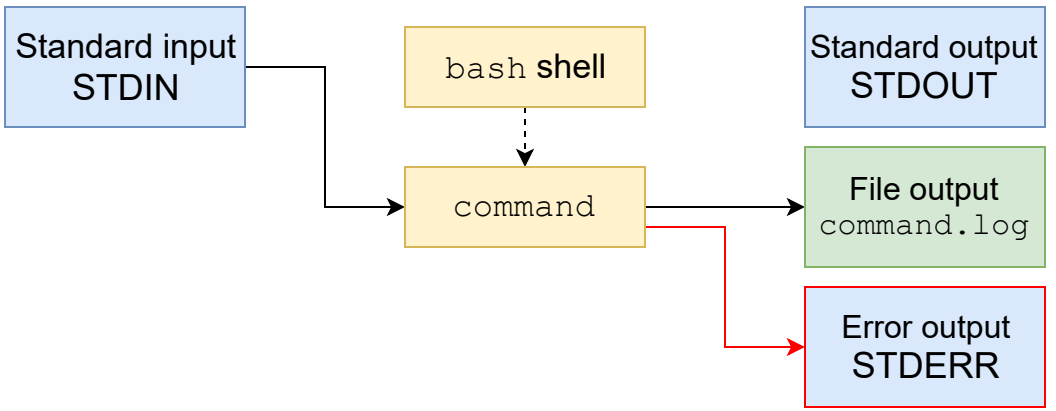
This is intentional: Linux actually has two output streams for its command line, STDOUT for normal data and **STDERR** for errors.



This simplifies debugging: errors are separate from data.

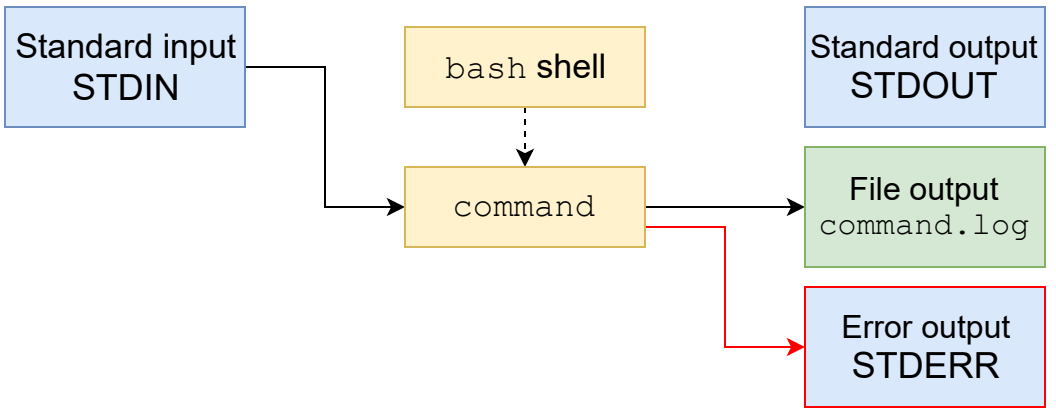
Error output

STDERR is not redirected when using > or >>:



Error output

STDERR is not redirected when using > or >>:



It's possible to redirect it as well with **2>** or **2>>**:

```
user@host:~/scits-training/io $ ls .. , > listing 2> errors
user@host:~/scits-training/io $ cat errors
ls: cannot access ,: No such file or directory
user@host:~/scits-training/io $ cat listing
..:
animals  io  moving  numbers  scripts
```

Discarding output

Sometimes we don't need output at all.

In this case, we can redirect it to a special file, **`/dev/null`**

It's a **device** that will accept any input, discarding it immediately.

Discarding output

Sometimes we don't need output at all.

In this case, we can redirect it to a special file, **`/dev/null`**

It's a **device** that will accept any input, discarding it immediately.

For example, one might want to silence errors:

```
user@host:~/scits-training/io $ ls .. , 2> /dev/null
..:
animals  io  moving  numbers  scripts
```

Interactive input

Most commands we've seen don't require any interactive input.

`tr` (for **translate**) is a command that transforms its input: it substitutes some characters with others.

For example, `tr 'a-z' 'A-Z'` would translate all lowercase letters into uppercase.

Interactive input

Most commands we've seen don't require any interactive input.

`tr` (for **translate**) is a command that transforms its input: it substitutes some characters with others.

For example, `tr 'a-z' 'A-Z'` would translate all lowercase letters into uppercase.

Let's try this:

```
user@host:~/scits-training/io $ tr 'a-z' 'A-Z'
Let's input some text
LET'S INPUT SOME TEXT
```

Interactive input

```
user@host:~/scits-training/io $ tr 'a-z' 'A-Z'
Let's input some text
LET'S INPUT SOME TEXT
But enter doesn't stop it!
BUT ENTER DOESN'T STOP IT!
```

A problem: text can contain many lines, and the program won't know when to stop.

Interactive input

```
user@host:~/scits-training/io $ tr 'a-z' 'A-Z'
Let's input some text
LET'S INPUT SOME TEXT
But enter doesn't stop it!
BUT ENTER DOESN'T STOP IT!
```

A problem: text can contain many lines, and the program won't know when to stop.

We can terminate the program with Ctrl+C, but it actually expects an **end of input**.

We can signal end of input with Ctrl+D on an empty line (or pressing it twice).

Interactive input

```
user@host:~/scits-training/io $ tr 'a-z' 'A-Z'
Let's input some text
LET'S INPUT SOME TEXT
But enter doesn't stop it!
BUT ENTER DOESN'T STOP IT!
```

A problem: text can contain many lines, and the program won't know when to stop.

We can terminate the program with Ctrl+C, but it actually expects an **end of input**.

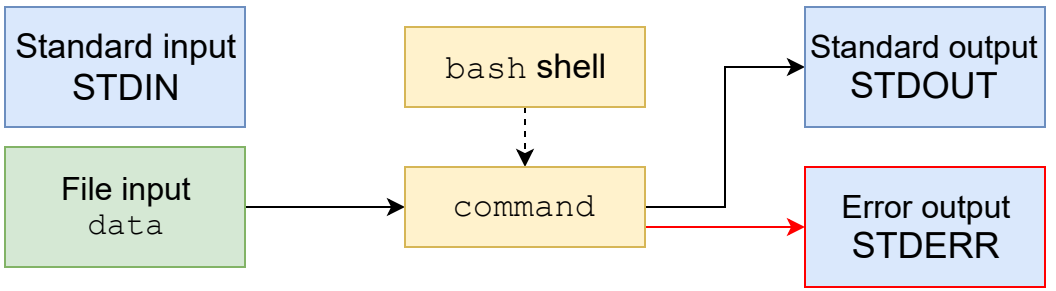
We can signal end of input with Ctrl+D on an empty line (or pressing it twice).

Exercise:

What happens if we press Ctrl+D while back at the shell prompt?

Redirecting input

What if we want to use a file as an input in a command that doesn't accept files as arguments, we need instruct the shell to use the file as the program's standard input:



This is done with adding **< FILE** to the command.

```
user@host:~/scits-training/io $ tr 'a-z' 'A-Z' < errors
LS: CANNOT ACCESS ,: NO SUCH FILE OR DIRECTORY
```

Redirects

Exercise:

1. Combine input and output redirection to save the output of last `tr` command into `errors.uppercase`
2. Use `cat` to verify the saved output.

Pipelines

We have shown how to save outputs to a file, and further process files as inputs.

Sometimes, we don't need to save this intermediate representation. In that case, we can directly connect the output of one program to the input of another with **pipes**.

To do so, separate two commands with **|**:

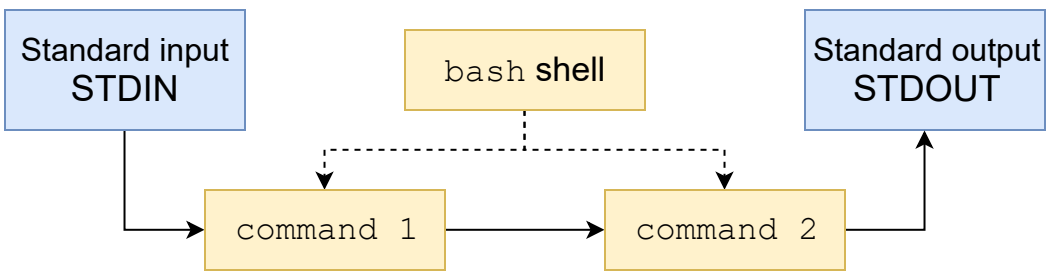
```
user@host:~/scits-training/io $ ls . | tr 'a-z' 'A-Z'
DATE.LOG
ERRORS
LISTING
```

Pipelines

```
user@host:~/scits-training/io $ ls . | tr 'a-z' 'A-Z'
DATE.LOG
ERRORS
LISTING
```

Given this command, shell starts two processes in parallel and ties their respective output and input together.

Standard input/output is connected at the ends of the chain:

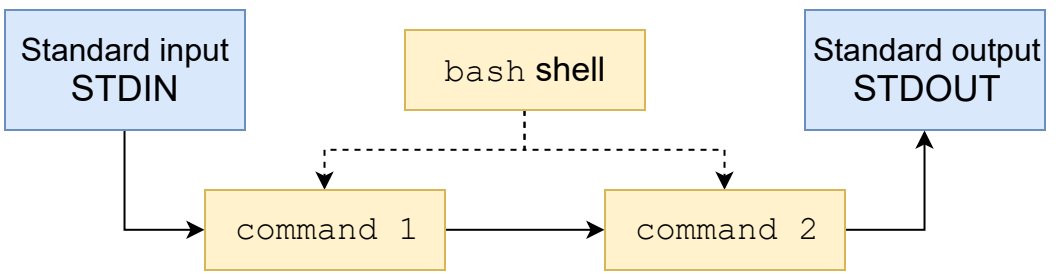


Pipelines

```
user@host:~/scits-training/io $ ls . | tr 'a-z' 'A-Z'
DATE.LOG
ERRORS
LISTING
```

Given this command, shell starts two processes in parallel and ties their respective output and input together.

Standard input/output is connected at the ends of the chain:



Such **pipelines** can be longer than two commands, and can be combined with file redirects.

Pipelines and errors

You will notice that all errors are still output normally:

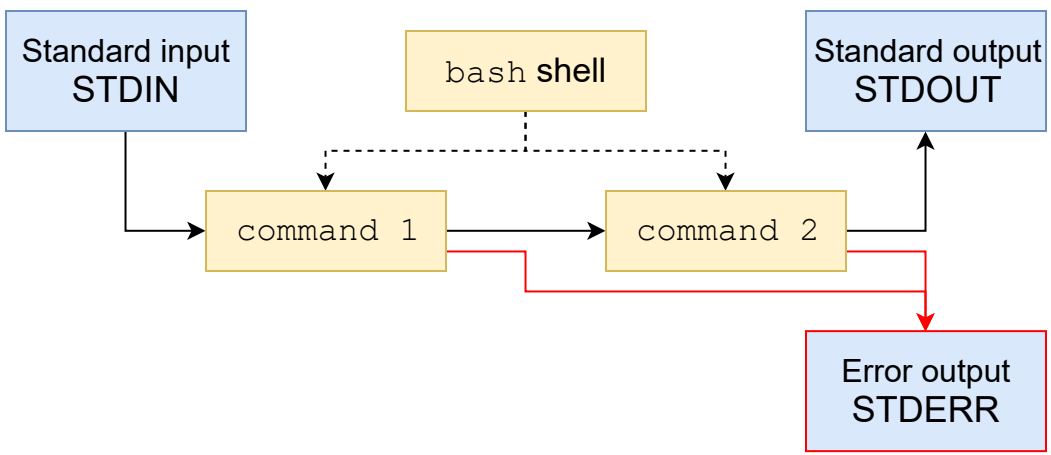
```
user@host:~/scits-training/io $ ls . , | tr 'a-z' 'A-Z'
ls: cannot access ,: No such file or directory
.:
DATE.LOG
ERRORS
LISTING
```

Pipelines and errors

You will notice that all errors are still output normally:

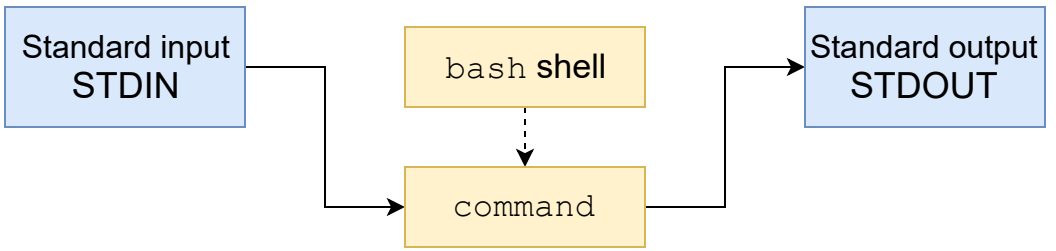
```
user@host:~/scits-training/io $ ls . , | tr 'a-z' 'A-Z'
ls: cannot access ,: No such file or directory
.:
DATE.LOG
ERRORS
LISTING
```

As before, errors are not normally redirected, and collected from all processes in the pipe:



Background jobs

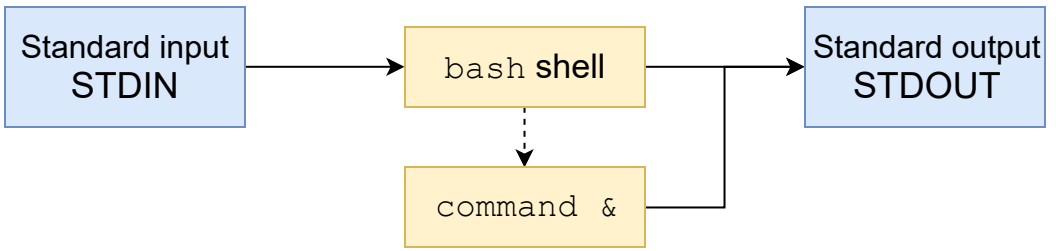
Recall that, when running a command, the shell waits until it is terminated: all input goes to the program (or nowhere).



Sometimes, we don't need to wait until the program terminates – we actually want it running in background.

Background jobs

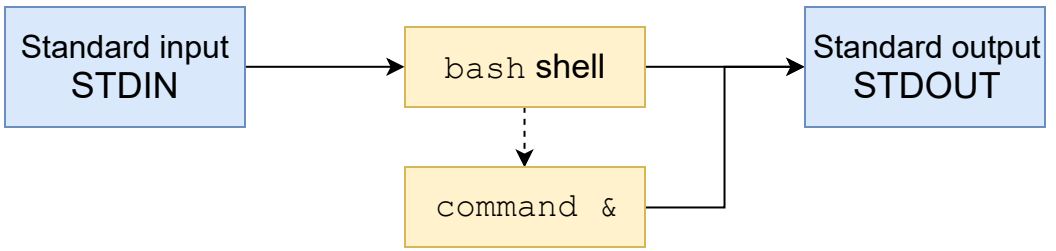
If you specify **&** at the end of the command, the shell will start it, but keep control of STDIN:



Instead of a **foreground** process, it becomes a **background** job.

Background jobs

If you specify **&** at the end of the command, the shell will start it, but keep control of STDIN:



Instead of a **foreground** process, it becomes a **background** job.

You are immediately returned to the shell and can run other commands while the job executes.

Note that both the shell and the background job are connected to STDOUT. Redirect output to prevent mix-ups.

Background jobs

Compare:

```
user@host:~/scits-training/io $ sleep 3
user@host:~/scits-training/io $ sleep 3 &
[1] 12231
user@host:~/scits-training/io $
```

Here, `[1]` is the **job number**, and `12231` is the **process ID**, or PID.

After 3 seconds and when another command finishes (you can just press Enter for an empty command), you'll be informed that the job terminated:

```
user@host:~/scits-training/io $
[1]+  Done                    sleep 3
user@host:~/scits-training/io $
```

Listing jobs

You can list running background jobs with **`jobs`**:

```
user@host:~/scits-training/io $ sleep 100 &
[1] 12232
user@host:~/scits-training/io $ sleep 0 &
[2] 12233
user@host:~/scits-training/io $ jobs
[1]-  Running                 sleep 100 &
[2]+  Done                    sleep 0
```

Terminating jobs

You can forcibly **terminate** a job with the **`kill`** command, which accepts either PID or job ID (with `%`):

```
user@host:~/scits-training/io $ sleep 100 &
[1] 12234
user@host:~/scits-training/io $ kill 12234
user@host:~/scits-training/io $ jobs
[1]+  Terminated              sleep 100

user@host:~/scits-training/io $ sleep 100 &
[1] 12235
user@host:~/scits-training/io $ kill %1
```

You can search for more process IDs to terminate with **`ps ax`**, in case something is misbehaving.

Stopped jobs

Background jobs have nothing connected to their standard input.

If a background job cannot continue without user input, it will **stop**, which the shell will signal to you:

```
user@host:~/scits-training/io $ tr 'a-z' 'A-Z' &
[1] 12236
user@host:~/scits-training/io $
[1]+  Stopped                 tr /a-z/ /A-Z/
user@host:~/scits-training/io $
```

Stopped jobs

Background jobs have nothing connected to their standard input.

If a background job cannot continue without user input, it will **stop**, which the shell will signal to you:

```
user@host:~/scits-training/io $ tr 'a-z' 'A-Z' &
[1] 12236
user@host:~/scits-training/io $
[1]+  Stopped                 tr /a-z/ /A-Z/
user@host:~/scits-training/io $
```

You can bring a job to **foreground** to pass STDIN from the shell to the running job with `fg` (or `fg %N` for a specific job number):

```
user@host:~/scits-training/io $ fg
tr /a-z/ /A-Z/
You are now talking to the job
YOU ARE NOW TALKING TO THE JOB
```

Stopping and resuming programs

You can stop most currently-running programs with `Ctrl+Z`:

```
user@host:~/scits-training/io $ sleep 100
^Z
[1]+  Stopped                 sleep 100
user@host:~/scits-training/io $
```

Stopping and resuming programs

You can stop most currently-running programs with `Ctrl+Z`:

```
user@host:~/scits-training/io $ sleep 100
^Z
[1]+  Stopped                 sleep 100
user@host:~/scits-training/io $
```

From there, you can use `fg` to resume normal execution of the program, or use **`bg`** to let it continue to run in the background.

```
user@host:~/scits-training/io $ bg
[1]+ sleep 100 &
user@host:~/scits-training/io $ jobs
[1]+  Running                 sleep 100 &
```

Background jobs are fragile

What will happen if you start a background job, and then close the terminal?

Background jobs are fragile

What will happen if you start a background job, and then close the terminal?

Closing the terminal (or disconnecting the SSH session) kills the shell you were talking to. Since the job was a **child process** of that shell, it will also be killed.

Background jobs are fragile

What will happen if you start a background job, and then close the terminal?

Closing the terminal (or disconnecting the SSH session) kills the shell you were talking to. Since the job was a **child process** of that shell, it will also be killed.

A minor inconvenience if you're working on your own machine (you can just leave the terminal open), but a much bigger problem with remote connections.

If the connection is broken, the shell is also terminated along with all processes launched from it.

How to protect against it?

screen

To protect your session, you can use **`screen`**.

`screen` starts a new shell that exists independently of your current one.

Even if the current shell dies (e.g. because you disconnected), the shell running in `screen` will continue together with all its child processes.

screen

To protect your session, you can use **screen**.

`screen` starts a new shell that exists independently of your current one.

Even if the current shell dies (e.g. because you disconnected), the shell running in `screen` will continue together with all its child processes.

Starting a new `screen` session is simple:

```
user@host:~/scits-training/io $ screen
[terminal screen is cleared]
user@host:~/scits-training/io $ echo "Hello, I'm in a screen"
Hello, I'm in a screen!
user@host:~/scits-training/io $
```

Reattaching to `screen`

Now suppose your connection was terminated.

Close the terminal where it is running to simulate that, then log in again.

Reattaching to `screen`

Now suppose your connection was terminated.

Close the terminal where it is running to simulate that, then log in again.

You can use **`screen -ls`** to list active sessions:

```
user@host:~ $ screen -ls
There is a screen on:
        13383.pts-2.host        (11/09/17 03:02:23)     (Detached)
1 Socket in /var/run/screen/S-user.
```

Reattaching to `screen`

Now suppose your connection was terminated.

Close the terminal where it is running to simulate that, then log in again.

You can use **`screen -ls`** to list active sessions:

```
user@host:~ $ screen -ls
There is a screen on:
        13383.pts-2.host        (11/09/17 03:02:23)     (Detached)
1 Socket in /var/run/screen/S-user.
```

You can attach to a screen session (possibly detaching it first, if it's being used somewhere) with **`-dR`** (for **detach, reattach**)

```
user@host:~ $ screen -dR
[terminal screen is cleared]
user@host:~/scits-training/io $ echo "Hello, I'm in a screen"
Hello, I'm in a screen!
user@host:~/scits-training/io $
```

Controlling screen

`screen` can be used for other things, such as having multiple parallel shell sessions open.

Controlling screen consists of pressing **`Ctrl+A`**, then a screen-specific command.

For example,

- **`c`** will **create** a new shell within screen
- **`n`** will switch to the **next** shell
- **`d`** will **detach** from `screen`, returning you to the original shell

Finally, you can use **`?`** to access built-in help, or use `man screen` for a more detailed manual.

Controlling screen

`screen` can be used for other things, such as having multiple parallel shell sessions open.

Controlling screen consists of pressing **`Ctrl+A`**, then a screen-specific command.

For example,

- **`c`** will **create** a new shell within screen
- **`n`** will switch to the **next** shell
- **`d`** will **detach** from `screen`, returning you to the original shell

Finally, you can use **`?`** to access built-in help, or use `man screen` for a more detailed manual.

Another popular alternative to `screen` is `tmux`. It will not be covered by this tutorial, but is worth looking into.

Users and groups

Before we discuss permissions, we need to understand users and groups in Linux.

A **user** is a unit of access control; it has a set of credentials to access the system and **owns** some files on it.

A **group** is a collection of users to facilitate shared access to resources. A user can belong to many groups but one group is considered primary.

You can use `id` to check your user and groups:

```
akashev@submit01:~ $ id
uid=7265(akashev) gid=1109(math) groups=1109(math),902(l_gaussian)
```

Here, `akashev` is my user, `math` is my primary group and `l_gaussian` is another group I belong to.

Permissions: `rwx`

Each file and directory in UNIX filesystems has 3 permissions (for a particular user).

Regular files:

- `r`, or **Read**, means that you can read the contents of a file.
- `w`, or **Write**, means that you can modify the file.
- `x`, or **eXecute**, means that the file may be launched as a program.

Directories:

- `r` means that you can read the list of files within the directory.
- `w` means that you can add or delete files from the directory.
- `x` means you can **traverse** the folder: enter it with `cd` and read the contents of its files.

Inspecting permissions

Try running `ls -la` to see permissions on files and folders:

```
$ ls -la
total 20
drwxrwxr-x 2 user group 4096 Sep 11 01:26 .
drwxrwxr-x 6 user group 4096 Sep 10 23:06 ..
-rw-rw-r-- 1 user group   90 Sep 10 23:08 date.log
-rw-rw-r-- 1 user group   47 Sep 11 00:50 errors
-rw-rw-r-- 1 user group   30 Sep 11 01:09 listing
```

We're interested in the first column: the cryptic `drwxrwxr-x` and `-rw-rwr--`, which are called **mode**.

Inspecting permissions

Try running `ls -la` to see permissions on files and folders:

```
$ ls -la
total 20
drwxrwxr-x 2 user group 4096 Sep 11 01:26 .
drwxrwxr-x 6 user group 4096 Sep 10 23:06 ..
-rw-rw-r-- 1 user group   90 Sep 10 23:08 date.log
-rw-rw-r-- 1 user group   47 Sep 11 00:50 errors
-rw-rw-r-- 1 user group   30 Sep 11 01:09 listing
```

We're interested in the first column: the cryptic `drwxrwxr-x` and `-rw-rwr--`, which are called **mode**.

- The first character denotes the **file type**.
  - `-` means "regular file".
  - `d` means "directory".
- The rest is divided in groups of three:
  - Access for the owner
  - Access for the group
  - Access for everyone else

File ownership

```
drwxrwxr-x 2 user group 4096 Sep 11 01:26 .
-rw-rw-r-- 1 user group   90 Sep 10 23:08 date.log
```

Each file in a UNIX filesystem has an **owner** and a **group** attached.

In the example above, `user` is the owner and `group` is the designated group.

Note that the user **doesn't have to be in the assigned group**.

Effective permissions

```
-rwxr-x--- 1 user group   90 Sep 10 23:08 script
```

To determine which permissions apply, the following is checked:

- If the user is the owner, the first set applies (`rwx`, full permissions)
- If the user is in the designated group, the second set applies (`r-x`, so cannot write)
- For all other users, the third set applies (`---`, so cannot do anything)

Effective permissions

```
-rwxr-x--- 1 user group   90 Sep 10 23:08 script
```

To determine which permissions apply, the following is checked:

- If the user is the owner, the first set applies (`rwx`, full permissions)
- If the user is in the designated group, the second set applies (`r-x`, so cannot write)
- For all other users, the third set applies (`---`, so cannot do anything)

A special user, **superuser** (normally called **root**), can completely disregard permissions and do anything to any file on the system.

Permissions: first match applies

Note that the system does not apply "best" permissions – only the first set that matches.

Let's reverse the situation:

```
----r-xrwx 1 user group   90 Sep 10 23:08 script
```

For this file, the owner cannot do anything to the file, anyone in `group` cannot modify it, but everyone else has full permissions.

Note: the owner can always change a file's permissions.

Modifying permissions

To modify a file's permissions, use **`chmod CHANGES FILE`**

Mnemonic:

`chmod` stands for **change mode**.

Possible changes:

- **`+r`**, **`+w`**, **`+x`** **add** permissions. Can combine: `+rw`
- **`-r`** **removes** permissions.
- **`=r`** sets pemissions to **exactly** `r--`.
- Prefix **`u`** changes permissions for the **user**, e.g. `u+r`.
- Prefix **`g`** changes permissions for the **group**, e.g. `g+rw`.
- Prefix **`o`** changes permissions for **others**, e.g. `o-w`.
- Prefix **`a`** or no prefix changes permissions for **all** three sets.
- An **octal number** (e.g. 750) sets permissions to a specific configuration (in this case, `rwxr-x---`).

Modifying permissions

Several changes can be applied at once, separated by commas:

```
user@host:~/scits-training/io $ ls -la date.log
-rwxrw-r-- 1 user group   90 Sep 10 23:08 date.log

user@host:~/scits-training/io $ chmod u+x,g=rx,o-r date.log

user@host:~/scits-training/io $ ls -la date.log
-rwxr-x--- 1 user group   90 Sep 10 23:08 date.log
```

Exercise:

Modify permissions on the file to be `r-xr--rw-`

Changing ownership

Similarly to `chmod`, the **`chown`** command allows changing a file's owner and group.

- `chown USER FILE` changes the owner
- `chown :GROUP FILE` changes the group
- `chown USER:GROUP FILE` changes both

Note: once the owner is changed, the old owner no longer can modify access to the file.

For this reason, only administrators can change the file owner, or assign a group the owner is not part of.

Exercise:

Use `groups` to list groups you belong to.

Change a file's group to one of them, and then back to the original one.

Shell scripting

Shell is not just an interface to launch other programs; it comes with its own scripting language to automate complex tasks.

You can have variables, loops, conditionals – a full-featured programming language.

We will only show the very basics.

Exercise:

Navigate to `~/scits-training/scripts` and open `boom.sh` in your favourite editor (nano, vim)

Shell scripting

```
#!/bin/bash

# I hope you get the reference
echo "Someone set up us the bomb."
for i in {5..1}
do
  echo "$i.."
  sleep 1
done
explosion="Boom!"
echo $explosion
```

The first line of the script is special:

```
#!/bin/bash
```

It's called a **"shebang"** (for **shell** and "!" **bang**).

It tells the shell what to execute the rest of the script with. Since we're writing a `bash` shell script, we put there the path to `/bin/bash` itself.

Shell scripting

```
#!/bin/bash

# I hope you get the reference
echo "Someone set up us the bomb."
for i in {5..1}
do
  echo "$i.."
  sleep 1
done
explosion="Boom!"
echo $explosion
```

Other lines starting with **#** are **comments**

```
# I hope you get the reference
```

They are ignored by `bash` and are used to leave notes to yourself or others.

Shell scripting

```
#!/bin/bash

# I hope you get the reference
echo "Someone set up us the bomb."
for i in {5..1}
do
  echo "$i.."
  sleep 1
done
explosion="Boom!"
echo $explosion
```

echo command outputs its arguments to STDIN.

```
echo "Someone set up us the bomb."
```

Quotes are used to make text with spaces in it a single argument; here, they are optional.

Shell scripting

```
#!/bin/bash

# I hope you get the reference
echo "Someone set up us the bomb."
for i in {5..1}
do
  echo "$i.."
  sleep 1
done
explosion="Boom!"
echo $explosion
```

`for` designates a **loop**: a **variable `i`** will change from 5 to 1.

```
for i in {5..1}
do
  # something
done
```

The code in `# something` will repeat with `i` as 5, 4, 3, 2 and 1.

`do` and `done` delimit the bounds of the loop.

Shell scripting

```
#!/bin/bash

# I hope you get the reference
echo "Someone set up us the bomb."
for i in {5..1}
do
  echo "$i.."
  sleep 1
done
explosion="Boom!"
echo $explosion
```

One can use the variable in expressions **prefixed by `$`**, i.e. **`$i`**:

```
echo "$i.."
```

If there is ambiguity as to where a variable name ends, use braces: **`${i}`**, e.g. `"Sample ${i}A"` for "Sample 1A", etc.

Shell scripting

```
#!/bin/bash

# I hope you get the reference
echo "Someone set up us the bomb."
for i in {5..1}
do
  echo "$i.."
  sleep 1
done
explosion="Boom!"
echo $explosion
```

Variables can also be simply **assigned to**:

```
explosion="Boom!"
echo $explosion
```

The lack of spaces around = is **significant**.
Otherwise Bash will try to execute `explosion` as a command.

Running a script

OK, suppose we wrote the above script. How to execute it?

1. We need to make sure that it's allowed to execute:

```
user@host:~/scits-training/scripts $ chmod +x boom.sh
```

2. For security reasons, the current directory is not automatically considered when starting other programs. We need to explicitly refer to it:

```
user@host:~/scits-training/scripts $ ./boom.sh
```

Exercise:

1. Execute the script, saving its output to a file.
2. Modify the script to count down from 10.

Scripting, take two

The next script you will type out yourselves.

Open `beer.sh` in your favourite editor.

We'll write a simple script to determine if a user is old enough to drink beer.

Scripting, take two

```
#!/bin/bash
```

Any bash script should start with an appropriate shebang.

We want to ask the user for his/her age; we can use the **`read`** command.

```
# -n prevents a line break, and note the extra space
echo -n "What's your age? "
read age
```

This will display a promt for the user and wait for input. The result is then stored in the variable `$age`.

For simplicity, we will not check that the input is indeed a valid number.

Scripting, take two

```
#!/bin/bash
echo -n "What's your age? "
read age
```

We need to make a decision based on age; we need an if-then-else construct.

```
if [ $age -lt 16 ]
then
  echo "You're too young to drink!"
else
  echo "You're old enough, have a beer!"
fi
```

`fi` here is `if` reversed, to close the **`if`** statement.

Conditionals in bash are a bit clunky, but **`-lt`** here stands for **less than**. Again, the whitespace here is **significant**.

Scripting, take two

```
#!/bin/bash
echo -n "What's your age? "
read age
if [ $age -lt 16 ]
then
  echo "You're too young to drink!"
else
  echo "You're old enough, have a beer!"
fi
```

Exercise:

1. Save this script to `beer.sh`.
2. Change the file's mode to allow execution.
3. Test the script with different values.

Scripting improvements

Let's add a little personal touch.

`whoami` is a command that returns the username. Let's edit `beer.sh` to use it:

```
then
  echo "$(whoami), you're too young to drink!"
else
  echo "$(whoami), you're old enough, have a beer!"
fi
```

`$(something)` allows you to execute a command and substitute the result within another command.

Exercise:

Test the new additions.

Scripting improvements

Let's read the age from the command line arguments.

`bash` automatically populates **`$0`** with the **name of the executable**, and **`$1`**, **`$2`** and so on with **arguments**.

Let's use `$1` as age if it's defined:

```
if [ $1 ]
then
  age=$1
else
  echo -n "What's your age? "
  read age
fi
```

Exercise:

Test that `./beer.sh` now automatically gets the age from its first argument, and still asks if no argument is provided.

Return values

Whenever a program terminates, it returns a single integer to the shell that called it; it's called the **return value**.

By convention:

- **0** means "no error".
- any **non-zero value** means "some kind of error".

Let's return appropriate values:

```
then
  echo "$(whoami), you're too young to drink!"
  exit 1
else
  echo "$(whoami), you're old enough, have a beer!"
  exit 0
fi
```

Chaining commands

You can chain commands in shell with `;` or `&&`.

`;` will execute commands one by one, regardless of errors.

```
$ command1; command2
```

`&&` will only execute the next command only if the previous one returned 0, i.e. finished without errors.

```
$ command1 && command2
```

Exercise:

1. Apply the return value changes to `beer.sh`
2. Test it with `./beer.sh && echo 'Cheers!'`

Customizing your environment

If you interact with a system often, you may want to add various shortcuts and customizations to your shell.

Customizing your environment

If you interact with a system often, you may want to add various shortcuts and customizations to your shell.

Bash uses two files, `~/.bash_profile` and `~/.bashrc`, to allow you to apply such customizations on every login.

Essentially, they are bash scripts that are run before the shell shows a prompt.

Customizing your environment

If you interact with a system often, you may want to add various shortcuts and customizations to your shell.

Bash uses two files, `~/.bash_profile` and `~/.bashrc`, to allow you to apply such customizations on every login.

Essentially, they are bash scripts that are run before the shell shows a prompt.

- `.bashrc` is applied when you open a new shell when already logged in.
- `.bash_profile` is applied when you log in (e.g. though SSH).

One can apply code in `.bashrc` from `.bash_profile` to cover both cases.

Environment variables

Your profile files can set various **environment variables**: snippets of data inherited by programs running from shell.

You can see your current environment variables with:

```
$ env | less
```

Some programs rely on environmental variables to change their behavior. Example:

```
# Will replace the default editor with vim in some commands
export EDITOR=vim
```

This works exactly like setting a variable in a script, except for the extra command **export**, which propagates this variable to child processes.

By convention, environment variables are UPPERCASE.

Environment variables

Your home directory is available in `$HOME` – in some contexts, using ~ is not possible.

Another important variable is `$PATH`.

It's a colon-separated list of directories which are searched when you try to run a program by name.

Notably, the current directory is not in `$PATH`.

If you have created some own scripts/programs and want them to be available by name from anywhere, you can put them in a folder (e.g. `~/bin`) and add it to `$PATH`:

```
export PATH="$PATH:$HOME/bin"
```

Custom shell prompt

The variable `$PS1` contains the format template for your shell prompt.

Throughout this training, we assumed the following:

```
export PS1="\u@\h:\w \\$ "
```

But you can customize it! Want the space for the command on a separate line?

```
export PS1="\u@\h:\w\\n\\$ " # \\n represents a line break
```

```
user@host:~
$
```

Want to add current time? Want to add some color?
There's a guide for that.

Aliases

If you use a certain command often, you can define a short name for it.

For example, if you want a shorter name for `ls -lh` because you always want to see human-readable sizes, you can make an alias:

```
$ alias lh="ls -lh"
$ lh
total 26M
-rw-r--r-- 1 user group 25M Sep 11 07:22 big_file
-rw-r--r-- 1 user group 735 Sep 11 07:22 description
-rw-r--r-- 1 user group   0 Sep 11 07:22 empty_file
-rw-r--r-- 1 user group 551 Sep 11 07:22 naming
drwxr-xr-x 0 user group 512 Sep 11 07:22 subfolder
```

Making customizations permanent

To make above tweaks permanent, they need to be added either to `.bash_profile` or `.bashrc`.

Then they will apply on each opened shell.

- `.bash_profile` is **sourced** at most once. Put things there that shouldn't be called multiple times.
- `.bashrc` is sourced almost every time `bash` is called, except for initial SSH shell. To be safe, you can "include" `.bashrc` into `.bash_profile` like this:

```
# In .bash_profile
# -f tests that file exists
# source executes commands in the current shell
if [ -f ~/.bashrc ]; then
  source ~/.bashrc
fi
```