Creating LAMP Infrastructure for Digital Humanities Projects

Jonathan Martin
Welcome to DHSI 2017!

Thanks for joining the DHSI community!

In this booklet, you will find essential course materials prefaced by some useful information about getting settled initially at UVic, finding your way around, getting logged in to our network (after you’ve registered the day before our courses begin, and received your login information), and so on.

Given our community’s focus on things computational, it will be a surprise to no one that we might expect additional information online for some of the classes - your instructors will let you know - or that the most current version of all DHSI-related information may be found on our website at dhsi.org.

Do check in there first if you need anything that’s not in this coursepak.

To access the DHSI wifi network, simply go into your wireless settings and connect to the “DHSI” network.

And please don’t hesitate to be in touch with us at institut@uvic.ca or via Twitter at @AlyssaA_DHSI or @DHInstitute if we can be of any help ....
Saturday, 3 June 2017 [Workshop: A Brief Introduction to DH]

9:00 to 4:00
Workshop: A Brief Introduction to DH (MacLaurin D115, Classroom)

Sunday, 4 June 2017 [DHSI Registration, Meetings, Workshops]

Full Day Workshops
- Data Wrangling for Digital Projects (MacLaurin D111 Classroom)
- Intersectionality and Surveillance (David Strong C124, Classroom)

9:00 to 4:00
- Early Class Meeting: 3. [Foundations] DH For Department Chairs and Deans (David Strong C114, Classroom)

Further details are available from instructors in early May to those registered in the class. Registration materials will be available in the classroom.

12:30 to 4:30
DHSI Registration (NEW LOCATION: MacLaurin Building, Room A100)

1:00 to 4:00
3-hour Workshops
- DHSI Knits: History of Textiles and Technology (David Strong C108, Classroom)
- Use Apache Spark to Explore and Process Large Datasets for Humanities Research (David Strong C114, Classroom)
- 3D Visualization for the Humanities (MacLaurin D105, Classroom)
- Archives for Digital Humanists (MacLaurin D010, Classroom)
- Dynamic Ontologies for the Humanities (MacLaurin D016, Classroom)

DHSI Welcome BBQ (Get the details, and let us know you're coming, via this link!)

4:30 to 6:00
After the welcome BBQ, many will wander to Cadboro Bay and the pub at Smuggler's Cove OR the other direction to Shelbourne Plaza and Maude Hunter's Pub.

Monday, 5 June 2017

Your hosts for the week are Ray Siemens and Dan Sondheim.

7:45 to 8:15
Last-minute Registration (MacLaurin Building, Room A100)

8:30 to 10:00
Welcome, Orientation, and Instructor Overview (MacLaurin A144)

10:15 to Noon
- Classes in Session (click for details and locations)
  1. [Foundations] Text Encoding Fundamentals and their Application (Clearihue A102, Lab)
  3. [Foundations] DH For Department Chairs and Deans (David Strong C114, Classroom)
  4. [Foundations] Fundamentals of Programming/Coding for Human(s|ists) (Clearihue A103, Lab)
  5. [Foundations] Understanding The Predigital Book: Technology and Texts (McPherson Library A003, Classroom)
  6. Out-of-the-Box Text Analysis for the Digital Humanities (Human and Social Development A160, Lab)
  7. Geographical Information Systems in the Digital Humanities (Human and Social Development A170, Lab)
  8. CloudPowering DH Research (Clearihue A012, Lab)
  9. Digital Storytelling (MacLaurin D111, Classroom)
  10. Critical Pedagogy and Digital Praxis in the Humanities (MacLaurin D105, Classroom)
  11. Text Processing - Techniques & Traditions (Corbett A229, Classroom)
  12. 3D Modelling for the Digital Humanities and Social Sciences (MacLaurin D010, Classroom)
  13. RDF and Linked Open Data (David Strong C108, Classroom)
  14. Conceptualizing and Creating a Digital Edition (MacLaurin D103, Classroom)
  15. Visualizing Information: Where Data Meets Design (MacLaurin D107, Classroom)
  16. Drupal for Digital Humanities Projects (MacLaurin D109, Classroom)
  17. Introduction to Electronic Literature in DH: Research and Practice (MacLaurin D115, Classroom)
18. Accessibility & Digital Environments (MacLaurin D101, Classroom)
19. Feminist Digital Humanities: Theoretical, Social, and Material Engagements (David Strong C124, Classroom)
20. XML Applications for Historical and Literary Research (MacLaurin D016, Classroom)
21. Open Access and Open Social Scholarship (MacLaurin D114, Classroom)
22. Ethical Collaboration in the Digital Humanities (Clearihue D131, Classroom)
24. Digital Games as Interactive Tools for Scholarly Research, Communication and Pedagogy (MacLaurin D110, Classroom)

12:15 to 1:15
Lunch break / Unconference Coordination Session
Undergraduate Meet-up, Brown-Bag (details via email)

1:30 to 4:00
Classes in Session

Institute Panel: Perspectives on DH (or, #myDHis ...)
Co-Chairs: Emily Murphy (Queens U) and Randa El-Khatib (U Victoria)
(MacLaurin A144)

▼ Jessica Otis (Carnegie Mellon U): "DH In The Big Tent."
Abstract: As a Digital Humanities specialist at Carnegie Mellon University, I experience the "Big Tent" of digital humanities on a regular basis. I support professors and students using everything from computer simulations for philosophical research to machine learning for studying old texts, while my own projects run the gamut from text encoding to network analysis to software containerization. For me, working in DH is a balancing act between pushing the borders of humanities knowledge while still maintaining a meaningful core of "what is DH?"

▼ David Wrisley (NYU Abu Dhabi / American U Beirut): "#myDHis messy"
Abstract: As a digital medievalist working with a textual record where orthographic variance is the norm, living in environments with complex multilingual medleys, and contributing to a number of local DH cultures across countries, my DH is perpetually messy. Some infrastructure can make things easier, but I would like to speak in praise of bricolage: making do with what you have, when you have it and for as long as you have it. Whereas some might argue that confusion is the mother of error, let us consider it instead as a necessary step towards creativity.

▼ Meaghan Brown (Folger Shakespeare Library): "#myDHis Dusty"
Abstract: As the Fellow for Data Curation at the Folger Shakespeare Library, the digital project I manage and the projects I coordinate with are deeply engaged with and indebted to our physical holdings. While Digital Humanities is often accused of being entranced with the shiny and new, I have found that descriptive bibliography and other ‘dusty’ disciplines have a great deal to offer as we create, curate, and connect digital objects. I want to ask how forms of scholarly description and organization can help us structure digital collections and think about the material implications of digital humanities work.

Abstract: An emerging critical discourse applying social justice theory and practice to digital/media scholarship is claiming growing interest. What might an intersectional framework accomplish with respect to digital humanities when an analysis with respect to race, class, gender, sexuality and difference is provided? Nieves will provide some points and reflection for further debate and discussion.

▼ Corina Koolen (U Amsterdam): "The Downside of Difference."
Abstract: Computational methods and models generally focus on differences between data sets, even though the overlap between data sets can be large. I argue that we miss part of the truth – and might even practice a form of cherry picking – when we overlook the commonalities. My argument will take the form of an example: the ‘gap’ between female and male authors in Dutch literary award nominees.

▼ Jacob Heil (C of Wooster): "#myDHis edgy and therefore slow"
Abstract: Hoping that you’ll forgive an admittedly fast-and-loose borrowing from graph theory, I want to wonder (with you) whether or not, in the work that we do and in our discourses of “collaboration,” we privilege the node over the edge. Do we think more about, say, expertise than we do about the natures of the relationships between experts? In these brief remarks I want to draw upon my experiences building up (and building upon) cultures of DH as a way of recentering the humanness of digital humanities.

▼ Michelle Schwartz (Ryerson U): "#myDHis radically inclusive"
Abstract: My entry into the digital humanities came via a community archive, rather than through traditional academia, and the focus of my work thus far has been to use DH tools to recreate the feeling of that community space online. Rather than working from a specific research question, my goal has been to use DH to make radical history relevant and accessible, and to inspire in people the joy of discovery. One starting point for that mission has been to work with undergraduate, rather than graduate students, to let those students set their own path, and to use their personal journeys to guide the project.

4:10 to 5:00

5:00 to 6:00
Opening Reception (University Club)
We are grateful to Gale Cengage for its sponsorship of the reception.
Research libraries have always played an important role in the long-term preservation of society’s documentary heritage. And while maintaining large collections of print resources over time is not without its difficulties, the challenges of managing digital materials for the long-term are enormous. New methodologies for building and sustaining our cultural heritage are being developed, and this talk will explore a variety of shared services being deployed by research libraries in Canada in support of digital stewardship and preservation activities.

Our collective cultural memory is increasingly stored in transitory bits and bytes, leading some to warn of an approaching digital dark age, where the historical record is slowly but inevitably overwritten in a thousand air-conditioned server rooms across the globe. Combine this with an environment of increasing political and environmental uncertainty, and the challenges associated with saving our cultural memory can seem daunting. So what can libraries do about it?

This presentation will define digital preservation as a complex set of systems and organizational activities required to ensure the long-term viability of digital materials over time, and provide an update.

BIO: Corey Davis is the Digital Preservation Network Manager for the Council of Prairie and Pacific University Libraries (COPPUL), a consortium of 22 university libraries in Western Canada, where he works to develop the technical and policy infrastructure to support long term preservation of digital objects for all COPPUL members. He is also Systems Librarian at the University of Victoria Libraries.
North America and the world for drinks and appetizers in our brand new Digital Scholarship Commons. This event will take place on the third floor of the Mearns Centre for Learning /McPherson Library on Thursday June 8th from 5:30-7:30 pm. RSVP here.

7:30 to 9:30

(Groovy?) Movie Night (MacLaurin A144)

Friday, 9 June 2017 [DHSI; SHARP Opening]

9:00 to Noon

DHSI Classes in Session

11:00 to 1:30

SHARP Conference Registration (MacLaurin A100)

Late registration is available at the SHARP information desk, at this same location.

12:15 to 1:15

DHSI Lunch Reception / Course E-Exhibits (MacLaurin A100)

1:30 to 1:50

DHSI Week 1 Farewell (Hickman 105)

2:00 to 2:45

SHARP Conference Opening, Welcome (MacLaurin A144)

2:45 to 3:45

Julia Flanders (Northeastern U): “Cultures of Reception: Readership and Discontinuity in the History of Women’s Writing.”

Chair: Sydney Shep (Victoria U Wellington)

(MacLaurin A144; the lecture will also be live-streamed, with love and care, to Hickman 105)

Abstract: The work of textual recovery and republication for which the Women Writers Project is well known is to all appearances an effort to rediscover a textual and artifactual history: a history of books, once in circulation, now lodged invisibly in remote libraries and inaccessible to scholars and students, but brought back into the light by digital remediation. But the more significant and difficult rediscovery has to do with readership. In republishing these texts we are also seeking to reinsert them into a cultural landscape that has forgotten how to read them. And in republishing them digitally we are also reopening the question of what it means to read. Our challenge is to develop mechanisms of circulation that avoid reproducing the original conditions of invisibility and disappearance in which women’s writing circulated. This presentation will examine the WWP’s work on readership and reception in the context of digital technologies of reading and textual circulation.

4:00 to 5:00

Joint Reception: SHARP and DHSI (University Club)

DHSI Colloquium Poster/Demo Session

SHARP Digital Demo and Poster Session

Saturday, 10 June 2017 [SHARP Conference + Suggested Outings!]

8:30 to 9:00

Late Registration (at the SHARP information desk) (MacLaurin A100)

8:45 to 5:00

SHARP Conference Sessions

5:00 to 6:00

Lisa Gitelman (NYU): “Emoji Dick, Prequels and Sequels.”

(MacLaurin A144)

Abstract: This is the second in a sequence of talks that takes a 2010 “translation” of Moby Dick into emoji as an opportunity to consider the conditions of possibility that might delimit books and literature in the contemporary moment. A massive white codex and extended work of crowd-sorcery, Emoji Dick points toward the varieties of reading and--especially--of not reading that characterize our ever more digitally mediated and data-described world. Here I proceed by locating Emoji Dick alongside a key group of precursors and successors.

Some ideas, for those who’d like to explore the area!

Suggested Outing 1, Botanical Beach (self-organised; car needed)

A self-guided visit to the wet, wild west coast tidal shelf (and historically-significant former research site) at Botanical Beach; we recommend departing early (around 8.00 am) to catch low tide for a better view of the wonderful underwater life! Consider bringing a packed lunch to nibble-on while looking at the crashing waves when there, and then have an afternoon drink enjoying the view from the deck of the Port Renfrew Hotel.

Suggested Outing 2, Butchart Gardens (self-organised)

A shorter journey to the resplendently beautiful Butchart Gardens and, if you like, followed by (ahem) a few minutes at the nearby Church and State Winery, in the Saanich Peninsular Peninsula. About an hour there by public bus from UVic, or 30 minutes by car.
**All day Suggested Outings**

- **Suggested Outing 3, Saltspring Island (self-organised; a full day, car/bus + ferry combo)**
  Why not take a day to explore and celebrate the funky, laid back, Canadian gulf island lifestyle on Saltspring Island. Ferry departs regularly from the Schwartz Bay ferry terminal, which is about one hour by bus / 30 minutes by car from UVic. You may decide to stay on forever ....

- **Suggested Outing 4, Paddling Victoria’s Inner Harbour (self-organised)**
  A shorter time, seeing Victoria’s beautiful city centre from the waterways that initially inspired its foundation. A great choice is the day is sunny and warm. Canoes, kayaks, and paddle boards are readily rented from Ocean River Adventures and conveniently launched from right behind the store. Very chill.

- **And more!**

  Self-organised High Tea at the Empress Hotel, scooter rentals, visit to the Royal BC Museum, darts at Christies Carriage House, a hangry breakfast at a local diner, whale watching, kayaking, brew pub sampling (at Spinnaker's, Swans, Moon Under Water, and beyond!), paddle-boarding, a tour of used bookstores, and more have also been suggested!

---

**Sunday, 11 June 2017 [SHARP + DHSI Registration, Workshops]**

8:30 to 9:00 Late Registration (at the SHARP information desk) *(MacLaurin A100)*

8:45 to 5:00 **SHARP Conference Sessions**

9:00 to Noon **DHSI Workshop: Race, Social Justice, and DH: Applied Theories and Methods** *(MacLaurin D110, Classroom)*

12.30 to 5:00 **DHSI Registration** *(NEW LOCATION: MacLaurin A100)*
After registration, many will wander to Cadboro Bay and the pub at Smuggler’s Cove OR the other direction to Shelbourne Plaza and Maude Hunter’s Pub.

1:00 to 4:00 **DHSI 3-hour Workshops**
- **DHSI Knits: Using Design Technology** *(MacLaurin D010, Classroom)*
- **Intersections of DH and LGBTTIQ+ Studies** *(MacLaurin D105, Classroom)*
- **Regular Expressions** *(MacLaurin D111, Classroom)*
- **Digital Publishing in the Humanities** *(MacLaurin D101, Classroom)*
- **Steering the XPath** *(MacLaurin D103, Classroom)*
- **Crowdsourcing as a Tool for Research and Public Engagement** *(MacLaurin D107, Classroom)*
- **Web Annotation as Critical Humanities Practice** *(MacLaurin D016, Classroom)*

5:00 to 6:00 **SHARP Lecture: Robert Bringhurst: “The Mind-Book Problem.”** *(MacLaurin A144)*

**Abstract:** I used to hear a lot, in philosophical circles, about the mind-brain problem and its cognate, the mind-body problem. More recently, in pedagogical circles, I hear about an issue which I’ve come to call the mind-book problem. It is, briefly, the failure of so many human minds, in a hyperliterate society, to find any deep, lasting, and fruitful engagement with the book.

All these problems (mind-brain, mind-body, and mind-book) seem to me related to one another. They are also related to something larger: the mind-world problem, familiar to philosophers and medical practitioners in all times and places. There are many who feel that the mind-world problem has reached epidemic proportions today, especially in the humanities. This may have something to do with the prevalence of the mind-book problem there as well.

The book has been praised as the ark of civilization, the measure of the human heart, and the voice of God incarnate. It has also, of course, been damned as a form of dalliance or the invention of the Devil. More recently, it has been patronized as an archaic cultural relic in need of replacement or technological upgrading. I will not deny that upgrading is possible, and on several fronts desirable. But a book without a mind – like a mindless brain, a mindless body, or a mindless civilization – is a problem for which a technological upgrade may not be the answer. I will explore the mind-book problem from this and other angles.

6:00 to 9:00 **SHARP Banquet** *(University Club)*

---

**Monday, 12 June 2017 [SHARP + DHSI]**

Your hosts for the week are Alyssa Arbuckle, Ray Siemens and Dan Sondheim.

7:45 to 8:15 **DHSI Last-minute Registration** *(MacLaurin A100)*

8:30 to 10:00 **DHSI Welcome, Orientation, and Instructor Overview** *(MacLaurin A144)***
### 8:45 to 4:00

**SHARP Conference Sessions**

- **DHSI Classes in Session** (click for details and locations)
  - 25. [Foundations] Intro to Computation for Literary Criticism (Clearhue A105, Lab)
  - 26. [Foundations] Developing a Digital Project (With Omeka) (Cornett A229, Classroom)
  - 27. [Foundations] Models for DH at Liberal Arts Colleges (& 4 yr Institutions) (MacLaurin D109, Classroom)
  - 28. [Foundations] Introduction to Javascript and Data Visualization (Clearhue D131, Classroom)
  - 29. Wrangling Big Data for DH (Clearhue A108, Lab)
  - 30. Stylometry with R: Computer-Assisted Analysis of Literary Texts (Clearhue A102, Lab)
  - 31. Sounds and Digital Humanities (MacLaurin D111, Classroom)
  - 32. Digital Humanities Pedagogy: Integration in the Curriculum (Cornett A121, Classroom)
  - 34. Creating LAMP Infrastructure for Digital Humanities Projects (MacLaurin D105, Classroom)
  - 35. Understanding Topic Modeling (MacLaurin D105, Classroom)
  - 36. Palpability and Wearable Computing (MacLaurin D016, Classroom)
  - 37. Building a Professional Identity and Skillset in the Digital Humanities (MacLaurin D101, Classroom)
  - 38. Digital Editing with TEI: Critical, Documentary and Genetic Editing (MacLaurin D114, Classroom)
  - 40. Understanding Digital Video (MacLaurin D103, Classroom)
  - 41. Beyond TEI: Metadata for Digital Humanities (David Strong C114, Classroom)
  - 42. Extracting Cultural Networks from Thematic Research Collections (Clearhue D132, Classroom)
  - 43. Digital Public Humanities (MacLaurin D010, Classroom)
  - 44. Using Fedora Commons / Islandora (Human and Social Development A160, Lab)
  - 45. Practical Software Development for Nontraditional Digital Humanities Developers (David Strong C124, Classroom)
  - 46. Documenting Born Digital Creative and Scholarly Works for Access and Preservation (MacLaurin D115, Classroom)
  - 47. An Introduction to Computational Humanities: Mining, Machine Learning and Future Challenges (MacLaurin D110, Classroom)
  - 48. Games for Digital Humanists (David Strong C108, Classroom & Human and Social Development A170, Lab)
  - 49. Introduction to XSLT for Digital Humanists (Cornett A128, Classroom)

### 10:15 to Noon

- DHSI Lunch break / Unconference Coordination Session
- DHSI Undergraduate Meet-up, Brown-Bag (details via email)

### 1:30 to 4:00

- DHSI Classes in Session

### 4:10 to 5:00

**Joint Institute Lecture (SHARP and DHSI):**
Brewster Kahle (Internet Archive) and Jo-Ann Roberts (CBC): "A Conversation with Brewster Kahle, moderated by Jo-Ann Roberts."

Chair: Jonathan Bengtson (U Victoria)

(MacLaurin A144; the conversation will also be live-streamed, with love and care, to Hickman 105)

**SHARP Conference Closing Remarks**

### 5:00 to 6:00

**Joint Reception: SHARP and DHSI** (University Club)

### Tuesday, 13 June 2017

- **9:00 to Noon**
  - Classes in Session

- **12:15 to 1:15**
  - Lunch break / Unconference
  - "Mystery" Lunches

- **1:30 to 4:00**
  - Classes in Session

- **4:15 to 5:45**
  - DHSI Colloquium Session 4 (MacLaurin A144)

- **6:00 to 8:00**
  - DHSI Newcomer's Beer-B-Q (Smuggler's Cove)

### Wednesday, 14 June 2017

- **9:00 to Noon**
  - Classes in Session

- **12:15 to 1:15**
  - Lunch break / Unconference
  - "Mystery" Lunches
### Thursday, 15 June 2017

<table>
<thead>
<tr>
<th>Time</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>1:30 to 4:00</td>
<td>Classes in Session</td>
</tr>
<tr>
<td>4:15 to 5:45</td>
<td>DHSI Colloquium Session 5 (<a href="#">MacLaurin A144</a>)</td>
</tr>
<tr>
<td>6:00 to 7:00</td>
<td>“Half Way There (yet again)!” Birds of a Feather Get-Together (<a href="#">Felicitas, Student Union Building</a>)</td>
</tr>
</tbody>
</table>

### Friday, 16 June 2017

<table>
<thead>
<tr>
<th>Time</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>9:00 to Noon</td>
<td>Classes in Session</td>
</tr>
<tr>
<td>12:15 to 1:15</td>
<td>Lunch break / Unconference</td>
</tr>
<tr>
<td></td>
<td>“Mystery” Lunches</td>
</tr>
<tr>
<td></td>
<td>[Instructor lunch meeting]</td>
</tr>
<tr>
<td>1:30 to 4:00</td>
<td>Classes in Session</td>
</tr>
<tr>
<td>4:15 to 5:45</td>
<td>DHSI Colloquium Session 6 (<a href="#">MacLaurin A144</a>)</td>
</tr>
<tr>
<td>7:30 to 9:30</td>
<td>(Groovier?) Movie(r) Night (<a href="#">MacLaurin A144</a>)</td>
</tr>
</tbody>
</table>

---

**Abstract:**

The Digital Humanities have developed in two main institutional contexts: within dedicated research centers and within more traditional discipline-based faculties -- with those in the former case in danger of closing themselves within ivory towers of a think-alike community and the “lone wolves” of the latter case most likely to have suffered isolation and lack of real engagement with their peers. More recently, these disparate experiences are seeing some convergence as, slowly but steadily, DH has raised its profile and its impact within the Humanities, with external and internal pressure on other fields now mounting (as demonstrated by intensification of newspapers articles, manifestos and positions being advertised). We are now at a turning point: will the experience of DH feed back and enrich disciplines such as English, Spanish, History, and so on, or will brand new disciplines stem from it, as has happened for Computational Linguistics as a clearly separated entity with respect to Linguistics? These are some of the questions that this lecture will address.

---

**Contact info:**

institut@uvic.ca  P: 250-472-5401  F: 250-472-5681
Instructor: Jonathan Martin (jonathan.martin79@icloud.com & @songsthatssaved)
Suggested Course Hashtag: #DHLinux

Figure 1. Obligatory XKCD 1 - https://xkcd.com/456/

Cheerful, Disarming Course Introduction

Hi, everyone!

Welcome to the course! In our time together at DHSI, we’re going to learn how to build a semi-traditional LAMP\(^1\) stack. In so doing, we’re going to become hackers. While this won’t involve spending time with Matthew Lillard, or – perhaps more poignantly – Chris Hemsworth, it will mean that we will soon have all the tools and knowledge we need to build a server anywhere, at any time, using freely available tools.

Even better, we’ll learn how to keep that server safe and stable. (We may learn how not to gloat about this. Maybe not.)

We’ll learn how to customize (hack) that server, and we’ll learn how to build programs from scratch to suit a wide range of project needs.\(^2\)

\(^1\) As you likely know, LAMP stands for “Linux, Apache, MySQL, and PHP.” Our ‘M’ is MariaDB, a fully interoperable fork of MySQL.
\(^2\) After all that, we’ll probably have lunch. :)
Below, you will find our schedule, some suggested resources, and some general reading material. Don't obsess too much over this last set of material, as we will cover most of it in class. I've provided it as a reference for the work ahead.

So, then, let me just say that I'm thrilled to embark on this journey with all of you, and I hope you will find the course helpful. Please don't hesitate to get in touch (during, or out of, class) with any questions or concerns.

Cheers,

Jonathan
## Draft Course Schedule (Implementation May Vary)

<table>
<thead>
<tr>
<th></th>
<th>Monday</th>
<th>Tuesday</th>
<th>Wednesday</th>
<th>Thursday</th>
<th>Friday</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Morning</strong></td>
<td>Command Line, Linux Conventions Review (lecture and hands-on)</td>
<td>Apache Overview (lecture), Setting up Apache (hands-on)</td>
<td>MariaDB/SQL Overview (lecture), Setting up MariaDB (hands-on)</td>
<td>Potpourri, Hands-on projects and practice</td>
<td>Nginx, Misc., Q&amp;A (lecture and demo), more hands-on</td>
</tr>
<tr>
<td><strong>Afternoon</strong></td>
<td>Installing, Updating, and Securing Linux (lecture and hands-on)</td>
<td>Configuring Apache, Setting up PHP, Introduction to Package Management (lecture and hands-on)</td>
<td>Server Administration Tools (lecture and hands-on), Resources for further study (lecture)</td>
<td>Hands-on projects and practice.</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Useful Resources:

- Our Virtual Environment: [VirtualBox](#)
- Our Linux Distro: [Debian](#)
- [The Command Line Crash Course](#) – Awesome for review purposes.
  - [Alternative review](#) at Codeacady.
- [ExplainShell.com](#) – Wonderful way to figure out “What the hell this command does...”
- [Course Wiki (in progress)](#) – Will contain all the material and instruction we use in class.
Finally, please remember that this course could save your life…

Plausible hypothetical:

Figure 2. Obligatory XKCD 2 - http://xkcd.com/1168/
Welcome to Linux! If you’re a new user, this book can serve as a quick introduction, as well as a guide to common and practical commands. If you have Linux experience, feel free to skip the introductory material.

What’s in This Book?

This book is a short guide, not a comprehensive reference. We cover important, useful aspects of Linux so you can work productively. We do not, however, present every single command and every last option (our apologies if your favorite was omitted), nor delve into detail about operating system internals. Short, sweet, and essential, that’s our motto.

We focus on commands, those pesky little words you type on a command line to tell a Linux system what to do. Here’s an example command that counts lines of text in a file, myfile:

```
wc -l myfile
```

We’ll cover the most important Linux commands for the average user, such as `ls` (list files), `grep` (search for text in a file), `amarok` (play audio files), and `df` (measure free disk space). We touch only briefly on graphical windowing environments like GNOME and KDE, each of which could fill a Pocket Guide by itself.
We’ve organized the material by function to provide a concise learning path. For example, to help you view the contents of a file, we introduce all file-viewing commands together: cat for short text files, less for longer ones, od for binary files,acroread for PDF files, and so on. Then we explain each command in turn, briefly presenting its common uses and options.

We assume you have an account on a Linux system and know how to log in with your username and password. If not, speak with your system administrator, or if the system is your own, use the account created when you installed Linux.

**What’s Linux?**

Linux is a popular, open source operating system that competes with Microsoft Windows and the Apple Macintosh. There are two ways to work with a Linux system:

- A graphical user interface with windows, icons, and mouse control.
- A command-line interface, called the **shell**, for typing and running commands like the preceding `wc`.

Windows and Mac OS computers can be operated by command line as well (Windows with its `cmd` and PowerShell command tools, and OS X with its Terminal application), but most of their users can survive without typing commands. On Linux, however, the shell is critical. If you use Linux without the shell, you are missing out.

**What’s a Distro?**

Linux is extremely configurable and includes thousands of programs. As a result, different varieties of Linux have arisen to serve different needs and tastes. They all share certain core components but may look different and include different programs and files. Each variety is called a **distro** (short for “distribution”). Popular distros include Ubuntu Linux, Red Hat
Enterprise Linux, Slackware, Mint, and more. This book covers core material that should apply to every distro.

**What’s a Command?**

A Linux command typically consists of a *program name* followed by *options* and *arguments*, typed within a shell, like this:

```
$ wc -l myfile
```

The program name (*wc*, the “word count” program) refers to a program somewhere on disk that the shell will locate and run. Options, which usually begin with a dash, affect the behavior of the program. In the preceding command, the `-l` option tells *wc* to count lines rather than words. The argument *myfile* specifies the file that *wc* should read and process. The leading dollar sign ($) is a *prompt* from the shell, indicating that it is waiting for your command.

Commands can have multiple options and arguments. Options may be given individually:

```
$ wc -l -w myfile
```

or combined behind a single dash:

```
$ wc -lw myfile
```

though some programs are quirky and do not recognize combined options. Multiple arguments are also OK:

```
$ wc -l myfile1 myfile2
```

Options are not standardized. The same option letter (say, `-l`) may have different meanings to different programs: in *wc* `-l` it means “lines of text,” but in *ls* `-l` it means “longer output.” In the other direction, two programs might use different options to mean the same thing, such as `-q` for “run quietly” versus `-s` for “run silently.”

Likewise, arguments are not standardized, unfortunately. They usually represent filenames for input or output, but they can be other things too, like directory names or regular expressions.
Commands can be more complex and interesting than a single program with options:

- Commands can run more than one program at a time, either in sequence (one program after another) or in a “pipeline” with the output of one command becoming the input of the next. Linux experts use pipelines all the time.
- The Linux command-line user interface—the shell—has a programming language built in. So instead of a command saying “run this program,” it might say, “if today is Tuesday, run this program; otherwise, run another command six times for each file whose name ends in .txt.”

### Reading This Book

We’ll describe many Linux commands in this book. Each description begins with a standard heading about the command; **Figure 1** shows one for the `ls` (list files) command. This heading demonstrates the general usage in a simple format:

```
ls [options] [files]
```

which means you’d type “ls” followed, if you choose, by options and then filenames. You wouldn’t type the square brackets “[” and “]”: they just indicate their contents are optional; and words in italics mean you have to fill in your own specific values, like names of actual files. If you see a vertical bar between options or arguments, perhaps grouped by parentheses:

```
(file | directory)
```

This indicates choice: you may supply either a filename or directory name as an argument.

The special heading also includes six properties of the command printed in black (supported) or gray (unsupported):

```
stdin
```

The command reads from standard input, i.e., your keyboard, by default. See “Input and Output” on page 12.
stdout

The command writes to standard output, i.e., your screen, by default. See “Input and Output” on page 12.

-file

When given a dash (-) argument in place of an input filename, the command reads from standard input; and likewise, if the dash is supplied as an output filename, the command writes to standard output. For example, the following wc command line reads the files file1 and file2, then standard input, then file3:

$ wc file1 file2 - file3

--opt

If you supply the command-line option “--” it means “end of options”: anything appearing later on the command line is not an option. This is sometimes necessary to operate on a file whose name begins with a dash, which otherwise would be (mistakenly) treated as an option. For example, if you have a file named -foo, the command wc -foo will fail because -foo will be treated as an (invalid) option. wc -- -foo works. If a command does not support “--”, you can prepend the current directory path “./” to the filename so the dash is no longer the first character:

$ wc ./-foo

--help

The option --help makes the command print a help message explaining proper usage, then exit.

--version

The option --version makes the command print its version information and exit.
Shell prompts

Some commands in this book can be run successfully only by the superuser, a special user with permission to do anything on the system. In this case, we use a hash mark (#) as the shell prompt:

```
# superuser command goes here
```

Otherwise, we will use the dollar sign prompt, indicating an ordinary user:

```
$ ordinary command goes here
```

Keystrokes

Throughout the book, we use certain symbols to indicate keystrokes. Like many other Linux documents, we use the ^ symbol to mean “press and hold the Control (Ctrl) key,” so for example, ^D (pronounced “control D”) means “press and hold the Control key and type D.” We also write ESC to mean “press the Escape key.” Keys like Enter and the space bar should be self-explanatory.

Your friend, the echo command

In many of our examples, we’ll print information to the screen with the echo command, which we’ll formally describe in “Screen Output” on page 168. echo is one of the simplest commands: it merely prints its arguments on standard output, once those arguments have been processed by the shell.

```
$ echo My dog has fleas
My dog has fleas
$ echo My name is $USER
My name is smith
```

Getting Help

If you need more information than this book provides, there are several things you can do.
Run the **man** command

The **man** command displays an online manual page, or *manpage*, for a given program. For example, to learn about listing files with **ls**, run:

```
$ man ls
```

To search for manpages by keyword for a particular topic, use the `-k` option followed by the keyword:

```
$ man -k database
```

Run the **info** command

The **info** command is an extended, hypertext help system covering many Linux programs.

```
$ info ls
```

While **info** is running, some useful keystrokes are:

- To get help, type **h**
- To quit, type **q**
- To page forward and backward, use the space bar and Backspace keys
- To jump between hyperlinks, press **TAB**
- To follow a hyperlink, press **Enter**

If **info** has no documentation on a given program, it displays the program’s manpage. For a listing of available documentation, type **info** by itself. To learn how to navigate the info system, type **info info**.

Use the **--help** option (if any)

Many Linux commands respond to the option **--help** by printing a short help message. Try:

```
$ ls --help
```

If the output is longer than the screen, pipe it into the **less** program to display it in pages (press **q** to quit):

```
$ ls --help | less
```
Examine the directory `/usr/share/doc`

This directory contains supporting documents for many programs, usually organized by program name and version. For example, files for the text editor `emacs`, version 23, are likely found (depending on distro) in `/usr/share/doc/emacs23`.

**GNOME and KDE Help**

For help with GNOME or KDE, visit [http://www.gnome.org](http://www.gnome.org) or [http://www.kde.org](http://www.kde.org).

**Distro-specific websites**

Most Linux distros have an official site that includes documentation, discussion forums for questions and answers, and other resources. Simply enter the distro name (e.g., “Ubuntu”) into any popular search engine to find its web site. You can also visit the web site for this book: [http://shop.oreilly.com/product/0636920023029.do](http://shop.oreilly.com/product/0636920023029.do).

**Linux help sites**

Many web sites answer Linux questions, such as [http://www.linuxquestions.org](http://www.linuxquestions.org), [http://unix.stackexchange.com](http://unix.stackexchange.com), [http://www.linuxhelp.net](http://www.linuxhelp.net), and [http://www.linuxforums.org](http://www.linuxforums.org).

**Web search**

To decipher a specific Linux error message, enter the message into a web search engine, word for word, and you will likely find helpful results.

---

**Linux: A First View**

Linux has four major parts:

**The kernel**

The low-level operating system, handling files, disks, networking, and other necessities we take for granted. Most users rarely notice the kernel.

**Supplied programs**

Thousands of programs for file manipulation, text editing, mathematics, web browsing, audio, video, computer
programming, typesetting, encryption, DVD burning... you name it.

**The shell**

A user interface for typing commands, executing them, and displaying the results. Linux has various shells: the Bourne shell, Korn shell, C shell, and others. This book focuses on bash, the Bourne-Again Shell, which is often the default for user accounts. However, all these shells have similar basic functions.

**X**

A graphical system that provides windows, menus, icons, mouse support, and other familiar GUI elements. More complex graphical environments are built on X; the most popular are KDE and GNOME. We’ll discuss a few programs that open X windows to run.

This book focuses on the second and third parts: supplied programs and the shell.

**The Graphical Desktop**

When you log into a Linux system, you’re likely to be greeted by a graphical desktop\(^1\) like Figure 2, which contains:

- A main menu or taskbar. Depending on your distro and system settings, this might be at the top, bottom, or side of the screen.
- Desktop icons representing the computer, a folder representing your home directory for personal files, a trash can, and more.
- Icons to run applications, such as the Firefox web browser and the Thunderbird email program.
- Controls for opening and closing windows and running multiple desktops at once.

\(^1\) Unless you’re logging in remotely over the network, in which case you’ll see just a command prompt, waiting for you to type a command.
- A clock and other small, informational icons.

Figure 2. Graphical desktops (CentOS Linux with GNOME, Ubuntu with KDE). Desktops can look wildly different, depending on your distro and system settings.
Linux systems have several graphical interfaces, the most common being GNOME and KDE. Identify yours by clicking your system’s equivalent of a main menu or start menu and looking for the words GNOME, KDE, Kubuntu (KDE on Ubuntu Linux), or similar.

**Running a Shell**

The icons and menus in GNOME and KDE are, for some users, the primary way to work with Linux. This is fine for simple tasks like reading email and browsing the Web. Nevertheless, the true power of Linux lies beneath this graphical interface, in the shell.

To get the most out of Linux, take the time to become proficient with the shell. (That’s what this book is all about.) It might initially be more difficult than icons and menus, but once you’re used to it, the shell is quite easy to use and very powerful.

To run a shell within GNOME, KDE, or any other graphical interface for Linux, you need to open a shell window: a window with a shell running in it. Figure 2 shows two shell windows with “$” shell prompts, awaiting your commands. Look through your system menus for an application to do this. Typical menu items are Terminal, xterm, gnome-terminal, konsole, and uxterm.

Don’t confuse the window program (like konsole) with the shell running inside it. The window is just a container—possibly with fancy features of its own—but the shell is what prompts you for commands and runs them.

If you’re not running a graphical interface—say, you’re logging in remotely over the network, or directly over an attached terminal—a shell will run immediately when you log in. No shell window is required.

This was just a quick introduction. We’ll discuss more details in “The Shell” on page 22, and cover more powerful constructs in “Programming with Shell Scripts” on page 195.
**Input and Output**

Most Linux commands accept input and produce output. Input can come from files or from *standard input*, which is usually your keyboard. Likewise, output is written to files or to *standard output*, which is usually your shell window or screen. Error messages are treated specially and displayed on *standard error*, which also is usually your screen but kept separate from standard output. Later we’ll see how to *redirect* standard input, output, and error to and from files or pipes. But let’s get our vocabulary straight. When we say a command “reads,” we mean from standard input unless we say otherwise. And when a command “writes” or “prints,” we mean on standard output, unless we’re talking about computer printers.

**Users and Superusers**

Linux is a multiuser operating system: multiple people can use a single Linux computer at the same time. On a given computer, each user is identified by a unique *username*, like “smith” or “funkyguy,” and owns a (reasonably) private part of the system for doing work. There is also a special user named *root*—the *superuser*—who has the privileges to do anything at all on the system. Ordinary users are restricted: though they can run most programs, in general they can modify only the files they own. The superuser, on the other hand, can create, modify, or delete any file and run any program.

To become the superuser, you needn’t log out and log back in; just run the *su* command (see “Becoming the Superuser” on page 138) and provide the superuser password:

```
$ su -l
Password: *******
#
```

2. For example, you can capture standard output in a file and still have standard error messages appear on screen.
The superuser prompt (#) indicates that you’re ready to run superuser commands. Alternatively, run the `sudo` command (if your system is configured to use it), which executes a single command as the superuser, then returns control to the original user:

```
$ sudo ls /private/secrets
Password: ******
secretfile1    secretfile2
$ View a protected directory
```

It worked!

The Filesystem

To make use of any Linux system, you need to be comfortable with Linux files and directories (a.k.a. folders). In a “windows and icons” system, the files and directories are obvious on screen. With a command-line system like the Linux shell, the same files and directories are still present but are not constantly visible, so at times you must remember which directory you are “in” and how it relates to other directories. You’ll use shell commands like `cd` and `pwd` to “move” between directories and keep track of where you are.

Let’s cover some terminology. As we’ve said, Linux files are collected into directories. The directories form a hierarchy, or tree, as in Figure 3: one directory may contain other directories, called subdirectories, which may themselves contain other files and subdirectories, and so on, into infinity. The topmost directory is called the root directory and is denoted by a slash (/).3

We refer to files and directories using a “names and slashes” syntax called a path. For instance, this path:

```
/one/two/three/four
```

refers to the root directory /, which contains a directory called `one`, which contains a directory `two`, which contains a directory

3. In Linux, all files and directories descend from the root. This is unlike Windows or DOS, in which different devices are accessed by drive letters.
three, which contains a final file or directory, four. If a path begins with the root directory, it’s called an absolute path, and if not, it’s a relative path. More on this in a moment.

Whenever you are running a shell, that shell is working “in” some directory (in an abstract sense). More technically, your shell has a current working directory, and when you run commands in that shell, they operate relative (there’s that word again) to the directory. More specifically, if you refer to a relative file path in that shell, it is relative to your current working directory. For example, if your shell is “in” the directory /one/two/three, and you run a command that refers to a file myfile, then the file is really /one/two/three/myfile. Likewise, a relative path a/b/c would imply the true path /one/two/three/a/b/c.

Two special directories are denoted . (a single period) and .. (two periods in a row). The former means your current directory, and the latter means your parent directory, one level above. So if your current directory is /one/two/three, then . refers to this directory and .. refers to /one/two.

You “move” your shell from one directory to another using the cd command:

```
$ cd /one/two/three
```

More technically, this command changes your shell’s current working directory to be /one/two/three. This is an absolute
change (since the directory begins with “/”); of course you can make relative moves as well:

$ cd d
   Enter subdirectory d
$ cd ../mydir
   Go up to my parent, then into directory mydir

File and directory names may contain most characters you expect: capital and lowercase letters\(^4\), numbers, periods, dashes, underscores, and most symbols (but not “/”, which is reserved for separating directories). For practical use, however, avoid spaces, asterisks, parentheses, and other characters that have special meaning to the shell. Otherwise, you’ll need to quote or escape these characters all the time. (See “Quoting” on page 29.)

**Home Directories**

Users’ personal files are often found in `/home` (for ordinary users) or `/root` (for the superuser). Your home directory is typically `/home/your-username` : `/home/smith`, `/home/jones`, etc. There are several ways to locate or refer to your home directory.

**cd**

With no arguments, the `cd` command returns you (i.e., sets the shell’s working directory) to your home directory.

**HOME variable**

The environment variable `HOME` (see “Shell variables” on page 25) contains the name of your home directory.

```bash
$ echo $HOME
/home/smith
```

When used in place of a directory, a lone tilde is expanded by the shell to the name of your home directory.

```bash
$ echo ~
/home/smith
```

\(^4\) Linux filenames are case-sensitive, so capital and lowercase letters are not equivalent.
When followed by a username (as in `~fred`), the shell expands this string to be the user’s home directory:

```
$ cd ~fred
$ pwd
/home/fred
```

The “print working directory” command

**System Directories**

A typical Linux system has tens of thousands of system directories. These directories contain operating system files, applications, documentation, and just about everything except personal user files (which typically live in `/home`).

Unless you’re a system administrator, you’ll rarely visit most system directories—but with a little knowledge you can understand or guess their purposes. Their names often contain three parts, which we’ll call the scope, category, and application. (These are not standard terms, but they’ll help you understand things.) For example, the directory `/usr/local/share/emacs`, which contains local data for the emacs text editor, has scope `/usr/local` (locally installed system files), category `share` (program-specific data and documentation), and application `emacs` (a text editor), shown in Figure 4. We’ll explain these three parts, slightly out of order.

![Figure 4. Directory scope, category, and application](image.png)

**Directory path part 1: category**

A category tells you the types of files found in a directory. For example, if the category is `bin`, you can be reasonably assured that the directory contains programs. Common categories are:
Categories for programs

**bin** Programs (usually binary files)

**sbin** Programs (usually binary files) intended to be run by the superuser

**lib** Libraries of code used by programs

**libexec** Programs invoked by other programs, not usually by users; think “library of executable programs”

Categories for documentation

**doc** Documentation

**info** Documentation files for emacs’s built-in help system

**man** Documentation files (manual pages) displayed by the man program; the files are often compressed, or sprinkled with typesetting commands for man to interpret

**share** Program-specific files, such as examples and installation instructions

Categories for configuration

**etc** Configuration files for the system (and other miscellaneous stuff)

**init.d** Configuration files for booting Linux

**rc.d** Configuration files for booting Linux; also *rc1.d, rc2.d, ...

Categories for programming

**include** Header files for programming

**src** Source code for programs

Categories for web files

**cgi-bin** Scripts/programs that run on web pages

**html** Web pages

**public_html** Web pages, typically in users’ home directories

**www** Web pages

Categories for display

**fonts** Fonts (surprise!)

**X11** X window system files

Categories for hardware

**dev** Device files for interfacing with disks and other hardware
categories for runtime files

<table>
<thead>
<tr>
<th>Category</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>var</strong></td>
<td>Files specific to this computer, created and updated as the computer runs</td>
</tr>
<tr>
<td><strong>lock</strong></td>
<td>Lock files, created by programs to say, “I am running”; the existence of a lock file may prevent another program, or another instance of the same program, from running or performing an action</td>
</tr>
<tr>
<td><strong>log</strong></td>
<td>Log files that track important system events, containing error, warning, and informational messages</td>
</tr>
<tr>
<td><strong>mail</strong></td>
<td>Mailboxes for incoming mail</td>
</tr>
<tr>
<td><strong>run</strong></td>
<td>PID files, which contain the IDs of running processes; these files are often consulted to track or kill particular processes</td>
</tr>
<tr>
<td><strong>spool</strong></td>
<td>Files queued or in transit, such as outgoing email, print jobs, and scheduled jobs</td>
</tr>
<tr>
<td><strong>tmp</strong></td>
<td>Temporary storage for programs and/or people to use</td>
</tr>
<tr>
<td><strong>proc</strong></td>
<td>Operating system state: see “Operating System Directories” on page 19</td>
</tr>
</tbody>
</table>

Directory path part 2: scope

The scope of a directory path describes, at a high level, the purpose of an entire directory hierarchy. Some common ones are:

- **/** System files supplied with Linux (pronounced “root”)
- **/usr** More system files supplied with Linux (pronounced “user”)
- **/usr/games** Games (surprise!)
- **/usr/local** System files developed “locally,” either for your organization or your individual computer
- **/usr/X11R6** Files pertaining to the X window system

So for a category like **lib** (libraries), your Linux system might have directories `/lib`, `/usr/lib`, `/usr/local/lib`, `/usr/games/lib`, and `/usr/X11R6/lib`. 
There isn’t a clear distinction between / and /usr in practice, but there is a sense that / is “lower-level” and closer to the operating system. So /bin contains fundamental programs like ls and cat, /usr/bin contains a wide variety of applications supplied with your Linux distribution, and /usr/local/bin contains programs your system administrator chose to install. These are not hard-and-fast rules but typical cases.

**Directory path part 3: application**

The application part of a directory path, if present, is usually the name of a program. After the scope and category (say, /usr/local/doc), a program may have its own subdirectory (say, /usr/local/doc/myprogram) containing files it needs.

**Operating System Directories**

Some directories support the Linux kernel, the lowest-level part of the Linux operating system.

/boot

Files for booting the system. This is where the kernel lives, typically named /boot/vmlinuz.

/lost+found

Damaged files that were rescued by a disk recovery tool.

/proc

Describes currently running processes; for advanced users.

The files in /proc provide views into the running kernel and have special properties. They always appear to be zero sized, read-only, and dated now:

```bash
$ ls -l /proc/version
-r--r--r-- 1 root root 0 Oct 3 22:55 /proc/version
```

However, their contents magically contain information about the Linux kernel:

```bash
$ cat /proc/version
Linux version 2.6.32-71.el6.i686 ...
```
Files in /proc are used mostly by programs, but feel free to explore them. Here are some examples:

/proc/ioports  A list of your computer’s input/output hardware.

/proc/version  The operating system version. The `uname` command prints the same information.

/proc/uptime  System uptime, i.e., seconds elapsed since the system was last booted. Run the `uptime` command for a more human-readable result.

/proc/nnn  Where `nnn` is a positive integer, information about the Linux process with process ID `nnn`.

/proc/self  Information about the current process you’re running; a symbolic link to a /proc/nnn file, automatically updated. Try `ls -l /proc/self` several times in a row: you’ll see /proc/self changing where it points.

File Protections

A Linux system may have many users with login accounts. To maintain privacy and security, most users can access only some files on the system, not all. This access control is embodied in two questions:

Who has permission?

Every file and directory has an owner who has permission to do anything with it. Typically the user who created a file is its owner, but relationships can be more complex.

Additionally, a predefined group of users may have permission to access a file. Groups are defined by the system administrator and are covered in “Group Management” on page 140.

Finally, a file or directory can be opened to all users with login accounts on the system. You’ll also see this set of users called the world or simply other.

What kind of permission is granted?

File owners, groups, and the world may each have permission to read, write (modify), and execute (run) particular files. Permissions also extend to directories, which
users may read (access files within the directory), write
(create and delete files within the directory), and execute
(enter the directory with cd).

To see the ownership and permissions of a file, run:

```
$ ls -l myfile
-rw-r--r-- 1 smith smith 7384 Jan 04 22:40 myfile
```

To see the ownership and permissions of a directory, run:

```
$ ls -ld dirname
drwxr-x--- 3 smith smith 4096 Jan 08 15:02 dirname
```

In the output, the file permissions are the 10 leftmost charac-
ters, a string of r (read), w (write), x (execute), other letters, and
dashes. For example:

```
-rwxr-x---
```

Here’s what these letters and symbols mean.

<table>
<thead>
<tr>
<th>Position</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>File type: - = file, d = directory, l = symbolic link, p = named pipe, c = character device, b = block device</td>
</tr>
<tr>
<td>2–4</td>
<td>Read, write, and execute permissions for the file’s owner</td>
</tr>
<tr>
<td>5–7</td>
<td>Read, write, and execute permissions for the file’s group</td>
</tr>
<tr>
<td>8–10</td>
<td>Read, write, and execute permissions for all other users</td>
</tr>
</tbody>
</table>

So our example -rwxr-x--- means a file that can be read, writ-
ten, and executed by the owner, read and executed by the
group, and not accessed at all by the rest of the world. We
describe ls in more detail in “Basic File Opera-
tions” on page 36. To change the owner, group ownership,
or permissions of a file, use the chown, chgrp, and chmod com-
mands, respectively, as described in “File Proper-
ties” on page 59.
The Shell

In order to run commands on a Linux system, you’ll need somewhere to type them. That “somewhere” is called the shell, which is Linux’s command-line user interface: you type a command and press Enter, and the shell runs whatever program (or programs) you’ve requested. (See “Running a Shell” on page 11 to learn how to open a shell window.)

For example, to see who’s logged in, you could execute this command in a shell:

```
$ who
silver        :0    Sep 23 20:44
byrnes    pts/0    Sep 15 13:51
barrett   pts/1    Sep 22 21:15
silver    pts/2    Sep 22 21:18
```

(The dollar sign is the shell prompt, which means the shell is ready to run a command.) A single command can also invoke several programs at the same time, and even connect programs together so they interact. Here’s a command that redirects the output of the `who` program to become the input of the `wc` program, which counts lines of text in a file; the result is the number of lines in the output of `who`:

```
$ who | wc -l
4
```

telling you how many users are logged in.\(^5\) The vertical bar, called a pipe, makes the connection between `who` and `wc`.

A shell is actually a program itself, and Linux has several. We focus on bash (the Bourne-Again Shell), located in `/bin/bash`, which is usually the default in Linux distros.

---

\(^{5}\) Actually, how many interactive shells those users are running. If a user has two shells running, like the user silver in our example, he’ll have two lines of output from `who`. 
The Shell Versus Programs

When you run a command, it might invoke a Linux program (like `who`), or instead it might be a built-in command, a feature of the shell itself. You can tell the difference with the `type` command:

```
$ type who
who is /usr/bin/who
$ type cd
cd is a shell builtin
```

It is helpful to know what the shell provides versus what Linux does. The next few sections describe features of the shell.

Selected Features of the bash Shell

A shell does much more than simply run commands. It also has powerful features to make this task easier: wildcards for matching filenames, a “command history” to recall previous commands quickly, pipes for making the output of one command become the input of another, variables for storing values for use by the shell, and more. Take the time to learn these features, and you will become faster and more productive with Linux. Let’s skim the surface and introduce you to these useful tools. (For full documentation, run `info bash`.)

Wildcards

Wildcards are a shorthand for sets of files with similar names. For example, `a*` means all files whose names begin with lowercase “a”. Wildcards are “expanded” by the shell into the actual set of filenames they match. So if you type:

```
$ ls a*
```

the shell first expands `a*` into the filenames that begin with “a” in your current directory, as if you had typed:

```
$ ls aardvark adamantium apple
```

`ls` never knows you used a wildcard: it sees only the final list of filenames after the shell expands the wildcard. Importantly,
this means *every* Linux command, regardless of its origin, works with wildcards and other shell features.

Wildcards never match two characters: a leading period, and the directory slash (/). These must be given literally, as in `.pro*` to match `.profile`, or `/etc/*conf` to match all filenames ending in `conf` in the `/etc` directory.

---

### Dot Files

Filenames with a leading period, called *dot files*, are special in Linux. When you name a file beginning with a period, it will not be displayed by some programs:

- `ls` will omit the file from directory listings, unless you provide the `-a` option
- Shell wildcards do not match a leading period

Effectively, dot files are hidden unless you explicitly ask to see them. As a result, sometimes they are called “hidden files.”

---

**Wildcard** | **Meaning**
--- | ---
* | Zero or more consecutive characters
? | Any single character
[set] | Any single character in the given set, most commonly a sequence of characters, like `[aeiouAEIOU]` for all vowels, or a range with a dash, like `[A-Z]` for all capital letters
[^set] | Any single character *not* in the given set (as in the earlier example)
![set] | Same as ^

When using character sets, if you want to include a literal dash in the set, put it first or last. To include a literal closing square bracket in the set, put it first. To include a ^ or ! symbol literally, don’t put it first.
Brace expansion

Similar to wildcards, expressions with curly braces also expand to become multiple arguments to a command. The comma-separated expression:

\{X,YY,ZZZ\}

expands first to X, then YY, and finally ZZZ within a command line, like this:

```
$ echo sand{X,YY,ZZZ}wich
sandXwich sandYYwich sandZZZwich
```

Braces work with any strings, unlike wildcards, which are limited to filenames. The preceding example works regardless of which files are in the current directory.

Shell variables

You can define variables and their values by assigning them:

```
$ MYVAR=3
```

To refer to a value, simply place a dollar sign in front of the variable name:

```
$ echo $MYVAR
3
```

Some variables are standard and commonly defined by your shell upon login.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>DISPLAY</td>
<td>The name of your X window display</td>
</tr>
<tr>
<td>HOME</td>
<td>Your home directory, such as /home/smith</td>
</tr>
<tr>
<td>LOGNAME</td>
<td>Your login name, such as smith</td>
</tr>
<tr>
<td>MAIL</td>
<td>Your incoming mailbox, such as /var/spool/mail/smith</td>
</tr>
<tr>
<td>OLDPWD</td>
<td>Your shell’s previous directory, prior to the last cd command</td>
</tr>
<tr>
<td>PATH</td>
<td>Your shell search path: directories separated by colons</td>
</tr>
<tr>
<td>PWD</td>
<td>Your shell’s current directory</td>
</tr>
<tr>
<td>SHELL</td>
<td>The path to your shell, e.g., /bin/bash</td>
</tr>
<tr>
<td>Variable</td>
<td>Meaning</td>
</tr>
<tr>
<td>----------</td>
<td>---------</td>
</tr>
<tr>
<td>TERM</td>
<td>The type of your terminal, e.g., xterm or vt100</td>
</tr>
<tr>
<td>USER</td>
<td>Your login name</td>
</tr>
</tbody>
</table>

To see a shell’s variables, run:

$ printenv

The scope of the variable (i.e., which programs know about it) is, by default, the shell in which it’s defined. To make a variable and its value available to other programs your shell invokes (i.e., subshells), use the `export` command:

$ export MYVAR

or the shorthand:

$ export MYVAR=3

Your variable is now called an environment variable, since it’s available to other programs in your shell’s “environment.” So in the preceding example, the exported variable `MYVAR` is available to all programs run by that same shell (including shell scripts: see “Variables” on page 196).

To make a variable value available to a specific program just once, prepend `variable=value` to the command line:

$ printenv HOME
/home/smith

$ HOME=/home/sally printenv HOME
/home/sally

$ printenv HOME
/home/smith

The original value is unaffected

**Search path**

Programs are scattered all over the Linux filesystem, in directories like `/bin` and `/usr/bin`. When you run a program via a shell command, how does the shell find it? The critical variable `PATH` tells the shell where to look. When you type any command:

$ who
the shell has to find the who program by searching through Linux directories. The shell consults the value of PATH, which is a sequence of directories separated by colons:

```
$ echo $PATH
/usr/local/bin:/bin:/usr/bin:/home/smith/bin
```

and looks for the who command in each of these directories. If it finds who (say, /usr/bin/who), it runs the command. Otherwise, it reports:

```
bash: who: command not found
```

To add directories to your shell’s search path temporarily, modify its PATH variable. For example, to append /usr/sbin to your shell’s search path:

```
$ PATH=$PATH:/usr/sbin
$ echo $PATH
/usr/local/bin:/bin:/usr/bin:/home/smith/bin:/usr/sbin
```

This change affects only the current shell. To make it permanent, modify the PATH variable in your startup file ~/.bash_profile, as explained in “Tailoring Shell Behavior” on page 36. Then log out and log back in.

### Aliases

The built-in command alias defines a convenient shorthand for a longer command, to save typing. For example:

```
$ alias ll='ls -l'
```

defines a new command ll that runs ls -l:

```
$ ll
total 436
-rw-r--r-- 1 smith 3584 Oct 11 14:59 file1
-rwxr-xr-x 1 smith 72 Aug 6 23:04 file2
...
```

Define aliases in your ~/.bashrc file (see “Tailoring Shell Behavior” on page 36) to be available whenever you log in.\(^6\) To list all your aliases, type alias. If aliases don’t seem powerful

---

\(^6\) Some setups use a separate file, ~/.bash_aliases, for this purpose.
enough for you (since they have no parameters or branching), see “Programming with Shell Scripts” on page 195, run info bash, and read up on “shell functions.”

**Input/output redirection**

The shell can redirect standard input, standard output, and standard error to and from files. In other words, any command that reads from standard input can have its input come from a file instead with the shell’s `<` operator:

```
$ mycommand < infile
```

Likewise, any command that writes to standard output can write to a file instead:

```
$ mycommand > outfile
$ mycommand >> outfile
```

Create/overwrite outfile

Append to outfile

A command that writes to standard error can have its output redirected to a file as well, while standard output still goes to the screen:

```
$ mycommand 2> errorfile
```

To redirect both standard output and standard error to files:

```
$ mycommand > outfile 2> errorfile
$ mycommand >& outfile
```

Separate files

Single file

**Pipes**

You can redirect the standard output of one command to be the standard input of another, using the shell’s pipe (`|`) operator. For example:

```
$ who | sort
```

sends the output of `who` into the `sort` program, printing an alphabetically sorted list of logged-in users. Multiple pipes work too. Here we sort the output of `who` again, extract the first column of information (using `awk`), and display the results one page at a time (using `less`):

```
$ who | sort | awk '{print $1}' | less
```
Combining commands

To invoke several commands in sequence on a single command line, separate them with semicolons:

```
$ command1 ; command2 ; command3
```

To run a sequence of commands as before, but stop execution if any of them fails, separate them with `&&` ("and") symbols:

```
$ command1 && command2 && command3
```

To run a sequence of commands, stopping execution as soon as one succeeds, separate them with `||` ("or") symbols:

```
$ command1 || command2 || command3
```

Quoting

Normally, the shell treats whitespace simply as separating the words on the command line. If you want a word to contain whitespace (e.g., a filename with a space in it), surround it with single or double quotes to make the shell treat it as a unit. Single quotes treat their contents literally, while double quotes let shell constructs be evaluated, such as variables:

```
$ echo 'The variable HOME has value $HOME'
The variable HOME has value $HOME
$ echo "The variable HOME has value $HOME"
The variable HOME has value /home/smith
```

Backquotes ("backticks") cause their contents to be evaluated as a shell command. The contents are then replaced by the standard output of the command:

```
$ whoami Program that prints your username
smith
$ echo My name is `whoami`
My name is smith
```

Escaping

If a character has special meaning to the shell but you want it used literally (e.g., * as a literal asterisk rather than a wildcard), precede the character with the backward slash "\" character. This is called escaping the special character:
$ echo a*  
  aardvark  agnostic  apple  
$ echo a\*  
  a*  
$ echo "I live in $HOME"  
  I live in /home/smith  
$ echo "I live in \$HOME"  
  I live in $HOME

As a wildcard, matching “a” filenames
As a literal asterisk
Dollar sign means a variable value
A literal dollar sign

You can also escape control characters (tabs, newlines, ^D, and so forth) to have them used literally on the command line, if you precede them with ^V. This is particularly useful for tab (^I) characters, which the shell would otherwise use for filename completion (see “Filename completion” on page 31).

$ echo "There is a tab between here^V^I and here"
  There is a tab between here and here

Command-line editing

Bash lets you edit the command line you’re working on, using keystrokes inspired by the text editors emacs and vi (see “File Creation and Editing” on page 54). To enable command-line editing with emacs keys, run this command (and place it in your ~/.bash_profile to make it permanent):

$ set -o emacs

For vi keys:

$ set -o vi

<table>
<thead>
<tr>
<th>emacs keystroke</th>
<th>vi keystroke (after ESC)</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>^P or up arrow</td>
<td>k</td>
<td>Go to previous command</td>
</tr>
<tr>
<td>^N or down arrow</td>
<td>j</td>
<td>Go to next command</td>
</tr>
<tr>
<td>^F or right arrow</td>
<td>l</td>
<td>Go forward one character</td>
</tr>
<tr>
<td>^B or left arrow</td>
<td>h</td>
<td>Go backward one character</td>
</tr>
<tr>
<td>^A</td>
<td>0</td>
<td>Go to beginning of line</td>
</tr>
<tr>
<td>^E</td>
<td>$</td>
<td>Go to end of line</td>
</tr>
<tr>
<td>^D</td>
<td>x</td>
<td>Delete next character</td>
</tr>
<tr>
<td>^U</td>
<td>^U</td>
<td>Erase entire line</td>
</tr>
</tbody>
</table>
**Command history**

You can recall previous commands you’ve run—that is, the shell’s *history*—and re-execute them. Some useful history-related commands are listed below.

<table>
<thead>
<tr>
<th>Command</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>history</code></td>
<td>Print your history</td>
</tr>
<tr>
<td><code>history N</code></td>
<td>Print the most recent <em>N</em> commands in your history</td>
</tr>
<tr>
<td><code>history -c</code></td>
<td>Clear (delete) your history</td>
</tr>
<tr>
<td><code>!!</code></td>
<td>Re-run previous command</td>
</tr>
<tr>
<td><code>!N</code></td>
<td>Re-run command number <em>N</em> in your history</td>
</tr>
<tr>
<td><code>! -N</code></td>
<td>Re-run the command you typed <em>N</em> commands ago</td>
</tr>
<tr>
<td><code>!$</code></td>
<td>Represents the last parameter from the previous command; great for checking that files are present before removing them:</td>
</tr>
<tr>
<td><code>$ ls a*</code></td>
<td>acorn.txt   affidavit</td>
</tr>
<tr>
<td></td>
<td>$ rm !$</td>
</tr>
<tr>
<td><code>!*</code></td>
<td>Represents all parameters from the previous command:</td>
</tr>
<tr>
<td><code>$ ls a b c</code></td>
<td>a   b   c</td>
</tr>
<tr>
<td></td>
<td>$ wc  !*</td>
</tr>
<tr>
<td></td>
<td>103   252   2904 a</td>
</tr>
<tr>
<td></td>
<td>12    25    384 b</td>
</tr>
<tr>
<td></td>
<td>25473 65510 988215 c</td>
</tr>
<tr>
<td></td>
<td>25588 65787 991503 total</td>
</tr>
</tbody>
</table>

**Filename completion**

Press the TAB key while you are in the middle of typing a filename, and the shell will automatically complete (finish typing) the filename for you. If several filenames match what you’ve typed so far, the shell will beep, indicating the match is ambiguous. Immediately press TAB again and the shell will present the alternatives. Try this:

```
$ cd /usr/bin
$ ls un<TAB><TAB>
```
The shell will display all files in /usr/bin that begin with un, such as uniq, units, and unzip. Type a few more characters to disambiguate your choice and press TAB again.

**Shell Job Control**

- `jobs` List your jobs.
- `&` Run a job in the background.
- `^Z` Suspend the current (foreground) job.
- `suspend` Suspend a shell.
- `fg` Unsuspend a job: bring it into the foreground.
- `bg` Make a suspended job run in the background.

All Linux shells have **job control**: the ability to run programs in the background (multitasking behind the scenes) and foreground (running as the active process at your shell prompt). A **job** is simply the shell’s unit of work. When you run a command interactively, your current shell tracks it as a job. When the command completes, the associated job disappears. Jobs are at a higher level than Linux processes; the Linux operating system knows nothing about them. They are merely constructs of the shell. Some important vocabulary about job control is:

- **Foreground job**
  - Running in a shell, occupying the shell prompt so you cannot run another command
- **Background job**
  - Running in a shell, but not occupying the shell prompt, so you can run another command in the same shell
- **Suspend**
  - To stop a foreground job temporarily
- **Resume**
  - To cause a suspended job to start running again
**jobs**

The built-in command `jobs` lists the jobs running in your current shell.

```
$ jobs
 [1]- Running emacs myfile &
 [2]+ Stopped su
```

The integer on the left is the job number, and the plus sign identifies the default job affected by the `fg` (foreground) and `bg` (background) commands.

---

**&**

Placed at the end of a command line, the ampersand causes the given command to run as a background job.

```
$ emacs myfile &
 [2] 28090
```

The shell’s response includes the job number (2) and the process ID of the command (28090).

---

**^Z**

Typing `^Z` in a shell, while a job is running in the foreground, will suspend that job. It simply stops running, but its state is remembered.

```
$ mybigprogram
 ^Z
 [1]+ Stopped mybigprogram
 $
```

Now you’re ready to type `bg` to put the command into the background, or `fg` to resume it in the foreground.

---

**suspend**

The built-in command `suspend` will suspend the current shell if possible, as if you’d typed `^Z` to the shell itself. For instance, if you’ve run the `su` command and want to return to your original shell:
bg

bg [%jobnumber]
The built-in command `bg` sends a suspended job to run in the background. With no arguments, `bg` operates on the most recently suspended job. To specify a particular job (shown by the `jobs` command), supply the job number preceded by a percent sign:

```
$ bg %2
```

Some types of interactive jobs cannot remain in the background—for instance, if they are waiting for input. If you try, the shell will suspend the job and display:

```
[2]+ Stopped command line here
```

You can now resume the job (with `fg`) and continue.

fg

fg [%jobnumber]
The built-in command `fg` brings a suspended or backgrounded job into the foreground. With no arguments, it selects a job, usually the most recently suspended or backgrounded one. To specify a particular job (as shown by the `jobs` command), supply the job number preceded by a percent sign:

```
$ fg %2
```
Killing a Command in Progress

If you’ve launched a command from the shell running in the foreground, and want to kill it immediately, type ^C. The shell recognizes ^C as meaning, “terminate the current foreground command right now.” So if you are displaying a very long file (say, with the cat command) and want to stop, type ^C:

```
$ cat bigfile
This is a very long file with many lines. Blah blah blah blah blahblahblah ^C
$
```

To kill a program running in the background, you can bring it into the foreground with fg and then type ^C, or alternatively, use the kill command (see “Controlling Processes” on page 121).

Typing ^C is not a friendly way to end a program. If the program has its own way to exit, use that when possible: see the sidebar for details.

Surviving a Kill

Killing a foreground program with ^C may leave your shell in an odd or unresponsive state, perhaps not displaying the keystrokes you type. This happens because the killed program had no opportunity to clean up after itself. If this happens to you:

1. Press ^J to get a shell prompt. This produces the same character as the Enter key (a newline) but will work even if Enter does not.
2. Type the shell command reset (even if the letters don’t appear while you type) and press ^J again to run this command. This should bring your shell back to normal.

^C works only with shells. It will likely have no effect if typed in a window that is not a shell window. Additionally, some programs are written to “catch” the ^C and ignore it: an example is the text editor emacs.
Terminating a Shell

To terminate a shell, either run the `exit` command or type `^D`.7

```
$ exit
```

Tailoring Shell Behavior

To configure all your shells to work in a particular way, edit the files `.bash_profile` and `.bashrc` in your home directory. These files execute each time you log in (`~/.bash_profile`) or open a shell (`~/.bashrc`). They can set variables and aliases, run programs, print your horoscope, or whatever you like.

These two files are examples of shell scripts: executable files that contain shell commands. We’ll cover this feature in more detail in “Programming with Shell Scripts” on page 195.

This concludes our basic overview of Linux and the shell. Now we turn to Linux commands, listing and describing the most useful commands for working with files, processes, users, networking, multimedia, and more.

Basic File Operations

- `ls` List files in a directory.
- `cp` Copy a file.
- `mv` Rename (“move”) a file.
- `rm` Delete (“remove”) a file.
- `ln` Create links (alternative names) to a file.

One of the first things you’ll need to do on a Linux system is manipulate files: copying, renaming, deleting, and so forth.

7. Control-D sends an “end of file” signal to any program reading from standard input. In this case, the program is the shell itself, which terminates.
The `ls` command (pronounced as it is spelled, *ell ess*) lists attributes of files and directories. You can list files in the current directory:

```bash
$ ls
```

in given directories:

```bash
$ ls dir1 dir2 dir3
```
or individually:

```bash
$ ls file1 file2 file3
```

The most important options are `-a`, `-l`, and `-d`. By default, `ls` hides files whose names begin with a dot, as explained in the sidebar “Dot Files” on page 24. The `-a` option displays all files.

```bash
$ ls
myfile1   myfile2
$ ls -a
.hidden_file   myfile1   myfile2
```

The `-l` option produces a long listing:

```
-rw-r--r--    1 smith users       149 Oct 28  2011 my.data
```

that includes, from left to right: the file’s permissions (-rw-r--r--), owner (smith), group (users), size (149 bytes), last modification date (Oct 28 2011) and name. See “File Protections” on page 20 for more information on permissions.

The `-d` option lists information about a directory itself, rather than descending into the directory to list its files.

```bash
$ ls -ld my.dir
drwxr-xr-x    1 smith users      4096 Oct 29  2011 my.dir
```

**Useful options**

- `-a` List all files, including those whose names begin with a dot.
- `-l` Long listing, including file attributes. Add the `-h` option (human-readable) to print file sizes in kilobytes, megabytes, and gigabytes, instead of bytes.
- `-F` Decorate certain filenames with meaningful symbols, indicating their types. Appends “/” to directories, “*x” to executables, “@” to symbolic links, “|” to named

---

**Basic File Operations | 37**
pipes, and “=” to sockets. These are just visual indicators for you, not part of the filenames!

- i  Prepend the inode numbers of the files.
- s  Prepend the size of the file in blocks, useful for sorting files by their size:

   $ ls -s | sort -n

- R  If listing a directory, list its contents recursively.
- d  If listing a directory, do not list its contents, just the directory itself.

---

**cp**

```
cp stdin stdout - file -- opt --help --version
```

`cp [options] files (file | directory)`

The `cp` command normally copies a file:

```
$ cp file file2
```

or copies multiple files into a directory:

```
$ cp file1 file2 file3 file4 destination_directory
```

Using the `-a` option, you can also recursively copy directories.

**Useful options**

- p  Copy not only the file contents, but also the file’s permissions, timestamps and, if you have sufficient permission to do so, its owner and group. (Normally the copies will be owned by you, timestamped now, with permissions set by applying your umask to the original permissions.)
- a  Copy a directory hierarchy recursively, preserving all file attributes and links.
- r  Copy a directory hierarchy recursively. This option does not preserve the files’ attributes such as permissions and timestamps. It does preserve symbolic links.
- i  Interactive mode. Ask before overwriting destination files.
- f  Force the copy. If a destination file exists, overwrite it unconditionally.

---

**mv**

```
mv stdin stdout - file -- opt --help --version
```

`mv [options] source target`

The `mv` (move) command can rename a file:
$ mv file1 file2

or move files and directories into a destination directory:

$ mv file1 file2 dir3 dir4 destination_directory

**Useful options**

- `-i` Interactive mode. Ask before overwriting destination files.
- `-f` Force the move. If a destination file exists, overwrite it unconditionally.

---

```
rm stdin stdout -file --opt --help --version
```

`rm` [options] files | directories

The `rm` (remove) command can delete files:

$ rm file1 file2 file3

or recursively delete directories:

$ rm -r dir1 dir2

**Useful options**

- `-i` Interactive mode. Ask before deleting each file.
- `-f` Force the deletion, ignoring any errors or warnings.
- `-r` Recursively remove a directory and its contents. Use with caution, especially if combined with the `-f` option, as it can wipe out all your files.

---

```
ln stdin stdout -file --opt --help --version
```

`ln` [options] source target

A *link* is a reference to another file, created by the `ln` command. Intuitively, links give the same file multiple names, allowing it to live in two (or more) locations at once.

There are two kinds of links. A *symbolic link* (also called a *symlink* or *soft link*) refers to another file by its path, much like a Windows “shortcut” or a Macintosh “alias.” To create a symbolic link, use the `-s` option:

$ ln -s myfile mysoftlink
If you delete the original file, the now-dangling link will be invalid, pointing to a nonexistent file path. A hard link, on the other hand, is simply a second name for a physical file on disk (in tech talk, it points to the same inode). If you delete the original file, the link still works. Figure 5 illustrates the difference. To create a hard link, type:

```
$ ln myfile myhardlink
```

```
Figure 5. Hard link versus symbolic link
```

Symbolic links can point to files on other disk partitions, since they are just references to file paths; hard links cannot, since an inode on one disk has no meaning on another. Symbolic links can also point to directories, whereas hard links cannot...unless you are the superuser and use the `-d` option.
Useful options

- Make a symbolic link. The default is a hard link.
- i Interactive mode. Ask before overwriting destination files.
- f Force the link. If a destination file exists, overwrite it unconditionally.
- d Create a hard link to a directory (superusers only).

It’s easy to find out where a symbolic link points with either of these commands:

```bash
$ readlink linkname
$ ls -l linkname
```

Directory Operations

**cd**  
Change your current directory.

**pwd**  
Print the name of your current directory, i.e., “where you are now” in the filesystem.

**basename**  
Print the final part of a file path.

**dirname**  
Print a file path without its final part.

**mkdir**  
Create (make) a directory.

**rmdir**  
Delete (remove) an empty directory.

**rm -r**  
Delete a nonempty directory and its contents.

We discussed the directory structure of Linux in “The Filesystem” on page 13. Now we’ll cover commands that create, modify, delete, and manipulate directories within that structure.

```
cd stdin stdout -f --opt --help --version
```

**cd [directory]**

The **cd** (change directory) command sets your current working directory:

```bash
$ cd /usr/games
```
With no directory supplied, `cd` defaults to your home directory:

```
$ cd
```

---

**pwd**

```
stdin  stdout  -file  --opt  --help  --version
```

The `pwd` command prints the absolute path of your current working directory:

```
$ pwd
/users/smith/mydir
```

---

**basename**

```
stdin  stdout  -file  --opt  --help  --version
```

`basename path [suffix]`

The `basename` command prints the final component in a file path:

```
$ basename /users/smith/finances/money.txt
money.txt
```

If you provide an optional suffix, it gets stripped from the result:

```
$ basename /users/smith/finances/money.txt .txt
money
```

---

**dirname**

```
stdin  stdout  -file  --opt  --help  --version
```

dirname `path`

The `dirname` command prints a file path with its final component removed:

```
$ dirname /users/smith/mydir
/users/smith
```

dirname does not change your current working directory. It simply manipulates a string, just like `basename` does.
mkdir

mkdir [options] directories

mkdir creates one or more directories:

$ mkdir directory1 directory2 directory3

Useful options

-p Given a directory path (not just a simple directory name), create any necessary parent directories automatically:

mkdir -p /one/two/three creates /one and /one/two if they don’t already exist, then /one/two/three.

-m mode Create the directory with the given permissions:

$ mkdir -m 0755 mydir

By default, your shell’s umask controls the permissions. See the chmod command in “File Properties” on page 59, and “File Protections” on page 20.

rmdir

rmdir [options] directories

The rmdir (remove directory) command deletes one or more empty directories you name:

$ rmdir /tmp/junk

Useful options

-p If you supply a directory path (not just a simple directory name), delete not only the given directory, but the specified parent directories automatically, all of which must be empty. So rmdir -p /one/two/three will delete not only /one/two/three, but also /one/two and /one.

To delete a nonempty directory and its contents, use (carefully) rm -r directory. Use rm -ri to delete interactively, or rm -rf to annihilate without any error messages or confirmation.
File Viewing

- **cat**: View files in their entirety.
- **less**: View text files one page at a time.
- **head**: View the first lines of a text file.
- **tail**: View the last lines of a text file.
- **nl**: View text files with their lines numbered.
- **strings**: Display text that’s embedded in a binary file.
- **od**: View data in octal (or other formats).
- **xxd**: View data in hexadecimal.
- **acroread**: View PDF files.
- **gv**: View PostScript or PDF files.
- **xdvi**: View TeX DVI files.

In Linux, you’ll encounter various types of files to view: plain text, PostScript, binary data, and more. Here we’ll explain how to view them. Note that commands for viewing graphics files are covered in “Graphics and Screensavers” on page 181, and video files in “Video” on page 188.

**cat**

```bash
stdin  stdout  -file  --opt  --help  --version
```

**cat** [options] [files]

The simplest viewer is **cat**, which just prints its files to standard output, concatenating them (hence the name). Large files will likely scroll off screen, so consider using **less** if you plan to read the output. That being said, **cat** is particularly useful for sending a set of files into a shell pipeline:

```bash
$ cat * | wc
```

**cat** can also manipulate its output in small ways, optionally displaying nonprinting characters, prepending line numbers (though **nl** is more powerful for this purpose), and eliminating whitespace.
Useful options

- T  Print tabs as ^I.
- E  Print newlines as $.
- v  Print other nonprinting characters in a human-readable format.
- n  Prepend line numbers to every line.
- b  Prepend line numbers to nonblank lines.
- s  Squeeze each sequence of blank lines into a single blank line.

\texttt{less} \hspace{1cm} stdin \hspace{1cm} stdout\textsuperscript{8} \hspace{1cm} -file \hspace{1cm} --opt \hspace{1cm} --help \hspace{1cm} --version

\texttt{less [options] [files]}

Use \texttt{less} to view text one “page” at a time (i.e., one window or screenful at a time). It’s great for text files, or as the final command in a shell pipeline with lengthy output.

\$ command1 | command2 | command3 | command4 | less

While running \texttt{less}, type \texttt{h} for a help message describing all its features. Here are some useful keystrokes for paging through files.

<table>
<thead>
<tr>
<th>Keystroke</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>h, H</td>
<td>View a help page.</td>
</tr>
<tr>
<td>Space bar, f, ^V, ^F</td>
<td>Move forward one screenful.</td>
</tr>
<tr>
<td>Enter</td>
<td>Move forward one line.</td>
</tr>
<tr>
<td>b, ^B, ESC-v</td>
<td>Move backward one screenful.</td>
</tr>
<tr>
<td>/</td>
<td>Enter search mode. Follow it with a regular expression and press Enter, and \texttt{less} will look for the first line matching it.</td>
</tr>
<tr>
<td>?</td>
<td>Same as /, but it searches backward in the file.</td>
</tr>
<tr>
<td>n</td>
<td>Repeat your most recent search forward.</td>
</tr>
<tr>
<td>N</td>
<td>Repeat your most recent search backward.</td>
</tr>
</tbody>
</table>

8. Although technically \texttt{less} can be plugged into the middle of a pipeline, or its output redirected to a file, there isn’t much point to doing this.
<table>
<thead>
<tr>
<th>Keystroke</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>v</td>
<td>Edit the current file with your default text editor (the value of environment variable VISUAL, or if not defined, EDITOR, or if not defined, vi).</td>
</tr>
<tr>
<td>&lt;</td>
<td>Jump to beginning of file.</td>
</tr>
<tr>
<td>&gt;</td>
<td>Jump to end of file.</td>
</tr>
<tr>
<td>:n</td>
<td>Jump to next file.</td>
</tr>
<tr>
<td>:p</td>
<td>Jump to previous file.</td>
</tr>
</tbody>
</table>

`less` has a mind-boggling number of features; we’re presenting only the most common. (For instance, `less` will display the contents of a compressed Zip file: try `less myfile.zip`.) The manpage is recommended reading.

**Useful options**

- `-c` Clear the screen before displaying the next page. This avoids scrolling and may be more comfortable on the eyes.
- `-m` Print a more verbose prompt, displaying the percentage of the file displayed so far.
- `-N` Display line numbers.
- `-r` Display control characters literally; normally `less` converts them to a human-readable format.
- `-s` Squeeze multiple, adjacent blank lines into a single blank line.
- `-S` Truncate long lines to the width of the screen, instead of wrapping.

```
head stdin stdout -file --opt --help --version
```

head [options] [files]

The `head` command prints the first 10 lines of a file: great for previewing the contents.

```
$ head myfile
$ head * | less

Preview all files in the current directory
```
It’s also good for previewing the first few lines of output from a pipeline:

```
$ grep 'E' very-big-file | head
```

**Useful options**

- `-N` Print the first $N$ lines instead of 10.
- `-n N` Print the first $N$ lines instead of 10.
- `-c N` Print the first $N$ bytes of the file.
- `-q` Quiet mode: when processing more than one file, don’t print a banner above each file. Normally, `head` prints a banner containing the filename.

---

**tail**

```
tail [options] [files]
```

The `tail` command prints the last 10 lines of a file, and does other tricks as well.

```
$ tail myfile
```

The ultra-useful `-f` option causes `tail` to watch a file actively while another program is writing to it, displaying new lines as they are written to the file. This is invaluable for watching log files in active use:

```
$ tail -f /var/log/messages
```

**Useful options**

- `-N` Print the last $N$ lines of the file instead of 10.
- `-n N` Print the last $N$ lines of the file instead of 10.
- `-n +N` Print all lines except the first $N$.
- `-c N` Print the last $N$ bytes of the file.
- `-f` Keep the file open, and whenever lines are appended to the file, print them. This is extremely useful. Add the `--retry` option if the file doesn’t exist yet, but you want to wait for it to exist.
- `-q` Quiet mode: when processing more than one file, don’t print a banner above each file. Normally `tail` prints a banner containing the filename.
nl copies its files to standard output, prepending line numbers.

```
$ nl myfile
  1  Once upon a time, there was
  2  a little operating system named
  3  Linux, which everybody loved.
```

It’s more flexible than `cat` with its `-n` and `-b` options, providing an almost bizarre amount of control over the numbering. `nl` can be used in two ways: on ordinary text files, and on specially marked-up text files with predefined headers and footers.

### Useful options

- `-b [a|t|n|pR]` Prepend numbers to all lines (`a`), nonblank lines (`t`), no lines (`n`), or only lines that contain regular expression `R`. (Default=`a`)
- `-v N` Begin numbering with integer `N`. (Default=`1`)
- `-i N` Increment the number by `N` for each line, so for example, you could use odd numbers only (`-i2`) or even numbers only (`-v2 -i2`). (Default=`1`)
- `-n [ln|rn|rz]` Format numbers as left-justified (`ln`), right-justified (`rn`), or right-justified with leading zeroes (`rz`). (Default=`ln`)
- `-w N` Force the width of the number to be `N` columns. (Default=`6`)
- `-s S` Insert string `S` between the line number and the text. (Default=`TAB`)

Additionally, `nl` has the wacky ability to divide text files into virtual pages, each with a header, body, and footer with different numbering schemes. For this to work, however, you must insert `nl`-specific delimiter strings into the file, such as `:\:\:\:` (start of header), `\:\:\:` (start of body), and `\:` (start of footer). Each must appear on a line by itself. Then you can use additional options (see the manpage) to affect line numbering in the headers and footers of your decorated file.
strings [options] [files]

Binary files, such as executable programs and object files, usually contain some readable text. The `strings` program extracts that text and displays it on standard output. You can discover version information, authors’ names, and other useful tidbits with `strings`.

```
$ strings /usr/bin/who
David MacKenzie
Copyright %s %d Free Software Foundation, Inc.
Report %s bugs to %s
...
```

Combine `strings` and `grep` to make your exploring more efficient. Here we look for email addresses:

```
$ strings /usr/bin/who | grep '@'
bug-coreutils@gnu.org
```

**Useful options**

- `-n length` Display only strings with length greater than `length` (the default is 4).

od [options] [files]

When you want to view a binary file, consider `od` (Octal Dump) for the job. It copies one or more files to standard output, displaying their data in ASCII, octal, decimal, hexadecimal, or floating point, in various sizes (byte, short, long). For example, this command:

```
$ od -w8 /usr/bin/who
0000000 042577 043114 000401 000001
0000010 000000 000000 000000 000000
0000020 000002 000003 000001 000000
...
```

displays the bytes in binary file `/usr/bin/who` in octal, eight bytes per line. The column on the left contains the file offset of each row, again in octal.
If your binary file also contains text, consider the \texttt{-tc} option, which displays character data. For example, binary executables like \texttt{who} contain the string “ELF” at the beginning:

\begin{verbatim}
$ od -tc -w8 /usr/bin/who | head -3
0000000 177 E L F 001 001 001 \0
0000010 \0 \0 \0 \0 \0 \0 \0
0000020 002 \0 003 \0 001 \0 \0
\end{verbatim}

**Useful options**

- \texttt{-N B} Display only the first \texttt{B} bytes of each file, specified in decimal, hexadecimal (by prepending 0x or 0X), 512-byte blocks (by appending \texttt{b}), kilobytes (by appending \texttt{k}), or megabytes (by appending \texttt{m}). (Default displays the entire file.)

- \texttt{-j B} Begin the output at byte \texttt{B}+1 of each file; acceptable formats are the same as for the \texttt{-N} option. (Default=0)

- \texttt{-w [ B ]} Display \texttt{B} bytes per line; acceptable formats are the same as in the \texttt{-N} option. Using \texttt{-w} by itself is equivalent to \texttt{-w32}. (Default=16)

- \texttt{-s [ B ]} Group each row of bytes into sequences of \texttt{B} bytes, separated by whitespace; acceptable formats are the same as in the \texttt{-N} option. Using \texttt{-s} by itself is equivalent to \texttt{-s3}. (Default=2)

- \texttt{-A (d|o|x|n)} Display file offsets in the leftmost column, in decimal (\texttt{d}), octal (\texttt{o}), hexadecimal (\texttt{h}), or not at all (\texttt{n}). (Default=\texttt{o})

- \texttt{-t (a|c)[z]} Display output in a character format, with nonalphanumeric characters printed as escape sequences (\texttt{c}) or by name (\texttt{a}). For \texttt{z}, see below.

- \texttt{-t (d|o|u|x)[SIZE[z]]} Display output in an integer format, including octal (\texttt{o}), signed decimal (\texttt{d}), unsigned decimal (\texttt{u}), hexadecimal (\texttt{x}). (For binary output, use \texttt{xxd} instead.) \texttt{SIZE} represents the number of bytes per integer; it can be a positive integer or any of the values \texttt{C}, \texttt{S}, \texttt{I}, or \texttt{L}, which stand for the size of a char, short, int, or long datatype, respectively. For \texttt{z}, see below.

- \texttt{-t f[SIZE[z]]} Display output in floating point. \texttt{SIZE} represents the number of bytes per integer; it can be a positive integer or any of the values \texttt{F}, \texttt{D}, or \texttt{L}, which stand for the size of a float, double, or long double datatype, respectively. For \texttt{z}, see below. If \texttt{-t} is omitted, the default is \texttt{-to2}. 

---

50 | Linux Pocket Guide
Appending \texttt{z} to the \texttt{-t} option prints a new column on the right-hand side of the output, displaying the printable characters on each line, much like the default output of \texttt{xxd}.

\begin{verbatim}
xxd

xxd [options] [files]

Similar to \texttt{od}, \texttt{xxd} produces a hexadecimal or binary dump of a file in several different formats. It can also do the reverse, converting from its hex dump format back into the original data. For example, here’s a hex dump of binary file /usr/bin/who:

\begin{verbatim}
$ xxd /usr/bin/who
0000000: 7f45 4c46 0101 0100 0000 ... 0000 .ELF.............
0000010: 0200 0300 0100 0000 a08c ... 0000 ..........4...
0000020: 6824 0000 0000 0000 3400 ... 2800 h$......4. ...(.
0000030: 1900 1800 0600 0000 3400 ... 0408 ........4...4...
...
\end{verbatim}

The left column indicates the file offset of the row, the next eight columns contain the data, and the final column displays the printable characters in the row, if any.

By default, \texttt{xxd} outputs three columns: file offsets, the data in hex, and the data as text (printable characters only).

**Useful options**

- \texttt{-l N} Display only the first \texttt{N} bytes. (Default displays the entire file.)
- \texttt{-s N} Skip the first \texttt{N} bytes of the file.
- \texttt{-s -N} Begin \texttt{N} bytes from the end of the file. (There is also a \texttt{+N} syntax for more advanced skipping through standard input; see the manpage.)
- \texttt{-c N} Display \texttt{N} bytes per row. (Default=16)
- \texttt{-g N} Group each row of bytes into sequences of \texttt{N} bytes, separated by whitespace, like od \texttt{--s} (Default=2)
- \texttt{-b} Display the output in binary instead of hexadecimal.
- \texttt{-u} Display the output in uppercase hexadecimal instead of lowercase.
- \texttt{-p} Display the output as a plain hexdump, 60 contiguous bytes per line.

\end{verbatim}
5 – Manipulating Files And Directories

At this point, we are ready for some real work! This chapter will introduce the following commands:

- `cp` – Copy files and directories
- `mv` – Move/rename files and directories
- `mkdir` – Create directories
- `rm` – Remove files and directories
- `ln` – Create hard and symbolic links

These five commands are among the most frequently used Linux commands. They are used for manipulating both files and directories.

Now, to be frank, some of the tasks performed by these commands are more easily done with a graphical file manager. With a file manager, we can drag and drop a file from one directory to another, cut and paste files, delete files, etc. So why use these old command line programs?

The answer is power and flexibility. While it is easy to perform simple file manipulations with a graphical file manager, complicated tasks can be easier with the command line programs. For example, how could we copy all the HTML files from one directory to another, but only copy files that do not exist in the destination directory or are newer than the versions in the destination directory? Pretty hard with a file manager. Pretty easy with the command line:

```
cp -u *.html destination
```

**Wildcards**

Before we begin using our commands, we need to talk about a shell feature that makes these commands so powerful. Since the shell uses filenames so much, it provides special characters to help you rapidly specify groups of filenames. These special characters are
called *wildcards*. Using wildcards (which is also known as *globbing*) allow you to select filenames based on patterns of characters. The table below lists the wildcards and what they select:

**Table 5-1: Wildcards**

<table>
<thead>
<tr>
<th>Wildcard</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>Matches any characters</td>
</tr>
<tr>
<td>?</td>
<td>Matches any single character</td>
</tr>
<tr>
<td><code>[characters]</code></td>
<td>Matches any character that is a member of the set <code>characters</code></td>
</tr>
<tr>
<td><code>![characters]</code></td>
<td>Matches any character that is not a member of the set <code>characters</code></td>
</tr>
<tr>
<td><code>[[:class:])</code></td>
<td>Matches any character that is a member of the specified <code>class</code></td>
</tr>
</tbody>
</table>

Table 5-2 lists the most commonly used character classes:

**Table 5-2: Commonly Used Character Classes**

<table>
<thead>
<tr>
<th>Character Class</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>[:alnum:]</code></td>
<td>Matches any alphanumeric character</td>
</tr>
<tr>
<td><code>[:alpha:]</code></td>
<td>Matches any alphabetic character</td>
</tr>
<tr>
<td><code>[:digit:]</code></td>
<td>Matches any numeral</td>
</tr>
<tr>
<td><code>[:lower:]</code></td>
<td>Matches any lowercase letter</td>
</tr>
<tr>
<td><code>[:upper:]</code></td>
<td>Matches any uppercase letter</td>
</tr>
</tbody>
</table>

Using wildcards makes it possible to construct very sophisticated selection criteria for filenames. Here are some examples of patterns and what they match:

**Table 5-3: Wildcard Examples**

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Matches</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>All files</td>
</tr>
<tr>
<td><code>g*</code></td>
<td>Any file beginning with “g”</td>
</tr>
<tr>
<td><code>b*.txt</code></td>
<td>Any file beginning with “b” followed by any characters and ending with “.txt”</td>
</tr>
</tbody>
</table>
### Wildcards

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Data??</code></td>
<td>Any file beginning with “Data” followed by exactly three characters</td>
</tr>
<tr>
<td><code>[abc]*</code></td>
<td>Any file beginning with either an “a”, a “b”, or a “c”</td>
</tr>
<tr>
<td><code>BACKUP.[0-9][0-9][0-9]</code></td>
<td>Any file beginning with “BACKUP.” followed by exactly three numerals</td>
</tr>
<tr>
<td><code>[:upper:]*</code></td>
<td>Any file beginning with an uppercase letter</td>
</tr>
<tr>
<td><code>![[:digit:]]*</code></td>
<td>Any file not beginning with a numeral</td>
</tr>
<tr>
<td><code>*[:lower:123]</code></td>
<td>Any file ending with a lowercase letter or the numerals “1”, “2”, or “3”</td>
</tr>
</tbody>
</table>

Wildcards can be used with any command that accepts filenames as arguments, but we’ll talk more about that in Chapter 8.

### Character Ranges

If you are coming from another Unix-like environment or have been reading some other books on this subject, you may have encountered the `[A-Z]` or the `[a-z]` character range notations. These are traditional Unix notations and worked in older versions of Linux as well. They can still work, but you have to be very careful with them because they will not produce the expected results unless properly configured. For now, you should avoid using them and use character classes instead.

### Wildcards Work In The GUI Too

Wildcards are especially valuable not only because they are used so frequently on the command line, but are also supported by some graphical file managers.

- In **Nautilus** (the file manager for GNOME), you can select files using the Edit/Select Pattern menu item. Just enter a file selection pattern with wildcards and the files in the currently viewed directory will be highlighted for selection.

- In **Dolphin** and **Konqueror** (the file managers for KDE), you can enter wildcards directly on the location bar. For example, if you want to see all the files starting with a lowercase “u” in the `/usr/bin` directory, type “/usr/bin/u*” into the location bar and it will display the result.
Many ideas originally found in the command line interface make their way into the graphical interface, too. It is one of the many things that make the Linux desktop so powerful.

**mkdir – Create Directories**

The *mkdir* command is used to create directories. It works like this:

```
mkdir directory...
```

A note on notation: When three periods follow an argument in the description of a command (as above), it means that the argument can be repeated, thus:

```
mkdir dir1
```

would create a single directory named “dir1”, while

```
mkdir dir1 dir2 dir3
```

would create three directories named “dir1”, “dir2”, and “dir3”.

**cp – Copy Files And Directories**

The *cp* command copies files or directories. It can be used two different ways:

```
cp item1 item2
```

to copy the single file or directory “item1” to file or directory “item2” and:

```
cp item... directory
```

to copy multiple items (either files or directories) into a directory.
Useful Options And Examples

Here are some of the commonly used options (the short option and the equivalent long option) for `cp`:

**Table 5-4: cp Options**

<table>
<thead>
<tr>
<th>Option</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-a, --archive</code></td>
<td>Copy the files and directories and all of their attributes, including ownerships and permissions. Normally, copies take on the default attributes of the user performing the copy.</td>
</tr>
<tr>
<td><code>-i, --interactive</code></td>
<td>Before overwriting an existing file, prompt the user for confirmation. <strong>If this option is not specified, cp will silently overwrite files.</strong></td>
</tr>
<tr>
<td><code>-r, --recursive</code></td>
<td>Recursively copy directories and their contents. This option (or the <code>-a</code> option) is required when copying directories.</td>
</tr>
<tr>
<td><code>-u, --update</code></td>
<td>When copying files from one directory to another, only copy files that either don't exist, or are newer than the existing corresponding files, in the destination directory.</td>
</tr>
<tr>
<td><code>-v, --verbose</code></td>
<td>Display informative messages as the copy is performed.</td>
</tr>
</tbody>
</table>

**Table 5-5: cp Examples**

<table>
<thead>
<tr>
<th>Command</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cp file1 file2</code></td>
<td>Copy file1 to file2. <strong>If file2 exists, it is overwritten with the contents of file1.</strong> If file2 does not exist, it is created.</td>
</tr>
<tr>
<td><code>cp -i file1 file2</code></td>
<td>Same as above, except that if file2 exists, the user is prompted before it is overwritten.</td>
</tr>
<tr>
<td><code>cp file1 file2 dir1</code></td>
<td>Copy file1 and file2 into directory dir1. dir1 must already exist.</td>
</tr>
<tr>
<td><code>cp dir1/* dir2</code></td>
<td>Using a wildcard, all the files in dir1 are copied into dir2. dir2 must already exist.</td>
</tr>
</tbody>
</table>
Manipulating Files And Directories

**cp**

```
cp -r dir1 dir2
```

Copy the contents of directory `dir1` to directory `dir2`. If directory `dir2` does not exist, it is created and, after the copy, will contain the same contents as directory `dir1`. If directory `dir2` does exist, then directory `dir1` (and its contents) will be copied into `dir2`.

**mv**

Move And Rename Files

The `mv` command performs both file moving and file renaming, depending on how it is used. In either case, the original filename no longer exists after the operation. `mv` is used in much the same way as `cp`:

```
mv item1 item2
```

to move or rename file or directory “item1” to “item2” or:

```
mv item... directory
```

to move one or more items from one directory to another.

Useful Options And Examples

`mv` shares many of the same options as `cp`:

<table>
<thead>
<tr>
<th>Option</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-i, --interactive</code></td>
<td>Before overwriting an existing file, prompt the user for confirmation. <strong>If this option is not specified, mv will silently overwrite files.</strong></td>
</tr>
<tr>
<td><code>-u, --update</code></td>
<td>When moving files from one directory to another, only move files that either don't exist, or are newer than the existing corresponding files in the destination directory.</td>
</tr>
<tr>
<td><code>-v, --verbose</code></td>
<td>Display informative messages as the move is</td>
</tr>
</tbody>
</table>
mv – Move And Rename Files

performed.

Table 5-7: mv Examples

<table>
<thead>
<tr>
<th>Command</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>mv file1 file2</td>
<td>Move file1 to file2. <strong>If file2 exists, it is overwritten with the contents of file1.</strong> If file2 does not exist, it is created. <strong>In either case, file1 ceases to exist.</strong></td>
</tr>
<tr>
<td>mv -i file1 file2</td>
<td>Same as above, except that if file2 exists, the user is prompted before it is overwritten.</td>
</tr>
<tr>
<td>mv file1 file2 dir1</td>
<td>Move file1 and file2 into directory dir1. dir1 must already exist.</td>
</tr>
<tr>
<td>mv dir1 dir2</td>
<td>If directory dir2 does not exist, create directory dir2 and move the contents of directory dir1 into dir2 and delete directory dir1. If directory dir2 does exist, move directory dir1 (and its contents) into directory dir2.</td>
</tr>
</tbody>
</table>

rm – Remove Files And Directories

The `rm` command is used to remove (delete) files and directories:

```
rm item...
```

where “item” is one or more files or directories.

Useful Options And Examples

Here are some of the common options for `rm`:

Table 5-8: rm Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>-i, --interactive</td>
<td>Before deleting an existing file, prompt the user for confirmation. <strong>If this option is not specified, rm will silently delete files.</strong></td>
</tr>
</tbody>
</table>
-r, --recursive
Recursively delete directories. This means that if a directory being deleted has subdirectories, delete them too. To delete a directory, this option must be specified.

-f, --force
Ignore nonexistent files and do not prompt. This overrides the --interactive option.

-v, --verbose
Display informative messages as the deletion is performed.

Table 5-9: rm Examples

<table>
<thead>
<tr>
<th>Command</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>rm file1</td>
<td>Delete file1 silently.</td>
</tr>
<tr>
<td>rm -i file1</td>
<td>Same as above, except that the user is prompted for confirmation before the deletion is performed.</td>
</tr>
<tr>
<td>rm -r file1 dir1</td>
<td>Delete file1 and dir1 and its contents.</td>
</tr>
<tr>
<td>rm -rf file1 dir1</td>
<td>Same as above, except that if either file1 or dir1 do not exist, rm will continue silently.</td>
</tr>
</tbody>
</table>

Be Careful With rm!

Unix-like operating systems such as Linux do not have an undelete command. Once you delete something with rm, it's gone. Linux assumes you're smart and you know what you're doing.

Be particularly careful with wildcards. Consider this classic example. Let's say you want to delete just the HTML files in a directory. To do this, you type:

```
rm *.html
```

which is correct, but if you accidentally place a space between the “*” and the “.html” like so:

```
rm * .html
```

the rm command will delete all the files in the directory and then complain that there is no file called “.html”.

Be Careful With rm!
Here is a useful tip. Whenever you use wildcards with `rm` (besides carefully checking your typing!), test the wildcard first with `ls`. This will let you see the files that will be deleted. Then press the up arrow key to recall the command and replace the `ls` with `rm`.

**ln – Create Links**

The `ln` command is used to create either hard or symbolic links. It is used in one of two ways:

```
ln file link
```

to create a hard link, and:

```
ln -s item link
```

to create a symbolic link where “item” is either a file or a directory.

**Hard Links**

Hard links are the original Unix way of creating links, compared to symbolic links, which are more modern. By default, every file has a single hard link that gives the file its name. When we create a hard link, we create an additional directory entry for a file. Hard links have two important limitations:

1. A hard link cannot reference a file outside its own file system. This means a link may not reference a file that is not on the same disk partition as the link itself.
2. A hard link may not reference a directory.

A hard link is indistinguishable from the file itself. Unlike a symbolic link, when you list a directory containing a hard link you will see no special indication of the link. When a hard link is deleted, the link is removed but the contents of the file itself continue to exist (that is, its space is not deallocated) until all links to the file are deleted.

It is important to be aware of hard links because you might encounter them from time to time, but modern practice prefers symbolic links, which we will cover next.
Symbolic Links

Symbolic links were created to overcome the limitations of hard links. Symbolic links work by creating a special type of file that contains a text pointer to the referenced file or directory. In this regard, they operate in much the same way as a Windows shortcut though of course, they predate the Windows feature by many years ;-)  

A file pointed to by a symbolic link, and the symbolic link itself are largely indistinguishable from one another. For example, if you write some something to the symbolic link, the referenced file is also written to. However when you delete a symbolic link, only the link is deleted, not the file itself. If the file is deleted before the symbolic link, the link will continue to exist, but will point to nothing. In this case, the link is said to be broken. In many implementations, the `ls` command will display broken links in a distinguishing color, such as red, to reveal their presence.

The concept of links can seem very confusing, but hang in there. We're going to try all this stuff and it will, hopefully, become clear.

Let's Build A Playground

Since we are going to do some real file manipulation, let's build a safe place to “play” with our file manipulation commands. First we need a directory to work in. We'll create one in our home directory and call it “playground.”

Creating Directories

The `mkdir` command is used to create a directory. To create our playground directory we will first make sure we are in our home directory and will then create the new directory:

```
[me@linuxbox ~]$ cd
[me@linuxbox ~]$ mkdir playground
```

To make our playground a little more interesting, let's create a couple of directories inside it called “dir1” and “dir2”. To do this, we will change our current working directory to `playground` and execute another `mkdir`:

```
[me@linuxbox ~]$ cd playground
[me@linuxbox playground]$ mkdir dir1 dir2
```

Notice that the `mkdir` command will accept multiple arguments allowing us to create
both directories with a single command.

**Copying Files**

Next, let's get some data into our playground. We'll do this by copying a file. Using the `cp` command, we'll copy the `passwd` file from the `/etc` directory to the current working directory:

```
[me@linuxbox playground]$ cp /etc/passwd .
```

Notice how we used the shorthand for the current working directory, the single trailing period. So now if we perform an `ls`, we will see our file:

```
[me@linuxbox playground]$ ls -l
total 12
  drwxr-xr-x 2 me me 4096 2008-01-10 16:40 dir1
  drwxr-xr-x 2 me me 4096 2008-01-10 16:40 dir2
  -rw-r--r-- 1 me me 1650 2008-01-10 16:07 passwd
```

Now, just for fun, let's repeat the copy using the “-v” option (verbose) to see what it does:

```
[me@linuxbox playground]$ cp -v /etc/passwd .
`/etc/passwd' -> `./passwd'
```

The `cp` command performed the copy again, but this time displayed a concise message indicating what operation it was performing. Notice that `cp` overwrote the first copy without any warning. Again this is a case of `cp` assuming that you know what you're doing. To get a warning, we'll include the “-i” (interactive) option:

```
[me@linuxbox playground]$ cp -i /etc/passwd .
cp: overwrite `./passwd'?
```

Responding to the prompt by entering a “y” will cause the file to be overwritten, any other character (for example, “n”) will cause `cp` to leave the file alone.
Moving And Renaming Files

Now, the name “passwd” doesn't seem very playful and this is a playground, so let's change it to something else:

```bash
[me@linuxbox playground]$ mv passwd fun
```

Let's pass the fun around a little by moving our renamed file to each of the directories and back again:

```bash
[me@linuxbox playground]$ mv fun dir1
to move it first to directory dir1, then:
[me@linuxbox playground]$ mv dir1/fun dir2
to move it from dir1 to dir2, then:
[me@linuxbox playground]$ mv dir2/fun .
to finally bringing it back to the current working directory. Next, let's see the effect of `mv` on directories. First we will move our data file into `dir1` again:

```bash
[me@linuxbox playground]$ mv fun dir1
```

then move `dir1` into `dir2` and confirm it with `ls`:

```
[me@linuxbox playground]$ mv dir1 dir2
[me@linuxbox playground]$ ls -l dir2
total 4
drwxrwxr-x 2 me  me   4096 2008-01-11 06:06 dir1
[me@linuxbox playground]$ ls -l dir2/dir1
total 4
-rw-r--r-- 1 me  me   1650 2008-01-10 16:33 fun
```
Note that since dir2 already existed, mv moved dir1 into dir2. If dir2 had not existed, mv would have renamed dir1 to dir2. Lastly, let's put everything back:

```bash
[me@linuxbox playground]$ mv dir2/dir1 .
[me@linuxbox playground]$ mv dir1/fun .
```

Creating Hard Links

Now we'll try some links. First the hard links. We’ll create some links to our data file like so:

```bash
[me@linuxbox playground]$ ln fun fun-hard
[me@linuxbox playground]$ ln fun dir1/fun-hard
[me@linuxbox playground]$ ln fun dir2/fun-hard
```

So now we have four instances of the file “fun”. Let's take a look our playground directory:

```bash
[me@linuxbox playground]$ ls -l
total 16
drwxrwxr-x 2 me  me  4096 2008-01-14 16:17 dir1
drwxrwxr-x 2 me  me  4096 2008-01-14 16:17 dir2
-rw-r--r-- 4 me  me  1650 2008-01-10 16:33 fun
-rw-r--r-- 4 me  me  1650 2008-01-10 16:33 fun-hard
```

One thing you notice is that the second field in the listing for fun and fun-hard both contain a “4” which is the number of hard links that now exist for the file. You'll remember that a file will always have at least one because the file's name is created by a link. So, how do we know that fun and fun-hard are, in fact, the same file? In this case, ls is not very helpful. While we can see that fun and fun-hard are both the same size (field 5), our listing provides no way to be sure. To solve this problem, we're going to have to dig a little deeper.

When thinking about hard links, it is helpful to imagine that files are made up of two parts: the data part containing the file's contents and the name part which holds the file's name. When we create hard links, we are actually creating additional name parts that all refer to the same data part. The system assigns a chain of disk blocks to what is called an inode, which is then associated with the name part. Each hard link therefore refers to a specific inode containing the file's contents.
The `ls` command has a way to reveal this information. It is invoked with the “-i” option:

```
[me@linuxbox playground]$ ls -li
total 16
12353539 drwxrwxr-x 2 me   me   4096 2008-01-14 16:17 dir1
12353540 drwxrwxr-x 2 me   me   4096 2008-01-14 16:17 dir2
12353538 -rw-r--r-- 4 me   me   1650 2008-01-10 16:33 fun
12353538 -rw-r--r-- 4 me   me   1650 2008-01-10 16:33 fun-hard
```

In this version of the listing, the first field is the inode number and, as we can see, both `fun` and `fun-hard` share the same inode number, which confirms they are the same file.

### Creating Symbolic Links

Symbolic links were created to overcome the two disadvantages of hard links: hard links cannot span physical devices and hard links cannot reference directories, only files. Symbolic links are a special type of file that contains a text pointer to the target file or directory.

Creating symbolic links is similar to creating hard links:

```
[me@linuxbox playground]$ ln -s fun fun-sym
[me@linuxbox playground]$ ln -s ../fun dir1/fun-sym
[me@linuxbox playground]$ ln -s ../fun dir2/fun-sym
```

The first example is pretty straightforward, we simply add the “-s” option to create a symbolic link rather than a hard link. But what about the next two? Remember, when we create a symbolic link, we are creating a text description of where the target file is relative to the symbolic link. It's easier to see if we look at the `ls` output:

```
[me@linuxbox playground]$ ls -l dir1
total 4
-rw-r--r-- 4 me   me 1650 2008-01-10 16:33 fun-hard
-lrwxrwxrwx 1 me   me 6 2008-01-15 15:17 fun-sym -> ../fun
```

The listing for `fun-sym` in `dir1` shows that it is a symbolic link by the leading “l” in the first field and that it points to “../fun”, which is correct. Relative to the location of `fun-sym`, `fun` is in the directory above it. Notice too, that the length of the symbolic link file is 6, the number of characters in the string “../fun” rather than the length of the
file to which it is pointing.

When creating symbolic links, you can either use absolute pathnames:

```
ln -s /home/me/playground/fun dir1/fun-sym
```

or relative pathnames, as we did in our earlier example. Using relative pathnames is more desirable because it allows a directory containing symbolic links to be renamed and/or moved without breaking the links.

In addition to regular files, symbolic links can also reference directories:

```
[me@linuxbox playground]$ ln -s dir1 dir1-sym
[me@linuxbox playground]$ ls -l
```

```
total 16
drwxrwxr-x 2 me   me   4096 2008-01-15 15:17 dir1
lrwxrwxrwx 1 me   me      4 2008-01-16 14:45 dir1-sym -> dir1
drwxrwxr-x 2 me   me   4096 2008-01-15 15:17 dir2
-rw-r--r-- 4 me   me   1650 2008-01-10 16:33 fun
-rw-r--r-- 4 me   me   1650 2008-01-10 16:33 fun-hard
lrwxrwxrwx 1 me   me      3 2008-01-15 15:15 fun-sym -> fun
```

Removing Files And Directories

As we covered earlier, the `rm` command is used to delete files and directories. We are going to use it to clean up our playground a little bit. First, let's delete one of our hard links:

```
[me@linuxbox playground]$ rm fun-hard
[me@linuxbox playground]$ ls -l
```

```
total 12
drwxrwxr-x 2 me   me   4096 2008-01-15 15:17 dir1
lrwxrwxrwx 1 me   me   4 2008-01-16 14:45 dir1-sym -> dir1
drwxrwxr-x 2 me   me   4096 2008-01-15 15:17 dir2
-rw-r--r-- 3 me   me   1650 2008-01-10 16:33 fun
lrwxrwxrwx 1 me   me   3 2008-01-15 15:15 fun-sym -> fun
```

That worked as expected. The file `fun-hard` is gone and the link count shown for `fun` is reduced from four to three, as indicated in the second field of the directory listing.

Next, we'll delete the file `fun`, and just for enjoyment, we'll include the “-i” option to show what that does:
Enter “y” at the prompt and the file is deleted. But let's look at the output of `ls` now. Noticed what happened to `fun-sym`? Since it's a symbolic link pointing to a non-existent file, the link is *broken*:

```bash
[me@linuxbox playground]$ ls -l
```

```
total 8
drwxrwxr-x 2 me   me   4096 2008-01-15 15:17 dir1
lrwxrwxrwx 1 me   me      4 2008-01-16 14:45 dir1-sym -> dir1
drwxrwxr-x 2 me   me   4096 2008-01-15 15:17 dir2
lrwxrwxrwx 1 me   me      3 2008-01-15 15:15 fun-sym -> fun
```

Most Linux distributions configure `ls` to display broken links. On a Fedora box, broken links are displayed in blinking red text! The presence of a broken link is not, in and of itself dangerous but it is rather messy. If we try to use a broken link we will see this:

```bash
[me@linuxbox playground]$ less fun-sym
fun-sym: No such file or directory
```

Let's clean up a little. We'll delete the symbolic links:

```bash
[me@linuxbox playground]$ rm fun-sym dir1-sym
[me@linuxbox playground]$ ls -l
```

```
total 8
drwxrwxr-x 2 me   me   4096 2008-01-15 15:17 dir1
drwxrwxr-x 2 me   me   4096 2008-01-15 15:17 dir2
```

One thing to remember about symbolic links is that most file operations are carried out on the link's target, not the link itself. `rm` is an exception. When you delete a link, it is the link that is deleted, not the target.

Finally, we will remove our playground. To do this, we will return to our home directory and use `rm` with the recursive option (-r) to delete playground and all of its contents, including its subdirectories:

```bash
[me@linuxbox playground]$ cd
```
Let's Build A Playground

Creating Symlinks With The GUI

The file managers in both GNOME and KDE provide an easy and automatic method of creating symbolic links. With GNOME, holding the Ctrl+Shift keys while dragging a file will create a link rather than copying (or moving) the file. In KDE, a small menu appears whenever a file is dropped, offering a choice of copying, moving, or linking the file.

Summing Up

We've covered a lot of ground here and it will take a while to fully sink in. Perform the playground exercise over and over until it makes sense. It is important to get a good understanding of basic file manipulation commands and wildcards. Feel free to expand on the playground exercise by adding more files and directories, using wildcards to specify files for various operations. The concept of links is a little confusing at first, but take the time to learn how they work. They can be a real lifesaver.

[me@linuxbox ~]$ rm -r playground
6 – Working With Commands

Up to this point, we have seen a series of mysterious commands, each with its own mysterious options and arguments. In this chapter, we will attempt to remove some of that mystery and even create some of our own commands. The commands introduced in this chapter are:

- **type** – Indicate how a command name is interpreted
- **which** – Display which executable program will be executed
- **man** – Display a command’s manual page
- **apropos** – Display a list of appropriate commands
- **info** – Display a command’s info entry
- **whatis** – Display a very brief description of a command
- **alias** – Create an alias for a command

What Exactly Are Commands?

A command can be one of four different things:

1. **An executable program** like all those files we saw in `/usr/bin`. Within this category, programs can be compiled binaries such as programs written in C and C++, or programs written in scripting languages such as the shell, perl, python, ruby, etc.

2. **A command built into the shell itself**. bash supports a number of commands internally called shell builtns. The `cd` command, for example, is a shell builtin.

3. **A shell function**. These are miniature shell scripts incorporated into the environment. We will cover configuring the environment and writing shell functions in later chapters, but for now, just be aware that they exist.

4. **An alias**. Commands that we can define ourselves, built from other commands.
Identifying Commands

It is often useful to know exactly which of the four kinds of commands is being used and Linux provides a couple of ways to find out.

**type – Display A Command’s Type**

The *type* command is a shell builtin that displays the kind of command the shell will execute, given a particular command name. It works like this:

```
$ type type
type is a shell builtin
$ type ls
ls is aliased to `ls --color=tty'
$ type cp
cmp is /bin/cp
```

Here we see the results for three different commands. Notice that the one for `ls` (taken from a Fedora system) and how the `ls` command is actually an alias for the `ls` command with the “-- color=tty” option added. Now we know why the output from `ls` is displayed in color!

**which – Display An Executable's Location**

Sometimes there is more than one version of an executable program installed on a system. While this is not very common on desktop systems, it’s not unusual on large servers. To determine the exact location of a given executable, the *which* command is used:

```
$ which ls
/bin/ls
```

*which* only works for executable programs, not builtins nor aliases that are substitutes for actual executable programs. When we try to use *which* on a shell builtin, for
example, cd, we either get no response or an error message:

```
[me@linuxbox ~]$ which cd
/usr/bin/which: no cd in
(/opt/jre1.6.0_03/bin:/usr/lib/qt-3.3/bin:/usr/kerberos/bin:/opt/jre1.6.0_03/bin:/usr/lib/ccache:/usr/local/bin:/usr/bin:/bin:/home/me/bin)
```

which is a fancy way of saying “command not found.”

**Getting A Command's Documentation**

With this knowledge of what a command is, we can now search for the documentation available for each kind of command.

**help – Get Help For Shell Builtins**

bash has a built-in help facility available for each of the shell builtins. To use it, type “help” followed by the name of the shell builtin. For example:

```
[me@linuxbox ~]$ help cd
cd: cd [-L|-P] [dir]
Change the current directory to DIR. The variable $HOME is the
default DIR. The variable CDPATH defines the search path for the
directory containing DIR. Alternative directory names in CDPATH are
separated by a colon (:) A null directory name is the same as the
current directory, i.e. `.'. If DIR begins with a slash (/), then
CDPATH is not used. If the directory is not found, and the shell
option `cdable_vars' is set, then try the word as a variable name.
If that variable has a value, then cd to the value of that variable.
The -P option says to use the physical directory structure instead of
following symbolic links; the -L option forces symbolic links to be
followed.
```

**A note on notation:** When square brackets appear in the description of a command's syntax, they indicate optional items. A vertical bar character indicates mutually exclusive items. In the case of the cd command above:

```
cd [-L|-P] [dir]
```

This notation says that the command cd may be followed optionally by either a “-L” or a “-P” and further, optionally followed by the argument “dir”.

While the output of help for the cd commands is concise and accurate, it is by no
Getting A Command's Documentation

means tutorial and as we can see, it also seems to mention a lot of things we haven't talked about yet! Don't worry. We'll get there.

--help – Display Usage Information

Many executable programs support a “--help” option that displays a description of the command's supported syntax and options. For example:

```bash
[me@linuxbox ~]$ mkdir --help
Usage: mkdir [OPTION] DIRECTORY...
Create the DIRECTORY(ies), if they do not already exist.

-Z, --context=CONTEXT (SELinux) set security context to CONTEXT
Mandatory arguments to long options are mandatory for short options too.
-m, --mode=MODE   set file mode (as in chmod), not a=rwx – umask
-p, --parents     no error if existing, make parent directories as needed
-v, --verbose     print a message for each created directory
--help        display this help and exit
--version     output version information and exit

Report bugs to <bug-coreutils@gnu.org>.
```

Some programs don't support the “--help” option, but try it anyway. Often it results in an error message that will reveal the same usage information.

man – Display A Program's Manual Page

Most executable programs intended for command line use provide a formal piece of documentation called a manual or man page. A special paging program called man is used to view them. It is used like this:

```
man program
```

where “program” is the name of the command to view.

Man pages vary somewhat in format but generally contain a title, a synopsis of the command's syntax, a description of the command's purpose, and a listing and description of each of the command's options. Man pages, however, do not usually include examples, and are intended as a reference, not a tutorial. As an example, let's try viewing
6 – Working With Commands

the man page for the `ls` command:

```
[me@linuxbox ~]$ man ls
```

On most Linux systems, `man` uses `less` to display the manual page, so all of the familiar `less` commands work while displaying the page.

The “manual” that `man` displays is broken into sections and not only covers user commands but also system administration commands, programming interfaces, file formats and more. The table below describes the layout of the manual:

*Table 6-1: Man Page Organization*

<table>
<thead>
<tr>
<th>Section</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>User commands</td>
</tr>
<tr>
<td>2</td>
<td>Programming interfaces kernel system calls</td>
</tr>
<tr>
<td>3</td>
<td>Programming interfaces to the C library</td>
</tr>
<tr>
<td>4</td>
<td>Special files such as device nodes and drivers</td>
</tr>
<tr>
<td>5</td>
<td>File formats</td>
</tr>
<tr>
<td>6</td>
<td>Games and amusements such as screen savers</td>
</tr>
<tr>
<td>7</td>
<td>Miscellaneous</td>
</tr>
<tr>
<td>8</td>
<td>System administration commands</td>
</tr>
</tbody>
</table>

Sometimes we need to look in a specific section of the manual to find what we are looking for. This is particularly true if we are looking for a file format that is also the name of a command. Without specifying a section number, we will always get the first instance of a match, probably in section 1. To specify a section number, we use `man` like this:

```
man section search_term
```

For example:

```
[me@linuxbox ~]$ man 5 passwd
```
Getting A Command's Documentation

This will display the man page describing the file format of the `/etc/passwd` file.

**apropos – Display Appropriate Commands**

It is also possible to search the list of man pages for possible matches based on a search term. It’s very crude but sometimes helpful. Here is an example of a search for man pages using the search term “floppy”:

```bash
[me@linuxbox ~]$ apropos floppy
create_floppy_devices (8) - udev callout to create all possible floppy device based on the CMOS type
fdformat (8) - Low-level formats a floppy disk
floppy (8) - format floppy disks
gfloppy (1) - a simple floppy formatter for the GNOME
mbadblocks (1) - tests a floppy disk, and marks the bad blocks in the FAT
mformat (1) - add an MSDOS filesystem to a low-level formatted floppy disk
```

The first field in each line of output is the name of the man page, the second field shows the section. Note that the `man` command with the “-k” option performs the exact same function as `apropos`.

**whatis – Display A Very Brief Description Of A Command**

The `whatis` program displays the name and a one line description of a man page matching a specified keyword:

```bash
[me@linuxbox ~]$ whatis ls
ls (1) - list directory contents
```

**The Most Brutal Man Page Of Them All**

As we have seen, the manual pages supplied with Linux and other Unix-like systems are intended as reference documentation and not as tutorials. Many man pages are hard to read, but I think that the grand prize for difficulty has got to go to the man page for `bash`. As I was doing my research for this book, I gave it careful review to ensure that I was covering most of its topics. When printed, it's
over eighty pages long and extremely dense, and its structure makes absolutely no sense to a new user.

On the other hand, it is very accurate and concise, as well as being extremely complete. So check it out if you dare and look forward to the day when you can read it and it all makes sense.

info – Display A Program's Info Entry

The GNU Project provides an alternative to man pages for their programs, called “info.” Info pages are displayed with a reader program named, appropriately enough, info. Info pages are hyperlinked much like web pages. Here is a sample:

```
File: coreutils.info,  Node: ls invocation,  Next: dir invocation,  Up: Directory listing

10.1 `ls': List directory contents
===============================

The `ls' program lists information about files (of any type, including directories). Options and file arguments can be intermixed arbitrarily, as usual.

For non-option command-line arguments that are directories, by default `ls' lists the contents of directories, not recursively, and omitting files with names beginning with `.'. For other non-option arguments, by default `ls' lists just the filename. If no non-option argument is specified, `ls' operates on the current directory, acting as if it had been invoked with a single argument of `.'.

By default, the output is sorted alphabetically, according to the
--zz-Info: (coreutils.info.gz)ls invocation, 63 lines --Top---------
```

The info program reads info files, which are tree structured into individual nodes, each containing a single topic. Info files contain hyperlinks that can move you from node to node. A hyperlink can be identified by its leading asterisk, and is activated by placing the cursor upon it and pressing the enter key.
Getting A Command's Documentation

To invoke `info`, type “info” followed optionally by the name of a program. Below is a table of commands used to control the reader while displaying an info page:

Table 6-2: `info` Commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>?</td>
<td>Display command help</td>
</tr>
<tr>
<td>PgUp or Backspace</td>
<td>Display previous page</td>
</tr>
<tr>
<td>PgDn or Space</td>
<td>Display next page</td>
</tr>
<tr>
<td>n</td>
<td>Next - Display the next node</td>
</tr>
<tr>
<td>p</td>
<td>Previous - Display the previous node</td>
</tr>
<tr>
<td>u</td>
<td>Up - Display the parent node of the currently displayed node, usually a menu.</td>
</tr>
<tr>
<td>Enter</td>
<td>Follow the hyperlink at the cursor location</td>
</tr>
<tr>
<td>q</td>
<td>Quit</td>
</tr>
</tbody>
</table>

Most of the command line programs we have discussed so far are part of the GNU Project's “coreutils” package, so typing:

```
[me@linuxbox ~]$ info coreutils
```

will display a menu page with hyperlinks to each program contained in the coreutils package.

**README And Other Program Documentation Files**

Many software packages installed on your system have documentation files residing in the `/usr/share/doc` directory. Most of these are stored in plain text format and can be viewed with `less`. Some of the files are in HTML format and can be viewed with a web browser. We may encounter some files ending with a “.gz” extension. This indicates that they have been compressed with the `gzip` compression program. The gzip package includes a special version of `less` called `zless` that will display the contents of gzip-compressed text files.
Creating Your Own Commands With alias

Now for our very first experience with programming! We will create a command of our own using the alias command. But before we start, we need to reveal a small command line trick. It's possible to put more than one command on a line by separating each command with a semicolon character. It works like this:

```
command1; command2; command3...
```

Here's the example we will use:

```
[me@linuxbox ~]$ cd /usr; ls; cd -
bin  games    kerberos  lib64  local  share  tmp
etc include  lib  libexec  sbin  src
/home/me
[me@linuxbox ~]$ 
```

As we can see, we have combined three commands on one line. First we change directory to /usr then list the directory and finally return to the original directory (by using 'cd -') so we end up where we started. Now let's turn this sequence into a new command using alias. The first thing we have to do is dream up a name for our new command. Let's try “test”. Before we do that, it would be a good idea to find out if the name “test” is already being used. To find out, we can use the type command again:

```
[me@linuxbox ~]$ type test

test is a shell builtin
```

Oops! The name “test” is already taken. Let's try “foo”:

```
[me@linuxbox ~]$ type foo

bash: type: foo: not found
```

Great! “foo” is not taken. So let's create our alias:

```
[me@linuxbox ~]$ alias foo='cd /usr; ls; cd -'
```
Creating Your Own Commands With alias

Notice the structure of this command:

```
alias name='string'
```

After the command “alias” we give alias a name followed immediately (no whitespace allowed) by an equals sign, followed immediately by a quoted string containing the meaning to be assigned to the name. After we define our alias, it can be used anywhere the shell would expect a command. Let’s try it:

```
[me@linuxbox ~]$ foo
bin  games    kerberos  lib64    local  share  tmp
etc  include  lib       libexec  sbin   src
/home/me
[me@linuxbox ~]$ 
```

We can also use the `type` command again to see our alias:

```
[me@linuxbox ~]$ type foo
foo is aliased to `cd /usr; ls ; cd -`
```

To remove an alias, the `unalias` command is used, like so:

```
[me@linuxbox ~]$ unalias foo
[me@linuxbox ~]$ type foo
bash: type: foo: not found
```

While we purposefully avoided naming our alias with an existing command name, it is not uncommon to do so. This is often done to apply a commonly desired option to each invocation of a common command. For instance, we saw earlier how the `ls` command is often aliased to add color support:

```
[me@linuxbox ~]$ type ls
ls is aliased to `ls --color=ttys'
```

To see all the aliases defined in the environment, use the `alias` command without arguments. Here are some of the aliases defined by default on a Fedora system. Try and
figure out what they all do:

```
[me@linuxbox ~]$ alias
alias l.='ls -d .* --color=tty'
alias ll='ls -l --color=tty'
alias ls='ls --color=tty'
```

There is one tiny problem with defining aliases on the command line. They vanish when your shell session ends. In a later chapter, we will see how to add our own aliases to the files that establish the environment each time we log on, but for now, enjoy the fact that we have taken our first, albeit tiny, step into the world of shell programming!

**Revisiting Old Friends**

Now that we have learned how to find the documentation for commands, go and look up the documentation for all the commands we have encountered so far. Study what additional options are available and try them out!

**Further Reading**

There are many online sources of documentation for Linux and the command line. Here are some of the best:

- The *Bash Reference Manual* is a reference guide to the *bash* shell. It’s still a reference work but contains examples and is easier to read than the *bash* man page.

- The *Bash FAQ* contains answers to frequently asked questions regarding *bash*. This list is aimed at intermediate to advanced users, but contains a lot of good information.
  [http://mywiki.wooledge.org/BashFAQ](http://mywiki.wooledge.org/BashFAQ)

- The GNU Project provides extensive documentation for its programs, which form the core of the Linux command line experience. You can see a complete list here:

- Wikipedia has an interesting article on man pages:
7 – Redirection

In this lesson we are going to unleash what may be the coolest feature of the command line. It's called **I/O redirection**. The “I/O” stands for *input/output* and with this facility you can redirect the input and output of commands to and from files, as well as connect multiple commands together into powerful command **pipelines**. To show off this facility, we will introduce the following commands:

- **cat** - Concatenate files
- **sort** - Sort lines of text
- **uniq** - Report or omit repeated lines
- **grep** - Print lines matching a pattern
- **wc** - Print newline, word, and byte counts for each file
- **head** - Output the first part of a file
- **tail** - Output the last part of a file
- **tee** - Read from standard input and write to standard output and files

**Standard Input, Output, And Error**

Many of the programs that we have used so far produce output of some kind. This output often consists of two types. First, we have the program's results; that is, the data the program is designed to produce, and second, we have status and error messages that tell us how the program is getting along. If we look at a command like `ls`, we can see that it displays its results and its error messages on the screen.

Keeping with the Unix theme of “everything is a file,” programs such as `ls` actually send their results to a special file called *standard output* (often expressed as `stdout`) and their status messages to another file called *standard error* (`stderr`). By default, both standard output and standard error are linked to the screen and not saved into a disk file.

In addition, many programs take input from a facility called *standard input* (`stdin`) which is, by default, attached to the keyboard.
I/O redirection allows us to change where output goes and where input comes from. Normally, output goes to the screen and input comes from the keyboard, but with I/O redirection, we can change that.

**Redirecting Standard Output**

I/O redirection allows us to redefine where standard output goes. To redirect standard output to another file besides the screen, we use the “>” redirection operator followed by the name of the file. Why would we want to do this? It's often useful to store the output of a command in a file. For example, we could tell the shell to send the output of the `ls` command to the file `ls-output.txt` instead of the screen:

```
[me@linuxbox ~]$ ls -l /usr/bin > ls-output.txt
```

Here, we created a long listing of the `/usr/bin` directory and sent the results to the file `ls-output.txt`. Let's examine the redirected output of the command:

```
[me@linuxbox ~]$ ls -l ls-output.txt
-rw-rw-r-- 1 me me 167878 2008-02-01 15:07 ls-output.txt
```

Good; a nice, large, text file. If we look at the file with `less`, we will see that the file `ls-output.txt` does indeed contain the results from our `ls` command:

```
[me@linuxbox ~]$ less ls-output.txt
```

Now, let's repeat our redirection test, but this time with a twist. We'll change the name of the directory to one that does not exist:

```
[me@linuxbox ~]$ ls -l /bin/usr > ls-output.txt
ls: cannot access /bin/usr: No such file or directory
```

We received an error message. This makes sense since we specified the non-existent directory `/bin/usr`, but why was the error message displayed on the screen rather than being redirected to the file `ls-output.txt`? The answer is that the `ls` program does not send its error messages to standard output. Instead, like most well-written Unix programs, it sends its error messages to standard error. Since we only redirected standard output and not standard error, the error message was still sent to the screen. We'll see
Redirecting Standard Output

how to redirect standard error in just a minute, but first, let's look at what happened to our output file:

```
[me@linuxbox ~]$ ls -l ls-output.txt
-rw-rw-r-- 1 me me 0 2008-02-01 15:08 ls-output.txt
```

The file now has zero length! This is because, when we redirect output with the “>” redirection operator, the destination file is always rewritten from the beginning. Since our `ls` command generated no results and only an error message, the redirection operation started to rewrite the file and then stopped because of the error, resulting in its truncation. In fact, if we ever need to actually truncate a file (or create a new, empty file) we can use a trick like this:

```
[me@linuxbox ~]$ > ls-output.txt
```

Simply using the redirection operator with no command preceding it will truncate an existing file or create a new, empty file.

So, how can we append redirected output to a file instead of overwriting the file from the beginning? For that, we use the “>>” redirection operator, like so:

```
[me@linuxbox ~]$ ls -l /usr/bin >> ls-output.txt
```

Using the “>>” operator will result in the output being appended to the file. If the file does not already exist, it is created just as though the “>” operator had been used. Let's put it to the test:

```
[me@linuxbox ~]$ ls -l /usr/bin >> ls-output.txt
[me@linuxbox ~]$ ls -l /usr/bin >> ls-output.txt
[me@linuxbox ~]$ ls -l /usr/bin >> ls-output.txt
```

```
-rw-rw-r-- 1 me me 503634 2008-02-01 15:45 ls-output.txt
```

We repeated the command three times resulting in an output file three times as large.

Redirecting Standard Error

Redirecting standard error lacks the ease of a dedicated redirection operator. To redirect
standard error we must refer to its file descriptor. A program can produce output on any of several numbered file streams. While we have referred to the first three of these file streams as standard input, output and error, the shell references them internally as file descriptors zero, one and two, respectively. The shell provides a notation for redirecting files using the file descriptor number. Since standard error is the same as file descriptor number two, we can redirect standard error with this notation:

```
[me@linuxbox ~]$ ls -l /bin/usr 2> ls-error.txt
```

The file descriptor “2” is placed immediately before the redirection operator to perform the redirection of standard error to the file `ls-error.txt`.

Redirecting Standard Output And Standard Error To One File

There are cases in which we may wish to capture all of the output of a command to a single file. To do this, we must redirect both standard output and standard error at the same time. There are two ways to do this. First, the traditional way, which works with old versions of the shell:

```
[me@linuxbox ~]$ ls -l /bin/usr > ls-output.txt 2>&1
```

Using this method, we perform two redirections. First we redirect standard output to the file `ls-output.txt` and then we redirect file descriptor two (standard error) to file descriptor one (standard output) using the notation `2>&1`.

Notice that the order of the redirections is significant. The redirection of standard error must always occur after redirecting standard output or it doesn't work. In the example above,

```
>ls-output.txt 2>&1
```

redirects standard error to the file `ls-output.txt`, but if the order is changed to

```
2>&1 >ls-output.txt
```

standard error is directed to the screen.

Recent versions of bash provide a second, more streamlined method for performing this
combined redirection:

```bash
[me@linuxbox ~]$ ls -l /bin/usr &> ls-output.txt
```

In this example, we use the single notation &> to redirect both standard output and standard error to the file `ls-output.txt`.

### Disposing Of Unwanted Output

Sometimes “silence is golden,” and we don’t want output from a command, we just want to throw it away. This applies particularly to error and status messages. The system provides a way to do this by redirecting output to a special file called “/dev/null”. This file is a system device called a *bit bucket* which accepts input and does nothing with it. To suppress error messages from a command, we do this:

```bash
[me@linuxbox ~]$ ls -l /bin/usr 2> /dev/null
```

### /dev/null In Unix Culture

The bit bucket is an ancient Unix concept and due to its universality, has appeared in many parts of Unix culture. When someone says he/she is sending your comments to `/dev/null`, now you know what it means. For more examples, see the [Wikipedia article on “/dev/null”](https://en.wikipedia.org/wiki/Dev_null).

### Redirecting Standard Input

Up to now, we haven’t encountered any commands that make use of standard input (actually we have, but we’ll reveal that surprise a little bit later), so we need to introduce one.

**cat – Concatenate Files**

The `cat` command reads one or more files and copies them to standard output like so:

```bash
cat [file...]
```
In most cases, you can think of `cat` as being analogous to the `TYPE` command in DOS. You can use it to display files without paging, for example:

```
[me@linuxbox ~]$ cat ls-output.txt
```

will display the contents of the file `ls-output.txt`. `cat` is often used to display short text files. Since `cat` can accept more than one file as an argument, it can also be used to join files together. Say we have downloaded a large file that has been split into multiple parts (multimedia files are often split this way on USENET), and we want to join them back together. If the files were named:

```
movie.mpeg.001  movie.mpeg.002  ...  movie.mpeg.099
```

we could join them back together with this command:

```
cat movie.mpeg.* > movie.mpeg
```

Since wildcards always expand in sorted order, the arguments will be arranged in the correct order.

This is all well and good, but what does this have to do with standard input? Nothing yet, but let's try something else. What happens if we type “cat” with no arguments:

```
[me@linuxbox ~]$ cat
```

Nothing happens, it just sits there like it's hung. It may seem that way, but it's really doing exactly what it's supposed to.

If `cat` is not given any arguments, it reads from standard input and since standard input is, by default, attached to the keyboard, it's waiting for us to type something! Try this:

```
[me@linuxbox ~]$ cat
The quick brown fox jumped over the lazy dog.
```

Next, type a `Ctrl-d` (i.e., hold down the Ctrl key and press “d”) to tell `cat` that it has reached end of file (EOF) on standard input:
Redirecting Standard Input

In the absence of filename arguments, `cat` copies standard input to standard output, so we see our line of text repeated. We can use this behavior to create short text files. Let’s say that we wanted to create a file called “lazy_dog.txt” containing the text in our example. We would do this:

```
[me@linuxbox ~]$ cat > lazy_dog.txt
The quick brown fox jumped over the lazy dog.
```

Type the command followed by the text we want in to place in the file. Remember to type `Ctrl-d` at the end. Using the command line, we have implemented the world’s dumbest word processor! To see our results, we can use `cat` to copy the file to stdout again:

```
[me@linuxbox ~]$ cat lazy_dog.txt
The quick brown fox jumped over the lazy dog.
```

Now that we know how `cat` accepts standard input, in addition to filename arguments, let’s try redirecting standard input:

```
[me@linuxbox ~]$ cat < lazy_dog.txt
The quick brown fox jumped over the lazy dog.
```

Using the “<” redirection operator, we change the source of standard input from the keyboard to the file `lazy_dog.txt`. We see that the result is the same as passing a single filename argument. This is not particularly useful compared to passing a filename argument, but it serves to demonstrate using a file as a source of standard input. Other commands make better use of standard input, as we shall soon see.

Before we move on, check out the man page for `cat`, as it has several interesting options.

**Pipelines**

The ability of commands to read data from standard input and send to standard output is utilized by a shell feature called `pipelines`. Using the pipe operator “|” (vertical bar), the
standard output of one command can be piped into the standard input of another:

```
command1 | command2
```

To fully demonstrate this, we are going to need some commands. Remember how we said there was one we already knew that accepts standard input? It's `less`. We can use `less` to display, page-by-page, the output of any command that sends its results to standard output:

```
[me@linuxbox ~]$ ls -l /usr/bin | less
```

This is extremely handy! Using this technique, we can conveniently examine the output of any command that produces standard output.

**Filters**

Pipelines are often used to perform complex operations on data. It is possible to put several commands together into a pipeline. Frequently, the commands used this way are referred to as filters. Filters take input, change it somehow and then output it. The first one we will try is `sort`. Imagine we wanted to make a combined list of all of the executable programs in `/bin` and `/usr/bin`, put them in sorted order and view it:

```
[me@linuxbox ~]$ ls /bin /usr/bin | sort | less
```

Since we specified two directories (`/bin` and `/usr/bin`), the output of `ls` would have consisted of two sorted lists, one for each directory. By including `sort` in our pipeline, we changed the data to produce a single, sorted list.

**uniq - Report Or Omit Repeated Lines**

The `uniq` command is often used in conjunction with `sort`. `uniq` accepts a sorted list of data from either standard input or a single filename argument (see the `uniq` man page for details) and, by default, removes any duplicates from the list. So, to make sure our list has no duplicates (that is, any programs of the same name that appear in both the `/bin` and `/usr/bin` directories) we will add `uniq` to our pipeline:
Pipelines

[me@linuxbox ~]$ ls /bin /usr/bin | sort | uniq | less

In this example, we use `uniq` to remove any duplicates from the output of the `sort` command. If we want to see the list of duplicates instead, we add the “-d” option to `uniq` like so:

[me@linuxbox ~]$ ls /bin /usr/bin | sort | uniq -d | less

**wc – Print Line, Word, And Byte Counts**

The `wc` (word count) command is used to display the number of lines, words, and bytes contained in files. For example:

[me@linuxbox ~]$ wc ls-output.txt
7902  64566 503634 ls-output.txt

In this case it prints out three numbers: lines, words, and bytes contained in `ls-output.txt`. Like our previous commands, if executed without command line arguments, `wc` accepts standard input. The “-l” option limits its output to only report lines. Adding it to a pipeline is a handy way to count things. To see the number of programs we have in our sorted list, we can do this:

[me@linuxbox ~]$ ls /bin /usr/bin | sort | uniq | wc -l
2728

**grep – Print Lines Matching A Pattern**

grep is a powerful program used to find text patterns within files. It's used like this:


grep pattern [file...]

When `grep` encounters a “pattern” in the file, it prints out the lines containing it. The patterns that `grep` can match can be very complex, but for now we will concentrate on simple text matches. We'll cover the advanced patterns, called *regular expressions* in a
Later chapter.

Let's say we want to find all the files in our list of programs that had the word “zip” embedded in the name. Such a search might give us an idea of some of the programs on our system that had something to do with file compression. We would do this:

```
[me@linuxbox ~]$ ls /bin /usr/bin | sort | uniq | grep zip
bunzip2
bzip2
gunzip
gzip
unzip
zip
zipcloak
zipgrep
zipinfo
zipnote
zipsplit
```

There are a couple of handy options for `grep`: “-i” which causes `grep` to ignore case when performing the search (normally searches are case sensitive) and “-v” which tells `grep` to only print lines that do not match the pattern.

**head / tail – Print First / Last Part Of Files**

Sometimes you don't want all of the output from a command. You may only want the first few lines or the last few lines. The `head` command prints the first ten lines of a file and the `tail` command prints the last ten lines. By default, both commands print ten lines of text, but this can be adjusted with the “-n” option:

```
[me@linuxbox ~]$ head -n 5 ls-output.txt
total 343496
-rw-r--r-- 1 root root 31316 2007-12-05 08:58 [ 
-rw-r--r-- 1 root root 8240 2007-12-09 13:39 411toppm
-rw-r--r-- 1 root root 111276 2007-11-26 14:27 a2p
-rw-r--r-- 1 root root 25368 2006-10-06 20:16 a52dec
[me@linuxbox ~]$ tail -n 5 ls-output.txt
-rw-r--r-- 1 root root 5234 2007-06-27 10:56 znew
-rw-r--r-- 1 root root 691 2005-09-10 04:21 zonetab2pot.py
-rw-r--r-- 1 root root 930 2007-11-01 12:23 zonetab2pot.pyc
-rw-r--r-- 1 root root 930 2007-11-01 12:23 zonetab2pot.pyo
lrwxrwxrwx 1 root root 6 2008-01-31 05:22 zsoelim -> soelim
```

These can be used in pipelines as well:
### pipelines

```bash
[me@linuxbox ~]$ ls /usr/bin | tail -n 5
znew
zonetab2pot.py
zonetab2pot.pyc
zonetab2pot.pyo
zsoelim
```

tail has an option which allows you to view files in real-time. This is useful for watching the progress of log files as they are being written. In the following example, we will look at the messages file in /var/log. Superuser privileges are required to do this on some Linux distributions, since the /var/log/messages file may contain security information:

```bash
[me@linuxbox ~]$ tail -f /var/log/messages
Feb  8 13:40:05 twin4 dhclient: DHCPACK from 192.168.1.1
Feb  8 13:40:05 twin4 dhclient: bound to 192.168.1.4 -- renewal in 1652 seconds.
Feb  8 13:55:32 twin4 mountd[3953]: /var/NFSv4/musicbox exported to both 192.168.1.0/24 and twin7.localdomain in 192.168.1.0/24,twin7.localdomain
Feb  8 14:07:37 twin4 dhclient: DHCPREQUEST on eth0 to 192.168.1.1 port 67
Feb  8 14:07:37 twin4 dhclient: DHCPACK from 192.168.1.1
Feb  8 14:07:37 twin4 dhclient: bound to 192.168.1.4 -- renewal in 1771 seconds.
Feb  8 14:09:56 twin4 smartd[3468]: Device: /dev/hda, SMART Prefailure Attribute: 8 Seek_Time_Performance changed from 237 to 236
Feb  8 14:10:37 twin4 mountd[3953]: /var/NFSv4/musicbox exported to both 192.168.1.0/24 and twin7.localdomain in 192.168.1.0/24,twin7.localdomain
Feb  8 14:25:07 twin4 sshd(pam_unix)[29234]: session opened for user me by (uid=0)
Feb  8 14:25:36 twin4 su(pam_unix)[29279]: session opened for user root by me(uid=500)
```

Using the “-f” option, tail continues to monitor the file and when new lines are appended, they immediately appear on the display. This continues until you type Ctrl-c.

### tee – Read From Stdin And Output To Stdout And Files

In keeping with our plumbing metaphor, Linux provides a command called tee which creates a “tee” fitting on our pipe. The tee program reads standard input and copies it to both standard output (allowing the data to continue down the pipeline) and to one or more
files. This is useful for capturing a pipeline's contents at an intermediate stage of processing. Here we repeat one of our earlier examples, this time including tee to capture the entire directory listing to the file ls.txt before grep filters the pipeline's contents:

```
[me@linuxbox ~]$ ls /usr/bin | tee ls.txt | grep zip
bunzip2
bzip2
gunzip
gzip
unzip
zip
zipcloak
zipgrep
zipinfo
zipnote
zipsplit
```

## Summing Up

As always, check out the documentation of each of the commands we have covered in this chapter. We have only seen their most basic usage. They all have a number of interesting options. As we gain Linux experience, we will see that the redirection feature of the command line is extremely useful for solving specialized problems. There are many commands that make use of standard input and output, and almost all command line programs use standard error to display their informative messages.

### Linux Is About Imagination

When I am asked to explain the difference between Windows and Linux, I often use a toy analogy.

Windows is like a Game Boy. You go to the store and buy one all shiny new in the box. You take it home, turn it on and play with it. Pretty graphics, cute sounds. After a while though, you get tired of the game that came with it so you go back to the store and buy another one. This cycle repeats over and over. Finally, you go back to the store and say to the person behind the counter, “I want a game that does this!” only to be told that no such game exists because there is no “market demand” for it. Then you say, “But I only need to change this one thing!” The person behind the counter says you can't change it. The games are...
all sealed up in their cartridges. You discover that your toy is limited to the games
that others have decided that you need and no more.

Linux, on the other hand, is like the world's largest Erector Set. You open it up
and it's just a huge collection of parts. A lot of steel struts, screws, nuts, gears,
pulleys, motors, and a few suggestions on what to build. So you start to play with
it. You build one of the suggestions and then another. After a while you discover
that you have your own ideas of what to make. You don't ever have to go back to
the store, as you already have everything you need. The Erector Set takes on the
shape of your imagination. It does what you want.

Your choice of toys is, of course, a personal thing, so which toy would you find
more satisfying?