CS6 Practical System Skills

Fall 2019 edition

Leonhard Spiegelberg
lspiegel@cs.brown.edu

08.98 Recap

Last lecture

- hostnames
- SSH
  - password based
  - key pair based authentication
  - configuration via ~/.ssh/config
  - logging in securely to a remote
  - running commands on a remote machine
- scp and rsync to copy files between local machine, remote(s)

More on scp/rsync

08.99 archives

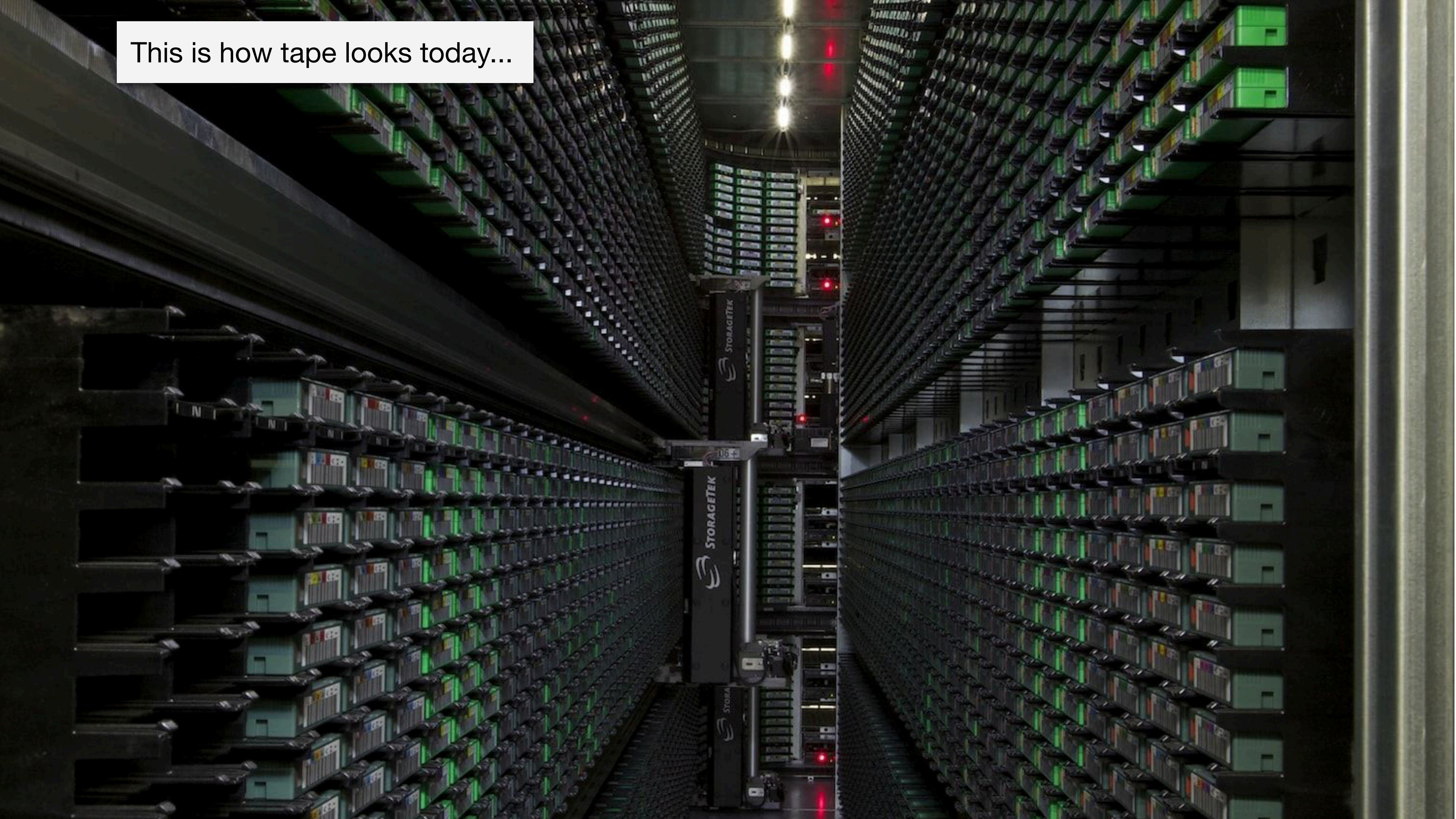
⇒ often more convenient to send one large file than many small ones

⇒ `tar` = tape archiver is a tool to create one file out of many

⇒ resulting file is called a tarball or tar archive

⇒ typical file extension: `.tar`

This is how tape looks today...



08.99 tar

⇒ tar allows you to drop the - when combining options.

option (can use with/without -)	meaning
`x`	extract files
`v`	verbose, print file names when extracted or compressed
`f`	use following tar archive for the operation
`c`	create archive
`z`	x or c is specified, interpret archive as gzipped file

08.99 tar examples

1. create archive `tar cvf archive.tar list`
2. extract all files from archive `tar xvf archive.tar`
3. extract files to directory `tar xvf archive.tar -C dest`
4. extract single/multiple files from tar archive via

   `tar xf archive.tar /path/to/file.txt`
5. list contents of archive.tar via `tar tf archive.tar`
6. appending files to archive.tar via `tar rf archive.tar`

08.99 compressing tar files

⇒ to reduce the size of a tarball, often it is compressed afterwards and the extension of the compression program appended

extension	compress	uncompress
.gz	gzip	gunzip
.bz2	bzip2	bunzip2
.xz	xz	unxz

there are many more compression algorithms, e.g. 7zip, rar, zip, snappy. The ones above are standard ones available typical on *NIX platform

08.99 tar + compression

⇒ to compress a tarball we can either pipe it with a compressor or run it in 2 commands

Example:

use - to signal tar to write to stdout

```
tar cf - *.txt | bzip2 > archive.tar.bz2

bunzip2 -c archive.tar.bz2 | tar xf -
```

use -c option to output to stdout

use - to signal tar to read from stdin

08.99 tar - the z option

⇒ for convenience tar has an option z to work with a compressed gzip file.

Example:

```
tar cvzf archive.tar.gz *.txt
```

```
tar xzf archive.tar.gz
```

⇒ to use bzip2 there is an option `j`, for general compress tool use `Z` or `a` to auto determine compression program

09 Processes

CS6 Practical System Skills

Fall 2019

Leonhard Spiegelberg *lspiegel@cs.brown.edu*

09.01 What is a process?

A process is a (running) instance of a program in memory.

Each process has 4 properties associated with it:

1.	**PID**	process-id unique identification number for a running process
2.	**PPID**	parent process-id process id of the process who launched the process
3.	**TTY**	(teletypewriter) terminal to which the process belongs to.
4.	**UID**	user-id user to which the process belongs to

09.02 Listing processes

`ps` prints **p**rocess **s**tatus

```
Leonhards-MacBook-Pro:~ sealion$ ps

  PID TTY           TIME CMD

 3761 ttys000    0:00.19 -bash
 4227 ttys001    0:00.08 -bash
 8875 ttys002    0:00.33 -bash
16867 ttys002    0:00.12 ssh tux@cs6server
16885 ttys003    0:00.07 -bash
16930 ttys003    0:01.04 ssh -X tux@cs6server
```

PID = process ID
TTY = terminal
TIME = CPU time given to the process
CMD = command used to start the process

09.02 ps selecting what information to display

```
ps -o commalist
```

with commalist being a list of keywords separated by comma

keyword	meaning
`%cpu`	cpu utilization of the process in %
`args` (can also use `cmd` or `command`)	command with all its argument as string
`cputime`	cumulative CPU time (check man page for format)
`pid`	process id
`ppid`	parent process id
`tty` (can also use `tt`)	terminal the process is connected to
`uid`	(effective) user id

09.02 ps -o example

```
tux@ip-172-31-29-145:~$ ps -o user,group,uid,pid,ppid,tty,cputime,%cpu,args
USER     GROUP      UID   PID  PPID TT            TIME %CPU COMMAND
tux      tux       1001 13311 13310 pts/0     00:00:00  0.0 -bash
tux      tux       1001 13832 13311 pts/0     00:00:00  0.0 ps -o user,group,...
```

```
this here reads fully
ps -o user,group,uid,pid,ppid,tty,cputime,%cpu,args
```

as always many more options, please read the man pages for your system!

09.02 Daemons & Zombies

A *daemon* is a process that runs continuously and is (usually) not attached to a terminal.

⇒ e.g. sshd is a daemon

⇒ daemons are named with d at the end often

A *zombie* is process which is not running more but still exists in the process table, i.e. still has a PID assigned to it.

09.03 ps - listing all processes

`-a` to list all process except session leaders and processes which are not associated with a terminal (i.e. daemons usually)

`-A` list all processes

⇒ there are quite a few processes running on a system. Helpful to feed them to e.g. head/tail or a pager like less/more

09.03 ps -A

Example:

```
tux@ip-172-31-29-145:~$ ps -A -o user,group,uid,pid,ppid,tty,args | tail -10
root     root           0 13422     2 ?        [kworker/u30:1]
root     root           0 13423  2280 ?        sshd: tux [priv]
tux      tux         1001 13504 13423 ?        sshd: tux@pts/1
tux      tux         1001 13505 13504 pts/1    -bash
root     root           0 13743     2 ?        [kworker/u30:3]
tux      tux         1001 13770 13505 pts/1    python3
tux      tux         1001 13857 13311 pts/0    ps -A -o user,group,uid,pid,...
tux      tux         1001 13858 13311 pts/0    tail -10
root     root           0 19603     2 ?        [xfsalloc]
root     root           0 19608     2 ?        [xfs_mru_cache]
```

09.03 listing all processes

`-x` lists all processes which are owned by you

⇒ often uses in combination with a, i.e. `ps -ax`

```
tux@ip-172-31-29-145:~$ ps -x
  PID TTY      STAT   TIME COMMAND
13193 ?        Ss     0:00 /lib/systemd/systemd --user
13194 ?        S      0:00 (sd-pam)
13310 ?        S      0:00 sshd: tux@pts/0
13311 pts/0    Ss     0:00 -bash
13504 ?        S      0:00 sshd: tux@pts/1
13505 pts/1    Ss     0:00 -bash
13770 pts/1    S+     0:00 python3
13879 pts/0    R+     0:00 ps -x
```

09.04 How a process is born

⇒ processes in UNIX are created using 2 steps: **fork** and **exec**

⇒ to create a new process, a fork system call is performed which creates a copy of the calling process

⇒ this forked process or child process, inherits everything that the parent (i.e. the calling process) has in memory, but gets a new pid

⇒ exec replaces the current process with a new one, i.e. loads a program into the current process space

09.04 How a process is born

⇒ first process started is an init system (here systemd) which launches system daemons and processes with PID=1, PPID=0.

```
tux@ip-172-31-29-145:~$ ps -o user,group,uid,pid,ppid,tty,args -ax
USER     GROUP      UID   PID  PPID TT       COMMAND
root     root         0     1     0 ?        /lib/systemd/systemd --system --deserialize 38
root     root         0     2     0 ?        [kthreadd]
root     root         0 13191  2280 ?        sshd: tux [priv]
tux      tux       1001 13193     1 ?        /lib/systemd/systemd -
tux      tux       1001 13194 13193 ?        (sd-pam)
systemd+ systemd+   100 13299     1 ?        /lib/systemd/systemd-r
tux      tux       1001 13310 13191 ?        sshd: tux@pts/0
tux      tux       1001 13311 13310 pts/0    -bash
systemd+ systemd+   101 13314     1 ?        /lib/systemd/sys
root     root         0 13345     2 ?        [kworker/0:0]
root     root         0 13423  2280 ?        sshd: tux [priv
tux      tux       1001 13504 13423 ?        sshd: tux@pts/1
tux      tux       1001 13505 13504 pts/1    -bash
root     root         0 13743     2 ?        [kworker/u30:3]
tux      tux       1001 13882 13311 pts/0    ps -o user,group,uid,pid,ppid,tty,args -ax
```

kernel thread daemon in linux

systemd is the first daemon launched. Under Mac OS X, the init system is /sbin/launchd

09.04 How a process is born - pstree

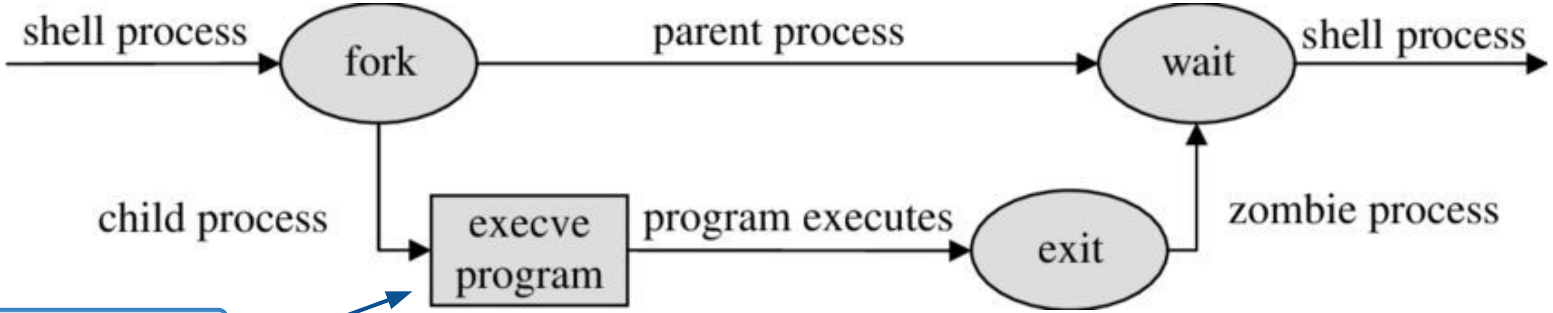
⇒ `pstree` can be used to show fork structure (add username as argument)

```
tux@ip-172-31-29-145:~$ pstree tux
sshd───bash───pstree

sshd───bash───python3

systemd───(sd-pam)
```

09.05 Running commands & forking



execve is just a variation of exec

⇒ when you type a command CMD in the terminal and press ENTER, the shell forks itself to create a shell child process and then runs exec CMD and waits for the child process to terminate

⇒ you can also execute CMD by running exec CMD, however this will replace the shell with the running process. I.e. when the process terminates, *there is no more shell.*

09.05 Example

```
Leonhards-MacBook-Pro:~ LeonhardS$ ssh tux@cs6server
Welcome to Ubuntu 18.04.2 LTS (GNU/Linux 4.15.0-1044-aws x86_64)
...
Last login: Tue Sep 26 02:08:38 2019 from 74.297.48.5
tux@ip-172-31-29-145:~$ ls /home/tux
decrypted.txt  encrypted.txt  example.sh  message.txt  script.sh
tux@ip-172-31-29-145:~$ exit
logout
Connection to cs6server closed.
Leonhards-MacBook-Pro:~ LeonhardS$
```

after child process terminated, back to bash process

```
Leonhards-MacBook-Pro:~ LeonhardS$ ssh tux@cs6server
Welcome to Ubuntu 18.04.2 LTS (GNU/Linux 4.15.0-1044-aws x86_64)
...
Last login: Tue Sep 26 02:08:38 2019 from 74.297.48.5
tux@ip-172-31-29-145:~$ exec ls /home/tux
decrypted.txt  encrypted.txt  example.sh  message.txt  script.sh
Connection to cs6server closed.
Leonhards-MacBook-Pro:~ LeonhardS$
```

bash was replaced with ls, thus "no more shell"

09.06 Variables

within a script we can read PID, PPID, UID via `$$`, `$PPID`, `$UID`

Example:

```
tux@ip-172-31-29-145:~$ echo $$
13505
tux@ip-172-31-29-145:~$ ./vars.sh
process ID: 14139
parent process ID: 13505
user ID: 1001
```

var.sh

```
#!/bin/bash
echo "process ID: $$"
echo "parent process ID: $PPID"
echo "user ID: $UID"
```

⇒ Under GNU/Linux `pidof name` can be used to find process ids of name processes

How to launch and work with long running processes?

09.07 Foreground and background processes

⇒ when we launch a new process (i.e. by typing a command), it runs per default as a foreground process

⇒ a **foreground process** is one that we can interact with using the terminal, i.e. it waits for user input via the attached terminal

⇒ a **background process** runs independently of the human user

09.08 Interacting with processes

⇒ to interact with processes we can send them signals

⇒ for a list of all supported signals, run `kill -l`

Synopsis:

```
kill [-s signal_name] pid ...
kill -l [exit_status]
kill -signal_name pid ...
kill -signal_number pid …
```

default signal send by kill is usually SIGTERM

09.08 Interacting with processes

signal name	abbreviation	code	english	meaning
`HUP`	`SIGHUP`	`1`	hang up	sent to process when its controlling terminal is closed
`INT`	`SIGINT`	`2`	interrupt	program interrupt, i.e. typically issued by user. Tell a process to stop doing what it is doing right now, used for REPLs a lot.
`QUIT`	`SIGQUIT`	`3`	quit	quit process for misbehaving process, usually produces a core dump.
`KILL`	`SIGKILL`	`9`	kill	non-catchable, non-ignorable kill
`TERM`	`SIGTERM`	`15`	terminate	software termination signal, politely ask program to terminate. Normal way to stop a process.

09.08 Sending signals to foreground process

⇒ when you work in the terminal, you can send signals to the foreground process with the following keyboard shortcuts (configurable)

`SIGINT`	`Ctrl + C`
`SIGQUIT`	`Ctrl + \`
`send EOF marker`	`Ctrl + D`

Note: Depending on the terminal you're using, different keyboard shortcuts are necessary. You may also configure it to send additional signals.

09.08 How to terminate a process?

1. Try `SIGINT` (`Ctrl + C`)
2. Some programs terminate if you send EOF marker via `Ctrl + D`
3. If this does not work, send `SIGTERM`, i.e. via `kill` (default signal)
4. Send `SIGQUIT` if `SIGTERM` did not work (`Ctrl + \`)
5. If all of this failed, use `kill -9 pid`

you can get pid via `pidof` or better, via `ps -ax`

09.09 Launching a background process

⇒ to launch a background process append &

Example:

```
./long-running-script.sh &
[1] 22745
```

long-running-script.sh

```bash
#!/bin/bash

echo "starting a slow script..."
for i in `seq 1 10`
do
  echo "iteration $i, let's go to sleep..."
  sleep 1s
done
echo "...done!"
```

⇒ will produce output with a job number and pid of the launched process

⇒ output of background process will be still printed to terminal!
Use redirection to avoid this!

09.09 listing background processes/jobs

⇒ we can get a list of running background processes by running the command `jobs`

silent-slow-script.sh

```
#!/bin/bash
for i in `seq 1 100`
do
  sleep 1s
done
```

```
tux@ip-172-31-29-145:~/lecture07$ jobs
[2]-  Running                 ./silent-slow-script.sh &
[3]+  Running                 ./silent-slow-script.sh &
```

09.09 Suspending a process, fg & bg

⇒ you can suspend the foreground process by issuing `Ctrl + Z`

⇒ to bring it back from suspended mode, use

`fg %num`	to make it the foreground process
`bg %num`	to make it a background process

⇒ `%num` is the job number, i.e. retrieved via `jobs`

⇒ instead of `%num`, you can use `%+` for the current job and `%-` for the previous one

09.10 Launching long-running remote processes

⇒ we can use ssh to start remotely a process and use & to not wait for it

Example:

```
ssh tux@cs6demo "/home/tux/long-running-script.sh > /home/tux/out.txt 2>&1 &"
```

note the quotes, else redirection would be locally!

09.10 Launching long-running remote processes

⇒ Alternative: login via ssh, start process via &, logout

⇒ Problem: When exiting the shell via exit (i.e. terminating the SSH session), a SIGHUP is issued.

⇒ This may cause some processes to terminate!

⇒ Solution: start process with nohup, to ignore HUP signal, i.e.

```
nohup ./some-process.sh &
```

End of lecture.

Next class: Tue, 4pm-5:20pm @ CIT 477