Operating Systems Week 8

Linux

Virtual machines

- Run a **guest os** on a **host os**

VM Security

- Better than nothing

Processes

- Each has, *inter alia*:
  - An owner (from last lecture: RUID, EUID, …)
  - A parent process
  - A scheduler priority (‘niceness’)
- What happens when a process starts another?

A shell (e.g. bash)

- What people think of as ‘a terminal’, or ‘the command line’
- Important notion: the **present working directory,** changed using **cd**.
- Most commands are executables on disk
  - e.g. /bin/ls; /bin/grep; /bin/more …
  - A command starts a process, with some **arguments (**`String[] args`**)**
  - Process **returns an exit code**: 0 for success, non-zero otherwise. In Java - System.exit(…).

UNIX Philosophy

Courtesy of Mike Gancarz:

1. Small is beautiful.
2. Make each program do one thing well.
3. ...

Con**CAT**enate

- Loop over a list of files
  - Open;
  - Print out contents;
  - Close

grep (**re** = regular expression)

- Take a regular expression
- Read some input
- Then:
  - By default, only print out lines that **match that regular expression**
  - `grep -v`: print out lines that do not match it
  - `grep -l`: print out filenames with a match
  - `grep -c`: count matching lines. `grep -cv`?

wc – word count

- Loop over a list of files
  - Open
  - Count how many words/characters/lines
  - Close
  - Print out data
- Print total line across all files

head, tail

- `head` – get the first lines:
  - Read 10 lines, print them out, stop
  - `head -lines=20` : read 20 lines then stop
- `tail` – get the last lines:
  - Read all the lines; once input has ended, print the last 10 out.
  - `tail -lines=20` …

Running commands in sequence

- Semicolons ~= newlines

```
cat a.txt; wc b.txt; head c.txt
```

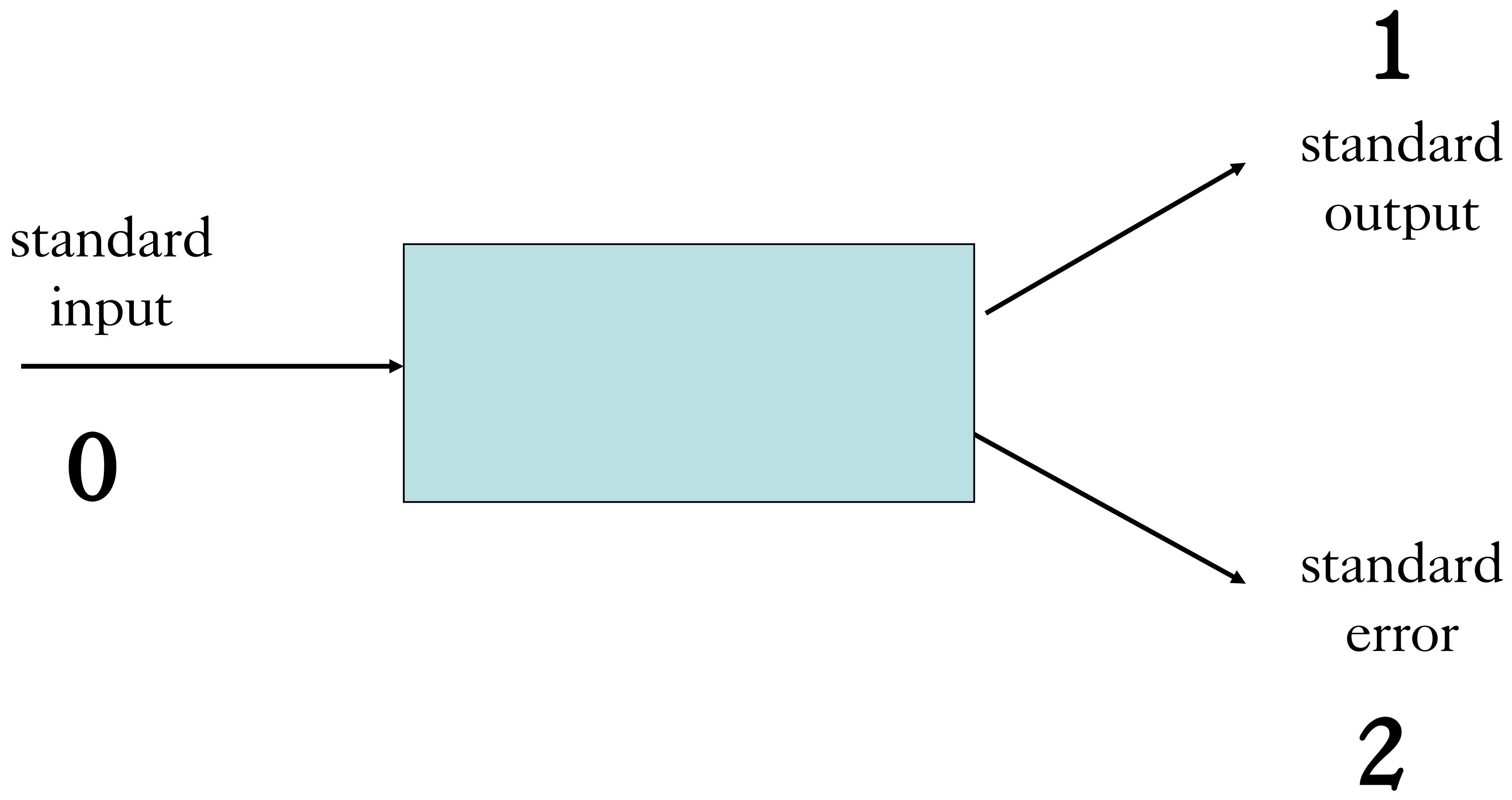
- Does not check the exit code was zero is at each stage – can be problematic e.g:

```
cd some_directory; rm a.txt
```

- Instead use `&&`:

```
cd some_directory && rm a.txt
```

Processes



System calls

Just like reading/writing to/from files:

```
write(1, “Hello World\n”, 12);
write(2, “Uh-oh”, 5);

char buf[256];
int bytesRead = read(0,buf,256);
```

I can haz Java?

Standard input = System.in

Standard output = System.out

Standard error = System.err

Reading from System.in

- System.in is an **`InputStream`**

```
byte[] buf = new byte[256];
int bytesRead =
  System.in.read(buf, 0, 256);
```

Leads to a system call

Unicode

- In Java: `chars` are 2 bytes each
  - There are more than 256 written characters in use in the world (no șit…)
- Unicode (in UTF-8 encoding) in a nutshell:
  - Read a byte. Easy case: the top-most bit is 0, turn into a char (following ASCII).
  - If the ***n*** top-most bits are 1, read another ***n*** **- 1** bytes, for ***n*** bytes total; combine into a char.
  - ***n*** can be up to 4.

Reading chars from System.in

- This is exactly what an `InputStreamReader` is for:

```
InputStreamReader charsIn = new
  InputStreamReader(System.in);
char[] buf = new char[256];
int charsRead = charsIn.read(buf,
  0, 256);
```

Back to system calls

- What if we write:

```
for (int i = 0; i < 256; ++i) {
   int x = charsIn.read();
   ...do something with x...
}
```

BufferedReader

```
BufferedReader b                          = new
 BufferedReader(new

        InputStreamReader(System.in));

for (int i = 0; i < 256; ++i) {
   int x = b.read();
   ...do something with x...
}
```

Java OutputStream

```
OutputStream o = …;
o.write(someBytes, 0, 256);
```

Java BufferedOutputStream

Same as an OutputStream but with buffering:

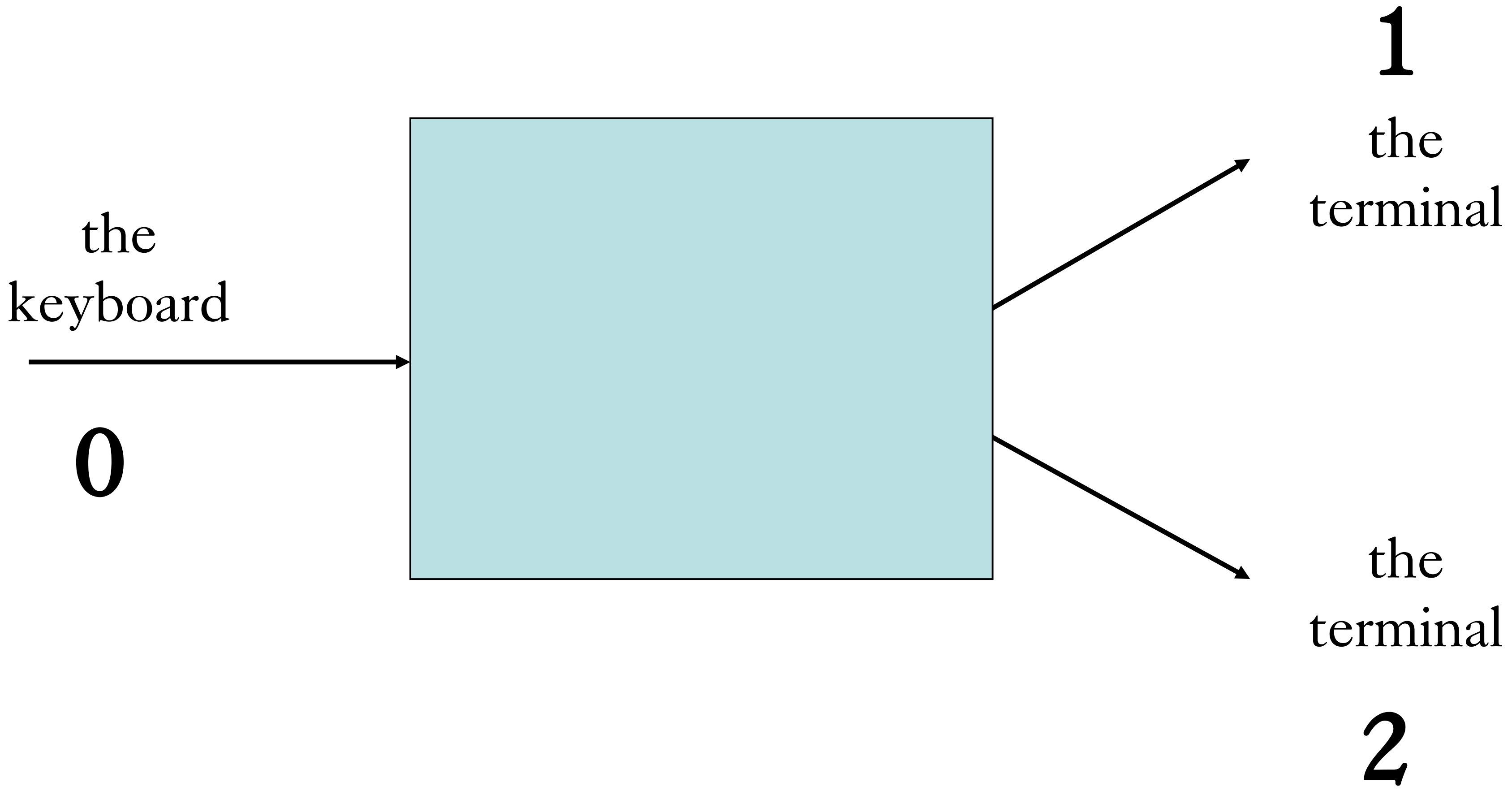
```
OutputStream o = …;
BufferedOutputStream b = new
  BufferedOutputStream(o);
for (int i = 0; i < 256; ++i) {
    b.write(...some byte...);
}
```

System.out

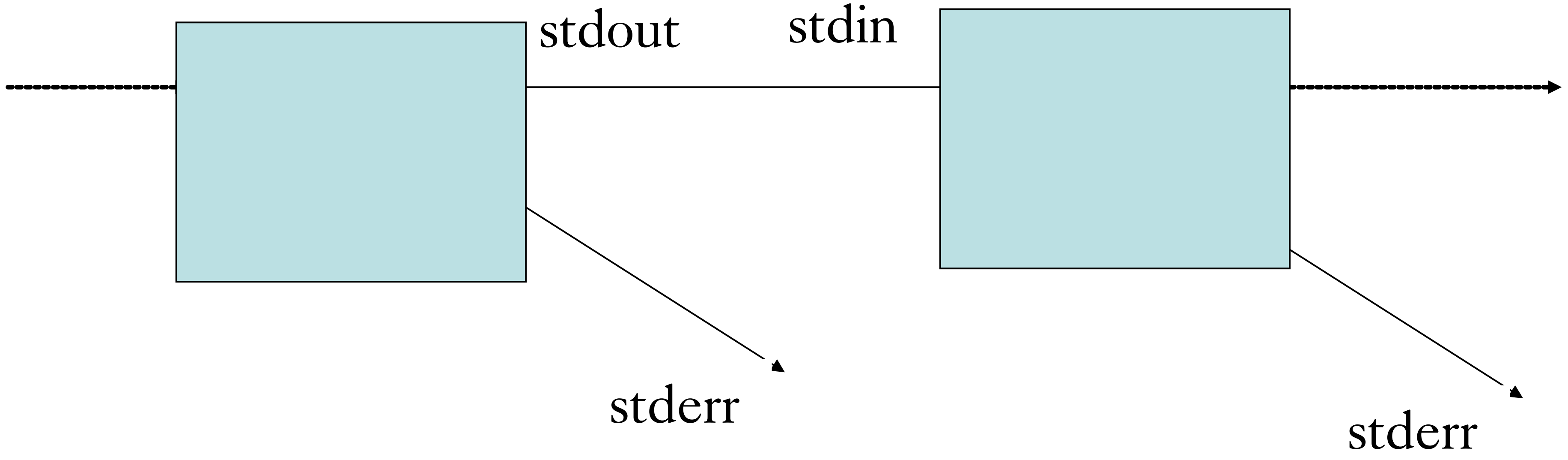
- …is a **PrintStream** – a BufferedOutputStream that also does char -> byte conversion

```
for (int i = 0; i < 256; ++i) {
  System.out.print(…some char…);
}
```

Processes in a Terminal



Pipes



Example pipes

- How many lines in a file contain the word dave?

```
cat file.txt | grep 'dave' | wc -l
```

- Files whose names match the regexp ‘steve*’, displaying results one page at a time:

```
ls | grep ‘steve*' | more
```

Inside the kernel

- Pipes = **producer-consumer**
  - **`ls`** is **producing** bytes into a buffer, every time it calls `write(1, …);`
  - **`grep`** is **consuming** bytes from a buffer, every time it calls `read(0, …);`
- Overkill solution: make the entire interrupt handler a **critical section**
  - In practice?

/proc/....

- /proc is a **virtual filesystem** (see also /dev)
- /proc/12345 contains information about the process 12345. **Who owns /proc/12345**?
- /proc/12345/environ: environment variables

- /proc/12345/fd lists all the **file descriptors**
  - Using `ls -l /proc/12345/fd` can see what 0,1,2 are connected to

/dev: Everything is a file

- /dev contains **hardware devices**
  - /dev/sda, /dev/sdb, /dev/sdc... - disks
  - /dev/snd/... - audio devices
  - /dev/video0 – a web cam
  - ...
- Opened/read from/written to, like files
- **Access restrictions:** ordinary users cannot read/write disks directly; groups for some devices (e.g. 'video' for video devices); ...

Redirection

- Produce a file containing all the lines from three input files, that match 'A[a-z]*A':

```
cat 1.txt 2.txt 3.txt |
  grep 'A[a-z]*A' > matched.txt
```

NB Do not confuse > and |

Redirection to /dev/...

Write a disk image:

```
cat disk.img > /dev/somedisk
```

Run a command, throw away output:

```
command > /dev/null
```

Send random input to a command:

```
cat /dev/random | somecommand
```

Buffering and redirection

```
some-command > output.log
```

- Q: How often is output.log updated?
- A: When the buffer fills, or flush() is called.

- Can we turn-off the buffer?

`stdbuf -o 0 <command>` : runs <command> with 0-byte output buffer

`stdbuf -oL  <command>` : flush at newlines

stdout, stderr

- By default, stdout is redirected, not stderr
- Can use 2 to redirect stderr:

```
cat a.txt b.txt > joined.txt 2> error.log
cat a.txt b.txt 2> error.log | wc -l
```

- Can also merge stderr with stdout:

```
cat a.txt 2>&1
```

More commands: sed

- sed: **s**tream **ed**itor
- `sed -e 's,cat,dog,'`
  - Replace cat with dog, once
- `sed -e 's,cat,dog,g'`
  - Replace cat with dog, **globally**

- `sed -e 's,\([a-z]+\)[0-9]*,\1,'`

find

- Finds files with certain properties, recursively

- `find -name "foo"`: find files named foo
- `find -iname "*.jpg"`: find jpg files, case **i**nsensitive (will match .JPG, .jpg...)
- `find -type d`: find all directories (`-type f` for all files)

find ... -exec

- -exec: run a command on each file found

```
find -iname "*.txt" -exec cat {} \;
```

placeholder for the filename

end of the command to run on the file

Putting it together

- A directory tree contains CSV files of exam results – one CSV per module, giving the mark for each candidate.
- The anonymous marking code T01234 is for the student Geoff Vader.
- Make a file results.csv, containing all results for T01234; and replacing the string "T01234" with the string "Geoff Vader"

ps: process status

- ps: shows processes running in the current terminal
- ps ux: show all the current **u**ser's processes, in e**x**tended detail
- ps aux: same but for **a**ll users
- ps auxf: ASCII-art process tree

- What can you see on the right on ps aux?

Security?

```
somecommand
   --username=derp
   --password=my_super_secret_password
```

Process signals: kill

- `kill 28125`: terminate process 28125
  - Sends `SIGTERM` – can optionally be caught, and terminate 'nicely' before closing
- `kill -SIGKILL 28125`: sends `SIGKILL`, killing it immediately without question
- `kill -SIGSTOP 28125`: pause 28125. (It is never chosen by the scheduler.)
- `kill -SIGCONT 28125`: resume 28125

Hangup

- The **hangup** signal is sent when the **terminal is disconnected** (e.g. window is closed)
- Or, `kill -SIGHUP 28125`

- Usually the same as kill -SIGTERM
- Or can ignore it using `nohup`:

```
nohup somecommand
```

Returning to the shell

- `cat a.txt b.txt c.txt`: cats three files, then returns to the shell prompt ready for the next command.
- `cat a.txt b.txt c.txt &` will return to the shell immediately.

```
nohup web-server >& output.log &
```

Priority

- Each process has a niceness. Default = 0
  - 20 = only gets 'idle' CPU cycles
  - -20 = highest possible priority

`/usr/bin/nice <command>`: runs <command> with nice 10

`/usr/bin/nice -n 20 <command>`: runs <command> with nice 20

`renice -n 20 28125`: set PID 28125 to nice 20

The rules of nice

- Processes start with nice 0
- **root** can give processes any nice value, from -20 to 20
- **You** can decrease **your** processes' priorities: bigger nice values
- You **cannot increase your processes' priorities** as that might undermine root's authority

Environment variables

- A map from strings to values
- `PWD`: the **present working directory**

```
echo $PWD
```

- `PATH`: where to look for commands
  - e.g. `/usr/sbin:/usr/bin:/sbin:/bin`
- `HOME`: path to the home directory
- `LANGUAGE`: current language
- …

cd

- A shell **built-in**: changes the present working directory. Also:
  - `pushd <dir>`: pushes PWD onto a stack; then does `cd <dir>`
  - `popd`: pop PWD off a stack
- Why is `cd` built into the shell? Could `cd` be an executable, e.g. saved as `/bin/cd`?

Scripting

- Writing magic sequences of commands, piped together, is tedious – write a script
- Simple case: just write the commands
- The shell also has loops, variables, …
- Could use a scripting language, e.g. perl

- If possible, write your scripts as **filters**: read from stdin, write to stdout