Working with Linux



Summer student lectures 2011

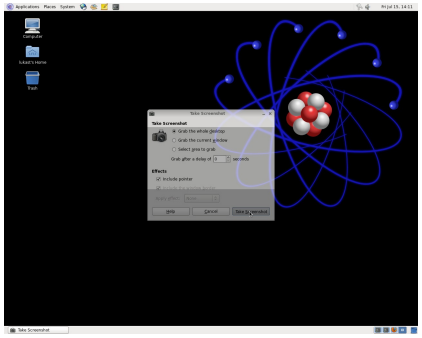
Wolfgang Friebel

HELMHOLTZ ASSOCIATION

DESY

The Window manager

- Before logging in a window manager can be chosen
  - If you are already used to e.g. Gnome or KDE select that one
  - Otherwise the recommended choice is Gnome
  - Other Windowmanagers available as well
    - KDE, a widely used alternative to Gnome
    - Icewm, a lightweight window manager
    - Windowmaker, modeled after the nextstep user interface



- After login the window manager fulfills several tasks
  - Start applications by selecting from menus or clicking on icons
  - Have applications running in windows
  - Move windows around and resize windows
  - Switch between virtual desktops thus enlarging your useable screen area
  - Copy and paste contents within or between applications
- The look and feel of the window manager is highly configurable

Editors

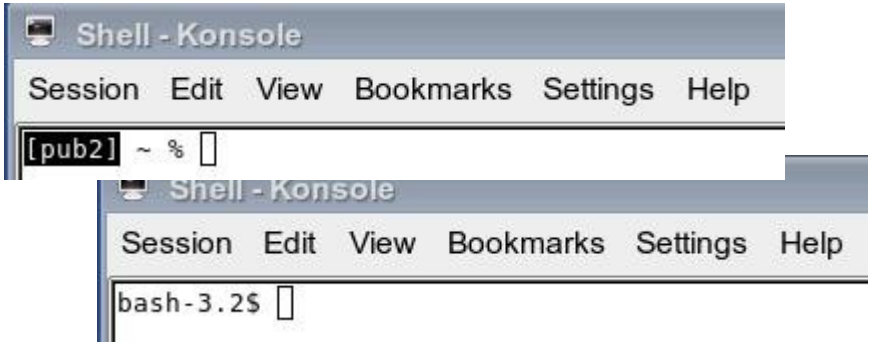
- A variety of editors to chose from for different purposes
  - for programming well suited editors have syntax highlighting like:
  - vi, vim, gvim very powerful editor[s], steep learning curve, very efficient afterwards
  - emacs another powerful editor, very rich command set and plugin system
  - nedit easy to use editor, especially for beginners, not very capable
  - gedit easy to use editor, comes with gnome
  - kedit easy to use editor, comes with KDE, no syntax highlighting(?)

The shell

- Many but not all programs can be started from the window manager
- Many more things are possible from within a shell
  - A window running a 'terminal program' and within that terminal program another program, a 'shell' is started that lets you interact with the computer
  - The (login) shell is started from the WM and inherits some variables (e.g. LANG)
  - A terminal program is usually found in the 'System' or 'Utilities' menu
    - Popular terminal programs are gnome-terminal, konsole and xterm (menu entry 'terminal')
- There are several shells available
  - zsh, bash, ksh, sh with similar functionality and syntax
  - tcsh and csh with a different syntax, not recommended
  - By default zsh is started in the terminal window (login shell)
  - zsh is almost identical to bash but has some advantages (see later)
  - bash cannot be started on login, but by typing bash you change the shell to bash

The shell prompt

- After invocation the shell displays a prompt and does wait for input
- The shell prompt looks different depending on the shell
  - zsh
  - bash



  - The zsh prompt displays the host name (pub2) and the working directory (~)
  - When typing after the prompt (the command line) a line editor is active
    - Backspace deletes before, Delete under the cursor, arrow keys, Home and End do work as expected, other keys such as CTRL-K, CTRL-W can be used to modify the line as well

Shell history, shell completion

- A history of text (commands) typed is preserved
  - Arrow keys up and down move within the history
  - Press Enter to execute a displayed command from the history
  - Press CTRL-O to execute a sequence of commands from the history instead
  - Search for commands starting with some text in the history by typing that text and then use PgUp, PgDn to search within the history
- Intelligent completion of commands, file names etc. is built in
  - Pressing Tab completes a unique part of all possible completions
    - les<Tab> displays less
  - Pressing Tab a second time shows all possible completions
    - less<Tab> displays less lessc lessecho lesskey
  - Pressing Tab further on cycles through the completions (Shift Tab backwards)
  - Completion depends on **context**: z<Tab> completes cmds, ls z<Tab> completes files
  - works on **commands** (e.g. z<Tab>), **users** (e.g. ~z<Tab>), **variables** (e.g. $Z<Tab>) **files** and **directories** (e.g ls z<Tab>) and even command line **options** (gcc -s<Tab>)

Shell syntax

- Two families of shells, only bash/ksh like syntax discussed
  - Shell scripts help automating simple tasks
  - All important language concepts existing: variables, conditionals, loops, functions...
  - Shell programming nevertheless not as flexible as other scripting languages
    - Correct quoting and amount of white space is crucial
    - For larger tasks perl, python etc. is recommended
- Use aliases or functions to define frequently used lengthy commands
  - See what aliases we have already defined: alias
  - Define new aliases, e.g. alias la='ls -la –color=tty | less'
  - See what functions are defined: functions
  - Define a new function: function v() { vim ${*} } ${*} means 'all arguments'
  - Aliases and functions may be set in ~/.zshrc

Shell variables

- Variables
  - Assignment: PRINTER=znlo1 — no space allowed, quote text with spaces
  - Usage: echo $PRINTER — can also be surrounded by “ (double quotes)
  - Visibility: export PRINTER — make vars visible to child processes (env vars)
- Environment variables control processes (env to list its names/values)
  - PATH — colon separated list of directories where the shell looks for commands
  - LANG — influences several aspects of language dependent input/output
  - Other useful ones like OS, HOST, SHELL, TERM, USER, LESSOPEN
- Variables can be set globally
  - Put the definitions in ~/.zprofile or ~/.zshenv
  - do not try to modify important environment variables such as PATH
  - Variables can be expicitly passed to commands: <VAR>=<value> <command>

Control flow

- Several forms of conditionals
  - `if [ $a = 1 ]; then; echo hi` terminate a command by newline or semicolon
    `else; echo ha; fi` spaces and semicolons are important here
  - `if test $a = 1; then; echo hi; fi` the cmd test is almost the same as the cmd [
  - `if [[ $a == 1 ]]; then; echo hi; fi` [[is not a command but a shell built in
  - `test $a = 1 && echo hi` calls echo if cmd before && returns 0 (success)
  - `test $a = 0 || echo hi` calls echo if cmd before || fails (content of $? not 0)
- Loops
  - `for i in 1 2 3; do; echo $i; done` (zsh only, bash needs a new line after do)
  - `for i in {1..10} do; echo $i; done;` (again, no one-liner in bash)
  - For other constructs (while, until) and many more details see man page

Shell scripts

- First line decides which program executes the script
  - #!/bin/zsh or #!/bin/bash or #!/bin/ksh or #!/bin/sh
  - Try to use shell syntax that is not specific to one of these shells
  - Can be used like any other command if the script is in the PATH and executable (chmod +x <script>), otherwise zsh <script> will work as well
- Scripts can be made more universal by using arguments
  - Available inside the script as $1, $2 etc.
  - or by 'eating' the arguments using <var>=shift
- Scripts can indicate success or failure
  - The return code (contained in variable $?) indicates success or failure
  - Can be explicitly set by exit 0 (success) or exit <number> failure
  - Will implicitly set otherwise by the last command executed in the script

Zsh documentation

- man zshall (compare also: man bash) — comprehensive
- http://zsh.sourceforge.net/ — reference URL
- http://grml.org/zsh/zsh-lovers.html — nice to read

Files and directories

- Some important things to remember
  - File and directory names are case sensitive
  - May contain arbitrary characters (not recommended though)
  - Some characters have to be escaped when typing, e.g. space in 'My\ Documents' (Tab completion would have done the correct escaping
  - file/directory names starting with dot are hidden from display (use ls -a instead of ls)
  - File name extensions such as .c .h may be important for proper functionality (compiling) of applications, others like e.g. .pl are a pure convention
  - Locations relative to the current dir can be written using . (this dir) and .. (parent dir)
- Hard and soft links
  - Hard links are referencing the same disk contents, are hard to detect in file listings, do not work across file systems etc. and should be avoided (ln oldfile newfile)
  - Soft links consist of text describing the path to a real file. They are easily recognized in ls -l listings and have less restrictions than hard links (ln -s oldfile newfile)

Special directory names

- Relative addressing of files and directories using
  - . (dot) the current directory
  - .. (two dots) the parent directory
- Home directories
  - most shells and some applications recognize the special meaning of the tilde char
  - ~ my home directory
  - ~foo home directory for a user with account name foo
- Special directory names
  - /afs/<cell_name> such as /afs/ifh.de, /afs/desy.de, /afs/cern.ch are mount points for (volumes in) the world wide uniform file system AFS
  - /lustre name space for mount points related to the lustre file system

Wildcards in the shell (globbing)

- Multiple file names (and some other strings) notation
  - Wildcard expansion is done on some types of info in the shell, most notably file and directory names
  - * (star) matches any string (including the empty string)
  - ** (zsh only, file names only) matches multiple directories
  - ? (question mark) matches any single character (not a null string)
  - [...] (chars enclosed in brackets) matches any of the enclosed characters
  - [^...] matches any character which is not in the given set
  - more wildcard patterns available depending on shell, see the man page
  - Powerful syntax: ls **/*.[ch] list all .c or .h files in current dir and subdirs
- When to quote wildcards
  - Wildcards are interpreted by the shell, not the application. To pass it on use quotes
    - tar tvf sources.tar.gz '*.h' lists only header files from the archive
    - scp pal.desy.de:'*.pl' ~/ copies .pl files from a Hamburg computer to here

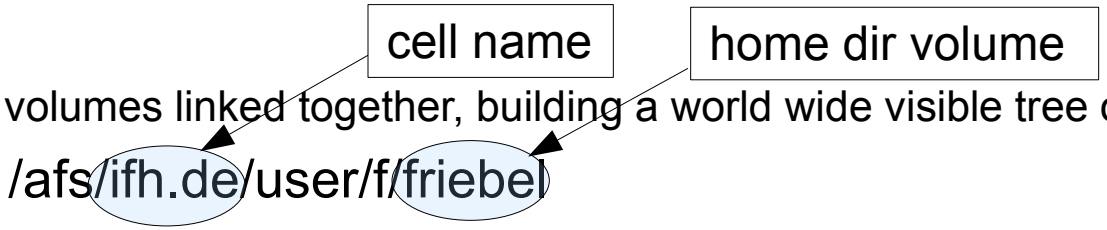
Navigating in the file system

- Using cd to change directory
  - cd — without arguments changes into your home directory
  - cd <path> — changes into a different directory (relative and absolute path notation)
  - cd - — changes to previous directory
  - cpd — lets you change into one of the previously used dirs (DESY zsh only)
  - cd $var — changes into a directory whose name is contained in $var
  - <path> — changes to <path> if path is not a command as well (zsh only)
- Finding a file or directory
  - locate <string> finds all files/dirs containing <string> in its absolute path. Only for files/dirs on the local disk, not in AFS and only for names that existed yesterday
  - find . -name <filename> — finds filename in or below current directory
  - ls =command (zsh only)(same as which command) absolute path for command

AFS Storage

- Most important for daily work, home directory is in AFS space
  - Important differences to local disk storage
  - No 'group' and 'other' UNIX rights
  - Almost all directories protected by ACL's
  - ACL changes can cause security holes, dirs may become world readable (literally!!!)
  - no ACL's for individual files
  - Restrictions for certain file types (hard links, named pipes) and concepts (like locking)
  - Authenticated access to AFS space, limited to 25 hours (AFS token)
  - Disk space is broken up into volumes having disk quota assigned

AFS space organization



- Individual volumes linked together, building a world wide visible tree of directories
- Home directory (**~/**) mounted under /afs/ifh.de/user/<initial>/<accountname>
- AFS space belonging to your group in /afs/ifh.de/group/<groupname>
- Quota in home dir is 500 MB, group space can be requested by group admins

  pts mem group:<groupname>_adm # yields the account names of admins

- A snapshot of the home directory taken at the evening before is in **~/.OldFiles** and can be used to retrieve accidentally deleted files (does not count towards your quota)
- Other cells address volumes in other institutions
  - /afs/cern.ch/... cern.ch CERN
  - /afs/infn.it/... infn.it INFN (Italy)
  - ...

Important AFS commands

- Useful AFS commands
  - klist # list validity of your Kerberos Tickets **and** the AFS token afs@IFH.DE
  - kinit # obtain a new Kerberos ticket **and** an AFS token (see also klog)
  - fs lq [<path>] # list the volume name and the quota for the current location or <path>
    - Volume names whose 2nd letter is n are usually scratch volumes (not backed up)
  - fs la [<path>] # list the access rights for the current dir or the dir belonging to <path>
    - The letters rlidwka stand for read/lookup/insert/delete/write/lock/admin rights
  - arcx recover # recover deleted or old versions of files from backup if existing
- Troubleshooting
  - Most access problems related to missing/expired AFS tokens (as well on Windows)
  - Use klist or tokens to verify you have a valid token, kinit or klog to get a fresh one
  - Locking your screen or logging out when leaving the computer helps (you get a new token on unlock or login)

AFS access to files

- AFS file and directory protections
  - All files in a directory do have the same ACLs in addition to the UNIX user rights rwx
  - All users belong to **system:anyuser**, authenticated users in addition to **system:authuser** and **<username>**
  - Users may belong to additional groups, which can be seen by

    pts mem <username>
  - Additional host based groups such as **desy-hosts** may be used to restrict access
  - Access to files is granted only if the access rights and group memberships allow it
- Default protections in your home dir
  - ~/ # **world wide** lookup access (ls), read and write access by you
  - ~/public # **world wide** read access, additionally writable by you
  - ~/private # read/write access by you, no other access rights
  - ~/.xxx # **some** directories such as ~/.ssh are accessible exclusively by you

Access rights for files and directories

- Display of rights
  - ls -l listing start with 10..11 chars displaying access rights
    - 1st char is l and d for symlinks and directories respectively, - otherwise
    - Then 3 x 3 chars rwx for read, write and execute rights for owner, group and others
    - Group and other rights do not have any meaning in AFS
    - The x right for directories is the right to cd into it, without that right no directory listing!
  - The permissions rwx for owner/group/other can be written as an octal number
    - If a bit in the number is set it means that the corresponding right is granted
    - Example: 644 means read and write right for owner, read rights for group and other users
  - In the AFS file system ACL's are used to further restrict access
    - fs la <path> displays ACL's in AFS
  - Very rarely further file attributes may have been set (see lsattr command)
  - SELinux may restrict rights further, not easily recognizable by users

Setting access rights for files and directories

- Non AFS file systems
  - chmod <mode> <file> sets rwx rights for owners, group and other
    - Mode written as string (who+-what): go-w (group and others without write right) or number
    - mode 600 - rw for owner, 644 – rw for owner, r for others, 755, rwx for owner, rx for others
    - chown and chgrp change ownerships, this is usually a privileged operation
- AFS
  - fs sa <path> <owner_or_group> <ACL_string> sets ACLs
  - ACL string looks the same as displayed with fs la, e.g.: fs sa ~ friebel rlidwka
    - rlidwka can be written as 'al'l, no rights are written as 'none' (deletes ACL entry)
    - Setting ACL's for other owners is not good, better define a new (pts) group and add users to that group, then restrict access using the newly created group
      pts creategroup <your_accountname>:<groupname>
      pts adduser <other_account> <your_accountname>:<groupname>
      fs setacl <path> <your_accountname>:<groupname> <access_rights>
    - Example: fs sa ~/www friebel:wwwfriends rliw

Basic commands for files and directories

mkdir [-p] <dirpath>	create a directory [and all missing dirs in <dirpath>]
rmdir <path>	deletes an empty directory (watch out for existing dot files!)
rm <path>	delete a file
rm -r <directory>	deletes <directory> and all its subdirs (even if non empty !!!)
cp <from> <to>	copy file with location <from> into directory <to> or file <to>
cp -a <dir> <to>	copy directory and all its subdirs into directory <to> by preserving all access rights and ownerships
mv <from> <to>	move (rename) file or directory to file or directory <to>
ln -s <old> <new>	the already mentioned command to create symlinks
touch <path>	create an empty file at location <path>

I/O Redirection

- Processes have by default 3 I/O streams open
  - 0 stdin input from the shell
  - 1 stdout standard output to the shell (the terminal window)
  - 2 stderr error messages to the shell
- I/O can be redirected
  - < inputfile read from a file instead from stdin
  - > outfile write to a fille instead to stdout
  - >> outfile append to an existing file
  - >>| outfile append to a file if existing, otherwise write to it
  - > outfile 2>&1 redirect stderr to stdout and then redirect all to outfile

Pipes

- Pipes connect stdout of one command with stdin of another one
  - cmd1 | cmd2 connects stdin of cmd1 to stdin of cmd2, stderr untouched
  - cmd1 |& cmd2 connects both stdout and stderr of cmd1 to stdin of cmd2
- One of the most powerful concepts in Linux
  - Linux/UNIX comes with lots of utilities working on stdin and producing stdout
  - Most of these utilities do a single task (comparable to words in a language)
  - Connections by pipes produce new commands (like a sentence formed of words)
  - In the Quick Reference Guide Scientific Linux marked by **F** (filter) or **f** (input filter)
- Sample more complex commands
  - du –exclude .OldFiles | sort -nr | head show disk usage and the 9 largest dirs
  - sort .history.pepe125.ifh.de | sed 's/.*;//' | awk '{print $1}' | sort | uniq -c | sort -nr|head
    show my 10 most frequently used commands on host pepe125

Processes

- Processes are always started from within another process
  - Normally this is the shell (which in turn was started by the WM)
  - Processes are started by duplicating the running process (**forking**)
  - From then on the duplicated process (**child process**) performs different operations
  - A child normally dies if the **parent process** is terminated/killed
- A process running in a shell can be put in background
  - Suspend the process by CTRL/Z, then continue it in background: bg
  - Show running background jobs: jobs
  - Bring a job back into foreground: fg [%<n>]
- Display processes
  - ps aux shows all processes
  - pp <pattern> show processes containing <pattern> in the ps output
  - top show processes using the most resources (cpu)
- Kill processes
  - kill [-signal] <PID> get the PID from ps, pp, top, signal is a number or a name
  - Signal 9 (TERM) terminates, 15 (KILL) kills
  - CTRL-C sends INT (2) and interrupts the process
  - A HUP (1) signal is sometimes used to signal processes rereading config information

Working with files

- Displaying file contents
  - less <filename> recommended way of displaying file contents
  - displays human readable information if possible for binary files, even more so after ini less
  - cat <filename[s]> writes contents to STDOUT, can be used to concatenate files
  - Editors are suited as well to display contents, danger of altering file, slower start up
- Other useful commands for working with files or used as filters
  - file <filename> determine file type based on its contents
  - diff <file1> <file2> display differences between two files (see also cmp)
  - grep <pattern> <file> find lines containing a string in a file
  - sort -k <n> <file> sort lines in a file
  - uniq <file> remove duplicate lines from a file (and optionally count duplicates)
  - wc <file> count chars, words and lines in a file
- Manipulating file contents
  - awk '<script>' <file> manipulate lines of space separated fields in a file
  - sed 'script' <file> manipulate lines in a file (using search/replace patterns)
- Working with archives (several files packed into one larger file)
  - Depending on archive type tar, zip, ar, rpm has to be used. To just browse in the archives less can be used (after 'ini less' for exploring archive contents further)

Using the batch system

- Why using batch
  - CPU or I/O intense tasks can heavily disturb interactive work on the same machine
  - Batch machines are usually faster than the local desktop, have more **memory** and the **network bandwidth** is typically a factor 10 bigger.
  - Important resources (e.g. Lustre) are available on batch machines only
  - Resources are shared among users in an optimal way without user communication
  - More fair distribution of resources using batch than we could do manually
    - Users cannot overuse resources (**fair scheduling**)
  - Important tasks can get **prioritized by admins** and get processed faster
  - Several CPUs can be assigned to one program for **parallel processing** without asking other users not to use these CPUs

Essential batch commands

- Submitting a batch job
  - qsub <script> — submits a batch job
  - It is essential to specify what resources the job will need, see later
- See the status of batch jobs
  - qstat [-u <account>] — show the status of jobs [for a certain account]
  - qstat -j <job_number> — show many details of a job, e.g. why still waiting
  - Finished jobs are visible using qstat only for a short time, afterwards use https://www-zeuthen.desy.de/dv-bin/batch/stat (make sure to select the proper farm)
- Delete one of your (waiting) jobs
  - qdel <job_number>deletes a waiting job (qdel -f for running jobs)
- See the status of available job queues
  - qstat -g c — very compact output
- See the status of available hosts for execution of jobs
  - qhost
- Get a machine for interactive work
  - qrsh — logs you onto a machine reserved for you (using ssh)

Requesting resources

- Without an explicit request minimal batch resources are provided
  - Usually not sufficient to run larger jobs
- Requesting resources
  - See https://dvinfo.ifh.de/Batch_System_Usage#Requesting_resources
  - Can be specified with qsub command or in script containing the job
  - Look into https://www-zeuthen.desy.de/dv-bin/batch/stat to see what resources the job really needed
  - Requesting to much resources puts you further down in the waiting queue and may block resources which otherwise could be used by others
  - When requesting to few resources your job **gets killed** on exceeding that resources
- Example: submit an 30 minute job with 2G memory
  - qsub -l h_vmem=2G -l h_cpu=00:30:00 <job_script>
  - To get a machine for 30' CPU time and 2G mem use the same parameters for qrsh

Batch job scripts

- Normal zsh or bash scripts
  - Start with usual shell script line #!/bin/zsh
  - Command line parameters for qsub can be written in the script using the prefix #$
  - A local directory that is removed at job end is provided: $TMPDIR
- Sample well commented job script
  - Can be found at https://dvinfo.ifh.de/Batch_System_Usage#Batch_job_submission
  - Never use /tmp to store temporary files, always use $TMPDIR
  - STDOUT and STDERR of job go by default into AFS home directory
    - Make sure the quota there is sufficient
    - Has bad impact to AFS file server if many batch jobs running simultaneously
    - Good practice is to redirect these files into a directory, where you do not read from
- Testing job scripts
  - Try to start with one or a few jobs only to avoid stressing the batch system
  - Minimize access of remote data during run time by copying data to/from $TMPDIR

Publishing results

- Might have experiences using Windows or Mac to produce documents
  - If you insist you can get access to a Windows machine from Linux using winrdp
  - For most tasks native Linux programs can be used
- Writing documents
  - Popular choices include OpenOffice (ooffice, oowriter) and LateX (latex)
- Producing graphics
  - Simple tasks can be done using OpenOffice (oograph), otherwise use gimp
- Plotting and Histogramming
  - gnuplot fairly simple yet powerful plotting package
  - rrdtool useful for time series plots, can be installed on request on SL5
  - root powerful data analysis framework, developed by HEP community
  - Commercial math packages like maple, mathematica, matlab
    - Plenty of licenses for maple only

On resource usage

- Resources provided by the computer center are shared by users
  - To help other users do their jobs do not waste resources
  - Usually there is a waiting queue for batch jobs due to lack of resources
  - Do not try to find tricks to circumvent the fair scheduling by the batch system
    - That does not work against the batch system but against other users (in your group)
- Try several methods to solve problems
  - Often a change in algorithms gives much more speedup than using more resources
    - One day of optimizing an algorithm can be worth several days of running jobs
  - Search the advice from your group members, they often do have recipes for tasks
  - Look in various sources in the internet, often the task (or a similar one) you need to program has already be done by somebody else or at least libraries do exist containing useful procedures to simplify your programming task

Finding and correcting bugs

- Write tests to verify the software (defined input produces known result)
  - Rerun the tests after changes
  - If something goes wrong typically the most recent changes contain the bug
- Write code that is easier to debug by others
  - Do not use tricky constructs to gain minimal speed increases (don't optimise)
  - Optionally produce test output where the level of detail can be controlled
  - Do comment the code, use variable and procedure names related to the problem
  - Write assertions at critical places (code that must never fail)
- Use system tools to find bugs
  - Use the debugger to run a program under its control gdb program
  - Use strace to see all system calls (especially important: open statements)
  - Watch the memory consumption of a process to find memory leaks

Final remarks

Thank you for your attention

- do you have questions now?
- If you have questions later: uco-zn@desy.de
- If you have suggestions
  - improving this talk I'd like to hear from you wolfgang.friebel@desy.de
  - improving the documentation (web pages, booklet) we would like to hear from you

Have a nice stay here at Zeuthen

HELMHOLTZ ASSOCIATION

DESY