Processes

ComS 252 — Iowa State University

Andrew Miner

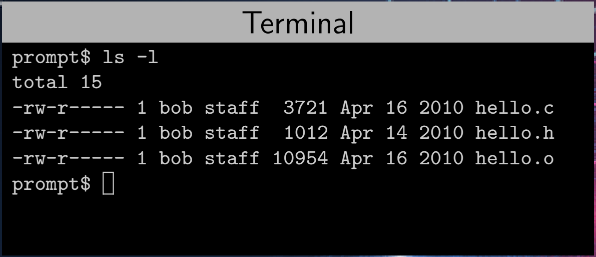
Random useful commands

`exit`: exit a shell

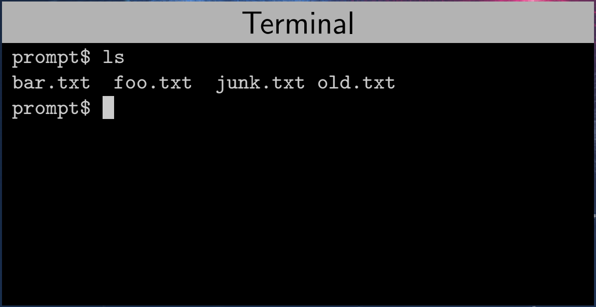
- If this is a login shell — you are logged out
- If this is a terminal window — window (normally) closes

`date`: display date and time

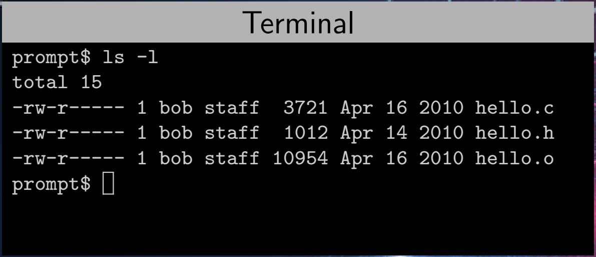
- Can change the format
- Read the `man` page for details



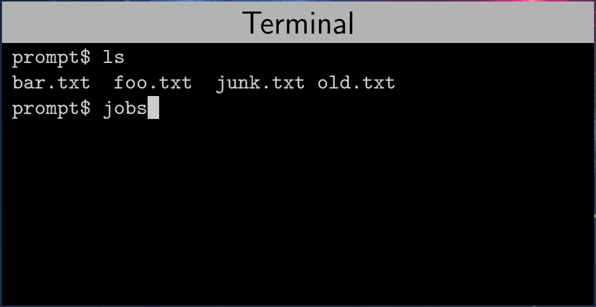
```
prompt$ ls -l
total 15
-rw-r----- 1 bob staff  3721 Apr 16 2010 hello.c
-rw-r----- 1 bob staff  1012 Apr 14 2010 hello.h
-rw-r----- 1 bob staff 10954 Apr 16 2010 hello.o
prompt$
```



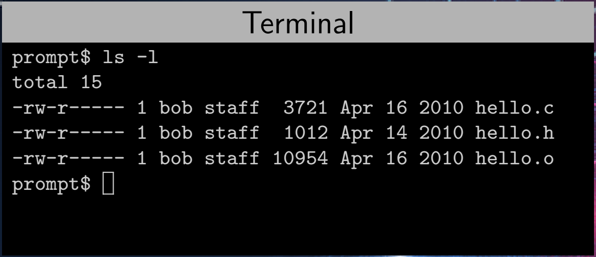
```
prompt$ ls
bar.txt  foo.txt  junk.txt old.txt
prompt$
```



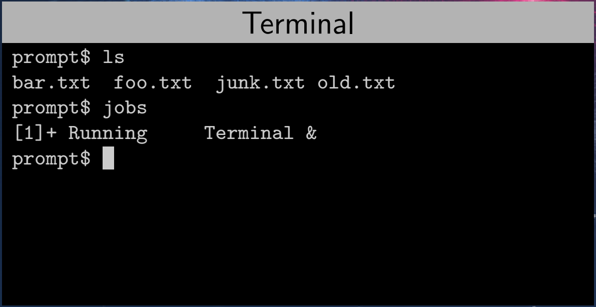
```
prompt$ ls -l
total 15
-rw-r----- 1 bob staff  3721 Apr 16 2010 hello.c
-rw-r----- 1 bob staff  1012 Apr 14 2010 hello.h
-rw-r----- 1 bob staff 10954 Apr 16 2010 hello.o
prompt$
```



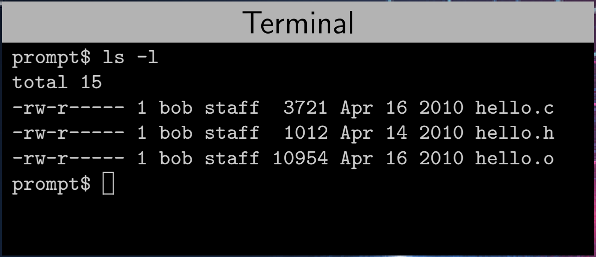
```
prompt$ ls
bar.txt  foo.txt  junk.txt old.txt
prompt$ jobs
```



```
prompt$ ls -l
total 15
-rw-r----- 1 bob staff  3721 Apr 16 2010 hello.c
-rw-r----- 1 bob staff  1012 Apr 14 2010 hello.h
-rw-r----- 1 bob staff 10954 Apr 16 2010 hello.o
prompt$ 
```

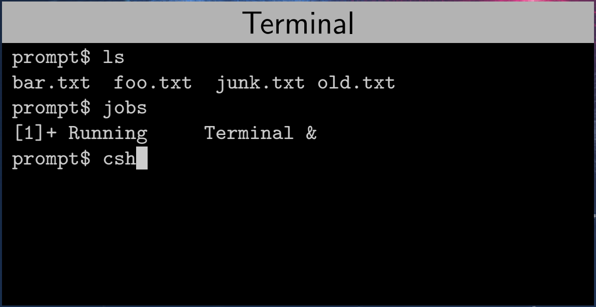


```
prompt$ ls
bar.txt  foo.txt  junk.txt old.txt
prompt$ jobs
[1]+ Running     Terminal &
prompt$ 
```



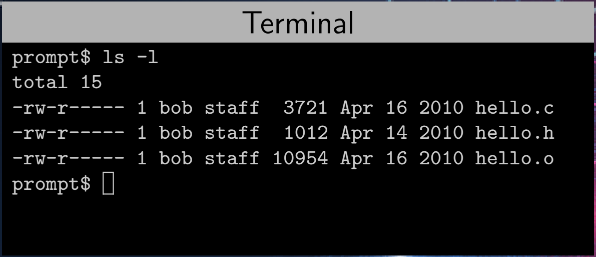
Terminal

```
prompt$ ls -l
total 15
-rw-r----- 1 bob staff  3721 Apr 16 2010 hello.c
-rw-r----- 1 bob staff  1012 Apr 14 2010 hello.h
-rw-r----- 1 bob staff 10954 Apr 16 2010 hello.o
prompt$
```

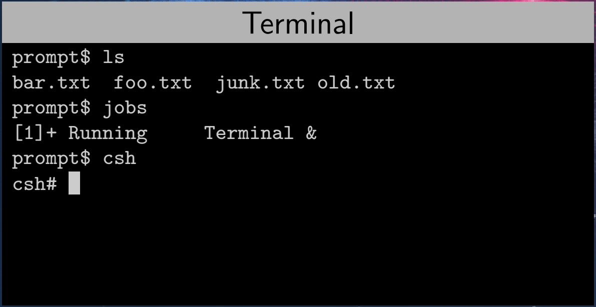


Terminal

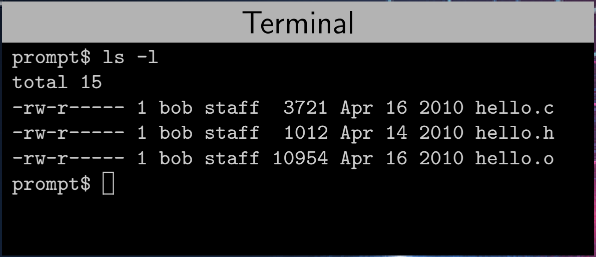
```
prompt$ ls
bar.txt  foo.txt  junk.txt old.txt
prompt$ jobs
[1]+ Running      Terminal &
prompt$ csh
```



```
prompt$ ls -l
total 15
-rw-r----- 1 bob staff  3721 Apr 16 2010 hello.c
-rw-r----- 1 bob staff  1012 Apr 14 2010 hello.h
-rw-r----- 1 bob staff 10954 Apr 16 2010 hello.o
prompt$ 
```

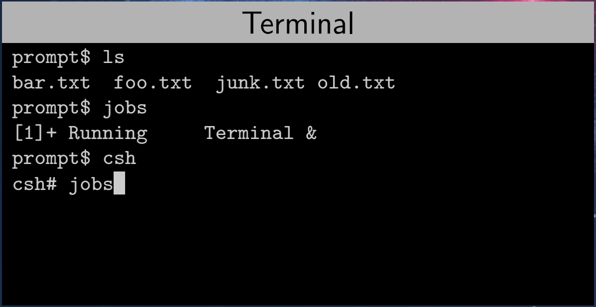


```
prompt$ ls
bar.txt  foo.txt  junk.txt old.txt
prompt$ jobs
[1]+ Running     Terminal &
prompt$ csh
csh# 
```



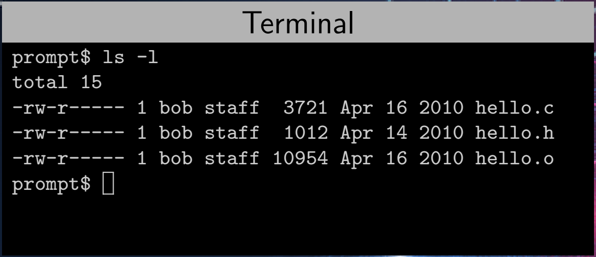
Terminal

```
prompt$ ls -l
total 15
-rw-r----- 1 bob staff  3721 Apr 16 2010 hello.c
-rw-r----- 1 bob staff  1012 Apr 14 2010 hello.h
-rw-r----- 1 bob staff 10954 Apr 16 2010 hello.o
prompt$ 
```

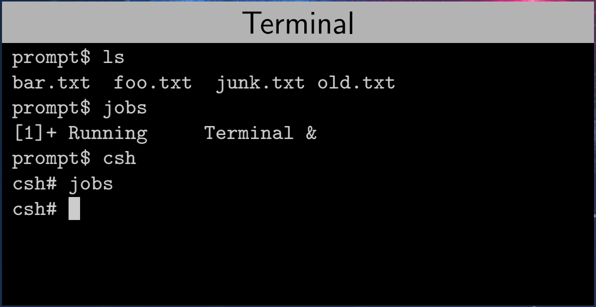


Terminal

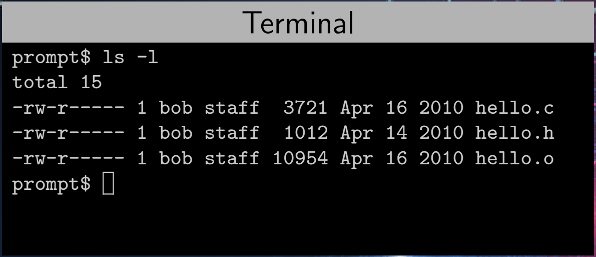
```
prompt$ ls
bar.txt  foo.txt  junk.txt old.txt
prompt$ jobs
[1]+ Running      Terminal &
prompt$ csh
csh# jobs
```



```
prompt$ ls -l
total 15
-rw-r----- 1 bob staff  3721 Apr 16 2010 hello.c
-rw-r----- 1 bob staff  1012 Apr 14 2010 hello.h
-rw-r----- 1 bob staff 10954 Apr 16 2010 hello.o
prompt$ 
```

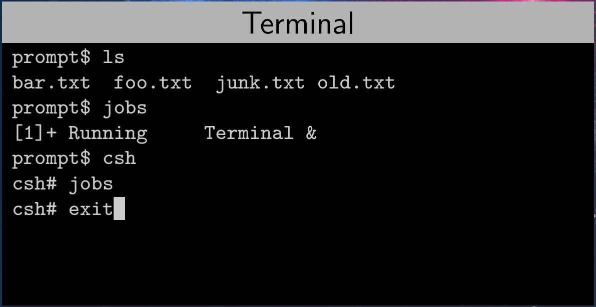


```
prompt$ ls
bar.txt  foo.txt  junk.txt old.txt
prompt$ jobs
[1]+ Running     Terminal &
prompt$ csh
csh# jobs
csh# 
```



Terminal

```
prompt$ ls -l
total 15
-rw-r----- 1 bob staff  3721 Apr 16 2010 hello.c
-rw-r----- 1 bob staff  1012 Apr 14 2010 hello.h
-rw-r----- 1 bob staff 10954 Apr 16 2010 hello.o
prompt$
```



Terminal

```
prompt$ ls
bar.txt  foo.txt  junk.txt old.txt
prompt$ jobs
[1]+ Running     Terminal &
prompt$ csh
csh# jobs
csh# exit
```

Terminal

```
prompt$ ls -l
total 15
-rw-r----- 1 bob staff  3721 Apr 16 2010 hello.c
-rw-r----- 1 bob staff  1012 Apr 14 2010 hello.h
-rw-r----- 1 bob staff 10954 Apr 16 2010 hello.o
prompt$ ▯
```

Terminal

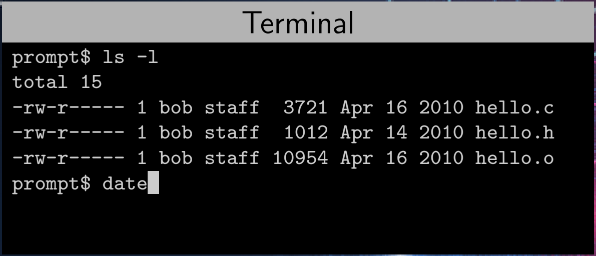
```
prompt$ ls
bar.txt  foo.txt  junk.txt old.txt
prompt$ jobs
[1]+ Running     Terminal &
prompt$ csh
csh# jobs
csh# exit
prompt$ █
```

Terminal

```
prompt$ ls -l
total 15
-rw-r----- 1 bob staff  3721 Apr 16 2010 hello.c
-rw-r----- 1 bob staff  1012 Apr 14 2010 hello.h
-rw-r----- 1 bob staff 10954 Apr 16 2010 hello.o
prompt$
```

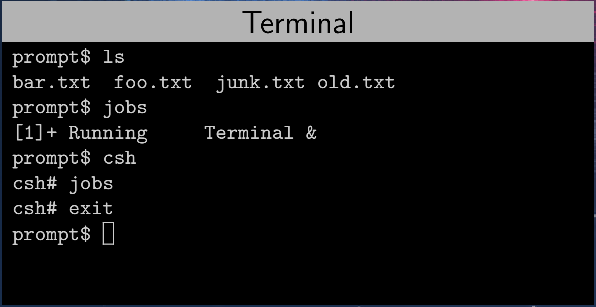
Terminal

```
prompt$ ls
bar.txt  foo.txt  junk.txt old.txt
prompt$ jobs
[1]+ Running     Terminal &
prompt$ csh
csh# jobs
csh# exit
prompt$
```



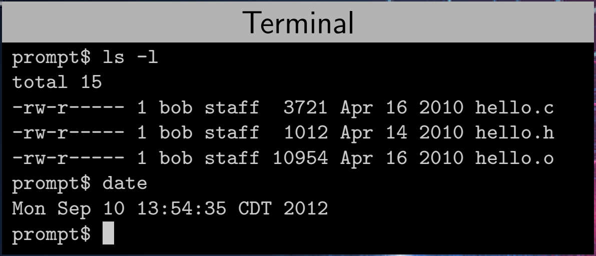
Terminal

```
prompt$ ls -l
total 15
-rw-r----- 1 bob staff  3721 Apr 16 2010 hello.c
-rw-r----- 1 bob staff  1012 Apr 14 2010 hello.h
-rw-r----- 1 bob staff 10954 Apr 16 2010 hello.o
prompt$ date
```



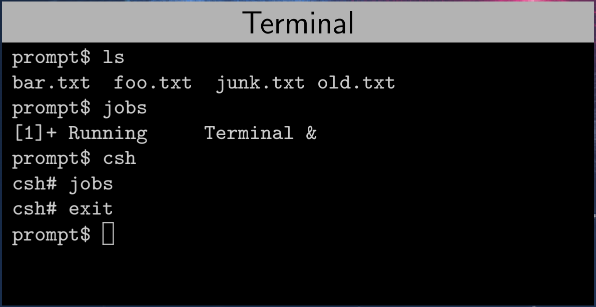
Terminal

```
prompt$ ls
bar.txt  foo.txt  junk.txt old.txt
prompt$ jobs
[1]+ Running     Terminal &
prompt$ csh
csh# jobs
csh# exit
prompt$ 
```



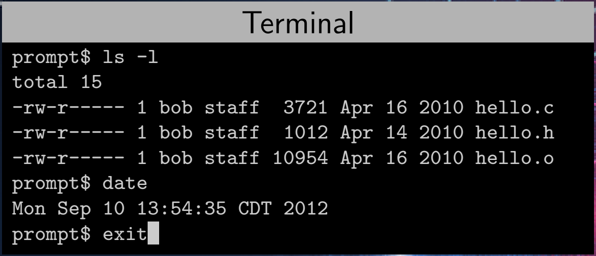
Terminal

```
prompt$ ls -l
total 15
-rw-r----- 1 bob staff  3721 Apr 16 2010 hello.c
-rw-r----- 1 bob staff  1012 Apr 14 2010 hello.h
-rw-r----- 1 bob staff 10954 Apr 16 2010 hello.o
prompt$ date
Mon Sep 10 13:54:35 CDT 2012
prompt$ █
```

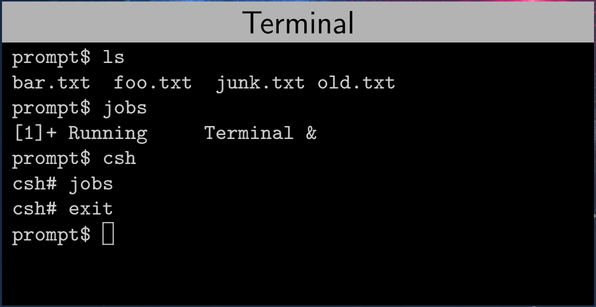


Terminal

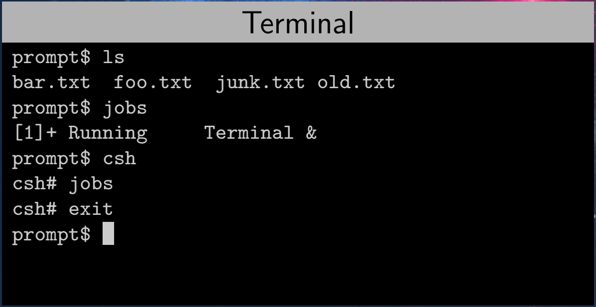
```
prompt$ ls
bar.txt  foo.txt  junk.txt old.txt
prompt$ jobs
[1]+ Running     Terminal &
prompt$ csh
csh# jobs
csh# exit
prompt$ □
```



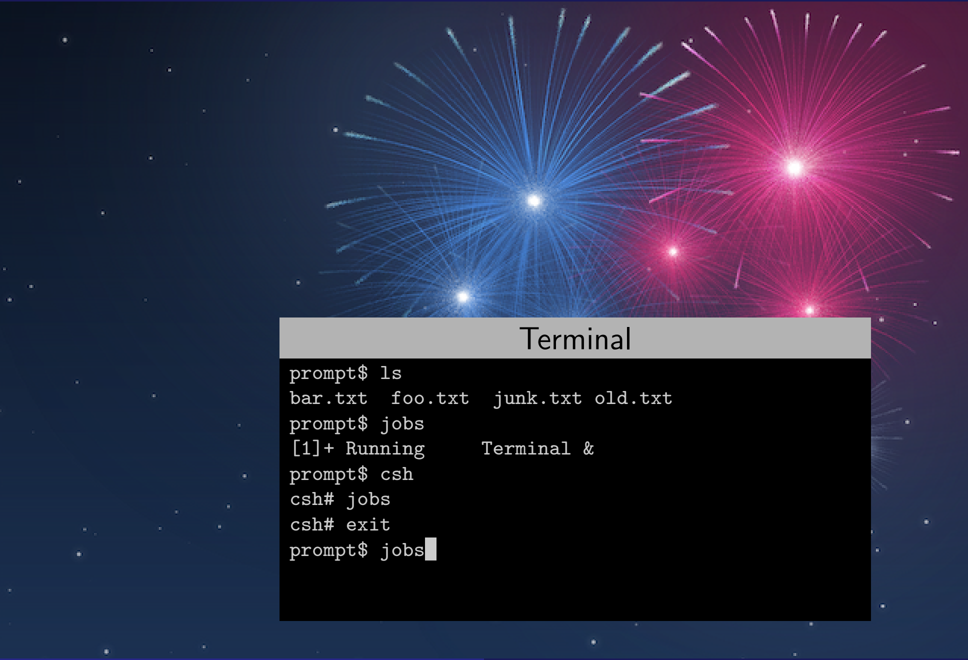
```
prompt$ ls -l
total 15
-rw-r----- 1 bob staff  3721 Apr 16 2010 hello.c
-rw-r----- 1 bob staff  1012 Apr 14 2010 hello.h
-rw-r----- 1 bob staff 10954 Apr 16 2010 hello.o
prompt$ date
Mon Sep 10 13:54:35 CDT 2012
prompt$ exit
```



```
prompt$ ls
bar.txt  foo.txt  junk.txt old.txt
prompt$ jobs
[1]+ Running      Terminal &
prompt$ csh
csh# jobs
csh# exit
prompt$
```

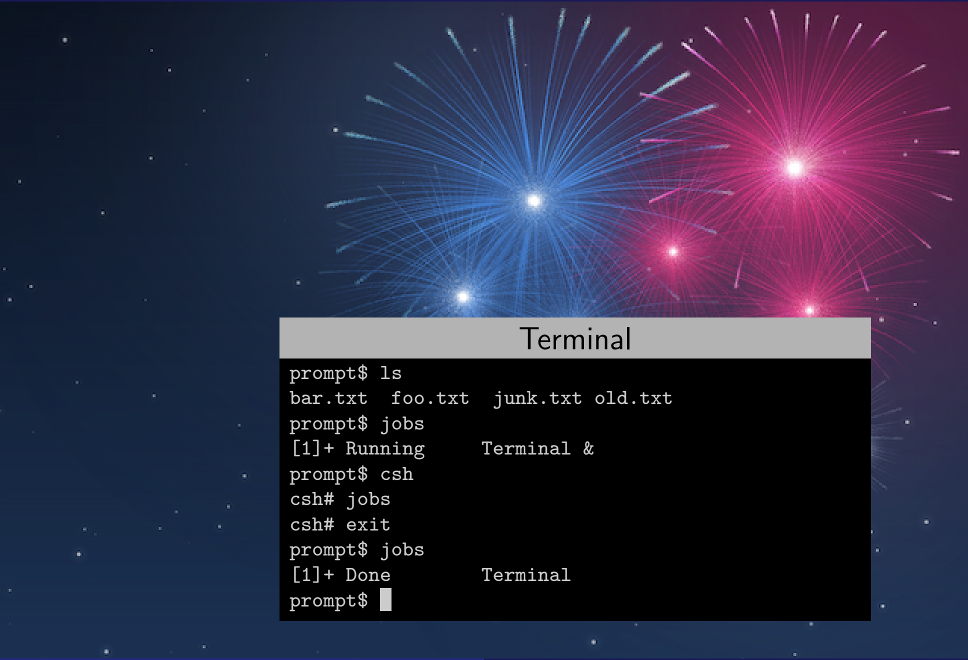


```
prompt$ ls
bar.txt  foo.txt  junk.txt old.txt
prompt$ jobs
[1]+ Running     Terminal &
prompt$ csh
csh# jobs
csh# exit
prompt$ █
```



Terminal

```
prompt$ ls
bar.txt  foo.txt  junk.txt old.txt
prompt$ jobs
[1]+ Running     Terminal &
prompt$ csh
csh# jobs
csh# exit
prompt$ jobs
```



Terminal

```
prompt$ ls
bar.txt  foo.txt  junk.txt old.txt
prompt$ jobs
[1]+ Running      Terminal &
prompt$ csh
csh# jobs
csh# exit
prompt$ jobs
[1]+ Done         Terminal
prompt$ █
```

Motivating questions

1. How does a job know where and how to display its output?
   - E.g., output of “ls” goes to which terminal window?
2. When I run multiple jobs, why don’t they clobber each other?
3. What happens to a job if its shell terminates?
4. How can I control a job from another shell?
5. Can I tell if a job finished successfully?

What is a process?

A process is a running program

- Like a job, but at the *kernel level*
- Has a unique identifier (Process ID or `pid`)
- Has an owner
  - The user running the process
  - Determines permissions
- Has its own memory space
  - Cannot access outside of this — get segmentation fault
- Performs its own I/O
  - Has its own working directory
  - Has its own list of open files
  - *Process owner* + *file permissions* dictate ability to open files
- Can receive signals (see next slide)
- Returns an exit status upon termination

Signals to processes

- Signals may originate from the kernel itself, or another process
- Some errors are handled by signals:

  `FPE`: Floating–point exception (e.g., divide by zero)
  `SEGV`: Segmentation violation
- A process may set its own signal handler by signal type
  - Except for `STOP` and `KILL` signals
- Otherwise a process uses the default signal handler
  - Default is either to ignore, or to terminate the process based on what makes sense for the signal

More fun with processes

Any process can create another process

- Not just a shell
- `fork()` system call in C
- Processes have parent / child relationships

There are lots of processes in memory at any given time

- At most one process can be running at a time, per CPU
- Kernel switches between processes
  - Each process runs for a short burst
  - Gives illusion that they are running "simultaneously"
- Process scheduler decides who goes next
  - Does not select a process that is waiting for I/O
  - Based on process's priority level

Example: life of a process

What happens when I type "`ls`" in a shell?

1. Shell creates a process (using `fork`)
2. Shell loads `ls` executable into the process (using `execve`)
3. Kernel selects `ls` process for execution
4. `ls` process gets to run for a while
5. Kernel switches to another process
6. Repeat (3)...(5) a few times
7. `ls` finishes (returns from `main()` or calls `exit()`)
8. Kernel marks process as "done executing"
   - Fun fact: process is now a zombie
9. Parent process (the shell) cleans up child process
   - This way, the parent can get the process's exit status
10. Kernel frees resources used by the process

Example: life of a process

What happens when I type "`ls`" in a shell?

1. Shell creates a process (using `fork`)
2. Shell loads `ls` executable into the process (using `execve`)
3. Kernel selects `ls` process for execution
4. `ls` process gets to run for a while
5. Kernel switches to another process
6. Repeat (3)…(5) a few times
7. `ls` finishes (returns from `main()` or calls `exit()`)
8. Kernel marks process as "done executing"
    - Fun fact: process is now a zombie
9. Parent process (the shell) cleans up child process
    - This way, the parent can get the process's exit status
10. Kernel frees resources used by the process

Example: life of a process

What happens when I type "`ls`" in a shell?

1. Shell creates a process (using `fork`)
2. Shell loads `ls` executable into the process (using `execve`)
3. Kernel selects `ls` process for execution
4. `ls` process gets to run for a while
5. Kernel switches to another process
6. Repeat (3)…(5) a few times
7. `ls` finishes (returns from `main()` or calls `exit()`)
8. Kernel marks process as "done executing"
   - Fun fact: process is now a zombie
9. Parent process (the shell) cleans up child process
   - This way, the parent can get the process's exit status
10. Kernel frees resources used by the process

Example: life of a process

What happens when I type “`ls`” in a shell?

1. Shell creates a process (using `fork`)
2. Shell loads `ls` executable into the process (using `execve`)
3. Kernel selects `ls` process for execution
4. `ls` process gets to run for a while
5. Kernel switches to another process
6. Repeat (3). . .(5) a few times
7. `ls` finishes (returns from `main()` or calls `exit()`)
8. Kernel marks process as “done executing”
   - Fun fact: process is now a zombie
9. Parent process (the shell) cleans up child process
   - This way, the parent can get the process’s exit status
10. Kernel frees resources used by the process

Example: life of a process

What happens when I type "`ls`" in a shell?

1. Shell creates a process (using `fork`)
2. Shell loads `ls` executable into the process (using `execve`)
3. Kernel selects `ls` process for execution
4. `ls` process gets to run for a while
5. Kernel switches to another process
6. Repeat (3)…(5) a few times
7. `ls` finishes (returns from `main()` or calls `exit()`)
8. Kernel marks process as "done executing"
    - Fun fact: process is now a zombie
9. Parent process (the shell) cleans up child process
    - This way, the parent can get the process's exit status
10. Kernel frees resources used by the process

Example: life of a process

What happens when I type "`ls`" in a shell?

1. Shell creates a process (using `fork`)
2. Shell loads `ls` executable into the process (using `execve`)
3. Kernel selects `ls` process for execution
4. `ls` process gets to run for a while
5. Kernel switches to another process
6. Repeat (3)…(5) a few times
7. `ls` finishes (returns from `main()` or calls `exit()`)
8. Kernel marks process as "done executing"
   - Fun fact: process is now a zombie
9. Parent process (the shell) cleans up child process
   - This way, the parent can get the process's exit status
10. Kernel frees resources used by the process

Example: life of a process

What happens when I type "`ls`" in a shell?

1. Shell creates a process (using `fork`)
2. Shell loads `ls` executable into the process (using `execve`)
3. Kernel selects `ls` process for execution
4. `ls` process gets to run for a while
5. Kernel switches to another process
6. Repeat (3)...(5) a few times
7. `ls` finishes (returns from `main()` or calls `exit()`)
8. Kernel marks process as "done executing"
   - Fun fact: process is now a zombie
9. Parent process (the shell) cleans up child process
   - This way, the parent can get the process's exit status
10. Kernel frees resources used by the process

Example: life of a process

What happens when I type “`ls`” in a shell?

1. Shell creates a process (using `fork`)
2. Shell loads `ls` executable into the process (using `execve`)
3. Kernel selects `ls` process for execution
4. `ls` process gets to run for a while
5. Kernel switches to another process
6. Repeat (3). . . (5) a few times
7. `ls` finishes (returns from `main()` or calls `exit()`)
8. Kernel marks process as “done executing”
    - Fun fact: process is now a zombie
9. Parent process (the shell) cleans up child process
    - This way, the parent can get the process’s exit status
10. Kernel frees resources used by the process

Example: life of a process

What happens when I type "`ls`" in a shell?

1. Shell creates a process (using `fork`)
2. Shell loads `ls` executable into the process (using `execve`)
3. Kernel selects `ls` process for execution
4. `ls` process gets to run for a while
5. Kernel switches to another process
6. Repeat (3)…(5) a few times
7. `ls` finishes (returns from `main()` or calls `exit()`)
8. Kernel marks process as "done executing"
    - Fun fact: process is now a zombie
9. Parent process (the shell) cleans up child process
    - This way, the parent can get the process's exit status
10. Kernel frees resources used by the process

Example: life of a process

What happens when I type “`ls`” in a shell?

1. Shell creates a process (using `fork`)
2. Shell loads `ls` executable into the process (using `execve`)
3. Kernel selects `ls` process for execution
4. `ls` process gets to run for a while
5. Kernel switches to another process
6. Repeat (3)...(5) a few times
7. `ls` finishes (returns from `main()` or calls `exit()`)
8. Kernel marks process as “done executing”
   - Fun fact: process is now a zombie
9. Parent process (the shell) cleans up child process
   - This way, the parent can get the process's exit status
10. Kernel frees resources used by the process

Example: life of a process

What happens when I type "`ls`" in a shell?

1. Shell creates a process (using `fork`)
2. Shell loads `ls` executable into the process (using `execve`)
3. Kernel selects `ls` process for execution
4. `ls` process gets to run for a while
5. Kernel switches to another process
6. Repeat (3). . . (5) a few times
7. `ls` finishes (returns from `main()` or calls `exit()`)
8. Kernel marks process as "done executing"
    - Fun fact: process is now a zombie
9. Parent process (the shell) cleans up child process
    - This way, the parent can get the process's exit status
10. Kernel frees resources used by the process

Some last thoughts

Processes vs. Jobs

- Every job corresponds to an underlying process
- Job IDs are local to the parent shell
- Process IDs are global — can be accessed in any shell

What happens when the parent process terminates?

- Can be configured in different ways
- Typical shell: running background jobs do not terminate
- Children processes are assigned to a new parent
  - Details not important for this class
  - Remember: parent process cleans up the terminated children
    Otherwise, get lots of stray zombie processes

ps: list processes

- Lots of options (consult your `man` pages)
  - Several for "which processes to display"
  - Several for "what information to display"
  - Beware — UNIX vs. BSD options

UNIX	:	preceeded by `-`, may be grouped
BSD	:	not preceeded by `-`, may be grouped

- `ps` default: display "your" processes tied to current terminal
- Useful BSD options:

`a`	:	display processes for all users
`x`	:	display processes not tied to current terminal
`u`	:	longer output

- Useful UNIX options:

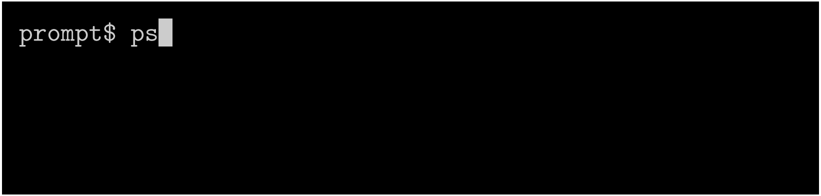
`-e`	:	Like `au`
`-f`	:	longer output but different from `u`

ps example (1)

```
prompt$ █
```

- These are my processes, tied to this terminal
- Column `PID`: process ID
- Column `TTY`: which terminal
- Column `TIME`: total CPU time used so far
- Column `CMD`: the command

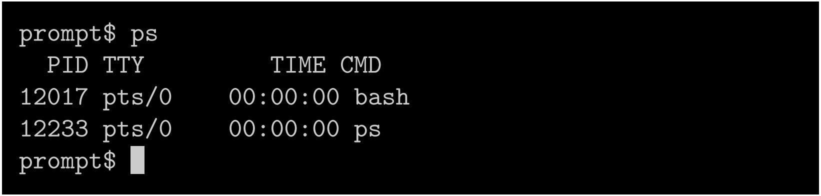
ps example (1)



```
prompt$ ps
```

- These are my processes, tied to this terminal
- Column `PID`: process ID
- Column `TTY`: which terminal
- Column `TIME`: total CPU time used so far
- Column `CMD`: the command

ps example (1)



```
prompt$ ps
  PID TTY          TIME CMD
12017 pts/0    00:00:00 bash
12233 pts/0    00:00:00 ps
prompt$ █
```

- These are my processes, tied to this terminal
- Column PID: process ID
- Column TTY: which terminal
- Column TIME: total CPU time used so far
- Column CMD: the command

ps example (1)

```
prompt$ ps
  PID TTY          TIME CMD
12017 pts/0    00:00:00 bash
12233 pts/0    00:00:00 ps
prompt$
```

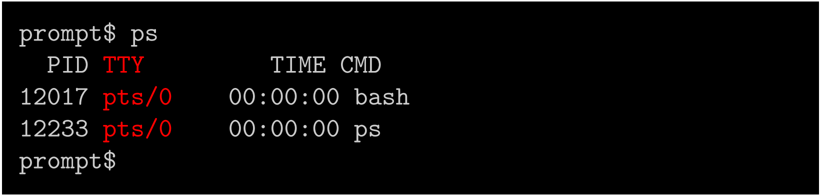
- These are my processes, tied to this terminal
- Column `PID`: process ID
- Column `TTY`: which terminal
- Column `TIME`: total CPU time used so far
- Column `CMD`: the command

ps example (1)

```
prompt$ ps
  PID TTY          TIME CMD
12017 pts/0    00:00:00 bash
12233 pts/0    00:00:00 ps
prompt$
```

- These are my processes, tied to this terminal
- Column `PID`: process ID
- Column `TTY`: which terminal
- Column `TIME`: total CPU time used so far
- Column `CMD`: the command

ps example (1)



```
prompt$ ps
  PID TTY          TIME CMD
12017 pts/0    00:00:00 bash
12233 pts/0    00:00:00 ps
prompt$
```

- These are my processes, tied to this terminal
- Column `PID`: process ID
- Column `TTY`: which terminal
- Column `TIME`: total CPU time used so far
- Column `CMD`: the command

ps example (1)

```
prompt$ ps
  PID TTY          TIME CMD
12017 pts/0    00:00:00 bash
12233 pts/0    00:00:00 ps
prompt$
```

- These are my processes, tied to this terminal
- Column `PID`: process ID
- Column `TTY`: which terminal
- Column `TIME`: total CPU time used so far
- Column `CMD`: the command

ps example (1)

```
prompt$ ps
  PID TTY          TIME CMD
12017 pts/0    00:00:00 bash
12233 pts/0    00:00:00 ps
prompt$
```

- These are my processes, tied to this terminal
- Column `PID`: process ID
- Column `TTY`: which terminal
- Column `TIME`: total CPU time used so far
- Column `CMD`: the command

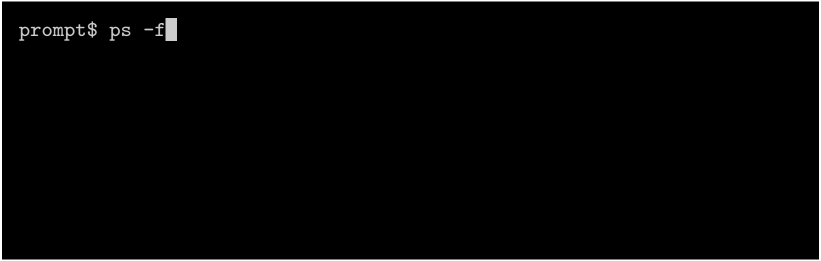
ps example (2)

```
prompt$ 
```

- Column UID/USER: process owner
- Column PPID: PID of parent process
- Column STAT: Process state

  R : running or runnable
  S : sleeping (waiting for something)
  ... : there are others
- Column STIME/START: time process was created

ps example (2)



```
prompt$ ps -f
```

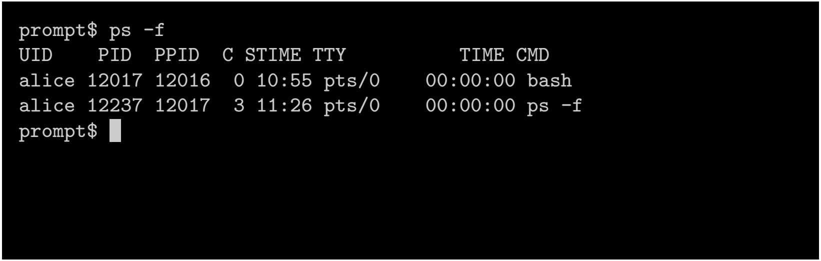
- Column UID/USER: process owner
- Column PPID: PID of parent process
- Column STAT: Process state

  R : running or runnable

  S : sleeping (waiting for something)

  . . . : there are others
- Column STIME/START: time process was created

ps example (2)



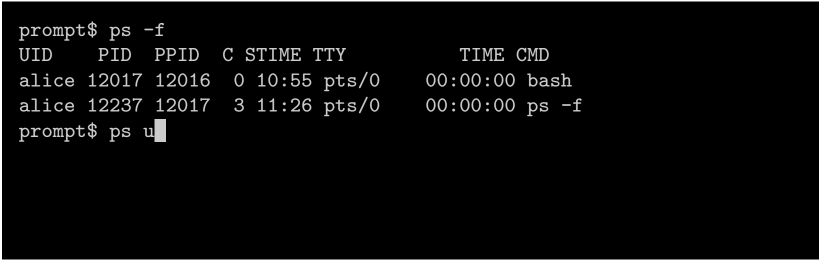
```
prompt$ ps -f
UID     PID  PPID  C STIME TTY          TIME CMD
alice 12017 12016  0 10:55 pts/0    00:00:00 bash
alice 12237 12017  3 11:26 pts/0    00:00:00 ps -f
prompt$ 
```

- Column `UID/USER`: process owner
- Column `PPID`: PID of parent process
- Column `STAT`: Process state

  R : running or runnable
  S : sleeping (waiting for something)
  . . . : there are others

- Column `STIME/START`: time process was created

ps example (2)



```
prompt$ ps -f
UID     PID  PPID  C STIME TTY          TIME CMD
alice 12017 12016  0 10:55 pts/0    00:00:00 bash
alice 12237 12017  3 11:26 pts/0    00:00:00 ps -f
prompt$ ps u
```

- Column `UID`/`USER`: process owner
- Column `PPID`: PID of parent process
- Column `STAT`: Process state

  R : running or runnable
  S : sleeping (waiting for something)
  ... : there are others

- Column `STIME`/`START`: time process was created

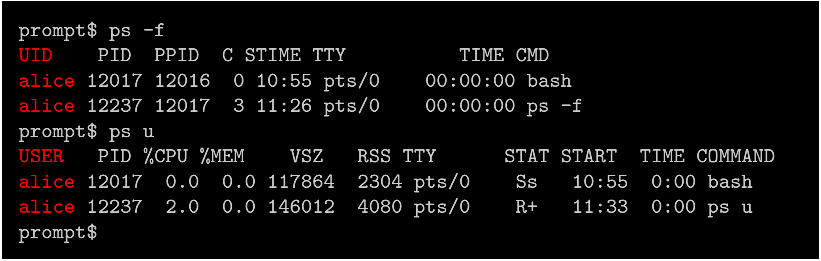
ps example (2)

```
prompt$ ps -f
UID     PID  PPID  C STIME TTY          TIME CMD
alice 12017 12016  0 10:55 pts/0    00:00:00 bash
alice 12237 12017  3 11:26 pts/0    00:00:00 ps -f
prompt$ ps u
USER    PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
alice 12017  0.0  0.0 117864  2304 pts/0     Ss   10:55   0:00 bash
alice 12237  2.0  0.0 146012  4080 pts/0     R+   11:33   0:00 ps u
prompt$ 
```

- Column `UID`/`USER`: process owner
- Column `PPID`: PID of parent process
- Column `STAT`: Process state

  R : running or runnable
  S : sleeping (waiting for something)
  ... : there are others
- Column `STIME`/`START`: time process was created

ps example (2)



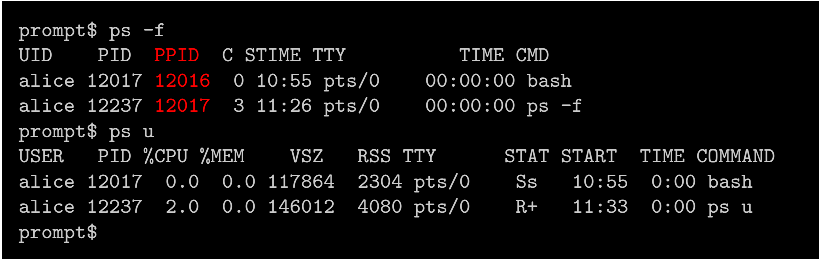
```
prompt$ ps -f
UID     PID  PPID  C STIME TTY          TIME CMD
alice 12017 12016  0 10:55 pts/0    00:00:00 bash
alice 12237 12017  3 11:26 pts/0    00:00:00 ps -f
prompt$ ps u
USER    PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
alice 12017  0.0  0.0 117864  2304 pts/0     Ss   10:55   0:00 bash
alice 12237  2.0  0.0 146012  4080 pts/0     R+   11:33   0:00 ps u
prompt$
```

- Column `UID`/`USER`: process owner
- Column `PPID`: PID of parent process
- Column `STAT`: Process state

  R : running or runnable
  S : sleeping (waiting for something)
  . . . : there are others

- Column `STIME`/`START`: time process was created

ps example (2)



```
prompt$ ps -f
UID     PID  PPID  C STIME TTY          TIME CMD
alice 12017 12016  0 10:55 pts/0    00:00:00 bash
alice 12237 12017  3 11:26 pts/0    00:00:00 ps -f
prompt$ ps u
USER    PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
alice 12017  0.0  0.0 117864  2304 pts/0     Ss   10:55   0:00 bash
alice 12237  2.0  0.0 146012  4080 pts/0     R+   11:33   0:00 ps u
prompt$
```

- Column UID/USER: process owner
- Column PPID: PID of parent process
- Column STAT: Process state

  R : running or runnable
  S : sleeping (waiting for something)
  ... : there are others

- Column STIME/START: time process was created

ps example (2)

```
prompt$ ps -f
UID     PID  PPID  C STIME TTY          TIME CMD
alice 12017 12016  0 10:55 pts/0    00:00:00 bash
alice 12237 12017  3 11:26 pts/0    00:00:00 ps -f
prompt$ ps u
USER    PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
alice 12017  0.0  0.0 117864  2304 pts/0     Ss   10:55   0:00 bash
alice 12237  2.0  0.0 146012  4080 pts/0     R+   11:33   0:00 ps u
prompt$
```

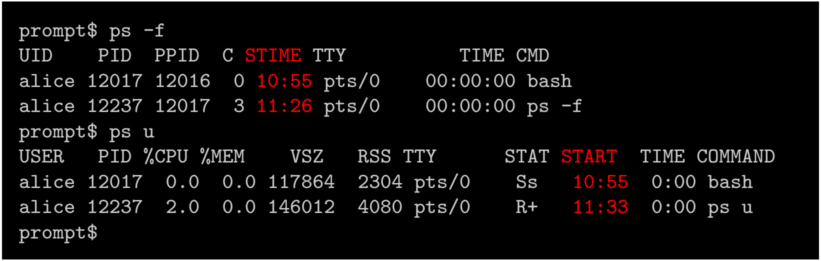
- Column `UID`/`USER`: process owner
- Column `PPID`: PID of parent process
- Column `STAT`: Process state

  R : running or runnable

  S : sleeping (waiting for something)

  . . . : there are others
- Column `STIME`/`START`: time process was created

ps example (2)



```
prompt$ ps -f
UID    PID  PPID  C STIME TTY          TIME CMD
alice 12017 12016  0 10:55 pts/0    00:00:00 bash
alice 12237 12017  3 11:26 pts/0    00:00:00 ps -f
prompt$ ps u
USER   PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
alice 12017  0.0  0.0 117864  2304 pts/0    Ss   10:55  0:00 bash
alice 12237  2.0  0.0 146012  4080 pts/0    R+   11:33  0:00 ps u
prompt$
```

- Column `UID`/`USER`: process owner
- Column `PPID`: PID of parent process
- Column `STAT`: Process state

  R : running or runnable
  S : sleeping (waiting for something)
  . . . : there are others

- Column `STIME`/`START`: time process was created

ps example (3)

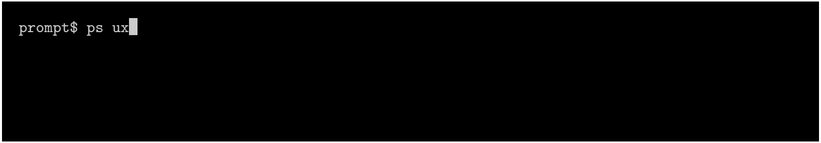
```
prompt$ 
```

- These are *all* of `alice`'s processes
- These columns have to do with memory usage

  `%MEM` : Percent of system memory used by this process
  `VSZ` : Total virtual memory used by this process
  `RSS` : Resident set size

  - Basically, (guess of) current "required" memory
  - I.e., Memory that should not be swapped to disk
  - For more details — take ComS 352

ps example (3)



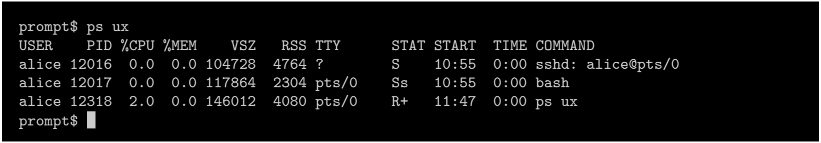
```
prompt$ ps ux
```

- These are *all* of `alice`'s processes
- These columns have to do with memory usage

  `%MEM` : Percent of system memory used by this process
  `VSZ` : Total virtual memory used by this process
  `RSS` : Resident set size

  - Basically, (guess of) current "required" memory
  - I.e., Memory that should not be swapped to disk
  - For more details — take ComS 352

ps example (3)



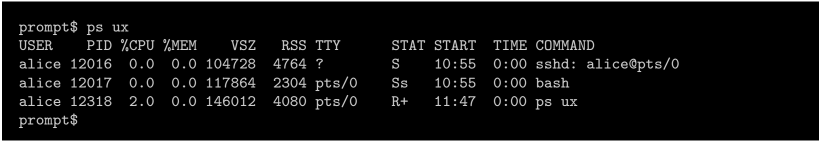
```
prompt$ ps ux
USER     PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
alice 12016  0.0  0.0 104728  4764 ?        S    10:55   0:00 sshd: alice@pts/0
alice 12017  0.0  0.0 117864  2304 pts/0    Ss   10:55   0:00 bash
alice 12318  2.0  0.0 146012  4080 pts/0    R+   11:47   0:00 ps ux
prompt$ 
```

- These are *all* of `alice`'s processes
- These columns have to do with memory usage

  `%MEM` : Percent of system memory used by this process
  `VSZ` : Total virtual memory used by this process
  `RSS` : Resident set size

  - Basically, (guess of) current “required” memory
  - I.e., Memory that should not be swapped to disk
  - For more details — take ComS 352

ps example (3)



```
prompt$ ps ux
USER     PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
alice 12016  0.0  0.0 104728  4764 ?        S    10:55   0:00 sshd: alice@pts/0
alice 12017  0.0  0.0 117864  2304 pts/0    Ss   10:55   0:00 bash
alice 12318  2.0  0.0 146012  4080 pts/0    R+   11:47   0:00 ps ux
prompt$
```

- These are *all* of `alice`'s processes
- These columns have to do with memory usage

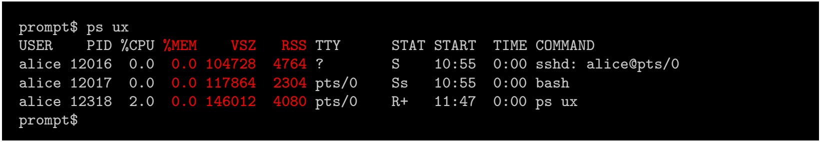
  `%MEM` : Percent of system memory used by this process

  `VSZ` : Total virtual memory used by this process

  `RSS` : Resident set size

  - Basically, (guess of) current "required" memory
  - I.e., Memory that should not be swapped to disk
  - For more details — take ComS 352

ps example (3)



```
prompt$ ps ux
USER     PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
alice 12016  0.0  0.0 104728  4764 ?        S    10:55   0:00 sshd: alice@pts/0
alice 12017  0.0  0.0 117864  2304 pts/0    Ss   10:55   0:00 bash
alice 12318  2.0  0.0 146012  4080 pts/0    R+   11:47   0:00 ps ux
prompt$
```

- These are *all* of `alice`'s processes
- These columns have to do with memory usage

  `%MEM` : Percent of system memory used by this process

  `VSZ` : Total virtual memory used by this process

  `RSS` : Resident set size

  - Basically, (guess of) current “required” memory
  - I.e., Memory that should not be swapped to disk
  - For more details — take ComS 352

ps for *all* processes

`ps -ef` or `ps aux`

```
prompt$
```

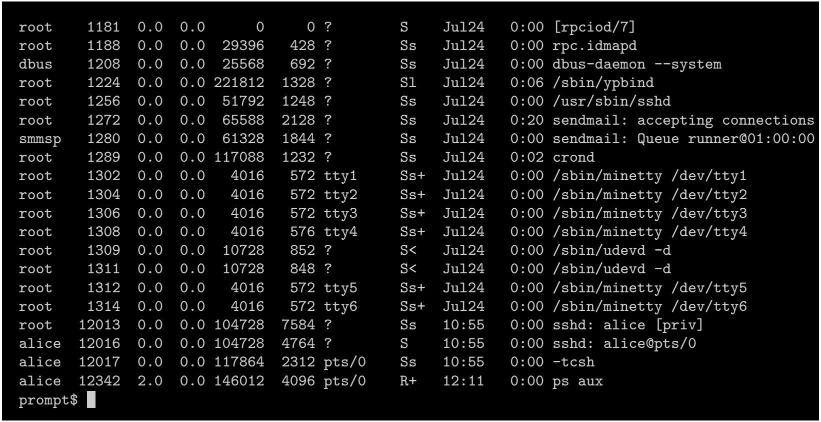
ps for *all* processes

`ps -ef` or `ps aux`

```
prompt$ ps aux
```

ps for *all* processes

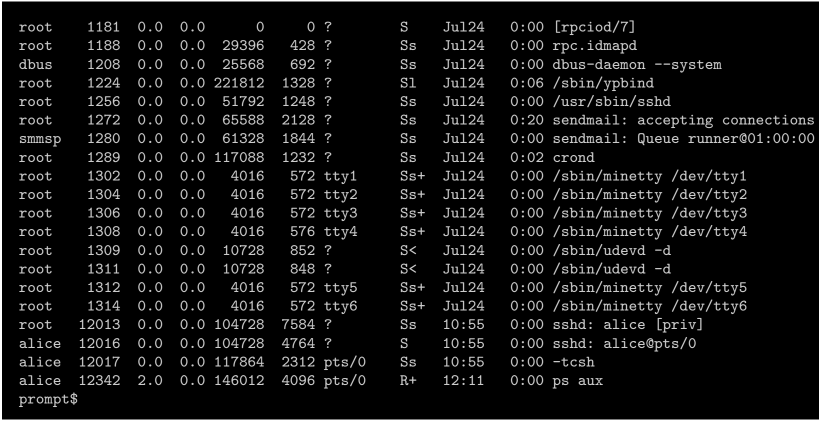
`ps -ef` or `ps aux`



```
root     1181  0.0  0.0      0     0 ?        S    Jul24   0:00 [rpciod/7]
root     1188  0.0  0.0  29396   428 ?        Ss   Jul24   0:00 rpc.idmapd
dbus     1208  0.0  0.0  25568   692 ?        Ss   Jul24   0:00 dbus-daemon --system
root     1224  0.0  0.0 221812  1328 ?        Sl   Jul24   0:06 /sbin/ypbind
root     1256  0.0  0.0  51792  1248 ?        Ss   Jul24   0:00 /usr/sbin/sshd
root     1272  0.0  0.0  65588  2128 ?        Ss   Jul24   0:20 sendmail: accepting connections
smmsp    1280  0.0  0.0  61328  1844 ?        Ss   Jul24   0:00 sendmail: Queue runner@01:00:00
root     1289  0.0  0.0 117088  1232 ?        Ss   Jul24   0:02 crond
root     1302  0.0  0.0   4016   572 tty1     Ss+  Jul24   0:00 /sbin/minetty /dev/tty1
root     1304  0.0  0.0   4016   572 tty2     Ss+  Jul24   0:00 /sbin/minetty /dev/tty2
root     1306  0.0  0.0   4016   572 tty3     Ss+  Jul24   0:00 /sbin/minetty /dev/tty3
root     1308  0.0  0.0   4016   576 tty4     Ss+  Jul24   0:00 /sbin/minetty /dev/tty4
root     1309  0.0  0.0  10728   852 ?        S<   Jul24   0:00 /sbin/udevd -d
root     1311  0.0  0.0  10728   848 ?        S<   Jul24   0:00 /sbin/udevd -d
root     1312  0.0  0.0   4016   572 tty5     Ss+  Jul24   0:00 /sbin/minetty /dev/tty5
root     1314  0.0  0.0   4016   572 tty6     Ss+  Jul24   0:00 /sbin/minetty /dev/tty6
root    12013  0.0  0.0 104728  7584 ?        Ss   10:55   0:00 sshd: alice [priv]
alice   12016  0.0  0.0 104728  4764 ?        S    10:55   0:00 sshd: alice@pts/0
alice   12017  0.0  0.0 117864  2312 pts/0    Ss   10:55   0:00 -tcsh
alice   12342  2.0  0.0 146012  4096 pts/0    R+   12:11   0:00 ps aux
prompt$ █
```

ps for *all* processes

`ps -ef` or `ps aux`



```
root     1181  0.0  0.0      0     0 ?        S    Jul24   0:00 [rpciod/7]
root     1188  0.0  0.0  29396   428 ?        Ss   Jul24   0:00 rpc.idmapd
dbus     1208  0.0  0.0  25568   692 ?        Ss   Jul24   0:00 dbus-daemon --system
root     1224  0.0  0.0 221812  1328 ?        Sl   Jul24   0:06 /sbin/ypbind
root     1256  0.0  0.0  51792  1248 ?        Ss   Jul24   0:00 /usr/sbin/sshd
root     1272  0.0  0.0  65588  2128 ?        Ss   Jul24   0:20 sendmail: accepting connections
smmsp    1280  0.0  0.0  61328  1844 ?        Ss   Jul24   0:00 sendmail: Queue runner@01:00:00
root     1289  0.0  0.0 117088  1232 ?        Ss   Jul24   0:02 crond
root     1302  0.0  0.0   4016   572 tty1     Ss+  Jul24   0:00 /sbin/minetty /dev/tty1
root     1304  0.0  0.0   4016   572 tty2     Ss+  Jul24   0:00 /sbin/minetty /dev/tty2
root     1306  0.0  0.0   4016   572 tty3     Ss+  Jul24   0:00 /sbin/minetty /dev/tty3
root     1308  0.0  0.0   4016   576 tty4     Ss+  Jul24   0:00 /sbin/minetty /dev/tty4
root     1309  0.0  0.0  10728   852 ?        S<   Jul24   0:00 /sbin/udevd -d
root     1311  0.0  0.0  10728   848 ?        S<   Jul24   0:00 /sbin/udevd -d
root     1312  0.0  0.0   4016   572 tty5     Ss+  Jul24   0:00 /sbin/minetty /dev/tty5
root     1314  0.0  0.0   4016   572 tty6     Ss+  Jul24   0:00 /sbin/minetty /dev/tty6
root    12013  0.0  0.0 104728  7584 ?        Ss   10:55   0:00 sshd: alice [priv]
alice   12016  0.0  0.0 104728  4764 ?        S    10:55   0:00 sshd: alice@pts/0
alice   12017  0.0  0.0 117864  2312 pts/0    Ss   10:55   0:00 -tcsh
alice   12342  2.0  0.0 146012  4096 pts/0    R+   12:11   0:00 ps aux
prompt$
```

Why so many processes?

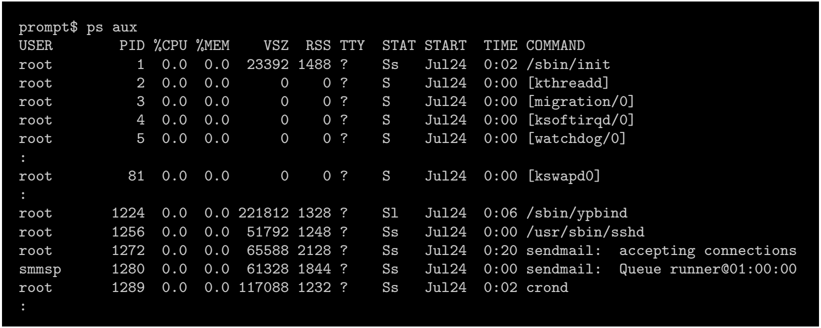
Why so many processes

The system uses processes to do various things

- Started when the system boots
- Do system tasks
- Many are daemons:
  - Designed to never terminate (except for `TERM` signal)
  - *Very* common for networking tasks
  - Often, names end with "`d`"
- Many run on behalf of the kernel
  - Often, names start with "`k`"
- Not connected to any terminal
  - Respond to signals
  - May allow incoming connections . . . more on this later

Some obvious daemons

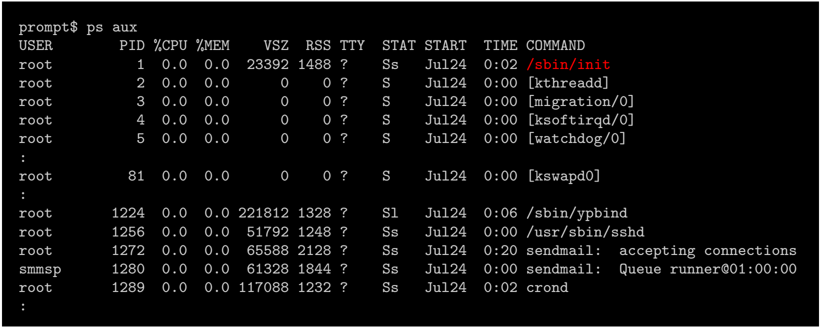
A closer look at ps aux



```
prompt$ ps aux
USER        PID %CPU %MEM    VSZ  RSS TTY   STAT START   TIME COMMAND
root          1  0.0  0.0  23392 1488 ?     Ss   Jul24   0:02 /sbin/init
root          2  0.0  0.0      0    0 ?     S    Jul24   0:00 [kthreadd]
root          3  0.0  0.0      0    0 ?     S    Jul24   0:00 [migration/0]
root          4  0.0  0.0      0    0 ?     S    Jul24   0:00 [ksoftirqd/0]
root          5  0.0  0.0      0    0 ?     S    Jul24   0:00 [watchdog/0]
:
root         81  0.0  0.0      0    0 ?     S    Jul24   0:00 [kswapd0]
:
root       1224  0.0  0.0 221812 1328 ?     Sl   Jul24   0:06 /sbin/ypbind
root       1256  0.0  0.0  51792 1248 ?     Ss   Jul24   0:00 /usr/sbin/sshd
root       1272  0.0  0.0  65588 2128 ?     Ss   Jul24   0:20 sendmail:  accepting connections
smmsp      1280  0.0  0.0  61328 1844 ?     Ss   Jul24   0:00 sendmail:  Queue runner@01:00:00
root       1289  0.0  0.0 117088 1232 ?     Ss   Jul24   0:02 crond
:
```

Some obvious daemons

A closer look at ps aux

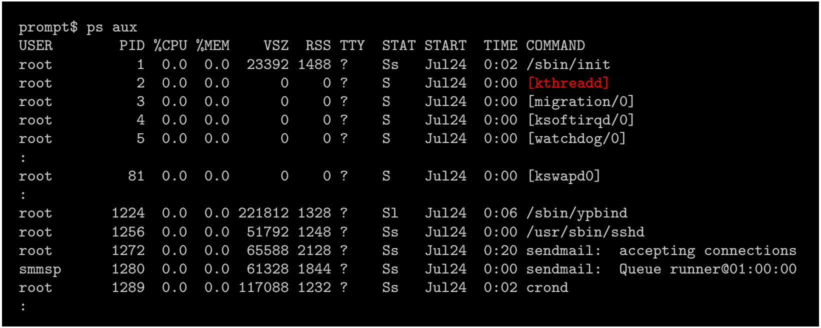


```
prompt$ ps aux
USER        PID %CPU %MEM    VSZ  RSS TTY    STAT START   TIME COMMAND
root          1  0.0  0.0  23392 1488 ?      Ss   Jul24   0:02 /sbin/init
root          2  0.0  0.0      0    0 ?      S    Jul24   0:00 [kthreadd]
root          3  0.0  0.0      0    0 ?      S    Jul24   0:00 [migration/0]
root          4  0.0  0.0      0    0 ?      S    Jul24   0:00 [ksoftirqd/0]
root          5  0.0  0.0      0    0 ?      S    Jul24   0:00 [watchdog/0]
:
root         81  0.0  0.0      0    0 ?      S    Jul24   0:00 [kswapd0]
:
root       1224  0.0  0.0 221812 1328 ?      Sl   Jul24   0:06 /sbin/ypbind
root       1256  0.0  0.0  51792 1248 ?      Ss   Jul24   0:00 /usr/sbin/sshd
root       1272  0.0  0.0  65588 2128 ?      Ss   Jul24   0:20 sendmail: accepting connections
smmsp      1280  0.0  0.0  61328 1844 ?      Ss   Jul24   0:00 sendmail: Queue runner@01:00:00
root       1289  0.0  0.0 117088 1232 ?      Ss   Jul24   0:02 crond
:
```

- init: Important system process, comes up at boot time

Some obvious daemons

A closer look at ps aux

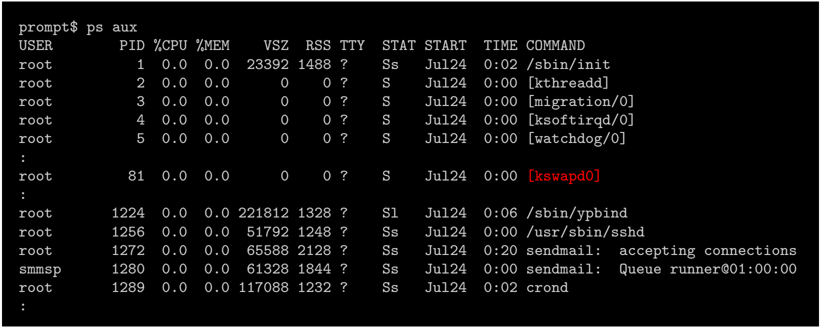


```
prompt$ ps aux
USER         PID %CPU %MEM    VSZ  RSS TTY  STAT START  TIME COMMAND
root           1  0.0  0.0  23392 1488 ?    Ss   Jul24  0:02 /sbin/init
root           2  0.0  0.0      0    0 ?    S    Jul24  0:00 [kthreadd]
root           3  0.0  0.0      0    0 ?    S    Jul24  0:00 [migration/0]
root           4  0.0  0.0      0    0 ?    S    Jul24  0:00 [ksoftirqd/0]
root           5  0.0  0.0      0    0 ?    S    Jul24  0:00 [watchdog/0]
:
root          81  0.0  0.0      0    0 ?    S    Jul24  0:00 [kswapd0]
:
root        1224  0.0  0.0 221812 1328 ?    Sl   Jul24  0:06 /sbin/ypbind
root        1256  0.0  0.0  51792 1248 ?    Ss   Jul24  0:00 /usr/sbin/sshd
root        1272  0.0  0.0  65588 2128 ?    Ss   Jul24  0:20 sendmail:  accepting connections
smmsp       1280  0.0  0.0  61328 1844 ?    Ss   Jul24  0:00 sendmail:  Queue runner@01:00:00
root        1289  0.0  0.0 117088 1232 ?    Ss   Jul24  0:02 crond
:
```

- `init`: Important system process, comes up at boot time
- `kthreadd`: kernel thread daemon

Some obvious daemons

A closer look at ps aux



```
prompt$ ps aux
USER        PID %CPU %MEM    VSZ  RSS TTY   STAT START  TIME COMMAND
root          1  0.0  0.0  23392 1488 ?     Ss   Jul24  0:02 /sbin/init
root          2  0.0  0.0      0    0 ?     S    Jul24  0:00 [kthreadd]
root          3  0.0  0.0      0    0 ?     S    Jul24  0:00 [migration/0]
root          4  0.0  0.0      0    0 ?     S    Jul24  0:00 [ksoftirqd/0]
root          5  0.0  0.0      0    0 ?     S    Jul24  0:00 [watchdog/0]
:
root         81  0.0  0.0      0    0 ?     S    Jul24  0:00 [kswapd0]
:
root       1224  0.0  0.0 221812 1328 ?     Sl   Jul24  0:06 /sbin/ypbind
root       1256  0.0  0.0  51792 1248 ?     Ss   Jul24  0:00 /usr/sbin/sshd
root       1272  0.0  0.0  65588 2128 ?     Ss   Jul24  0:20 sendmail: accepting connections
smmsp      1280  0.0  0.0  61328 1844 ?     Ss   Jul24  0:00 sendmail: Queue runner@01:00:00
root       1289  0.0  0.0 117088 1232 ?     Ss   Jul24  0:02 crond
:
```

- `init`: Important system process, comes up at boot time
- `kthreadd`: kernel thread daemon
- `kswapd`: kernel swap daemon (for virtual memory)

Some obvious daemons

A closer look at ps aux

```
prompt$ ps aux
USER        PID %CPU %MEM    VSZ  RSS TTY   STAT START  TIME COMMAND
root          1  0.0  0.0  23392 1488 ?     Ss   Jul24  0:02 /sbin/init
root          2  0.0  0.0      0    0 ?     S    Jul24  0:00 [kthreadd]
root          3  0.0  0.0      0    0 ?     S    Jul24  0:00 [migration/0]
root          4  0.0  0.0      0    0 ?     S    Jul24  0:00 [ksoftirqd/0]
root          5  0.0  0.0      0    0 ?     S    Jul24  0:00 [watchdog/0]
:
root         81  0.0  0.0      0    0 ?     S    Jul24  0:00 [kswapd0]
:
root       1224  0.0  0.0 221812 1328 ?     Sl   Jul24  0:06 /sbin/ypbind
root       1256  0.0  0.0  51792 1248 ?     Ss   Jul24  0:00 /usr/sbin/sshd
root       1272  0.0  0.0  65588 2128 ?     Ss   Jul24  0:20 sendmail: accepting connections
smmsp      1280  0.0  0.0  61328 1844 ?     Ss   Jul24  0:00 sendmail: Queue runner@01:00:00
root       1289  0.0  0.0 117088 1232 ?     Ss   Jul24  0:02 crond
:
```

- `init`: Important system process, comes up at boot time
- `kthreadd`: kernel thread daemon
- `kswapd`: kernel swap daemon (for virtual memory)
- More daemons, but not *kernel* ones

top: display all processes

- Processes are sorted by CPU usage
- Display is updated every few seconds (q to quit)
- Useful if you are waiting for something to finish

```
prompt$ 
```

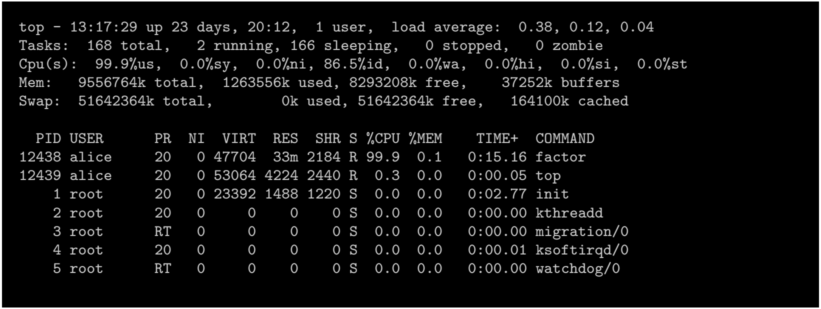
top: display all processes

- Processes are sorted by CPU usage
- Display is updated every few seconds (q to quit)
- Useful if you are waiting for something to finish

```
prompt$ top
```

top: display all processes

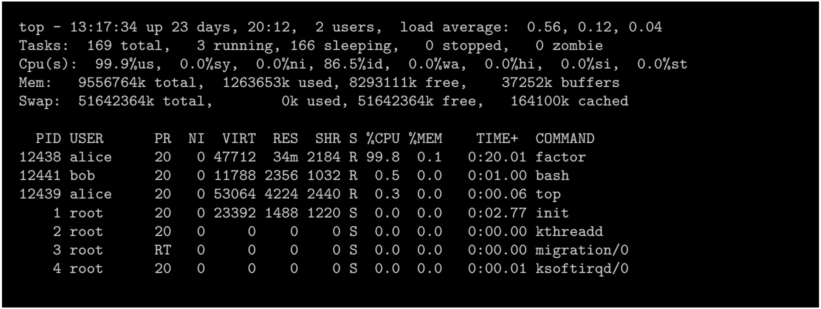
- Processes are sorted by CPU usage
- Display is updated every few seconds (q to quit)
- Useful if you are waiting for something to finish



```
top - 13:17:29 up 23 days, 20:12,  1 user,  load average:  0.38, 0.12, 0.04
Tasks:  168 total,   2 running, 166 sleeping,   0 stopped,   0 zombie
Cpu(s):  99.9%us,  0.0%sy,  0.0%ni, 86.5%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Mem:   9556764k total,  1263556k used, 8293208k free,    37252k buffers
Swap:  51642364k total,        0k used, 51642364k free,   164100k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
12438 alice     20   0 47704  33m 2184 R 99.9  0.1   0:15.16 factor
12439 alice     20   0 53064 4224 2440 R  0.3  0.0   0:00.05 top
    1 root      20   0 23392 1488 1220 S  0.0  0.0   0:02.77 init
    2 root      20   0     0    0    0 S  0.0  0.0   0:00.00 kthreadd
    3 root      RT   0     0    0    0 S  0.0  0.0   0:00.00 migration/0
    4 root      20   0     0    0    0 S  0.0  0.0   0:00.01 ksoftirqd/0
    5 root      RT   0     0    0    0 S  0.0  0.0   0:00.00 watchdog/0
```

top: display all processes

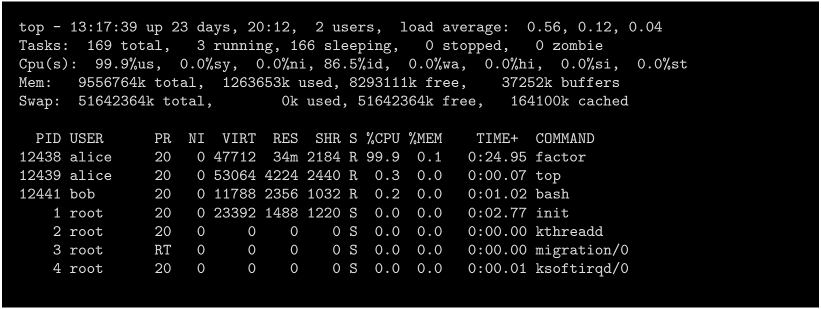
- Processes are sorted by CPU usage
- Display is updated every few seconds (q to quit)
- Useful if you are waiting for something to finish



```
top - 13:17:34 up 23 days, 20:12,  2 users,  load average:  0.56, 0.12, 0.04
Tasks:  169 total,   3 running, 166 sleeping,   0 stopped,   0 zombie
Cpu(s):  99.9%us,  0.0%sy,  0.0%ni, 86.5%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Mem:   9556764k total,  1263653k used, 8293111k free,    37252k buffers
Swap:  51642364k total,        0k used, 51642364k free,   164100k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
12438 alice     20   0 47712  34m 2184 R 99.8  0.1   0:20.01 factor
12441 bob       20   0 11788 2356 1032 R  0.5  0.0   0:01.00 bash
12439 alice     20   0 53064 4224 2440 R  0.3  0.0   0:00.06 top
    1 root      20   0 23392 1488 1220 S  0.0  0.0   0:02.77 init
    2 root      20   0     0    0    0 S  0.0  0.0   0:00.00 kthreadd
    3 root      RT   0     0    0    0 S  0.0  0.0   0:00.00 migration/0
    4 root      20   0     0    0    0 S  0.0  0.0   0:00.01 ksoftirqd/0
```

top: display all processes

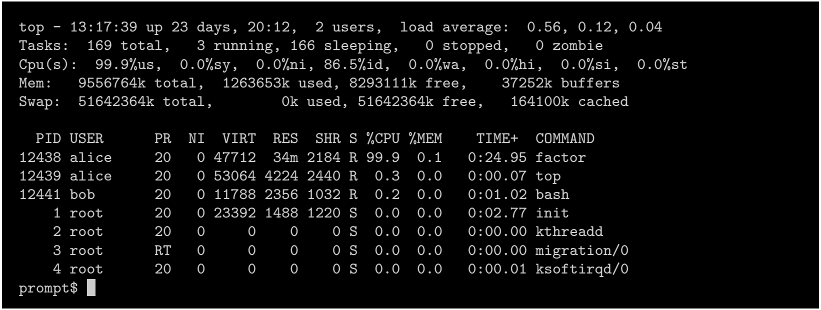
- Processes are sorted by CPU usage
- Display is updated every few seconds (q to quit)
- Useful if you are waiting for something to finish



```
top - 13:17:39 up 23 days, 20:12,  2 users,  load average:  0.56, 0.12, 0.04
Tasks:  169 total,   3 running, 166 sleeping,   0 stopped,   0 zombie
Cpu(s):  99.9%us,  0.0%sy,  0.0%ni, 86.5%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Mem:   9556764k total,  1263653k used, 8293111k free,    37252k buffers
Swap:  51642364k total,        0k used, 51642364k free,   164100k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
12438 alice     20   0 47712  34m 2184 R 99.9  0.1   0:24.95 factor
12439 alice     20   0 53064 4224 2440 R  0.3  0.0   0:00.07 top
12441 bob       20   0 11788 2356 1032 R  0.2  0.0   0:01.02 bash
    1 root      20   0 23392 1488 1220 S  0.0  0.0   0:02.77 init
    2 root      20   0     0    0    0 S  0.0  0.0   0:00.00 kthreadd
    3 root      RT   0     0    0    0 S  0.0  0.0   0:00.00 migration/0
    4 root      20   0     0    0    0 S  0.0  0.0   0:00.01 ksoftirqd/0
```

top: display all processes

- Processes are sorted by CPU usage
- Display is updated every few seconds (q to quit)
- Useful if you are waiting for something to finish



```
top - 13:17:39 up 23 days, 20:12,  2 users,  load average:  0.56, 0.12, 0.04
Tasks:  169 total,   3 running, 166 sleeping,   0 stopped,   0 zombie
Cpu(s):  99.9%us,  0.0%sy,  0.0%ni, 86.5%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Mem:   9556764k total,  1263653k used, 8293111k free,    37252k buffers
Swap:  51642364k total,        0k used, 51642364k free,   164100k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
12438 alice     20   0 47712  34m 2184 R 99.9  0.1   0:24.95 factor
12439 alice     20   0 53064 4224 2440 R  0.3  0.0   0:00.07 top
12441 bob       20   0 11788 2356 1032 R  0.2  0.0   0:01.02 bash
    1 root      20   0 23392 1488 1220 S  0.0  0.0   0:02.77 init
    2 root      20   0     0    0    0 S  0.0  0.0   0:00.00 kthreadd
    3 root      RT   0     0    0    0 S  0.0  0.0   0:00.00 migration/0
    4 root      20   0     0    0    0 S  0.0  0.0   0:00.01 ksoftirqd/0
prompt$ █
```

Some job utilities that work with processes

`kill`: send a signal

Usage:

1. `kill -l`: list signals
2. `kill [-signal] %n`: send signal to job `n`
3. `kill [-signal] pid`: send signal to process `pid`

`wait`: wait for a job or process

Usage: `wait [%job] [pid] ...`

- `wait` is a shell builtin
- Can only wait for a process that is a child of the shell

Some job utilities that work with processes

`kill`: send a signal

Usage:

1. `kill -l`: list signals
2. `kill [-signal] %n`: send signal to job `n`
3. `kill [-signal] pid`: send signal to process `pid`
    - Only if you own the process, or are `root`

`wait`: wait for a job or process

Usage: `wait [%job] [pid] ...`

- `wait` is a shell builtin
- Can only wait for a process that is a child of the shell

Process priority

- Every process has a *priority*
  - Integer value, influences the scheduler
  - Linux: higher integer means lower priority

`nice`: run a command with lower priority

Usage: `nice cmd arg1 arg2 ...`

`renice`: adjust the priority of processes

- Usage: `renice change pid ...`
- Ordinary user: can only lower priority, if you own the process
- `root`: can raise or lower priority of any process

One last crazy thing (Linux only)

- There is a virtual filesystem under `/proc`
  - virtual: files are not stored on any disk
- Can access all kinds of system information — as files
  - E.g., have a look at `/proc/cpuinfo` and `/proc/meminfo`
- Subdirectory *n/*: info for process with PID *n*
- File permissions are as you would expect
  - Owner of subirectory *n/* is owner of process *n*
- `ps` and `top` simply read information from `/proc`

What is a “service”?

- A service is one or more processes (daemons)
- Their job is to provide a “service” (thus the name)
- For example:
  - The ssh daemon `sshd` handles incoming remote logins
- We will discuss these in depth when we get to “networks”
- For now we will discuss
  1. Utilities to start and stop services
  2. Utilities to control what is started at boot time

What is a “service”?

- A service is one or more processes (daemons)
- Their job is to provide a “service” (thus the name)
- For example:
  - The ssh daemon `sshd` handles incoming remote logins
- We will discuss these in depth when we get to “networks”
- For now we will discuss
  1. Utilities to start and stop services
  2. Utilities to control what is started at boot time

  Can’t I just start or kill the correct processes, by hand?

What is a “service”?

- A service is one or more processes (daemons)
- Their job is to provide a “service” (thus the name)
- For example:
  - The ssh daemon `sshd` handles incoming remote logins
- We will discuss these in depth when we get to “networks”
- For now we will discuss
  1. Utilities to start and stop services
  2. Utilities to control what is started at boot time

Can’t I just start or kill the correct processes, by hand?

- Of course you can (if you’re `root`). But...
- ...starting a service cleanly may involve multiple steps
- ...stopping a service cleanly may involve multiple steps

How is a Unix–style system initialized?

1. BIOS / EFI runs first
2. Control goes to the Bootloader
3. The Kernel is started
4. The `init` process starts
5. `init` initializes the rest of the system
    - How?
    - Unfortunately, it depends on the distribution

System V style initialization

- Comes from “AT&T System V”, mid 1980’s
- Abbreviated as “`sysvinit`”
- Uses *Runlevels*: numbered collection of scripts to run
    - Boot / halt / reboot: change runlevels
    - Runs scripts one at a time, in a specific order
    - No parallelism
- Fairly easy to configure and maintain
    - Specify which services to run in which runlevels
- SLOW
    - Services are started sequentially

- Before 2010: used by many mainstream Linux distributions

systemd style initialization

- Designed as a replacement for `sysvinit`
  - "Backwards compatible with `sysvinit`"
- Services are started in parallel
  - Subject to dependencies
  - Faster boot and shutdown times
- Has been adopted as the default by many distributions

Fedora	2011	RHEL	2014
Debian	2015	Ubuntu	2015

- Management of services done using `systemctl` utility
- Has seen some controversy and criticism
  - Too large and complex
  - Violates UNIX philosophy of "small, interconnected utilities"

Boot targets

Useful targets

`multi-user.target` : Multi-user, command-line

`graphical.target` : Multi-user, graphical

- Default target is symbolic link `/etc/systemd/system/default.target`
- On newer systems, can determine the default using

```
prompt$ systemctl get-default
```

- On newer systems, can set the default using

```
prompt$ systemctl set-default target-name[.target]
```

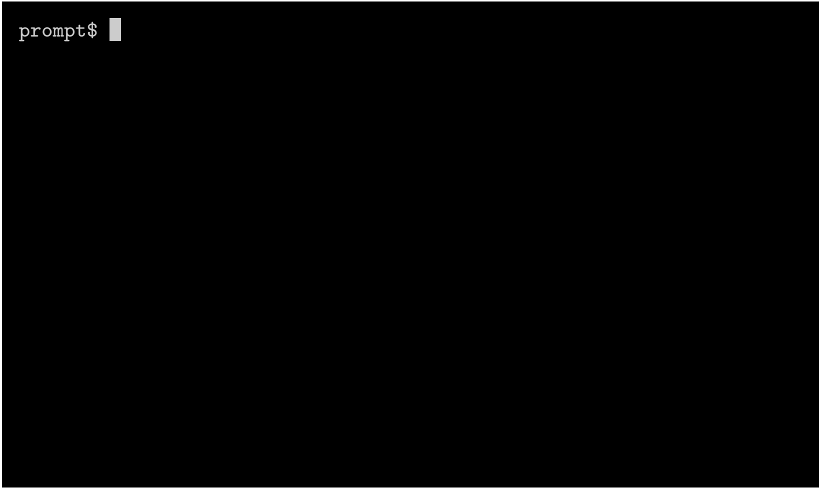
Starting and stopping services using `systemctl`

- Use the following arguments to `systemctl` to manage services

`start foobard`	:	Start service `foobard`
`stop foobard`	:	Stop service `foobard`
`restart foobard`	:	Restart service `foobard`
`status foobard`	:	Show current status of `foobard`
`enable foobard`	:	Turn `foobard` on at boot time
`disable foobard`	:	Turn `foobard` off at boot time
`is-enabled foobard`	:	Will `foobard` be started at boot?

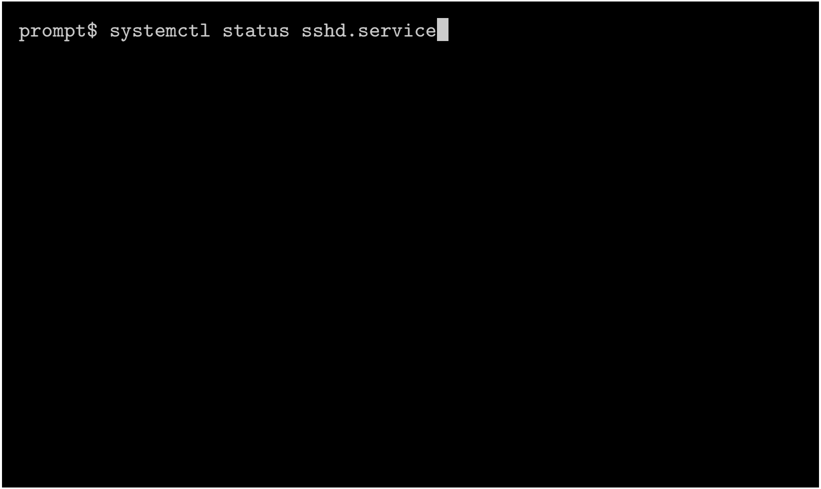
- On older systems, add "`.service`" to the service name
- On newer systems, the "`.service`" is optional
- Check the `man` page of `systemctl` for more information

systemctl example



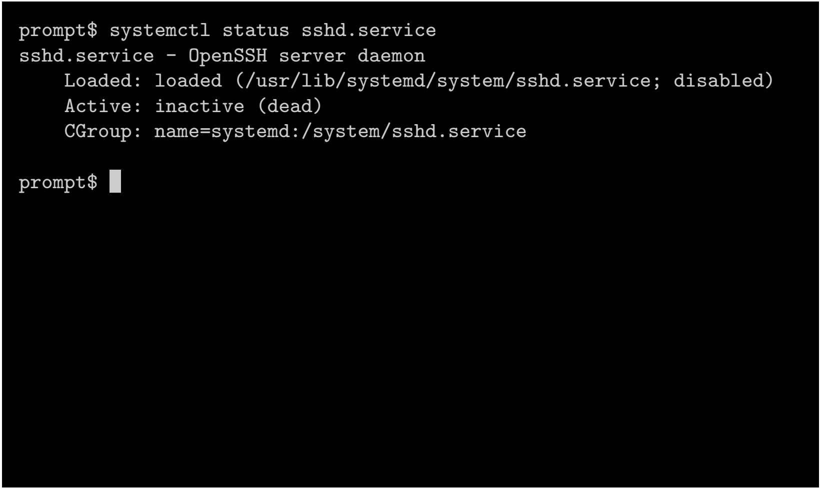
```
prompt$
```

systemctl example



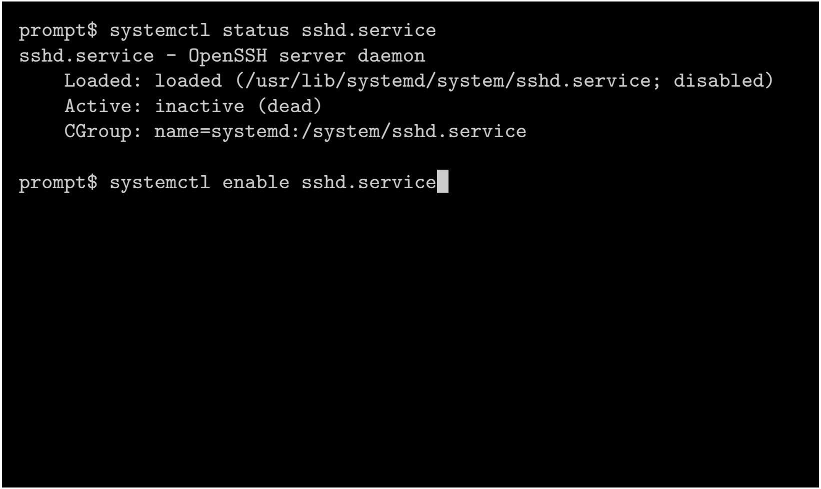
```
prompt$ systemctl status sshd.service
```

systemctl example



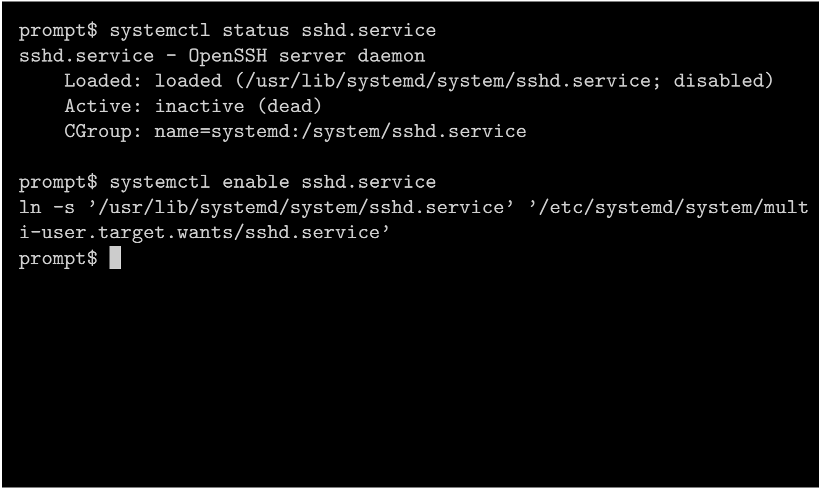
```
prompt$ systemctl status sshd.service
sshd.service - OpenSSH server daemon
    Loaded: loaded (/usr/lib/systemd/system/sshd.service; disabled)
    Active: inactive (dead)
    CGroup: name=systemd:/system/sshd.service

prompt$ █
```

systemctl example



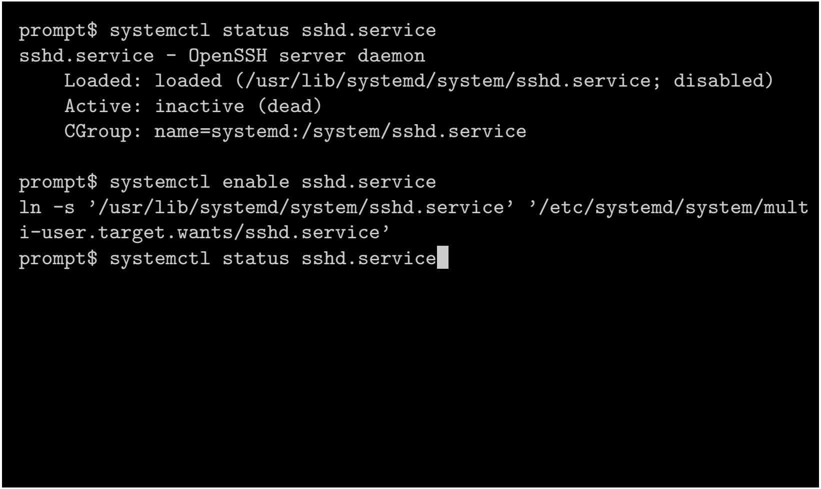
```
prompt$ systemctl status sshd.service
sshd.service - OpenSSH server daemon
    Loaded: loaded (/usr/lib/systemd/system/sshd.service; disabled)
    Active: inactive (dead)
    CGroup: name=systemd:/system/sshd.service

prompt$ systemctl enable sshd.service
```

`systemctl` example



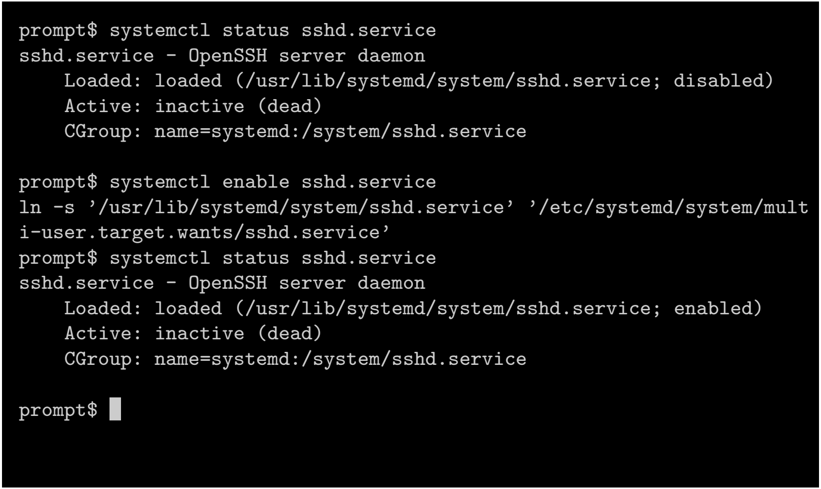
```
prompt$ systemctl status sshd.service
sshd.service - OpenSSH server daemon
    Loaded: loaded (/usr/lib/systemd/system/sshd.service; disabled)
    Active: inactive (dead)
    CGroup: name=systemd:/system/sshd.service

prompt$ systemctl enable sshd.service
ln -s ’/usr/lib/systemd/system/sshd.service’ ’/etc/systemd/system/mult
i-user.target.wants/sshd.service’
prompt$ █
```

systemctl example



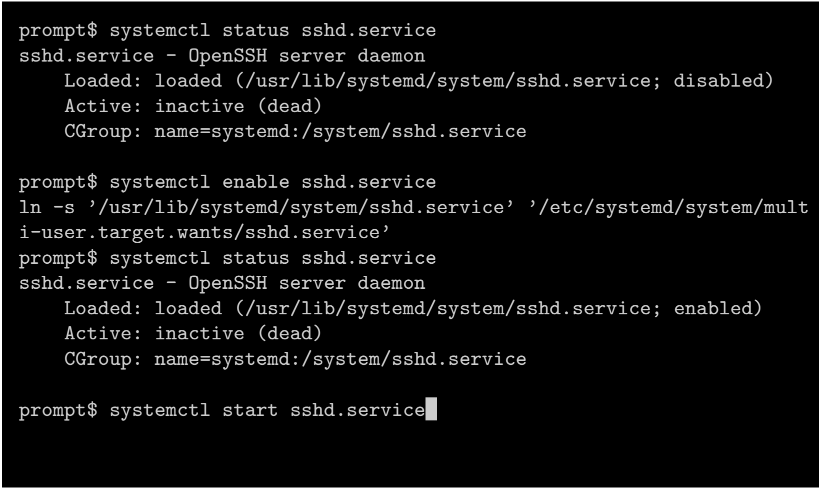
```
prompt$ systemctl status sshd.service
sshd.service - OpenSSH server daemon
    Loaded: loaded (/usr/lib/systemd/system/sshd.service; disabled)
    Active: inactive (dead)
    CGroup: name=systemd:/system/sshd.service

prompt$ systemctl enable sshd.service
ln -s ’/usr/lib/systemd/system/sshd.service’ ’/etc/systemd/system/mult
i-user.target.wants/sshd.service’
prompt$ systemctl status sshd.service
```

systemctl example



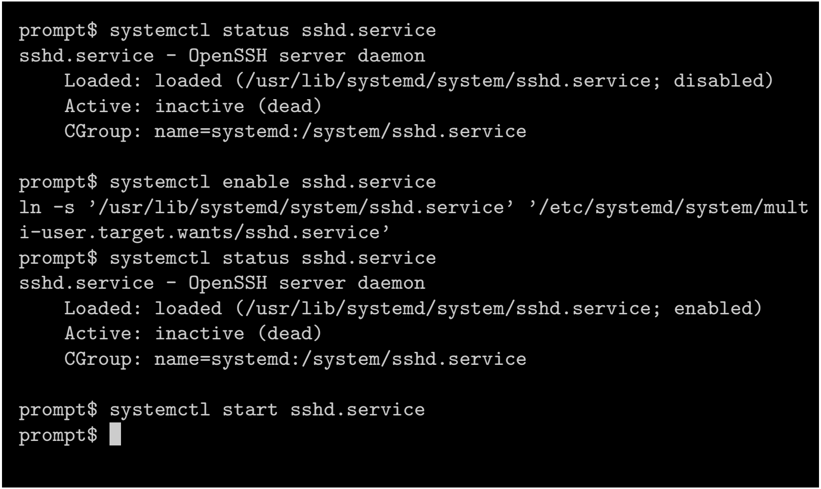
```
prompt$ systemctl status sshd.service
sshd.service - OpenSSH server daemon
    Loaded: loaded (/usr/lib/systemd/system/sshd.service; disabled)
    Active: inactive (dead)
    CGroup: name=systemd:/system/sshd.service

prompt$ systemctl enable sshd.service
ln -s '/usr/lib/systemd/system/sshd.service' '/etc/systemd/system/mult
i-user.target.wants/sshd.service'
prompt$ systemctl status sshd.service
sshd.service - OpenSSH server daemon
    Loaded: loaded (/usr/lib/systemd/system/sshd.service; enabled)
    Active: inactive (dead)
    CGroup: name=systemd:/system/sshd.service

prompt$ █
```

systemctl example



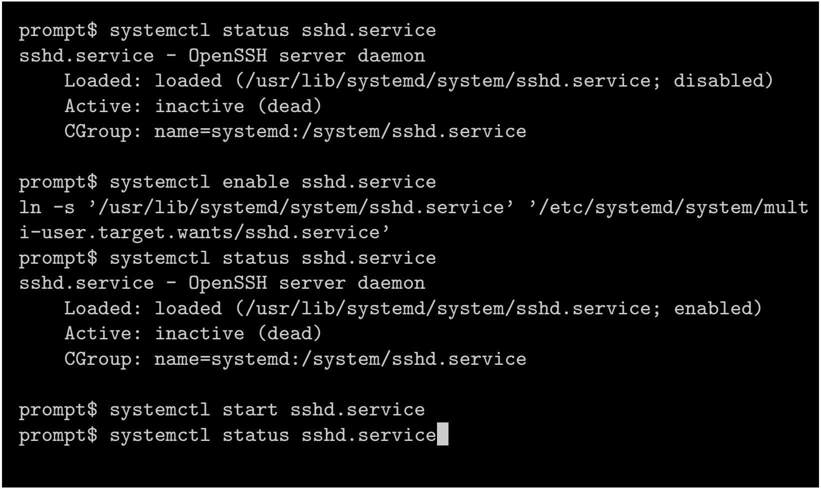
```
prompt$ systemctl status sshd.service
sshd.service - OpenSSH server daemon
    Loaded: loaded (/usr/lib/systemd/system/sshd.service; disabled)
    Active: inactive (dead)
    CGroup: name=systemd:/system/sshd.service

prompt$ systemctl enable sshd.service
ln -s ’/usr/lib/systemd/system/sshd.service’ ’/etc/systemd/system/mult
i-user.target.wants/sshd.service’
prompt$ systemctl status sshd.service
sshd.service - OpenSSH server daemon
    Loaded: loaded (/usr/lib/systemd/system/sshd.service; enabled)
    Active: inactive (dead)
    CGroup: name=systemd:/system/sshd.service

prompt$ systemctl start sshd.service
```

systemctl example



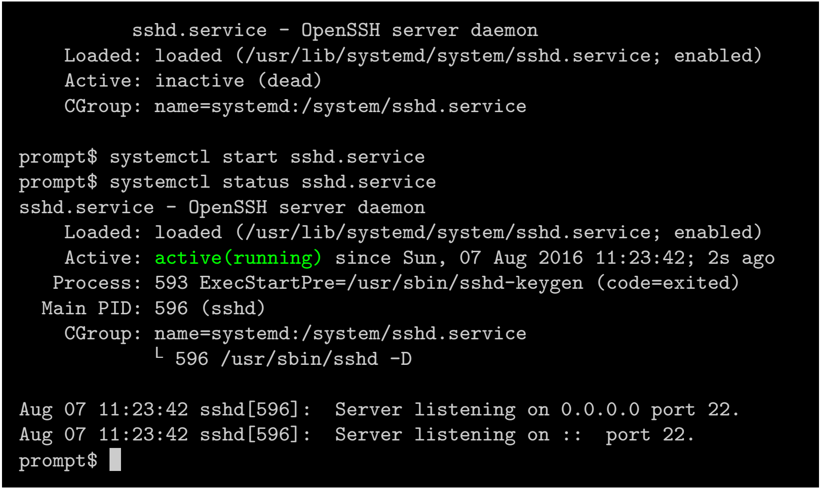
```
prompt$ systemctl status sshd.service
sshd.service - OpenSSH server daemon
    Loaded: loaded (/usr/lib/systemd/system/sshd.service; disabled)
    Active: inactive (dead)
    CGroup: name=systemd:/system/sshd.service

prompt$ systemctl enable sshd.service
ln -s ’/usr/lib/systemd/system/sshd.service’ ’/etc/systemd/system/mult
i-user.target.wants/sshd.service’
prompt$ systemctl status sshd.service
sshd.service - OpenSSH server daemon
    Loaded: loaded (/usr/lib/systemd/system/sshd.service; enabled)
    Active: inactive (dead)
    CGroup: name=systemd:/system/sshd.service

prompt$ systemctl start sshd.service
prompt$
```

systemctl example



```
prompt$ systemctl status sshd.service
sshd.service - OpenSSH server daemon
    Loaded: loaded (/usr/lib/systemd/system/sshd.service; disabled)
    Active: inactive (dead)
    CGroup: name=systemd:/system/sshd.service

prompt$ systemctl enable sshd.service
ln -s ’/usr/lib/systemd/system/sshd.service’ ’/etc/systemd/system/mult
i-user.target.wants/sshd.service’
prompt$ systemctl status sshd.service
sshd.service - OpenSSH server daemon
    Loaded: loaded (/usr/lib/systemd/system/sshd.service; enabled)
    Active: inactive (dead)
    CGroup: name=systemd:/system/sshd.service

prompt$ systemctl start sshd.service
prompt$ systemctl status sshd.service
```

systemctl example



```
          sshd.service - OpenSSH server daemon
    Loaded: loaded (/usr/lib/systemd/system/sshd.service; enabled)
    Active: inactive (dead)
    CGroup: name=systemd:/system/sshd.service

prompt$ systemctl start sshd.service
prompt$ systemctl status sshd.service
sshd.service - OpenSSH server daemon
    Loaded: loaded (/usr/lib/systemd/system/sshd.service; enabled)
    Active: active(running) since Sun, 07 Aug 2016 11:23:42; 2s ago
   Process: 593 ExecStartPre=/usr/sbin/sshd-keygen (code=exited)
  Main PID: 596 (sshd)
    CGroup: name=systemd:/system/sshd.service
            └ 596 /usr/sbin/sshd -D

Aug 07 11:23:42 sshd[596]:  Server listening on 0.0.0.0 port 22.
Aug 07 11:23:42 sshd[596]:  Server listening on ::  port 22.
prompt$ 
```

Exit status revisited

- Exit status is an integer returned by a process
- In UNIX: exit status is 8 bits
    - Values 0,. . . ,255
- Convention:
    - 0 means “success”
    - All other values mean some error occurred
    - Typically — different value used for each type of error
    - Some systems specify preferred values: try `man sysexits`
- Can we get the exit status of a command, in the shell?

Exit status revisited

- Exit status is an integer returned by a process
- In UNIX: exit status is 8 bits
  - Values 0,. . . ,255
- Convention:
  - 0 means "success"
  - All other values mean some error occurred
  - Typically — different value used for each type of error
  - Some systems specify preferred values: try `man sysexits`
- Can we get the exit status of a command, in the shell?
  - Yes, there are a few ways to do this
  - We will start with simple ways to determine success or failure
  - To get the actual status code — we will discuss later

Shell logic

```
cmd1; cmd2; cmd3
```

- Execute commands, in order

Shell logic

```
cmd1; cmd2; cmd3
```

- Execute commands, in order

```
cmd1 && cmd2 && cmd3
```

- Execute commands in order, stop after failure
- Succeeds if and only if all commands succeed

Shell logic

`cmd1; cmd2; cmd3`

- Execute commands, in order

`cmd1 && cmd2 && cmd3`

- Execute commands in order, stop after failure
- Succeeds if and only if all commands succeed

`cmd1 || cmd2 || cmd3`

- Execute commands in order, stop after success
- Fails if and only if all commands fail

Shell logic

```
cmd1; cmd2; cmd3
```

- Execute commands, in order

```
cmd1 && cmd2 && cmd3
```

- Execute commands in order, stop after failure
- Succeeds if and only if all commands succeed

```
cmd1 || cmd2 || cmd3
```

- Execute commands in order, stop after success
- Fails if and only if all commands fail

These can be nested with parentheses

Shell logic examples

```
prompt$ mkdir ~/backup && cp bigfile ~/backup
```

- Try to make a backup copy in directory `~/backup`
- If `mkdir` fails, then the file will not be copied

Shell logic examples

```
prompt$ mkdir ~/backup && cp bigfile ~/backup
```

- Try to make a backup copy in directory `~/backup`
- If `mkdir` fails, then the file will not be copied

```
prompt$ (mkdir ~/backup && cp bigfile ~/backup) ||
(mkdir /tmp/backup && cp bigfile /tmp/backup)
```

- First, try to make a backup in `~/backup`
- If that fails, then try `/tmp/backup`

So, how can I tell if a command succeeds?

So, how can I tell if a command succeeds?

```
cmd args && echo Success
```

- “Success” is printed if and only if `cmd args` succeeds

So, how can I tell if a command succeeds?

```
cmd args && echo Success
```

- “Success” is printed if and only if `cmd args` succeeds

```
cmd args || echo Failed
```

- “Failed” is printed if and only if `cmd args` fails

So, how can I tell if a command succeeds?

```
cmd args && echo Success
```

- "Success" is printed if and only if `cmd args` succeeds

```
cmd args || echo Failed
```

- "Failed" is printed if and only if `cmd args` fails

```
(cmd args && echo Success) || echo Failed
```

- If `cmd args` succeeds: prints "Success"
- If `cmd args` fails: prints "Failed"

Process files

- Each process has its own list of open files
- The list is indexed, starting at 0
- The indexes are called file descriptors
- The first three file descriptors are set aside in C as follows:

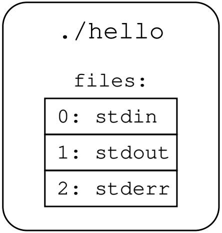
  0 : Standard input
  1 : Standard output
  2 : Standard error

  (You are not required to follow this, but there is no compelling reason not to.)
- For all commands we have seen so far:
  - Any “user input” is read from standard input
  - Any “ordinary” output is written to standard output
  - Any error messages, and some prompts, are written to standard error

File descriptor illustration

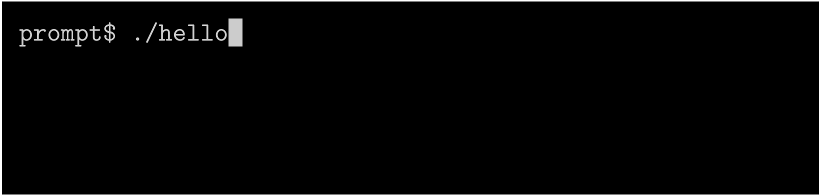
```
prompt$ 
```

Abstract view of the `./hello` process:



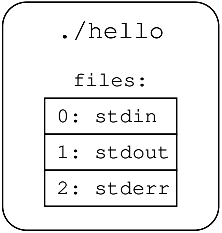
```
./hello

files:
0: stdin
1: stdout
2: stderr
```

File descriptor illustration



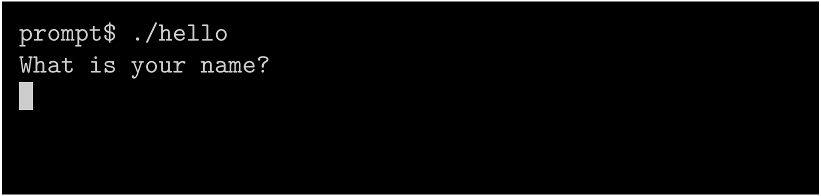
```
prompt$ ./hello
```

Abstract view of the `./hello` process:



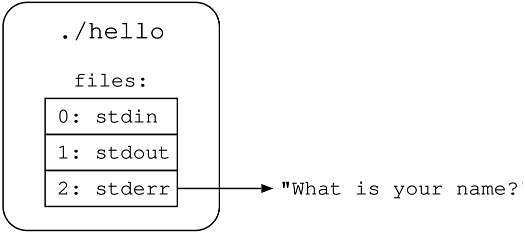
```
./hello

files:
0: stdin
1: stdout
2: stderr
```

File descriptor illustration

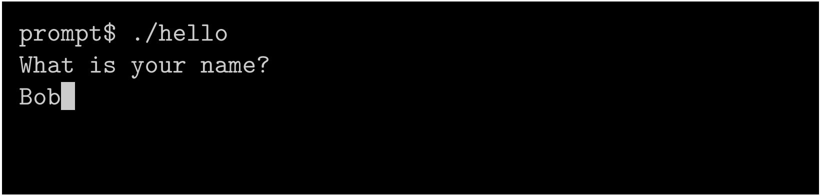


```
prompt$ ./hello
What is your name?
█
```

Abstract view of the `./hello` process:

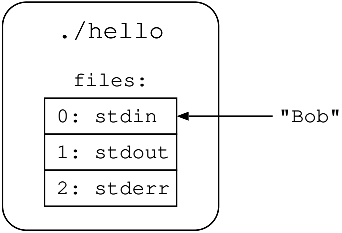


File descriptor illustration

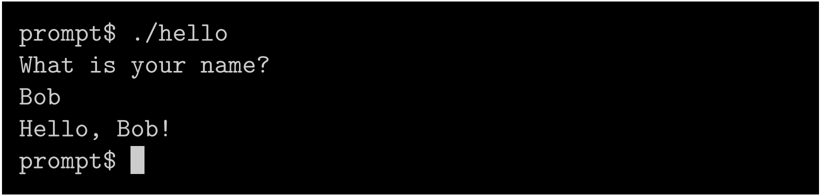


```
prompt$ ./hello
What is your name?
Bob
```

Abstract view of the `./hello` process:

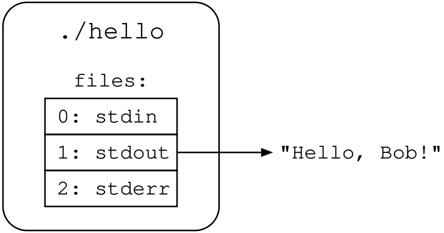


File descriptor illustration



```
prompt$ ./hello
What is your name?
Bob
Hello, Bob!
prompt$ █
```

Abstract view of the `./hello` process:



Redirection

In the shell, we can change the file descriptors around:

`command args < file` : `stdin` reads from file

`command args > file` : `stdout` writes to file

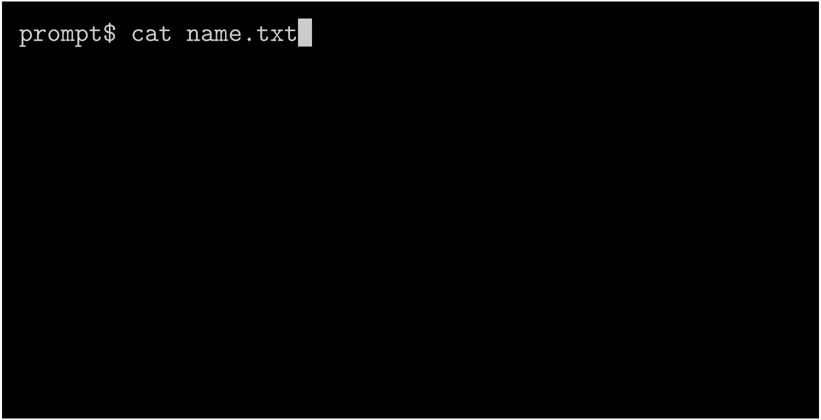
```
prompt$ █
```

Redirection

In the shell, we can change the file descriptors around:

`command args < file` : `stdin` reads from file

`command args > file` : `stdout` writes to file



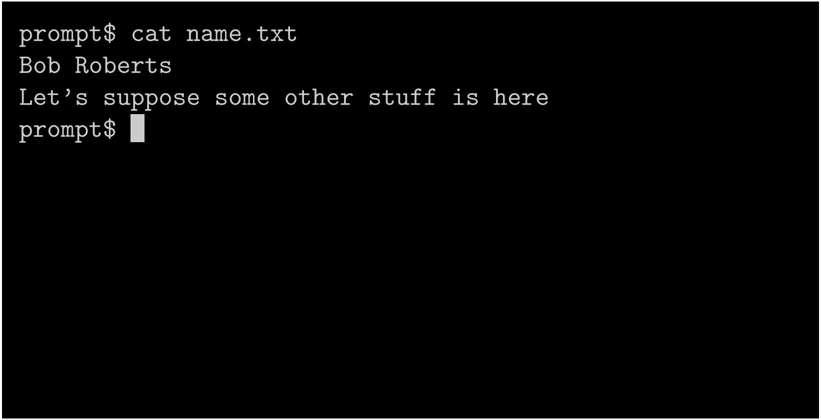
```
prompt$ cat name.txt
```

Redirection

In the shell, we can change the file descriptors around:

`command args < file` : stdin reads from file

`command args > file` : stdout writes to file



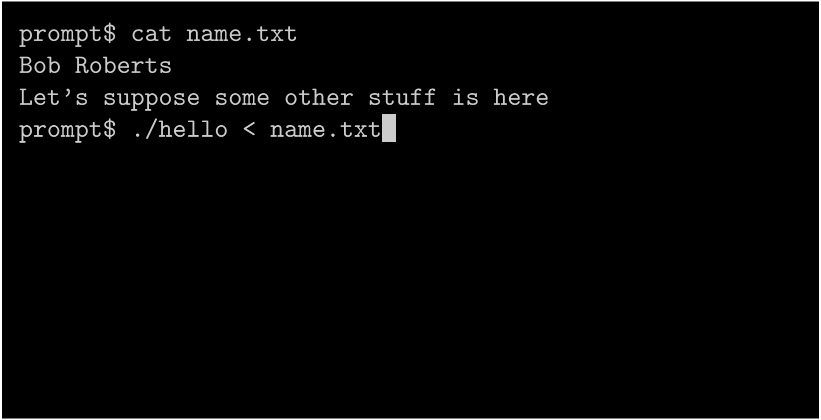
```
prompt$ cat name.txt
Bob Roberts
Let's suppose some other stuff is here
prompt$ █
```

Redirection

In the shell, we can change the file descriptors around:

`command args < file` : `stdin` reads from file

`command args > file` : `stdout` writes to file



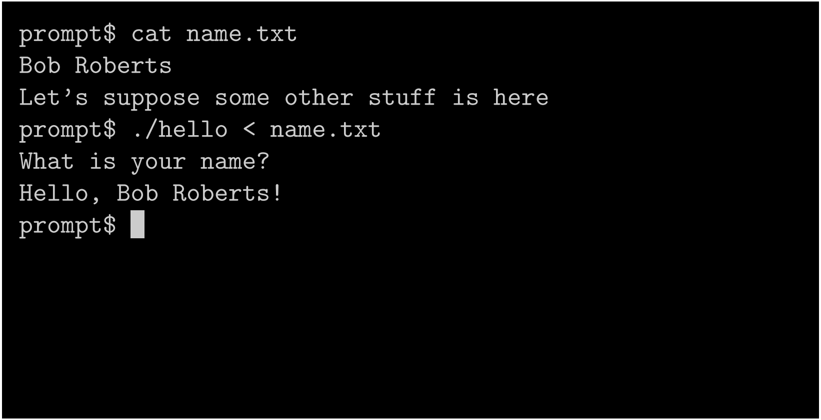
```
prompt$ cat name.txt
Bob Roberts
Let's suppose some other stuff is here
prompt$ ./hello < name.txt
```

Redirection

In the shell, we can change the file descriptors around:

`command args < file` : stdin reads from file

`command args > file` : stdout writes to file



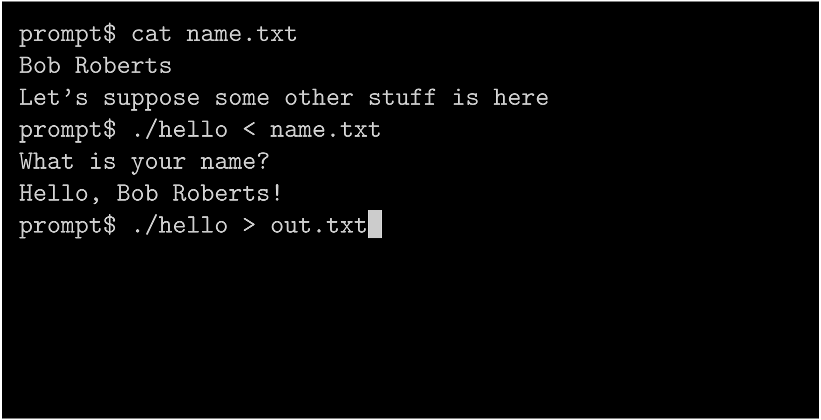
```
prompt$ cat name.txt
Bob Roberts
Let's suppose some other stuff is here
prompt$ ./hello < name.txt
What is your name?
Hello, Bob Roberts!
prompt$ █
```

Redirection

In the shell, we can change the file descriptors around:

`command args < file` : stdin reads from file

`command args > file` : stdout writes to file



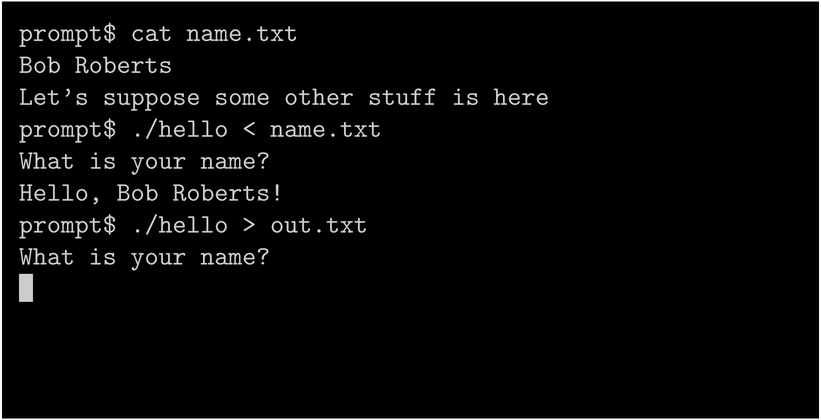
```
prompt$ cat name.txt
Bob Roberts
Let's suppose some other stuff is here
prompt$ ./hello < name.txt
What is your name?
Hello, Bob Roberts!
prompt$ ./hello > out.txt
```

Redirection

In the shell, we can change the file descriptors around:

`command args < file` : `stdin` reads from file

`command args > file` : `stdout` writes to file



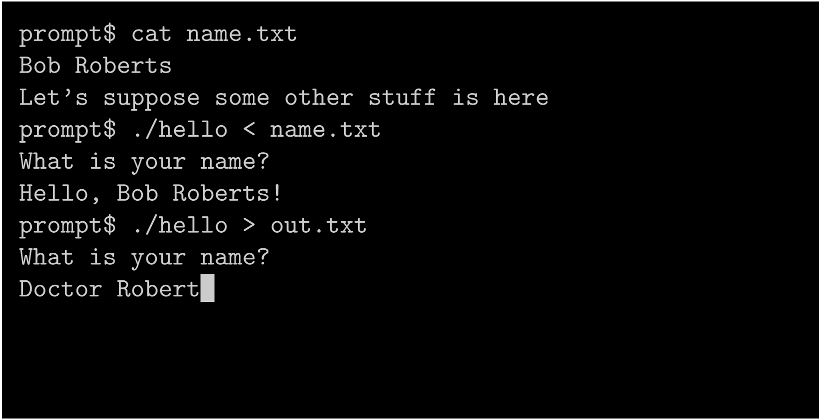
```
prompt$ cat name.txt
Bob Roberts
Let's suppose some other stuff is here
prompt$ ./hello < name.txt
What is your name?
Hello, Bob Roberts!
prompt$ ./hello > out.txt
What is your name?
```

Redirection

In the shell, we can change the file descriptors around:

`command args < file` : stdin reads from file

`command args > file` : stdout writes to file



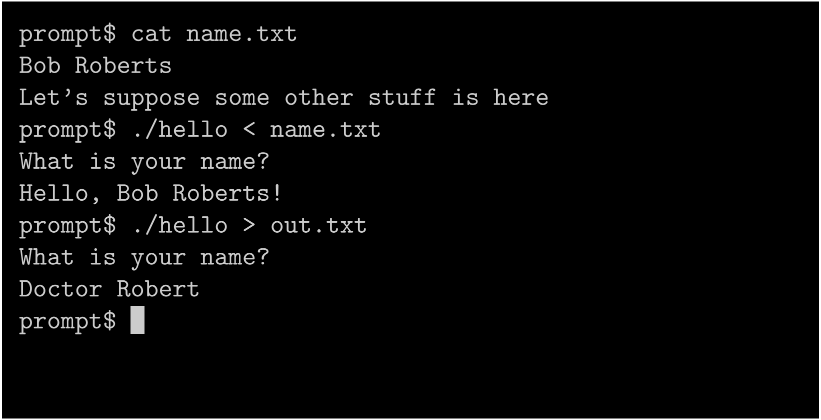
```
prompt$ cat name.txt
Bob Roberts
Let's suppose some other stuff is here
prompt$ ./hello < name.txt
What is your name?
Hello, Bob Roberts!
prompt$ ./hello > out.txt
What is your name?
Doctor Robert
```

Redirection

In the shell, we can change the file descriptors around:

`command args < file` : `stdin` reads from file

`command args > file` : `stdout` writes to file



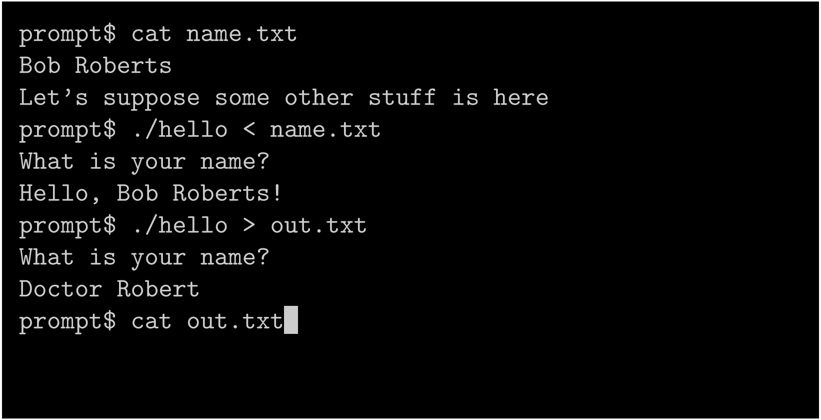
```
prompt$ cat name.txt
Bob Roberts
Let's suppose some other stuff is here
prompt$ ./hello < name.txt
What is your name?
Hello, Bob Roberts!
prompt$ ./hello > out.txt
What is your name?
Doctor Robert
prompt$ █
```

Redirection

In the shell, we can change the file descriptors around:

`command args < file` : stdin reads from file

`command args > file` : stdout writes to file



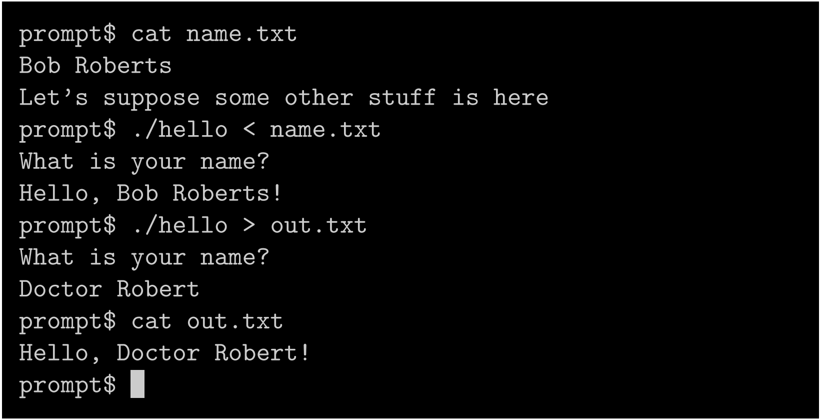
```
prompt$ cat name.txt
Bob Roberts
Let's suppose some other stuff is here
prompt$ ./hello < name.txt
What is your name?
Hello, Bob Roberts!
prompt$ ./hello > out.txt
What is your name?
Doctor Robert
prompt$ cat out.txt
```

Redirection

In the shell, we can change the file descriptors around:

`command args < file` : stdin reads from file

`command args > file` : stdout writes to file



```
prompt$ cat name.txt
Bob Roberts
Let's suppose some other stuff is here
prompt$ ./hello < name.txt
What is your name?
Hello, Bob Roberts!
prompt$ ./hello > out.txt
What is your name?
Doctor Robert
prompt$ cat out.txt
Hello, Doctor Robert!
prompt$
```

Some redirection questions

Can I redirect both `stdin` and `stdout`?

Some redirection questions

Can I redirect both stdin and stdout?

Of course:

```
prompt$ command args < infile > outfile
```

Or you can use:

```
prompt$ command args > outfile < infile
```

(Order does not matter)

Some redirection questions

Can I redirect both stdin and stdout?

Of course:

```
prompt$ command args < infile > outfile
```

Or you can use:

```
prompt$ command args > outfile < infile
```

(Order does not matter)

What if I send stdout to an existing file?

Some redirection questions

Can I redirect both stdin and stdout?

Of course:

```
prompt$ command args < infile > outfile
```

Or you can use:

```
prompt$ command args > outfile < infile
```

(Order does not matter)

What if I send stdout to an existing file?

- Depends on shell settings
- Default is to "clobber" (overwrite) the file

What about `stderr`?

A couple of options, depending on what you want:

- `0< file` : `stdin` reads from file (same as <)
- `1> file` : `stdout` writes to file (same as >)
- `2> file` : `stderr` writes to file

Another option:

- `2>&1` : send `stderr` to where `stdout` is currently going (order is important here)

```
prompt$ 
```

What about stderr?

A couple of options, depending on what you want:

- `0< file` : `stdin` reads from file (same as <)
- `1> file` : `stdout` writes to file (same as >)
- `2> file` : `stderr` writes to file

Another option:

- `2>&1` : send `stderr` to where `stdout` is currently going (order is important here)

```
prompt$ ./hello 2> out.txt
```

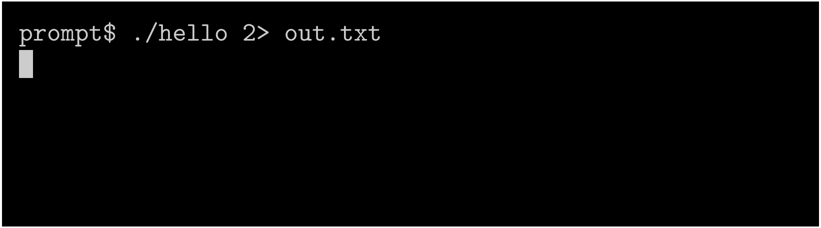
What about stderr?

A couple of options, depending on what you want:

- `0< file` : stdin reads from file (same as <)
- `1> file` : stdout writes to file (same as >)
- `2> file` : stderr writes to file

Another option:

- `2>&1` : send stdout to where stdout is currently going (order is important here)



```
prompt$ ./hello 2> out.txt
█
```

What about stderr?

A couple of options, depending on what you want:

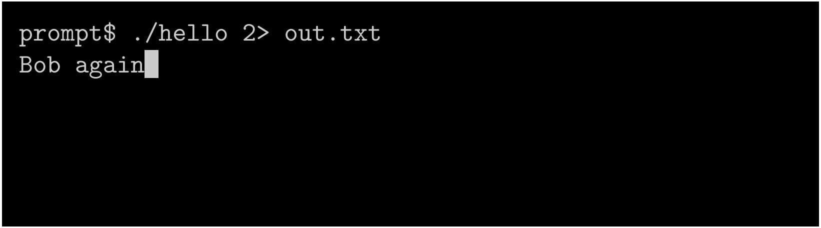
`0< file` : stdin reads from file (same as <)

`1> file` : stdout writes to file (same as >)

`2> file` : stderr writes to file

Another option:

`2>&1` : send stderr to where stdout is currently going (order is important here)



```
prompt$ ./hello 2> out.txt
Bob again
```

What about stderr?

A couple of options, depending on what you want:

- `0< file` : `stdin` reads from file (same as <)
- `1> file` : `stdout` writes to file (same as >)
- `2> file` : `stderr` writes to file

Another option:

`2>&1` : send `stderr` to where `stdout` is currently going (order is important here)

```
prompt$ ./hello 2> out.txt
Bob again
Hello, Bob again!
prompt$ █
```

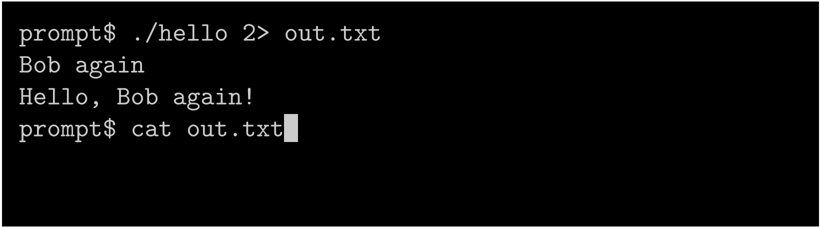
What about stderr?

A couple of options, depending on what you want:

- `0< file` : stdin reads from file (same as <)
- `1> file` : stdout writes to file (same as >)
- `2> file` : stderr writes to file

Another option:

- `2>&1` : send stderr to where stdout is currently going (order is important here)



```
prompt$ ./hello 2> out.txt
Bob again
Hello, Bob again!
prompt$ cat out.txt
```

What about stderr?

A couple of options, depending on what you want:

- 0< file : stdin reads from file (same as <)
- 1> file : stdout writes to file (same as >)
- 2> file : stderr writes to file

Another option:

- 2>&1 : send stderr to where stdout is currently going (order is important here)

```
prompt$ ./hello 2> out.txt
Bob again
Hello, Bob again!
prompt$ cat out.txt
What is your name?
prompt$ █
```

Fun with >&

```
cmd > out.txt 2>&1
```

```
cmd 2>&1 > out.txt
```

```
cmd > outA.txt 2>&1 > outB.txt
```

```
cmd 2> outA.txt 1>&2 2> outB.txt
```

Fun with >&

```
cmd > out.txt 2>&1
```

Both `stdout` and `stderr` go to `out.txt`

```
cmd 2>&1 > out.txt
```

```
cmd > outA.txt 2>&1 > outB.txt
```

```
cmd 2> outA.txt 1>&2 2> outB.txt
```

Fun with >&

```
cmd > out.txt 2>&1
```

Both `stdout` and `stderr` go to `out.txt`

```
cmd 2>&1 > out.txt
```

`stdout` goes to `out.txt`, `stderr` goes to terminal

```
cmd > outA.txt 2>&1 > outB.txt
```

```
cmd 2> outA.txt 1>&2 2> outB.txt
```

Fun with >&

```
cmd > out.txt 2>&1
```

Both `stdout` and `stderr` go to `out.txt`

```
cmd 2>&1 > out.txt
```

`stdout` goes to `out.txt`, `stderr` goes to terminal

```
cmd > outA.txt 2>&1 > outB.txt
```

`stdout` goes to `outB.txt`, `stderr` goes to `outA.txt`

```
cmd 2> outA.txt 1>&2 2> outB.txt
```

Fun with >&

```
cmd > out.txt 2>&1
```

Both stdout and stderr go to out.txt

```
cmd 2>&1 > out.txt
```

stdout goes to out.txt, stderr goes to terminal

```
cmd > outA.txt 2>&1 > outB.txt
```

stdout goes to outB.txt, stderr goes to outA.txt

```
cmd 2> outA.txt 1>&2 2> outB.txt
```

stdout goes to outA.txt, stderr goes to outB.txt

Using devices

- Can I get input from, and send output to, a device?

Using devices

- Can I get input from, and send output to, a device?
  - Of course. Devices are files, remember?

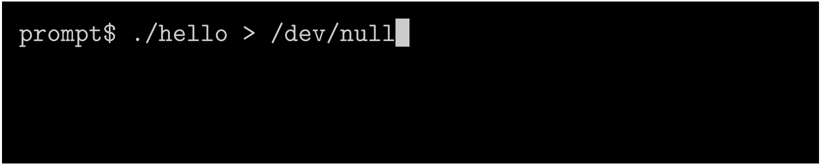
Using devices

- Can I get input from, and send output to, a device?
  - Of course. Devices are files, remember?

```
prompt$ █
```

Using devices

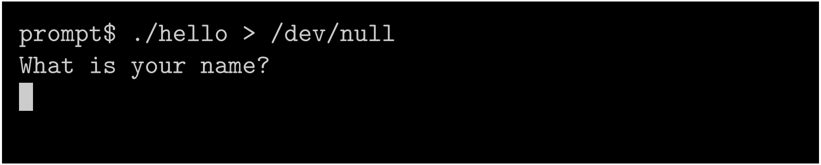
- Can I get input from, and send output to, a device?
  - Of course. Devices are files, remember?



```
prompt$ ./hello > /dev/null
```

Using devices

- Can I get input from, and send output to, a device?
    - Of course. Devices are files, remember?



```
prompt$ ./hello > /dev/null
What is your name?
```

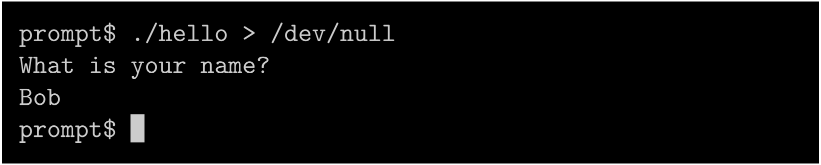
Using devices

- Can I get input from, and send output to, a device?
  - Of course. Devices are files, remember?

```
prompt$ ./hello > /dev/null
What is your name?
Bob
```

Using devices

- Can I get input from, and send output to, a device?
    - Of course. Devices are files, remember?



```
prompt$ ./hello > /dev/null
What is your name?
Bob
prompt$ 
```

Using devices

- Can I get input from, and send output to, a device?
  - Of course. Devices are files, remember?

```
prompt$ ./hello > /dev/null
What is your name?
Bob
prompt$
```

- What happened to “Hello, Bob!”?

Using devices

- Can I get input from, and send output to, a device?
  - Of course. Devices are files, remember?

```
prompt$ ./hello > /dev/null
What is your name?
Bob
prompt$
```

- What happened to “Hello, Bob!”?
  - It went to `/dev/null`, which discards everything

Using devices

- Can I get input from, and send output to, a device?
  - Of course. Devices are files, remember?

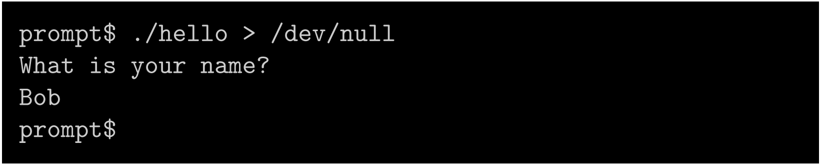
```
prompt$ ./hello > /dev/null
What is your name?
Bob
prompt$
```

- What happened to “Hello, Bob!”?
  - It went to `/dev/null`, which discards everything

```
prompt$ 
```

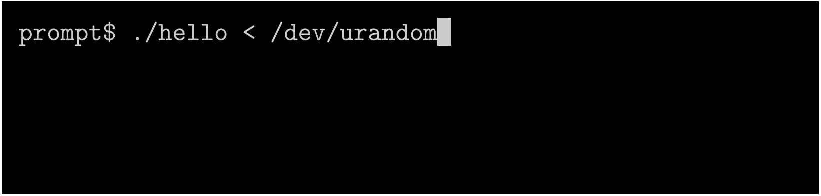
Using devices

- Can I get input from, and send output to, a device?
  - Of course. Devices are files, remember?



```
prompt$ ./hello > /dev/null
What is your name?
Bob
prompt$
```

- What happened to “Hello, Bob!”?
  - It went to `/dev/null`, which discards everything



```
prompt$ ./hello < /dev/urandom
```

Using devices

- Can I get input from, and send output to, a device?
  - Of course. Devices are files, remember?

```
prompt$ ./hello > /dev/null
What is your name?
Bob
prompt$
```

- What happened to "Hello, Bob!"?
  - It went to `/dev/null`, which discards everything

```
prompt$ ./hello < /dev/urandom
What is your name?
Hello,?[?QP??S? ???<yH
                      ?-?Q ??p??θ????2*?i!
prompt$ █
```

Adding to a file

- It is possible to append to a file

  `>> file` : stdout appends to file

  `1>> file` : stdout appends to file

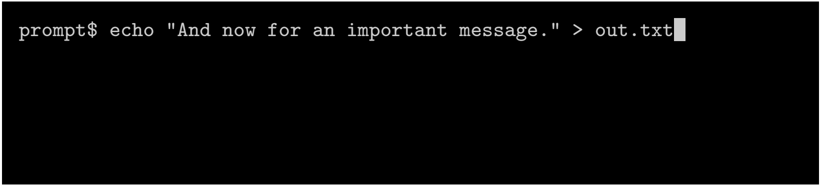
  `2>> file` : stderr appends to file

- If file does not exist already:
  - Depends on shell and settings
  - May complain
  - May create an empty file (act like >)

```
prompt$
```

Adding to a file

- It is possible to append to a file
  - `>> file` : stdout appends to file
  - `1>> file` : stdout appends to file
  - `2>> file` : stderr appends to file
- If file does not exist already:
  - Depends on shell and settings
  - May complain
  - May create an empty file (act like >)



```
prompt$ echo "And now for an important message." > out.txt
```

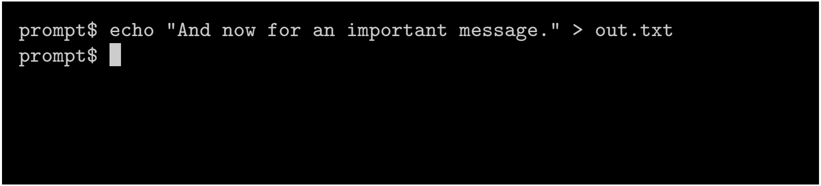
Adding to a file

- It is possible to append to a file

  `>> file` : stdout appends to file

  `1>> file` : stdout appends to file

  `2>> file` : stderr appends to file
- If file does not exist already:
  - Depends on shell and settings
  - May complain
  - May create an empty file (act like >)



```
prompt$ echo "And now for an important message." > out.txt
prompt$ █
```

Adding to a file

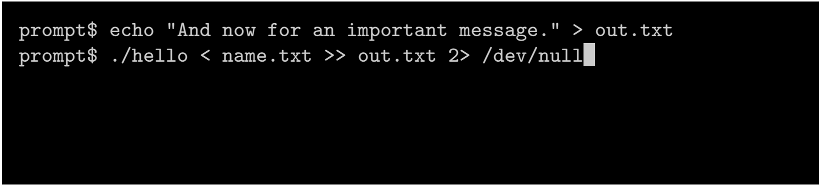
- It is possible to append to a file

  `>> file` : stdout appends to file

  `1>> file` : stdout appends to file

  `2>> file` : stderr appends to file

- If file does not exist already:
  - Depends on shell and settings
  - May complain
  - May create an empty file (act like >)



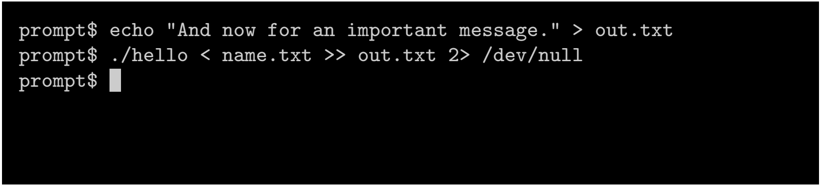
```
prompt$ echo "And now for an important message." > out.txt
prompt$ ./hello < name.txt >> out.txt 2> /dev/null
```

Adding to a file

- It is possible to append to a file

  `>> file` : stdout appends to file
  `1>> file` : stdout appends to file
  `2>> file` : stderr appends to file

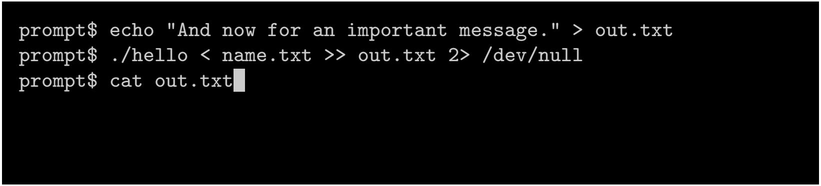
- If file does not exist already:
  - Depends on shell and settings
  - May complain
  - May create an empty file (act like >)



```
prompt$ echo "And now for an important message." > out.txt
prompt$ ./hello < name.txt >> out.txt 2> /dev/null
prompt$ 
```

Adding to a file

- It is possible to append to a file
  - `>> file` : stdout appends to file
  - `1>> file` : stdout appends to file
  - `2>> file` : stderr appends to file
- If file does not exist already:
  - Depends on shell and settings
  - May complain
  - May create an empty file (act like >)



```
prompt$ echo "And now for an important message." > out.txt
prompt$ ./hello < name.txt >> out.txt 2> /dev/null
prompt$ cat out.txt
```

Adding to a file

- It is possible to append to a file

  ` >> file` : stdout appends to file

  `1>> file` : stdout appends to file

  `2>> file` : stderr appends to file

- If file does not exist already:
  - Depends on shell and settings
  - May complain
  - May create an empty file (act like >)

```
prompt$ echo "And now for an important message." > out.txt
prompt$ ./hello < name.txt >> out.txt 2> /dev/null
prompt$ cat out.txt
And now for an important message.
Hello, Bob Roberts!
prompt$ █
```

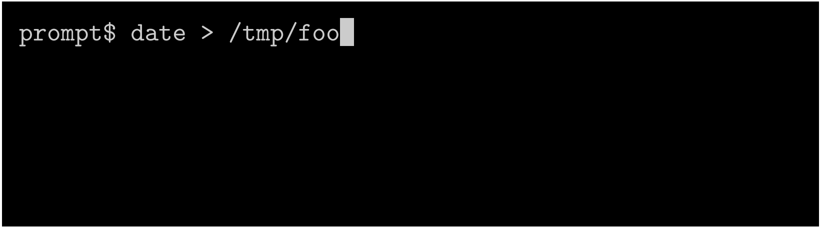
One command’s output as another command’s input

- How to set this up using redirection?

```
prompt$ █
```

One command's output as another command's input

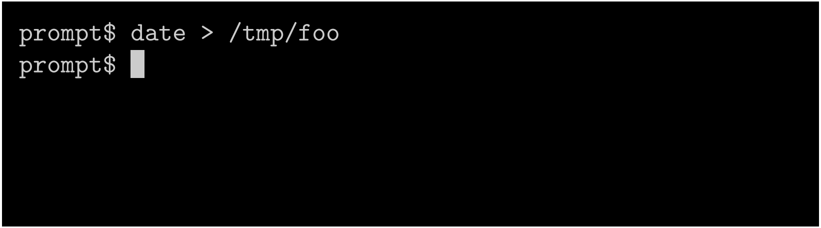
- How to set this up using redirection?
- Send output of first command to a temporary file



```
prompt$ date > /tmp/foo
```

One command's output as another command's input

- How to set this up using redirection?
- Send output of first command to a temporary file



```
prompt$ date > /tmp/foo
prompt$ █
```

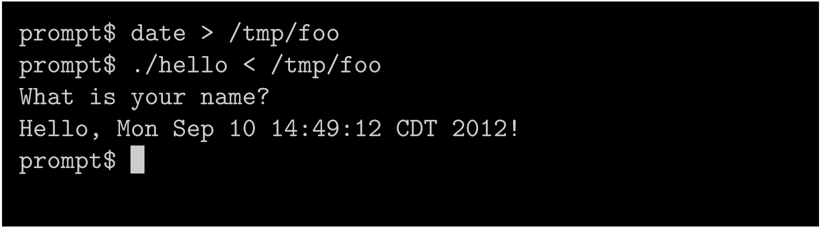
One command’s output as another command’s input

- How to set this up using redirection?
- Send output of first command to a temporary file
- Run second command, reading from the temporary file

```
prompt$ date > /tmp/foo
prompt$ ./hello < /tmp/foo
```

One command's output as another command's input

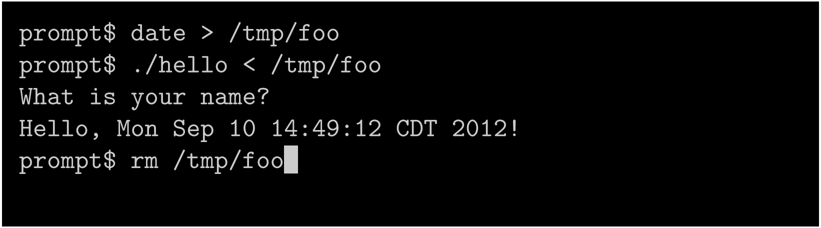
- How to set this up using redirection?
- Send output of first command to a temporary file
- Run second command, reading from the temporary file



```
prompt$ date > /tmp/foo
prompt$ ./hello < /tmp/foo
What is your name?
Hello, Mon Sep 10 14:49:12 CDT 2012!
prompt$ █
```

One command's output as another command's input

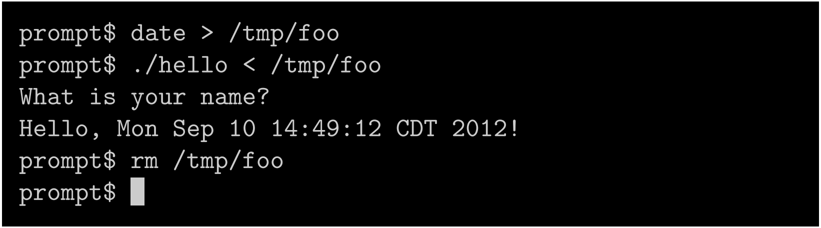
- How to set this up using redirection?
- Send output of first command to a temporary file
- Run second command, reading from the temporary file
- Discard the file



```
prompt$ date > /tmp/foo
prompt$ ./hello < /tmp/foo
What is your name?
Hello, Mon Sep 10 14:49:12 CDT 2012!
prompt$ rm /tmp/foo
```

One command's output as another command's input

- How to set this up using redirection?
- Send output of first command to a temporary file
- Run second command, reading from the temporary file
- Discard the file
- There is a much nicer way to do this. . .



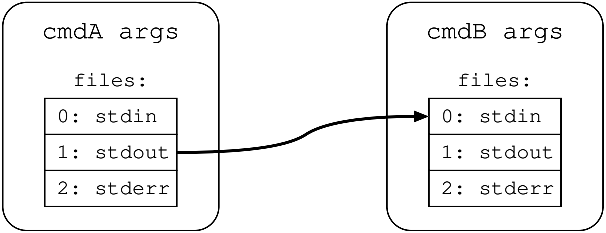
```
prompt$ date > /tmp/foo
prompt$ ./hello < /tmp/foo
What is your name?
Hello, Mon Sep 10 14:49:12 CDT 2012!
prompt$ rm /tmp/foo
prompt$ █
```

What is a pipe?

- In a shell, a pipe sends the output of one command *directly* as input to another command
- To set this up:

```
prompt$ cmdA args | cmdB args
```

- The shell uses two processes for this



cmdA args

files:

0: stdin
1: stdout
2: stderr

cmdB args

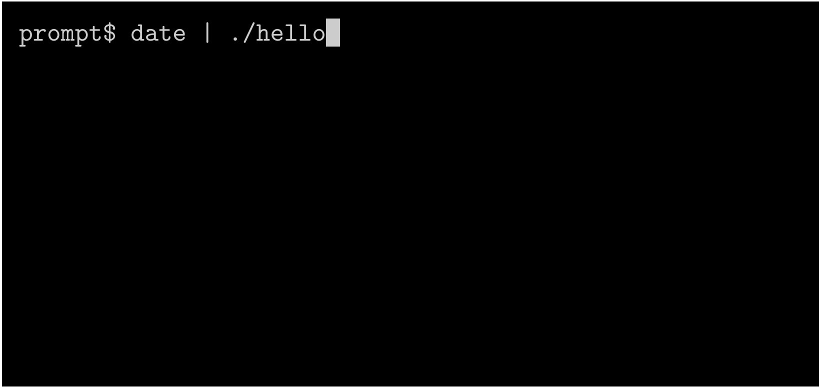
files:

0: stdin
1: stdout
2: stderr

Simple pipe examples

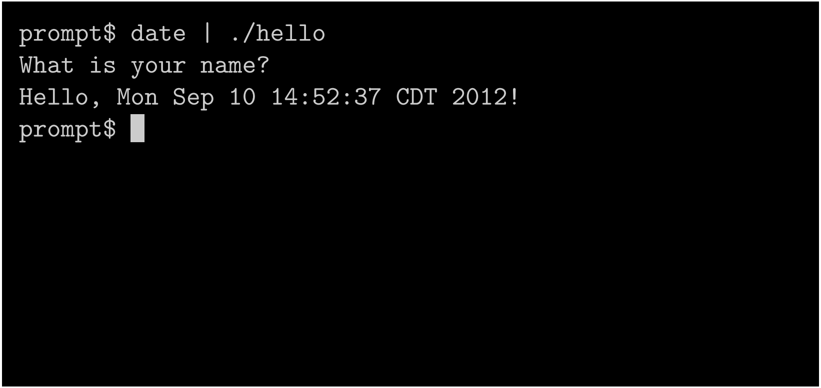
```
prompt$ █
```

Simple pipe examples



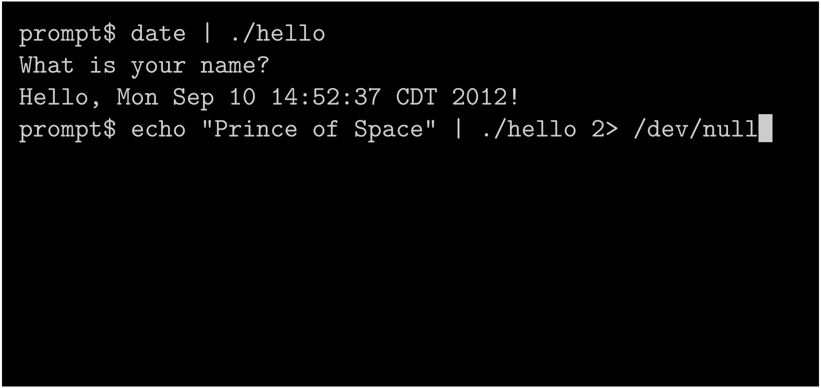
```
prompt$ date | ./hello
```

Simple pipe examples



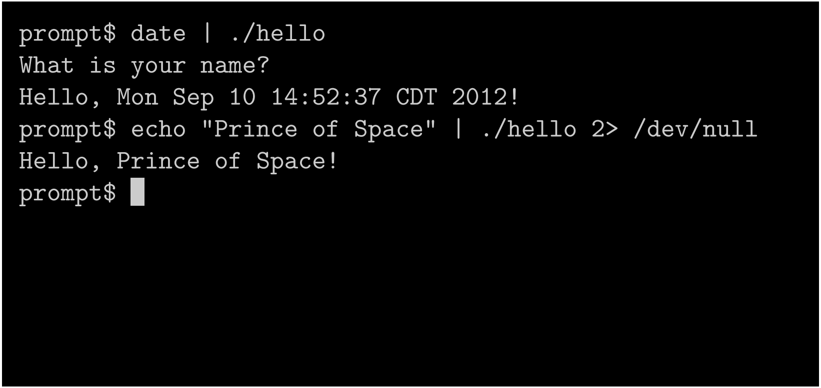
```
prompt$ date | ./hello
What is your name?
Hello, Mon Sep 10 14:52:37 CDT 2012!
prompt$ █
```

Simple pipe examples



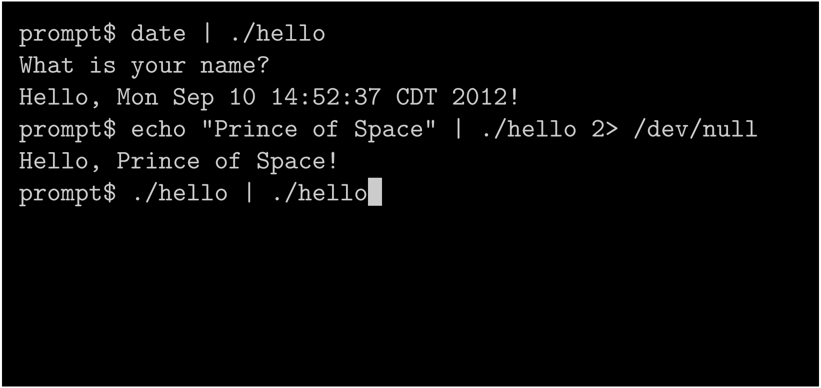
```
prompt$ date | ./hello
What is your name?
Hello, Mon Sep 10 14:52:37 CDT 2012!
prompt$ echo "Prince of Space" | ./hello 2> /dev/null
```

Simple pipe examples



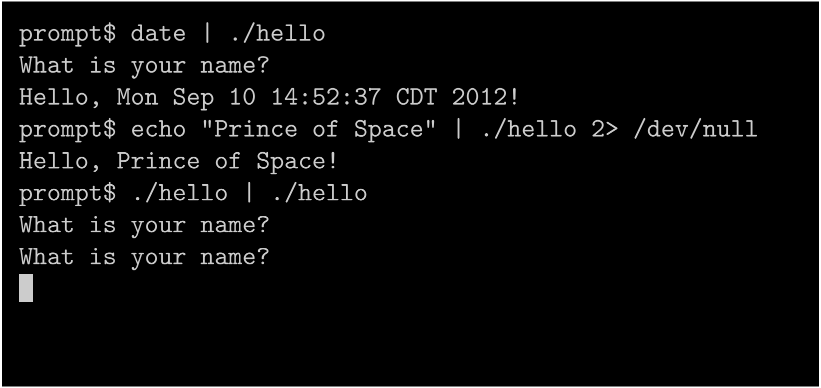
```
prompt$ date | ./hello
What is your name?
Hello, Mon Sep 10 14:52:37 CDT 2012!
prompt$ echo "Prince of Space" | ./hello 2> /dev/null
Hello, Prince of Space!
prompt$ █
```

Simple pipe examples



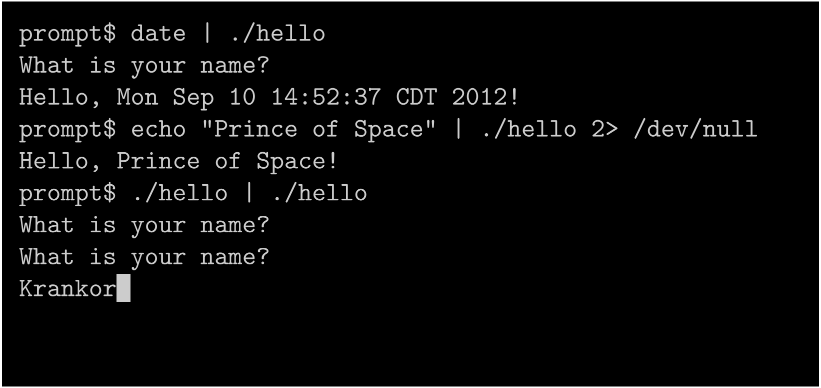
```
prompt$ date | ./hello
What is your name?
Hello, Mon Sep 10 14:52:37 CDT 2012!
prompt$ echo "Prince of Space" | ./hello 2> /dev/null
Hello, Prince of Space!
prompt$ ./hello | ./hello
```

Simple pipe examples



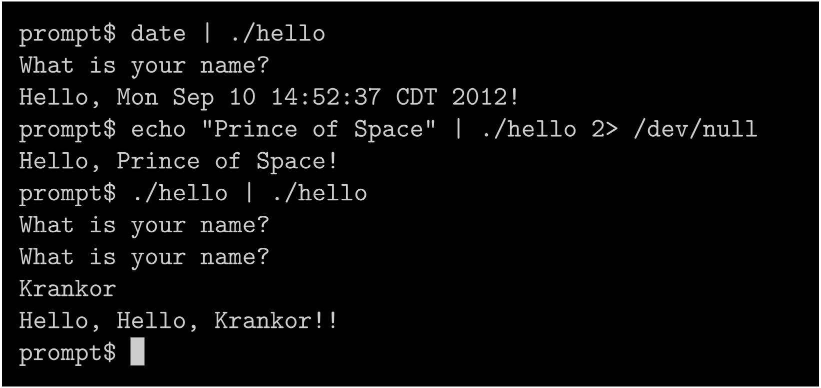
```
prompt$ date | ./hello
What is your name?
Hello, Mon Sep 10 14:52:37 CDT 2012!
prompt$ echo "Prince of Space" | ./hello 2> /dev/null
Hello, Prince of Space!
prompt$ ./hello | ./hello
What is your name?
What is your name?
```

Simple pipe examples



```
prompt$ date | ./hello
What is your name?
Hello, Mon Sep 10 14:52:37 CDT 2012!
prompt$ echo "Prince of Space" | ./hello 2> /dev/null
Hello, Prince of Space!
prompt$ ./hello | ./hello
What is your name?
What is your name?
Krankor
```

Simple pipe examples

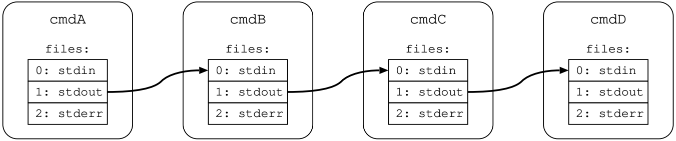


```
prompt$ date | ./hello
What is your name?
Hello, Mon Sep 10 14:52:37 CDT 2012!
prompt$ echo "Prince of Space" | ./hello 2> /dev/null
Hello, Prince of Space!
prompt$ ./hello | ./hello
What is your name?
What is your name?
Krankor
Hello, Hello, Krankor!!
prompt$ █
```

Pipelines

- We can connect a pipe at “both ends” of a process:

```
prompt$ cmdA | cmdB | cmdC | cmdD
```



cmdA
files:
0: stdin
1: stdout
2: stderr

cmdB
files:
0: stdin
1: stdout
2: stderr

cmdC
files:
0: stdin
1: stdout
2: stderr

cmdD
files:
0: stdin
1: stdout
2: stderr

- No fundamental limit on length of a pipeline chain
- Practical limits dictated by:
  - System memory available (to hold processes)
  - Number of allowed processes
  - Number of input characters per line allowed by shell
  - User ability

Deep questions about pipes

```
prompt$ foo | bar
```

- We have two processes, `foo` and `bar`
- We have no idea how the kernel will schedule these to execute
- We have no idea if someone will `kill` one of these processes

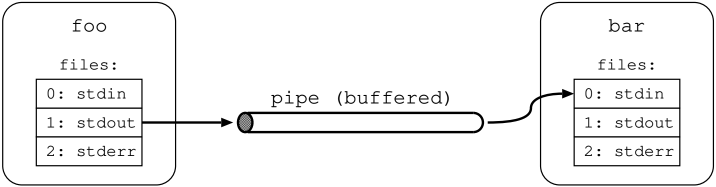
Fun questions:

1. What if `foo` writes, but `bar` isn't ready to read?
2. What if `bar` reads, but `foo` isn't ready to write?
3. What if `foo` terminates first?
4. What if `bar` terminates first?

1. What if foo writes before bar is ready

1. What if `foo` writes before `bar` is ready

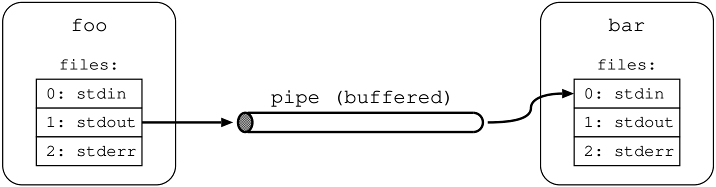
The pipe is *buffered*:



- If the pipe is not full, `foo` will write into the pipe
- If the pipe is full, `foo` will block until it can write
    - Just like writing to a device that is busy

2. What if `bar` reads before `foo` is ready

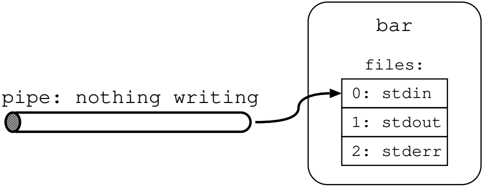
2. What if bar reads before foo is ready



- If there is “enough” data in the pipe, bar can read it
- If there is not enough in the pipe, bar will block
  - Just like waiting for user input

3. What if `foo` terminates

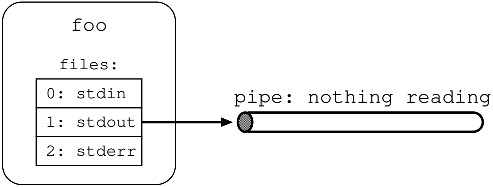
3. What if `foo` terminates



- `bar` can continue to read from the pipe
- When the pipe is empty, acts like end of file
  - This happens only when nothing can write into the pipe
  - `bar` needs to check for EOF anyway

4. What if bar terminates

4. What if bar terminates



- If foo writes to the pipe, kernel sends a "broken pipe" signal
- foo can catch this, or default behavior is to terminate
  - Terminate makes sense, usually:
    nobody will see anything else written by foo,
    no sense continuing the computation

Answers to motivating questions

1. How does a job know where and how to display its output?

Answers to motivating questions

1. How does a job know where and how to display its output?

   Actually, it doesn't know. It just writes to `stdout` and `stderr`.

Answers to motivating questions

1. How does a job know where and how to display its output?

   Actually, it doesn't know. It just writes to `stdout` and `stderr`.
2. When I run multiple jobs, why don't they clobber each other?

Answers to motivating questions

1. How does a job know where and how to display its output?

   Actually, it doesn't know. It just writes to `stdout` and `stderr`.

2. When I run multiple jobs, why don't they clobber each other?

   Each job becomes a process with its own memory space.

Answers to motivating questions

1. How does a job know where and how to display its output?

   Actually, it doesn't know. It just writes to `stdout` and `stderr`.

2. When I run multiple jobs, why don't they clobber each other?

   Each job becomes a process with its own memory space.

3. What happens to a job if its shell terminates?

Answers to motivating questions

1. How does a job know where and how to display its output?

   Actually, it doesn't know. It just writes to `stdout` and `stderr`.
2. When I run multiple jobs, why don't they clobber each other?

   Each job becomes a process with its own memory space.
3. What happens to a job if its shell terminates?

   It depends. Typically, stopped jobs will terminate, and running background jobs will keep running.

Answers to motivating questions

1. How does a job know where and how to display its output?

   Actually, it doesn't know. It just writes to `stdout` and `stderr`.

2. When I run multiple jobs, why don't they clobber each other?

   Each job becomes a process with its own memory space.

3. What happens to a job if its shell terminates?

   It depends. Typically, stopped jobs will terminate, and running background jobs will keep running.

4. How can I control a job from another shell?

Answers to motivating questions

1. How does a job know where and how to display its output?

   Actually, it doesn't know. It just writes to `stdout` and `stderr`.

2. When I run multiple jobs, why don't they clobber each other?

   Each job becomes a process with its own memory space.

3. What happens to a job if its shell terminates?

   It depends. Typically, stopped jobs will terminate, and running background jobs will keep running.

4. How can I control a job from another shell?

   Find its process ID and control the process directly.

Answers to motivating questions

1. How does a job know where and how to display its output?

   Actually, it doesn't know. It just writes to `stdout` and `stderr`.

2. When I run multiple jobs, why don't they clobber each other?

   Each job becomes a process with its own memory space.

3. What happens to a job if its shell terminates?

   It depends. Typically, stopped jobs will terminate, and running background jobs will keep running.

4. How can I control a job from another shell?

   Find its process ID and control the process directly.

5. Can I tell if a job finished successfully?

Answers to motivating questions

1. How does a job know where and how to display its output?

   Actually, it doesn't know. It just writes to `stdout` and `stderr`.

2. When I run multiple jobs, why don't they clobber each other?

   Each job becomes a process with its own memory space.

3. What happens to a job if its shell terminates?

   It depends. Typically, stopped jobs will terminate, and running background jobs will keep running.

4. How can I control a job from another shell?

   Find its process ID and control the process directly.

5. Can I tell if a job finished successfully?

   Yes, by checking the exit status. This can only be done in the same shell.

`date`	:	Display the current date and time
`exit`	:	Exit a shell
`kill`	:	Signal a process
`nice`	:	Run with lower priority
`ps`	:	List processes
`renice`	:	Adjust priority of a process
`systemctl`	:	Manage services (using `systemd`)
`wait`	:	Wait for one or more processes

End of lecture