Introduction to UNIX

CS2023 Winter 2003

Outcomes: Introduction to UNIX

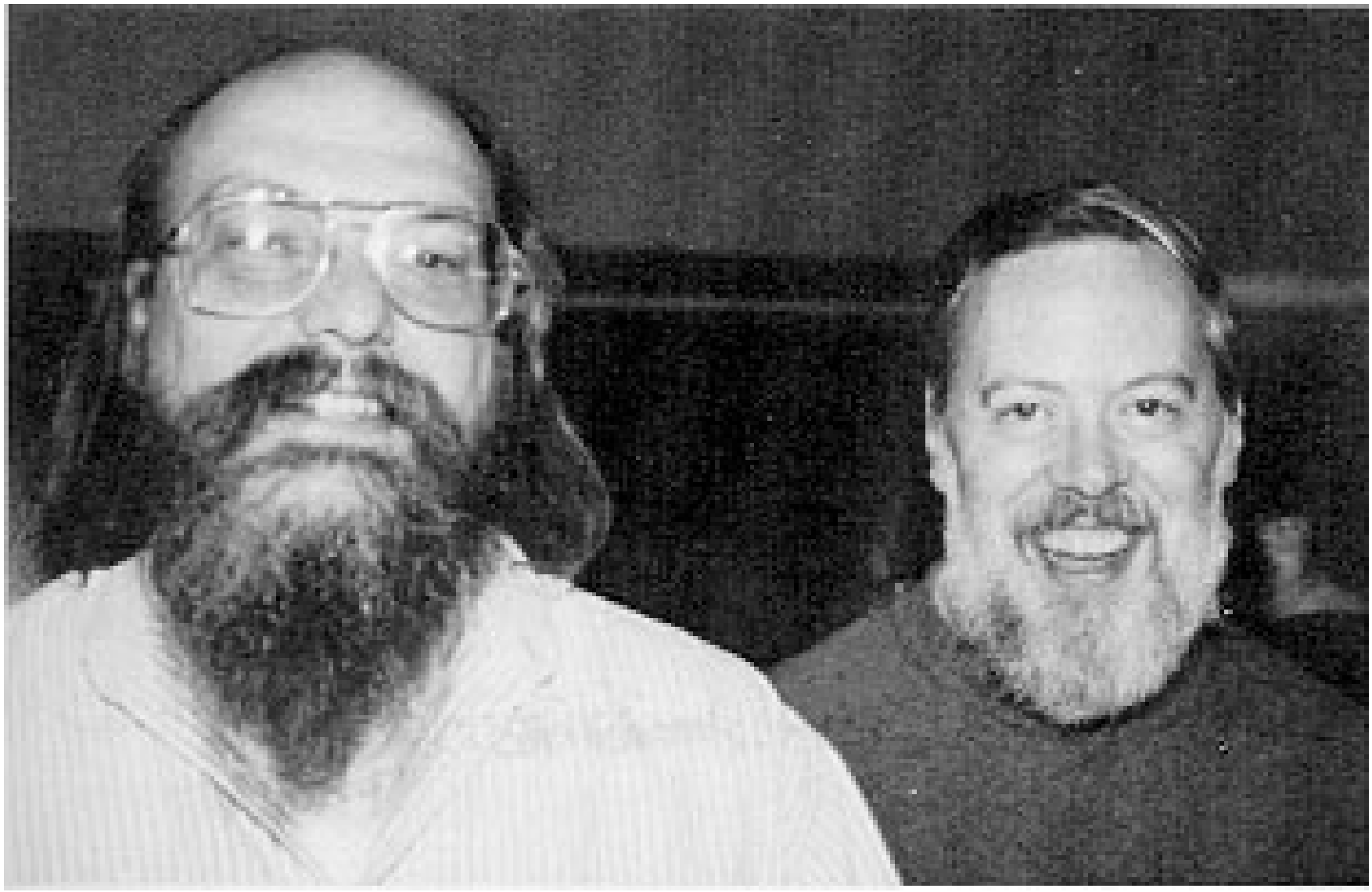
- After the conclusion of this section you should be able to
  - log onto the FCS Linux system
  - understand the concept of current working directory
  - traverse and manipulate the UNIX filesystem
  - describe the role of the shell within the UNIX environment
  - use simple commands to manipulate files (**`cd`**, **`ls`**, **`cp`**, **`rm`**, **`cat`**)
  - use standard I/O, piping, and redirection from the UNIX shell

Resources

- On reserve in library: *Your UNIX*, *The Ultimate Guide*, Sumitabha Das
- Online, from CS2023 web site:
  - *C Program Development with Linux*
    - My notes
  - *UNIX Quick Reference Sheet*
  - *Linux User's Guide*, by Larry Greenfield
  - *Developing on Linux*, by Nathan Thomas, RedHat

What is UNIX?

- A computer operating system
- A software development environment
- Built in late '60s, early '70s by Kent Thompson and Dennis Ritchie
- Originally written in assembler, later rewritten in C (allowing greater portability), a language invented by Ritchie



What is UNIX?

- 1983: U California (Berkeley) created its own: BSD UNIX
  - TCP/IP built-in
- ATT revised UNIX: System V, release 4 (SVR4)
- UNIX comes in several flavours:
  - BSD-based: SunOS, Linux
  - SVR4-based: HP-UX, CRAY UNICOS, IBM AIX

What is an operating system?

Operating Systems

- Interacts with:
  - Applications
  - Users, through a command language interpreter
- OS offers services:
  - Scheduling of multiple programs
  - Memory management
  - Access to hardware
  - Reports errors to applications

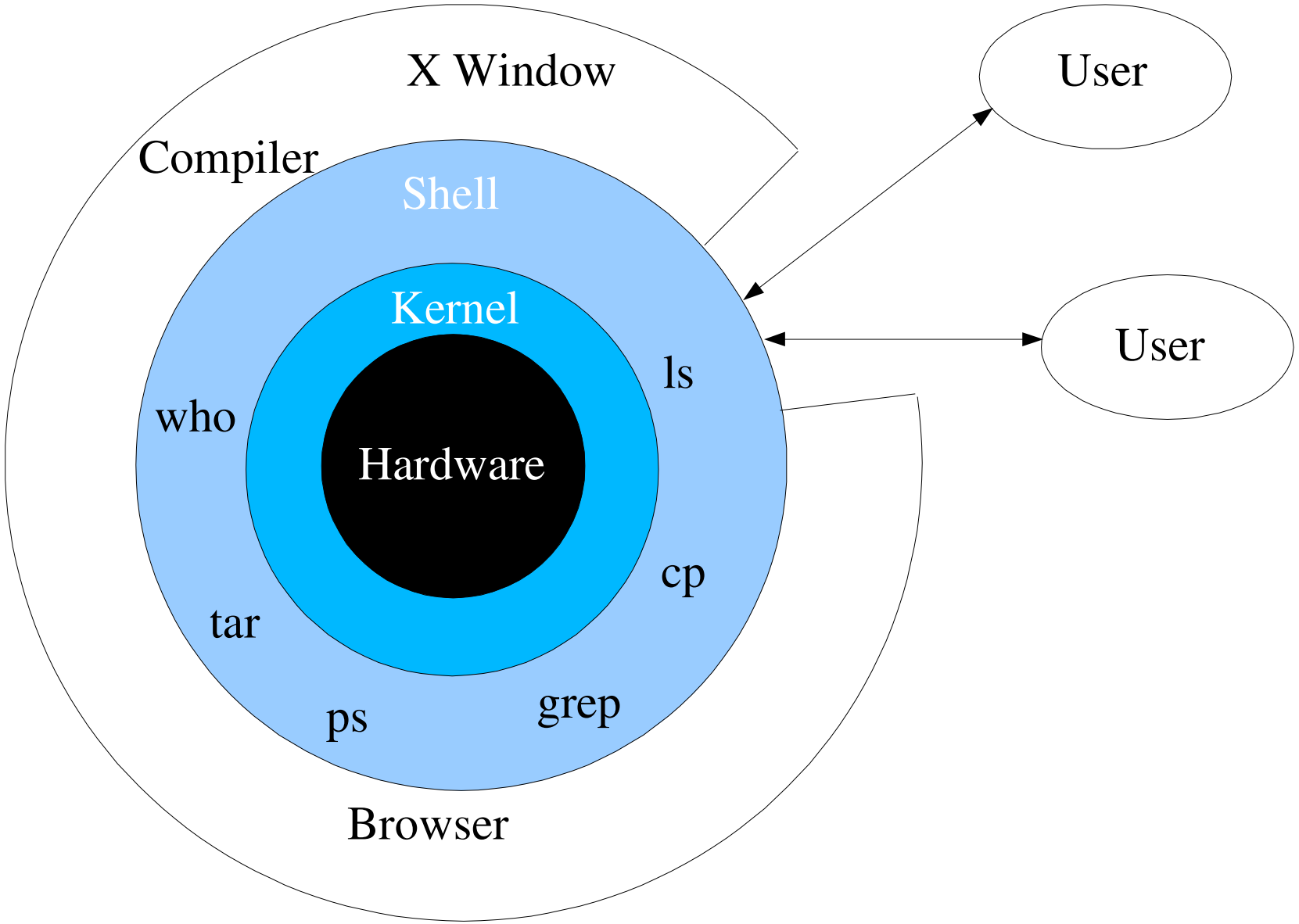
UNIX Philosophy

- Make each program do one thing well.
  - Reusable software tools: 1 tool = 1 function
- Expect the output of every program to become the input of another, yet unknown, program to combine simple tools to perform complex tasks
- Everything seen as a file

UNIX Features

- Multi-user
- Hierarchical file system
- Multi-tasking
- Threads
- Virtual memory
- Built-in networking
- Extensive set of utilities

Inside UNIX

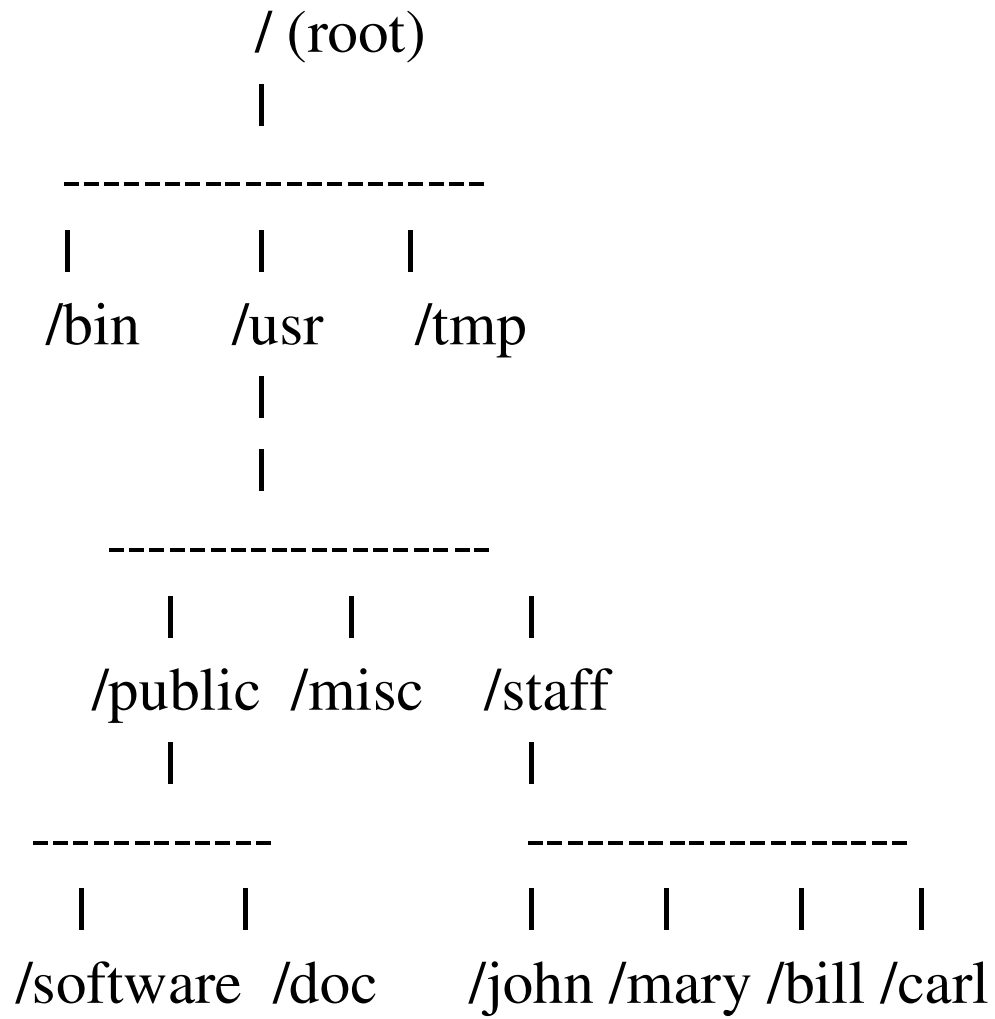

X Window
Compiler
Shell
Kernel
Hardware
who
ls
cp
tar
ps
grep
Browser
User
User

File System

- "Files have places and processes have life"
  - Kaare Christian
- All files are "flat": just a sequence of bytes
- File system is hierarchical

File System

- Organized as a tree
  - Each node is a directory
  - Each directory can contain other files or directories or both
  - Root: "/"
- Each file in a given directory must be unique
- UNIX is cAsE sEnSiTiVe


/ (root)
/bin
/usr
/tmp
/public
/misc
/staff
/software
/doc
/john
/mary
/bill
/carl

File System

- Files are referenced by name
  - absolute reference: beginning with "/"
  - relative reference: based on current directory
- Shortcuts:
  - "..": parent directory
  - ".": current directory
  - "~": home directory

Logging In

- To log in to a Unix machine you can either:
  - sit at the *console* (the computer itself)
  - access via the net (using telnet, rsh, ssh, kermit, or some other remote access client).
  - To access machines in ITD415, use id415mxx.cs.unb.ca as hostname, where xx = 01-40
- The system prompts you for your username and password.
- Usernames and passwords are case sensitive!

Session Startup

- Once you log in, your shell will be started and it will display a prompt.
- When the shell is started it looks in your home directory for some customization files.
  - You can change the shell prompt, your PATH, and a bunch of other things by creating customization files.

Your Home Directory

- Every Unix process has a notion of the "current working directory".

- Your shell (which is a process) starts with the current working directory set to your home directory.

Interacting with the Shell

- The shell prints a prompt and waits for you to type in a command.
- The shell can deal with a couple of types of commands:
  - shell internals - commands that the shell handles directly.
  - External programs - the shell runs a program for you.

Some Simple Commands

- Here are some simple commands to get you started:
  - **`ls`** lists file names (like DOS dir command).
  - **`who`** lists users currently logged in.
  - **`date`** shows the current time and date.
  - **`pwd`** print working directory

The `ls` command

- The ls command displays the names of some files.

- If you give it the name of a directory as a *command line parameter* it will list all the files in the named directory.

`ls` Command Line Options

- We can modify the output format of the **`ls`** program with a *command line option*.
- The ls command support a bunch of options:
  - **`l`** *long* format (include file times, owner and permissions)
  - **`a`** *all* (shows hidden* files as well as regular files)
  - **`t`** sort by modification time.

*hidden files have names that start with "."

File Names

- .c: C source files
- .h: C header files
- .o: compiled program (object file)
- files that begin with "." (hidden files, *e.g.* .bashrc) are not displayed by default by ls
- **file** command: determines file type

Moving Around in the Filesystem

- The cd command can change the current working directory:

  cd *change directory*

- The general form is:

  cd [directoryname]

cd

- With no parameter, the **cd** command changes the current directory to your home directory.
- You can also give **cd** a relative or absolute pathname:

```
cd /usr
cd ..
```

Some more commands and command line options

- **`ls -R`** will list everything in a directory and in all the subdirectories recursively (the entire hierarchy).
  - you might want to know that Ctrl-C will cancel a command (stop the command)!
- **`pwd`**: print working directory
- **`df`**: shows what disk holds a directory.

Copying Files

- The **`cp`** command copies files:

  `cp [options] source dest`

- The source is the name of the file you want to copy.
- dest is the name of the new file.
- source and dest can be relative or absolute.

Another form of **cp**

- If you specify a dest that is a directory, cp will put a copy of the source in the directory.
- The filename will be the same as the filename of the source file.

```
cp [options] source destdir
```

Deleting (removing) Files

- The **`rm`** command deletes files:

  `rm [options] names...`

- **`rm`** stands for "remove".
- You can remove many files at once:

```
rm foo /tmp/blah ../ass1/test.c
```

File attributes

- Every file has some attributes:
  - Access Times:
    - when the file was created
    - when the file was last changed
    - when the file was last read
  - Size
  - Owners (user and group)
  - Permissions

File System Security

- Each file has three sets of permission bits:
  - user
  - group
  - other
- Each set has three bits that represent:
  - read
  - write
  - execute

File System Security

- If a file's permission is "execute", is means it can be ran as a other utility or command.
- Directories need to be
  - readable to see the files they contain
  - Executable to change directory to them
  - Writable to create,edit or remove files from them.

File Time Attributes

- Time Attributes:
  - when the file was last changed **`ls -l`**
  - when the file was created* **`ls -lc`**
  - when the file was last read (accessed) **`ls -ul`**

*actually it's the time the file status last changed.

Other filesystem and file commands

- **`mkdir`** make directory
- **`rmdir`** remove directory
- **`touch`** change file timestamp (can also create a blank file)
- **`cat`** concatenate files and print out to terminal.

Shells

Also known as: Unix Command Interpreter

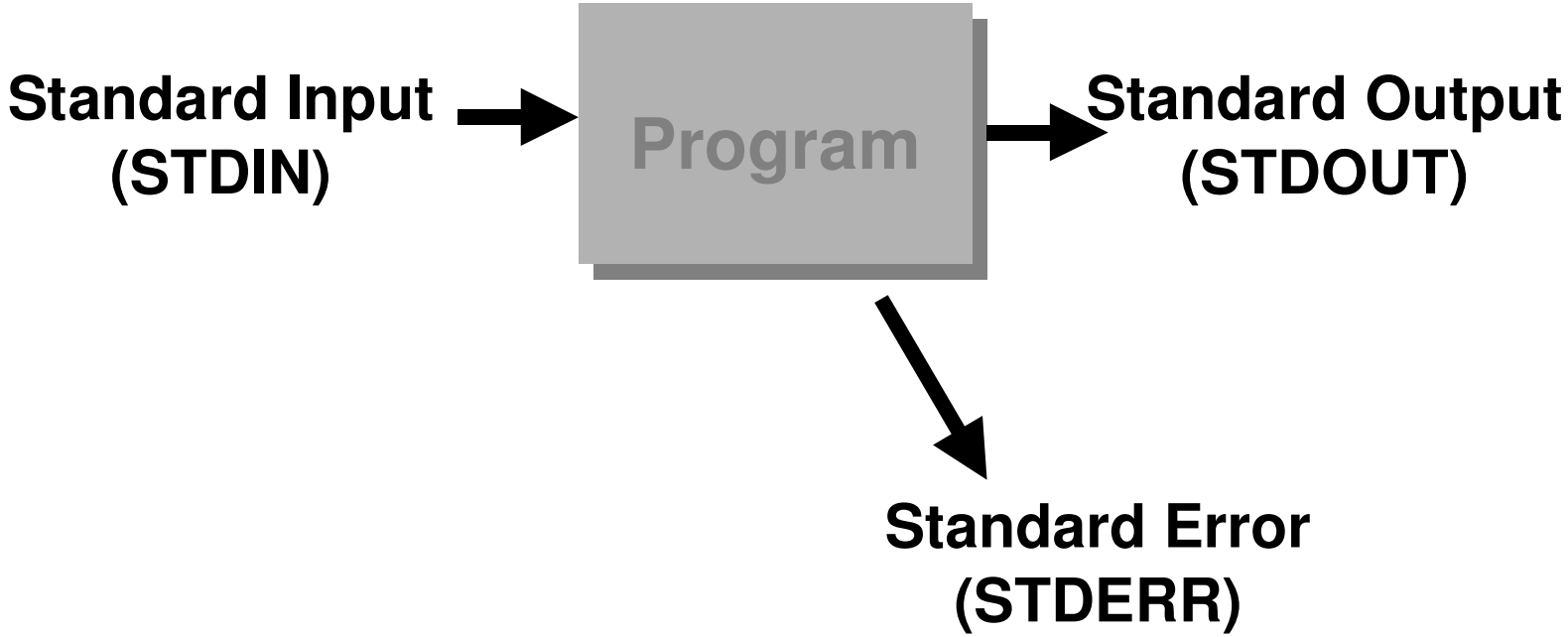
Shell as a user interface

- A shell is a command interpreter that turns text that you type (at the command line) in to actions:
  - runs a program, perhaps the **`ls`** program.
  - allows you to edit a *command line*.
  - can establish alternative sources of input and destinations for output for programs.

Running a Program

- You type in the name of a program and some command line options:
  - The shell reads this line, finds the program and runs it, feeding it the options you specified.
  - The shell establishes 3 I/O *channels*:
    - Standard Input
    - Standard Output
    - Standard Error

Programs and Standard I/O



Unix Commands

- Most Unix commands (programs):
  - read something from standard input.
  - send something to standard output (typically depends on what the input is!).
  - send error messages to standard error.

Defaults for I/O

- When a shell runs a program for you:
  - standard input is your keyboard.
  - standard output is your screen/window.
  - standard error is your screen/window.

Terminating Standard Input

- If standard input is your keyboard, you can type stuff in that goes to a program.
- To end the input you press Ctrl-D (^D) on a line by itself, this ends the input *stream*.
- The shell is a program that reads from standard input.
- What happens when you give the shell ^D?

Popular Shells

sh	Bourne Shell
ksh	Korn Shell
csh	C Shell
bash	Bourne-Again Shell

Customization

- Each shell supports some customization.
  - User prompt
  - Where to find mail
  - Shortcuts
- The customization takes place in *startup* files – files that are read by the shell when it starts up

Startup files

```
sh,ksh:
  /etc/profile (system defaults)
    ~/.profile
bash:
  ~/.bash_profile
  ~/.bashrc
  ~/.bash_logout
csh:
  ~/.cshrc
  ~/.login
  ~/.logout
```

Wildcards (metacharacters) for filename abbreviation

- When you type in a command line the shell treats some characters as special.
- These special characters make it easy to specify filenames.
- The shell processes what you give it, using the special characters to replace your command line with one that includes a bunch of file names.

The special character *

- * matches anything.
- If you give the shell * by itself (as a command line argument) the shell will remove the * and replace it with all the filenames in the current directory.
- **“a*b”** matches all files in the current directory that start with **a** and end with **b**.

Understanding *

- The **`echo`** command prints out whatever you give it:

  ```
  > echo hi
  hi
  ```

- Try this:

  ```
  > echo *
  ```

* and `ls`

- Things to try:

```
ls *
ls -al *
ls a*
ls *b
```

Input Redirection

- The shell can attach things other than your keyboard to standard input.
  - A file (the contents of the file are fed to a program as if you typed it).
  - A pipe (the output of another program is fed as input as if you typed it).

Output Redirection

- The shell can attach things other than your screen to standard output (or stderr).
  - A file (the output of a program is stored in file).
  - A pipe (the output of a program is fed as input to another program).

How to tell the shell to redirect things

- To tell the shell to store the output of your program in a file, follow the command line for the program with the “>” character followed by the filename:

```
ls > lsout
```

the command above will create a file named **`lsout`** and put the output of the **`ls`** command in the file.

Input redirection

- To tell the shell to get standard input from a file, use the "<" character:

```
sort -g < nums
```

- The command above would sort the lines in the file nums and send the result to stdout.

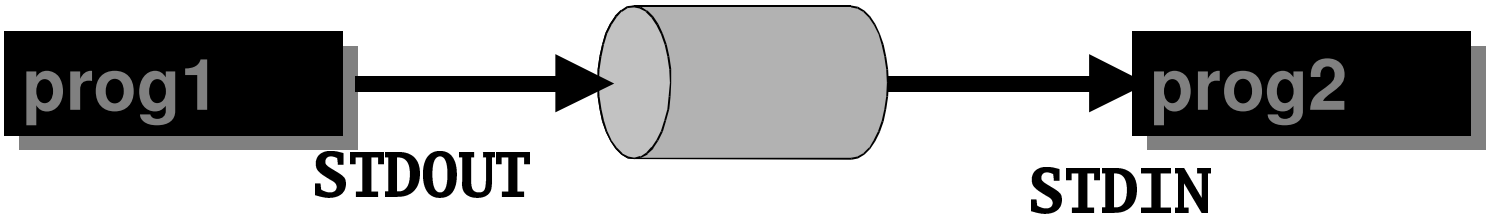
You can do both!

```
sort -g < nums > sortednums
```

```
tr a-z A-Z < letter > rudeletter
```

Pipes

- A pipe is a holder for a stream of data.
- A pipe can be used to hold the output of one program and feed it to the input of another.



Asking for a pipe

- Separate 2 commands with the “|” character.
- The shell does all the work!

```
ls | sort
ls | sort > sortedls
```

Pipes Examples

Count the files in a directory

```
% ls | wc -l
```

Show the last 5 files in a sorted directory listing

```
% ls | sort | tail -5
```

Search a file for all occurrence of the string "system" and pause at each page

```
% cat file1.txt | grep "system" | more
```

We can combine these with redirections

```
% cat file1.txt | grep "system" >
  output.txt
```

Shell Variables

- The shell keeps track of a set of parameter names and values.
- Some of these parameters determine the behavior of the shell.
- We can access these variables:
  - set new values for some to customize the shell.
  - find out the value of some to help accomplish a task.

Example Shell Variables

sh / ksh / bash

PWD	*current working directory*
PATH	*list of places to look for commands*
HOME	*home directory of user*
MAIL	*where your email is stored*
TERM	*what kind of terminal you have*
HISTFILE	*where your command history is saved*

Displaying Shell Variables

- Prefix the name of a shell variable with "$".
- The **echo** command will do:

```
echo $HOME
echo $PATH
```

- You can use these variables on any command line:

```
ls -al $HOME
```

Setting Shell Variables

- You can change the value of a shell variable with the set command (this is a shell *builtin* command). For (t)csh:

```
set HOME=/etc
set PATH=/usr/bin:/usr/etc:/sbin
set NEWVAR="blah blah blah"
```

Setting Shell Variables

- For bash:

```
HOME=/etc
PATH=/usr/bin:/usr/etc:/sbin
NEWVAR="blah blah blah"
```

`set` command (shell builtin)

- The **`set`** command with no parameters will print out a list of all the shell varibles.
- You'll probably get a pretty long list…
- Depending on your shell, you might get other stuff as well...

The **PATH**

- Each time you give the shell a command line it does the following:
  - Checks to see if the command is a shell built-in.
  - If not - tries to find a program whose name (the filename) is the same as the command.
- The **PATH** variable tells the shell where to look for programs (non built-in commands).

echo **$PATH**

```
[aubanel@id415m01 ~]$ echo $PATH
.:/fcs/bin:/home1/staff/aubanel/bin:/bin:/usr
  /bin:/usr/local/bin:/usr/X11R6/bin:/fcs/jav
  a/jdk/bin
```

- The **PATH** is a list of ":" delimited directories.
- The **PATH** is a list and a *search order*.

- You can add stuff to your PATH by changing the shell startup file (**~/.bashrc**)

Job Control

- The shell allows you to manage *jobs*
  - place *jobs* in the *background*
  - move a job to the foreground
  - suspend a job
  - kill a job

Background jobs

- If you follow a command line with "&", the shell will run the *job* in the background.
  - you don't need to wait for the job to complete, you can type in a new command right away.
  - you can have a bunch of jobs running at once.
  - you can do all this with a single terminal (window).

```
ls -lR > saved_ls &
```

Listing jobs

- The command *jobs* will list all background jobs:

```
> jobs
[1] Running      ls -lR > saved_ls &
>
```

- The shell assigns a number to each job (this one is job number 1).

Suspending and Killing the Foreground Job

- You can suspend the foreground job by pressing ^Z (Ctrl-Z).
  - Suspend means the job is stopped, but not dead.
  - The job will show up in the **`jobs`** output.

- You can *kill* the foreground job by pressing ^C (Ctrl-C).
  - It's gone...

Programming

- Text editors
  - emacs, vi
  - Can also use any PC editor if you can get at the files from your PC.
- Compilers: gcc.
- Debuggers: gdb