Linux Tips For Researchers

By: Dustin Minnich

nohup

nohup allows you to keep commands running when you disconnect from an SSH session or when you close a terminal.

```
Open a terminal
Issue find / -print
Issue ps aux | grep find in another terminal. It is
running.
Close and re-open the terminals.
Issue ps aux | grep find.  Find is no longer running.
```

This same thing happens if you ssh into your box from home, start a compile/job and then turn off your computer at home.

To fix this, issue add a **nohup** in front of the **find / -print** command in the sequence above and the command won't be killed when you close the terminal.

Obviously, replace the **nohup find / -print** command with whatever command you want to run.

NOTE: Some jobs may need to be sent to the background for this to work. You can do this by adding a **&** after the command.

rsync

keep the same files in multiple places.

```
On your workstation
    mkdir project
    cd project
    touch a
    touch b
    vim a (write your code)
    cd ..
```

```
On your server
    mkdir project
On your workstation
    rsync -avz project/ netid@server:project/
On your server
    cd prjoect
    vim c (write more code)
    cd ..
    rsync -avz project/ netid@workstation:project/
```

In this example you are writing code on two different machines. Each time you run the rsync command you make sure the other machine has the exact same copies of your files as the machine you are working on. Rsync only copies the changed files to and from places, so it is super efficient if you are working with large example data sets you un-tarred--because they would only need to be copied over the first time.

NOTE: You will need to use full paths (/md2/netid/project) for the destination instead of project/ like I did above.

top/htop
view an updating snapshot of your systems performance

top and **htop** are simple commands you can use to see an interactive list of all of the processes running on the machine you are logged into. From inside of top you can also see how much memory and processing power the machine is currently using. This is a quick and easy way to see how hard a shared machine is currently working so you can decide if now is a good time for you to run your job or not.

Use "**q**" to exit top.
Use "**h**" to learn more. Like how to change what column the processes are sorted by and how to kill unruly processes.

tail -f
watch how a text file changes in real time.

```
Open a terminal
echo hi >> tail_example.txt
```

```
Open another terminal
```
tail -f tail_example.txt
```
Go back to the original terminal
```
echo bye >> tail_example.txt
```
Go back to the 2ⁿᵈ terminal.  You should see bye on your
screen.
```

This is useful for watching log files. If your compile/job writes status reports to a log file while it is running, you can use **tail -f** to easily keep up with what it is doing.

watch
run the same command over and over again so you can watch for changes

```
Open a terminal
```
mkdir watch_example
cd watch_example
```
Open another terminal
```
watch ls -lha watch_example
```
Go back to the original terminal
```
touch a
```
a should show up in the ls output of the 2ⁿᵈ terminal.
```

If your compile/job creates predictable file names, this is an easy way to keep track of what it is doing in real time. Also, you don't have to use an **ls,** you can continuously see the output of any command.

batch/at
run commands at specific times.

```
Open a terminal
Create a script file containing the command that you want
to have ran.
```
echo "echo 'blah' >> /tmp/at_example.txt" > at.sh
chmod +x at.sh
at -f at.sh -v 09:25am
cat /tmp/at_example.txt at 9:26am. You will see "blah".

Now you can run jobs in the middle of the night when nobody else is using the system without having to stay up to kick them off.

Google *at examples* or **man at** for setting specific dates or to learn how to see and remove pending jobs.

```
Open a terminal
batch
echo at2 >> /tmp/at_example.txt
ctrl+d
cat /tmp/at_example.txt
```

batch runs your command when it determines that the system you are on isn't under a heavy load. I don't know how good it is at determining this or how several batch jobs are prioritized (first in first out?), but this could be an interesting thing for people working on shared machines to play with. Everybody could submit their jobs and they would run one after the other or possibly at the same time if they aren't intense jobs all without human interaction.

ssh –X –Y
run a GUI application from another Linux machine on your Linux or Mac machine.

```
Open a terminal
ssh -X -Y netid@remotemachine.nicholas.duke.edu
xclock
The clock application on the remote machine will appear on
your screen.
```

You can run things like Matlab like this as well.

NOTE: This is possible from a Windows client machine as well but I don't have time to go into that now. Google *Putty+Xming* or shoot me an email and I'll gladly help.

screen

ever present and persistent terminals


```
Open a terminal
screen
ctrl+a c (create a new tab)
ctrl+a n (go to the next tab)
ctrl+a p (go to the previous tab)
ctrl+a d (detach from screen)
screen -r (reattach to a detached screen).
```

This is awesome because it allows you to use the keyboard more and because you keep one work environment all the time. For instance, you can run screen on your workstation and open one tab and start a local compile job that logs to standard out/your monitor. You can then open another tab, ssh into another machine and start another compile job that logs to your monitor. You can then open another tab to do some other standard work. Now it is time to go home. You hit **ctrl+a d.** Then when you get home you ssh into your machine and do a **screen -r** and you see the status of the jobs and everything else is just how you left it.

Screen is super-powerful and very customizable. Google *.screenrc examples* for different ways you can make it look.

Bash scripting

Automating simple things using the shell. Please note that this is bash and NOT C shell.


```
Open a terminal
find / -print
Open another terminal
if [ $(ps aux | grep find | grep -vc grep) -ge 1 ];
then
echo "find is running"
else
echo "find is not running"
fi
```

You should see "find is running"(if you typed fast enough)
Close the terminal running find and issue the command
again. You should see "find is not running".

Now you know how to do simple **if/else** constructs and how I prevent long running rsync backups from kicking off over and over again in cron if they happen to be taking more than 24 hours to complete.

```
Open a terminal
mkdir for_example
cd for_example
touch {1..30}.txt (create some fake data files)
ls
for X in *.txt
do
cp ${X} ${X}.`date +%Y-%m-%d`  (backup your data files
before you do any of todays work)
done
vim 1.txt  (do your daily work)
touch {1..5}.txt.2010-03-15  (create some fake files –
symbolizing that you have done this same process everyday
for years)
rm -f *.txt.2010* (every once and a while you delete old
backups)
ls
```

Now you know how to use **for** constructs and have created yourself a poor mans version control system.

git
version control system

git is a complicated but extremely powerful beast. I barely use it myself and am by no means an expert. If you plan on using it, you should probably buy a book on it.

```
Open a terminal
```

```
mkdir git_example
cd git_example
git init
git status
vim basecode.sh   (v1 code snippet)
git status
git add basecode.sh
git status
git commit (v1 of base code)
git status
vim basecode.sh (delete all lines -symbolizing a mistake)
cat basecode.sh (you see nothing)
git status
git add basecode.sh
git status
git commit (oops i deleted stuff)
git log
git reset --hard [commit uuid of the v1 of base code]
cat basecode.sh (your data is back)
git log (you went back in time)
```

Now, lets say you have two projects. One studying the atlantic ocean and one
studying the pacific ocean. You want them both to use the same base code, but
you want to write different pieces of extension code or want to use the input
files with the same name but different data for each project. You can use git
branches to do this.

```
git checkout -b atlantic (-b creates a branch)
vim data.txt (insert data pertaining to atlantic)
git add data.txt
git commit (added atalantic data.txt)
git log
git checkout master (main branch with just base code)
git checkout -b pacific
vim data.txt (insert data pertaining to pacific)
git add data.txt
git commit (added pacific data.txt)
git checkout atlantic
cat data.txt
```

```
git checkout pacific
cat data.txt
```

You can merge branches as well. Say you wanted to update your base code and wanted those changes to appear in both the atlantic and pacific projects.

```
git checkout master
vim basecode.sh  (v2 code snippet)
git add basecode.sh
git commit (v2 of base code)
git log
git checkout atlantic
git rebase master
cat *
git checkout pacific
git rebase master
cat *
git log
```

http://library.edgecase.com/git_immersion/index.html and **man git** has more information and talk about proper use if multiple people are working on the code