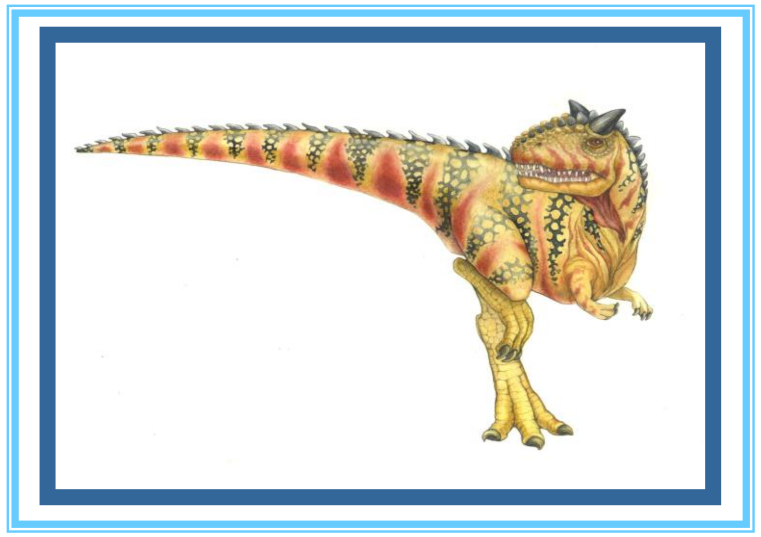
Project 5
Linux System Programming



Linux System Programming

- This is a large set of problems that covers many topics of Linux system programming
- It is not meant to be done in one week or even two or three weeks. It should serve as a repository of problems for the student to learn as many aspects as possible of the Linux system, for the rest of the semester
- The intention is to provide many examples of Linux programs and techniques, and serve as a good repository for training toward the final course exam
- We will try to solve some problems from this set along the semester but may not have time to cover everything. You are encouraged to solve as many as you can independently
- Some of these problems will serve as a basis for the final exam (probably with some modifications)

How to get the source code

- The source code to all problems is located at:

```
~samyz/os/proj5
```

- You can copy it to your Linux account by following the steps in the next slide:

```
cp –r ~samyz/os/proj5 ~
```

  Remember that '~' is a short name to your **home directory**!

Copying to Your Linux Account

- Connect to your Linux account using the putty.exe program. If you forgot how to do that, then read again this link: http://www.samyzaf.com/braude/OS/index.html#linux
- Your first task is to copy all source files from my account to your account:

```
cp -r ~samyz/os/proj5 ~
```

- This command will create a new directory **`~/proj5`** in your account with all the source files inside
- List the files in this directory to make sure it exists and that it contains all the source files in this project
- If you have to, you can remove this directory by the command:

```
rm -rf ~/proj5
```

  and then get a fresh copy of this directory:

```
cp -r ~samyz/os/proj5 ~
```

The gcc compiler

Problem 1:

- Enter the directory **~/proj5**
- Use the **gcc** compiler to compile the **C** program `copyfile.c`:

```
gcc copyfile.c -o ~/bin/copyfile
```

- Describe what happened exactly, and try to understand how to use the new `copyfile` system program?
- Experiment with the `copyfile` command in different directories:
  - What happens if you try to copy a file which does not exist?
  - What happens if you repeat the command twice on same arguments?

Playing with the copyfile command

- Go back to directory ~/proj5
- Download the text file cinderella.txt by the command :

```
wget http://www.samyzaf.com/braude/OS/code/cinderella.txt
```

(make sure you copy this path as is: Linux paths are case sensitive! OS != os)

- Run the command:

```
copyfile cinderella.txt cinderella_copy.txt
```

  - Did you get the new file `cinderella_copy.txt` ?
  - Is this file an exact copy of `cinderella.txt` ?
  - How can you be 100% sure ?!
    Hint: read about the Unix `diff` command
    Challenge: write `compare_files` program in Python
    It should return `True` if the two files are identical, `False` if not.
    (to check your answer see: `~samyz/bin/compare_files`)

Fixing the copyfile command

Problem 2:

- Use the Linux command "`ls -l`" for finding the last time the file `cinderella_copy.txt` was modified.
- Run the command in stage 3 again. What happened to the old `cinderella_copy.txt` file?
- Fix the problem in `copyfile.c` so that it does not overwrite an existing file but issues a warning only
  Hint: try to open the target file for reading, if you cannot, then it does not exists …

```
        if ((f2 = open(argv[2], O_RDONLY, 0)) >= 0)...
```

The g++ compiler

Problem 3:

- Go back to directory **`~/proj5`**
- Use the g++ compiler to compile the C++ program **`pid_info.cpp`**:

```
g++ pid_info.cpp -o ~/bin/pid_info
```

- Run the command **`pid_info`** from the command line and try to understand its output (add it as a comment in the program code)
- Use the Unix **`ps`** command to find who is the parent process of the **`pid_info`** process?
- Write the same program in Python: **`pid_info.py`**

Working with directories

Problem 4:

- Go back to `~/proj5`
- Read the C program `dirtest.c`
- Compile the program:

```
gcc dirtest.c –o ~/bin/dirtest
```

- Try to understand what it does and write your explanations at the program code (as comments)
- Write similar program `listfiles.c` which accepts a directory path and lists its files:

```
u12345@brdlinux:~> listfiles /usr/games
banner chess fortune tetris checkers candycrash …
```

Note: this is the minimal start for the Unix `ls` command …

Working with processed

Problem 5:

- Go back to `~/proj5`
- Read the C program `who_is_my_parent.c`
- Compile the program:

```
        g++ who_is_my_parent.cpp –o ~/bin/who_is_my_parent
```

- Use the Linux `ps` command to find the parent of this process
  To get full help on the `ps` command run: `ps --help all`

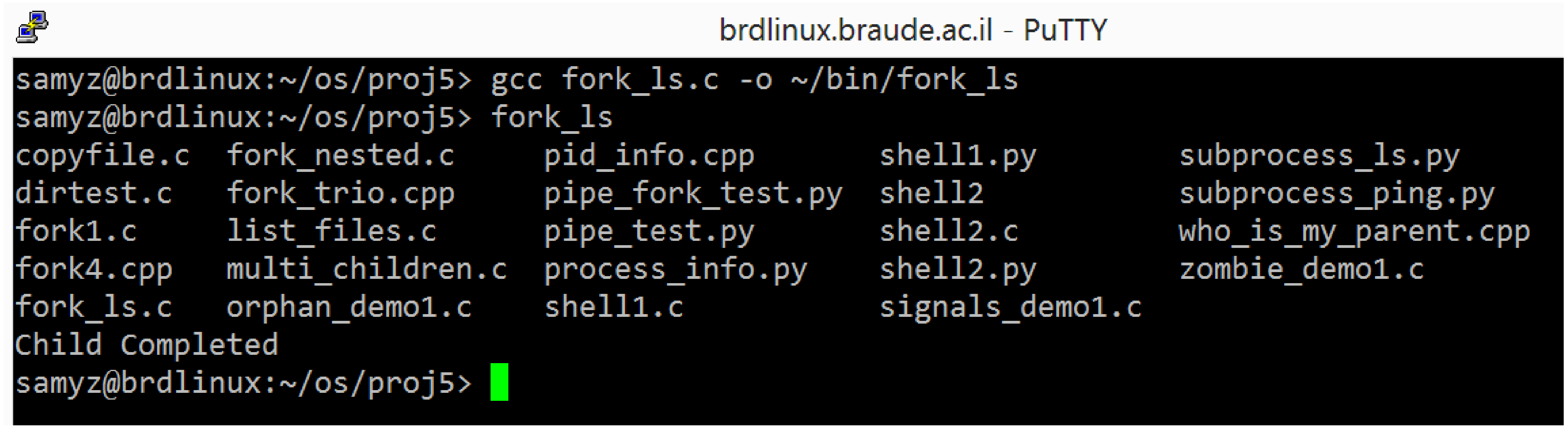
fork ls

Problem 6:

- Go back to `~/proj5` and read the program `fork_ls.c`
- Compile this program by running:

```
gcc fork_ls.c -o ~/bin/fork_ls
```

- Run this program and explain its output



```
samyz@brdlinux:~/os/proj5> gcc fork_ls.c -o ~/bin/fork_ls
samyz@brdlinux:~/os/proj5> fork_ls
copyfile.c  fork_nested.c     pid_info.cpp       shell1.py       subprocess_ls.py
dirtest.c   fork_trio.cpp     pipe_fork_test.py  shell2          subprocess_ping.py
fork1.c     list_files.c      pipe_test.py       shell2.c        who_is_my_parent.cpp
fork4.cpp   multi_children.c  process_info.py    shell2.py       zombie_demo1.c
fork_ls.c   orphan_demo1.c    shell1.c           signals_demo1.c
Child Completed
samyz@brdlinux:~/os/proj5> 
```

fork, execlp, and wait

Problem 7:

- Go back to `~/proj5`
- Read the C program `fork1.c`
- Compile the program:

```
gcc fork1.c –o ~/bin/fork1
```

- Try to understand what it does and write your explanations at the program code (as comments)

Fork: apples and bananas

Problem 8:

- Run the following programs:

  `/home/samyz/bin/apples`
  `/home/samyz/bin/bananas`

- Go back to `~/proj5`. You can view there the source code for `apples.c` and `bananas.c`, but you don't have to compile them since you will run them from my account (`/home/samyz/bin)`

- Write a program `fork_apples_bananas.c` which forks two processes:
  - Child process runs `/home/samyz/bin/apples` using the execlp() system call
  - Parent process runs `/home/samyz/bin/bananas` using the execlp() system call

- Hint: Look at the previous program and get the idea

- To check if your solution is OK, compare it to my solution at:

  `~samyz/os/proj5.sol/fork_apples_bananas.c`

Fork Trio

Problem 9:

- Go back to `~/proj5` and read the program `fork_trio.cpp` (this is C++ program that uses the `std` namespace)
- Compile this program by running:

```
g++ fork_trio.cpp -o ~/bin/fork_trio
```

- Run this program and explain its output
- How many processes are created after running this program ? Please explain!
- How many times the last line is printed and find which processes are printing it? (Hint: use `getpid()` ...)

Fork Nested

Problem 10:

- Go back to `~/proj5` and read the program `fork_nested.c`
- Compile this program by running:

  ```
  gcc fork_nested.c -o ~/bin/fork_nested
  ```

- Run this program and explain its output
- How many process area created by this program?
- Edit the program so that it shows for each line which process is printing it

- To check if your solution is OK, compare it to my solution at:

  ```
  ~samyz/os/proj5.sol/fork_nested.c
  ```

Zombie Process

Problem 11:

- Go back to `~/proj5` and read the program `zombie_demo1.c`
- Compile this program by running:

```
gcc zombie_demo1.c –o ~/bin/zombie_demo1
```

- A zombie process (**defunct** in Linux) is a process which has terminated but cannot be removed by the OS from the **process table** since its parent is too busy to collect its exit status (or forgot about it)
- Run the `zombie_demo1` program. The parent should enter an infinite loop, so you cannot use the terminal
- Start a new putty.exe session and use the Linux `ps` command to trace the child process (by its pid) and its status

Orphan Process

Problem 12:

- Go back to `~/proj5` and read the program `orphan_demo1.c`
- Compile this program by running:

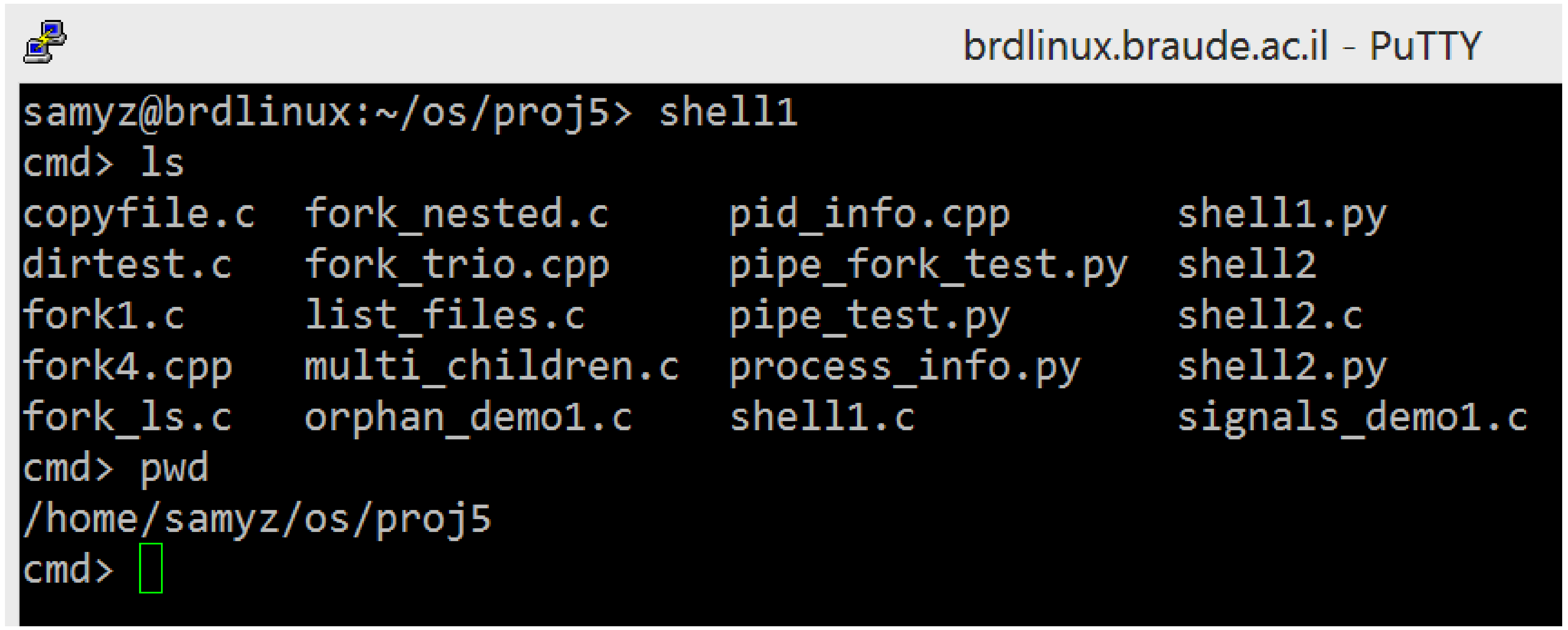
```
gcc orphan_demo1.c –o ~/bin/orphan_demo1
```

- An **orphan process** is a process whose parent has terminated
  - Every Unix process (except init) must have a parent
  - The question is what happens when to a child process when its parent has terminated?
  - To see the new parent run the command: `ps -u <username> -l`
- Run the `zombie_demo1` program. The parent should enter an infinite loop, so you cannot use the putty.exe terminal
- Start a new putty.exe session and use the Linux `ps` command to trace the child process (by its pid) and its status
- To terminate the child process, get his pid and use the command: `kill -9 pid`

Very Simple Shell

Problem 13:

- Go back to **`~/proj5`** and read the program **`shell1.c`**
- Compile this program by running:
  `gcc shell1.c -o ~/bin/shell1`
- Run this command (shell1) and feed it with a few examples of command names like: **`ls, dir, whoami, pwd, ...`**
- Can you run a command that accepts arguments? (like: '**`ls /usr/src`**')



```
samyz@brdlinux:~/os/proj5> shell1
cmd> ls
copyfile.c  fork_nested.c     pid_info.cpp       shell1.py
dirtest.c   fork_trio.cpp     pipe_fork_test.py  shell2
fork1.c     list_files.c      pipe_test.py       shell2.c
fork4.cpp   multi_children.c  process_info.py    shell2.py
fork_ls.c   orphan_demo1.c    shell1.c           signals_demo1.c
cmd> pwd
/home/samyz/os/proj5
cmd> 
```

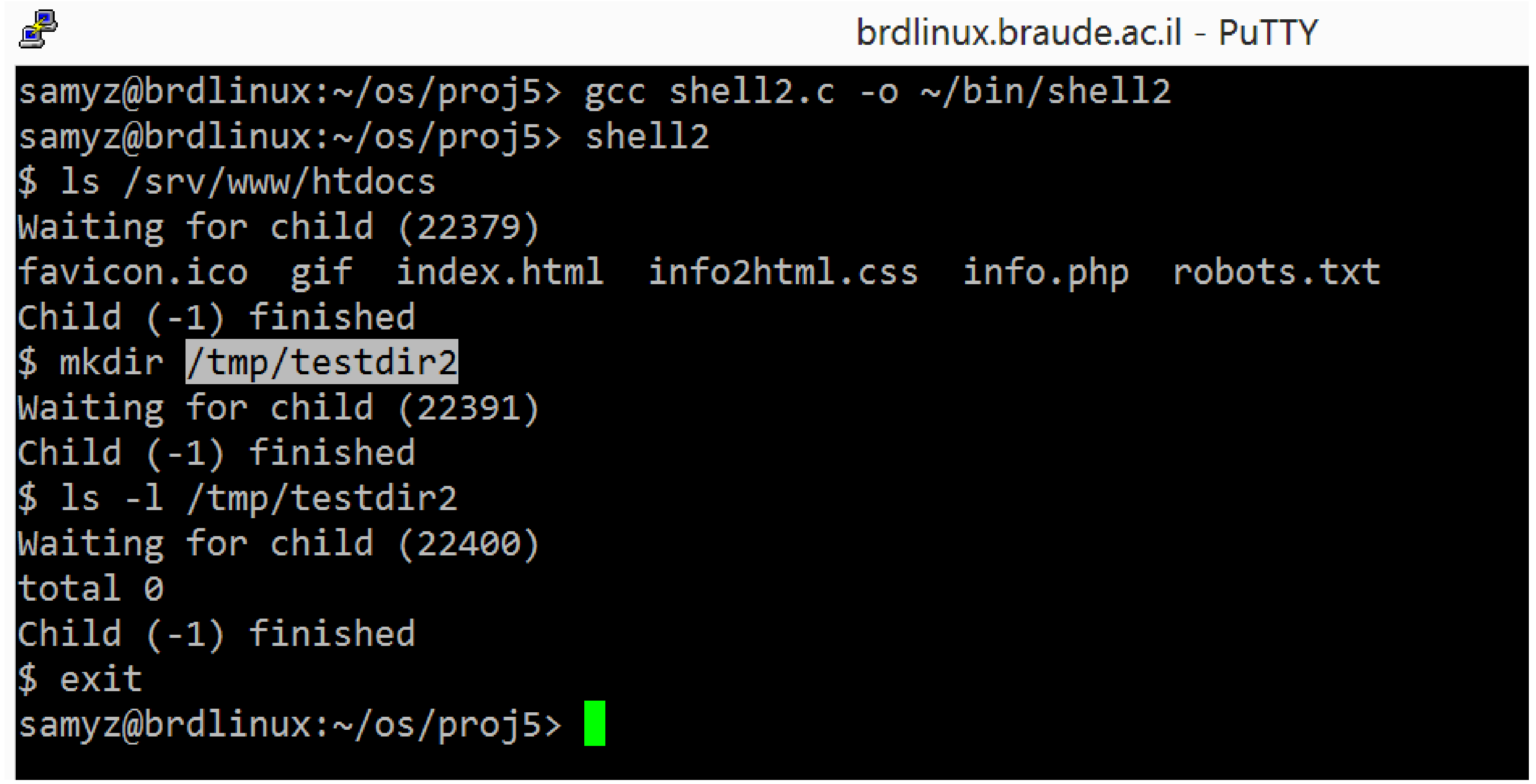
Simple Shell 2

Problem 14:

- Go back to **~/proj5** and read the program **shell2.c**
- Compile this program by running:

```
gcc shell2.c -o ~/bin/shell2
```

- Run this command (**shell2**) and feed it with Linux commands with arguments!
- This time it's not necessary to understand all the details in this program (complex)
- Instead try to run an equivalent Python program: **shell2.py** (see next slide)


brdlinux.braude.ac.il - PuTTY

```
samyz@brdlinux:~/os/proj5> gcc shell2.c -o ~/bin/shell2
samyz@brdlinux:~/os/proj5> shell2
$ ls /srv/www/htdocs
Waiting for child (22379)
favicon.ico  gif  index.html  info2html.css  info.php  robots.txt
Child (-1) finished
$ mkdir /tmp/testdir2
Waiting for child (22391)
Child (-1) finished
$ ls -l /tmp/testdir2
Waiting for child (22400)
total 0
Child (-1) finished
$ exit
samyz@brdlinux:~/os/proj5>
```

Simple Unix Shell in Python (shell2.py)

Problem 15:

- Since `shell2.c` is not easy to understand and modify, let's try to rewrite the same program in Python
- Here are the Python functions you need to use:
  - `cmdline = raw_input("cmd> ")`
  - `pid = os.fork()`
  - `os.waitpid(pid,0)`
  - `argv = cmdline.split()`
  - `os.execvp() or os.execlp() or subporcess.Popen`
- To check if your solution is correct, compare it with my solution:
  `~samyz/os/proj5.sol/shell2.py`

Parent with multiple children

Problem 16:

- Go back to `~/proj5` and read the program `multi_children.c`
- Compile this program by running:

```
gcc multi_children.c –o ~/bin/multi_children
```

- A parent process can create multiple children processes and wait for each one of them
- To get a deep understanding of the code you must read about the **wait** system call (and about the **WIFEXITED** and **WEXITSTATUS** macros)
  - Use the command "**man wait**" in your Linux terminal to find all the information you need!
  - You will also find info here: http://linuxmanpages.com/man2/wait.2.php
- Make sure you understand the difference between **exit_code** and **exit_status**
- **Write a Python version of this program !**
- **Note:** the original version of this program had several errors and it has been fixed several times since then! (make sure you have the updated: ~samyz/os/proj5/multi_children.c)

Unix Signals

<u>Problem 17:</u>

- Go back to **~/proj5** and read the program `signals_demo.c`
- Compile this program by running:

```
gcc signals_demo.c -o ~/bin/signals_demo
```

- A parent process can create multiple children processes and then can send a signal (kill) and wait for each one of them
- To get a deep understanding of the code you must read about the **kill** and **wait** system call (and about the **WIFEXITED** and **WEXITSTATUS** macros)
  - Use the command "**man wait**" and "**man kill**" in your Linux terminal to find all the information you need!
  - You will also find info here: http://linuxmanpages.com/man2/kill.2.php
- Try to explain the reverse order of process termination
- **Write a Python version of this program !**

Unix Pipe

Problem 18:

- Go back to `~/proj5` and read the program `pipe_demo.c`
- Compile this program by running:

```
gcc pipe_demo.c -o ~/bin/pipe_demo
```

- Read about Unix pipes from the following sources:
  http://www.cs.rutgers.edu/~pxk/416/notes/c-tutorials/pipe.html
  http://linuxprograms.wordpress.com/category/pipes
- For more advanced pipe examples look at the programs:

```
pipe_ls_wc1.c
pipe_ls_wc2.c
```

- If we have time we may take a look at them and try to understand them, but we'll probably prefer the Python API which will be easier to master and use in later projects

Python Interface to the pipe system call

Problem 19:

- Go back to `~/proj5`
- The Python os module has a simple interface `os.pipe` to the Linux pipe system call, and os.fork() for the fork system call
- Read the files: `pipe_test.py`, `pipe_fork_test.py` for very basic usage examples
- After understanding these examples, try to write an equivalent Python program `pipe_ls_wc1.py` for `pipe_ls_wc1.c`

Back to Python subprocess module

Problem 20:

- Go back to `~/proj5`
- Read the file `subprocess_ls.py` and explain the code there
- Make it executable by running the following two commands:

```
cp subprocess_ls.py ~/bin/subprocess_ls
chmod +x ~/bin/subprocess_ls
```

- Write a similar Python program to calculate the total size of programs (in `/usr/bin`) that were installed before April 1, 2013?
  Hint: look up the `--time-styl` option of the command `ls`

Back to Python subprocess module

Problem 21:

- Go back to `~/proj5`
- Read the Python program `process_info.py` and make sure you understand all the lines there
- Write a similar Python program to calculate the total size of programs (in `/usr/bin`) that were installed before April 1, 2013?
  Hint: look up the `--time-styl` option of the command `ls`

More on Python subprocess module

Problem 22:

- Run the command: `ping www.cyberciti.biz`
  In your Linux (or DOS) terminal and get acquainted with its output form
- The `ping` system program is used by network administrators for debugging and checking connectivity speeds
- Here is a simple Python interface to ping thru the subprocess module:
  `subprocess_ping.py`
- Read this file and write a new file ping_speed.py which usee the Linux ping command for calculating the average reply time to a given internet address:

```
ping_speed www.cyberciti.biz
=> 175 milliseconds
```