The Book of the Class for cpe 357
Fall 2023

Phillip L. Nico
Department of Computer Science
California Polytechnic State University
San Luis Obispo, CA 93407
pnico@calpoly.edu

DRAFT(January 11, 2024 at 16:03)

Abstract

These are my lecture notes and nothing more. That means that they are full of omissions and errors and were meant only as a means of helping me remember what I intended to talk about in class. They may or may not bear any resemblance to what was actually said. They may, however, be useful in providing some small insight into the author's state of mind and approach to the course material.

Contents

List of Figures

List of Tables

0 Syllabus

cpe357 — Systems Programming (Nico)
Fall 2023

Administrivia

Professor: Dr. Phillip Nico
`pnico@calpoly.edu`
Office: 14-205

Office Hours:

Monday:	2:10pm–3:00pm[1]
Tuesday:	10:10am–11:00am[1]
Wednesday:	2:10pm–3:00pm[1]
Thursday:	—
Friday:	2:10pm–3:00pm[1]

or by appointment

Texts: Brian W. Kernighan and Dennis M. Ritchie, *The C Programming Language*, Second edition, Prentice-Hall, 1989.

W. Richard Stevens and Stephen A. Rago, *Advanced Programming in the UNIX Environment,* Third edition, The Addison-Wesley Professional Computing Series, Addison-Wesley, 2013.

Webpage: http://www.csc.calpoly.edu/~pnico/class/now/cpe357

Lecture:

Section 5:	MWF 9:10am–10:00am	21-237
Section 7:	MWF 11:10pm–1:00pm	14-301

Lab:

Section 6:	MWF 11:10am–noon	186-c101
Section 8:	MWF 3:10pm–4:00pm	14-232a

Quiz on C: Wednesday, October 18th, 2023 (subject to change)

Midterm: Wednesday, November 8th, 2023 (subject to change)

Final[2]: (Verify with published final exam schedule):

Section 1:	Wednesday, December 13th, 2023	7:10am–10:00am
Section 3:	Friday, December 15th, 2023	10:10am–1:00pm

Grading: The approximate grading breakdown (subject to change) is:

Programs	30%
Labs/Problem Sets	10%
C Language Quiz	—%
Midterm	25%
Final	35%

[1] Office hours are guaranteed until the earlier of the posted end time or the time at which there are no more students. If you know that you will be coming later in the time, let me know in advance.

[2] There is a very good chance that the final will be moved to one of the common final times this quarter.

Course Objectives

The purpose of this course is to gain experience with low-level programming in the UNIX environment. In the process, you will:

- Learn to read and write complex programs in the C programming language.
- Become familiar with standard UNIX user-level commands and the UNIX software development environment.
- Learn about the architecture of the UNIX operating system from a system-programmer's perspective and be able to write programs that use operating system services (system calls) directly.

What you learn, principally, is more about the inner workings of any complex computer system. System services need to be provided and managed, and at some level, someone has to "know" about the actual implementation.

You also learn to distinguish the language from the operating system.

These skills will serve beyond programming on UNIX to help with software development wherever you are working.

Prerequisites

The prerequisites for this course are csc/cpe 203 and cpe 225 or cpe 233. If you have any doubts, come talk to me.

Course Format

The course consists of three lectures and three labs a week. The labs will not meet formally every time—possibly not ever—but I reserve the right to call lab meetings for demos or exercises, etc. The time, however, is intended to be used for the Labs/Problem Sets described below. You are responsible for all material covered in either lecture or lab. If you miss a class, consult a classmate for any missed materials.

The purpose of the class is for everyone to understand C and UNIX systems programming. To this end, if you don't understand something during class, ask. If you are confused, it is likely that a few dozen of your classmates are as well. Also, listen to others' questions. Many times you'll think you understand a concept until you hear someone else's question about it. Dialogue is the best way to learn things, so don't be afraid to speak up.

Office Hours

Office hours are as listed above **or by appointment.** If you are unable to come to the posted office hours, contact me and we can arrange to meet. There is no reason why any of you should be unable to see me if you need to.

Other Resources

I will maintain a class web page at http://www.csc.calpoly.edu/~pnico/class/now/cpe357. On it I will keep information, assignments, announcements, etc. If there any class announcements, corrections, etc., to be made, I will post them in the Announcements section of the class web page, or, for important announcements, distribute them by email. Please check the web page and your email regularly. I will try to make any announcements in class as well, but I cannot guarantee it, and you don't want to miss anything.

I usually post my lecture notes on the web. These are guaranteed to be incomplete and are not a substitute for class attendance. They will, I hope, provide a framework for your own notes and emphasize what I think is important about the class.

Depending on how much time I have, I will also maintain a FAQ on the class page of questions from office hours or the newsgroup that seem to be of general interest.

Laboratory Exercises and Programming Assignments

Programs:

There will be a number of programming assignments over the quarter. Together they will comprise 25% of the total final grade for the course. Because the total number is not known at this point, the individual program weights will not be determined until the end of the quarter. I expect there to be 6–7 programming assignments, but this may vary.

Programming assignments will be distributed on the web, and each assignment will specify both when it is due and whether or not partnerships are allowed.

Labs/Problem Sets:

Most weeks of the quarter there will be a set of laboratory exercises intended to supplement the coursework. These will consist of written exercises and/or experimental work to be performed in the lab. Submission requirements will be posted with each

Late Policy:

Each student will be allowed three (3) discretionary late days to be applied over the quarter. I will keep track of each person's late day balance and charge one for each calendar day (not work day) or portion thereof after the due date. To submit work after the submission directory for an assignment has been closed, use the `latedays` program to reopen it for you. Instructions will be provided on the class web site.

Note: There may be some assignments for which the use of late days will not be allowed. This would be to facilitate the posting of solutions before the midterm or some other reason like that. It will be noted on these assignments that late days are not permitted.

If you are unable to complete an assignment by the specified time and do not have any more late days, turn in what you have for partial credit. **Late assignments will receive no credit.**

Partnerships:

On some of the assignments you may be working with partners. Collaboration tends to help with figuring out difficult concepts and generally makes the whole process more pleasant. A word of caution, though: While it is tempting to just divide up the work, be sure each partner understands the whole project. Concepts learned on the assignments will show up on exams which are worth far more than the individual assignments in the final analysis. Even if your partner bails you out of a tight spot, be sure you understand the work, or it will come back to haunt you. Be absolutely certain that both partners' names appear on all assignments. Credit will be given only to students whose names appear on the assignment.

Each assignment will specify whether working with a partner is permitted.

Submitting Written Work:

Written work should be submitted in class on the day due (as modified by late days) or in the CSC Department drop box outside of (14-254). If you use the drop box, be sure that the names of the instructor, course, section, and assignment (in addition to your own name) are written clearly. Also be aware that materials dropped in the box after it has been collected for the day will be recorded the next day as late. If you are unsure whether the box has been emptied for the day, ask in the office.

For all written work, in order to facilitate grading:

- Start a new page for a new problem if you will not be able to complete it on the current page.
- Place the problems in the order assigned.
- Do not write in red.
- Use proper English grammar and punctuation.
- Write legibly.

Neatness will not necessarily help you, but sloppiness will definitely hurt. If I can't read it, I can't grade it, and I will not guess at what you meant. (This also applies to exams.)

These rules may seem extreme, but they are intended to make grading easier so I can return your papers more quickly.

Submitting Programs:

Programming assignments will be submitted online on the CSL computers using the `handin` program. Instructions for submitting programs are provided on the class web page.

When turning in programming assignments, be careful to submit your final version and to have tested it before submitting.

Programs that fail to compile will receive no credit.

Exams

There will be a midterm and a final. The midterm will be worth 25% of the final course grade and the final will be worth 35%. Exams will emphasize insight and problem solving ability rather than memorization.

Missed Exams: Makeup exams will only be given for the gravest of reasons. If you must miss an exam due to extreme illness, etc., contact the instructor (phone or email is fine) or leave a message with the Department of Computer Science office (805-756-2824) *before* the exam. Be sure to leave both the reason for missing the exam and how to reach you.

Grading Policy

The final grade for this course will be constructed of the following components in the given proportions:

Programs	30%
Labs/Problem Sets	10%
C Language Quiz	—%
Midterm	25%
Final	35%

In general, letter grades will be computed according to the scale:

Minimum A−:	90%
Minimum B−:	80%
Minimum C−:	70%
Minimum D−:	60%

That is, if you score 90% on an exam you will be guaranteed at least an A- for that exam. On some assignments and exams the minimum cutoffs may (and probably will) be lowered to account for difficulty.

Note: I reserve the right to take other circumstances into account when assigning final grades. These include, but are not limited to, such things as substantial improvement over the quarter, significant differences between exam and homework performance, missing homeworks, etc.

Nothing would make me happier than to give everyone an A.

Missing Assignments: Unless special arrangements have been made in advance, a good-faith effort is required of each student for every program. Failure to submit a homework disqualifies the student from the top grade level possible for a final grade. Failure to submit two disqualifies from the top two levels, etc.

Regrades: In general, papers to be considered for regrades must be submitted within one week of the time they become available for picking up. The same is true for misrecorded grades. They must be reported within a week of their posting. Grade feedback will be mailed out periodically as things are graded, and a snapshot of my gradebook for each of you will be available linked from the class web page. Please check them to be sure they agree with your own records.

Collaboration and Cheating

Policy on Collaboration

Programming assignments in this class are intended to be demonstrations of individual or partnership abilities. To this end, programs are to be written only by the designated authors.

High-level discussion of problems and problem-solving techniques, however, is beneficial to all involved. You are encouraged to discuss approaches so long as those with whom you consult are given due credit in your program headers.

> It is *never* acceptable to allow someone else to have your work for reference or to refer to someone else's work while writing your own.

In this case, "someone else's work" means not only other students' programs, but also materials from any other source, including, but not limited to, the world wide web, other reference books, or previous course materials. Also, "not giving your work to others" includes taking reasonable precautions to prevent them from taking it.

Collaboration that goes beyond general approaches or that is uncredited will be considered cheating. If you are unsure about what constitutes proper or improper collaboration, consult the instructor for guidance.

Policy on Cheating

Don't. I consider academic dishonesty a serious offense. Any instances of cheating or plagiarism will be referred to the Office of Student Rights and Responsibilities. The Cal Poly rules and policies are listed in the catalog as well as at the OSRR web site, `http://www.osrr.calpoly.edu`. The general policy, however, is very simply stated in the Campus Administrative Manual (C.A.M. 684):

> **Cheating requires an "F" course grade**

Turning in work is presumed to be a claim of authorship unless explicitly stated otherwise.

If the course rules are unclear or you are unsure of how they apply, ask the instructor *beforehand.*

Feedback

One of the frustrations of teaching is that the instructor rarely gets any feedback on the course until the teaching evaluations at the very end when it is too late to do anything about it. If you like, dislike, or don't understand something I'm doing with the course, please stop by my office hours, send me email, or paste together a note from newspaper clippings and drop it in my mailbox. I won't always change things, but I will always explain why I'm doing them the way I am.

Tentative class schedule

The tentative schedule for the course is given below. This is what we hope to accomplish and when we hope to accomplish it. There may be changes, but this is a rough roadmap for the quarter.

Fall 2023 Preliminary Course Outline

(Subject to Change)

Week	Lecture	Topic	Reading	Date
	1	Introduction and Background	K&R Ch. 1	September 22
	2	Getting started with C	K&R Ch. 2–4	September 25
2	—	*I got sick*		September 27
	3	Getting started with Unix		September 29
	4	Pointers and Arrays in C	K&R Ch. 5	October 2[2]
3	5	Complex Data in C	K&R Ch. 6	October 4
	6	Dynamic Data Structures		October 6
	7	Wrapping Up C		October 9
4	8	Unix from 30,000 feet	Stevens Ch. 1	October 11
	9	System Overview, cont.	(skim Ch. 2)	October 13
	10	Unix File IO		October 16
5	11	**C Language Quiz**		October 18
	12	File IO	Stevens Ch. 3	October 20
	13	Unbuffered vs. Buffered IO		October 23
6	14	The Unix Filesystem	Stevens Ch. 4	October 25
	15	Filesystem, cont.		October 27
	16	Filesystem wrapup		October 30
7	17	Signals	Stevens Ch. 10	November 1
	18	POSIX Signals, Timers, and Alarms		November 3
	19	Terminal IO	Stevens Ch. 18	November 6
8	20	**Midterm**		November 8
	—	*Veterans' Day*		November 10
	21	Terminal IO, cont.	Stevens Ch. 7,8	November 13
9	22	Processes		November 15
	23	Processes, cont.		November 17
	—	*Thanksgiving*		November 20
10	—	*Thanksgiving*		November 22
	—	*Thanksgiving*		November 24
	24	Loading a Program: the execs		November 27
11	25	Building A Shell	Stevens Ch. 15.2	November 29
	26	Shells, cont.		December 1
	27	Managing Concurrency		December 4
12	28	Wrapping Up		December 6
	29	(if needed)		December 8
Final Exam		Section 5: Wednesday, December 13th, 7:10am–10:00am Section 7: Friday, December 15th, 10:10am–1:00pm (*Verify with published schedule)*		

[2]Last day to drop classes is Monday, October 2.

The Last Page

This page is so I can gather a little information about you at the beginning of the class. Please fill it out, tear it off and leave it with me on the way out.

Who are you?

Name: _____
Section: _____
Major: _____
Email: _____

Class Expectations?

Please take a minute to write out what your goals and expectations are for cpe357. This helps me to know where people are coming from and helps to guide the course.

I hate to do this, but to be sure there's no confusion on the matter...

Below, please copy the two boxed text segments from page 5 about academic dishonesty and sign the pledge (assuming you will comply, of course). Without this you will automatically receive a grade of zero for all assignments.

1)

2)

Pledge

I will do my own work in this class. That is, unless it is explicitly permitted by the assignment, I will neither use others' work as my own nor make my work available for others to use. I understand that either of these actions constitutes cheating sufficient to merit a grade of F *for the course.*

Signature Date

1 Lecture: Introduction and Background

Outline:

- What this class is about
- Motivation: Systems Programmer as Toolmaker
  - C and Unix vs. Java and Windows
- Enrollment
- What if you get sick
- Syllabus (Read it)
- The Last Page
- Preparation
- Foundations
  - Unix Systems Programming
  - The C Programming Language
- Summary: C and UNIX
- Relevant Standards
- The Laboratory Environment
- Computer Accounts
- Marching Orders
- Tool of the Week
- Programming Style
  - A note on Style: Obfuscated C

1.1 What this class is about

1.2 Motivation: Systems Programmer as Toolmaker

All along we've been telling you to maintain abstraction: hide the machine under the OS, hide the OS under the language, and don't let any of them know about the other. This class is where we explore the synergy that can happen when all of them know about each other.

What happens when you take advantage of your knowledge of the underlying structures while still playing *within* the rules. It's fail-safe.

It's also about culture: Trust me, the unix environment is a good one. To that end, try and learn to do your development on the Unix machines. The early programs are portable, so you don't have to, but the later ones will not run on windows machines. You will have to develop on a Unix box and you don't want to be learning the tools then.

Objectives:

What is this course about?

The purpose of this course is to gain experience with low-level programming in the UNIX environment. In the process, you will:

- Learn to read and write complex programs in the C programming language.
- Learn to write robust (secure, reliable, responsible) programs in any language.
- Become familiar with standard UNIX user-level commands and the UNIX software development environment.
- Learn about the architecture of the UNIX operating system from a system-programmer's perspective and be able to write programs that use operating system services (system calls) directly.
- What you learn, principally, is more about the inner workings of any complex computer system. System services need to be provided and managed, and at some level, someone has to "know" about the actual implementation.

  You also learn to distinguish the language from the operating system.

  These skills will serve beyond programming on UNIX to help with software development whereever you are working.

1.2.1 C and Unix vs. Java and Windows

The difference betwween a ferarri/jeep and a cadillac. (road trip?)

1.3 Enrollment

Let's see who shows up?

Section	Instructor	Enrolled	Waitlist
cpe357-01	nico	32	16
cpe357-03	Nico	32	18
cpe357-05	Humer	32	20
cpe357-07	Humer	32	22
cpe357-09	Migler	32	17
cpe357-11	Migler	32	19
cpe357-13	Eckhardt	32	19
cpe357-15	Humer	32	15

We're full. Room capacity is 35

1.4 What if you get sick

1.5 Syllabus (Read it)

Really, read it. These are just the highlights.

- The Course Staff
  me: Phillip Nico
  pnico@calpoly.edu
  Office: 14-205
- Prerequisites —

  cpe203 and (cpe233 or cpe225).

  What I expect is that you will have a good programming background and a knowledge of basic machine structures and data representations.
- Format: Three lectures and three labs a week. The labs will not always meet formally, but I do reserve the right to call meetings in the labs. **Speak Up**.
- Office Hours(with high probability):

Monday:	2:10pm–3:00pm[1]
Tuesday:	10:10am–11:00am[1]
Wednesday:	2:10pm–3:00pm[1]
Thursday:	—
Friday:	2:10pm–3:00pm[1]

  Office hours are guaranteed until the earlier of the posted end time or the time at which there are no more students. If you know that you will be coming later in the time, let me know in advance.

  Please do come see me.

  EXCEPT TODAY
- The forum(Yeah, piazza called it winter):

- Texts:

  1. Brian W. Kernighan and Dennis M. Ritchie, *The C Programming Language*, Second edition, Prentice-Hall, 1989. ISBN 0-13-110362-8

  2. W. Richard Stevens and Stephen A. Rago, *Advanced Programming in the UNIX Environment,* Second edition, The Addison-Wesley Professional Computing Series, Addison-Wesley, 2005.

  3. The online manuals sections 2 and 3 (`% man man`)
     How to choose a section?

Section	Subject
1	User commands
2	System calls
3	C language reference
4	special files.
5	file formats and protocols
6	games
7	conventions and misc.
8	Administration commands
9	overview of device driver interfaces

     Solaris: `man -s <section> <keyword>`
     elsewhere: `man <section> <keyword>`

- Web Page. The primary source for course information. Read it.
  `http://www.csc.calpoly.edu/~pnico/class/now/cpe357`
- **email:** Regardless of where you normally read email, read your Cal Poly email or forward it.
  Make sure to use a name! (not `skaterdude23`)
- Grading policy:

Programs	25%
Labs/Problem Sets	10%
C Language Quiz	10%
Midterm	25%
Final	30%

- Assignments and labs ("Exercises"):
  - Labs:
    Most weeks of the quarter there will be a set of laboratory exercises intended to supplement the course-work. These will consist of written exercises and experimental work to be performed in the lab.
    * Lab 1—a warm up—is out (on the web site) and due Friday:
      1. Log in
      2. write a program using a real editor
      3. compile and run it
      4. turn it in
  - Assignments:
    There will be a set of assignments over the quarter intended to demonstrate mastery.
    * Asgn 1 (on the web site) is out. It's due a week from Friday night.
    * Asgn1 is far from difficult. Asgn2 is much more so.
      If this assignment seems fairly straightforward, it is.
      That doesn't mean it's not subtler than it looks:
      · Watch edge cases
      Asgn1 is far from difficult. Asgn2 is much more so.
    * "Good faith effort" required on all assignments. (Missing assignment reduces maximum possible grade.)
    * tryAsgn1
  - Late Policy: Three discretionary late days.

        * Instructions on the web site
        * Only use three.
    - Partnerships and proper collaboration.
        * DO: use each other for help
        * DO NOT: simply split up the work
    - Submitting Work (paper, online) neatness.
    - Testing:
        * Do it.
        * Test edge cases
        * on multiple platforms (32- vs. 64-bit?)
        * Try to break it. I will
- **Classroom Expectations**
    - Decorum. (papers, snoring, gizmometry)
    - 25–35
    - Cameras, etc.
- Exams: quiz, a midterm and a final. Emphasis on problem solving.
- Grading: Policy. 1-week deadline for regrades/errors.
- Academic Misconduct: Don't.
    - Proper Collaboration
        * general principles/approaches
        * debugging consultation
    - Improper Collaboration
        * "Here's my code"
        * *specific* solutions
    - Apply the "Competition" rule
    - Credit any work that comes from anyplace else. (Not doing your homework is better than getting caught cheating. Your name isn't Stevens, is it?)
    - Not giving away code includes taking resonable precautions to not have it stolen.

1.6 The Last Page

Please fill this out as it helps me to know who you are, where you're coming from, and what you expect from the course. It also helps me gauge enrollment.

1.7 Preparation

Where are you all coming from?

- C Programming language
    - Been there, done that
    - Comfortable
    - Some Experience
    - C?
- Unix — user-level
    - Been there, done that
    - Comfortable
    - Some Experience

  - why doesn't "dir" work?

- Unix — development
  - Been there, done that
  - Comfortable
  - Some Experience
  - unix?

1.8 Foundations

1.8.1 Unix Systems Programming

- History
  - Multics (Multiplexed Information and Computing Service)
  - The famous PDP-7
  - Space Travel (Thompson and Ritchie)[1]
- Features
  - Small system
  - modular
  - portable (written in C! (a "high-level" language))
  - designed for people who know what they're doing
  - It's the anti-windows (not meant as a slight — it's a very different philosophy)
- Family—since distributed as source, it mutated
  - AT&T
  - BSD (Berkeley Software Distributions)
  - Linux?

1.8.2 The C Programming Language

- History
  - BCPL (untyped (all datatypes are bitfields))
  - B (untyped)
  - C (weakly typed)
  - Ansi C (more strongly typed) – 1989
  - C++?
- Features
  - Low-level (not weak)
  - Flexible

[1] $75 in 1969 is $541 in 2019 dollars.

1.9 Summary: C and UNIX

Remember where we left off:

- Both are minimalist systems
- Designed by/for experts
- Evolved to work together (Since C is UNIX's implementation language)

Unix philosophy (the anti-windows (not meant as a slight — but it is a very different philosophy))

- Built in layers (see Figure 1)
- build around small tools that can interact.
- Things that are *not* part of the os:
  - Command interpreters
  - Compilers
  - Browsers

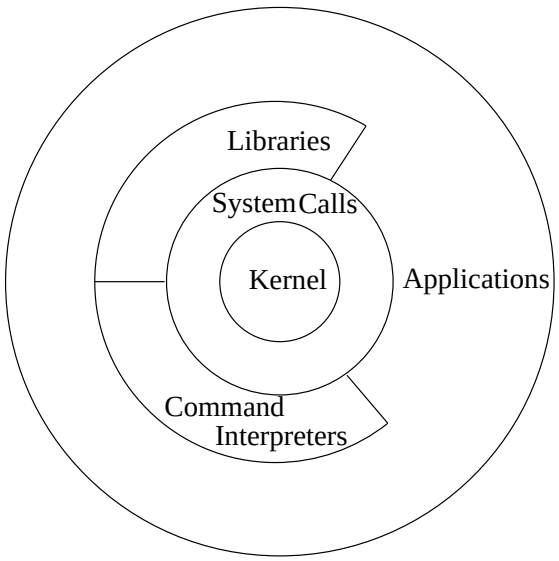


Figure 1: Layering of a UNIX system

1.10 Relevant Standards

- C

  K&R C The first C specification based on the book (1978) . Now mostly lost to history..

  ANSI C (C89) The first real standard C. This is the C standard to which I expect you to adhere in this class (`-std=c89`)

  C99 An updated version of the ANSI standard with some additions for numeric processing. (and the addition of the `restrict` storage class)

- UNIX

  POSIX Portable operating System Interface — 1988

  SUS Single Unix Specification — a superset of POSIX, currently at SUS3

1.11 The Laboratory Environment

Interface

- The shell (sh, bash, csh, tcsh)
  Aside about finding commands.
- `man` (Start w/`man man`)
  - The online manuals sections 1, 2 and 3 (`% man man`)

Section	Subject
1	User commands
2	System calls
3	C language reference
4	special files.
5	file formats and protocols
6	games
7	conventions and misc.
8	Administration commands
9	overview of device driver interfaces

  - How to choose a section?
    Solaris: `man -s <section> <keyword>`
    elsewhere: `man <section> <keyword>`
- Conventions:
  `ls(1)` means `ls` in section 1 of the manual

Editors Use one

- vi
- emacs
- not pico
- definitely not notepad

Compilers

- Traditionally called `cc`
- We'll use the Gnu C Compiler, `gcc`
- For the next few weeks use `-Wall -ansi -pedantic -std=c89`
- C compiles to non-portable object and executable code

1.12 Computer Accounts

Test your programs on multiple platforms.

- unix[1–5].csc.calpoly.edu
- 32- and 64-bit modes

1.13 Marching Orders

1. Get your accounts
2. Go home and read K&R Chapter 1
3. Do Lab 1

1.14 Tool of the Week

A section wherein various tools are showcased.

- Tool of the week candidates:
  1. make
  2. script
  3. grep
  4. expand
  5. profiling

1.15 Programming Style

Aim to be

- Clear
- Consistent
- Then, Efficient

Also, be sure to follow the style guide.

1.15.1 A note on Style: Obfuscated C

C is a very forgiving language: almost anything will parse. This alone makes good programming style important. Programs submitted in this class will be graded for following acceptable programming practices.

Figure 2 shows an obfuscated C program, an example of how *not* to write C. Its output is shown in Figure 3. (Look at the program before you look at the output.)

```
#include <stdio.h>
int main(t,_,a)
char *a;
{
return!0<t?t<3?main(-79,-13,a+main(-87,1-_,main(-86,0,a+1)+a)):
1,t<_?main(t+1,_,a):3,main(-94,-27+t,a)&&t==2?_<13?
main(2,_+1,"%s %d %d\n"):9:16:t<0?t<-72?main(_,t,
"@n'+,#'/*{}w+/w#cdnr/+,{}r/*de}+,/*{*+,/w{%+,/w#q#n+,/#{l+,/n{n+,/+#n+,/#\
;#q#n+,/+k#;*+,/'r :'d*'3,}{w+K w'K:'+}e#';dq#'l \
q#'+d'K#!/+k#;q#'r}eKK#}w'r}eKK{nl]'/#;#q#n')()#}w'){){nl]'/+#n';d}rw' i;#\
 ){nl]!/n{n#'; r{#w'r nc{nl]'/#{l,+'K {rw' iK{;[{nl]'/w#q#n'wk nw' \
iwk{KK{nl]!/w{%'l##w#' i; :{nl]'/*{q#'ld;r'}{nlwb!/*de}'c \
;;{nl'-{}rw]'/+,}##'*}#nc,',#nw]'/+kd'+e}+;#'rdq#w!  nr'/ ') }+}{rl#'{n' ')# \
}'+}##(!!/")
 :t<-50?_==*a?putchar(31[a]):main(-65,_,a+1):main((*a=='/')+t,_,a+1)
  :0<t?main(2,2,"%s"):*a=='/'||main(0,main(-61,*a,
"!ek;dc i@bK'(q)-[w]*%n+r3#l,{}:\nuwloca-O;m .vpbks,fxntdCeghiry"),a+1);
}
```

Figure 2: How not to write C: WhoKnows.c

```
On the first day of Christmas my true love gave to me
a partridge in a pear tree.

On the second day of Christmas my true love gave to me
two turtle doves
and a partridge in a pear tree.

On the third day of Christmas my true love gave to me
three french hens, two turtle doves
and a partridge in a pear tree.

On the fourth day of Christmas my true love gave to me
four calling birds, three french hens, two turtle doves
and a partridge in a pear tree.

On the fifth day of Christmas my true love gave to me
five gold rings;
four calling birds, three french hens, two turtle doves
and a partridge in a pear tree.

On the sixth day of Christmas my true love gave to me
six geese a-laying, five gold rings;
four calling birds, three french hens, two turtle doves
and a partridge in a pear tree.

On the seventh day of Christmas my true love gave to me
seven swans a-swimming,
six geese a-laying, five gold rings;
four calling birds, three french hens, two turtle doves
and a partridge in a pear tree.

On the eigth day of Christmas my true love gave to me
eight maids a-milking, seven swans a-swimming,
six geese a-laying, five gold rings;
four calling birds, three french hens, two turtle doves
and a partridge in a pear tree.

On the ninth day of Christmas my true love gave to me
nine ladies dancing, eight maids a-milking, seven swans a-swimming,
six geese a-laying, five gold rings;
four calling birds, three french hens, two turtle doves
and a partridge in a pear tree.

On the tenth day of Christmas my true love gave to me
ten lords a-leaping,
nine ladies dancing, eight maids a-milking, seven swans a-swimming,
six geese a-laying, five gold rings;
four calling birds, three french hens, two turtle doves
and a partridge in a pear tree.

On the eleventh day of Christmas my true love gave to me
eleven pipers piping, ten lords a-leaping,
nine ladies dancing, eight maids a-milking, seven swans a-swimming,
six geese a-laying, five gold rings;
four calling birds, three french hens, two turtle doves
and a partridge in a pear tree.

On the twelfth day of Christmas my true love gave to me
twelve drummers drumming, eleven pipers piping, ten lords a-leaping,
nine ladies dancing, eight maids a-milking, seven swans a-swimming,
six geese a-laying, five gold rings;
four calling birds, three french hens, two turtle doves
and a partridge in a pear tree.
```

Figure 3: Output of WhoKnows.c

2 Lecture: Getting Started with C and Unix

Outline:

Announcements
Programming Style
Summary: C and UNIX
Coping in the UNIX environment
    The Shell
    Choose a "real" editor: Yes, really.
    The Compiler: gcc
Onwards: The Players
The C Programming Language
    What it is
    A first C Program: Hello.c
Overview of today
The C Language
Pointer Types
Arrays and Strings in C
A C Program's Environment
A C Program's layout
Thoughts on the Assignment: detab

2.1 Announcements

- Coming attractions:

Event	Subject	Due Date			Notes
asgn6	shell	Fri	Dec 8	23:59	

  Use your own discretion with respect to timing/due dates.
- How to run a program
- late days
- Asgn1 is out
- .snapshot
- binary files and `cat -v` and/or `od/xxd`
- "`vim -u NONE`" to ignore .vimrc
- tryAsgn1 is online as ~pn-cs357/demos/tryAsgn1. Remember, error messages (usage messages, too) should go to stderr. (see `fprintf(3)`)
- good faith effort
- Remember to do lab01.
  If you're done with your lab, you don't have to be there. I'll get the time back from you.
- Tutoring center (email me if interested)
- Be reading K&R. It'll help. (really):

Basic Language Concepts	Now	Chapters 2, 3, 4
Pointers and Memory	Next time	Chapters 5, 6
IO	Whenever	Chapters 7

- Asgn 2 out shortly. It will not be trivial.
- Meaning of numbers in parens
- Assignment 1: detab
  - Due next Wednesday, the drop deadline, submitted online to pn-cs357 in the CSL by 11:59pm
    If this assignment seems fairly straightforward, it is. (my soln is 40 lines w/o comments)
    That doesn't mean it's not subtler than it looks:
    * Watch edge cases
    * The handin directory is good to go now.

  - a test script, too. (`tryAsgn1`)

- Compiler options `-ansi -pedantic -Wall -g -std=c89`
  Remember:
  - The style guide requires your programs to compile cleanly w/-Wall
  - Try multiple platforms.
- paths, tr trick, etc.

2.2 Programming Style

Aim to be

- Clear
- Consistent
- Then, Efficient

Also, be sure to follow the style guide.

2.3 Summary: C and UNIX

Remember where we left off:

- Both are minimalist systems
- Designed by/for experts
- Evolved to work together (Since C is UNIX's implementation language)

Unix philosophy (the anti-windows (not meant as a slight — but it is a very different philosophy))

- Built in layers (see Figure 4)
- build around small tools that can interact.
- Things that are *not* part of the os:
  - Command interpreters
  - Compilers
  - Browsers

2.4 Coping in the UNIX environment

2.4.1 The Shell

The shell is the command interpreter to which you give commands
On the CSL, the standard shell given new users is /bin/bash
If you want to find out what the command to do something, try "`apropos`" (or "`man -k`" depending on the flavor)
Role of PATH

2.4.2 Choose a "real" editor: Yes, really.

emacs tutorial C-h t http://www.fsf.org/software/emacs/emacs.html

vi Info on the web site

See www.refcards.com for "cheat sheets."

2.4.3 The Compiler: gcc

Free Software Foundation (and philosophy)
Use this compiler in this course for portability.
In the beginning, everything should compile cleanly with `-Wall -ansi -pedantic -std=c89`. Later on we'll have to drop the ansi requirement, because we'll be doing unix-dependent code.

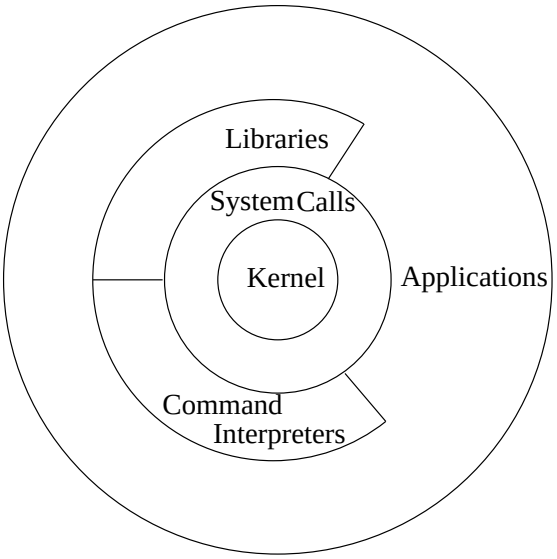


Figure 4: Layering of a UNIX system

2.5 Onwards: The Players

In any system there are many components. This is no different. We have:

- Hardware
- Operating system
- Language (we actually have two)
  - Preprocessor
  - Compiler
- Libraries
  - Static
  - Dynamic

The compilation process is illustrated in Figure 5. Filenames are put in parentheses, program names are in pointy brackets.

The good news is that mostly you don't have to think about these stages.

2.6 The C Programming Language

Note: your second language is the hardest

2.6.1 What it is

C is a low-level language — its types mirror the machine's types: characters, numbers, and addresses.

- What it has:
  - Basic data types: int (short and long), float (double), char
    int is a boolean (0==false)
  - Compound data types:
    arrays, structs
    (null-terminated array of characters is a string)

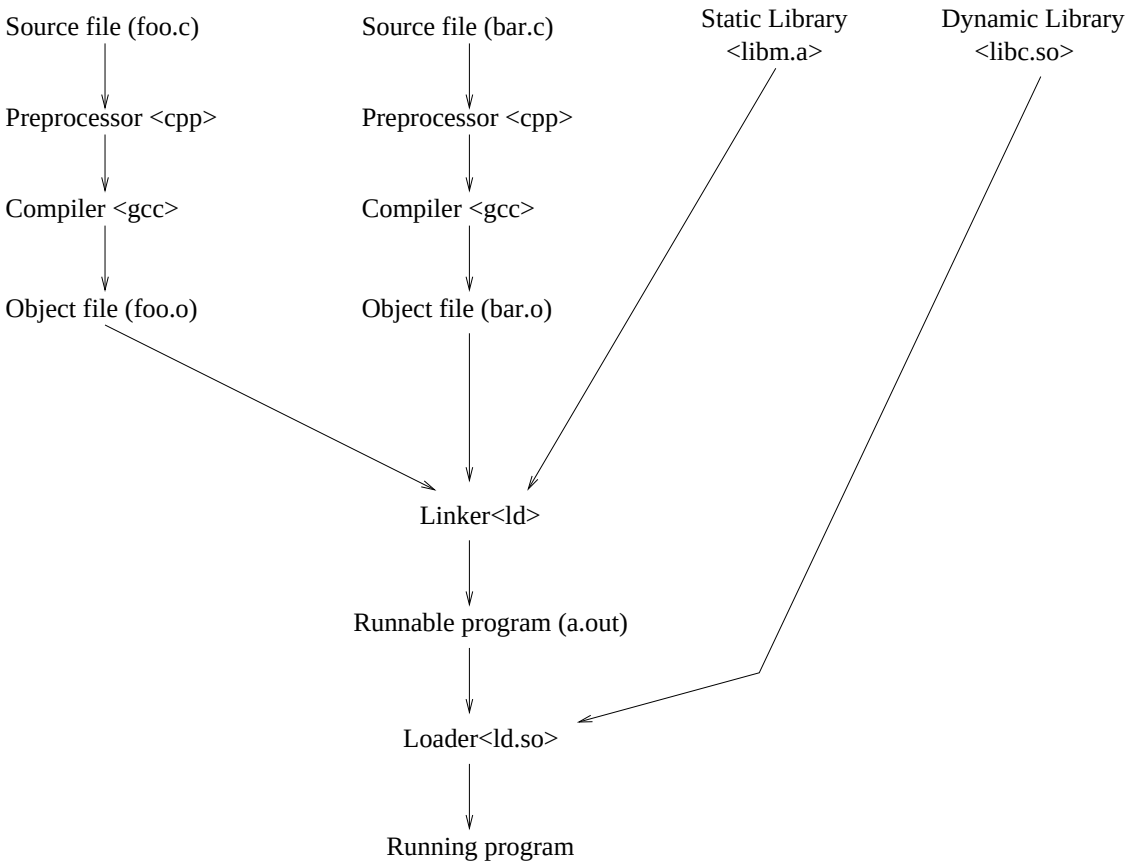


Figure 5: Compiling and linking a C program

  - Pointer types. (like references)
    This means that the model of memory is explicit: the programmer can see it.
  - control structures
  - functions

- What it does not have
  - IO
  - Memory Management (allocation and deallocation)
  - Many other fancy things. (e.g. facilities for complex data manipulation).
- Many of these are provided by libraries that wrap system facilities.
  - stdio
  - strings

2.6.2 A first C Program: Hello.c

A first C program, "Hello World," is shown in Figure 6.
To run a program, just name it.

```
#include <stdio.h>
int main() {
  printf("Hello, World!\n");
  return 0;
}

% gcc Hello.c
% a.out
Hello, World!
%
```

Figure 6: A First C Program: Hello.c

2.7 Overview of today

This will be the last of the "background" lectures.

Next time we will look at some more advanced C programming concepts, then on into the programmers view of Unix.

2.8 The C Language

This time for real.

- Filename conventions:
  - `.c` C source file
  - `.h` Header file (type definitions, prototypes and declarations)
  - `.o` Object file
- A whirlwind tour of C syntax (all this and more can be found in K&R):
  - A C function
  - Declarations (data, local): *[modifier] type name*

    C data types are designed to be mappable to things already available in the architecture.

    char 8 bits
    short $\geq$ 16bits
    long $\geq$ 32bits
    int short $\leq$ int $\leq$ long
    float $\geq$ 32bits
    double $\geq$ 32bits
    boolean no such thing. 0 is false nonzero is true.

    To actually know the size, use `sizeof()`

    (`stdint.h` provides fixed-size types if you need 'em)
  - Naming conventions:
    - constants and macros are ALL CAPS
    - Only 31 characters are significant
  - Can be modified with:

unsigned	unsigned arithmetic
static	link now.
register	optimize for a register
extern	Extern declarations (for variables in libraries)
const	For things that are unchangable
volatile	For things that change by themselves
restrict	promises no aliasing

    Of these, `unsigned`, `static`, `extern` and `const` are important for you to know.

- Operators
- Control structures:
  - if-else
  - switch
  - while
  - do-while
  - for
- Functions
  - Parameters: **all passed by value**
  - Function declarations (prototypes)
    - K&R style
    - ANSI style (use these!)
  - Function definitions
- The pre-processor: macros and conditional compilation
  - `#include`
  - `#define`
  - `#ifdef`
  - `#ifndef`
- How to know which headers you need

2.9 Pointer Types

- A pointer is a variable that stores a location
- Read pointer (really all) declarations inside→out and backwards. e.g.,

  ```
  int *x
  ```

  as "x is a thing that points to an integer"

2.10 Arrays and Strings in C

Just like Java, except:

- Syntax is a little different
  - Type and shape.
  - (to come later) Arrays as parameters: The size of the first dimension can be left empty.
- Creation is the same as declaration.
- Not-resizable
- **No bounds checking**

No dynamic allocation is necessary for asgn1.

2.11 A C Program's Environment

- It's *called*
- (parameters to main)
- stdin/stdout/stderr
- Exit status (what good is it.)

2.12 A C Program's layout

The OS divides the running program (a *process*) into segments:

- Text
- Data
- Stack
- Heap

2.13 Thoughts on the Assignment: detab

- Think about what you know
- Think about what you don't know (line length?)
- What would you do if this were a Java program? (it only differs by a few words).

3 Lecture: Getting started with Unix

Outline:

Announcements
Thoughts on the Assignment: detab
From last time: A C Program's Environment
From last time: A C Program's layout
Pointer Types
Arrays and Strings in C
Coping in the UNIX environment
Unix Family tree
Unix Stuff
    RTFM
The Shell
User-level UNIX
    Useful Commands
Foreshadowing
From last time: A C Program's Environment
A C Program's layout
Aliasing `rm` to move to Trash
The snapshots directory in the CSL

3.1 Announcements

- Coming attractions:

Event	Subject	Due Date			Notes
asgn6	shell	Fri	Dec 8	23:59	

  Use your own discretion with respect to timing/due dates.

- Asgn1 test script `~pn-cs357/bin/tryAsgn1`

```
  The UNIX philosophy basically involves giving you enough rope to
hang yourself.  And then a couple of feet more, just to be sure.
```

- "`vim -u NONE`" to ignore .vimrc
- Paths and finding gcc (example)
- Line buffering
- Characters are *signed*
- binary files and `cat -v` and/or `od/xxd`
- Program that prints its own arguments
- You can find the test cases (The test script prints 'em)
- tryAsgn1 is online as ~pn-cs357/demos/tryAsgn1. Remember, error messages (usage messages, too) should go to stderr. (see `fprintf(3)`)
- Trivia (wrt)Asgn1
  - std{in,out,err}
  - redirection
  - tr trick
  - tilde expansion
- Be reading K&R. It'll help. (really):

Basic Language Concepts	Now	Chapters 2, 3, 4
Pointers and Memory	Next time	Chapters 5, 6
IO	Whenever	Chapters 7

- Lab 02 will be out shortly.
  1. a few quick written exercises

2. Picture exercise
3. A GDB exercise
4. A Make exercise
5. A simple utility, `uniq`.

- Piazza forum will be live as soon as enrollment has stabilized stabilized
- The syllabus has been updated with a calendar including exams

3.2 Thoughts on the Assignment: detab

- Think about what you know (and think like a typewriter)
- Think about what you don't know (line length?)
- The `tr(1)` trick.
- What would you do if this were a Java program? (it only differs by a few words).

3.3 From last time: A C Program's Environment

- It's *called*
- (parameters to main)
- Three open FILE *s: stdin/stdout/stderr
- Exit status (what good is it.)

3.4 From last time: A C Program's layout

The OS divides the running program (a *process*) into segments:

- Text
- Data
- Stack
- Heap

3.5 Pointer Types

- A pointer is a variable that stores a location
- Read pointer (really all) declarations inside→out and backwards. e.g.,

  ```
  int *x
  ```

  as "x is a thing that points to an integer"

3.6 Arrays and Strings in C

Just like Java, except:

- Syntax is a little different
  - Type and shape.
  - (to come later) Arrays as parameters: The size of the first dimension can be left empty.
- Creation is the same as declaration.
- Not-resizable (There are techniques for dynamic allocation, but we'll talk about them later.)
- A string is an array of `char` with a zero byte at the end
- **No bounds checking**

No dynamic allocation is necessary for asgn1.

3.7 Coping in the UNIX environment

3.8 Unix Family tree

An overview of the family.

3.9 Unix Stuff

Connecting to the machine: ssh
A quick overview of the users' view of the system: (the *file* and the *process*) (The static and dynamic entities in the system.)

filesystem No matter how many disks and other devices exist on the machine, the system makes it look like one large filesystem starting at a "root" called /

On our systems, the home directory filesystem is shared everywhere.

users Individual user accounts each have their own identity and ownership. (uid, gid)

processes Every program execution creates a new process on the system. ps(1). A *process* is an active entity in the system, whereas a program is simply a file with execute permission.

Processes have identity:

- User ID
- Group ID
- Process ID

Processes have resources:

- memory
- time
- IO Streams (stdin,stdout,stderr)

commands Most user commands reside in one of three places: `/bin`, `/usr/bin`, or `/usr/local/bin`. When you try to run a program it looks in these locations.

shell The shell is the command interpreter you are talking to. The shell executes commands and keeps track of your environment. The shell can read commands interactively or from a file. (.login, .cshrc, .logout or .bashrc (interactive) and .bash_profile (login))

conventions:

- `ls(1)` means `ls` in section 1 of the manual
- files or directories are referred to by paths
  - A path starting at / is **absolute**
  - Any other path is **relative** (to the current working directory. Every process (and the shell is one) has a concept of a current working directory.

  t csh and bash expand ~ to be your own home directory and *~user* to be *user's* home directory.

3.9.1 RTFM

Read the manual!

- On the system you're using
- How to know which headers you need?

3.10 The Shell

Shell families (/etc/shells)

The shell does everything necessary to run programs for you and sets up proper IO for them:

- Path searches (how to see your path)
- IO Redirection: "into" and "from" (> and <)
- IO Redirection: stderr

  {t,}csh: `a.out >& item`

  {ba,}sh: `a.out > outfile 2>& 1`
- The uses of bang (!)
- IO Redirection: pipes
- Different quotes
- Some shell behavior can be controlled by variables
  - Two kinds: shell and environment (don't worry about the difference now)
  - e.g. PATH, noclobber
- Globbing

3.11 User-level UNIX

Philosophy

- the system is designed for the expert (contrast Windoze)
- most commands do simple things. Many are designed as filters that can be combined into more complex systems.

  ```
  cat Roster* | grep SO | sort | uniq | wc -l
  ```
- e.g. how many unique students are on the waitlist.
- tr example for `wc(1)` (or `expand(1)`) for seeing spaces.

3.11.1 Useful Commands

Subject areas:

Subject	Related commands/concepts
Connectivity	**ssh(1)**, scp(1)
Navigation	pwd(1), mkdir(1), rmdir(1), cd(1), (pushd(1), popd(1), dirs(1))
File manipulation	ls(1), cat(1), more(1), rm(1), mv(1), IO redirection
Process manipulation	ps(1), kill(1)
File permissions	ls -l, chmod(1)
Shell tricks	alias, .login, .cshrc, .bashrc, paths, redirection, running programs, shell and environment variables
Utilities	diff(1), grep(1), more(1), sort(1), uniq(1), lpr(1), ...
The Joy of X(7)	Life at the console is a good thing.
Other Stuff	Useful information

3.12 Foreshadowing

For the next few lectures will continue our crash course in C with some more advanced C topics. In particular:

1. Memory management (pointers, malloc(), free())
2. Compound data types (structs, arrays, unions?, typedef)
3. Development tools and techniques (headers, cpp, make, gdb)

We will see the first example of the difference between OS services (`sbrk(2)`) and C library services (`malloc(3)`).

3.13 From last time: A C Program's Environment

- It's *called*
- (parameters to main)
- stdin/stdout/stderr
- Exit status (what good is it?)

3.14 A C Program's layout

The OS divides the running program (a *process*) into segments:

- Text
- Data
- Stack
- Heap

3.15 Aliasing `rm` to move to Trash

This is in response to a question asked in class where I said I'd look up the answer. It's really straightforward in `csh` and derivatives, but a little more complicated in `bash`

- First, create your `Trash` directory:

```
mkdir ~/Trash
```

- Now, if you're running `csh` you create the alias thusly

```
% alias rm /bin/mv \!\* ~/Trash
```

- Now, if you're running `bash` you create a function, then alias it:

```
$ saferm () { /bin/mv $@ ~/Trash; }
$ alias rm=saferm
```

Either way, once you've worked out what you want it to be, you'll want to install it in your `.bashrc` or `.cshrc`.

3.16 The snapshots directory in the CSL

- `.shapshot`
- it'll save you
- even if you're not using source code control, back your stuff up.

4 Lecture: Pointers and Arrays in C

Outline:

- Announcements
- Array types
- A programming example: cat
  - The stdio library
  - A program example: `cat`
- The C Memory Model
- Pointers and Addresses
  - Basics
- Pointers: Simple
  - Declaration and use
- Pointers and Arithmetic
- Pointers and Arrays
- Arrays as parameters
- Strings
- Add a discussion of scope and lifetime here
- Dynamic Memory Management: malloc() and free()
  - Strings and dynamic memory: `strdup()`
- Error handling
  - Example: safe_malloc()
- Tool of the week: Make(1)

4.1 Announcements

- Coming attractions:

Event	Subject	Due Date			Notes
asgn6	shell	Fri	Dec 8	23:59	

  Use your own discretion with respect to timing/due dates.

- Lab02 will be out shortly.

  1. A few written exercises
  2. Picture
  3. GDB
  4. Learn Make(1)
  5. write `uniq(1)`

- Asgn2 out shortly
- how to determine which headers you need
- `getopt(3)` is not ANSI. It's still in the library, though.
- Web authentication will be through the portal. (for solutions)
- Reminder about how to use late days

4.2 Array types

4.3 A programming example: cat

Last time we did a major overview of C, let's do some.

4.3.1 The stdio library

This is Chapter 7 of K&R.

Based on `FILE *` streams

```
FILE *fopen(const char *path, const char *mode);
int fclose(FILE *stream);
int fgetc(FILE *stream);
char *fgets(char *s, int size, FILE *stream);
int getc(FILE *stream);
int getchar(void);
int ungetc(int c, FILE *stream);
int printf(const char *format, ...);
int fprintf(FILE *stream, const char *format, ...);
int feof(FILE *stream);
```

4.3.2 A program example: `cat`

```
#include <stdio.h>

int main(int argc, char *argv[]) {
  int c;

  for(c=getchar(); c != EOF ; c = getchar() ) {
    putchar(c);
  }

  return 0;
}
```

Figure 7: A simple implementation of `cat()`

```
#include <stdio.h>

int main(int argc, char *argv[]) {
  int c;

  while ( (c=getchar()) != EOF )
    putchar(c);

  return 0;
}
```

Figure 8: A more clever version `cat()`

```
#include <stdio.h>

int main(int argc, char *argv[]) {
  int c;

  while ( EOF != putchar(c=getchar()))
    /* This program is broken */;

  return 0;
}
```

Figure 9: A version that's so clever it's broken

4.4 The C Memory Model

C exposes memory completely to the programmer:

- quick review of what memory is and layout
- quick review of layout of process memory.
- quick review of memory access from the assembly level.

4.5 Pointers and Addresses

K&R Chapter 5: Pointers

4.5.1 Basics

Pointers expose the underlying memory system:

- Necessary: Allows functions to modify parameters, e.g.
- Dangerous: Allows functions to modify parameters

4.6 Pointers: Simple

4.6.1 Declaration and use

Pointers are *typed*. That is, a pointer must know what sort of thing it is pointing to.

Pointers to basic types: int *, char *, double *, void *

Assignment

Operators:

`&` address of

`*` dereference

Note:

- To do anything useful, they must be *pointed at* something.
- `NULL` is defined to be an invalid pointer. It's not necessarily zero, but guaranteed to compare as zero. (it's a standards thing.)

The standard example: `swap(int *a, int *b)` can be seen in Figure 17.

```
void swap ( int *a, int *b ) {
  /* using indirection, we can exchange the values */
  int t;
  t  = *a;
  *a = *b;
  *b = t;
}

/* A sample call: */
int x,y;
swap(&x,&y);
```

Figure 10: Exchanging two values: `swap.c`

4.7 Pointers and Arithmetic

Because pointers are memory addresses, they are subject to arithmetic and other operations, but in order for them to make sense, we have to talk about arrays again..

Onwards: Before discussing pointers with respect to compound data types, we need to talk about C's compound data types.

4.8 Pointers and Arrays

Arrays are allocated blocks of data of the given type.

An array's name can be used as a pointer to the head of the array, so, given `int foo[5]` exists, `foo[3]` is equivalent to `*(foo+3)`.

$$\begin{aligned} A[3] &= *(A+3) \\ &= *(3+A) \\ &= 3[A] \end{aligned}$$

4.9 Arrays as parameters

Array parameters can be declared two ways:

- `type *name`
- `type name[]` (Empty-braces form only in parameter lists.)

4.10 Strings

Strings are arrays of characters with a zero byte at the end. Compare the different implementations of two library calls:

Function	Header	Array	Pointer
`size_t strlen(const char *s);`	`#include<string.h>`	Fig. 18	Fig. 19
`int puts(const char *s);`	`#include<stdio.h>`	Fig. 20	Fig. 21

Figures 18 through 21 illustrate the difference between approaching a parameter as an array vs. as a pointer.

4.11 Add a discussion of scope and lifetime here

- Automatics
- data segment/heap objects (small=o)

4.12 Dynamic Memory Management: malloc() and free()

(*remember C doesn't itself provide memory management. This is part of the library. Contrast to:*

```
int brk(void *end_data_segment);
void *sbrk(ptrdiff_t increment);
```

provided by the OS)

void *malloc(size_t size) Allocates a new block of memory of the requested size and returns a pointer to it (or returns NULL).

void free(void *ptr); returns the pointed-to segment of memory to the library pool to be reallocated by a subsequent call to malloc();

void *realloc(void *ptr, size_t size) takes the given block of memory and allocates a (possibly different) block of memory of `size` bytes with the same contents as the original block. `realloc(3)` returns a pointer to the new block and may free the old one.

Using malloc() and free():

- Always use `sizeof()` for portability. This is particularly true for structures. (There may be gaps due to alignment issues.)
- Always cast your return value appropriately.
- **Always** check your return value for success and handle it appropriately. I usually write something like `safe_malloc()` (below)
- Always free() anything you allocated once you're done with it. (Memory leaks are a bad thing.) This can be trickier than it looks for complex programs. Reference counters can be helpful.
- Never free something that was not allocated by malloc().

```
int strlen ( char s[] ) {
  int len;
  for ( len = 0 ; s[len] != '\0' ; len++ )
     /* nothing */;
  return len;
}
```

Figure 11: An array-based version of `strlen()`.

```
int strlen ( char *s ) {
  /* this might be a bit too clever for its own good. */
  int len = 0;
  while ( *s++ )
    len++;
  return len;
}
```

Figure 12: A pointer-based version of `strlen()`.

```
void puts ( char s[] ) {
  int i;
  for ( i = 0 ; s[i] != '\0' ; i++ )
    putchar(s[i]);
}
```

Figure 13: An array-based version of `puts()`.

```
void puts ( char *s ) {
  while( *s != '\0' )
    putchar( *s++ );
}
```

Figure 14: A pointer-based version of `puts()`.

4.12.1 Strings and dynamic memory: `strdup()`

Function	Header	Implementation
`char *strdup(const char *s);`	`#include<string.h>`	Fig. 15

```
char *strdup(char *s) {
  /* returns a copy of the given string in a newly allocated
   * region of memory
   */
  char *new=NULL;
  if ( s ) {
    new = malloc ( strlen(s)+1 );
    if ( new )
      strcpy(new,s);
  }
  return new;
}
```

Figure 15: A string duplication function: `strdup()`.

4.13 Error handling

Always check the return values of library functions and system calls. (except `exit()`)

```
#include <stdio.h>

void perror(const char *s);

#include <errno.h>

const char *sys_errlist[];
int sys_nerr;
int errno;
```

4.13.1 Example: safe_malloc()

This demonstrates error-checking for malloc() and perror() for error handling.

Errors set `errno` to indicate the error. Calls to `perror()` use this value, too.

From the Solaris Manual:

```
ENOMEM    The physical limits of the system are exceeded  by
          size  bytes of memory which cannot be allocated.

EAGAIN    There is not enough memory available at this point
          in time to allocate size  bytes of memory; but the
          application could try again later.
```

The code can be found in Figure 16

```
void *safe_malloc ( int size ) {
  void *new;
  new = malloc(size);
  if ( new == NULL ) {
    perror(_FUNCTION_);
    exit(−1);
  }
  return new;
}
```

Figure 16: A malloc() that always succeeds (or never comes back)

4.14 Tool of the week: Make(1)

Make is a program to control program builds automagically, but it can be much, much more.

- Based on dependencies—there's no need to regenerate a file if its source hasn't changed.
  A dependency looks like: `target:  source`.
  A particular target can have multiple dependency lines
- Implicit Rules—Make knows how to do certain things (compile C source, for example: if a .o file depends on a .c file if there is one of the same name in the directory).
  You can define your own if you want.
- Explicit rules: After a tab, an series of instructions for making the thing:

```
foo.o: foo.c
        gcc -c -Wall -ansi -pedantic foo.c
```

- It supports variables. In particular CC, SHELL, CFLAGS.
  $(VARNAME) to evaluate. $$ for a dollar sign
- Remember TABS
  (Quite possibly the dumbest decision since `creat()`).
- In rules:

@	do a line silently
#	comment
-	proceed even in the face of errors

- Interfaces nicely with RCS (will check out files for you.)
- To use: `make thing`
  if "thing" isn't specified, it makes the first target.
- Emacs: M-x compile

For more info, read the man page, or the info page.

5 Lecture: Complex Data and Dynamic Data Structures

Outline:

5.1 Announcements

- Coming attractions:

Event	Subject	Due Date			Notes
asgn6	shell	Fri	Dec 8	23:59	

  Use your own discretion with respect to timing/due dates.

- valgrind
- Missing office hours this afternoon
- Gradebook
- TUI (Text User Interface) mode (C-x A)
- Shell history tricks
- Shell quotes
- Redirection details
- gcc -v to get search path

```
/usr/lib/gcc/x86_64-linux-gnu/5/include
/usr/local/include}
/usr/lib/gcc/x86_64-linux-gnu/5/include-fixed
/usr/include/x86_64-linux-gnu
/usr/include
```

- Lab02 and asgn2 now have due dates
- Lab02: Reading long lines
  - Guess a length
  - Be prepared to go back for more
- -O — turns on dataflow analysis, but can break gdb
- Asgn2: `tryAsgn2` is out.

5.2 Today

1. gdb demo
2. Multi-dimensional arrays vs. arrays of arrays
3. Let's build a list

5.3 Pointers: Simple

5.3.1 Declaration and use

Pointers are *typed*. That is, a pointer must know what sort of thing it is pointing to.

Pointers to basic types: int *, char *, double *, void *

Assignment

Operators:

`&` address of

`*` dereference

Note:

- To do anything useful, they must be *pointed at* something.
- `NULL` is defined to be an invalid pointer. It's not necessarily zero, but guaranteed to compare as zero. (it's a standards thing.)

The standard example: `swap(int *a, int *b)` can be seen in Figure 17.

```
void swap ( int *a, int *b ) {
  /* using indirection, we can exchange the values */
  int t;
  t  = *a;
  *a = *b;
  *b = t;
}

/* A sample call: */
int x,y;
swap(&x,&y);
```

Figure 17: Exchanging two values: `swap.c`

5.4 Pointers and Arithmetic

Because pointers are memory addresses, they are subject to arithmetic and other operations, but in order for them to make sense, we have to talk about arrays again..

Onwards: Before discussing pointers with respect to compound data types, we need to talk about C's compound data types.

5.5 Pointers and Arrays

Arrays are allocated blocks of data of the given type.

An array's name can be used as a pointer to the head of the array, so, given `int foo[5]` exists, `foo[3]` is equivalent to `*(foo+3)`.

$$\begin{aligned} A[3] &= *(A+3) \\ &= *(3+A) \\ &= 3[A] \end{aligned}$$

5.6 Arrays as parameters

Array parameters can be declared two ways:

- `type *name`
- `type name[]` (Empty-braces form only in parameter lists.)

5.7 Strings

Strings are arrays of characters with a zero byte at the end. Compare the different implementations of two library calls:

Function	Header	Array	Pointer
`size_t strlen(const char *s);`	`#include<string.h>`	Fig. 18	Fig. 19
`int puts(const char *s);`	`#include<stdio.h>`	Fig. 20	Fig. 21

Figures 18 through 21 illustrate the difference between approaching a parameter as an array vs. as a pointer.

5.8 Add a discussion of scope and lifetime here

- Automatics
- data segment/heap objects (small=o)

5.9 From last time: Dynamic allocation

void *malloc(size_t size) Allocates a new block of memory of the requested size and returns a pointer to it (or returns NULL).

void free(void *ptr); returns the pointed-to segment of memory to the library pool to be reallocated by a subsequent call to malloc();

void *calloc(size_t nmemb, size_t size); allocates memory for an array of `nmemb` elements of size bytes each and returns a pointer to the allocated memory. The memory is set to zero.

void *realloc(void *ptr, size_t size) takes the given block of memory and allocates a (possibly different) block of memory of `size` bytes with the same contents as the original block. `realloc(3)` returns a pointer to the new block and may free the old one.

5.10 Pointers Summary

Pointers and arrays (Chapter 5).

(A quick review of pointer vs. pointee. Also, see Fig. 22)

- Pointers are just that: pointers.
- To do anything useful, they must be *pointed at* something.
- `NULL` is defined to be an invalid pointer. It's not necessarily zero, but guaranteed to compare as zero. (it's a standards thing.)

5.10.1 Strings and dynamic memory: `strdup()`

Function	Header	Implementation
`char *strdup(const char *s);`	`#include<string.h>`	Fig. 23

Take care as to whether it's reasonable to treat your data as a string at all. (e.g. in `xlat`, '\0' is a valid character.

```
int strlen ( char s[] ) {
  int len;
  for ( len = 0 ; s[len] != '\0' ; len++ )
     /* nothing */;
  return len;
}
```

Figure 18: An array-based version of `strlen()`.

```
int strlen ( char *s ) {
  /* this might be a bit too clever for its own good. */
  int len = 0;
  while ( *s++ )
    len++;
  return len;
}
```

Figure 19: A pointer-based version of `strlen()`.

```
void puts ( char s[] ) {
  int i;
  for ( i = 0 ; s[i] != '\0' ; i++ )
    putchar(s[i]);
}
```

Figure 20: An array-based version of `puts()`.

```
void puts ( char *s ) {
  while( *s != '\0' )
    putchar( *s++ );
}
```

Figure 21: A pointer-based version of `puts()`.

Figure 22: `http://xkcd.com/371/`

```
char *strdup(char *s) {
  /* returns a copy of the given string in a newly allocated
   * region of memory
   */
  char *new=NULL;
  if ( s ) {
    new = malloc ( strlen(s)+1 );
    if ( new )
      strcpy(new,s);
  }
  return new;
}
```

Figure 23: A string duplication function: `strdup()`.

5.11 Tool of last century: rcs(1)

RCS (Revision Control System) is a version-control system.

This is very old fashioned and superceded by CVS, SVN, git, etc., but it requires no setup.

mkdir RCS Create repository

ci -u foo.c Checks in "foo.c" and removes it

ci -u foo.c Checks in "foo.c" and keeps a read-only copy (unlocked)

ci -i foo.c Checks in "foo.c" and keeps a read-write copy (unlocked)

co -u foo.c Checks out "foo.c" read-only (unlocked)

co -l foo.c Checks out "foo.c" read-write (locked)

rcsdiff foo.c shows the difference between this version and the last

rlog foo.c show the log messages.

Emacs interface:

C-x v i Install buffer into RCS

C-x C-q Toggle-read-only checks the file in or out

C-c C-c Complete checkin

5.12 Tool of the week: Make(1)

Make is a program to control program builds automagically, but it can be much, much more.

- Based on dependencies—there's no need to regenerate a file if its source hasn't changed.
  A dependency looks like: `target: source`.
  A particular target can have multiple dependency lines
- Implicit Rules—Make knows how to do certain things (compile C source, for example: if a .o file depends on a .c file if there is one of the same name in the directory).
  You can define your own if you want.
- Explicit rules: After a tab, an series of instructions for making the thing:

```
foo.o: foo.c
        gcc -c -Wall -ansi -pedantic foo.c
```

- It supports variables. In particular CC, SHELL, CFLAGS.
  $(VARNAME) to evaluate. $$ for a dollar sign
- Remember TABS
  (Quite possibly the dumbest decision since `creat()`).
- In rules:

@	do a line silently
#	comment
-	proceed even in the face of errors

- Interfaces nicely with RCS (will check out files for you.)
- To use: `make thing`
  if "thing" isn't specified, it makes the first target.
- Emacs: M-x compile

For more info, read the man page, or the info page.

Arrays as parameters Array parameters can be declared two ways:

- `type *name`
- `type name[]` (Empty-braces form only in parameter lists.)

Multi-dimensional arrays

- declaration
- as parameter (the first dimension can be unspecified)

We've been doing arrays. There's no news there.

5.13 Subtle Distinctions: Multidimensional Arrays vs. arrays of arrays

Many programmers are sloppy about vocabulary.

5.13.1 Example: Two-d Array

```
int A[2][3];
int i,j;
for(i=0;i<2;i++)
  for(j=0;j<3;j++)
    A[i][j] = 10 * i + j;
```

Logical layout:

0	1	2
10	11	12

Physical layout:

0	1	2	10	11	12

5.13.2 Example: Strings and command-line arguments

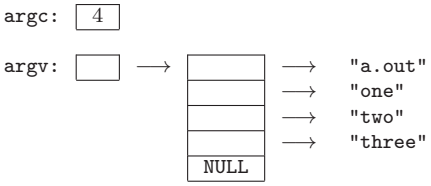
C strings are simply arrays of characters with a nul ('\0') at the end.

Pointers and strings play into the "real" definition of how a C main() is supposed to be defined. The definition takes the number of command-line arguments and an array of their values: `main(int argc, char *argv[])`

So, if your program is invoked as:

```
% a.out one two three
```

these parameters would look like:



If you wanted to print these command line arguments out, you could do something like the program shown in Figure 24. If you're an old-timer and like pointer arithmetic, you could do something like Figure 25, but don't. First of all, it's impossible to read, and second, it destroys the parameters in the process. What if you needed them?

5.13.3 Structures and Typedef

A structure allows you to group data:

```
struct list_node {
  int data;
  struct list_node *next;
};
```

```
#include <stdio.h>

int main(int argc, char *argv[]){
  int i;
  for ( i = 0; i < argc; i++ ) {
    printf("%s ", argv[i]);
  }
  printf("\n");

  return 0;
}
```

Figure 24: Echoing command-line arguments.

```
#include <stdio.h>

int main(int argc, char *argv[]){

  while ( argc-- )
    printf("%s ", *argv++);
  printf("\n");

  return 0;
}
```

Figure 25: Hard to read version of Figure 24.

Note in the example above that it is possible to use the type definition to define a pointer type before it is complete.

Select a field with ".", like:

```
struct list_node thing;
thing.data = 3;
thing.next = NULL;
```

A **typedef** allows you to define a new type. It's just like declaring a variable except the "variable" becomes the name for the new type:

```
typedef struct list_node *node;
```

or, in conjunction with the definition of the node:

```
typedef struct list_node *node;
struct list_node {
  int data;
  node next;
};
```

5.14 Pointers to Structures: More Complex

More or less everything works in the same way with one shortcut.

```
struct list_node n;
node np;

np = &n;
n.data = 3;
n.next = NULL;
```

Dereferencing deep structures:

```
(*((*((*n).next)).next)).data
```

vs.

```
n->next->next->data
```

6 Lecture: Wrapping Up C

Outline:

- Announcements
- Notes on Asgn2
- Thoughts on the assignment
- Pointers to Structures: More Complex
- Example: Dynamic Data Structures (linked list)
  - Dynamic Data Structures
  - Writing and recovering data: A linked list example
  - Or it could be done recursively
- Compound Data Wrapup
  - So Far
  - Unions and enumerated types
- Pointer review: It's all the same
  - Pointers to Functions: Ridiculous
  - Summing up
- C odds and ends
  - Short-circuit evaluation
  - The Ternary Operator
  - The Comma Operator
  - Variable modifiers
  - Promotion rules

6.1 Announcements

- Coming attractions:

Event	Subject	Due Date			Notes
asgn6	shell	Fri	Dec 8	23:59	

  Use your own discretion with respect to timing/due dates.
- on partner ones, only one copy
- Pre-existing stdin, stdout, stderr.
- **Array stuff we skipped last time**
- Discuss readlonglines
- C Language Quiz coming up
  - Will be similar to the questions published on the web page
  - Old exams on the web. (think 202-like material)
  - Bring questions for review Friday before the exam
- Piazza link on web page
- Headers and libraries: They're in the man page
- Skim Ch. 2.
- Assignment stats:

Assignment	Submitted	Make	Compiled	Passed all	Passed none	complaints
Lab01	60	—	59	50	0	0(!)
Asgn1	58	—	58	50	0	0(!)

- Asgn2: fw
  - Reading long strings and `realloc(3)`
  - Relationship between reading/writing, counting, finding the top k, and reading long strings: not much. Maintain abstraction.
  - If you're using a hash table, sorting after collecting them all is the approach I'd recommend. Note:
    - There's no way to know in advance which will be your k most common words: so you have to keep all the words you are given.

    - If there are n words on the input stream, of which m are unique, To check each one against your top k would require at best $O(n \lg k)$ time.
    - To do the same thing at the end with the m unique words would be $O(m \lg k)$.
      Since $m \leq n$, you can't lose and you're very likely to win since it's unlikely that there will be not a single repeat in your input stream of words.
    - You have to build your own data structures

- -O — turns on dataflow analysis, but can break gdb

6.2 Notes on Asgn2

- All code here must be your own.
- Design! This program decomposes fairly well.
- Reading long strings and `realloc(3)`
- My asgn2 is 399 lines total. (this is not a large program)
- `time(1)`
- `-O` turns on the optimizer
- Go ahead and look up a hash function, just don't appropriate someone's code. (cite it!)
- "Notes on Partnerships" — why

See example in Figure 36.

```
lagniappe% wc /usr/share/dict/words
 99171  99171 938848 /usr/share/dict/words
lagniappe%
falcon% time fw.solaris /usr/dict/words /usr/share/man/*/*
The top 10 words (out of 69900) are:
   452919 fr
   275401 the
   201833 sp
   156752 in
    88201 to
    87961 fb
    81270 pp
    78950 n
    76514 is
    75718 of
6.61u 0.53s 0:12.01 59.4%
falcon%
```

Figure 26: fw in action

`/usr/man` on unixX is approx. 110MB
`/usr/share/dict/words` is 4.8MB and contains 480k words.

6.3 Thoughts on the assignment

Chapter 6; Arrays, Structures, enum, typedef, unions?

6.4 Pointers to Structures: More Complex

More or less everything works in the same way with one shortcut.

```
struct list_node n;
node np;

np = &n;
n.data = 3;
n.next = NULL;
```

Dereferencing deep structures:

```
(*((*((*n).next)).next)).data
```

vs.

```
n->next->next->data
```

6.5 Example: Dynamic Data Structures (linked list)

In which we look at various aspects of dynamic data structures and trees that might be of use in the homework.

6.5.1 Dynamic Data Structures

When dealing with dynamic data:

- Remember to allocate memory for it
- Be sure it's enough
- Be sure it succeeded.
- When you blow it: Use the debugger to figure out where.

6.5.2 Writing and recovering data: A linked list example

These code snippets (Figures 27 through 31) show one way of writing and recovering a linked list of integers.

Fundamentally, the process consists of choosing a format that will faithfully represent your data structure and implementing it.

6.5.3 Or it could be done recursively

As in Figures 32 and 33.

```
typedef struct node_st *node;
struct node_st {
  int data;
  node next;
};
```

Figure 27: A struct for a linked list of integers

```
node new_node(int data){
  /* Allocate and initialize a new node with
   * the given data value.
   * Returns the node on success, NULL on
   * failure
   */
  node n;

  n = (node) malloc(sizeof(struct node_st));
  if ( ! n ) {
    perror("malloc");
    exit(1);
  }
  n->data = data;
  n->next = NULL;
  return n;
}
```

Figure 28: Allocating a new node

```
void write_list(FILE *outfile, node list) {
  /* Write the current node to the given stream
   */
  while( list ) {
    fprintf(outfile, "%d\n", list->data);
    list = list->next;
  }
}
```

Figure 29: A function to write a list of integers (iterative version)

```
node append_list(node list, node rest){
  /* concatenate two well–formed lists */
  node t;
  if ( list ) {
    for(t=list; t->next != NULL; t=t->next )
      /* nothing */;
    t->next = rest;
  } else {
    list = rest;
  }
  return list;
}

node read_list(FILE *infile) {
  /* Read the current node, then read the rest
   * Reads numbers from infile until either EOF or a
   * non–number is found.
   */
  int num;
  node n, list;

  list = NULL;

  while ( 1 == fscanf(infile," %d",&num) ) {
    /* a return value of 1 indicates a successful match */
    n = new_node(num);
    list = append_list(list,n);
  }

  return list;
}
```

Figure 30: A function to read a list of integers (iterative version)

```
node read_list(FILE *infile) {
  /* Read the current node, then read the rest
   * Reads numbers from infile until either EOF or a
   * non–number is found.
   */
  int num;
  node n, list;
  node *tail;       /* note: this is a double pointer! */

  list = NULL;
  tail = &list;

  while ( 1 == fscanf(infile," %d",&num) ) {
    /* a return value of 1 indicates a successful match */
    n = new_node(num);
    *tail = n;                  /* tie the new node in */
    tail = &n->next;            /* move the tail pointer along */
  }

  return list;
}
```

Figure 31: A more clever function to read a list of integers (iterative version)

```
void write_list(FILE *outfile, node list) {
  /* writes the current node, then the rest
   */
  if ( list ) {
    fprintf(outfile, "%d\n", list->data);
    write_list(outfile,list->next);
  }
}
```

Figure 32: A function to write a list of integers (recursive version)

```
node read_list(FILE *infile) {
  /* read the current node, then read the rest
   * Reads numbers from infile until either EOF or a
   * non–number is found.
   */
  int num;
  node n;

  if ( 1 == fscanf(infile," %d",&num) ) {
    /* a return value of 1 indicates a successful match */
    n = new_node(num);
    n->next = read_list(infile);
  } else {
    n = NULL;
  }

  return n;
}
```

Figure 33: A function to read a list of integers (recursive version)

6.6 Compound Data Wrapup

6.6.1 So Far

- Arrays
- Structs

6.6.2 Unions and enumerated types

There are two more types we haven't discussed: unions and enums

An enumerated (**enum**) type allows you to use symbolic names for values:

```
typedef enum {mon,tue,wed,thu,fri,sat,sun} day;
day tomorrow = fri;
```

or

```
typedef enum {false, true} boolean;
boolean b = false;
```

A **union** is a like a structure except all of the fields occupy the same space.

```
union kludge{
  int num;
  char bytes[sizeof(int)];
};
int i;
union kludge k;
k.num = 0x0A0B0C0D;
for(i=0; i<sizeof(k.num); i++)
   printf("Byte[%d]: 0x%02x\n", i, k.byte[i]);
putchar('\n');
```

6.7 Pointer review: It's all the same

Pointers are memory addresses.

- Pointers to simple data
- Pointers to compound data
- Pointers to functions? See below

6.7.1 Pointers to Functions: Ridiculous

Don't even worry about knowing about these at this point. It's bad enough to know they exist:

Declaration: int (*fname)(int);

Asignment: (assiming foo() exists) fname = foo;

Use: x = (*fname)(2)

6.7.2 Summing up

Things you know:

- Dynamic data structures are no more complicated than allocating space and remembering where you put things.
- A pointer is a variable that contains a memory address.
- Pointers are uninitialized; they do not necessarily point at anything in particular.
- Operators:

&	address of
*	dereference

- NULL is a standard invalid pointer.

- Think *types*. Pointer vs. pointee.

  Given an integer pointer, p stored at address 0x12345678 and an integer i, stored at 0xABCDABCD, containing the value 3:

```
int i=3;
int *p=&i
```

Expression:	&p	p	*p
Type:	int **	int *	int
Value:	0x1234	0xABCD	3

	Expression					
Variable	**&x**	**x**	***x**	****x**	*****x**	******x**
int x	int *	int	—	—	—	—
int *x	int **	int *	int	—	—	—
int **x	int ***	int **	int *	int	—	—
int ***x	int ****	int ***	int **	int *	int	—

- `sizeof()` is a macro that evaluates to the size of a given data structure.
- The name of an array is equivalent to the address of that array. Even though it is casually called a pointer, it is not a variable.
- The same is true of the name of a function.
- Free not that which thou didst not malloc().

6.8 C odds and ends

6.8.1 Short-circuit evaluation

C conditionals are evaluated left to right and evaluation stops as soon as the overall sense of the expression is known.

Thus, the following is safe:

```
if ( p && *p ) ...
```

6.8.2 The Ternary Operator

Useful in some situations, but never at the cost of clarity. Good usage:

```
printf("Found %d word%s.\n", num, (num==1)?"":"s");
```

6.8.3 The Comma Operator

The operands of *comma* are evaluated left to right, and the value of the overall expression is the value of the rightmost operand. (i.e., the value of x, y is y)

For use, see Figure 35.

```
void Pof2(int n) {
    /* print the first n powers of two */
    unsigned long i, num;
    for(i=0, num=1; i < n; i++, num*=2)
        printf("2^%-2d = %u\n", i, num);
}
```

Figure 34: Pof2(): Printing the first n powers of 2.

6.8.4 Variable modifiers

unsigned
const
extern
static
volatile
register
restrict

6.8.5 Promotion rules

C data types will automatically be promoted to "wider" types. Note that while magnitude is preserved, precision may not be. (e.g., `int` to `float`)

Promotion is only done as needed. Consider

```
f = 100.0 * 3/4;
g = 3/4 * 100.0;
printf("f = %f\ng = %f\n", f, g);

f = 75.000000
g = 0.000000
```

7 Lecture: Starting in with unix/Unbuffered vs. Buffered IO

Outline:

7.1 Announcements

- Coming attractions:

Event	Subject	Due Date			Notes
asgn6	shell	Fri	Dec 8	23:59	

  Use your own discretion with respect to timing/due dates.

- Comment on class leading the assignments
- C quiz next Wednesday.
  - Old exams on the web. (think 202-like material)
  - Bring questions for review Monday
- For Next Time: Know, love, and become one with Stevens Ch. 1.
- Nul-terminate your damned strings
- C quiz samples out

- Dealing with binary data. `cat -v` or `od -tx1` or others
- `time(1)` `/usr/bin/time` tells you how long a program runs: user, system, and wallclock.
- Only one late day allowed for asgn2.
- Only turn in one...
- asgn1

```
% tallac% grep Passed */COMMENT | grep out | sed 's/.*://' | sort | uniq -c | sort -rn -k 3
     72 Passed 13 out of 13 tests.
      4 Passed 12 out of 13 tests.
      1 Passed 8 out of 13 tests.
      1 Passed 1 out of 13 tests.
tallac%
```

7.2 From last time: Compound Data Wrapup

7.2.1 So Far

- Arrays
- Structs

7.2.2 Type definitions

7.2.3 Unions and enumerated types

There are two more types we haven't discussed: unions and enums

An enumerated (**enum**) type allows you to use symbolic names for values:

```
typedef enum {mon,tue,wed,thu,fri,sat,sun} day;
day tomorrow = fri;
```

or

```
typedef enum {false, true} boolean;
boolean b = false;
```

A **union** is a like a structure except all of the fields occupy the same space.

```
union kludge{
  int num;
  char bytes[sizeof(int)];
};
int i;
union kludge k;
k.num = 0x0A0B0C0D;
for(i=0; i<sizeof(k.num); i++)
   printf("Byte[%d]: 0x%02x\n", i, k.byte[i]);
putchar('\n');
```

7.3 Pointer review: It's all the same

Pointers are memory addresses.

- Pointers to simple data
- Pointers to compound data
- Pointers to functions? See below

7.3.1 Pointers to Functions: Ridiculous

Don't even worry about knowing about these at this point. It's bad enough to know they exist:

Declaration: int (*fname)(int);

Asignment: (assiming foo() exists) fname = foo;

Use: x = (*fname)(2)

7.3.2 Summing up

Things you know:

- Dynamic data structures are no more complicated than allocating space and remembering where you put things.
- A pointer is a variable that contains a memory address.
- Pointers are uninitialized; they do not necessarily point at anything in particular.
- Operators:

&	address of
*	dereference

- NULL is a standard invalid pointer.
- Think *types.* Pointer vs. pointee.
  Given an integer pointer, p stored at address 0x12345678 and an integer i, stored at 0xABCDABCD, containing the value 3:

```
int i=3;
int *p=&i
```

Expression:	&p	p	*p
Type:	int **	int *	int
Value:	0x1234	0xABCD	3

	Expression					
Variable	**&x**	**x**	***x**	****x**	*****x**	******x**
int x	int *	int	—	—	—	—
int *x	int **	int *	int	—	—	—
int **x	int ***	int **	int *	int	—	—
int ***x	int ****	int ***	int **	int *	int	—

- `sizeof()` is a macro that evaluates to the size of a given data structure.
- The name of an array is equivalent to the address of that array. Even though it is casually called a pointer, it is not a variable.
- The same is true of the name of a function.
- Free not that which thou didst not malloc().

7.4 C odds and ends

7.4.1 Short-circuit evaluation

C conditionals are evaluated left to right and evaluation stops as soon as the overall sense of the expression is known.
Thus, the following is safe:
`if ( p && *p ) ...`

7.4.2 The Ternary Operator

Useful in some situations, but never at the cost of clarity. Good usage:

```
printf("Found %d word%s.\n", num, (num==1)?"":"s");
```

7.4.3 The Comma Operator

The operands of *comma* are evaluated left to right, and the value of the overall expression is the value of the rightmost operand. (i.e., the value of x, y is y)
For use, see Figure 35.

```
void Pof2(int n) {
    /* print the first n powers of two */
    unsigned long i, num;
    for(i=0, num=1; i < n; i++, num*=2)
        printf("2^%-2d = %u\n", i, num);
}
```

Figure 35: Pof2(): Printing the first n powers of 2.

7.4.4 Variable modifiers

unsigned
const
extern
static
volatile
register
restrict

7.4.5 Promotion rules

C data types will automatically be promoted to "wider" types. Note that while magnitude is preserved, precision may not be. (e.g., `int` to `float`)

Promotion is only done as needed. Consider

```
f = 100.0 * 3/4;
g = 3/4 * 100.0;
printf("f = %f\ng = %f\n", f, g);

f = 75.000000
g = 0.000000
```

7.5 Change of plan for today

- C Preprocessor
- Predefined macros and conditional compilation
- Macros with paramters
- Uniq
- Unix to Friday

7.6 Notes on Asgn2

- All code here must be your own.
- Design! This program decomposes fairly well.
- Reading long strings and `realloc(3)`
- My asgn2 is 569 lines total. (this is not a large program)
- `time(1)`
- `-O` turns on the optimizer
- Go ahead and look up a hash function, just don't appropriate someone's code. (cite it!)
- "Notes on Partnerships" — why

See example in Figure 36.

```
lagniappe% wc /usr/share/dict/words
 99171  99171 938848 /usr/share/dict/words
lagniappe%
falcon% time fw.solaris /usr/dict/words /usr/share/man/*/*
The top 10 words (out of 69900) are:
   452919 fr
   275401 the
   201833 sp
   156752 in
    88201 to
    87961 fb
    81270 pp
    78950 n
    76514 is
    75718 of
6.61u 0.53s 0:12.01 59.4%
falcon%
```

Figure 36: fw in action

`/usr/man` on unixX is approx. 110MB
`/usr/share/dict/words` is 4.8MB and contains 480k words.

7.7 Separate Compilation and the C Preprocessor

7.7.1 The C-Preprocessor

- `#include "file.h"`
- `#include <file.h>`

- `#ifdef ...#endif`
- `#ifndef ...#endif`
- null-macros: `#define BLAH`
- simple-macros: `#define BLAH OTHERTHING`
  Some of these are predifined and very useful. For example

  gcc `__FUNCTION__`, `__FILE__`, `__LINE__`, `__GNUC__`,

  c++ `__cplusplus`

  architecture-dependent Things like `__sun`, `sparc`, `unix`, etc.
  See Figure 37.
- real-macros: `#define sqr(c) ((c)*(c))`
  be very careful with parenthesization and side-effects

```
falcon% touch foo.c
falcon% gcc -dM -E foo.c
#define __GCC_NEW_VARARGS__ 1
#define __sparc 1
#define __svr4__ 1
#define __GNUC_MINOR__ 95
#define __sun 1
#define sparc 1
#define __sun__ 1
#define __unix 1
#define __unix__ 1
#define __SVR4 1
#define sun 1
#define __GNUC__ 2
#define __sparc__ 1
#define unix 1
falcon%
```

(a) SunOS on a Sparc

```
pnico% touch foo.c
pnico% gcc -E -dM foo.c
#define __USER_LABEL_PREFIX__
#define __SIZE_TYPE__ unsigned int
#define __PTRDIFF_TYPE__ int
#define __HAVE_BUILTIN_SETJMP__ 1
#define __i386 1
#define __GNUC_PATCHLEVEL__ 0
#define __ELF__ 1
#define __WCHAR_TYPE__ long int
#define __GNUC_MINOR__ 96
#define __WINT_TYPE__ unsigned int
#define __tune_i386__ 1
#define __unix 1
#define unix 1
#define __REGISTER_PREFIX__
#define __linux 1
#define __GNUC__ 2
#define i386 1
#define __linux__ 1
#define __VERSION__ "2.96 20000731 (Red Hat Linux 7.0)"
#define __i386__ 1
#define linux 1
#define __unix__ 1
pnico%
```

(b) Linux on an x86

Figure 37: Pre-defined macros on different platforms

7.8 Separate compilation

(extern data)

- Compilation
- linkage
- libraries
- Names

  extern name is known, but not defined here.

  static name is defined here and only here. (file scope)

  static allocated in the data segment, not the stack. (Link it now.)

7.9 Final Observation

This has been a whirlwind presentation of a complex language. I can't teach you C. Practice is required.

7.10 Tools of last Week: grep and truss

grep locate strings in a file. from ed: g/re/p

Example:

```
% grep -il "STDIN_FILENO" /usr/include/*.h
/usr/include/unistd.h
%  grep -il
```

strace (linux), truss (solaris) watch system calls go by: These programs show you all system calls made along with their arguments and

7.11 Unix Overview

When discussing an operating system, everything depends on everything else. Today we want to talk about a process's view of its world in terms of various services an OS provides.

A note on the examples: They use code contained in Appendix B.

`http://www.kohala.com/start/apue.html`

A process's view of the world:

- Identity (uid, gid)
- Filesystem (storage, organization, ownership, access)
- IO (What good is a filesystem if you can't use it)
- Programs and Processes
- Interprocess Communication: Signals

7.12 Identity Issues: logging in

Before doing anything else on a unix machine, you need to work out who you are. (User and group identities)

Identity consists of uid and gid

Identity is controlled by several system files:

- `/etc/passwd` — this maps login name to uid (and stores passwd)
- `/etc/group` — primary and supplemental groups
- shadow passwds: `/etc/shadow`
- yellowpages (or other directory schemes, LDAP, etc.): shared info across machines

A password line:

```
csc357:*:1659:350:csc357:/home/cscstd/abcd/csc357:/bin/tcsh
```

This is "login:passwd:uid:gid:comment:home dir:shell"

How the password algorithm works: Your typed password and the salt (the first two characters) are combined via a cryptograhic hash to produce 13 printable characters in "[a–zA–Z0–9./]".

```
#include <unistd.h>

char *crypt(const char *key, const char *salt);
```

Traditionally a variation on DES, now likely to be something else..

The salt perturbs the dictionary 4096 different ways.

Your uid is your identity. The password file is the only place your login name appears in the system.

This exposure makes them vulnerable.

7.12.1 Looking at system files

real vs. shadow or yp

shadow/yp on hornet and drseuss

7.13 Files and Directories

The entire system is connected through the filesystem. (discussion of limits.h)

```
NAME_MAX:
Solaris: 512
linux: 256
```

7.13.1 Directories

A directory is a file containing a list of:

- name
- attributes (not actually in the directory)
- where (inode)

discribed in "dirent.h":

```
struct dirent
{
    long d_ino;                 /* inode number */
    off_t d_off;                /* offset to this dirent */
    unsigned short d_reclen;    /* length of this d_name */
    char d_name [NAME_MAX+1];   /* file name (null-terminated) */
}
```

Directory entries are unordered.
Directories are protected.

7.13.2 Directory Manipulation

Manipulation functions `opendir(3), readdir(3), closedir(3), rewinddir(3)`

```
#include <sys/types.h>
#include <dirent.h>
DIR *opendir(const char *name);
int closedir(DIR *dir);
struct dirent *readdir(DIR *dirp);
void rewinddir(DIR *dir);
```

As part of its environment, every process has a:

Home Directory defined in /etc/passwd

Current Working Directory starts at home and follows chdir()

7.14 System Calls

Before we talk about IO services...

Look like C functions, but are direct requests for OS services.

A system call is an entry into the kernel.

Linux: (RH7.0) 222
Solaris: 253
Minix: 53

7.15 IO Services

What good is a filesystem if you can't use it?

7.15.1 Unbuffered IO: unix

IO is done in terms of file descriptors: small non-negative integers.

By convention, shells open `STDIN_FILENO`, `STDOUT_FILENO`, and `STDERR_FILENO`.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
```

`open(2), read(2), write(2), close(2)`

```
int open(const char *pathname, int flags,  mode_t mode);
ssize_t read(int fd, void *buf, size_t count);
ssize_t write(int fd, const void *buf, size_t count);
int close(int fd);
```

`O_RDONLY, O_WRONLY or O_RDWR`

Return the number of bytes read, or -1 on error. 0 indicates there's nothing more to read (EOF).

7.15.2 Buffered IO: stdio (C)

Hides implementation details.

Works on *streams* (FILE*).

`fopen(3), fclose(3), getchar(3), putchar(3)`

String oriented: `fgets(3), fputs(3), printf(3), scanf(3)`

Defines: `stdin`, `stdout`, `stderr`.

7.16 Programs and Processes

7.16.1 Programs

A program is an executable file. (Aside on permissions(chmod)?)

Once a program is executed, it becomes a process.

`fork()` and `exec()`

7.16.2 Processes

processes share the identity of their creator:

getpid(), getpgid()

Creating one:

fork(), exec(), wait() (waitpid())

A process doesn't really terminate until it's waited for. (Init's job in this.)

7.16.3 Flavors of ps

ps -aux vs. ps -edf

Berkeley-style ps -aux

-a all processes with a tty
x processes without a controlling tty
u display user-oriented format
g everything, even "uninteresting" processes

SYSV-style ps -eadf

-e List information about every process now running.
-f Generate a full listing.

Also useful to know: "-u logname" list processes belonging to that login name.

7.16.4 Job Control

Starting, stopping and whatnotting jobs

7.17 Interprocess Communication: Signals

kill -l

SIGINT
SIGSTOP
SIGCONT
SIGTERM (15)
SIGKILL (9)
SIGCHLD

Possible actions:

- default
- ignore
- catch

8 Lecture: Programming Demonstration: uniq

Outline:

Announcements
Programming Review: uniq
Sec01's uniq
Sec03's uniq

8.1 Announcements

- Coming attractions:

Event	Subject	Due Date			Notes
asgn6	shell	Fri	Dec 8	23:59	

  Use your own discretion with respect to timing/due dates.

- Asgn2:
  - Test scripts online:
    - `~pn-cs357/demos/tryAsgn2`
  - Only one late day allowed for asgn2.
  - `sizeof(char)` is 1
  - gprof
  - `getopt(3)` — there's a learning curve, but it's worth it.
- INTRO TO LAB03 (maybe)
- The purpose of an exercise! From here on out not a single scrap of code that is not your own.
- Thoughts brought on by uniq:
  - memory is uninitialized
  - Remember to free() things!
  - No reason to copy to your before line, move the pointers

8.2 Programming Review: uniq

Rather than publishing solutions, let's look at this problem.
See Figures 38, 40, 41 and 42.

char *fgets(char *s, int size, FILE *stream); reads in at most one less than size characters from stream and stores them into the buffer pointed to by `s`. Reading stops after an EOF or a newline. If a newline is read, it is stored into the buffer. A `'\0'` is stored after the last character in the buffer.

return s on success, and NULL on error or when end of file occurs while no characters have been read.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "readlongline.h"

int main(int argc, char *argv[]){
    /* read lines from stdin until there are no more lines.  For each line,
     * compare it to the previous line.  If they are the different, print
     * the previous line.  If the same, discard the previous line.
     */
    char *last, *next;

    last = readlongline(stdin);     /* read an initial line */

    /* now, keep reading lines until there are no more lines */
    while ( (NULL != last) && (NULL != (next=readlongline(stdin)))) {
        if ( strcmp(last, next) ) { /* print the old line if different */
            fputs(last, stdout);
        }
        free(last);             /* we're done with last now */
        last = next;
    }

    if ( last )                 /* print the last line if there is one */
        fputs(last, stdout);

    return 0;
}
```

Figure 38: A main program that uses `readlongline()`

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "readlongline.h"

int main(int argc, char *argv[]){
    /* read lines from stdin until there are no more lines.  For each line,
     * compare it to the previous line.  If they are the different, print
     * the previous line.  If the same, discard the previous line.
     */
    char *last, *next;

    last = readlongline(stdin);
    if ( last )
      fputs(last, stdout);

    while ( (next = readlongline(stdin)) ) {
      if ( !strcmp(last, next) ) {
        free(next);             /* they're the same, drop it    */
      } else {                  /* they're different, write it. */
        fputs(next, stdout);
        free(last);             /* remember to clean up the old one. */
        last = next;
      }
    }

    return 0;
}
```

Figure 39: Another way of going about it.

```
#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>
#include <string.h>

#define CHUNK 80

#define DEBUG

extern char *readlongline(FILE *where) {
  /* Read a string from given stream.  Returns the string, or NULL if
   * EOF is encoutered.  Approach: allocate a buffer of size CHUNK,
   * and as the string grows expand the buffer as necessary.
   * returns NULL on EOF;
   */
    char *buff;
    int sofar, len;

    /* get an initial buffer, and make it a well–formed string */
    if (NULL == (buff = (char *) malloc(CHUNK))) {
        perror("malloc");
        exit(−1);
    }
    buff[0] = '\0';
    sofar = 0;                  /*  we don't have anything yet */

    /* now, read the string, expanding as necessary.  Loop until
     * we either hit a newline of EOF
     */
    while (fgets(buff + sofar, CHUNK, where)) {
        len = strlen(buff + sofar);

        /* now, we either have a whole line or not.  Check. */
        sofar += len;           /* add in the new part of the string */
        if ( buff[sofar−1] == '\n' )
            break;              /* it's a newline, so we're done. */

        /* if we got here, it's not a newline, so we're both
         * not done and out of buffer.  Reallocate and go 'round again.
         */
        buff = (char*)realloc(buff, sofar+CHUNK);
        if ( NULL == buff ) {   /* realloc failed.  */
            perror("realloc");
            exit(2);
        }
    }

    /* Now we have the whole string, but we might have allocated too
     * much memory.  If it's empty we hit EOF, free it and return
     * NULL.  If not, trim it down to size.
     */

    if ( sofar == 0  ) {        /* EOF */
        free(buff);
        buff = NULL;
    } else {                    /* trim to size */
        buff = realloc(buff, sofar + 1);
        if ( NULL == buff ) {   /* realloc failed.  */
            perror("realloc");
            exit(2);
        }
    }

    return buff;
}
```

Figure 40: A function that reads a long line

```
#ifndef READLONGLINEH
#define READLONGLINEH

#include <stdio.h>
extern char *readlongline(FILE *where);
#endif
```

Figure 41: The header for `readlongline()`

```
MAIN=uniq

CC = gcc

CFLAGS = -g -Wall

OBJS = main.o readlongline.o

$(MAIN): $(OBJS)
	$(CC)  -o $(MAIN) $(OBJS)

readlongline.o: readlongline.c readlongline.h
	$(CC) $(CFLAGS) -c readlongline.c

main.o: main.c readlongline.h
	$(CC) $(CFLAGS) -c main.c

clean:
	rm -f $(OBJS) core* *~
```

Figure 42: And a makefile for it

8.3 Sec01's uniq

This is the uniq that we developed in sec01

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>

#include "rll.h"

int main(int argc, char *argv[]) {
  char *old, *new;

  /* read initial line */
  old = rll(stdin);

  if ( old ) {
    while ( (new=rll(stdin)) ) {
      if ( strcmp(old,new) ) {  /* different */
        puts(old);
        /*        free(old); */
        old=new;
      } else {                  /* same */
        free(new);
      }
    }
  }
  if ( old ) {
    puts(old);
    free(old);
  }
  return 0;
}
```

Figure 43: sec01 main program that uses `rll()`

```
#ifndef RLLH
#define RLLH

char *rll(FILE *in);          /* read a line and return a pointer to it.
                              * return NULL on EOF
                              */

#endif
```

Figure 44: sec01: The header for `rll()`

```
#ifndef RLLH
#define RLLH

char *rll(FILE *in);          /* read a line and return a pointer to it.
                              * return NULL on EOF
                              */

#endif
```

Figure 45: sec01: `rll()`

```
CC = gcc

CFLAGS = -Wall -ansi -g -pedantic

uniq: rll.o uniq.o
	$(CC) $(CFLAGS) −o uniq rll.o uniq.o

rll.o: rll.c rll.h
	$(CC) $(CFLAGS) −c rll.c

uniq.o: uniq.c rll.h
	$(CC) $(CFLAGS) −c uniq.c

test: uniq
	/home/pnico/Class/cpe357/now/Asgn/Handin/lib/lab02/testuniq /home/pnico/Class/cpe357/now/Asgn/Handi
```

Figure 46: sec01: makefile for it

8.4 Sec03's uniq

This is the uniq that we developed in sec03

```
#include<stdio.h>
#include<stdlib.h>
#include <string.h>
#include "rll.h"

int main(int argc, char *argv[]) {
  char *old, *new;

  old = rll(stdin);

  if ( old ) {
    while ( (new=rll(stdin)) ) {
      if ( strcmp(old,new) ) {  /* different */
        puts(old);              /* print string w/newline */
        free(old);
        old=new;
      } else {                  /* same */
        free(new);
      }
    }
    puts(old);
  }

  return 0;
}
```

Figure 47: sec03 main program that uses `rll()`

```
#ifndef RLLH
#define RLLH

#include <stdio.h>

char *rll(FILE *in);          /* read a line of arbitrary size.  Return
                              * NULL on EOF
                              */
#endif
```

Figure 48: sec03: The header for `rll()`

```
#ifndef RLLH
#define RLLH

#include <stdio.h>

char *rll(FILE *in);          /* read a line of arbitrary size.  Return
                              * NULL on EOF
                              */
#endif
```

Figure 49: sec03: `rll()`

```
CC = gcc

CFLAGS = -Wall -ansi -pedantic -g

uniq: uniq.o rll.o
	$(CC) -o uniq $(CFLAGS) uniq.o rll.o

uniq.o: uniq.c rll.h
	$(CC) $(CFLAGS) -c uniq.c

rll.o: rll.c rll.h
	$(CC) $(CFLAGS) -c rll.c

test: uniq
	/home/pnico/Class/cpe357/now/Asgn/Handin/lib/lab02/testuniq /home/pnico/Class/cpe357/now/Asgn/Handi
```

Figure 50: sec03: makefile for it

9 Lecture: Unbuffered IO

Outline:

Announcements
Unix Overview
Identity Issues: logging in
    Looking at system files
From Last, Last, Last Time: Unix Overview
Files and Directories
    Directories
    Directory Manipulation
System Calls
From last time: Files and the filesystem
Basic File IO
    open(2)
    creat(2)
    close(2)
    read(2)
    write(2)
Performance: Buffered vs. Unbuffered
Review: Unbuffered IO
Onwards: lseek(2)
Next Time
If there's time: Lab03/Asgn3
    The assignment
From Email: Huffman
    Huffman Codes
    Reminder: Setting and clearing bits

9.1 Announcements

- Coming attractions:

Event	Subject	Due Date			Notes
asgn6	shell	Fri	Dec 8	23:59	

  Use your own discretion with respect to timing/due dates.
- getline is verboten
- Gradesheet snapshop
- Test, test, test. And Test!
- Reminder about the potential common final
- Assignments are out
- Enough rope to hang yourselves..
- qsort demo?
- Things to talk about
  - qsort(3)
  - pointers and memory
    * Pointers need to point to something to be useful
    * This does not mean you *must* call `malloc(3)`
    * Draw pictures as needed

9.2 qsort

```
#include <stdlib.h>

void qsort(void *base, size_t nmemb, size_t size,
           int (*compar)(const void *, const void *));
```

9.3 Thoughts on debugging technique

Slow and steady is the way...

- Build incrementally (and test at each step)
- Stress your program so faults show up early. (and test at each step)
  You want to break your program before somebody else does.
- Write defensive code: validate inputs, check return codes, etc.
- Be especially suspicious of memory manipulation:
  - Don't free things too soon.
  - Be sure to initialize things you expect to be initialized

Debug only what you wrote, not what you think you wrote

9.3.1 Programming stuff

We programmed some stuff that exist on the following pages

```
CC = gcc

CFLAGS = -Wall -ansi -g -pedantic

MAIN = baz

$(MAIN): $(MAIN).c
	$(CC) $(CFLAGS) -o $(MAIN) $(MAIN).c

test: $(MAIN) infile                                                    10
	./$(MAIN) < infile
```

```
#include<stdio.h>
#include<stdlib.h>

typedef int (*ifun)(int);

int tryme(ifun fun, int x) {
  return (*fun)(x);              /* make it blindingly obvious what we're doing */
}

int foo(int x) {                                                                  10
  return 2*x;
}

int bar(int x) {
  return -1*x;
}

int main(int argc, char *argv[]) {
  int i,num;
                                                                                  20
  for(i=1;i<argc;i++) {
    num = atoi(argv[i]);
    printf("First function:  %d\n", tryme(foo,num));
    printf("Second function:  %d\n\n", tryme(bar,num));
  }

  return 0;
}
```

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>

struct line {
  char *line;
  struct line *next;
};

#define MAX 1024                                                    10

struct line* append(struct line *list, struct line *rest) {
  struct line *tail;
  if ( !list ) {
    list = rest;
  } else {
    for(tail=list;tail->next;tail=tail->next)
      /* whee */;
    tail->next = rest;
  }                                                                 20
  return list;
}

void print_list(struct line *l) {
  for(;l;l=l->next) {
    printf("%s",l->line);
  }
}

void free_list ( struct line *l ) {                                 30
  struct line *next;
  for(;l;l=next) {
    next = l->next;
    if ( l->line )
      free(l->line);
    free(l);
  }
}

int main(int argc, char *argv[]) {                                  40
  char buf[MAX];
  struct line *list,*new;

  list = NULL;
  while ( fgets(buf,MAX,stdin) ) {
    new = malloc(sizeof(struct line));
    if ( !new ) {
      perror("malloc");
      exit(EXIT_FAILURE);
    }                                                               50
    new->line = malloc( strlen(buf) + 1 );
    if ( ! new-> line ) {
      perror("malloc");
      exit(EXIT_FAILURE);
    }
    strcpy(new->line, buf);
    new->next = NULL;
    list = append(list,new);
  }
                                                                    60
  /* print the resutlt */
  print_list(list);

  free_list(list);

  return 0;
}
```

```
#include<stdio.h>
#include<stdlib.h>

#define SIZE 15

#ifdef DONTLOOKHERE
int x;
int *xp;
void *foo(int);
void (*bar)(int);
int (*compar)(const void *, const void *)
#endif

int compare(const void *ap, const void *bp) {
  int a, b;
  a = *(int *) ap;
  b = *(int *) bp;

  return b-a;
}

void print_nums(int A[], int size) {
  int i;
  for(i=0;i<size;i++)
    printf("A[%02d] = %d\n",i,A[i]);
  putchar('\n');
}

int main(int argc, char *argv[]) {
  int A[SIZE],i;

  /* initialize array */
  for(i=0;i<SIZE;i++)
    A[i] = rand() % SIZE;

  /* print 'em */
  print_nums(A,SIZE);

  /* sort 'em */
  qsort(A,SIZE,sizeof(int),compare);

  /* print 'em */
  print_nums(A,SIZE);

  return 0;
}
```

10 Lecture: Unbuffered IO, cont.

Outline:

Announcements
Unix Overview
Identity Issues: logging in
    Looking at system files
From Last, Last, Last Time: Unix Overview
Files and Directories
    Directories
    Directory Manipulation
System Calls
From last time: Files and the filesystem
Basic File IO
    open(2)
    creat(2)
    close(2)
    read(2)
    write(2)
Performance: Buffered vs. Unbuffered
Review: Unbuffered IO
Onwards: lseek(2)
Next Time
If there's time: Lab03/Asgn3
    The assignment
From Email: Huffman
    Huffman Codes
    Reminder: Setting and clearing bits

10.1 Announcements

- Coming attractions:

Event	Subject	Due Date			Notes
asgn6	shell	Fri	Dec 8	23:59	

  Use your own discretion with respect to timing/due dates.
- Back from the dead. Catching up:
  - We have 18 classes left. We can do this. I counted; It fits.
  - Weights in Gradesheet Snapshot are currently meaningless
  - No asgn2 (I'll post it if you want to play)
  - No C Quiz (old questions are on the web if you want to challenge yourself)
  - Do we need to talk more about Make or are you good?
  - Lab02 and asgn1 will close Monday. It's time to move on
  - Other dates will happen soon (famous last words)
- The "Gradesheet snapshot".
- gprof(1)
- Assignments: Hours spent are irrelevant.
  - Effective hours
  - Decomposition
  - Incremental development
  - Incremental testing
  - avoid the Death March mentality
- I gave you test cases. It seems almost nobody ran them. This would have solved UI problems.
- qsort() demo
- there's stuff in the notes that's not always in class

10.2 Unix Overview

When discussing an operating system, everything depends on everything else. Today we want to talk about a process's view of its world in terms of various services an OS provides.

A note on the examples: They use code contained in Appendix B.

`http://www.kohala.com/start/apue.html`

A process's view of the world:

- Identity (uid, gid)
- Filesystem (storage, organization, ownership, access)
- IO (What good is a filesystem if you can't use it)
- Programs and Processes
- Interprocess Communication: Signals

10.3 Identity Issues: logging in

Before doing anything else on a unix machine, you need to work out who you are. (User and group identities)

Identity consists of uid and gid

Identity is controlled by several system files:

- `/etc/passwd` — this maps login name to uid (and stores passwd)
- `/etc/group` — primary and supplemental groups
- shadow passwds: `/etc/shadow`
- yellowpages (or other directory schemes, LDAP, etc.): shared info across machines

A password line:

```
csc357:*:1659:350:csc357:/home/cscstd/abcd/csc357:/bin/tcsh
```

This is "login:passwd:uid:gid:comment:home dir:shell"

How the password algorithm works: Your typed password and the salt (the first two characters) are combined via a cryptograhic hash to produce 13 printable characters in "[a–zA–Z0–9./]".

```
#include <unistd.h>

char *crypt(const char *key, const char *salt);
```

Traditionally a variation on DES, now likely to be something else..

The salt perturbs the dictionary 4096 different ways.

Your uid is your identity. The password file is the only place your login name appears in the system.

This exposure makes them vulnerable.

10.3.1 Looking at system files

real vs. shadow or yp

shadow/yp on hornet and drseuss

10.4 Files and Directories

The entire system is connected through the filesystem. (discussion of limits.h)

```
NAME_MAX:
Solaris: 512
linux: 256
```

10.4.1 Directories

A directory is a file containing a list of:

- name
- attributes (not actually in the directory)
- where (inode)

discribed in "dirent.h":

```
struct dirent
{
    long d_ino;                 /* inode number */
    off_t d_off;                /* offset to this dirent */
    unsigned short d_reclen;    /* length of this d_name */
    char d_name [NAME_MAX+1];   /* file name (null-terminated) */
}
```

Directory entries are unordered.
Directories are protected.

10.4.2 Directory Manipulation

Manipulation functions `opendir(3), readdir(3), closedir(3), rewinddir(3)`

```
#include <sys/types.h>
#include <dirent.h>
DIR *opendir(const char *name);
int closedir(DIR *dir);
struct dirent *readdir(DIR *dirp);
void rewinddir(DIR *dir);
```

As part of its environment, every process has a:

Home Directory defined in /etc/passwd

Current Working Directory starts at home and follows chdir()

10.5 System Calls

Before we talk about IO services...

Look like C functions, but are direct requests for OS services.

A system call is an entry into the kernel.

Linux: (RH7.0) 222
Solaris: 253
Minix: 53

10.6 From last time: Files and the filesystem

- A file is a collection of stuff
- Files go into the filesystem, managed by Directories

10.7 Basic File IO

The basic file IO functions can be handled by the following system calls:

```
int open(const char *pathname, int flags, mode_t mode);
ssize_t read(int fd, void *buf, size_t count);
ssize_t write(int fd, const void *buf, size_t count);
int close(int fd);
off_t lseek(int fildes, off_t offset, int whence);
```

10.7.1 open(2)

```
       #include <sys/types.h>
       #include <sys/stat.h>
       #include <fcntl.h>

       int open(const char *pathname, int flags);
       int open(const char *pathname, int flags, mode_t mode);
       int creat(const char *pathname, mode_t mode);

       flags is one of O_RDONLY, O_WRONLY or O_RDWR which request
       opening  the  file  read-only,  write-only  or read/write,
       respectively.

       O_CREAT
              If the file does not exist it will be created.

       O_TRUNC
              If the file already exists it will be truncated.

       O_APPEND
              The  file  is  opened  in  append mode. [ Before each
write the pointer is moved to the end ]
```

Symbolic names are supplied. 0600, e.g. is `S_IRUSR | S_IWUSR`

S_IRWXU	S_IWGRP
S_IRUSR	S_IXGRP
S_IWUSR	**S_IRWXO**
S_IXUSR	S_IROTH
S_IRWXG	S_IWOTH
S_IRGRP	S_IXOTH

Man `stat(2)` on linux; `stat(3HEAD)` on solaris. (Go figure)

`open(2)` returns a valid file descriptor or -1 and sets `errno`.

ENOENT	given file name doesn exist
ENOTDIR	a component is not a directory
EFAULT	bad address passed for name
EACCESS	Permission denied
ENOMEM	out of memory
ENAMETOOLONG	file name is too long

10.7.2 creat(2)

```
SYNOPSIS
       #include <sys/types.h>
       #include <sys/stat.h>
       #include <fcntl.h>

       int creat(const char *pathname, mode_t mode);
```

Equivalent to `open(path, O_WRONLY | O_CREAT | O_TRUNC, mode)`

10.7.3 close(2)

```
       #include <unistd.h>

       int close(int fd);

RETURN VALUE
       close()  returns  zero on success.  On error, -1 is returned, and errno
       is set appropriately.

ERRORS
       EBADF  fd isn't a valid open file descriptor.

       EINTR  The close() call was interrupted by a signal; see signal(7).

       EIO    An I/O error occurred.
```

10.7.4 read(2)

```
SYNOPSIS
       #include <unistd.h>

       ssize_t read(int fd, void *buf, size_t count);
```

(`ssize_t` is signed. `size_t` is unsigned)
Returns number of bytes read or -1 on error..

10.7.5 write(2)

```
SYNOPSIS
       #include <unistd.h>

       ssize_t write(int fd, const void *buf, size_t count);
```

Returns number of bytes written or -1 on error..

10.8 Performance: Buffered vs. Unbuffered

The issue of whether to use buffered IO or not often depends on performance.

```
#include <stdio.h>

void stdio_cp(char *inname, char*outname){
  FILE *infile,*outfile;
  int c;

  if ( NULL == (infile = fopen(inname,"r")) ) {
    perror(inname);
    exit(-1);
  }
  if ( NULL == (outfile = fopen(outname,"w")) ){
    perror(outname);
    exit(-1);
  }

  while((c=getc(infile))!=EOF) {
    if ( putc(c,outfile) == EOF ) {
      perror("putc");          /* putc returns EOF on error */
      exit(-1);
    };
  }

  fclose(infile);
  fclose(outfile);
}
```

Figure 51: cp based on stdio

```
#include <unistd.h>
#include<stdlib.h>
#include<string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

void unbuffered_cp(char *inname, char* outname, int size){
  int infd, outfd;
  int num;
  char *buffer;

  buffer = (char*)safe_malloc(size);

  if ( (infd = open(inname, O_RDONLY)) < 0 ) {
    perror(inname);
    exit(-1);
  }
  if ( (outfd = open(outname,
        (O_WRONLY | O_CREAT | O_TRUNC),
        (S_IRUSR | S_IWUSR))) < 0 ) {
    perror(outname);
    exit(-1);
  }

  while((num=read(infd,buffer,size)) > 0){
    if ( write(outfd,buffer,num) != num){
      perror("write");
      exit(-1);
    }
  }
  if ( num < 0 ) {
    perror("read");
    exit(-1);
  }

  if ( close(infd) ) {
    perror("close");
    exit(-1);
  }
  if ( close(outfd) ) {
    perror("close");
    exit(-1);
  }

  free(buffer);
}
```

Figure 52: cp based on unbuffered IO

```
% ll KeepArchive.tar.gz
-rw-------    1 pnico    pnico    51163086 Mar 20  2001 KeepArchive.tar.gz

% /usr/bin/time mycp KeepArchive.tar.gz outfile
37.29user 7.09system 0:50.99elapsed 87%CPU

% /usr/bin/time unbuffered KeepArchive.tar.gz outfile 1
47.64user 260.28system 5:12.10elapsed 98%CPU

% /usr/bin/time unbuffered KeepArchive.tar.gz outfile 4096
0.32user 19.27system 0:29.83elapsed 65%CPU

% /usr/bin/time unbuffered KeepArchive.tar.gz outfile 8192
0.23user 21.72system 0:28.94elapsed 75%CPU

% /usr/bin/time unbuffered KeepArchive.tar.gz outfile 65536
0.10user 23.70system 0:29.08elapsed 81%CPU

% /usr/bin/time unbuffered KeepArchive.tar.gz outfile 131072
0.04user 23.97system 0:28.80elapsed 83%CPU
```

For comparison's sake:

```
% /usr/bin/time /bin/cp KeepArchive.tar.gz outfile
0.58user 20.54system 0:30.86elapsed 68%CPU
```

10.9 Review: Unbuffered IO

```
int open(const char *pathname, int flags, mode_t mode);
ssize_t read(int fd, void *buf, size_t count);
ssize_t write(int fd, const void *buf, size_t count);
int close(int fd);
off_t lseek(int fildes, off_t offset, int whence);
```

10.10 Onwards: lseek(2)

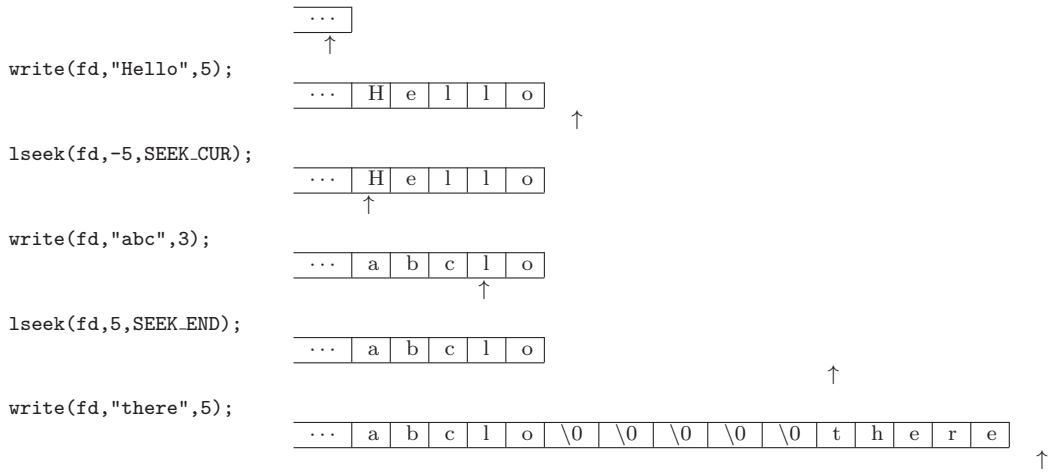
Move the file pointer (or find out where it is). This movement has no effect on the file until the next write.

`lseek(2)` introduces the concept of "holes", places in the file where nothing has ever been written. These locations are read as `'\0'`.

```
SYNOPSIS
       #include <sys/types.h>
       #include <unistd.h>

       off_t lseek(int fildes, off_t offset, int whence);

DESCRIPTION
       The  lseek  function  repositions  the  offset of the file
       descriptor fildes to the argument offset according to  the
       directive whence as follows:

       SEEK_SET
              The offset is set to offset bytes.

       SEEK_CUR
              The offset is set to its current location plus off-
              set bytes.

       SEEK_END
              The offset is set to the size of the file plus off-
              set bytes.
```

Examples:

`lseek(fd, 0, SEEK_SET)`	rewind the file
`lseek(fd, 0, SEEK_END)`	get ready to append
`pos = lseek(fd, 0, SEEK_CUR)`	get current location



10.11 Next Time

C language quiz:

Quiz next time: K&R 1–7: (Yes, actually read it.)

- Types, operators, expressions
- Control flow
- Functions (and macros)
- Pointers and Arrays
- Complex data: structures, typedef, etc.
- Memory Management
- IO (stdio)
- **not** chapter 8

My usual exam style is short-answer questions for breadth and a couple of longer ones to demonstrate depth of knowledge. I try to emphasize understanding over rote knowledge, but bear in mind that in a field such as this two are hard to separate.

- Short answer for breadth, longer answer for depth
- Don't worry about cramming the whole C library into your head. Useful things will show up at the end of the exam and/or be described.
- There are sample C quizzes linked from the web page

10.12 If there's time: Lab03/Asgn3

10.12.1 The assignment

Discussion of the assignment:

Write two programs to compress and uncompress files:

Usage:

hencode infile [outfile]

hdecode [(infile | -) [outfile]]

10.13 From Email: Huffman

```
Right, you're only padding the last byte with bits.  My concern about
endianness is this: If you shift bits into an int, the bytes will wind
up in the wrong order on a little-endian machine.  Consider:

int x = 0;
x |= 0x0A << 24;
x |= 0x0B << 16;
x |= 0x0C <<  8;
x |= 0x0D <<  0;
printf("0x%08x\n", x);

This will print 0x0A0B0C0D because that's the integer, but, if you
write this to the disk using "write(fd, &x, sizeof(int);", the bytes
on the disk will be 0x0D 0x0C 0x0B 0x0A.  This is because on a little-
endian machine the least significant end goes first.  This is why
I say build it a byte at a time so there's no confusion about
endianness.  You can build it into an array of chars and write the
whole array (or at least a hunk of it) at a time, but build it as
bytes, not integers.
```

10.13.1 Huffman Codes

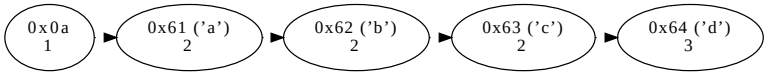
David A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the Institute of Radio Engineers*, 40(9):1098–1101, September 1952.

Building the tree

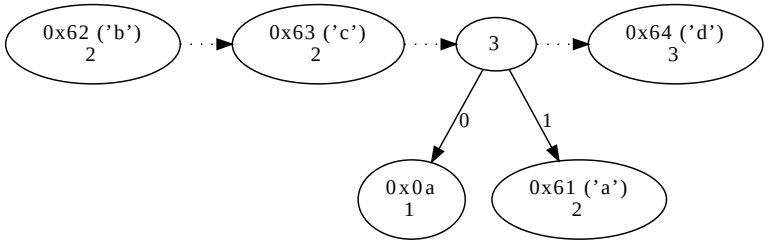
```
% cat test
aabbccddd
%
```

Byte		Count
0x0a	'\n'	1
0x61	'a'	2
0x62	'b'	2
0x63	'c'	2
0x64	'd'	3

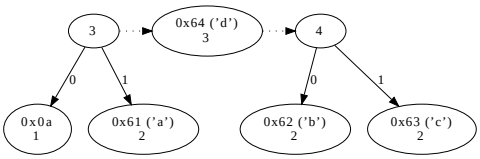
1. Building the tree: Stage 0



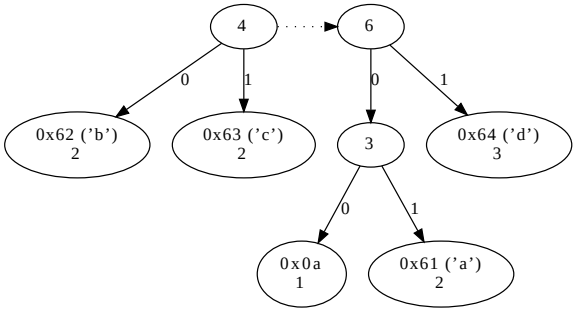
2. Building the tree: Stage 1



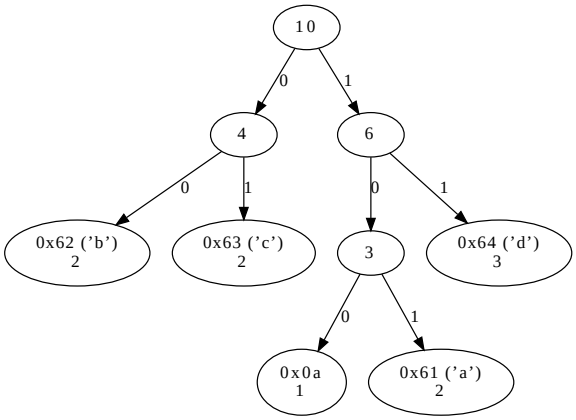
3. Building the tree: Stage 2



4. Building the tree: Stage 3



5. Building the tree: Stage 4



File Format

Num - 1	**c1**	**count of c1**	**c2**	**count of c2**	...	**c**n	**count of c**n
(uint8_t)	(uint8_t)	(uint32_t)	(uint8_t)	(uint32_t)	...	(uint8_t)	(uint32_t)
1 byte	1 byte	4 bytes	1 byte	4 bytes	...	1 byte	4 bytes

...

... encoded characters ...

Encoding the File

Byte		Code
0x0a	'\n'	100
0x61	'a'	101
0x62	'b'	00
0x63	'c'	01
0x64	'd'	11

File: "aabbccddd\n"

0000 0100 04	0000 1010 0a	0000 0000 00	0000 0000 00	0000y0000 00	0000 0001 01	011 00001 61	0000 0000 00
0000 0000 00	0000 0000 00	00000010 02	01100010 62	0000 0000 00	0000 0000 00	0000 0000 00	00000010 02
0110 0011 63	0000 0000 00	0000 0000 00	0000 0000 00	0000 0010 02	0110 0100 64	0000 0000 00	0000 0000 00
0000 0000 00	0000 0011 03						
1011 0100 b4	0001 0111 17	1111 1000 f8					

Results

```
% hencode test test.huff
% ls -l test test.huff
-rw-------  1 pnico pnico 10 Oct 17 09:55 test
-rw-------  1 pnico pnico 32 Oct 17 12:19 test.huff
% od -tx1 test.huff
0000000 04 0a 00 00 00 01 61 00 00 00 02 62 00 00 00 02
0000020 63 00 00 00 02 64 00 00 00 03 b4 17 f8
0000035
```

10.13.2 Reminder: Setting and clearing bits

It's important to remember how to set and clear bits in a bitfield while preserving the rest of the bits:
Given a word, *word,* and a bitmask *mask*

- To Set Bits:
  Bitwise-OR the word with the mask:
  $$word \vee mask$$
  Bitwise, anything ORed with 1 is set and ORed with 0 is itself:

	a	b	c	d	e
OR	1	0	1	1	0
	1	b	1	1	e

- To Clear Bits:
  Bitwise-AND the word with the inverse of mask:
  $$word \wedge \overline{mask}$$
  Bitwise, anything ANDed with 1 is itself and ANDed with 0 is 0:

	a	b	c	d	e
AND	1	0	1	1	0
	a	0	c	d	0

- In C:
  Bitwise OR: |
  Bitwise AND: &
  Bitwise NOT: ~
  Bitwise XOR: ∧

An example in Figure 54.

```
void print_bits(unsigned char c) {
  unsigned char mask;
  for(mask=0x80; mask; mask>>=1)
    if ( mask & c )
      putchar('1');
    else
      putchar('0');
}
```

Figure 53: printing the bits in a byte.

11 Lecture: C Language Quiz

Outline:
C Language Quiz

11.1 C Language Quiz

The C Quiz wasn't held this day.

12 Lecture: Filesystem Wrapup for Real

Outline:

12.1 Announcements

- Coming attractions:

Event	Subject	Due Date			Notes
asgn6	shell	Fri	Dec 8	23:59	

  Use your own discretion with respect to timing/due dates.

- Share the load Use unix{1,2,3,4,5}
- Do your own work. (No code from anywhere that is not your own.)
- BUILD YOUR PROGRAMS BEFORE SUBMITTING, and submit all of it.
- Office Hours this MW will be 3-4
- Hours spent are irrelevant.
  - Effective hours
  - Decomposition
  - Incremental development
  - Incremental testing
  - Avoid the Death March mentality
  - Make sure it compiles.
- Today
  - List insertion
  - list appending
  - qsort(3)

```
void qsort(void *base, size_t nmemb, size_t size,
        int (*compar)(const void *, const void *));
```

  - umask? — permissions not to give

12.2 Huffman Codes

12.2.1 The assignment

Discussion of the assignment:

Write two programs to compress and uncompress files:

Usage:

hencode infile [outfile]

hdecode [(infile | -) [outfile]]

12.3 From Email: Huffman

```
Right, you're only padding the last byte with bits.  My concern about
endianness is this: If you shift bits into an int, the bytes will wind
up in the wrong order on a little-endian machine.  Consider:

int x = 0;
x |= 0x0A << 24;
x |= 0x0B << 16;
x |= 0x0C <<  8;
x |= 0x0D <<  0;
printf("0x%08x\n", x);

This will print 0x0A0B0C0D because that's the integer, but, if you
write this to the disk using "write(fd, &x, sizeof(int);", the bytes
on the disk will be 0x0D 0x0C 0x0B 0x0A.  This is because on a little-
endian machine the least significant end goes first.  This is why
I say build it a byte at a time so there's no confusion about
endianness.  You can build it into an array of chars and write the
whole array (or at least a hunk of it) at a time, but build it as
bytes, not integers.
```

12.3.1 Huffman Codes

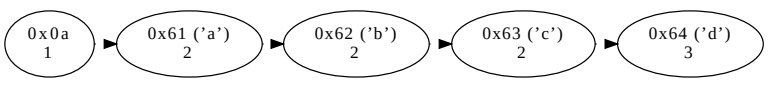
David A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the Institute of Radio Engineers*, 40(9):1098–1101, September 1952.

Building the tree

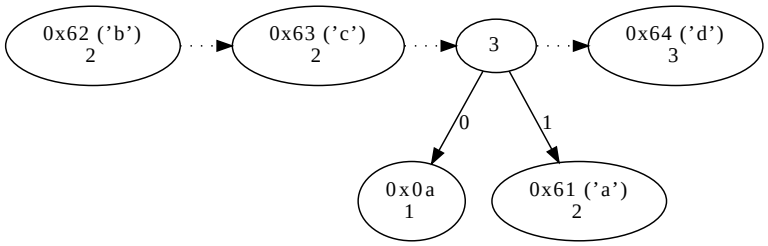
```
% cat test
aabbccddd
%
```

Byte		Count
0x0a	'\n'	1
0x61	'a'	2
0x62	'b'	2
0x63	'c'	2
0x64	'd'	3

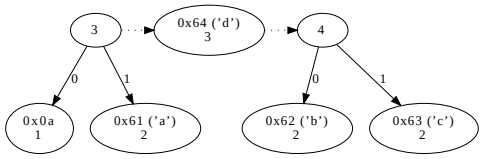
1. Building the tree: Stage 0



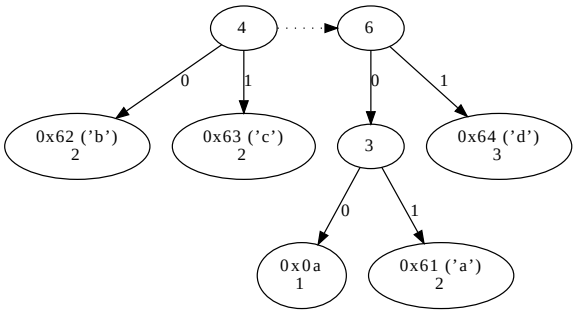
2. Building the tree: Stage 1



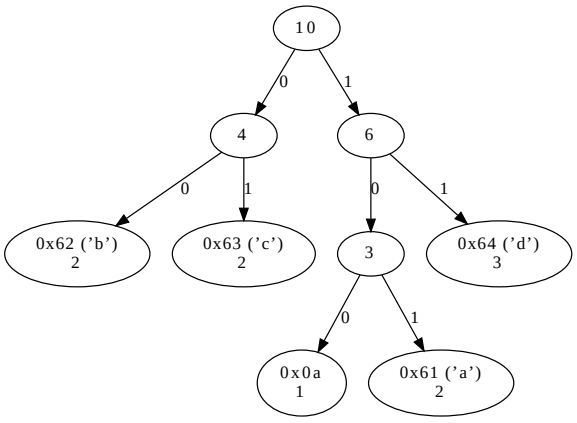
3. Building the tree: Stage 2



4. Building the tree: Stage 3



5. Building the tree: Stage 4



File Format

Num - 1 (uint8_t) 1 byte	**c1** (uint8_t) 1 byte	**count of c1** (uint32_t) 4 bytes	**c2** (uint8_t) 1 byte	**count of c2** (uint32_t) 4 bytes	... ...	**cn** (uint8_t) 1 byte	**count of cn** (uint32_t) 4 bytes

...

...encoded characters ...

Encoding the File

Byte		Code
0x0a	'\n'	100
0x61	'a'	101
0x62	'b'	00
0x63	'c'	01
0x64	'd'	11

File: "aabbccddd\n"

0000 0100	0000 1010	0000 0000	0000 0000	0000 0000	0000 0001	0110 00001	0000 0000
04	0a	00	00	00	01	61	00
0000 0000	0000 0000	00000010	01100010	0000 0000	0000 0000	0000 0000	00000010
00	00	02	62	00	00	00	02
0110 0011	0000 0000	0000 0000	0000 0000	0000 0010	0110 0100	0000 0000	0000 0000
63	00	00	00	02	64	00	00
0000 0000	0000 0011						
00	03						
1011 0100	0001 0111	1111 1000					
b4	17	f8					

Results

```
% hencode test test.huff
% ls -l test test.huff
-rw-------  1 pnico pnico 10 Oct 17 09:55 test
-rw-------  1 pnico pnico 32 Oct 17 12:19 test.huff
% od -tx1 test.huff
0000000 04 0a 00 00 00 01 61 00 00 00 02 62 00 00 00 02
0000020 63 00 00 00 02 64 00 00 00 03 b4 17 f8
0000035
```

12.3.2 Logistics: extracting the codes

- You'll have to walk the tree (remember 202)
- Remember strings are mutable
- Remember string termination
- Choose useful forms

12.3.3 Reminder: Setting and clearing bits

It's important to remember how to set and clear bits in a bitfield while preserving the rest of the bits:
Given a word, *word,* and a bitmask *mask*

- To Set Bits:
  Bitwise-OR the word with the mask:
  $$word \vee mask$$
  Bitwise, anything ORed with 1 is set and ORed with 0 is itself:

	a	b	c	d	e
OR	1	0	1	1	0
	1	b	1	1	e

- To Clear Bits:
  Bitwise-AND the word with the inverse of mask:
  $$word \wedge \overline{mask}$$
  Bitwise, anything ANDed with 1 is itself and ANDed with 0 is 0:

	a	b	c	d	e
AND	1	0	1	1	0
	a	0	c	d	0

- In C:
  Bitwise OR: |
  Bitwise AND: &
  Bitwise NOT: ~
  Bitwise XOR: ∧

An example in Figure 54.

```
void print_bits(unsigned char c) {
  unsigned char mask;
  for(mask=0x80; mask; mask>>=1)
    if ( mask & c )
      putchar('1');
    else
      putchar('0');
}
```

Figure 54: printing the bits in a byte.

12.4 From last time

A recursuve version of inserting in a sorted linked list is included in Figure 55.

12.5 Our story so far

The basic file IO functions can be handled by the following system calls:

```
int open(const char *pathname, int flags, mode_t mode);
ssize_t read(int fd, void *buf, size_t count);
ssize_t write(int fd, const void *buf, size_t count);
int close(int fd);
off_t lseek(int fildes, off_t offset, int whence);
```

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>


typedef struct node_st {
  int num;
  struct node_st *next;
} *Node;

Node new_node (int num) {
  Node new;
  new = malloc(sizeof(struct node_st));
  if ( !new ) {
    perror("malloc");
    exit(EXIT_FAILURE);
  }
  new->num = num;
  new->next = NULL;
  return new;
}

void print_list(Node list) {
  for ( ; list ; list = list->next) {
    printf("%d%c",list->num,list->next?',':'\n');
  }
}

Node insert(int num, Node list) {
  Node new;
  if ( !list || num <= list->num ) {
    new = new_node(num);
    new->next = list;
    list = new;
  } else {
    list->next = insert(num,list->next);
  }
  return list;
}

int main(int argc, char *argv[]) {
  int num;
  Node l=NULL;
  while ( 1==scanf("%d",&num) ) {
    l = insert(num,l);
    print_list(l);
  }
  return 0;
}
```

Figure 55: Recursively inserting in a list

12.6 Onwards: lseek(2)

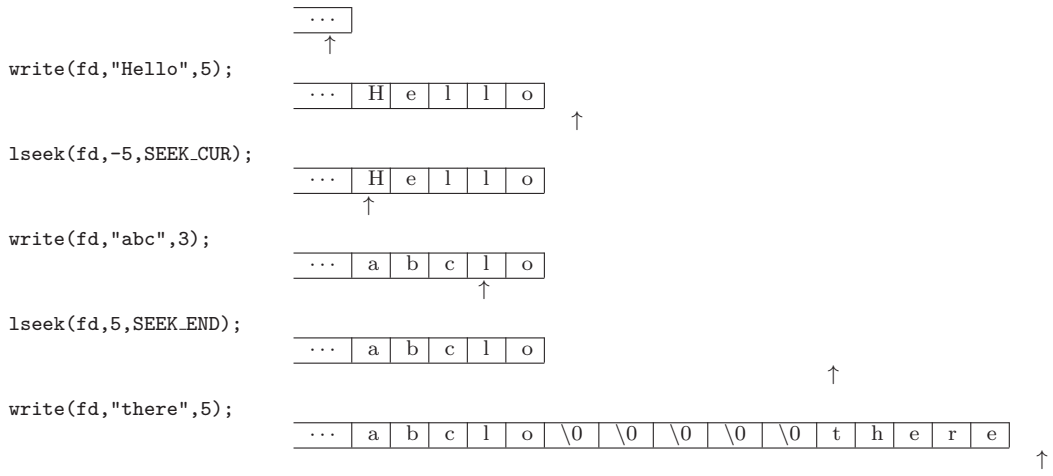
Move the file pointer (or find out where it is). This movement has no effect on the file until the next write.

`lseek(2)` introduces the concept of "holes", places in the file where nothing has ever been written. These locations are read as `'\0'`.

```
SYNOPSIS
       #include <sys/types.h>
       #include <unistd.h>

       off_t lseek(int fildes, off_t offset, int whence);

DESCRIPTION
       The  lseek  function  repositions  the  offset of the file
       descriptor fildes to the argument offset according to  the
       directive whence as follows:

       SEEK_SET
              The offset is set to offset bytes.

       SEEK_CUR
              The offset is set to its current location plus off-
              set bytes.

       SEEK_END
              The offset is set to the size of the file plus off-
              set bytes.
```

Examples:

`lseek(fd, 0, SEEK_SET)`	rewind the file
`lseek(fd, 0, SEEK_END)`	get ready to append
`pos = lseek(fd, 0, SEEK_CUR)`	get current location



12.7 Other calls: dup, dup2

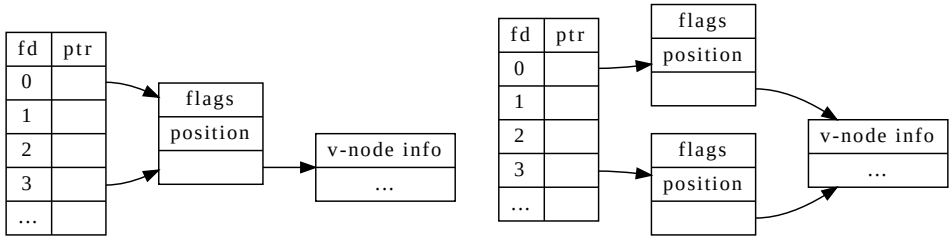
Wrapping up the system calls that affect a process' view of IO.

Here's were it gets really interesting...

12.7.1 dup() and dup2()

dup() creates a copy of an already open file descriptor. This is different from open()ing the file again, because both descriptors share the same state.

This difference can be seen in Figure 56.



(a) new fd created with dup() (b) new fd created with open()

Figure 56: Data structures for dup() vs. reopening.

```
SYNOPSIS
       #include <unistd.h>

       int dup(int oldfd);
       int dup2(int oldfd, int newfd);

DESCRIPTION
       dup and dup2 create a copy of the file descriptor oldfd.
```

```
After  successful  return  of dup or dup2, the old and new
descriptors may be used interchangeably. They share locks,
file position pointers and flags; for example, if the file
position is modified by using lseek on one of the descrip-
tors, the position is also changed for the other.

The  two  descriptors do not share the close-on-exec flag,
however.
```

Consider the two programs included as Figures 57 and 58. Each writes two strings to the same file, but `nodup.c` reopens the file, while `dup.c` uses dup() to duplicate the file descriptor.

The results of running each are shown in Figure 59.

12.7.2 Uses for dup() and dup2()

So you can make a copy of a file descriptor, so what?

There are two common uses:

- It allows you to write programs that work on either stdin or a file in the same way.
- It allows a process to rearrange stdin and stdout before performing an exec(). This is how IO redirection is done.

Consider the version of cat() in Figures 60 and 61. The main() program (Fig. 60) behaves differently in the presence of arguments, but the cat() (Fig. 61) routine never knows about it.

12.8 Onwards: Structure of the Filesystem

12.8.1 File Types

- ordinary files
- directories
- devices
  - character-special
  - block-special
- FIFO (named pipe)
- socket
- symlink

12.8.2 Implementation

Filesystem:

Boot	Superblock	i-list	directory and data blocks

The filesystem is managed via directories, i-nodes, and blocks as shown in Figure 62.

12.8.3 Structure of the filesystem

The filesystem consists of directories containing links to files and other directories. Each directory has a self-link called "." and a parent link called "..". The parent of "/" is "/".[2]

- The filesystem is a collection of files and directories
- *The Filesystem* may in fact be a collection of smaller mounted filesystems.
- All file properties (of interest) are described by the file's i-node. (even if it's not actually implemented that way)
- Directories are files that know where other files are stored:

[2]That is, dot dot of slash is slash :)

```c
/*
 * nodup.c
 *   Experiment with having two file descriptors open to the same file,
 * open()ing each independently.
 */

#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>

#define STR1 "This is string one.\n"
#define STR2 "This is string two.\n"

int main(int argc, char *argv[]){
  int fd1, fd2;
  char *fname;

  if ( argc != 2 ) {
    fprintf(stderr,"usage:  %s filename\n",argv[0]);
    exit(-1);
  }

  fname = argv[1];

  fd1 = open(fname, (O_WRONLY | O_CREAT ), (S_IRUSR | S_IWUSR) );
  if ( -1 == fd1 ) {
    perror(fname);
    exit(-1);
  }

  fd2 = open(fname, (O_WRONLY | O_CREAT ), (S_IRUSR | S_IWUSR) );
  if ( -1 == fd2 ) {
    perror(fname);
    exit(-1);
  }

  /* For the write()s and close()s, just report errors.
   * We'll be exiting soon enough.
   */

  if ( write ( fd1, STR1, strlen(STR1)) < 0 ) {
    perror("write1");
  }

  if ( write ( fd2, STR2, strlen(STR2)) < 0 ) {
    perror("write2");
  }

  if ( close(fd1) ) {
    perror(fname);
  };

  if ( close(fd2) ) {
    perror(fname);
  };

  return 0;
}
```

Figure 57: Writing two strings to the same file with open()

```c
/*
 * dup.c
 *   Experiment with having two file descriptors open to the same file,
 * open()ing the file once, then using dup().
 */

#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>

#define STR1 "This is string one.\n"
#define STR2 "This is string two.\n"

int main(int argc, char *argv[]){
  int fd1, fd2;
  char *fname;

  if ( argc != 2 ) {
    fprintf(stderr,"usage:  %s filename\n",argv[0]);
    exit(-1);
  }

  fname = argv[1];

  fd1 = open( fname, (O_WRONLY | O_CREAT ), (S_IRUSR | S_IWUSR) );
  if ( -1 == fd1 ) {
    perror(fname);
    exit(-1);
  }

  fd2 = dup(fd1);
  if ( -1 == fd2 ) {
    perror(fname);
    exit(-1);
  }

  /* For the write()s and close()s, just report errors.
   * We'll be exiting soon enough.
   */

  if ( write ( fd1, STR1, strlen(STR1) ) < 0 ) {
    perror("write1");
  }

  if ( write ( fd2, STR2, strlen(STR2) ) < 0 ) {
    perror("write2");
  }

  if ( close(fd1) ) {
    perror(fname);
  };

  if ( close(fd2) ) {
    perror(fname);
  };

  return 0;
}
```

Figure 58: Writing two strings to the same file with dup()

```
% cat testfile.orig
This is a long line that's going to be overwritten.
% cp testfile.orig testfile.dup
% cp testfile.orig testfile.nodup

% ./dup testfile.dup
% ./nodup testfile.nodup

% cat testfile.dup
This is string one.
This is string two.
verwritten.
% cat testfile.nodup
This is string two.
that's going to be overwritten.
%
```

Figure 59: Comparing the output of dup.c and nodup.c

  - Direct reading and writing of directories is restricted to the kernel.
  - All directories have at least two entries, "." and ".."
  - Directory contents consist of links.

- Files are connected to the system via links:
  - connects a name to an i-node
  - changing a link only requires access to the directory, not the file.
  - files are only removed when the last link is removed
  - hard links can only be made within a filesystem.
  - only root may make a hard link to a directory
- Symbolic links:
  - like a link, but links to a name, not an i-node.
  - More fragile than a hard link, but can work across filesystems and to directories.

12.8.4 More examples: Links, symbolic and otherwise

A rather detailed discussion of adding/removing elements from directory trees ensued here.

Consider the process below of creating a file, creating a link to it, creating a symlink, then unlinking the original. Assume the directory at inode 12 is the root of the filesystem.

```
/*
 * A simple version of cat.
 *
 *   If there are no arguments, copy stdin to stdout.  If there are args,
 * open each one in turn and copy it to stdout.
 *   If unable to open a file, print an error message and continue.
 * If the dup() fails, however, we consider that a real error and
 * terminate the program.
 *
 *  $Log: mycat.c,v $
 *  Revision 1.1  2002-02-04 10:03:36-08  pnico
 *  Initial revision
 */

#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

#define BUFFSIZE 4096
void cat();
int main(int argc, char *argv[]){
  /* if no arguments, copy stdin to stdout, othewise* copy each file in turn */
  int i,fd;

  if ( argc == 1 ) {             /* there are no args */
    cat();
  } else {                       /* foreach arg... */
    for(i=1;i<argc;i++){
      fd = open ( argv[i], O_RDONLY );
      if ( fd  < 0 ) {           /* FAILURE: print the error message */
        perror(argv[i]);
      } else {                   /* SUCCESS: dup and go...   */
        /* dup2 closes the old fd and duplicates the new one in its place.  */
        if ( dup2(fd, STDIN_FILENO) < 0 ) { /* replace stdin with infile */
          perror("dup2");
          exit(-1);
        }
        if (close(fd))           /* we're done duping */
          perror(argv[i]);       /* close it... */
        cat();                   /* copy stdin to stdout */
      }
    }
  }
  return 0;
}
```

Figure 60: The main program for cat demonstrating dup2()

```
void cat() {
  /* copy stdin to stdout until EOF
   * exits on error
   */
  char buffer[BUFFSIZE];
  int num;

  while((num=read(STDIN_FILENO,buffer,BUFFSIZE)) > 0){
    if ( write(STDOUT_FILENO ,buffer,num) != num){
      perror("write");
      exit(−1);
    }
  }
  if ( num < 0 ) {
    perror("read");
    exit(−1);
  }
}
```

Figure 61: cat() just copies stdin to stdout

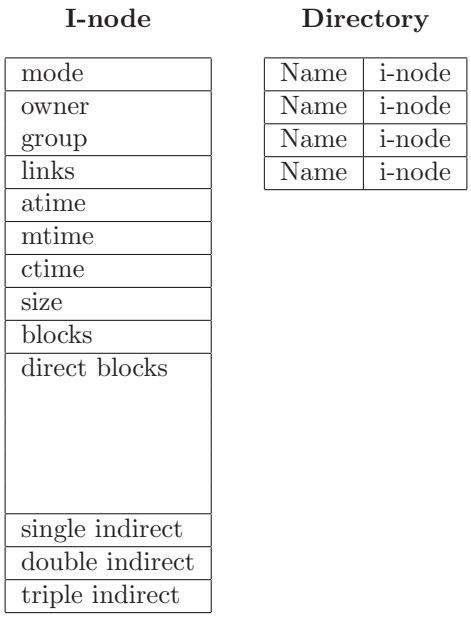
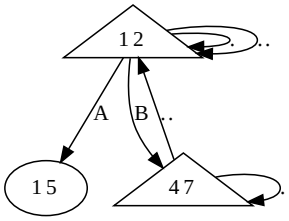
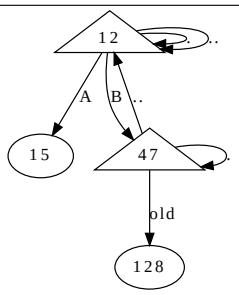
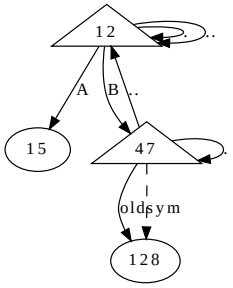
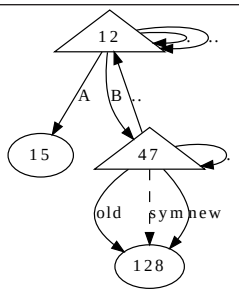
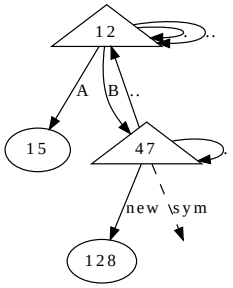


Figure 62: Structures for Directories and I-nodes

<table>
<tr><th>Command</th><th>/
(inode 12)</th><th>/B
(inode 47)</th><th>Resulting Tree
Structure</th></tr>
<tr><td>—</td><td><table><tr><th>Name</th><th>i-node</th></tr><tr><td>.</td><td>12</td></tr><tr><td>..</td><td>12</td></tr><tr><td>B</td><td>47</td></tr><tr><td>A</td><td>15</td></tr></table></td><td><table><tr><th>Name</th><th>i-node</th></tr><tr><td>.</td><td>47</td></tr><tr><td>..</td><td>12</td></tr></table></td><td> 
</td></tr>
<tr><td><code>cd /B</code>
<code>ls > old</code></td><td><table><tr><th>Name</th><th>i-node</th></tr><tr><td>.</td><td>12</td></tr><tr><td>..</td><td>12</td></tr><tr><td>B</td><td>47</td></tr><tr><td>A</td><td>15</td></tr></table></td><td><table><tr><th>Name</th><th>i-node</th></tr><tr><td>.</td><td>47</td></tr><tr><td>..</td><td>12</td></tr><tr><td>old</td><td>128</td></tr></table></td><td> 
</td></tr>
<tr><td><code>ln -s old sym</code></td><td><table><tr><th>Name</th><th>i-node</th></tr><tr><td>.</td><td>12</td></tr><tr><td>..</td><td>12</td></tr><tr><td>B</td><td>47</td></tr><tr><td>A</td><td>15</td></tr></table></td><td><table><tr><th>Name</th><th>i-node</th></tr><tr><td>.</td><td>47</td></tr><tr><td>..</td><td>12</td></tr><tr><td>old</td><td>128</td></tr><tr><td>sym</td><td>“old”</td></tr></table></td><td> 
</td></tr>
<tr><td><code>ln old new</code></td><td><table><tr><th>Name</th><th>i-node</th></tr><tr><td>.</td><td>12</td></tr><tr><td>..</td><td>12</td></tr><tr><td>B</td><td>47</td></tr><tr><td>A</td><td>15</td></tr></table></td><td><table><tr><th>Name</th><th>i-node</th></tr><tr><td>.</td><td>47</td></tr><tr><td>..</td><td>12</td></tr><tr><td>old</td><td>128</td></tr><tr><td>sym</td><td>“old”</td></tr><tr><td>new</td><td>128</td></tr></table></td><td> 
</td></tr>
<tr><td><code>unlink old</code></td><td><table><tr><th>Name</th><th>i-node</th></tr><tr><td>.</td><td>12</td></tr><tr><td>..</td><td>12</td></tr><tr><td>B</td><td>47</td></tr><tr><td>A</td><td>15</td></tr></table></td><td><table><tr><th>Name</th><th>i-node</th></tr><tr><td>.</td><td>47</td></tr><tr><td>..</td><td>12</td></tr><tr><td>—</td><td>—</td></tr><tr><td>sym</td><td>“old”</td></tr><tr><td>new</td><td>128</td></tr></table></td><td> 
</td></tr>
</table>

12.8.5 Directory Reading Functions

Manipulation functions `opendir(3), readdir(3), closedir(3), rewinddir(3)`

```
#include <sys/types.h>
#include <dirent.h>
DIR *opendir(const char *name);
int closedir(DIR *dir);
struct dirent *readdir(DIR *dirp);
void rewinddir(DIR *dir);
```

```
struct dirent {
    ino_t          d_ino;       /* inode number */
    char           d_name[256]; /* filename */
};
```

The only fields in the dirent structure that are mandated by POSIX.1 are: d_name[], of unspecified size, with at most NAME_MAX characters preceding the terminating null byte ('\0'); and (as an XSI extension)

13 Lecture: C Quiz Post-Mortem

Outline:

Announcements
The Exam
Discussion of the Exam
Selected Solutions
Aside: Software Reliability and Getting it right

13.1 Announcements

- Coming attractions:

Event	Subject	Due Date			Notes
asgn6	shell	Fri	Dec 8	23:59	

  Use your own discretion with respect to timing/due dates.

- Notes:
  - It's '\0' not '/0'
  - NULL vs. nul ('\0') not '/0'
  - `A[i] vs. *(A + i)`. The latter is weird.
- MAX CODE is 255 bits
- asgn2: 67 submissions, 51 built

13.1.1 The Exam

- fear not expressions. I.e. Don't do:

```
Atemp = s[i];
Btemp = t[i];
if ( Atemp == Btemp )
```

- No: *(s+i)
- Read the question!
- How to turn O(n) into O(n^2):

```
 for (s=string; *s; s++)
vs.
for(i=0; i < strlen(string); i++ )
vs.
 len = strlen(string)
      for(i=0; i < len; i++ )
```

  This is an argument for "`const char *s`"
- add strlen() to the soln set.
- Don't make things harder than they need to be.
- sizeof() vs. strlen()
- lseek(2)

C Quiz		
High	60.0	100.0%
Low	21.0	30.0%
Mean	39.4	70.9%
Median	38.0	71.7%
S.D.	9.6	14.3%

Grade	Cutoff	Percent
Min A−	52.5	(87.5%)
Min B−	45.0	(75.0%)
Min C−	37.5	(62.5%)
Min D−	30.0	(50.0%)

13.2 Discussion of the Exam

Final thought: This was not a particularly difficult exam. In most cases what I asked you to do was either a fairly simple example of a concept or something you'd done before.

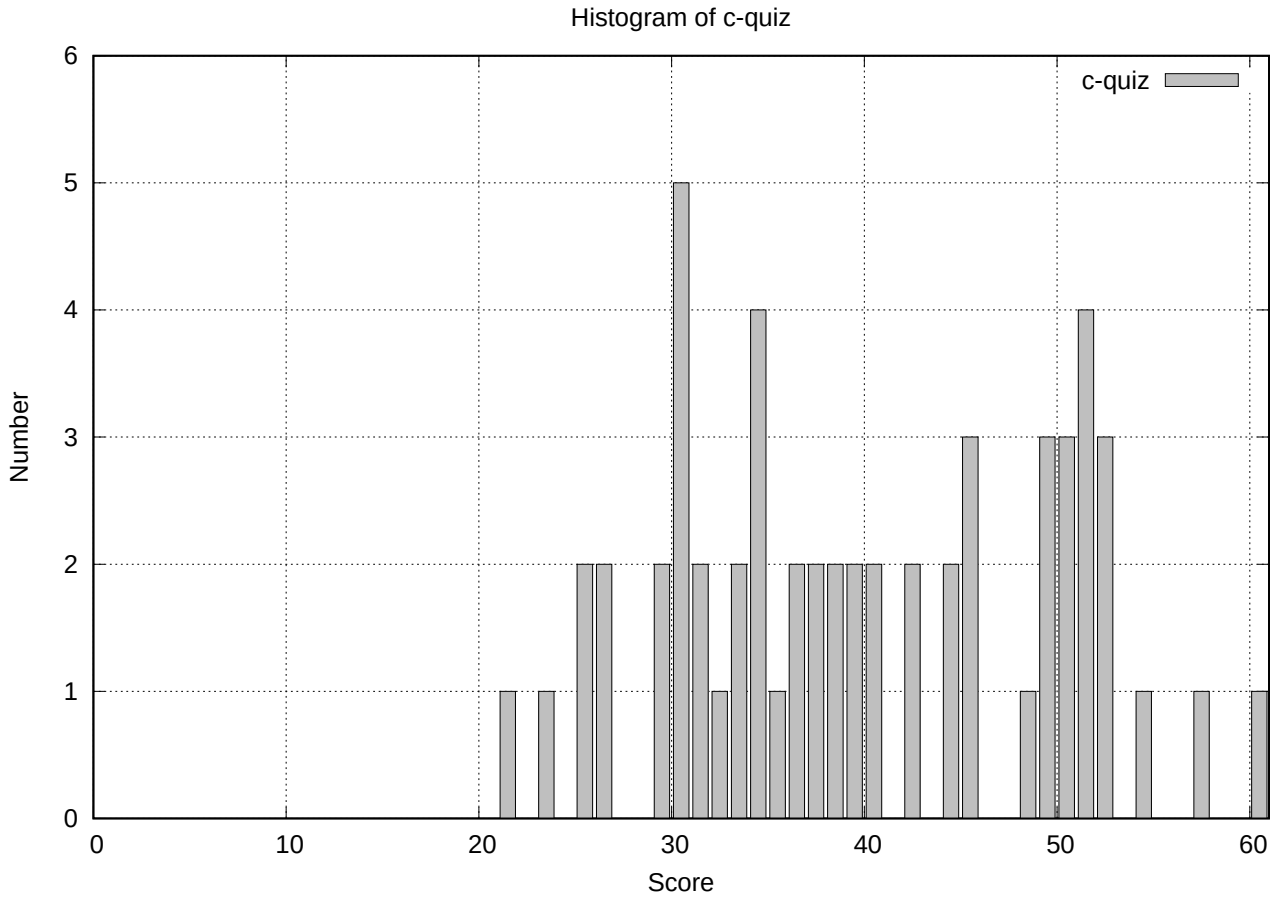


Figure 63: Histogram of scores for the quiz

13.3 Selected Solutions

We discussed some things. (Read!)

Notes:

- C99 vs. ANSI C:
  - decls and code may not be mixed

- variable-sized arrays do not exist

- Keep it as simple as possible.
- The type is "struct node_st", etc.
- **Read the exam**

Problem Notes:

1. The macro version (putc) might allow for more efficient execution because it will be expanded in place avoiding the overhead of a function call, but a macro expansion might evaluate its argument more than once making it an inappropriate choice in a situation where one of its arguments might have side-effects. Consider the effect such an expansion with multiple evaluation would have on:
   ```
   while ( *s ) { putc(s[i++],stdout) ...
   ```
2. If the given header is included by more than one source file, there would be a linker error because of multiply defined symbols.
3. mean(): overflow and types
4. strchr()
   - Remember the special behavior with nul
   - '\0' is not NULL
   - Check for NULL
5. merge_sorted
   -

13.4 Aside: Software Reliability and Getting it right

It's harder than it looks.

A class exercise in getting it right. Consider `rand()`

How do you take a value $r \in [0, \text{RAND\_MAX}]$ and map it onto $[A, B]$.

remainder	uneven
$r' = A + \left\lfloor \frac{r}{\text{RAND\_MAX}} \times (B - A) \right\rfloor$	Also uneven (only one value could give you B)
$r' = A + \left\lfloor \frac{r}{\text{RAND\_MAX}} \times (B - A + 1) \right\rfloor$	Now you can get B+1!
$r' = A + \left\lfloor \frac{r}{\text{RAND\_MAX}+1} \times (B - A + 1) \right\rfloor$	Works, but what if RAND_MAX is $2^{31} - 1$?

Double has 53 bits of precision.

14 Lecture: Filesystem wrapup for reals

Outline:

Announcements
Review: Unbuffered IO
Onwards: lseek(2)
    Structure of the filesystem
    From last time: Links, symbolic and otherwise
Manipulating the filesystem
Directories

14.1 Announcements

- Coming attractions:

Event	Subject	Due Date			Notes
asgn6	shell	Fri	Dec 8	23:59	

  Use your own discretion with respect to timing/due dates.

```
*** (To get that tree, though,
    requires exponential growth. Each subtree would have to have the
    same number of characters.  Doubling each time means the number of
    characters in the file would be on the order of 2^256.  To put
    that in context, the number of particles in the universe is
    estimated to be somewhere between 2^232 and 2^289.  You're
    probably safe.)

    In fact, since our total count is limited to (3^32-)1 * 256, any file
    we can encode can't have more than a 40-bit code.
```

- valgrind is your friend. Do not ignore it.
- Consider turning on the optimizer. It turns on the dataflow engine and can find errors.
- Garbage collected or not
- Assignment stats:

Assignment	Submitted	Make	Compiled	Passed all	Passed none	complaints
Lab01	64	—	63	63	0	0(!)
Asgn1	63	—	54	50	0	0(!)
Lab02	63	—	62	17	3	0(!)
Lab03	57	—	54	15	10	0(!)

  I published the test suites...

- documentation? Any?
- Test harness? Run it!
- Comment!
- Don't cheat

14.2 Review: Unbuffered IO

```
int open(const char *pathname, int flags, mode_t mode);
ssize_t read(int fd, void *buf, size_t count);
ssize_t write(int fd, const void *buf, size_t count);
int close(int fd);
off_t lseek(int fildes, off_t offset, int whence);
```

14.3 Onwards: lseek(2)

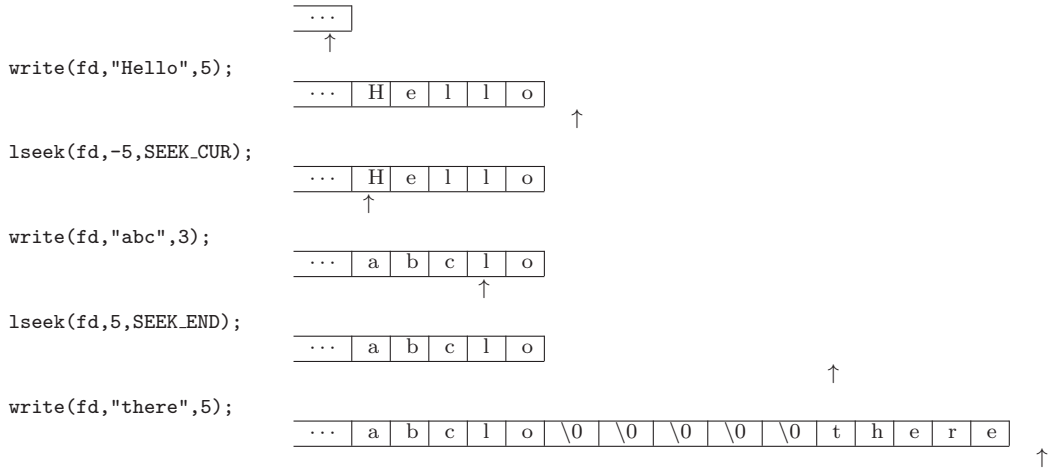
Move the file pointer (or find out where it is). This movement has no effect on the file until the next write.

`lseek(2)` introduces the concept of "holes", places in the file where nothing has ever been written. These locations are read as `'\0'`.

```
SYNOPSIS
       #include <sys/types.h>
       #include <unistd.h>

       off_t lseek(int fildes, off_t offset, int whence);

DESCRIPTION
       The  lseek  function  repositions  the  offset of the file
       descriptor fildes to the argument offset according to  the
       directive whence as follows:

       SEEK_SET
              The offset is set to offset bytes.

       SEEK_CUR
              The offset is set to its current location plus off-
              set bytes.

       SEEK_END
              The offset is set to the size of the file plus off-
              set bytes.
```

Examples:

`lseek(fd, 0, SEEK_SET)`	rewind the file
`lseek(fd, 0, SEEK_END)`	get ready to append
`pos = lseek(fd, 0, SEEK_CUR)`	get current location



14.3.1 Structure of the filesystem

The filesystem consists of directories containing links to files and other directories. Each directory has a self-link called "." and a parent link called "..". The parent of "/" is "/".[3]

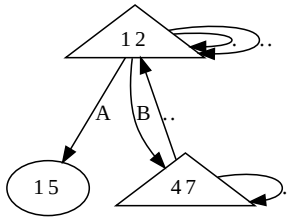
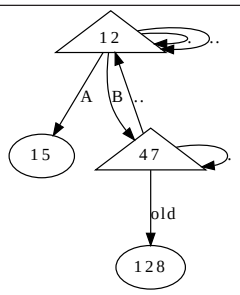
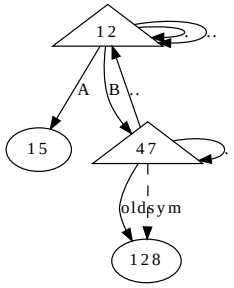
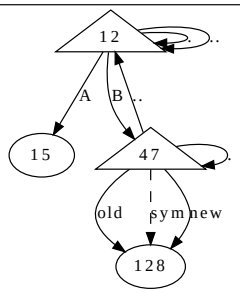
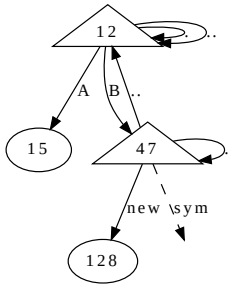
- The filesystem is a collection of files and directories
- *The Filesystem* may in fact be a collection of smaller mounted filesystems.
- All file properties (of interest) are described by the file's i-node. (even if it's not actually implemented that way)
- Directories are files that know where other files are stored:
  - Direct reading and writing of directories is restricted to the kernel.
  - All directories have at least two entries, "." and ".."
  - Directory contents consist of links.
- Files are connected to the system via links:
  - connects a name to an i-node
  - changing a link only requires access to the directory, not the file.
  - files are only removed when the last link is removed
  - hard links can only be made within a filesystem.
  - only root may make a hard link to a directory
- Symbolic links:
  - like a link, but links to a name, not an i-node.
  - More fragile than a hard link, but can work across filesystems and to directories.

14.3.2 From last time: Links, symbolic and otherwise

A rather detailed discussion of adding/removing elements from directory trees ensued here.

Consider the process below of creating a file, creating a link to it, creating a symlink, then unlinking the original. Assume the directory at inode 12 is the root of the filesystem.

[3]That is, dot dot of slash is slash :)

<table>
<tr><th>Command</th><th>/
(inode 12)</th><th>/B
(inode 47)</th><th>Resulting Tree
Structure</th></tr>
<tr><td>—</td><td><table><tr><th>Name</th><th>i-node</th></tr><tr><td>.</td><td>12</td></tr><tr><td>..</td><td>12</td></tr><tr><td>B</td><td>47</td></tr><tr><td>A</td><td>15</td></tr></table></td><td><table><tr><th>Name</th><th>i-node</th></tr><tr><td>.</td><td>47</td></tr><tr><td>..</td><td>12</td></tr></table></td><td> 
</td></tr>
<tr><td><code>cd /B</code>
<code>ls > old</code></td><td><table><tr><th>Name</th><th>i-node</th></tr><tr><td>.</td><td>12</td></tr><tr><td>..</td><td>12</td></tr><tr><td>B</td><td>47</td></tr><tr><td>A</td><td>15</td></tr></table></td><td><table><tr><th>Name</th><th>i-node</th></tr><tr><td>.</td><td>47</td></tr><tr><td>..</td><td>12</td></tr><tr><td>old</td><td>128</td></tr></table></td><td> 
</td></tr>
<tr><td><code>ln -s old sym</code></td><td><table><tr><th>Name</th><th>i-node</th></tr><tr><td>.</td><td>12</td></tr><tr><td>..</td><td>12</td></tr><tr><td>B</td><td>47</td></tr><tr><td>A</td><td>15</td></tr></table></td><td><table><tr><th>Name</th><th>i-node</th></tr><tr><td>.</td><td>47</td></tr><tr><td>..</td><td>12</td></tr><tr><td>old</td><td>128</td></tr><tr><td>sym</td><td>“old”</td></tr></table></td><td> 
</td></tr>
<tr><td><code>ln old new</code></td><td><table><tr><th>Name</th><th>i-node</th></tr><tr><td>.</td><td>12</td></tr><tr><td>..</td><td>12</td></tr><tr><td>B</td><td>47</td></tr><tr><td>A</td><td>15</td></tr></table></td><td><table><tr><th>Name</th><th>i-node</th></tr><tr><td>.</td><td>47</td></tr><tr><td>..</td><td>12</td></tr><tr><td>old</td><td>128</td></tr><tr><td>sym</td><td>“old”</td></tr><tr><td>new</td><td>128</td></tr></table></td><td> 
</td></tr>
<tr><td><code>unlink old</code></td><td><table><tr><th>Name</th><th>i-node</th></tr><tr><td>.</td><td>12</td></tr><tr><td>..</td><td>12</td></tr><tr><td>B</td><td>47</td></tr><tr><td>A</td><td>15</td></tr></table></td><td><table><tr><th>Name</th><th>i-node</th></tr><tr><td>.</td><td>47</td></tr><tr><td>..</td><td>12</td></tr><tr><td>—</td><td>—</td></tr><tr><td>symlink</td><td>“old”</td></tr><tr><td>new</td><td>128</td></tr></table></td><td> 
</td></tr>
</table>

14.4 Manipulating the filesystem

```
#include <unistd.h>

int link(const char *oldpath, const char *newpath);
int unlink(const char *pathname);

int symlink(const char *oldpath, const char *newpath);
int readlink(const char *path, char *buf, size_t bufsiz);
         returns character count or -1 (e.g., ENAMETOOLONG)
         (not nul-terminated)

int mkdir(const char *pathname, mode_t mode);
int rmdir(const char *pathname);

int chdir(const char *path);
int fchdir(int fd);

char *getcwd(char *buf, size_t size);
         NULL and ERANGE on error
```

14.5 Directories

Note, that even though there are many fields described in the `struct dirent`, only `d_ino` and `d_name` are required by POSIX.1 and its extension. (POSIX.1 only requires `d_name`.)

```
#include <dirent.h>

       struct dirent
       {
           long d_ino;                /* inode number */
           off_t d_off;               /* offset to this dirent */
           unsigned short d_reclen;   /* length of this d_name */
           char d_name [NAME_MAX+1];  /* file name (null-terminated) */
       }


DIR *opendir(const char *name);
struct dirent *readdir(DIR *dir);
off_t telldir(DIR *dir);
void rewinddir(DIR *dir);
void seekdir(DIR *dir, off_t offset);
int closedir(DIR *dir);
```

15 Lecture: Filesytem wrapup, for sure, for sure

Outline:

15.1 Announcements

- Coming attractions:

Event	Subject	Due Date			Notes
asgn6	shell	Fri	Dec 8	23:59	

  Use your own discretion with respect to timing/due dates.
- Use the test harness
- Calloc is not a panacea
- `sizeof(char)` is 1
- Remember not to open your input files for writing, e.g.
- DTA until 11/24
- From email:

> We are hosting our largest pitch competition of the year, Startup Launch Weekend, from November 10-12. This Shark Tank-themed event will start downtown in the CIE Hothouse with guest speakers, catered food, and a business model canvas workshop. Throughout the weekend, students will form groups and develop a business idea, for the chance to win over $2100 in prizes!

15.2 Example: A recursive traversal for codes?

15.3 A note on the lab

Implementing:

```
char *getcwd(char *buf, size_t size);
        NULL and ERANGE on error
```

Inodes are unique to a filesystem, but inode and device are sufficient.

- current working directory and chdir()
- opendir, readdir, closedir
- Inodes on mounted filesystems.

15.4 From last time: Filesystem functions

```
#include <unistd.h>

int link(const char *oldpath, const char *newpath);
int unlink(const char *pathname);

int symlink(const char *oldpath, const char *newpath);
int readlink(const char *path, char *buf, size_t bufsiz);
         returns character count or -1 (e.g., ENAMETOOLONG)

int mkdir(const char *pathname, mode_t mode);
int rmdir(const char *pathname);

int chdir(const char *path);
int fchdir(int fd);

char *getcwd(char *buf, size_t size);
         NULL and ERANGE on error



DIR *opendir(const char *name);
struct dirent *readdir(DIR *dir);
off_t telldir(DIR *dir);
void rewinddir(DIR *dir);
void seekdir(DIR *dir, off_t offset);
int closedir(DIR *dir);
```

15.5 File Status Information: the stats

`stat()`, `lstat()`, and `fstat()` are the system calls for finding out about a file. (`stat(2)` on linux, `stat(3HEAD)` on solaris) They all populate a `struct stat`, shown in Figure 64.

```
NAME
       stat, fstat, lstat - get file status

SYNOPSIS
       #include <sys/types.h>
       #include <sys/stat.h>
       #include <unistd.h>

       int stat(const char *file_name, struct stat *buf);
       int fstat(int filedes, struct stat *buf);
       int lstat(const char *file_name, struct stat *buf);
```

EBADF	filedes is bad.
ENOENT	A component of the path file_name does not exist, or the path is an empty string.
ENOTDIR	A component of the path is not a directory.
ELOOP	Too many symbolic links encountered while traversing the path.
EFAULT	Bad address.
EACCES	Permission denied.
ENOMEM	Out of memory (i.e. kernel memory).
ENAMETOOLONG	File name too long.

```
struct stat
{
    dev_t         st_dev;      /* device */
    ino_t         st_ino;      /* inode */
    mode_t        st_mode;     /* protection */
    nlink_t       st_nlink;    /* number of hard links */
    uid_t         st_uid;      /* user ID of owner */
    gid_t         st_gid;      /* group ID of owner */
    dev_t         st_rdev;     /* device type (if inode device) */
    off_t         st_size;     /* total size, in bytes */
    blksize_t     st_blksize;  /* blocksize for filesystem I/O */
    blkcnt_t      st_blocks;   /* number of blocks allocated */
    time_t        st_atime;    /* time of last access */
    time_t        st_mtime;    /* time of last modification */
    time_t        st_ctime;    /* time of last change */
};
```

Figure 64: struct stat returned by stat()

Structure Info.	Access Info.	Audit Info.
• What is it? (type) • Number of links • Where is it? Device(`st_dev`) and inode (`st_ino`) • How big is it? • If it's a device file, which device (`st_rdev`)	• owner • group • other	• atime • mtime • ctime

Consider how you can identify a **mounted** filesystem.

15.6 Some Examples

15.6.1 Example: mv

For an example of directory manipulation, see the example of renaming a file in Figure 65.

15.6.2 Example: A minimally functional ls

An example of listing the contents of a directory is found in Figure 66.

```
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <stdio.h>
#include <unistd.h>

int main(int argc, char *argv[]){
  char *old, *new;
  struct stat sb;
  int status = 0;

  if ( argc != 3 ) {
    fprintf(stderr,"usage:  %s oldname newname\n",argv[0]);
    exit(-1);
  }

  old = argv[1];
  new = argv[2];

  if ( stat(old,&sb) ) {    /* make sure source exists */
    perror(old);
    status++;
  } else {
    if ( link(old, new) ) {     /* link the new name */
      perror(new);
      status ++;
    } else {
      if ( unlink(old) ) {      /* unlink the old one */
        perror(old);
        status++;
      }
    }
  }

  return status;
}
```

Figure 65: Renaming a file with link() and unlink()

```
#include <stdio.h>
#include <sys/types.h>
#include <dirent.h>


void listdir(char *path){
  /* list the contents of the given directory */
  DIR *d;
  struct dirent *ent;

  printf("Contents of %s:\n", path);

  if ( NULL == (d = opendir(path))) {
    perror(path);
  } else {
    while( (ent = readdir(d)) ) {
      printf("   %s\n", ent->d_name);
    }

    closedir(d);
  }
}

int main(int argc, char *argv[]) {
  int i;
  for(i=1; i < argc; i++)
    listdir(argv[i]);
  return 0;
}
```

Figure 66: A minimally functional implementation of ls.

15.7 Directories

Note, that even though there are many fields described in the `struct dirent`, only `d_ino` and `d_name` are required by POSIX.1 and its extension. (POSIX.1 only requires `d_name`.)

```
#include <dirent.h>

       struct dirent
       {
           long d_ino;                /* inode number */
           off_t d_off;               /* offset to this dirent */
           unsigned short d_reclen;   /* length of this d_name */
           char d_name [NAME_MAX+1];  /* file name (null-terminated) */
       }


DIR *opendir(const char *name);
struct dirent *readdir(DIR *dir);
off_t telldir(DIR *dir);
void rewinddir(DIR *dir);
void seekdir(DIR *dir, off_t offset);
int closedir(DIR *dir);
```

15.8 Example: microls.c

We developed something like Figure 67 in class today.

15.9 Identity Issues Revisited

Any process has several simultaneous ids:

real user id
real group id
effective user id
effective group id
(saved suid)
(saved sgid)

A file has two identities: user and group. The uid is the effective UID of the user who created it. The gid is either the effective group ID of the creating user, or the GID of the directory in which it is created.

15.10 File Protection Modes Revisited

The following POSIX macros are defined to check the file type:

Macro	Meaning
`S_ISLNK(m)`	is it a symbolic link?
`S_ISREG(m)`	regular file?
`S_ISDIR(m)`	directory?
`S_ISCHR(m)`	character device?
`S_ISBLK(m)`	block device?
`S_ISFIFO(m)`	fifo?
`S_ISSOCK(m)`	socket?

RWX permissions and their meanings

SUID—change user id on execution

SGID—change group id on execution

Sticky (saved-text)—various

See Figure 68 for the meanings of the particular bits. In general, use the macros.

```
#include<dirent.h>
#include<limits.h>
#include<stdio.h>
#include<stdlib.h>
#include<sys/stat.h>
#include<sys/types.h>
#include<unistd.h>

void ls_file(const char *path) {
  struct stat sb;
  if ( -1 == stat(path,&sb)) {
    perror(path);
  } else {
    printf("%s %ld\n",path,(long)sb.st_size);
  }
}

void ls_dir(const char *path) {
  DIR *d;
  struct dirent *ent;
  char name[PATH_MAX+1];

  if ( NULL == ( d = opendir(path))) {
    perror(path);
  } else {
    while(NULL != (ent=readdir(d))) {
      snprintf(name,PATH_MAX,"%s/%s",path,ent->d_name);
      ls_file(name);
    }
  }
}

int main(int argc, char *argv[]) {
  int i;
  struct stat sb;
  for(i=1;i<argc;i++) {
    if ( -1 == stat(argv[i],&sb)) {
      perror(argv[i]);
    } else if ( S_ISDIR(sb.st_mode)) {
      ls_dir(argv[i]);
    } else {
      ls_file(argv[i]);
    }
  }
  return 0;
}
```

Figure 67: A less-minimally functional implementation of ls.

Mode Bits		
Macro	**Value**	**Meaning**
S_IFMT	0170000	bitmask for the file type bitfields
S_IFSOCK	0140000	socket
S_IFLNK	0120000	symbolic link
S_IFREG	0100000	regular file
S_IFBLK	0060000	block device
S_IFDIR	0040000	directory
S_IFCHR	0020000	character device
S_IFIFO	0010000	fifo
S_ISUID	0004000	set UID bit
S_ISGID	0002000	set GID bit (see below)
S_ISVTX	0001000	sticky bit (see below)
S_IRWXU	00700	mask for file owner permissions
S_IRUSR	00400	owner has read permission
S_IWUSR	00200	owner has write permission
S_IXUSR	00100	owner has execute permission
S_IRWXG	00070	mask for group permissions
S_IRGRP	00040	group has read permission
S_IWGRP	00020	group has write permission
S_IXGRP	00010	group has execute permission
S_IRWXO	00007	mask for permissions for others (not in group)
S_IROTH	00004	Others have read permission
S_IWOTH	00002	Others have write permission
S_IXOTH	00001	Others have execute permission

Figure 68: Macros for bits in file modes

15.11 Where a new file gets its values

uid	effective uid of creator
gid	egid of creator or gid of directory (BSD/solaris/linux)
	Linux is configurable(mount(8)): grpid or bsdgroups / nogrpid or sysvgroups
permissions	`mode && ~umask`

15.12 Changing these values

```
#include <sys/types.h>
#include <sys/stat.h>


int     chmod(const char *path, mode_t mode);
int     fchmod(int fildes, mode_t mode);
mode_t  umask(mode_t mask);
int     chown(const char *path, uid_t owner, gid_t group);
int     fchown(int fd, uid_t owner, gid_t group);
int     lchown(const char *path, uid_t owner, gid_t group);
```

Restrictions on chmod() and chown():

- Only the owner can chown() or chmod() a file.
- **chmod()** clears sticky bit if not root. Sgid turned off by chmod() if owner is not a member of the group.(WHY?)
- suid and sgid turned off on write by a process is not owner (or not root).
- **chown()** Can only chown to self if not root. Can only chgrp to group where one is a member.

15.13 File Permissions: `umask(2)`

The umask us used to remove particular permissions during the process of file-creation with `open(2)` or `creat(2)`. When one of those functions is called, the given permissions are ANDed with the bitwise inverse of the umask to get the final permissions.

```
#include <sys/types.h>
#include <sys/stat.h>

mode_t umask(mode_t mask);
```

15.14 Catchall calls: fcntl, ioctl

15.14.1 fcntl() — modifying an open file descriptor

```
SYNOPSIS
       #include <unistd.h>
       #include <fcntl.h>

       int fcntl(int fd, int cmd);
       int fcntl(int fd, int cmd, long arg);
       int fcntl(int fd, int cmd, struct flock * lock);
```

Of interest now:

- **File descriptor flags:** `F_GETFD`, `F_SETFD`

  There is only one flag of interest here: `FD_CLOEXEC`, close on exec.

- **File status flags:** `F_GETFL`, `F_SETFL`

  Allows you to examine or change the way file IO is handled on an open file descriptor: `O_RDONLY`, `O_WRONLY`, `O_RDWR` cannot be changed, but `O_APPEND` (and `O_NONBLOCK`, `O_SYNC` `O_ASYNC` for that matter) can be.

15.14.2 ioctl() — the kitchen sink

```
#include <sys/ioctl.h>

int ioctl(int d, int request, ...);
```

ioctl() is the traditional catchall for device manipulation. Its semantics are defined by each particular device driver.

15.15 Linking internal and external identity

```
int chown(const char *path, uid_t owner, gid_t group);

#include <pwd.h>
#include <sys/types.h>
struct passwd *getpwnam(const char * name);
struct passwd *getpwuid(uid_t uid);

#include <grp.h>
#include <sys/types.h>
struct group *getgrnam(const char *name);
struct group *getgrgid(gid_t gid);
```

```
struct passwd {
        char    *pw_name;       /* user name */
        char    *pw_passwd;     /* user password */
        uid_t   pw_uid;         /* user id */
        gid_t   pw_gid;         /* group id */
        char    *pw_gecos;      /* real name */
        char    *pw_dir;        /* home directory */
        char    *pw_shell;      /* shell program */
};
```

```
struct group {
        char    *gr_name;       /* group name */
        char    *gr_passwd;     /* group password */
        gid_t   gr_gid;         /* group id */
        char    **gr_mem;       /* group members */
};
```

15.16 Just a Matter of `time(2)`

Unix represents time as the number of seconds since Midnight, January 1st, 1970, UTC (coordinated universal time). Anything else is just a prettification.

```
SYNOPSIS
       #include <time.h>

       time_t time(time_t *t);

DESCRIPTION
       time  returns the time since the Epoch (00:00:00 UTC, Jan-
       uary 1, 1970), measured in seconds.

       If t is non-NULL, the return value is also stored  in  the
       memory pointed to by t.
```

The three times associated with any file are:

stat(2)	ls(1) -l	Meaning
atime	-u	Time of last access
mtime	<none>	Time of last modification
ctime	-c	Time of last change (incl. of the inode)

15.16.1 Time Display Routines <time.h>

```
NAME
       asctime,  ctime,  gmtime,  localtime,  mktime  - transform
```

```
       binary date and time to ASCII

SYNOPSIS
       #include <time.h>

       char *asctime(const struct tm *timeptr);

       char *ctime(const time_t *timep);

       struct tm *gmtime(const time_t *timep);

       struct tm *localtime(const time_t *timep);

       time_t mktime(struct tm *timeptr);

       extern char *tzname[2];
       long int timezone;
       extern int daylight;

       size_t strftime(char *s, size_t max, const char *format,
                       const struct tm *tm);



              struct tm
              {
                      int     tm_sec;         /* seconds */
                      int     tm_min;         /* minutes */
                      int     tm_hour;        /* hours */
                      int     tm_mday;        /* day of the month */
                      int     tm_mon;         /* month */
                      int     tm_year;        /* year */
                      int     tm_wday;        /* day of the week */
                      int     tm_yday;        /* day in the year */
                      int     tm_isdst;       /* daylight saving time */
              };
```

15.16.2 Time Manipulation Routines: utime()

```
UTIME(2)            Linux Programmer's Manual            UTIME(2)

NAME
       utime, utimes - change access and/or modification times of
       an inode

SYNOPSIS
       #include <sys/types.h>
       #include <utime.h>

       int utime(const char *filename, struct utimbuf *buf);

DESCRIPTION
       utime changes the access and  modification  times  of  the
       inode  specified  by  filename  to  the actime and modtime
       fields of buf respectively.  If  buf  is  NULL,  then  the
       access  and  modification times of the file are set to the
       current time.  The utimbuf structure is:

              struct utimbuf {
                      time_t actime;  /* access time */
                      time_t modtime; /* modification time */
```

```
            };
```

15.17 Implementation details: the buffer cache

```
NAME
       sync - commit buffer cache to disk.

SYNOPSIS
       #include <unistd.h>

       int sync(void);
       int fsync(int fd);
```

16 Lecture: Signals, POSIX Signals, Timers and Alarms

Outline:

16.1 Announcements

- Coming attractions:

Event	Subject	Due Date			Notes
asgn6	shell	Fri	Dec 8	23:59	

  Use your own discretion with respect to timing/due dates.
- Today: From last time: the stats, chdir, chmod, chown, holes
- Are you reading the feedback and/or running the test cases?
- Back to umask
- lab05 How to do mypwd()?
- Midterm archive out.
- You will always lose points for unchecked malloc()s
- Bring questions Wedensday and Friday.
- Midterm is scheduled for next Monday.
  - All (well, most) prototypes and constants you'll need will be on the back of the exam.

16.2 stuff to talk about

- Where does file ownership come from?
- Number of inodes is limited

16.3 Resource Limitations

Consider what goes into creating a file:

- Inode
- Dirent
- Space to store data (exept holes)

16.4 A note on the lab

Implementing:

```
char *getcwd(char *buf, size_t size);
        NULL and ERANGE on error
```

Inodes are unique to a filesystem, but inode and device are sufficient.

- current working directory and chdir()
- opendir, readdir, closedir
- Inodes on mounted filesystems.

16.5 getpwd()

```
The _only_ part of the struct dirent that is reliable is the d_name
field.  Everything else requires one of the stats.

Why?  Well, what's happened is that you've encountered a mountpoint.
Readdir() reads the directory file itself, so it's reading the name and
inode number that are stored in the file.  It has no knowledge of
mounted filesystems, though, so it doesn't know that there's a
filesystem mounted on that directory.  In the struct dirent you're
seeing the inode number of the directory that the other fs was mounted
on top of.  When you call stat(), though, it consults the kernel's mount
tables and gives you the inode/device/rest of the info pertaining to the
root of the mounted filesystem which is what you actually want.

I hope this helps.
```

16.6 The rest of the quarter

16.6.1 Subjects

- Signals
- tty line disciplines (IO device management) (maybe)
- Process Management (fork(), the exec()s, process groups, etc.)
- Process Environments (env, resource limits)
- Other useful stuff (strtok(), setjmp(), longjmp())

The next two subjects (Terminal IO and Signals) show the most differentiation between the UNIX family members.

16.7 Asynchronous Events and signals

In the systems programming realm we have to give up the precious illusion that programs exist in isolation. Sometimes things come up. (e.g. a child process exits or a seg fault.) Sometimes it's events we asked for (SIGALRM or SIGVTALRM).

A Signal is a software interrupt indicating some "important" condition. There are three possible reactions to a signal:

`SIG_IGN`	Ignore it
`SIG_DFL`	default (ignore or die)
catch	whatever you want

- From Hardware: SIGILL, SIGPWR, SIGFPE
- From Software: via `kill(1)`, `kill(2)`, `killpg(2)`, `raise(3)`

```
#include <sys/types.h>
#include <signal.h>

int kill(pid_t pid, int sig);
int raise (int sig);
```

- From Terminal Line discipline (non-canonical (cooked) mode processing)

For all default actions, see page 266 in Stevens (table 10.1)
Behavior is undefined after ignoring certain signals (E.g., SIGSEGV or SIGFPE)

16.8 Unreliable Signal Handling: `signal(2)`

- one shot vs. repetition

```
SIGNAL(2)           Linux Programmer's Manual           SIGNAL(2)

NAME
```

```
       signal - ANSI C signal handling

SYNOPSIS
       #include <signal.h>

       void (*signal(int signum, void (*sighandler)(int)))(int);
```

Possible values:

- `SIG_IGN`
- `SIG_DFL`
- pointer to handler
- `SIG_ERR` on error

SIGKILL and SIGSTOP cannot be caught or ignored.

16.8.1 Aside: Function pointers

That horrible prototype could better be re-written:

```
typedef void (*sigfun)(int);
sigfun signal(int signum, sigfun handler);
```

An example of signal handling code can be found in Figure 69.

16.9 What is unreliable about it?

To understand this, we have to look at how signals are delivered.

16.10 Signal Delivery

Because signals are asynchronous events, it is hard to say too much about when they will be delivered. Generally, however, processes that have asked to be notified of events want to know about them, so the system tries to deliver them quickly.

Certain "slow" system calls (those that may block forever: e.g. `wait()`, `read()`, `write()`, `sleep()`, `pause()`) are interrupted when a signal is delivered.

Some implementations (4.3BSD, e.g.) automatically restart some of these calls. POSIX (below) allows for such behavior, but doesn't enforce it. (What if you don't want it to restart?)

What happens when a signal is delivered:

- If the handler is set to `SIG_IGN`, the signal is ignored.
- If the handler is set to `SIG_DFL`, the default action is performed.
- If the handler is set to a function:
  1. The handler is (usually) set to `SIG_DFL` (or implementation-dependent blocking)

     (Compare Linux to Solaris — Linux implements the BSD4.3 blocking semantics, Solaris restores the default handler.)
  2. The signal handler is called with argument `signum`

The resetting of the signal action upon delivery of a signal causes the unreliability: Even if the signal handler is immediately re-registered, there is a window of time in which another instance of the signal will have the default action, not the intended one.

Also, it is not possible to find out what is currently being done with a signal without changing it (however briefly). You can do something like the code in Figure 70, but you can't always get it.

These *race conditions* make signal handling in this manner unreliable.

```
/*
 * sigtst:
 *
 * Author: Dr. Phillip Nico
 *         Department of Computer Science
 *         California Polytechnic State University
 *         One Grand Avenue.
 *         San Luis Obispo, CA  93407  USA
 *
 * Email:  pnico@calpoly.edu
 *
 * Revision History:
 *         $Log:$
 */

#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>

#include <signal.h>

typedef void (*sigfun)(int);

void handler(int sig) {
  /* Works once */
  printf("Hello, I caught a SIGINT.  Neener, neener!");
  fflush(stdout);
}

int main(int argc, char *argv[]){
  sigfun old;

  if ( ( old = signal(SIGINT,handler) ) == SIG_ERR ) {
    perror("signal");
    exit(-1);
  }

  for (;;)
    /* nothing */;

  return 0;
}
```

Figure 69: A program that catches SIGINT

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>

#include <signal.h>
#include <sys/types.h>

typedef void (*sigfun)(int);

static int ctr = 0;

void temp(int sig) {
  signal(SIGINT,temp);
  ctr++;
}


int main(int argc, char *argv[]){
  int i;
  sigfun old;

  ctr = 0;                    /* reset the counter */
  old = signal(SIGINT,temp);  /* get the old value */
  signal(SIGINT,old);         /* put it back */


  #ifdef ONEWAY
  switch ( (int)old ) {       /* cheat a little and act like it's an int */
  case SIG_IGN:
    /* ignore it */
    break;
  case SIG_DFL:
    /* resend the signal */
    kill ( getpid(), SIGINT);
    break;
  default:
    while(ctr--) {            /* run old() for all the missed signals */
      (*old)(SIGINT);
    }
    break;
  }
  #else
  while(ctr--) {              /* re-send all the missed signals */
    kill(getpid(),SIGINT);
  }
  #endif

  return 0;
}
```

Figure 70: Checking the current action with unreliable signals

16.10.1 Afterwards

The program continues merrily on its way. If a system call was interrupted, it returns with an error and sets `errno` to `EINTR`.

Example: if a signal is caught during:

```
c = getchar()
```

`c` will be `EOF`, whether or not the real end of file has been reached, and `errno` will be `EINTR`. (Use `feof(FILE *)` to find out.)

16.11 Tool of the Week: rcs(1)

RCS (Revision Control System) is a version-control system. (not nearly as fancy as SVN, CVS, git, . . . , but requires almost no setup.)

mkdir RCS Create repository

ci -u foo.c Checks in "foo.c" and removes it

ci -u foo.c Checks in "foo.c" and keeps a read-only copy (unlocked)

ci -i foo.c Checks in "foo.c" and keeps a read-write copy (unlocked)

co -u foo.c Checks out "foo.c" read-only (unlocked)

co -l foo.c Checks out "foo.c" read-write (locked)

rcsdiff foo.c shows the difference between this version and the last

rlog foo.c show the log messages.

Emacs interface:

C-x v i Install buffer into RCS

C-x v v Toggle-read-only checks the file in or out

C-c C-c Complete checkin

17 Lecture: Signals

Outline:

17.1 Announcements

- Weekend.
- Bring questions monday and wednesday.
- Midterm is scheduled for next week
  - any prototypes/struts will be on the exam
- Coming attractions:

Event	Subject	Due Date			Notes
asgn6	shell	Fri	Dec 8	23:59	

  Use your own discretion with respect to timing/due dates.
- Previous Exams to web.
- Midterm coming
  - Bring questions next time

17.2 getpwd()

```
The _only_ part of the struct dirent that is reliable is the d_name
field.  Everything else requires one of the stats.

Why?  Well, what's happened is that you've encountered a mountpoint.
Readdir() reads the directory file itself, so it's reading the name and
inode number that are stored in the file.  It has no knowledge of
mounted filesystems, though, so it doesn't know that there's a
filesystem mounted on that directory.  In the struct dirent you're
seeing the inode number of the directory that the other fs was mounted
on top of.  When you call stat(), though, it consults the kernel's mount
tables and gives you the inode/device/rest of the info pertaining to the
root of the mounted filesystem which is what you actually want.

I hope this helps.
```

17.3 The rest of the quarter

17.3.1 Subjects

- Signals
- tty line disciplines (IO device management) (maybe)
- Process Management (fork(), the exec()s, process groups, etc.)
- Process Environments (env, resource limits)
- Other useful stuff (strtok(), setjmp(), longjmp())

The next two subjects (Terminal IO and Signals) show the most differentiation between the UNIX family members.

17.3.2 interaction

Communication with signal handlers must be done through globals or functions with side-effects.

17.4 Critical Sections

Sometimes a program just has to say "not now"...

(E.g. updating a linked list and a handler writes the list)

Can ignore the signal (might miss it) or set a flag for later(tricky). (POSIX lets the signal be blocked and delivered later... tune in later.)

One possible approach is to change the signal handler to something else while handling one instance.

17.5 Communicating with signal handlers

Communicating with signal handlers is a little crude because a signal handler can't return anything since it doesn't know when it's being called.

Requires the use of globals...

17.6 From Last Time: Unreliable Signal Handling: `signal(2)`

```
void (*signal(int signum, void (*sighandler)(int)))(int);
```

On delivery:

- signal handler is called with the signal number
- (maybe) the handler is reset to `SIG_DFL`
- (maybe) the further deliver of the same signal is blocked
- (maybe) slow system calls are interrupted, returning error and setting `errno` to `EINTR`

Signal Generation

- From Hardware: SIGILL, SIGPWR, SIGFPE
- From Software: via `kill(1)`, `kill(2)`, `killpg(2)`, `raise(3)`

```
#include <sys/types.h>
#include <signal.h>

int kill(pid_t pid, int sig);
int raise (int sig);
```

- From Terminal Line discipline (non-canonical (cooked) mode processing)

17.7 Reliable Signal Handling (POSIX)

Components that are of interest:

- `sigaction` — reliable signal handling
- masking
- generation.
- waiting (sigsuspend(), pause())

Look into `sigaction(2)` and `sigsetops(2)`. (See Figure 75.) The reliable signal semantics allows for signal handlers to be installed until told otherwise. By default, when a signal is caught, further instances are blocked (and possibly queued) until the handler completes.

The POSIX semantics provide support for all sorts of tuning one might want to do, but this makes the interface more complicated.

sigaction
`int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact);`
`int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);`
sigsetops
`int sigemptyset(sigset_t *set);`
`int sigfillset(sigset_t *set);`
`int sigaddset(sigset_t *set, int signum);`
`int sigdelset(sigset_t *set, int signum);`
`int sigismember(const sigset_t *set, int signum);`
Other
`int sigsuspend(const sigset_t *mask);`
`int pause(void);`

Figure 71: System calls associated with POSIX signal handling

```
struct sigaction {
    void (*sa_handler)(int);
    void (*sa_sigaction)(int, siginfo_t *, void *);
    sigset_t sa_mask;
    int sa_flags;
}
```

The siginfo_t parameter to sa_sigaction is a struct with the following elements

```
siginfo_t {
    int      si_signo;  /* Signal number */
    int      si_errno;  /* An errno value */
    int      si_code;   /* Signal code */
    pid_t    si_pid;    /* Sending process ID */
    uid_t    si_uid;    /* Real user ID of sending process */
    int      si_status; /* Exit value or signal */
    clock_t  si_utime;  /* User time consumed */
    clock_t  si_stime;  /* System time consumed */
    sigval_t si_value;  /* Signal value */
    int      si_int;    /* POSIX.1b signal */
    void *   si_ptr;    /* POSIX.1b signal */
    void *   si_addr;   /* Memory location which caused fault */
    int      si_band;   /* Band event */
    int      si_fd;     /* File descriptor */
}
```

(sa_sigaction is active if SA_SIGINFO flag is set) Example flags:

`SA_RESTART`	restart interrupted system calls
`SA_RESETHAND`	this handler only handles once
`SA_SIGINFO`	use the siginfo parameter
`SA_NODEFER`	Does not block signals during handling

`SA_RESETHAND` and `SA_NODEFER` is equivalent to the old unreliable semantics.

With `sigaction()`:

sigprocmask can have three possible values of *how:*

`SIG_BLOCK`	The set of blocked signals is the union of the current set and the set argument.
`SIG_UNBLOCK`	The signals in set are removed from the current set of blocked signals. It is legal to attempt to unblock a signal which is not blocked.
`SIG_SETMASK`	The set of blocked signals is set to the argument set.

17.8 Sigaction Example

See Figures 76 and 77 and 78.

Note the difference between "`pause()`" and "`for(;;)`"

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <signal.h>

static int caught;

void handler(int signum){
  /* Handler for both SIGINT and SIGQUIT */
  fprintf(stderr,"Caught signal %d.     That's %d.\n", signum, ++caught);
}

int main(int argc, char *argv[]){
  struct sigaction sa;

  sa.sa_handler         = handler;  /* set up the handler */
  sigemptyset(&sa.sa_mask);  /* mask nothing */
  sa.sa_flags   = 0;                /* no special handling */

  caught = 0;

  if ( -1 == sigaction(SIGINT, &sa,NULL) ) {
    perror("sigaction");
    exit(-1);
  }

  if ( -1 == sigaction(SIGQUIT, &sa,NULL) ) {
    perror("sigaction");
    exit(-1);
  }

  /* wait for a signal */
  while ( caught < 3 )
      pause();                  /* wait for a signal */

  return 0;
}
```

Figure 72: A program that catches SIGINT and SIGQUIT using sigaction

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <signal.h>

static int caught;

void handler(int signum){
  /* Handler for both SIGINT and SIGQUIT */
  fprintf(stderr,"Caught signal %d.     That's %d.\n", signum, ++caught);
}

int main(int argc, char *argv[]){
  struct sigaction sa;
  sigset_t mask;

  sa.sa_handler         = handler;  /* set up the handler */
  sigemptyset(&sa.sa_mask);  /* mask nothing */
  sa.sa_flags   = 0;                /* no special handling */

  caught = 0;

  if ( -1 == sigaction(SIGINT, &sa,NULL) ) {
    perror("sigaction");
    exit(-1);
  }

  if ( -1 == sigaction(SIGQUIT, &sa,NULL) ) {
    perror("sigaction");
    exit(-1);
  }

  /* wait only for SIGQUIT */
  sigfillset(&mask);
  sigdelset(&mask, SIGQUIT);
  while ( caught < 3 )
      sigsuspend(&mask);        /* wait for a signal */

  return 0;
}
```

Figure 73: A program that catches SIGINT and SIGQUIT using sigaction using `sigsuspend()`

```
% one
^C
Caught signal 2. That's 1.
^\
Caught signal 3. That's 2.
^C
Caught signal 2. That's 3.

% two
^C^\
Caught signal 3. That's 1.
Caught signal 2. That's 2.
^C^\
Caught signal 3. That's 3.
Caught signal 2. That's 4.
%
```

Figure 74: The output from these programs

17.9 Signal generation: Alarm clocks

17.10 alarm(2)

```
#include <unistd.h>
unsigned int alarm(unsigned int seconds);
```

`alarm(2)` is not very flexible, but it is a simple interface. after `seconds` seconds have elapsed a `SIGALRM` is sent.

The process is not guaranteed to be scheduled immediately, though, and timing errors can compound (illustrate clock drift).

> Make sure to illustrate the problem of clock drift with repeated alarms, relative to mytimer

`alarm(0)` clears alarms.
`alarm()` returns the number of seconds remaining.

17.11 setitimer(2)

The interval timer is finer-grained and more accurate:

```
#include <sys/time.h>

int getitimer(int which, struct itimerval *value);
int setitimer(int which, const  struct  itimerval  *value,
       struct itimerval *ovalue);


     struct itimerval {
         struct timeval it_interval; /* next value */
         struct timeval it_value;    /* current value */
     };

     struct timeval {
         long tv_sec;                /* seconds */
         long tv_usec;               /* microseconds */
     };
```

The `usec` number is optimistic, however. It depends on the system clock's resolution. (linux box: 10ms)
It comes in three flavors:

`ITIMER_REAL`	decrements in real time, and delivers SIGALRM upon expiration.
`ITIMER_VIRTUAL`	decrements only when the process is executing and delivers SIGVTALRM upon expiration.
`ITIMER_PROF`	decrements both when the process executes and when the system is executing on behalf of the process. Coupled with ITIMER_VIRTUAL, this timer is usually used to profile the time spent by the application in user and kernel space. SIGPROF is delivered upon expiration.

17.12 sleep(3), pause(2), and sigsuspend(2)

17.12.1 sleep(3)

```
SYNOPSIS
       #include <unistd.h>

       unsigned int sleep(unsigned int seconds);

DESCRIPTION
       sleep()  makes  the  current  process  sleep until seconds seconds have
       elapsed or a signal arrives which is not ignored.

RETURN VALUE
       Zero if the requested time has elapsed, or the number of  seconds  left
       to sleep.
```

17.12.2 pause(2)

```
       #include <unistd.h>

       int pause(void);

DESCRIPTION
       The  pause  library function causes the invoking process (or thread) to
       sleep until a signal is received that either terminates it or causes it
       to call a signal-catching function.
```

17.12.3 sigsuspend(2)

```
       #include <signal.h>

       int sigsuspend(const sigset_t *mask);

DESCRIPTION
       The  sigsuspend call temporarily replaces the signal mask for the pro-
       cess with that given by mask and then suspends  the  process  until  a
       signal is received.


RETURN VALUE
       The function sigsuspend always returns -1, normally with the error
       EINTR.
```

17.13 Unwanted Interactions: sleep(3)

Flying at different levels can be dangerous: when you tweak system-dependent features, you may upset other implementations: The Solaris `sleep(3)` function uses the interval timer. You can't use both. You can use `pause(2)`.

`pause(2)` means "wait for a signal"

18 Lecture: Reliable Signal Handling and Timers

Outline:

- Announcements
- The rest of the quarter
  - Subjects
  - interaction
- Critical Sections
- Communicating with signal handlers
- From Last Time: Unreliable Signal Handling: `signal(2)`
- Reliable Signal Handling (POSIX)
- Sigaction Example
- Signal generation: Alarm clocks
- `alarm(2)`
- `setitimer(2)`
- sleep(3), pause(2), and sigsuspend(2)
  - sleep(3)
  - pause(2)
  - sigsuspend(2)
- Unwanted Interactions: sleep(3)
- Mid(end?)term
  - Style
  - Resources
- Mid(end?)term
  - Style
  - Resources
  - Material
- Mid(end?)term
  - Style
  - Resources
  - Material

18.1 Announcements

- Coming attractions:

Event	Subject	Due Date			Notes
asgn6	shell	Fri	Dec 8	23:59	

  Use your own discretion with respect to timing/due dates.

- Dumb `mypwd`: 4:16.56
- Midterm coming
- Be working on Asgn4 (now's the chance to make it good.)
- Lab05:mypwd() due today

```
  Possible: 60.0
      High: 60.0    (100.0%)
       Low: 10.0    ( 16.7%)
      Mean: 43.1    ( 71.8%)
    Median: 45.0    ( 75.0%)
      S.D.: 10.4    ( 17.4%)

   Enrolled:   60
  Submitted:   49    ( 81.7%)

lagniappe% gradescale 50 12.5
Max Possible:  50.0
     Buckets:  12.5% of  50.0

  Min A-: 43.8  (87.5%)
```

```
Min B-: 37.5  (75.0%)
Min C-: 31.2  (62.5%)
Min D-: 25.0  (50.0%)
```

18.2 The rest of the quarter

18.2.1 Subjects

- Signals
- tty line disciplines (IO device management) (maybe)
- Process Management (fork(), the exec()s, process groups, etc.)
- Process Environments (env, resource limits)
- Other useful stuff (strtok(), setjmp(), longjmp())

The next two subjects (Terminal IO and Signals) show the most differentiation between the UNIX family members.

18.2.2 interaction

Communication with signal handlers must be done through globals or functions with side-effects.

18.3 Critical Sections

Sometimes a program just has to say "not now"...

(E.g. updating a linked list and a handler writes the list)

Can ignore the signal (might miss it) or set a flag for later(tricky). (POSIX lets the signal be blocked and delivered later. . . tune in later.)

One possible approach is to change the signal handler to something else while handling one instance.

18.4 Communicating with signal handlers

Communicating with signal handlers is a little crude because a signal handler can't return anything since it doesn't know when it's being called.

Requires the use of globals...

18.5 From Last Time: Unreliable Signal Handling: `signal(2)`

```
void (*signal(int signum, void (*sighandler)(int)))(int);
```

On delivery:

- signal handler is called with the signal number
- (maybe) the handler is reset to `SIG_DFL`
- (maybe) the further deliver of the same signal is blocked
- (maybe) slow system calls are interrupted, returning error and setting `errno` to `EINTR`

Signal Generation

- From Hardware: SIGILL, SIGPWR, SIGFPE
- From Software: via `kill(1)`, `kill(2)`, `killpg(2)`, `raise(3)`

```
#include <sys/types.h>
#include <signal.h>

int kill(pid_t pid, int sig);
int raise (int sig);
```

- From Terminal Line discipline (non-canonical (cooked) mode processing)

18.6 Reliable Signal Handling (POSIX)

Components that are of interest:

- `sigaction` — reliable signal handling
- masking
- generation.
- waiting (sigsuspend(), pause())

Look into `sigaction(2)` and `sigsetops(2)`. (See Figure 75.) The reliable signal semantics allows for signal handlers to be installed until told otherwise. By default, when a signal is caught, further instances are blocked (and possibly queued) until the handler completes.

The POSIX semantics provide support for all sorts of tuning one might want to do, but this makes the interface more complicated.

sigaction
`int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact);`
`int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);`
sigsetops
`int sigemptyset(sigset_t *set);`
`int sigfillset(sigset_t *set);`
`int sigaddset(sigset_t *set, int signum);`
`int sigdelset(sigset_t *set, int signum);`
`int sigismember(const sigset_t *set, int signum);`
Other
`int sigsuspend(const sigset_t *mask);`
`int pause(void);`

Figure 75: System calls associated with POSIX signal handling

```
struct sigaction {
    void (*sa_handler)(int);
    void (*sa_sigaction)(int, siginfo_t *, void *);
    sigset_t sa_mask;
    int sa_flags;
}
```

The siginfo_t parameter to sa_sigaction is a struct with the following elements

```
siginfo_t {
    int      si_signo;  /* Signal number */
    int      si_errno;  /* An errno value */
    int      si_code;   /* Signal code */
    pid_t    si_pid;    /* Sending process ID */
    uid_t    si_uid;    /* Real user ID of sending process */
    int      si_status; /* Exit value or signal */
    clock_t  si_utime;  /* User time consumed */
    clock_t  si_stime;  /* System time consumed */
    sigval_t si_value;  /* Signal value */
    int      si_int;    /* POSIX.1b signal */
    void *   si_ptr;    /* POSIX.1b signal */
    void *   si_addr;   /* Memory location which caused fault */
    int      si_band;   /* Band event */
    int      si_fd;     /* File descriptor */
}
```

(sa_sigaction is active if SA_SIGINFO flag is set) Example flags:

`SA_RESTART`	restart interrupted system calls
`SA_RESETHAND`	this handler only handles once
`SA_SIGINFO`	use the siginfo parameter
`SA_NODEFER`	Does not block signals during handling

`SA_RESETHAND` and `SA_NODEFER` is equivalent to the old unreliable semantics.

With `sigaction()`:

sigprocmask can have three possible values of *how:*

`SIG_BLOCK`	The set of blocked signals is the union of the current set and the set argument.
`SIG_UNBLOCK`	The signals in set are removed from the current set of blocked signals. It is legal to attempt to unblock a signal which is not blocked.
`SIG_SETMASK`	The set of blocked signals is set to the argument set.

18.7 Sigaction Example

See Figures 76 and 77 and 78.

Note the difference between "`pause()`" and "`for(;;)`"

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <signal.h>

static int caught;

void handler(int signum){
  /* Handler for both SIGINT and SIGQUIT */
  fprintf(stderr,"Caught signal %d.     That's %d.\n", signum, ++caught);
}

int main(int argc, char *argv[]){
  struct sigaction sa;

  sa.sa_handler         = handler;  /* set up the handler */
  sigemptyset(&sa.sa_mask);  /* mask nothing */
  sa.sa_flags   = 0;                /* no special handling */

  caught = 0;

  if ( -1 == sigaction(SIGINT, &sa,NULL) ) {
    perror("sigaction");
    exit(-1);
  }

  if ( -1 == sigaction(SIGQUIT, &sa,NULL) ) {
    perror("sigaction");
    exit(-1);
  }

  /* wait for a signal */
  while ( caught < 3 )
      pause();                  /* wait for a signal */

  return 0;
}
```

Figure 76: A program that catches SIGINT and SIGQUIT using sigaction

```c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <signal.h>

static int caught;

void handler(int signum){
  /* Handler for both SIGINT and SIGQUIT */
  fprintf(stderr,"Caught signal %d.     That's %d.\n", signum, ++caught);
}

int main(int argc, char *argv[]){
  struct sigaction sa;
  sigset_t mask;

  sa.sa_handler         = handler;  /* set up the handler */
  sigemptyset(&sa.sa_mask);  /* mask nothing */
  sa.sa_flags   = 0;                /* no special handling */

  caught = 0;

  if ( -1 == sigaction(SIGINT, &sa,NULL) ) {
    perror("sigaction");
    exit(-1);
  }

  if ( -1 == sigaction(SIGQUIT, &sa,NULL) ) {
    perror("sigaction");
    exit(-1);
  }

  /* wait only for SIGQUIT */
  sigfillset(&mask);
  sigdelset(&mask, SIGQUIT);
  while ( caught < 3 )
      sigsuspend(&mask);        /* wait for a signal */

  return 0;
}
```

Figure 77: A program that catches SIGINT and SIGQUIT using sigaction using `sigsuspend()`

```
% one
^C
Caught signal 2. That's 1.
^\
Caught signal 3. That's 2.
^C
Caught signal 2. That's 3.

% two
^C^\
Caught signal 3. That's 1.
Caught signal 2. That's 2.
^C^\
Caught signal 3. That's 3.
Caught signal 2. That's 4.
%
```

Figure 78: The output from these programs

18.8 Signal generation: Alarm clocks

18.9 alarm(2)

```
#include <unistd.h>
unsigned int alarm(unsigned int seconds);
```

`alarm(2)` is not very flexible, but it is a simple interface. after `seconds` seconds have elapsed a `SIGALRM` is sent.

The process is not guaranteed to be scheduled immediately, though, and timing errors can compound (illustrate clock drift).

> Make sure to illustrate the problem of clock drift with repeated alarms, relative to mytimer

`alarm(0)` clears alarms.
`alarm()` returns the number of seconds remaining.

18.10 setitimer(2)

The interval timer is finer-grained and more accurate:

```
#include <sys/time.h>

int getitimer(int which, struct itimerval *value);
int setitimer(int which, const  struct  itimerval  *value,
       struct itimerval *ovalue);


    struct itimerval {
        struct timeval it_interval; /* next value */
        struct timeval it_value;    /* current value */
    };

    struct timeval {
        long tv_sec;                /* seconds */
        long tv_usec;               /* microseconds */
    };
```

The `usec` number is optimistic, however. It depends on the system clock's resolution. (linux box: 10ms)
It comes in three flavors:

`ITIMER_REAL`	decrements in real time, and delivers SIGALRM upon expiration.
`ITIMER_VIRTUAL`	decrements only when the process is executing and delivers SIGVTALRM upon expiration.
`ITIMER_PROF`	decrements both when the process executes and when the system is executing on behalf of the process. Coupled with ITIMER_VIRTUAL, this timer is usually used to profile the time spent by the application in user and kernel space. SIGPROF is delivered upon expiration.

18.11 sleep(3), pause(2), and sigsuspend(2)

18.11.1 sleep(3)

```
SYNOPSIS
       #include <unistd.h>

       unsigned int sleep(unsigned int seconds);

DESCRIPTION
       sleep()  makes  the  current  process  sleep until seconds seconds have
       elapsed or a signal arrives which is not ignored.

RETURN VALUE
       Zero if the requested time has elapsed, or the number of  seconds  left
       to sleep.
```

18.11.2 pause(2)

```
       #include <unistd.h>

       int pause(void);

DESCRIPTION
       The  pause  library function causes the invoking process (or thread) to
       sleep until a signal is received that either terminates it or causes it
       to call a signal-catching function.
```

18.11.3 sigsuspend(2)

```
       #include <signal.h>

       int sigsuspend(const sigset_t *mask);

DESCRIPTION
       The  sigsuspend call temporarily replaces the signal mask for the pro-
       cess with that given by mask and then suspends  the  process  until  a
       signal is received.


RETURN VALUE
       The function sigsuspend always returns -1, normally with the error
       EINTR.
```

18.12 Unwanted Interactions: sleep(3)

Flying at different levels can be dangerous: when you tweak system-dependent features, you may upset other implementations: The Solaris `sleep(3)` function uses the interval timer. You can't use both. You can use `pause(2)`.

`pause(2)` means "wait for a signal"

18.13 Mid(end?)term

The midterm will be Monday, May 25th.

18.13.1 Style

My usual midterm style is short-answer questions for breadth and a couple of longer ones to demonstrate depth of knowledge. I try to emphasize understanding over rote knowledge, but bear in mind that in a field such as this two are hard to separate.

18.13.2 Resources

- Any necessary system call prototypes will be listed on the exam. (Unnecessary ones, too.)

18.14 Mid(end?)term

The midterm will be Monday, May 25th.

18.14.1 Style

My usual midterm style is short-answer questions for breadth and a couple of longer ones to demonstrate depth of knowledge. I try to emphasize understanding over rote knowledge, but bear in mind that in a field such as this two are hard to separate.

18.14.2 Resources

- Any necessary system call prototypes will be listed on the exam. (Unnecessary ones, too.)

18.14.3 Material

- The C Programming Language (background noise)
  - The Basics: Type system, flow control, calling protocols, etc.
  - Structures and types
  - Memory management: malloc() and free() and traversing dynamic data structures.
  - Preprocessor controls, headers, etc.
- Unix Systems Programming
  - System architecture:
    - identity (users and groups (uid, gid, euid, egid))
    - programs
    - processes
    - the filesystem
    - IO (buffered vs. not and the issues involved)
    - ipc
    - system calls
  - The Unix Family: (Chapter 2 stuff)
  - Unbuffered IO – the basic system calls and how they work
    - open, close, creat, read, write, lseek
    - dup, dup2
    - fcntl() and ioctl()
  - The filesystem: Directories and structure
    - Structure of the filesystem (inodes and whatnot)
    - opendir(), readdir(), etc.
    - the stats
    - link, unlink, symlink, readlink
    - identity
    - How times work

18.15 Mid(end?)term

The midterm will be Monday, May 25th.

18.15.1 Style

My usual midterm style is short-answer questions for breadth and a couple of longer ones to demonstrate depth of knowledge. I try to emphasize understanding over rote knowledge, but bear in mind that in a field such as this two are hard to separate.

18.15.2 Resources

- Any necessary system call prototypes will be listed on the exam. (Unnecessary ones, too.)

18.15.3 Material

- The C Programming Language (background noise)
  - The Basics: Type system, flow control, calling protocols, etc.
  - Structures and types
  - Memory management: malloc() and free() and traversing dynamic data structures.
  - Preprocessor controls, headers, etc.
- Unix Systems Programming
  - System architecture:
    - identity (users and groups (uid, gid, euid, egid))
    - programs
    - processes
    - the filesystem
    - IO (buffered vs. not and the issues involved)
    - ipc
    - system calls
  - The Unix Family: (Chapter 2 stuff)
  - Unbuffered IO – the basic system calls and how they work
    - open, close, creat, read, write, lseek
    - dup, dup2
    - fcntl() and ioctl()
  - The filesystem: Directories and structure
    - Structure of the filesystem (inodes and whatnot)
    - opendir(), readdir(), etc.
    - the stats
    - link, unlink, symlink, readlink
    - identity
    - How times work

19 Lecture: Terminal IO Management

19.1 Announcements

- Coming attractions:

Event	Subject	Due Date	Notes
asgn6	shell	Fri Dec 8 23:59	

  Use your own discretion with respect to timing/due dates.

- Be working on Asgn4 (now's the chance to make it good.)
- `/bin/pwd` under linux calls `getcwd(3)` which reads the `/proc` filesystem

56 submitted.17 failed to build

19.1.1 Example: A simple clock

See the code in figure 79.

```
#include<stdio.h>
#include<stdlib.h>
#include<signal.h>
#include<sys/time.h>

void ticker(int signum) {
  static int num=0;
  if( ++num%2 )
    printf("Tick\n");
  else                                                    10
    printf("Tock\n");
}

int main(int argc, char *argv[]) {
  struct sigaction sa;
  struct itimerval itv;

  /* first set up the handler */
  sa.sa_handler = ticker;
  sigemptyset(&sa.sa_mask);                               20
  sa.sa_flags = 0;
  if ( -1 == sigaction(SIGALRM, &sa, NULL)) {
    perror("sigaction");
    exit(1);
  }

  /* then set up the timer: one signal/second */
  itv.it_interval.tv_sec  = 1;
  itv.it_interval.tv_usec = 0;
  itv.it_value.tv_sec  = 1;                               30
  itv.it_value.tv_usec = 0;
  if ( -1 == setitimer(ITIMER_REAL, &itv, NULL)) {
    perror("setitimer");
    exit(1);
  }

  /* now await developments */
  for(;;)
    pause();
                                                          40

  return 0;
}
```

Figure 79: A simple clock

19.2 Terminal IO Managament

We talk about "The Terminal" a lot, but what is one?

> For the purposes here, a terminal is any interactive IO device from an xterm to a real tty

19.2.1 Why do we care?

Stevens Chapter 18.

Terminal processing is messy because of the amount of variability in terminals:

- does it have lowercase?
- can it back up (backspace)?
- can it scroll up?
- does it support tabs?
- etc.

Most IO is processed in CANONICAL mode: Input is assembled into lines and returned at that point. There is a limit to the line size (`MAX_CANON`) that can be read in Canonical mode. (usu. beeps when filled) In addition:

- (usually) characters are echoed
- (usually) backspaces are honored
- (usually) signals are generated from special characters

This corresponds to *cooked* mode.

19.2.2 ioctl() — the kitchen sink

ioctl() is the traditional catchall for device manipulation. Its semantics are defined by each particular device driver.

```
#include <sys/ioctl.h>

int ioctl(int d, int request, ...)

[The  "third"  argument  is  traditionally char *argp, and
will be so named for this discussion.]
```

19.2.3 termios — ioctl() rationalized

```
#include <termios.h>
#include <unistd.h>

int tcgetattr ( int fd, struct termios *termios_p );
int tcsetattr ( int fd, int optional_actions,
                        struct termios *termios_p );
int tcdrain ( int fd );
int tcflush ( int fd, int queue_selector );
```

struct termios has:

```
        tcflag_t c_iflag;      /* input modes */
        tcflag_t c_oflag;      /* output modes */
        tcflag_t c_cflag;      /* control modes */
        tcflag_t c_lflag;      /* local modes */
        cc_t c_cc[NCCS];       /* control chars */
```

`tcdrain()` waits until all output written to the object referred to by fd has been transmitted.

`tcflush()` discards data written to the object referred to by fd but not transmitted, or data received but not read, depending on the value of queue_selector:

`TCIFLUSH`	flushes data received but not read.
`TCOFLUSH`	flushes data written but not transmitted.
`TCIOFLUSH`	flushes both data received but not read, and data written but not transmitted.

19.2.4 The `tcsetattr()` action flags

tcsetattr() action flags:

TCSANOW	the change occurs immediately.
TCSADRAIN	the change occurs after all output written to fd has been transmitted. This function should be used when changing parameters that affect output.
TCSAFLUSH	the change occurs after all output written to the object referred by fd has been transmitted, and all input that has been received but not read will be discarded before the change is made.

19.2.5 The termios flag sets

There are an improbable number of flags, (see Chapter 18 or the man page) but manipulation of them is the same. `tcflag_t` is a bitfield and can be manipulated as usual.

See also table 11.3 in Stevens.

c_iflag input flags — things on the input line (e.g. `IGNPAR`, ignore parity).

c_oflag output flags — things on the output line (e.g. `OLCUC`, map lowercase to uppercase)

c_cflag control flags — control issues (modem, e.g.) (e.g. `PARODD`, expect odd parity (else even))

c_lflag local flags — local processing. (e.g.

`ECHO`	—	enable echoing or not
`ICANON`	—	Canonical Mode or not
`ISIG`	—	Generate Signals, or not

19.2.6 The termios character array

The `c_cc` array allows the program to access (and set) many of the special characters used by the tty line driver. This includes such characters as `ERASE`, the delete character, or `INTR`, the character that generates a `SIGINT`.

`NL` and `CR` cannot be changed.

19.3 Non-canonical IO

What you really need to know:

First turn off the `ICANON` bit. Now what?

Now the system won't buffer lines, but we don't really want `read()` to return once for each byte. (remember?)

The solution: Two more entries in `c_cc`:

`c_cc[VMIN]`	—	Minimum amount of data to return
`c_cc[VTIME]`	—	How long to wait ($\times 0.1s$)

`TIME` of zero means wait forever.

	MIN > 0	MIN = 0
TIME > 0	TIME is an interbyte timer; Returns when either MIN characters have been read, or TIME has elapsed between characters. Caller can block indefinitely.	TIME is a read timer Returns when TIME has elapsed.
TIME = 0	returns when MIN characters have been read Caller can block indefinitely	returns immediately.

19.4 Termios Example: turning off echoing

Figures 80, 81, and 82, and 83 show a program that turns off echoing and copies its input to its output. This program is still in canonical mode.

19.5 AnOTHer fUn eXAMPLe using non-canonical IO

Figure 84 shows another example where the case of echoed letters is perturbed randomly. This uses non-canonical IO so the weirdness is visible in real time. (There's really no end to the fun this kind of thing can produce.)

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <termios.h>

void echo_off(int fd);
void echo_on(int fd);
void cat (FILE *in, FILE *out);


int main(int argc, char *argv[]){

  echo_off(fileno(stdin));
  cat(stdin,stdout);
  echo_on(fileno(stdin));

  return 0;
}
```

Figure 80: A `cat` without echoing: `main()`

```
void echo_off(int fd) {
  struct termios tio;

  tcgetattr(fd, &tio);          /* get the current values */

  tio.c_lflag &= ~ECHO;         /* unset the ECHO bit */

  tcsetattr(fd, TCSADRAIN, &tio); /* make new values current */
}
```

Figure 81: A `cat` without echoing: `echo_off()`

```
void echo_on(int fd) {
  struct termios tio;

  tcgetattr(fd, &tio);          /* get the current values */

  tio.c_lflag |= ECHO;          /* set the ECHO bit */

  tcsetattr(fd, TCSADRAIN, &tio); /* make new values current */

}
```

Figure 82: A `cat` without echoing: `echo_on()`

```
void cat (FILE *in, FILE *out) {
  /* copy the infile to the outfile */
  int c;

  while ( (c=getc(in)) != EOF )
    putc(c,out);

}
```

Figure 83: A `cat` without echoing: `cat()`

19.6 Example: expanding tabs

For this class, most of what you're going to be interested in is in the local flags, but the input and output flags can be very interesting. Consider the following:

One of the fields in the output flags (`c_oflag`) is `TABDLY` to control the handling of tabs. Possible values are `TAB0`, `TAB1`, `TAB2`, and `XTABS`.

Why delay? Consider a real teletype. The head has to move.

`XTABS` causes it to expand tabs to spaces. Using this, we can write `expand` in such a way that we don't have to do the work. The program is in Figures 85, and 86 (the `cat()` function Figure 87 is the same as above).

19.7 But it only works on ttys

Look at Figure 88. Isn't that disappointing?

You can try some tty-specific thing and look for `ENOTTY` in `errno`, or

You can check with `isatty()`: `int isatty ( int desc );`

```c
#include <ctype.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <termios.h>

#define CTRL_D 4

int upcase_some(int c) {
    c = tolower(c);
    if (random() > RAND_MAX/2)
        c = toupper(c);
    return c;
}

int main() {
    struct termios old, new;
    int c;

    /* get the attributes */
    if ( -1 == tcgetattr(STDIN_FILENO, &old) ) {
        perror("stdin");
        exit ( -1 );
    }

    new = old;
    new.c_lflag &= ~ECHO;
    new.c_lflag &= ~ICANON;
    new.c_cc[VMIN] = 1;
    new.c_cc[VTIME] = 0;
    if ( -1 == tcsetattr(STDIN_FILENO, TCSADRAIN, &new) ) {
        perror("stdin");
        exit ( -1 );
    }

    /* because we're not in canonical mode we won't be able to
     * detect EOF in the normal way. We'll just check for ^D.
     */
    while (CTRL_D != (c=getchar()))
        putchar(upcase_some(c));

    if ( -1 == tcsetattr(STDIN_FILENO, TCSADRAIN, &old) ) {
        perror("stdin");
        exit ( -1 );
    }
    return 0;
}
```

Figure 84: Echo with random case

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <termios.h>

static tcflag_t set_tabs(int fd, tcflag_t tabstyle);
void cat (FILE *in, FILE *out);

int main(int argc, char *argv[]){
  tcflag_t tabs;

  tabs = set_tabs(fileno(stdout), XTABS);

  cat(stdin,stdout);

  set_tabs(fileno(stdout), tabs);

  return 0;
}
```

Figure 85: A termio-based `expand`: `main()`

```
static tcflag_t set_tabs(int fd, tcflag_t tabstyle) {
  /* Set the tab handling to the given style and return
   * the old tab handling mode.
   *   Assumes the given file descriptor is a tty, and
   * aborts the program if not.
   */
  struct termios tio;
  tcflag_t oldtabs;

  if ( tcgetattr(fd, &tio) ) {  /* get the current termio info */
    perror("tcgetattr");
    exit(-1);
  }

  oldtabs = tio.c_oflag & TABDLY; /* select current tab processing */

  tio.c_oflag &= ~TABDLY;       /* mask out current tab handling */
  tio.c_oflag |= tabstyle;      /* set to the given style        */

  /* turn on the new attributes */
  if ( tcsetattr(fd, TCSADRAIN, &tio) ) {
    perror("tcsetattr");
    exit(-1);
  }

  return oldtabs;               /* return the old version */
}
```

Figure 86: A termio-based `expand`: `SetTabs()`

```
void cat (FILE *in, FILE *out) {
  /* copy the infile to the outfile */
  int c;

  while ( (c=getc(in)) != EOF )
    putc(c,out);

}
```

Figure 87: A termio-based `expand`: `cat()`

```
% myexpand
Hello   tabbing world
Hello   tabbing world

% myexpand | tr ' ' '+'
tcgetattr: Invalid argument

% myexpand > outfile
tcgetattr: Inappropriate ioctl for device
%
```

Figure 88: termio expand only works on ttys

20 Lecture: The Midterm

Outline:
Announcements

20.1 Announcements

- Coming attractions:

Event	Subject	Due Date			Notes
asgn6	shell	Fri	Dec 8	23:59	

  Use your own discretion with respect to timing/due dates.

- The midterm was this day. I hope.

21 Lecture: Introduction to Networks

Outline:

- Announcements
- Asgn 5 and Socket Programming
- Basics
- How it works
  - Sockets
- UDP
  - A note on addresses
  - The functions
  - Example: Passing messages with UDP
- TCP
  - Details
  - Example: Passing messages with TCP
- A real chat system
  - Choosing between I/O streams: select(2) and poll(2)
  - The Client
  - The Server
- Add content to this and to asgn5 writeup
- handin directories
- libraries to CSL

Outline:

- Announcements
- Asgn 5 and Socket Programming
- Basics
- How it works
  - Sockets
- UDP
  - A note on addresses
  - The functions
  - Example: Passing messages with UDP
- TCP
  - Details
  - Example: Passing messages with TCP
- A real chat system
  - Choosing between I/O streams: select(2) and poll(2)
  - The Client
  - The Server

21.1 Announcements

- Coming attractions:

Event	Subject	Due Date			Notes
asgn6	shell	Fri	Dec 8	23:59	

  Use your own discretion with respect to timing/due dates.

- Be working on Asgn4 (now's the chance to make it good.)
- `/bin/pwd` under linux calls `getcwd(3)` which reads the `/proc` filesystem

21.2 Asgn 5 and Socket Programming

This is APUE Chapter 16. (This is not cpe464.)

So we have a network, how does one connect to it?

21.3 Basics

- A network connects two computers
- Communications are governed by protocols
- To talk to another machine you need to be able to name it (address)
- To talk to a particular program on another machine you need to be able to name it (port)
- These functions look a little hairy, but they're not as bad as they look. Keep your man pages handy for structures.
- We will be using IPV4
- The unix*N* servers can talk to each other, but not off campus
- This just scratches the surface (take 464)

21.4 How it works

21.4.1 Sockets

- Invented by BSD as a network endpoint
- Created with `socket(2)`

```
#include <sys/types.h>          /* See NOTES */
#include <sys/socket.h>

int socket(int domain, int type, int protocol);
```

You will want `AF_INET` and either
`SOCK_STREAM` TCP (Transmssion Control Protocol), stream oriented
`SOCK_DGRAM` UDP (User Datagram Protocol), datagram oriented
`protocol` is a sub-protocol, and in this case you'll want it to be 0.
Return value is a file descriptor.

21.5 UDP

Client	Server
`socket(2)`	`socket(2)`
	`bind(2)`
`recv_from(2)`	`send_to(2)`
`send_to(2)`	`recv_from(2)`
`close(2)`	`close(2)`

- UDP is **connectionless**
- UDP is **unreliable**. You'll never know if it got there at all.
- Create a socket, and send a message.
- An un-bound socket is issued a binding on the way out so replies are possible.

21.5.1 A note on addresses

To talk to something you have to be able to name it.

- Both of these protocols use an **IP address** and a **port** to identify an endpoint.
- Many of these functions take a `struct sockaddr *` as a parameter.
- `struct sockaddr` is really a family. The Internet version is:

```
struct sockaddr_in {
    sa_family_t    sin_family; /* address family: AF_INET */
    in_port_t      sin_port;   /* port in network byte order */
    struct in_addr sin_addr;   /* internet address */
};
```

```
/* Internet address. */
struct in_addr {
    uint32_t       s_addr;     /* address in network byte order */
};
```

- man `ip(7)` for IPv4 structures
- What if you don't know the address: `gethostbyname(3)` or `getaddrinfo(3)`
- What if you have an address and want to know who it belongs to? `gethostbyaddr(3)` or `getnameinfo(3)` .
- What if you don't care? `INADDR_ANY`

Address Translation

```
int getaddrinfo(const char *node, const char *service,
                const struct addrinfo *hints,
                struct addrinfo **res);

void freeaddrinfo(struct addrinfo *res);

    struct addrinfo {
        int              ai_flags;
        int              ai_family;
        int              ai_socktype;
        int              ai_protocol;
        size_t           ai_addrlen;
        struct sockaddr *ai_addr;
        char            *ai_canonname;
        struct addrinfo *ai_next;
    };
```

```
uint32_t getaddress(const char *hostname) {
  /* return the IPv4 address for the given host, or 0
   * Address is in network order   */
  struct addrinfo *ai,hints;
  uint32_t res=0;
  int rvalue;

  memset(&hints,0,sizeof(hints));
  hints.ai_family = AF_INET;
  if ( 0 == (rvalue=getaddrinfo(hostname,NULL,&hints,&ai))) {
    if ( ai )
      res = ((struct sockaddr_in*)ai->ai_addr)->sin_addr.s_addr;
    freeaddrinfo(ai);
  } else {
    fprintf(stderr,"%s:%s\n", hostname, gai_strerror(rvalue));
  }
  return res;
}
```

21.5.2 The functions

socket(2): create a socket

```
int socket(int domain, int type, int protocol);
```

You will want `AF_INET` and either
`SOCK_STREAM` TCP (Transmission Control Protocol), stream oriented
`SOCK_DGRAM` UDP (User Datagram Protocol), datagram oriented
`protocol` is a sub-protocol, and in this case you'll want it to be 0.
Return value is a file descriptor, or -1 on error.

bind(2): Attach an address to a socket

```
int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
```

getsockname(2)

```
int getsockname(int sockfd, struct sockaddr *addr, socklen_t *addrlen);

getsockname()  returns  the  current  address  to which the socket
sockfd is bound
```

inet_ntop(3)

```
const char *inet_ntop(int af, const void *src,char *dst, socklen_t size);

This function converts the network address structure src in the af
address  family  into a character string.  The resulting string is
copied to the buffer pointed to by dst, which must be  a  non-NULL
pointer.   The  caller  specifies the number of bytes available in
this buffer in the argument size.
```

(`inet_pton(3)` does it backwards)

recvfrom(2)

```
ssize_t recvfrom(int sockfd, void *buf, size_t len, int flags,
                 struct sockaddr *src_addr, socklen_t *addrlen);
```

Returns bytes received.
`INADDR_ANY` is good to know about.

sendto(2)

```
ssize_t sendto(int sockfd, const void *buf, size_t len, int flags,
               const struct sockaddr *dest_addr, socklen_t addrlen);
```

Returns bytes received.
`send(2)`, `recv(2)`, `sendto(2)` and `recvfrom(2)` are just like `read(2)`, and `write(2)` except for the `flags` field. (e.g.`MSG_DONTWAIT`)

21.5.3 Example: Passing messages with UDP

Figures 89 and 90 show the two halves of a UDP conversation.

21.6 TCP

TCP provides a reliable transport mechanism between two processes

- it is connection-oriented. That is, a connection has to be established before it can be used to communicate.
- It is reliable
- It **does not** preserve boundaries between messages.
- It's almost exactly like a pipe, except typically one uses `send(2)` and `recv(2)` rather than `read(2)` and `write(2)`

Client	Server
socket(2)	socket(2)
	bind(2)
	listen(2)
connect(2)	accept(2)
	getsockname(2)
send(2)	recv(2)
recv(2)	send(2)
close(2)	close(2)

```
#include <arpa/inet.h>
#include <getopt.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <unistd.h>

#define MESG "Hello, Client\n"
#define MAXLEN 1000

int main(int argc, char *argv[]) {
  int sockfd;
  struct sockaddr_in sa;
  socklen_t len;
  char localaddr[INET_ADDRSTRLEN];
  char buff[MAXLEN+1];
  int mlen,port = 10000;
  socklen_t slen;

  /* Create the socket */
  sockfd = socket(AF_INET, SOCK_DGRAM, 0);

  sa.sin_family      = AF_INET;
  sa.sin_port        = htons(port);     /* use our port  */
  sa.sin_addr.s_addr = htonl(INADDR_ANY);   /* all local interfaces */

  /* bind */
  bind(sockfd, (struct sockaddr *)&sa, sizeof(sa));

  /* receive */
  slen = sizeof(sa);
  mlen = recvfrom(sockfd, buff, sizeof(buff),0,(struct sockaddr *)&sa,&slen);

  /* who is this guy? */
  inet_ntop(AF_INET, &sa.sin_addr.s_addr,localaddr, sizeof(localaddr));
  printf("Message from <%s:%d>:  ",localaddr,ntohs(sa.sin_port));
  fflush(stdout);

  write(STDOUT_FILENO,buff,mlen);

  mlen = strlen(MESG);
  len=sendto(sockfd, MESG, mlen, 0, (struct sockaddr *)&sa, sizeof(sa));

  close(sockfd);  /* all done, clean up */
  return 0;
}
```

Figure 89: UDP Server

```
#include <arpa/inet.h>
#include <getopt.h>
#include <netdb.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <unistd.h>

#define MESG "Hello, Server\n"
#define MAXLEN 1000
int main(int argc, char *argv[]) {
  int sockfd;                   /* socket descriptors */
  struct sockaddr_in sa;
  struct hostent  *hostent;
  char buff[MAXLEN+1];
  int len,mlen;
  socklen_t slen;

  int port = 10000;
  char *hostname = "localhost";

  /* figure out who we're talking to */
  hostent = gethostbyname(hostname);

  /* Create the socket */
  sockfd = socket(AF_INET, SOCK_DGRAM, 0);

  /* connect it (the AP address is already network ordered) */
  sa.sin_family      = AF_INET;
  sa.sin_port        = htons(port); /* use our port  */
  sa.sin_addr.s_addr = *(uint32_t*)hostent->h_addr_list[0];

  mlen = strlen(MESG);
  len=sendto(sockfd, MESG, mlen, 0, (struct sockaddr *)&sa, sizeof(sa));

  sa.sin_family      = AF_INET;
  sa.sin_port        = htons(port);     /* use our port  */
  sa.sin_addr.s_addr = htonl(INADDR_ANY);   /* all local interfaces */

  slen = sizeof(sa);
  len = recvfrom(sockfd, buff, sizeof(buff),0,(struct sockaddr *)&sa,&slen);
  write(STDOUT_FILENO,buff,len);

  close(sockfd);
  return 0;
}
```

Figure 90: UDP Client

21.6.1 Details

Also uses:

- `socket(2)` to create a socket
- `bind(2)` to attach an address to a socket

and may use `getsockname(2)`, `getpeerinfo(2)`, and `inet_ntop(2)` to find and report info about the other end.

listen(2): wait for someone to `connect()`

This is the server-side call. Wait for something to happen.

```
int listen(int sockfd, int backlog);
```

backlog is how many connections to allow to be pending before "Connection Refused"

connect(2): try to talk to a `listen(2)`ing socket

```
int connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
```

accept(2): complete the connection

```
int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
```

Important: this returns a new socket. The addr is info on our peer. The original listening socket is unaffected.

getsockname(2)

```
int getsockname(int sockfd, struct sockaddr *addr, socklen_t *addrlen);

getsockname()  returns  the  current  address  to which the socket
sockfd is bound
```

send(2)

```
ssize_t send(int sockfd, const void *buf, size_t len, int flags);
```

recv(2)

```
ssize_t recv(int sockfd, void *buf, size_t len, int flags);
```

21.6.2 Example: Passing messages with TCP

```
#include <arpa/inet.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <unistd.h>
#define DEFAULT_BACKLOG 100
#define MESG "Hello, TCP Client.\n"
#define MAXLEN 1000

int main(int argc, char *argv[]) {
  int mlen,sockfd, newsock,port = 10000;
  struct sockaddr_in sa, newsockinfo, peerinfo;
  socklen_t len;
  char localaddr[INET_ADDRSTRLEN], peeraddr[INET_ADDRSTRLEN], buff[MAXLEN+1];

  sockfd = socket(AF_INET, SOCK_STREAM, 0);  /* Create the socket */

  sa.sin_family        = AF_INET;
  sa.sin_port          = htons(port);       /* use our port  */
  sa.sin_addr.s_addr   = htonl(INADDR_ANY); /* all local interfaces */
  bind(sockfd, (struct sockaddr *)&sa, sizeof(sa));

  listen(sockfd,DEFAULT_BACKLOG);  /* listen */

  len = sizeof(newsockinfo);    /* accept */
  newsock = accept(sockfd, (struct sockaddr *)&peerinfo, &len);

  /* now we're in business, I suppose */
  len = sizeof(newsockinfo);
  getsockname(newsock, (struct sockaddr *) &newsockinfo, &len);

  /* find out about the new socket and the other end */
  inet_ntop(AF_INET, &newsockinfo.sin_addr.s_addr,localaddr, sizeof(localaddr));
  inet_ntop(AF_INET, &peerinfo.sin_addr.s_addr,peeraddr, sizeof(peeraddr));

  printf("New Connection:  %s:%d->%s:%d\n",peeraddr,ntohs(peerinfo.sin_port),
                              localaddr,ntohs(newsockinfo.sin_port));

  /* pass some data receive, then send*/
  mlen = recv(newsock, buff, sizeof(buff), 0);
  write(STDOUT_FILENO,buff,mlen);
  mlen = send(newsock, MESG, strlen(MESG), 0);

  close(sockfd);               /* all done, clean up */
  close(newsock);
  return 0;
}
```

Figure 91: TCP Server

```
#include <arpa/inet.h>
#include <getopt.h>
#include <netdb.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <unistd.h>

#define MESG "Hello, TCP Server.\n"
#define MAXLEN 1000

int main(int argc, char *argv[]) {
  int port,len;
  int sockfd;                   /* socket descriptors */
  struct sockaddr_in sa;
  struct hostent  *hostent;
  const char *hostname;
  char buff[MAXLEN+1];

  port = 10000;
  hostname = "localhost";

  /* figure out who we're talking to */
  hostent = gethostbyname(hostname);

  /* Create the socket */
  sockfd = socket(AF_INET, SOCK_STREAM, 0);

  /* connect it */
  sa.sin_family       = AF_INET;
  sa.sin_port         = htons(port);              /* use our port  */
  sa.sin_addr.s_addr  = *(uint32_t*)hostent->h_addr_list[0]; /* net order */

  connect(sockfd, (struct sockaddr *)&sa, sizeof(sa));

  len = send(sockfd, MESG, strlen(MESG), 0);

  len = recv(sockfd, buff, sizeof(buff), 0);
  write(STDOUT_FILENO,buff,len);

  close(sockfd);                /* clean up and go home */

  return 0;
}
```

Figure 92: TCP Client

21.7 A real chat system

These programs represent a bare-bones TCP-based chat system. These have no error-checking, which is terrifying, but they'll fit on a page this way.

21.7.1 Choosing between I/O streams: select(2) and poll(2)

What if you have multiple potential sources of input or output and don't know which one to service next. You have two options:

- Polling using nonblocking IO
- `Select(2)` or `poll(2)`
-

`Select(2)` or `poll(2)` is a better approach.

`select(2)`

```
int select(int nfds, fd_set *readfds, fd_set *writefds,
           fd_set *exceptfds, struct timeval *timeout);

void FD_CLR(int fd, fd_set *set);
int  FD_ISSET(int fd, fd_set *set);
void FD_SET(int fd, fd_set *set);
void FD_ZERO(fd_set *set);
```

`poll(2)`

```
int poll(struct pollfd *fds, nfds_t nfds, int timeout);

    struct pollfd {
        int   fd;         /* file descriptor */
        short events;     /* requested events */
        short revents;    /* returned events */
    };

POLLIN There is data to read.
```

21.7.2 The Client

It's easier to talk about the client first, so we will. It establishes a connection with the waiting server and then relays messages from the console to the server or the server to the console until the message is "bye".

Two parts:

- `main()` (figure 93) sets up the connection
- `chat()` (figure 94) handles the actual conversation.

21.7.3 The Server

The server is a little more complicated.

- `main()` (figure 97) sets up the connection a listening socket then waits.
- `chat()` (figure 98)
  - accepts connections on the listener
  - accepts input from any client and forwards it on to all other clients.

```
#include <arpa/inet.h>
#include <getopt.h>
#include <netdb.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <sys/select.h>
#include <unistd.h>

#define MAX_CONNECTIONS 100
#define MAXLINE 1024
void chatclient(int sockfd);

int main(int argc, char *argv[]) {
  int sockfd;                   /* socket descriptors */
  struct sockaddr_in sa;
  struct hostent  *hostent;
  int port;
  char *hostname = "localhost";

  if ( argc==2 )
    port = atoi(argv[1]);
  else
    port = 10000;

  /* figure out who we're talking to */
  hostent = gethostbyname(hostname);

  /* Create the socket */
  sockfd = socket(AF_INET, SOCK_STREAM, 0);

  /* connect it */
  sa.sin_family = AF_INET;
  sa.sin_port   = htons(port);              /* use our port  */
  sa.sin_addr.s_addr   = *(uint32_t*)hostent->h_addr_list[0]; /* net order */

  connect(sockfd, (struct sockaddr *)&sa, sizeof(sa));

  chatclient(sockfd);

  /* send some data */
  close(sockfd);
  return 0;
}
```

Figure 93: MiniChat client, `main()`

```
void chatclient(int sockfd) {
  int done, len, mlen;
  fd_set readfds;
  char buff[MAXLINE];
  socklen_t slen;

  done = 0;
  do {
    FD_ZERO(&readfds);
    FD_SET(STDIN_FILENO,&readfds);
    FD_SET(sockfd,&readfds);
    select(sockfd+1,&readfds,NULL,NULL,NULL);

    if (FD_ISSET(STDIN_FILENO,&readfds)) {
      /* read a line and send it to remote */
      len = read(STDIN_FILENO,buff,MAXLINE);
      mlen=send(sockfd, buff, len, 0);
    }
    if ( FD_ISSET(sockfd,&readfds)) {
      /* receive a line and print it */
      slen = sizeof(struct sockaddr_in);
      mlen = recv(sockfd, buff, sizeof(buff),0);

      write(STDOUT_FILENO,buff,mlen);
    }
    if ( !strncmp(buff,"bye",3) )
      done = 1;
  } while (!done);
}
```

Figure 94: MiniChat client, `chat()`

```
#include "minichat.h"
#include <poll.h>

#define LOCAL  0
#define REMOTE (LOCAL + 1)

void chatclient(int sockfd) {
  int done, len, mlen;
  char buff[MAXLINE];
  struct pollfd fds[REMOTE+1];

  fds[LOCAL].fd = STDIN_FILENO;
  fds[LOCAL].events = POLLIN;
  fds[LOCAL].revents = 0;
  fds[REMOTE]=fds[LOCAL];
  fds[REMOTE].fd = sockfd;

  done = 0;
  do {

    poll(fds,sizeof(fds)/sizeof(struct pollfd),-1); /* negative->wait forever */
    if (fds[LOCAL].revents & POLLIN ) {
      /* read a line and send it to remote */
      len = read(STDIN_FILENO,buff,MAXLINE);
      mlen=send(sockfd, buff, len, 0);
    }
    if (fds[REMOTE].revents & POLLIN ) {
      /* receive a line and print it */
      mlen = recv(sockfd, buff, sizeof(buff),0);

      write(STDOUT_FILENO,buff,mlen);
    }
    if ( !strncmp(buff,"bye",3) )
      done = 1;
  } while (!done);
}
```

Figure 95: MiniChat client, `chat()`, `poll(2)` version

```
#include "minichat.h"
#include <poll.h>

#define LISTENER 0

void chatserver(int listener) {
    int i, j, done, mlen, newsock, num = 0;
    struct sockaddr_in peerinfo;
    socklen_t slen;
    char buff[MAXLINE], addr[INET_ADDRSTRLEN];
    struct pollfd fds[MAX_CONNECTIONS + 1];
    done = 0;

    buff[0] = '\0';  /* Make sure this is a string for strncmp below */
    fds[LISTENER].fd = listener;
    fds[LISTENER].events = POLLIN;
    fds[LISTENER].revents = 0;
    num = 1;
    do {
        poll(fds, num, -1);
        /* check for connections */
        if ((fds[LISTENER].revents & POLLIN) && (num <= MAX_CONNECTIONS)) {
            /* accept a new connection and add the client to the list */
            slen = sizeof(peerinfo);
            newsock = accept(listener, (struct sockaddr *)&peerinfo, &slen);
            inet_ntop(AF_INET, &peerinfo.sin_addr.s_addr, addr, sizeof(addr));
            printf("New connection from:  %s:%d\n", addr, htons(peerinfo.sin_port));
            fds[num].fd = newsock;
            fds[num].events = POLLIN;
            fds[num].revents = 0;
            num++;
        }
        /* check for data */
        for (i = 1; i < num; i++) {
            if (fds[i].revents & POLLIN) {
                /* read from this client and broadcast to all */
                slen = sizeof(struct sockaddr_in);
                mlen = recv(fds[i].fd, buff, sizeof(buff), 0);
                for (j = 1; j < num; j++) /* broadcast to everyone else*/
                    if (i != j)             /* not self */
                        send(fds[j].fd, buff, mlen, 0);
            }
        }
        if (!strncmp(buff, "bye", 3))
            done = 1;
    } while (!done);
}
```

Figure 96: MiniChat client, `chat()`, `poll(2)` version

```
#include <arpa/inet.h>
#include <getopt.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/select.h>
#include <sys/types.h>
#include <unistd.h>

#define MAX_CONNECTIONS 100
#define DEFAULT_BACKLOG MAX_CONNECTIONS
#define MAX_ADDR 100
#define MAXLINE 1024
void chatserver(int sockfd);

int main(int argc, char *argv[]) {
  int sockfd;             /* socket descriptors */
  struct sockaddr_in sa;
  int port;

  if ( argc==2 )
    port = atoi(argv[1]);
  else
    port = 10000;

  /* Create the socket */
  sockfd = socket(AF_INET, SOCK_STREAM, 0);

  /* bind it to our address*/
  sa.sin_family = AF_INET;
  sa.sin_port   = htons(port);     /* use our port  */
  sa.sin_addr.s_addr   = htonl(INADDR_ANY);   /* all local interfaces */

  bind(sockfd, (struct sockaddr *)&sa, sizeof(sa));

  listen(sockfd,DEFAULT_BACKLOG);

  chatserver(sockfd);  /* accept connections and chat */

  /* all done, clean up */
  close(sockfd);
  return 0;
}
```

Figure 97: MiniChat server, `main()`

```
void chatserver(int listener) {
  int clients[MAX_CONNECTIONS];
  int max,i,j, done, mlen, newsock, num_clients = 0;
  fd_set readfds;
  struct sockaddr_in peerinfo;
  socklen_t slen;
  char buff[MAXLINE],addr[INET_ADDRSTRLEN];

  done = 0;
  do {
    FD_ZERO(&readfds);
    FD_SET(listener,&readfds);
    max = listener;
    for(i=0;i<num_clients; i++) {
      FD_SET(clients[i],&readfds);
      if ( clients[i] > max )
        max = clients[i];
    }
    select(max+1,&readfds,NULL,NULL,NULL); /* wait for it... */

    if (FD_ISSET(listener,&readfds) && ( num_clients < MAX_CONNECTIONS - 1 )) {
      /* accept a new connection and add the client to the list */
      slen = sizeof(peerinfo);
      newsock = accept(listener, (struct sockaddr *)&peerinfo, &slen);

      inet_ntop(AF_INET, &peerinfo.sin_addr.s_addr,addr, sizeof(addr));
      printf("New connection from:  %s:%d\n", addr, htons(peerinfo.sin_port));
      clients[num_clients++] = newsock;
    } else {
      for(i=0;i<num_clients;i++) {
        if ( FD_ISSET(clients[i],&readfds) ) {
          /* read from this client and broadcast to all */
          slen = sizeof(struct sockaddr_in);
          mlen = recv(clients[i], buff, sizeof(buff),0);

          for(j=0;j<num_clients;j++)          /* broadcast to everyone else*/
            if ( i != j )       /* not self */
              send(clients[j], buff, mlen, 0);
        }
      }
    }
    if ( !strncmp(buff,"bye",3) )
      done = 1;
  } while (!done);
}
```

Figure 98: MiniChat server, `chat()`

22 Lecture: Process Life Cycle

Outline:

- Announcements
- The Life-cycle of a Unix Process
  - Where they come from?
  - Where they go?
  - Reaping a process: `wait()`
  - Parent vs. Child
  - Process Groups
  - Cleanup
  - Parent vs. Child
  - Example

22.1 Announcements

- Coming attractions:

Event	Subject	Due Date			Notes
asgn6	shell	Fri	Dec 8	23:59	

  Use your own discretion with respect to timing/due dates.

- Midterm/Plan for the final

22.2 The Life-cycle of a Unix Process

Birth	fork()
Death	termination (exit(),_exit(),return,abort(),signal)
Afterlife	reaping with wait()

22.2.1 Where they come from?

As in life, we become our parents. `fork()` creates an exact copy of the current process. Returns -1 (error) or child's pid to original process, and 0 to the child.

```
NAME
       fork - create a child process

SYNOPSIS
       #include <unistd.h>
       pid_t fork(void);
```

Most of the process environment is inherited over the `fork()`:

uid	umask
gid	signal mask
euid	environment
egid	resource limits
cwd	etc.

22.2.2 Where they go?

There are five ways for a unix process to end:

normal	return from `main()`
normal	call `exit()`
normal	call `_exit()`
abnormal	call `abort()`
abnormal	terminated by a signal

Eventually they exit. The process continues to exist as a zombie until somebody waits for it.
A waited-for process returns its status to its parent.

22.2.3 Reaping a process: `wait()`

A waited-for process returns its status to its parent.

```
NAME
       wait, waitpid - wait for process termination

SYNOPSIS
       #include <sys/types.h>
       #include <sys/wait.h>

       pid_t wait(int *status)
       pid_t waitpid(pid_t pid, int *status, int options);
```

The value of pid can be one of:

< -1	which means to wait for any child process whose process group ID is equal to the absolute value of pid.
-1	which means to wait for any child process; this is the same behaviour which wait exhibits.
0	which means to wait for any child process whose process group ID is equal to that of the calling process.
> 0	which means to wait for the child whose process ID is equal to the value of pid.

The value of options is an OR of zero or more of the following con- stants:

WNOHANG	which means to return immediately if no child has exited.
WUNTRACED	which means to also return for children which are stopped (but not traced), and whose status has not been reported. Status for traced children which are stopped is provided also without this option.

Wait returns the pid of the child that exited or -1 on error. The status parameter encodes the program's termination status: `WIFEXITED(status)`, `WEXITSTATUS(status)` among other things (e.g. `WIFSIGNALED(status)`, `WTERMSIG(status)`,)

22.2.4 Parent vs. Child

Fork preserves	Fork changes
• open file descriptors (dup()-style) • identity • current working directory • current root directory • signal mask and handlers • umask • environment • resource limits	• process id • parent process id • return value of `fork()` • pending alarms are cleared • pending signals are cleared • child's run times are reset

22.2.5 Process Groups

Since we're discussing process families, we should talk about process groups.

```
int setpgid(pid_t pid, pid_t pgid);
pid_t getpgid(pid_t pid);

setpgid  sets  the process group ID of the process specified by pid to
pgid.  If pid is zero, the process ID of the current process is  used.
If  pgid  is  zero,  the process ID of the process specified by pid is
used.  If setpgid is used to move a process from one process group  to
another (as is done by some shells when creating pipelines), both pro-
cess groups must be part of the same session.  In this case, the  pgid
specifies an existing process group to be joined and the session ID of
that group must match the session ID of the joining process.
```

```
getpgid returns the process group ID of the process specified by  pid.
If pid is zero, the process ID of the current process is used.

On success,  setpgid  and  setpgrp  return  zero.  On  error,  -1  is
returned, and errno is set appropriately.

getpgid returns a process group on success.  On error, -1 is returned,
and errno is set appropriately.
```

Of particular interest is the *foreground process group*:

- This is the process that gets terminal-generated signals
- managed by

```
pid_t tcgetpgrp(int filedes)               −1 on error
int tcsetpgrp(int filedes, pid_t pgrpid)   −1 on error
```

Process groups can be organized into *sessions,* too, but we're not going to worry about that now.

22.2.6 Cleanup

A waited-for process returns its status to its parent.

```
NAME
       wait, waitpid - wait for process termination

SYNOPSIS
       #include <sys/types.h>
       #include <sys/wait.h>

       pid_t wait(int *status)
       pid_t waitpid(pid_t pid, int *status, int options);
```

The value of pid can be one of:

< -1	which means to wait for any child process whose process group ID is equal to the absolute value of pid.
-1	which means to wait for any child process; this is the same behaviour which wait exhibits.
0	which means to wait for any child process whose process group ID is equal to that of the calling process.
> 0	which means to wait for the child whose process ID is equal to the value of pid.

The value of options is an OR of zero or more of the following con- stants:

WNOHANG	which means to return immediately if no child has exited.
WUNTRACED	which means to also return for children which are stopped (but not traced), and whose status has not been reported. Status for traced children which are stopped is provided also without this option.

Wait returns the pid of the child that exited or -1 on error. The status parameter encodes the program's termination status: `WIFEXITED(status)`, `WEXITSTATUS(status)` among other things (e.g. `WIFSIGNALED(status)`, `WTERMSIG(status)`,)

22.2.7 Parent vs. Child

Fork preserves	Fork changes
• open file descriptors (dup()-style) • identity • current working directory • current root directory • signal mask and handlers • umask • environment • resource limits	• process id • parent process id • return value of `fork()` • pending alarms are cleared • pending signals are cleared • child's run times are reset

22.2.8 Example

See the purported solution to Lab06 in Figure 99

```
#include<signal.h>
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<sys/time.h>


char *msg=NULL;

void ticker(int signum) {
  if ( msg ) {
    printf("%s", msg);
    fflush(stdout);
  }
}

void countdown(int ticks) {
  /* a two-headed ticker.  Parent does ticks, child does tocks */
  pid_t child;
  int status, inParent;
  struct sigaction sa;
  struct itimerval it;

  /* set up the sigaction */
  sa.sa_handler = ticker;
  sigemptyset(&sa.sa_mask);
  sa.sa_flags = 0;
  if ( -1 == sigaction(SIGALRM, &sa, NULL)) {
    perror("sigaction");
    exit(1);
  }

  /* set up shared parts of the timer */
  it.it_interval.tv_sec = 1;
  it.it_interval.tv_usec = 0;

  inParent = (child=fork());    /* do the fork */

  if ( -1 == child ) {
    perror("fork()");
    exit(1);
  }
  if ( inParent ) {
    it.it_value.tv_sec = 0;
    it.it_value.tv_usec = 500000;
    msg = "Tick...";
  } else {
    it.it_value.tv_sec = 1;
    it.it_value.tv_usec = 0;
    msg = "Tock.\n";
  }

  /* set the timer */
  if (-1==setitimer(ITIMER_REAL, &it, NULL)) {
    perror("setitimer");
    exit(1);
  }

  /* count off the ticks. */
  while ( ticks-- )
    pause();

  if ( inParent ) {            /* Parent waits for the child to exit. */
    if ( -1==wait(&status) )  {
      perror("wait");
      exit(1);
    }
  }
}

int main(int argc, char *argv[]) {
  int sec, err=0;
  if ( argc != 2 ) {
    fprintf(stderr, "usage:  %s seconds\n", argv[0]);
    exit (1);
  }

  sec = atoi(argv[1]); /* yes, strtol() is better.  This is board code */

  err = countdown(sec);

  return err;
}
```

Figure 99: An over-complicated countdown timer

23 Lecture: Nothing in particular

Outline:

23.1 Announcements

- Coming attractions:

Event	Subject	Due Date	Notes
asgn6	shell	Fri Dec 8 23:59	

  Use your own discretion with respect to timing/due dates.

- Remember the upcoming lab
- Remember the "mid"term
- This class was actually canceled

23.2 From last time: `fork()` and `wait()`

23.2.1 Process Creation

```
#include <unistd.h>
pid_t fork(void);
```

`fork()` creates an exact copy of the current process. Returns -1 (error) or child's pid to original process, and 0 to the child.

23.2.2 Process Termination

There are five ways for a unix process to end:

normal	return from `main()`
normal	call `exit()`
normal	call `_exit()`
abnormal	call `abort()`
abnormal	terminated by a signal

23.3 From last time: Process Creation

23.3.1 Parent vs. Child

Fork preserves	Fork changes
• open file descriptors (dup()-style) • identity • current working directory • current root directory • signal mask and handlers • umask • environment • resource limits	• process id • parent process id • return value of `fork()` • pending alarms are cleared • pending signals are cleared • child's run times are reset

23.4 The Environment of a Unix Process

One of the things we discussed as being preserved is the *environment*

- `main()` is called with argc and argv (by `crt0`, added by the linker). These args come from the kernel. `argc` is the count, `argv` is a null terminated array of pointers to strings.
- Each has its own uid, gid, pid (`getpid()`), pwd, etc.
- Every process has a parent (`getppid()`);
- and its own environment— a series of strings

  Access to the environment: Traditionally via:

  `main(int argc, char *argv[], char *env[])`

  but now, a NULL-terminated global list of strings:

  `char *environ[]`

```
SYNOPSIS
       #include <stdlib.h>

       char *getenv(const char *name);

       int setenv(const char *name, const char *value, int overwrite);

       void unsetenv(const char *name);
```

FWIW, this explains the difference between shell variables and environment variables.

23.5 Loading a new program: the exec()s

The execs are a family of functions that overlay a new program on a process:

syscall	prototype
execl	int execl(const char *path, const char *arg, ...);
execlp	int execlp(const char *file, const char *arg, ...);
execle	int execle(const char *path, const char *arg , ..., char * const envp[]);
execv	int execv(const char *path, char *const argv[]);
execvp	int execvp(const char *file, char *const argv[]);
execve	int execve (const char *filename, char *const argv [], char *const envp[]);

23.5.1 Return from `exec()`?

We say exec() never returns, but what if it does? (bad command, etc.)

23.5.2 What is in inherited over exec()?

see p234.

- pid and parent pid
- real uid and real group id
- supplementary ids
- process group id
- session id
- controlling terminal
- time left until alarm clock (**alarm() persists!**)
- current working directory
- root directory (chroot() perists)
- file mode creation mask
- file locks
- **process signal mask**
- **pending signals**
- resource limits...
- `tmp_utime`, etc. values (process time)

23.6 Summary `fork()` and `exec()`

`fork()` starts new processes `exec()` gets them to do something else. Some of the important aspects of this are summarized in Figure 100.

23.7 Origins of Processes

Boot leads to process 0, the swapper.

Process 1 is init (`/etc/init` if you're an old-timer, `/sbin/init` for newbies). Runs update, gettys, reaps orphans, etc.

	fork()	**exec()**
Preserves:	• open file descriptors (dup()-style) • identity • current working directory • current root directory • signal mask and handlers • umask • environment • resource limits	• process id • parent process id • real uid and gid • open file descriptors (dup()-style) (unless close-on-exec is set) • **pending alarms** • current working directory • current root directory • umask • environment • **signal mask** • pending signals • run time values
Changes:	• process id • parent process id • return value of `fork()` • pending alarms are cleared • interval timers are cleared • pending signals are cleared • child's run times are reset	• effective uid and gid if suid • just about everything else (entire contents of memory)

Figure 100: Some important properties of `fork()` and `exec()`

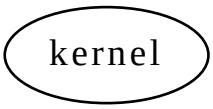


Figure 101: In the beginning there was the kernel

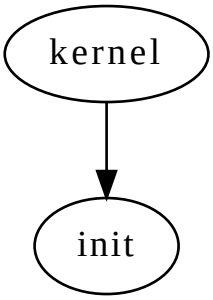


Figure 102: Then there was init

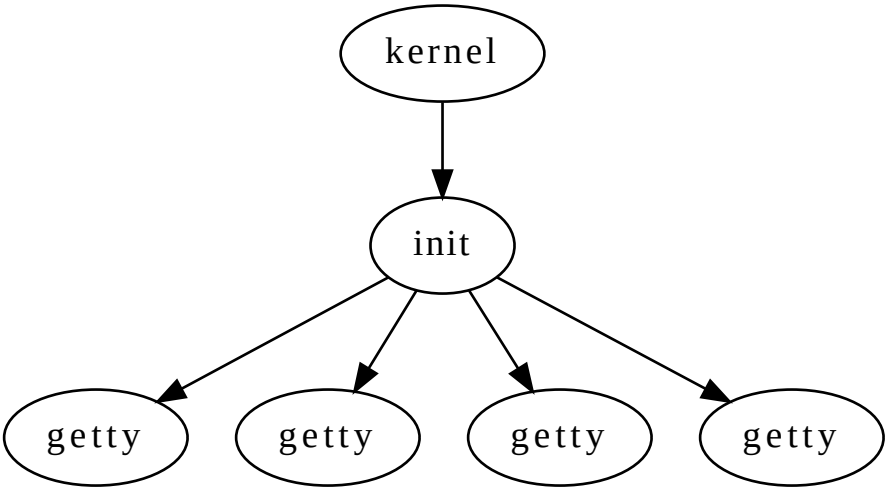


Figure 103: Launching gettys

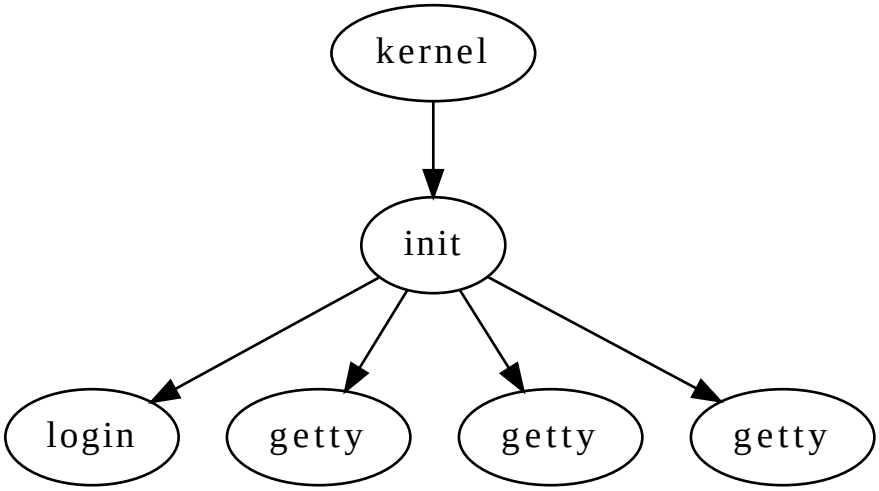


Figure 104: execing login

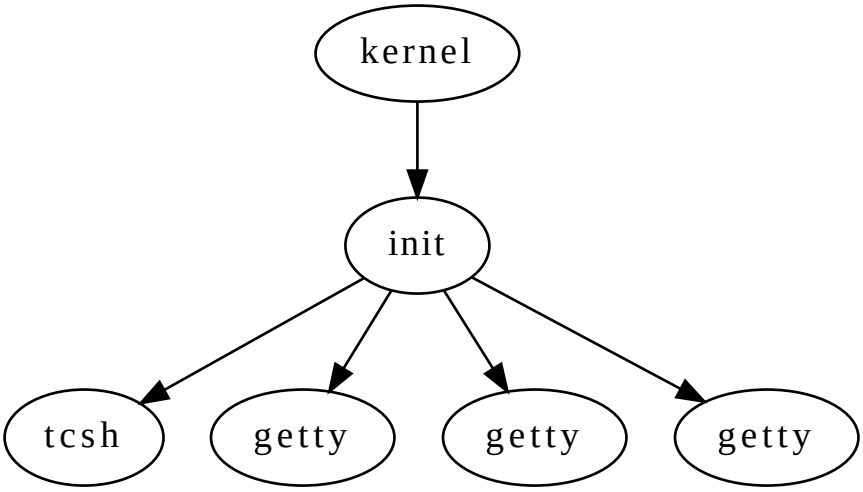


Figure 105: execing the shell

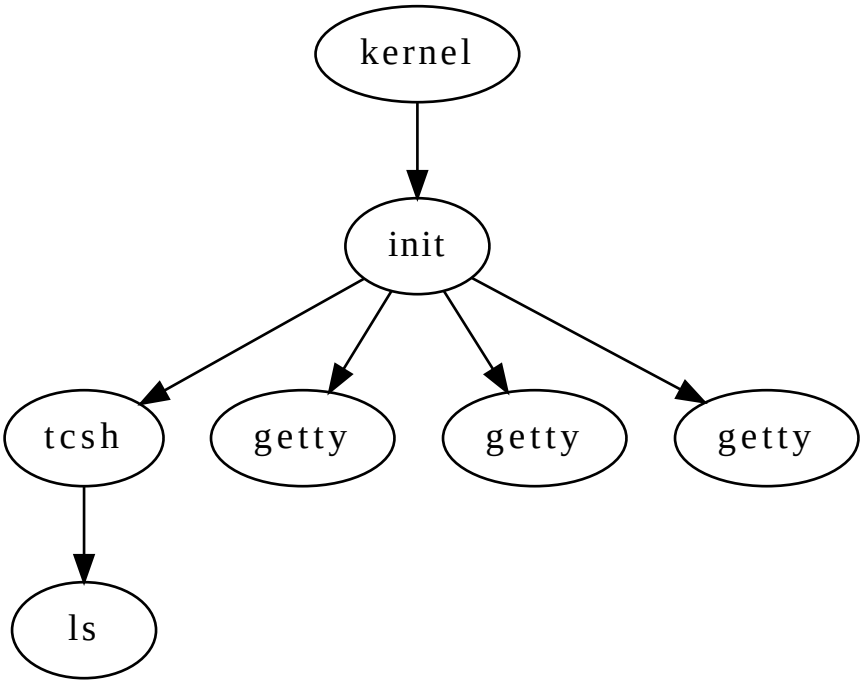


Figure 106: Running ls

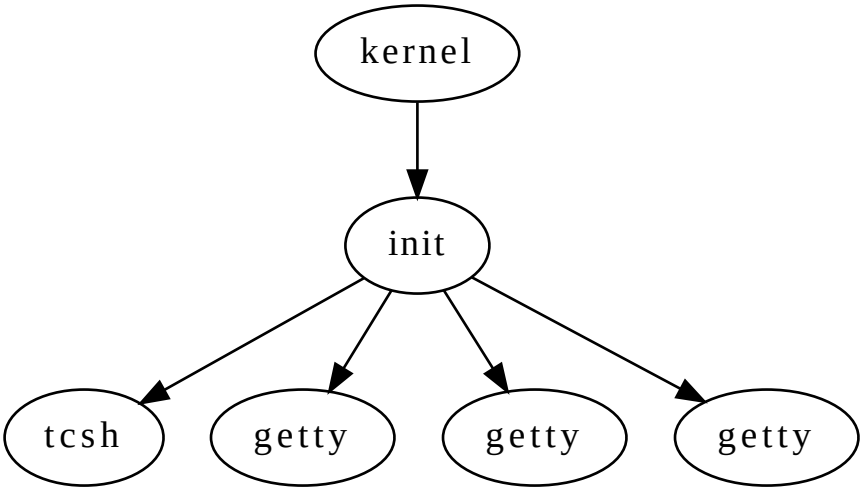


Figure 107: After ls

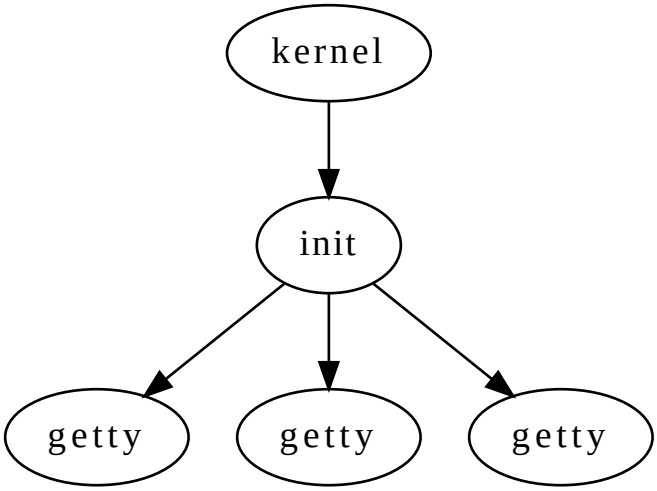


Figure 108: Shell exits

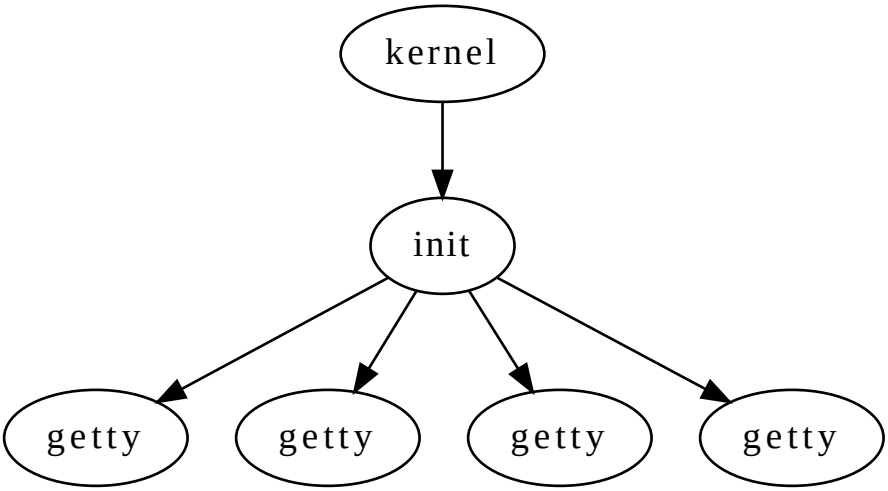


Figure 109: Respawning the getty

24 Lecture: Interprocess Communication

Outline:

24.1 Announcements

- "evaluate the stream" Why just the stream? (cuz man page)
- Assignment road map

```
      Possible: 61.0
          High: 55.0    ( 90.2%)
           Low:  3.0    (  4.9%)
          Mean: 39.2    ( 64.3%)
        Median: 42.0    ( 68.9%)
          S.D.:  9.7    ( 15.8%)

Max Possible:  61.0
     Buckets:  12.5% of  61.0

  Min A-: 53.4  (87.5%)
  Min B-: 45.8  (75.0%)
  Min C-: 38.1  (62.5%)
  Min D-: 30.5  (50.0%)
```

24.2 The Midterm

```
      Possible: 56.0
          High: 54.0    ( 96.4%)
           Low:  2.0    (  3.6%)
          Mean: 29.3    ( 52.4%)
        Median: 33.0    ( 58.9%)
          S.D.: 12.5    ( 22.3%)

      Enrolled:   66
     Submitted:   47    ( 78.3%)
```

Possible: 56.0 High: 54.0 (Low: 2.0 (Mean: 29.3 (Median: 33.0 (S.D.: 12.5 (
Enrolled: 66 Submitted: 56 (84.8

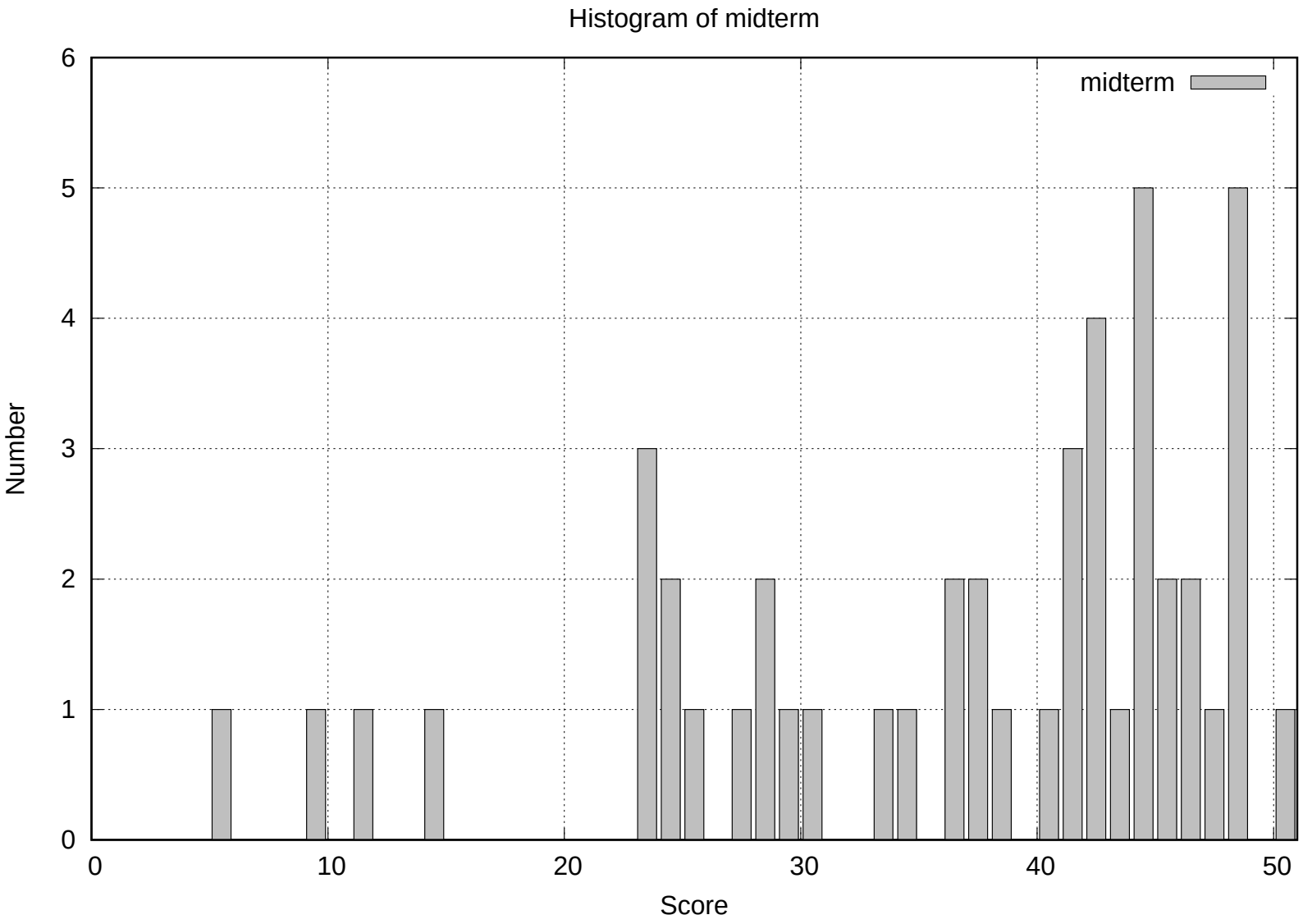


Figure 110: Histogram of scores for the Midterm

24.2.1 POSIX Timers

the use of the POSIX timers API (timer_gettime(2), timer_settime(2),

```
  #include <time.h>

int timer_settime(timer_t timerid, int flags,
                  const struct itimerspec *new_value,
                  struct itimerspec *old_value);
int timer_gettime(timer_t timerid, struct itimerspec *curr_value);

Link with -lrt.
```

`See also: timer_create()`

24.2.2 Interprocess Communication: Signals

Processes can communicate via signals....works, but pretty darn low bandwidth. (Imagine, say, using `SIGUSR1` for 0 and `SIGUSR2` for 1 to communicate bits...)

Also, consider the synchronization problem. Who runs first? If one signals before the other installs a handler, what happens?

24.2.3 Interprocess Communication: Pipes

Last time we talked a little about how processes can communicate through signals (very low bandwidth). Process can also communicate in other ways, so long as they are pre-arranged.

A pipe is a FIFO constructed of two file descriptors and a buffer. (**Section 14.2 in Stevens**)

> Remember, a read on a pipe will not return EOF until there are **no open write descriptors** to it.

```
#include <unistd.h>

int pipe(int filedes[2]);


DESCRIPTION
       pipe  creates  a  pair  of file descriptors, pointing to a
       pipe inode, and places them in the  array  pointed  to  by
       filedes.   filedes[0]  is  for  reading, filedes[1] is for
       writing.

RETURN VALUE
       On success, zero is returned.  On error, -1  is  returned,
       and errno is set appropriately.

ERRORS
       EMFILE Too  many  file  descriptors are in use by the pro-
              cess.

       ENFILE The system file table is full.

       EFAULT filedes is not valid.
```

So what good are they? A process that opens a pipe can use it to communicate:

1. create pipe(s)
2. `fork()`—file descriptors are duped on `fork()`
3. child: use `dup2()` to dup appropriate ends to approprate fds
4. parent and child both close un-used ends
5. child execs program...

24.3 The Environment of a Unix Process

One of the things we discussed as being preserved is the *environment*

- `main()` is called with argc and argv (by `crt0`, added by the linker). These args come from the kernel. `argc` is the count, `argv` is a null terminated array of pointers to strings.
- Each has its own uid, gid, pid (`getpid()`), pwd, etc.
- Every process has a parent (`getppid()`);
- and its own environment— a series of strings

  Access to the environment: Traditionally via:

  ```
  main(int argc, char *argv[], char *env[])
  ```

  but now, a NULL-terminated global list of strings:

  ```
  char *environ[]
  ```

  ```
  SYNOPSIS
         #include <stdlib.h>

         char *getenv(const char *name);

         int setenv(const char *name, const char *value, int overwrite);

         void unsetenv(const char *name);
  ```

  FWIW, this explains the difference between shell variables and environment variables.

24.4 For Next Time: The execs

syscall	**prototype**
execl	int execl(const char *path, const char *arg, ...);
execlp	int execlp(const char *file, const char *arg, ...);
execle	int execle(const char *path, const char *arg , ..., char * const envp[]);
execv	int execv(const char *path, char *const argv[]);
execvp	int execvp(const char *file, char *const argv[]);
execve	int execve (const char *filename, char *const argv [], char *const envp[]);

24.5 Copy on Write

A discussion of vfork() and "copy on write" ensued.

24.6 Putting it all together: Launching a new program

- Set up arguments (list or argv), **terminated by a NULL**. Remember, by convention, argv[0] is the name of the program being executed.
- fork() (optional: could happen before setting up the args, or not at all if this process is done working)
- dup() any file descriptors necessary to set up stdin, stdout, and stderr.
- Close unnecessary files (the newly-executed program will not know about them)
- Once the house is in order, call exec().

24.7 Interprocess Communication

24.7.1 Interprocess Communication: Signals

Processes can communicate via signals....works, but pretty darn low bandwidth. (Imagine, say, using `SIGUSR1` for 0 and `SIGUSR2` for 1 to communicate bits...)

Also, consider the synchronization problem. Who runs first? If one signals before the other installs a handler, what happens?

	fork()	**exec()**
Preserves:	• open file descriptors (dup()-style) • identity • current working directory • current root directory • signal mask and handlers • umask • environment • resource limits	• process id • parent process id • real uid and gid • open file descriptors (dup()-style) (unless close-on-exec is set) • **pending alarms** • current working directory • current root directory • umask • environment • **signal mask** • pending signals • run time values
Changes:	• process id • parent process id • return value of `fork()` • pending alarms are cleared • interval timers are cleared • pending signals are cleared • child's run times are reset	• effective uid and gid if suid • just about everything else (entire contents of memory)

Figure 111: Some important properties of `fork()` and `exec()`

24.7.2 Interprocess Communication: Pipes

Last time we talked a little about how processes can communicate through signals (very low bandwidth). Process can also communicate in other ways, so long as they are pre-arranged.

A pipe is a FIFO constructed of two file descriptors and a buffer. (**Section 14.2 in Stevens**)

> Remember, a read on a pipe will not return EOF until there are **no open write descriptors** to it.

```
       #include <unistd.h>

       int pipe(int filedes[2]);

DESCRIPTION
       pipe  creates  a  pair  of file descriptors, pointing to a
       pipe inode, and places them in the  array  pointed  to  by
       filedes.   filedes[0]  is  for  reading, filedes[1] is for
       writing.

RETURN VALUE
       On success, zero is returned.  On error, -1  is  returned,
       and errno is set appropriately.

ERRORS
       EMFILE Too  many  file  descriptors are in use by the pro-
              cess.

       ENFILE The system file table is full.

       EFAULT filedes is not valid.
```

So what good are they? A process that opens a pipe can use it to communicate:

1. create pipe(s)
2. `fork()`—file descriptors are duped on `fork()`
3. child: use `dup2()` to dup appropriate ends to approprate fds
4. parent and child both close un-used ends
5. child execs program...

24.7.3 Example: `exectest.c`

Figures 113 and 112 show an example of using a pipe to read the output of a child process execing `/bin/ls`.

```
void cat (int in, int out) {
  /* copy the infile to the outfile */
  int num;
  char buffer[MAX];

  while ( (num=read(in, buffer, MAX)) > 0 ) {
    if ( write(out,buffer,num) < 0 ) {
      perror("write");
      break;
    }
  }
  if ( num < 0 ) {
    perror("read");
  }
}
```

Figure 112: The `cat()` routine for `exectest.c`

```c
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

#define READ_END  0
#define WRITE_END 1

#define MAX 256

void cat (int in, int out) ;

int main(int argc, char *argv[]){
  int newpipe[2];
  pid_t child;
  char *prog;

  if ( pipe(newpipe) < 0 ) {
    perror("pipe");
    exit(-1);
  }

  child=fork();

  if ( child == 0 ) {    /* child */

    if ( dup2(newpipe[WRITE_END], STDOUT_FILENO) == -1 ) {
      perror("dup2");
    }
    close(newpipe[READ_END]);   /* we're not going to use it */

    prog = "/bin/ls";
    execl(prog,prog,NULL);
    perror("exec");

    exit(-1);
    /* child exits by this point whether or not exec() succeeds */
  }
  /* only parent gets here */

  close(newpipe[WRITE_END]);    /* otherwise program (cat) will hang */

  cat(newpipe[READ_END],STDOUT_FILENO);

  wait(NULL);                   /* what're we gonna do if it fails? */

  return 0;
}
```

Figure 113: Forking a child and reading its output from a pipe: `exectest.c`

24.8 Go Back to...

This is things that are just getting pushed ahead for now:

- TermIOS
- process groups, esp foreground (for job control/signals)

25 Lecture: Putting it all together: Launching Programs

Outline:

25.1 Announcements

- Coming attractions:

Event	Subject	Due Date			Notes
asgn6	shell	Fri	Dec 8	23:59	

  Use your own discretion with respect to timing/due dates.

- Final plans
- timer lab reminder
- *pow(3)*, just no...

25.2 `fork()` loads a new program

25.3 Loading a new program: The execs

syscall	prototype
execl	int execl(const char *path, const char *arg, ...);
execlp	int execlp(const char *file, const char *arg, ...);
execle	int execle(const char *path, const char *arg , ..., char * const envp[]);
execv	int execv(const char *path, char *const argv[]);
execvp	int execvp(const char *file, char *const argv[]);
execve	int execve (const char *filename, char *const argv [], char *const envp[]);

25.4 The Environment of a Unix Process

One of the things we discussed as being preserved is the *environment*

- `main()` is called with argc and argv (by `crt0`, added by the linker). These args come from the kernel. `argc` is the count, `argv` is a null terminated array of pointers to strings.
- Each has its own uid, gid, pid (`getpid()`), pwd, etc.
- Every process has a parent (`getppid()`);
- and its own environment— a series of strings
  Access to the environment: Traditionally via:
  ```
  main(int argc, char *argv[], char *env[])
  ```
  but now, a NULL-terminated global list of strings:
  ```
  char *environ[]
  ```

```
SYNOPSIS
       #include <stdlib.h>

       char *getenv(const char *name);

       int setenv(const char *name, const char *value, int overwrite);

       void unsetenv(const char *name);
```

FWIW, this explains the difference between shell variables and environment variables.

	fork()	**exec()**
Preserves:	• open file descriptors (dup()-style) • identity • current working directory • current root directory • signal mask and handlers • umask • environment • resource limits	• process id • parent process id • real uid and gid • open file descriptors (dup()-style) (unless close-on-exec is set) • **pending alarms** • current working directory • current root directory • umask • environment • **signal mask** • pending signals • run time values
Changes:	• process id • parent process id • return value of `fork()` • pending alarms are cleared • interval timers are cleared • pending signals are cleared • child's run times are reset	• effective uid and gid if suid • just about everything else (entire contents of memory)

Figure 114: Some important properties of `fork()` and `exec()`

25.4.1 Example: `exectest.c`

How to launch another process:

1. create pipe(s)
2. `fork()`—file descriptors are duped on `fork()`
3. child: use `dup2()` to dup appropriate ends to approprate fds
4. parent and child both close un-used ends
5. child execs program...

Figures 116 and 115 show an example of using a pipe to read the output of a child process execing `/bin/ls`.

```
void cat (int in, int out) {
  /* copy the infile to the outfile */
  int num;
  char buffer[MAX];

  while ( (num=read(in, buffer, MAX)) > 0 ) {
    if ( write(out,buffer,num) < 0 ) {
      perror("write");
      break;
    }
  }
  if ( num < 0 ) {
    perror("read");
  }
}
```

Figure 115: The `cat()` routine for `exectest.c`

```c
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

#define READ_END  0
#define WRITE_END 1

#define MAX 256

void cat (int in, int out) ;

int main(int argc, char *argv[]){
  int newpipe[2];
  pid_t child;
  char *prog;

  if ( pipe(newpipe) < 0 ) {
    perror("pipe");
    exit(-1);
  }

  child=fork();

  if ( child == 0 ) {    /* child */

    if ( dup2(newpipe[WRITE_END], STDOUT_FILENO) == -1 ) {
      perror("dup2");
    }
    close(newpipe[READ_END]);   /* we're not going to use it */

    prog = "/bin/ls";
    execl(prog,prog,NULL);
    perror("exec");

    exit(-1);
    /* child exits by this point whether or not exec() succeeds */
  }
  /* only parent gets here */

  close(newpipe[WRITE_END]);    /* otherwise program (cat) will hang */

  cat(newpipe[READ_END],STDOUT_FILENO);

  wait(NULL);                   /* what're we gonna do if it fails? */

  return 0;
}
```

Figure 116: Forking a child and reading its output from a pipe: `exectest.c`

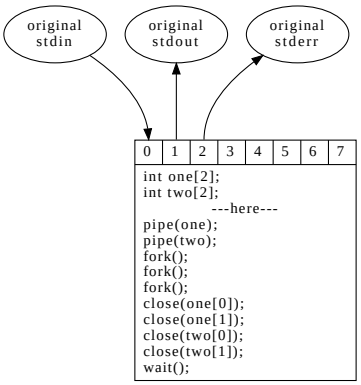


Figure 117: At the start of the program

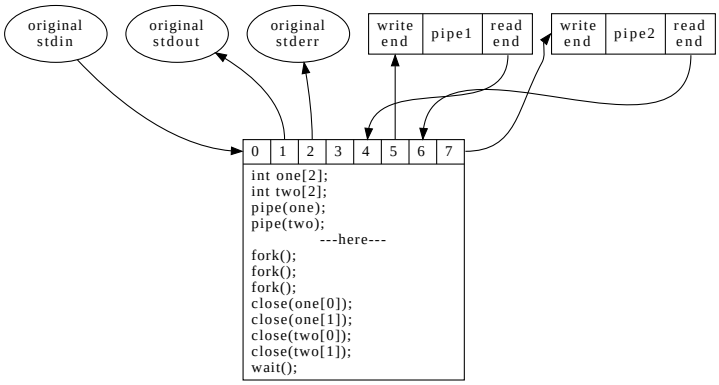


Figure 118: Before the `fork`s

25.4.2 Shell Examples

- Fork **all** children: don't do them sequentially. (otherwise deadlock may result. (also consider the effect on signal handling))
- Example of closing pipes appropriately. Figures 117–121 show the launching of the pipe "`ls | sort -r | more`". The code for this program is given in Figure 122.

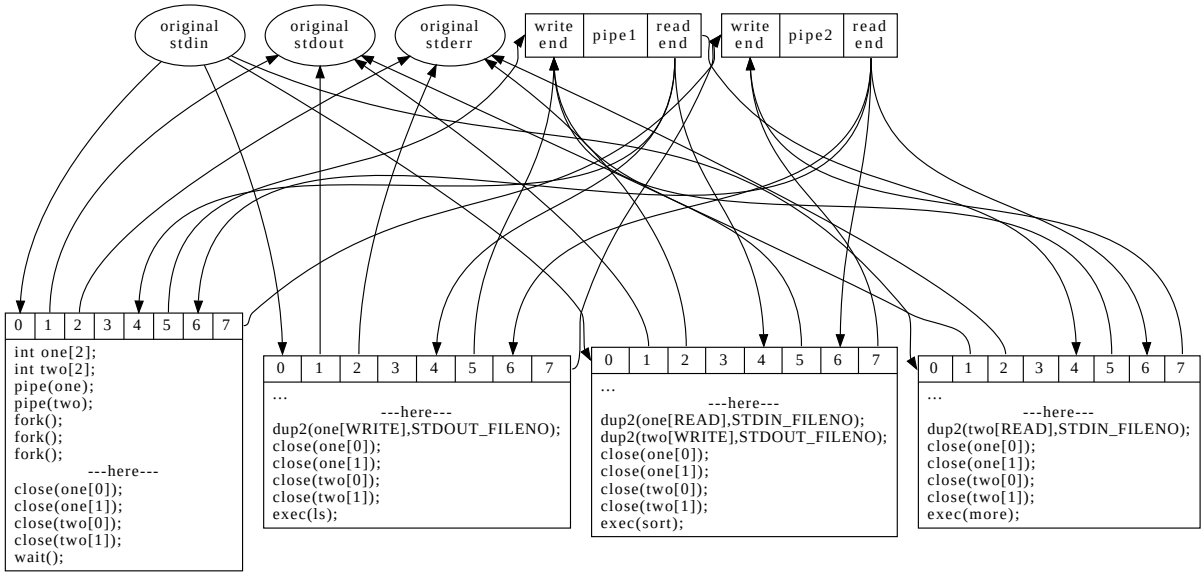


Figure 119: After the `fork`s

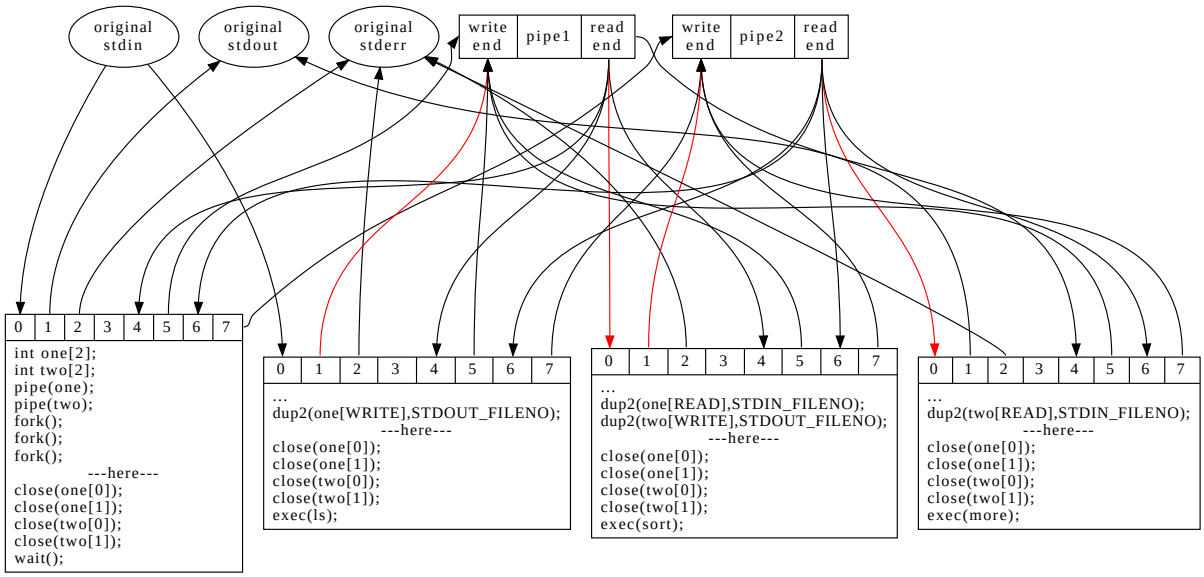


Figure 120: After the `dup()`s

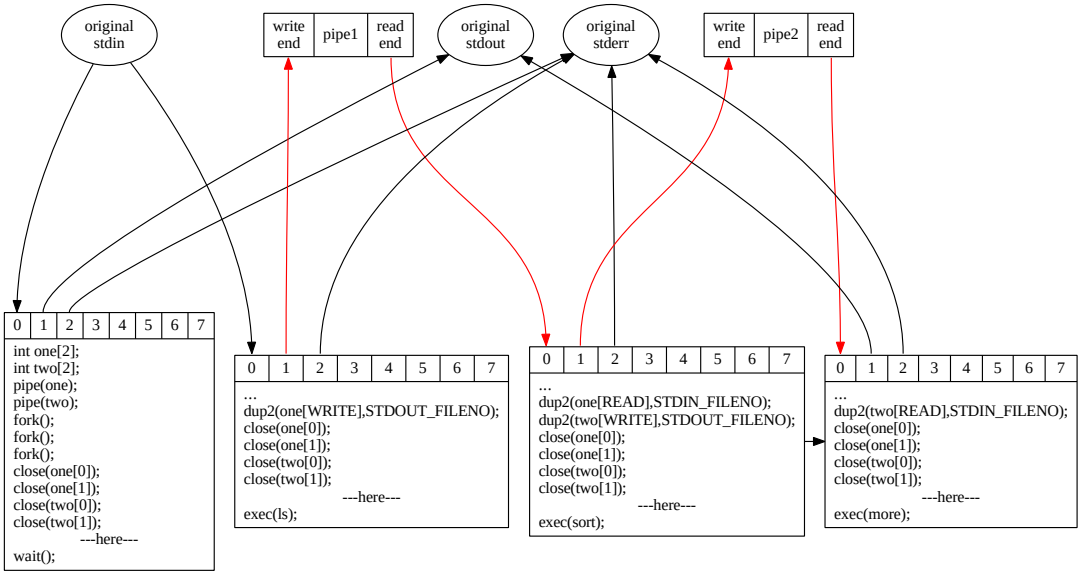


Figure 121: After the `close()`s

```
/*
 * Demonstration of building a three-stage pipeline
 * ls | sort -r | more
 */

#include <unistd.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>

#define READ_END  0
#define WRITE_END 1

int main(int argc, char *argv[]) {
  int one[2];
  int two[2];
  pid_t child1, child2, child3;

  if ( pipe(one) ) {
    perror("First pipe");
    exit(-1);
  }

  if ( pipe(two) ) {
    perror("Second pipe");
    exit(-1);
  }

  if ( ! (child1 = fork()) ) {
    /* child one stuff */
    /* dup appropriate pipe ends */
    if ( -1 == dup2(one[WRITE_END],STDOUT_FILENO)  ) {
      perror("dup2");
      exit(-1);
    }

    /* clean up */
    close(one[READ_END]);
    close(one[WRITE_END]);
    close(two[READ_END]);
    close(two[WRITE_END]);

    /* do the exec */
    execl("/bin/ls","ls",NULL);
    perror("/bin/ls");
    exit(-1);
  }

  if ( ! (child2 = fork()) ) {
    /* child two stuff */
    if ( -1 == dup2(one[READ_END],STDIN_FILENO) ) {
      perror("dup2");
      exit(-1);
    }
    if ( -1 == dup2(two[WRITE_END],STDOUT_FILENO) ) {
      perror("dup2");
      exit(-1);
    }

    /* clean up */
    close(one[READ_END]);
    close(one[WRITE_END]);
    close(two[READ_END]);
    close(two[WRITE_END]);

    /* do the exec */
    execl("/bin/sort","sort","-r",NULL);
    perror("/bin/sort");
    exit(-1);
  }

  if ( ! (child3 = fork()) ) {
    /* child three stuff */
    if ( -1 == dup2(two[READ_END],STDIN_FILENO) ) {
      perror("dup2");
      exit(-1);
    }

    /* clean up */
    close(one[READ_END]);
    close(one[WRITE_END]);
    close(two[READ_END]);
    close(two[WRITE_END]);

    /* do the exec */
    execl("/bin/more","more",NULL);
    perror("/bin/more");
    exit(-1);
  }

  /* parent stuff */

  /* clean up */
  close(one[READ_END]);
  close(one[WRITE_END]);
  close(two[READ_END]);
  close(two[WRITE_END]);

  if ( -1 == wait(NULL) ) {       /* wait for one child */
    perror("wait");
  }
  if ( -1 == wait(NULL) ) {       /* wait for the middle child */
    perror("wait");
  }
  if ( -1 == wait(NULL) ) {       /* wait for the last child */
    perror("wait");
  }

  exit(0);
}
```

Figure 122: Demonstration of a three stage pipeline

Midterm		
High	49.0	98.0%
Low	8.0	16.0%
Mean	36.8	73.6%
Median	38.0	76.0%
S.D.	10.0	19.9%

Grade	Cutoff	Percent
Min A−	43.8	(87.5)%
Min B−	37.5	(75.0)%
Min C−	31.2	(62.5)%
Min D−	25.0	(50.0)%

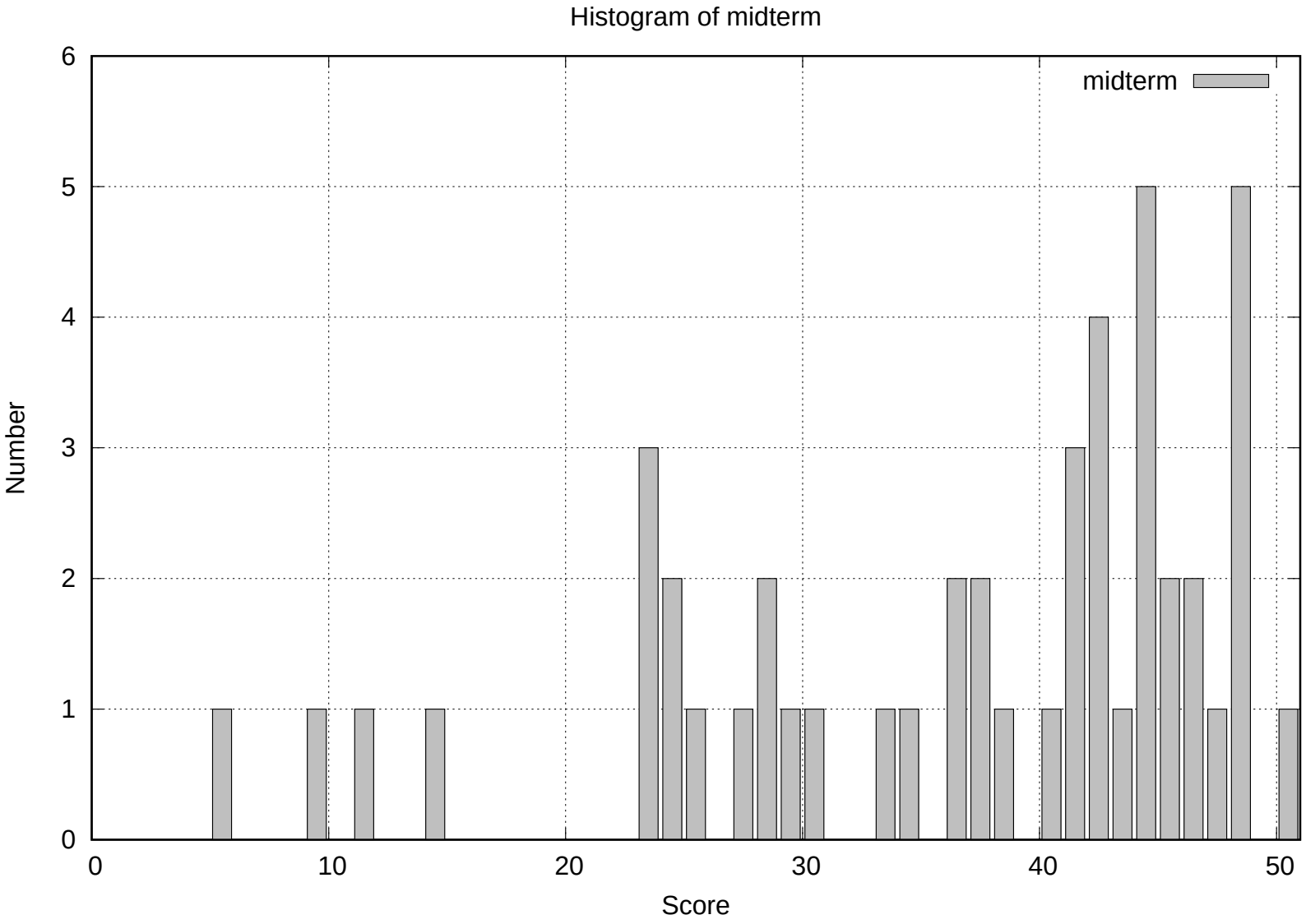


Figure 123: Histogram of scores for the Midterm

25.5 Selected Midterm Solutions

Let's talk midterm.

General:

- $*(A + i)$ is weird.
- Every one of these was in the published set

1. "It might be out of inodes." True, but that's a property of the filesystem, not the file. There's a much better answer.

   The question was phrased "to another filesystem", not specifying which one. What are the odds that every other filesystem is full?

2. Be careful not to do real subtraction. Do the calculation
3. division is far more expensive than shifting/ masking

   - Count 7 or 8 bits, but if you count 8, be sure not to screw it up by changing the 8th bit
   - XOR is cute here
   - locals are not initialized. Realy. Expect noting.

4. Be careful, be careful, be careful. This one should not have been unexpected.
5. - don't print or exit()
   - don't opendir() the path or leak file descriptors
   - There's no reason to search for root (and if you do, you'll find it, but it'll have nothing to do with `path`
   - Don't destroy the current working directory
   - Just declaring a `struct stat *` doesn't mean it's pointing anywhere

26 Lecture: Shell Building

Outline:

Announcements
Onwards: Plumbing
Onwards: Building A Shell
Stuff
    Shell Examples
The world beyond C
Higher: Shell Programming

26.1 Announcements

- Coming attractions:

Event	Subject	Due Date			Notes
asgn6	shell	Fri	Dec 8	23:59	

Use your own discretion with respect to timing/due dates.

You can do this

26.2 Onwards: Plumbing

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

void usage(char *name){
  fprintf(stderr,"usage:  %s <stages>\n", name);
  exit(EXIT_FAILURE);
}

#define WRITE_END 1
#define READ_END  0
#define MSG "Boo\n"

void telephone(int id) {
  int c;
  printf("This is process %d (pid:  %d).\n",id,getpid());
  while ((EOF != (c=getchar())) )
    putchar(c);
}

int main(int argc, char *argv[]) {
  int num,i;
  char *end;
  int old[2], next[2];
  pid_t child;

  if ( argc != 2 )
    usage(argv[0]);

  num = strtol(argv[1],&end,0);
  if (num <= 0 || *end)
    usage(argv[0]);

  if ( pipe(old) ) {
    perror("old pipe");
    exit(EXIT_FAILURE);
  }
  write(old[WRITE_END],MSG,strlen(MSG));

  for(i=0; i<num; i++) {
    if ( i < num-1 ) {      /* create new pipe */
      if ( pipe(next) ) {
        perror("next pipe");
        exit(EXIT_FAILURE);
      }
    }

    if ( !(child = fork()) ) {
      /* child */
      if( -1 == dup2(old[READ_END], STDIN_FILENO) ) {
        perror("dup2 old");
      }
      if ( i < num-1 ) {
        if ( -1 == dup2(next[WRITE_END], STDOUT_FILENO)) {
          perror("dup2 new");
        }
      }
      close(old[0]);            /* clean house */
      close(old[1]);
      close(next[0]);
      close(next[1]);
      telephone(i);
      exit(EXIT_SUCCESS);
    }

    /* parent */
    /* close up old pipe */
    close(old[0]);
    close(old[1]);
    old[0]=next[0];
    old[1]=next[1];
  }

  while ( num-- ) {
    if ( -1 == wait(NULL)) {
      perror("wait");
    }
  }

  return 0;
}
```

Figure 124: A process version of the telephone game

26.3 Onwards: Building A Shell

- Design an overall approach
- Then develop subsystems:
  - Parsing — what're you going to do
  - Launch — doing it
  - Retrieval — finding out you've done it
  - Signal Handling — what to do when interrupted

* tty signals get sent to entire foreground process group
* but different processes may behave differently
* the shell must reset itself. What exactly depends on its state:

  running it must abandon the current input line and re-prompt

  waiting it must wait for the children to all exit

  in-between ??? Good question. Finish? Kill?

- Test these, and you're off.

26.4 Stuff

syscall	prototype
execl	int execl(const char *path, const char *arg, ...);
execlp	int execlp(const char *file, const char *arg, ...);
execle	int execle(const char *path, const char *arg , ..., char * const envp[]);
execv	int execv(const char *path, char *const argv[]);
execvp	int execvp(const char *file, char *const argv[]);
execve	int execve (const char *filename, char *const argv [], char *const envp[]);

```
#include <sys/types.h>
#include <sys/wait.h>

pid_t wait(int *status)
pid_t waitpid(pid_t pid, int *status, int options);

WIFEXITED(status)
       is non-zero if the child exited normally.

WEXITSTATUS(status)
       evaluates to the least significant  eight  bits  of
       the  return  code  of  the  child which terminated,
       which may have been set as the argument to  a  call
       to exit() or as the argument for a return statement
       in the main program.  This macro can only be evalu-
       ated if WIFEXITED returned non-zero.
```

`fork()`	creates a new process that is an exact image of the current one

`execl()`
`execlp()`
`execle()`
`execv()`
`execvp()`
	known collectively as "the execs", this family of functions overwrites the current process with a new program. For this assignment, look closely at `execlp()` and `execvp()`.

`kill()`	sends a signal to a process
`wait()` `waitpid()`	waits for a child process to terminate
`pipe()`	returns an array of two connected file descriptors, one for reading and the other for writing
`isatty()`	for determining whether a particular file descriptor is associated with a tty
`feof()`	Determines whether a FILE * stream has reached its end of file. This is useful for telling when a stdio function's return value of EOF means end-of-file or interrupted system call.

`sscanf()`
`strchar()`
`index()`
`strtok()`
`strpbrk()`
etc.
	The string functions, defined in `string.h` and `strings.h`, are helpful for parsing strings

`isspace()` etc.	One of many functions defined in `ctype.h` for text processing. Very useful.

Other useful things to remember in no particular order (read these and save time debugging):

- It can't be said too many times: stdout is buffered. Be sure to `fflush()` stdout after commands complete. There may be unwritten output still in the buffer.
- Remember that signal masks are inherited over a `fork()` and `exec()`. If you block `SIGINT` while launching your pipeline stages, remember to unblock it in each child before the `exec()`. If you forget, the children will be ever-after deaf to `SIGINT`.
- Keep a list (or at least a count) of child processes. This is the only way to know how many to wait for and which processes to kill (should you need to)
- Remember that `wait()` will get interrupted by the signal handler, so be sure to check its return value to be sure a child actually exited. If you don't, you risk losing count of your children.
- Setting up pipelines involves opening a lot of file descriptors. Be careful to remember to close these file descriptors when done. If you don't, the file descriptor table can fill up and prevent you from running any more commands.

  After each `fork()`, the child has a copy of all file descriptors open in the parent. All unneeded ones should be closed before the `exec()`.
- Be particularly careful to close the *parent's* copy of the write-end of a pipe. The process reading from the read-end will never get an end of file from the pipe until all open descriptors to the write end are closed. *If you forget to do this, pipelines like "*`ls | more`*" will hang forever.* (Even when the `ls` terminates, the parent still has an open descriptor to the pipe.)
- For what it's worth, my implementation—documentation and all—is approximately 1000 lines of C code. I expect mine is more verbose than yours will be. I have also implemented features not required here.

```
SYNOPSIS
       #include <string.h>

       char *strtok(char *s, const char *delim);

#include <string.h>
int strcasecmp(const char *s1, const char *s2);
char *strcat(char *dest, const char *src);
char *strchr(const char *s, int c);
int strcmp(const char *s1, const char *s2);
int strcoll(const char *s1, const char *s2);
char *strcpy(char *dest, const char *src);
size_t strcspn(const char *s, const char *reject);
```

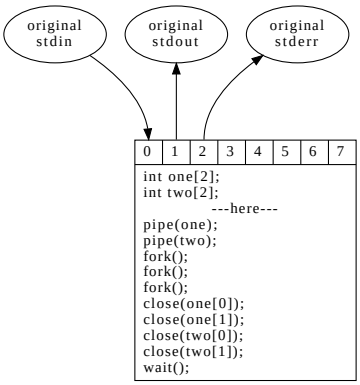


Figure 125: At the start of the program

```
char *strdup(const char *s);
char *strfry(char *string);
size_t strlen(const char *s);
char *strncat(char *dest, const char *src, size_t n);
int strncmp(const char *s1, const char *s2, size_t n);
char *strncpy(char *dest, const char *src, size_t n);
int strncasecmp(const char *s1, const char *s2, size_t n);
char *strpbrk(const char *s, const char *accept);
char *strrchr(const char *s, int c);
char *strsep(char **stringp, const char *delim);
size_t strspn(const char *s, const char *accept);
char *strstr(const char *haystack, const char *needle);
char *strtok(char *s, const char *delim);
size_t strxfrm(char *dest, const char *src, size_t n);
char *index(const char *s, int c);
char *rindex(const char *s, int c);
```

26.4.1 Shell Examples

- Fork **all** children: don't do them sequentially. (otherwise deadlock may result. (also consider the effect on signal handling))
- Example of closing pipes appropriately. Figures 145–149 show the launching of the pipe "`ls | sort -r | more`". The code for this program is given in Figure 150.

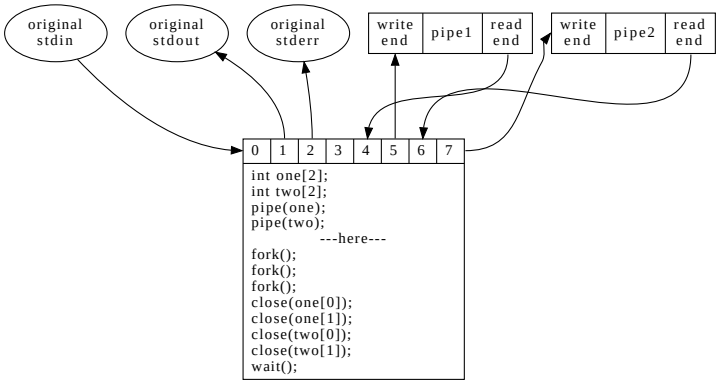


Figure 126: Before the `fork`s

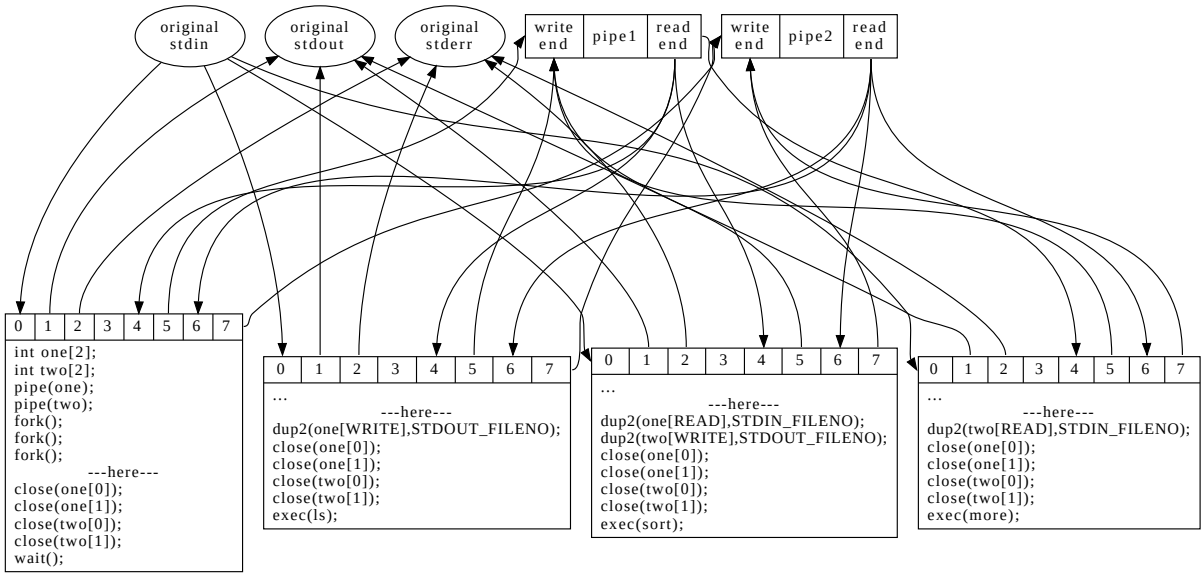


Figure 127: After the `fork`s

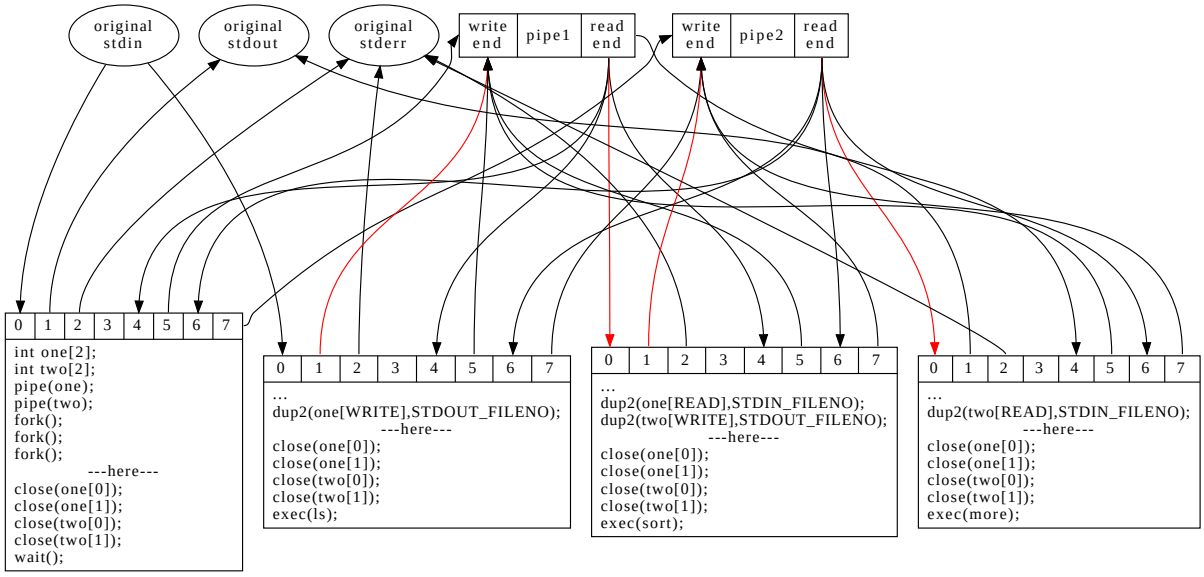


Figure 128: After the `dup()`s

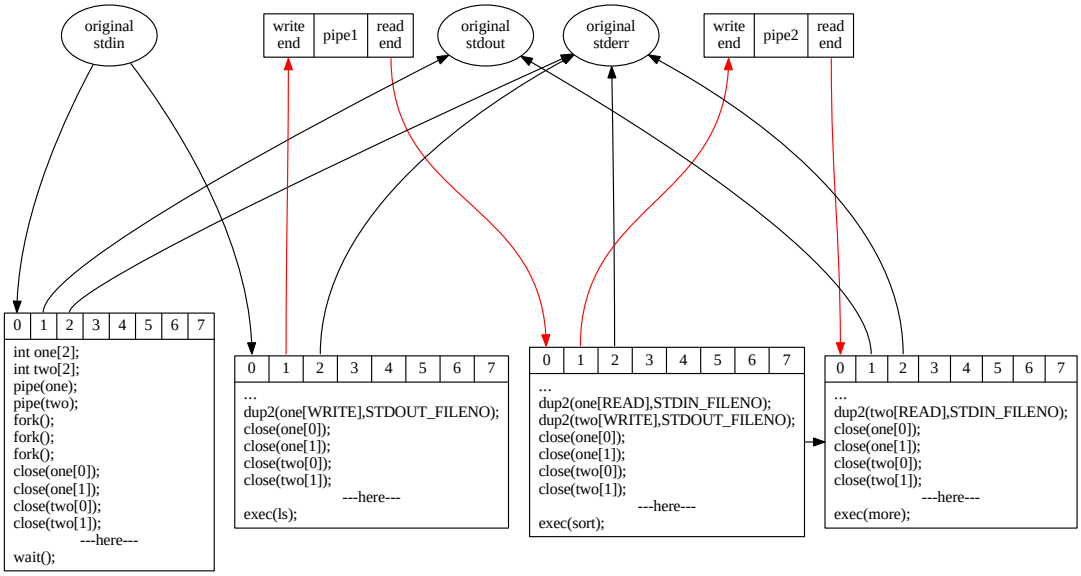


Figure 129: After the `close()`s

```c
/*
 * Demonstration of building a three–stage pipeline
 * ls | sort –r | more
 */

#include <unistd.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>

#define READ_END  0
#define WRITE_END 1

int main(int argc, char *argv[]) {
  int one[2];
  int two[2];
  pid_t child1, child2, child3;

  if ( pipe(one) ) {
    perror("First pipe");
    exit(−1);
  }

  if ( pipe(two) ) {
    perror("Second pipe");
    exit(−1);
  }

  if ( ! (child1 = fork()) ) {
    /* child one stuff */
    /* dup appropriate pipe ends */
    if ( −1 == dup2(one[WRITE_END],STDOUT_FILENO)  ) {
      perror("dup2");
      exit(−1);
    }

    /* clean up */
    close(one[READ_END]);
    close(one[WRITE_END]);
    close(two[READ_END]);
    close(two[WRITE_END]);

    /* do the exec */
    execl("/bin/ls","ls",NULL);
    perror("/bin/ls");
    exit(−1);
  }

  if ( ! (child2 = fork()) ) {
    /* child two stuff */
    if ( −1 == dup2(one[READ_END],STDIN_FILENO) ) {
      perror("dup2");
      exit(−1);
    }
    if ( −1 == dup2(two[WRITE_END],STDOUT_FILENO) ) {
      perror("dup2");
      exit(−1);
    }

    /* clean up */
    close(one[READ_END]);
    close(one[WRITE_END]);
    close(two[READ_END]);
    close(two[WRITE_END]);

    /* do the exec */
    execl("/bin/sort","sort","-r",NULL);
    perror("/bin/sort");
    exit(−1);
  }

  if ( ! (child3 = fork()) ) {
    /* child three stuff */
    if ( −1 == dup2(two[READ_END],STDIN_FILENO) ) {
      perror("dup2");
      exit(−1);
    }

    /* clean up */
    close(one[READ_END]);
    close(one[WRITE_END]);
    close(two[READ_END]);
    close(two[WRITE_END]);

    /* do the exec */
    execl("/bin/more","more",NULL);
    perror("/bin/more");
    exit(−1);
  }

  /* parent stuff */

  /* clean up */
  close(one[READ_END]);
  close(one[WRITE_END]);
  close(two[READ_END]);
  close(two[WRITE_END]);

  if ( −1 == wait(NULL) ) {       /* wait for one child */
    perror("wait");
  }
  if ( −1 == wait(NULL) ) {       /* wait for the middle child */
    perror("wait");
  }
  if ( −1 == wait(NULL) ) {       /* wait for the last child */
    perror("wait");
  }

  exit(0);
}
```

Figure 130: Demonstration of a three stage pipeline

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

void usage(char *name){
  fprintf(stderr,"usage:  %s <stages>\n", name);
  exit(EXIT_FAILURE);
}

#define WRITE_END 1
#define READ_END  0
#define MSG "Boo\n"

void telephone(int id) {
  int c;
  printf("This is process %d (pid:  %d).\n",id,getpid());
  while ((EOF != (c=getchar())) )
    putchar(c);
}

int main(int argc, char *argv[]) {
  int num,i;
  char *end;
  int old[2], next[2];
  pid_t child;

  if ( argc != 2 )
    usage(argv[0]);

  num = strtol(argv[1],&end,0);
  if (num <= 0 || *end)
    usage(argv[0]);

  if ( pipe(old) ) {
    perror("old pipe");
    exit(EXIT_FAILURE);
  }
  write(old[WRITE_END],MSG,strlen(MSG));

  for(i=0; i<num; i++) {
    if ( i < num-1 ) {      /* create new pipe */
      if ( pipe(next) ) {
        perror("next pipe");
        exit(EXIT_FAILURE);
      }
    }

    if ( !(child = fork()) ) {
      /* child */
      if( -1 == dup2(old[READ_END], STDIN_FILENO) ) {
        perror("dup2 old");
      }
      if ( i < num-1 ) {
        if ( -1 == dup2(next[WRITE_END], STDOUT_FILENO)) {
          perror("dup2 new");
        }
      }
      close(old[0]);            /* clean house */
      close(old[1]);
      close(next[0]);
      close(next[1]);
      telephone(i);
      exit(EXIT_SUCCESS);
    }

    /* parent */
    /* close up old pipe */
    close(old[0]);
    close(old[1]);
    old[0]=next[0];
    old[1]=next[1];
  }

  while ( num-- ) {
    if ( -1 == wait(NULL)) {
      perror("wait");
    }
  }

  return 0;
}
```

Figure 131: A process version of the telephone game

26.5 The world beyond C

Believe it or not, there is one. I want to discuss two different modes of programming:

- Sometimes you want to simply combine pre-existing programs to perform a task. Pipelines are one example of this (consider the example in Figure 136), but they don't have control semantics. Most shells provide some sort of control flow.
- After that we will consider the question of what to do if you need access to something below the language level. What if you need to get to actual architectural features to write a device driver or to implement OS features.

26.6 Higher: Shell Programming

The whole C Shell programming interface is described in the man page, but here are a few tidbits to get you going:

```
cat <files>              |\  #   cat all the files
sed 's/[^A-Za-z]/ /g'    |\  #   change all non-letters to spaces
tr ' \t' '\n\n'          |\  #   change all whitespace to newlines
egrep -v '^$'            |\  #   remove all empty lines. The output now consists
                                  of a list of words, one per line.
tr 'A-Z' 'a-z'           |\  #   map all uppercase to lowercase
sort                     |\  #   sort the list of words so duplicates are together
uniq -c                  |\  #   replace all duplicates with a count and one in-
                                  stance of the word
sort -r -n               |\  #   sort the lines backwards as if numbers
head -<k>                  #     keep the first k of them
```

Figure 132: A pipeline to find the k most common words out of a set of files

Quotes As a preliminary, the C Shell supports three different types of quotes with different meanings:

single (')	The included string literally, with no variable expansion
double (")	The included string literally, but shell variables are expanded
backquote (‘)	The command included between the backquotes is executed and its output is substituted for the string

Variables Shell variables are set with "`set name=value`", and evaluated with "`$name`". In many shells (including csh) you can see all the variables that are set using the built-in command "set".

Example:

```
% set prog=/tmp/a.out
% echo $prog
/tmp/a.out
%
```

csh also allows for simple arithmetic:

Example:

```
% @ x = 1
% echo $x
1
% @ x++
% echo $x
2
% while ( $x < 10 )
while?  echo $x
while?  @ x++
while?  end
2
3
4
5
6
7
8
9
%
```

Variable evaluation goes beyond the simple getting of a value

`$var`	Value of the variable
`$?var`	True is `var` is set, false otherwise
`$#var`	Number of elements in an array
`$var[i]`	Value of element i of `var`
`$$`	The current pid

Control The C Shell contains conditionals, a for-loop, a while-loop, and a foreach statement. We're going to discuss two of them here.

The conditional (if) has the form:

```
if ( condition ) then                    if ( condition ) then
    something                                something
endif                        —or—        else
                                             something else
                                         endif
```

The condition can take many forms, but some of them are:

Condition	Meaning
`-f` *name*	true if *name* exists and is a normal file
`-x` *name*	true if *name* exists and is an executable file
`-d` *name*	true if *name* exists and is a directory
"*str1*" `==` "*str2*"	true if strings *str1* and *str2* are the same
"*str1*" `!=` "*str2*"	true if strings *str1* and *str2* are different
{ *command* }	runs command and evaluates to true if *command* exits with zero exit status

All the usual logical connectives (&&, ||, !) apply.

Example:

```
if ( -f Makefile ) then
   make
else
   echo "Makefile not found"
endif
```

After the conditional, the loops are easy

```
foreach var ( list )          iteratively set the variable var to one of the values in list
   stuff                      and do stuff.
end


while ( condition )           Continue doing stuff so long as condition is true
    stuff
end
```

An example using all of the above can be seen in Figure 137 and a more substantial example in 138. The penalty for ease of implementation is paid in performance as seen in Figure 139.

```
#!/bin/csh -f
# Other lines beginning with # represent comments

set testdir=/tmp/tests
set prog=mywc
set reference=mywc.reference

# run prog against reference on each input file in $testdir
foreach test ( $testdir/* )
  $prog $test      > prog.out
  $reference $test > ref.out
  if ( { diff prog.out ref.out } ) then
    # exit value of 0 means they were the same
    echo "Passed test $test"
  else
    echo "FAILED test $test"
  endif
  rm prog.out ref.out
end
```

Figure 133: An example script to test `mywc`

```
#!/bin/csh −f
set k = 10
set files = ()

@ i = 1
while ( $i <= $#argv )
  if ( "$argv[$i]" == "-n" ) then
    @ i++
    if ( $i <= $#argv ) then
      set k = $argv[$i]
    else
      set k = "missing arg"
    endif
  else
    set files = ( $files $argv[$i] )
  endif
  @ i++
end

if ( { ( echo $k | grep '[^0-9]' > /dev/null ) } ) then
  echo "usage:  fw -n <number>"
  exit −1
endif

set words = ( `cat $files              |\
  tr 'A-Z' 'a-z'       |\
  tr −c 'a-z' ' '      |\
  sed 's/[ \t]/\n/g'   |\
  grep −v '^$'         |\
  sort                 |\
  uniq −c              |\
  sort −r −n`          )

@ num = $#words / 2
echo "The top $k words (out of $num) are:"

@ i = 1
@ iters = 2 * $k
while ( $i < $#words && $i <= $iters )
  set n = $words[$i]
  @ i ++
  set w = $words[$i]
  @ i ++
  printf "%9d %s\n" $n $w
end
```

Figure 134: Example: Asgn2 (fw) as a shell script

```
% time fw /usr/man/man1/*
The top 10 words (out of 6356) are:
    10882 the
     4591 fp
     3833 is
     3258 to
     2984 of
     2959 a
     2636 b
     2503 tp
     2390 and
     2208 fr
3.01u 0.26s 0:02.28 143.4%


% time ~pn-cs357/demos/fw /usr/man/man1/*
The top 10 words (out of 6356) are:
    10882 the
     4591 fp
     3833 is
     3258 to
     2984 of
     2959 a
     2636 b
     2503 tp
     2390 and
     2208 fr
0.25u 0.01s 0:00.30 86.6%
```

Figure 135: fw script performance

27 Lecture: Above and Below

Outline:

Announcements
The world beyond C
Higher: Shell Programming
Aside: Reentrancy
Lower: Embedded Assembly instructions
ASM Examples

27.1 Announcements

- Coming attractions:

Event	Subject	Due Date		Notes
asgn6	shell	Fri Dec 8	23:59	

  Use your own discretion with respect to timing/due dates.

- Office hours finals week: TBA

27.2 The world beyond C

Believe it or not, there is one. I want to discuss two different modes of programming:

- Sometimes you want to simply combine pre-existing programs to perform a task. Pipelines are one example of this (consider the example in Figure 136), but they don't have control semantics. Most shells provide some sort of control flow.

```
cat <files>             |\  #   cat all the files
sed 's/[^A-Za-z]/ /g'   |\  #   change all non-letters to spaces
tr ' \t' '\n\n'         |\  #   change all whitespace to newlines
egrep -v '^$'           |\  #   remove all empty lines. The output now consists
                                 of a list of words, one per line.
tr 'A-Z' 'a-z'          |\  #   map all uppercase to lowercase
sort                    |\  #   sort the list of words so duplicates are together
uniq -c                 |\  #   replace all duplicates with a count and one in-
                                 stance of the word
sort -r -n              |\  #   sort the lines backwards as if numbers
head -<k>                 #    keep the first k of them
```

Figure 136: A pipeline to find the k most common words out of a set of files

- After that we will consider the question of what to do if you need access to something below the language level. What if you need to get to actual architectural features to write a device driver or to implement OS features.

27.3 Higher: Shell Programming

The whole C Shell programming interface is described in the man page, but here are a few tidbits to get you going:

Quotes As a preliminary, the C Shell supports three different types of quotes with different meanings:

single (')	The included string literally, with no variable expansion
double (")	The included string literally, but shell variables are expanded
backquote (`)	The command included between the backquotes is executed and its output is substituted for the string

Variables Shell variables are set with "`set name=value`", and evaluated with "`$name`". In many shells (including csh) you can see all the variables that are set using the built-in command "set".

Example:

```
% set prog=/tmp/a.out
% echo $prog
/tmp/a.out
%
```

csh also allows for simple arithmetic:
Example:

```
% @ x = 1
% echo $x
1
% @ x++
% echo $x
2
% while ( $x < 10 )
while?  echo $x
while?  @ x++
while?  end
2
3
4
5
6
7
8
9
%
```

Variable evaluation goes beyond the simple getting of a value

`$var`	Value of the variable
`$?var`	True is `var` is set, false otherwise
`$#var`	Number of elements in an array
`$var[i]`	Value of element i of `var`
`$$`	The current pid

Control The C Shell contains conditionals, a for-loop, a while-loop, and a foreach statement. We're going to discuss two of them here.

The conditional (if) has the form:

```
if ( condition ) then                    if ( condition ) then
    something                                something
endif                        —or—        else
                                             something else
                                         endif
```

The condition can take many forms, but some of them are:

Condition	Meaning
`-f` *name*	true if *name* exists and is a normal file
`-x` *name*	true if *name* exists and is an executable file
`-d` *name*	true if *name* exists and is a directory
"*str1*" `==` "*str2*"	true if strings *str1* and *str2* are the same
"*str1*" `!=` "*str2*"	true if strings *str1* and *str2* are different
{ *command* }	runs command and evaluates to true if *command* exits with zero exit status

All the usual logical connectives (&&, ||, !) apply.

Example:

```
if ( -f Makefile ) then
   make
else
   echo "Makefile not found"
endif
```

After the conditional, the loops are easy

foreach *var* (*list*) *stuff* end	iteratively set the variable *var* to one of the values in *list* and do *stuff*.
while (*condition*) *stuff* end	Continue doing *stuff* so long as *condition* is true

An example using all of the above can be seen in Figure 137 and a more substantial example in 138. The penalty for ease of implementation is paid in performance as seen in Figure 139.

```
#!/bin/csh -f
# Other lines beginning with # represent comments

set testdir=/tmp/tests
set prog=mywc
set reference=mywc.reference

# run prog against reference on each input file in $testdir
foreach test ( $testdir/* )
  $prog $test      > prog.out
  $reference $test > ref.out
  if ( { diff prog.out ref.out } ) then
    # exit value of 0 means they were the same
    echo "Passed test $test"
  else
    echo "FAILED test $test"
  endif
  rm prog.out ref.out
end
```

Figure 137: An example script to test `mywc`

```
#!/bin/csh −f
set k = 10
set files = ()

@ i = 1
while ( $i <= $#argv )
  if ( "$argv[$i]" == "-n" ) then
    @ i++
    if ( $i <= $#argv ) then
      set k = $argv[$i]
    else
      set k = "missing arg"
    endif
  else
    set files = ( $files $argv[$i] )
  endif
  @ i++
end

if ( { ( echo $k | grep '[^0-9]' > /dev/null ) } ) then
  echo "usage:  fw -n <number>"
  exit −1
endif

set words = ( `cat $files              |\
  tr 'A-Z' 'a-z'       |\
  tr −c 'a-z' ' '      |\
  sed 's/[ \t]/\n/g'   |\
  grep −v '^$'         |\
  sort                 |\
  uniq −c              |\
  sort −r −n`          )

@ num = $#words / 2
echo "The top $k words (out of $num) are:"

@ i = 1
@ iters = 2 * $k
while ( $i < $#words && $i <= $iters )
  set n = $words[$i]
  @ i ++
  set w = $words[$i]
  @ i ++
  printf "%9d %s\n" $n $w
end
```

Figure 138: Example: Asgn2 (fw) as a shell script

```
% time fw /usr/man/man1/*
The top 10 words (out of 6356) are:
    10882 the
     4591 fp
     3833 is
     3258 to
     2984 of
     2959 a
     2636 b
     2503 tp
     2390 and
     2208 fr
3.01u 0.26s 0:02.28 143.4%


% time ~pn-cs357/demos/fw /usr/man/man1/*
The top 10 words (out of 6356) are:
    10882 the
     4591 fp
     3833 is
     3258 to
     2984 of
     2959 a
     2636 b
     2503 tp
     2390 and
     2208 fr
0.25u 0.01s 0:00.30 86.6%
```

Figure 139: fw script performance

27.4 Aside: Reentrancy

Reentrant code is code that written such that a single copy in memory can be shared between many applications at once. That is, a reentrant function is one where it is possible to safely have more than one activation at the same time.

What that means:

- no self-modifying code
- no static variables (except those that actually pertain to truly global state).

Examples:

`strcat()` is reentrant.
`strtok()` is not.

27.5 Lower: Embedded Assembly instructions

Sometimes you need to get access to actual machine features that are not namable via C. E.g., registers.

`asm ( `*instruction* `:` *outputs* `:` *inputs* `:` *clobbers* `)`

These fields are:

instruction	This looks like a format string for `printf()`. Arguments are specified by place.
outputs	This is a comma-separated list of register specification pairs specifying the outputs of this instruction
inputs	This is a comma-separated list of register specification pairs specifying the inputs of this instruction
clobbers	A comma-separated list of registers overwritten as side effects. These are registers not mentioned specifically as outputs

Register specifications:

=	denotes an output register
r,g	denote general registers
f	denotes a floating point register
(name)	gives a variable name to which this operand is bound

Example: "`"=g" (foo)`" describes a general purpose output register corresponding to the variable `foo`

27.5.1 ASM Examples

- Using system-specific instructions. The Intel x86 family of processors (those newer than the Pentium anyway) have a build in cycle counter that can be very useful for performance analysis, but there's no way to name it from a high-level language.

  Consider Intel's RDTSC *(read time stamp counter)* instruction.

  See Figures 140– 143.

- the implementation of a system call. See Figure 144.

```
#ifndef CYCLECOUNTH
#define CYCLECOUNTH

#ifndef _i386
#error "This code can only be compiled on an x86"
#endif

#define ccstamp(h,l) \
  asm ( "cpuid" : : : "eax", "ebx", "ecx", "edx"); \
  asm ( "rdtsc" : : : "eax", "edx" );           \
  asm ( "mov %%eax,%0" : "=g" ((l)) : );         \
  asm ( "mov %%edx,%0" : "=g" ((h)) : );         \


#endif
```

Figure 140: Accessing architecture-specific features, macro version

```
#include <stdio.h>

#ifndef _i386
#error "This code can only be compiled on an x86"
#endif

#define TOP    1
#define BOTTOM 0

long long cyclecount() {
  /* returns the value of the pentium's cycle counter
   *
   * The CPUID instruction is issued before and after because it is
   *   a serializing function:  it will not be executed out of
   *   order, so we know when the RDTSC actually executes.
   */
  long lowbits,highbits;

  union {
    unsigned long half[2];
    unsigned long long whole;
  } rvalue;

  asm ( "cpuid" : : : "eax", "ebx", "ecx", "edx"); /* serialize */
  asm ( "rdtsc" : : : "eax", "edx" );
  asm ( "mov %%eax,%0" : "=g" (lowbits) : );
  asm ( "mov %%edx,%0" : "=g" (highbits) : );

  rvalue.half[TOP] = highbits;
  rvalue.half[BOTTOM] = lowbits;
  return rvalue.whole;
}
```

Figure 141: Accessing architecture-specific features: `cyclecount()`

```
#include <stdio.h>
#include <unistd.h>

#include "cyclecount.h"

extern long long cyclecount();

#define HIGH 1
#define LOW  0

int main(int argc, char *argv[]) {
  long long once,again,delta;

  union {
    long half[2];
    long long whole;
  } one,two;

  ccstamp(one.half[HIGH],one.half[LOW]);
  ccstamp(two.half[HIGH],two.half[LOW]);

  once  = one.whole;
  again = two.whole;
  delta = again - once;

  printf("Macro Once:     %lld\n",once);
  printf("Macro Again:   %lld\n",again);
  printf("Macro delta:   %lld\n",delta);



  /* ********************** */
  printf("\n\n");

  once  = cyclecount();
  again = cyclecount();
  delta = again - once;

  printf("Function Once:     %lld\n",once);
  printf("Function Again:   %lld\n",again);
  printf("Function delta:   %lld\n",delta);

  return 0;
}
```

Figure 142: Accessing architecture-specific features `main.c`

```
% cyclecount
Macro Once:  474251422557334
Macro Again: 474251422557902
Macro delta: 568


Function Once:  474251424004250
Function Again: 474251424004830
Function delta: 580
%
```

Note, for a 2Ghz cpu this means:

568 cycles * 1/2000000000 s/cycle	=	0.000 000 284s	=	284ns
580 cycles * 1/2000000000 s/cycle	=	0.000 000 290s	=	290ns

That's a runtime difference of 6 nanoseconds!

Figure 143: Accessing architecture-specific features: output

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdarg.h>
#include <errno.h>

#define SYSCALL_OPEN  0x5

/* make sure this is legitimate */
#ifndef _i386
#error "This can only be compiled for an x86"
#endif

int open(const char *pathname, int flags, ... ) {
  mode_t mode;
  va_list ap;
  int res;

  /* use the variable arguments support */
  va_start(ap,flags);
  mode = va_arg(ap,mode_t);   /* extract the mode from the top of the stack */
  va_end(ap);

  /* load the given values into the registers and execute
   * the given syscall.   The syscall code goes into eax
   * and return value comes from there.
   */

  asm ("movl %0,%%eax" :             : "g" (SYSCALL_OPEN) : "eax");
  asm ("movl %0,%%ebx" :             : "g" (pathname)     : "ebx");
  asm ("movl %0,%%ecx" :             : "g" (flags)        : "ecx");
  asm ("movl %0,%%edx" :             : "g" (mode)         : "edx");
  asm ("int  $0x80"    :             :                    : "eax");
  asm ("movl %%eax,%0" : "=g" (res)  :);

  if ( res < 0 ) {
    errno = -res;
    res = -1;
  }

  return res;
}
```

Figure 144: An implementation of the `open()` system call for x86 linux

28 Lecture: Wrapping Up

Outline:

28.1 Announcements

- Coming attractions:

Event	Subject	Due Date			Notes
asgn6	shell	Fri	Dec 8	23:59	

  Use your own discretion with respect to timing/due dates.

- (Email coming)
  - other stuff
- Next time (Possibly)
  - Wrapping up
  - Class Photo
  - Bring questions (what do you want to talk about?)
- I'm grading as fast as I can...

28.2 Onwards: Building A Shell

- Design an overall approach
- Then develop subsystems:
  - Parsing — what're you going to do
  - Launch — doing it
  - Retrieval — finding out you've done it
  - Signal Handling — what to do when interrupted
    - tty signals get sent to entire foreground process group
    - but different processes may behave differently
    - the shell must reset itself. What exactly depends on its state:
      running it must abandon the current input line and re-prompt
      waiting it must wait for the children to all exit
      in-between ??? Good question. Finish? Kill?
- Test these, and you're off.

28.3 Stuff

syscall	prototype
execl	int execl(const char *path, const char *arg, ...);
execlp	int execlp(const char *file, const char *arg, ...);
execle	int execle(const char *path, const char *arg , ..., char * const envp[]);
execv	int execv(const char *path, char *const argv[]);
execvp	int execvp(const char *file, char *const argv[]);
execve	int execve (const char *filename, char *const argv [], char *const envp[]);

```
#include <sys/types.h>
#include <sys/wait.h>

pid_t wait(int *status)
pid_t waitpid(pid_t pid, int *status, int options);

WIFEXITED(status)
       is non-zero if the child exited normally.

WEXITSTATUS(status)
       evaluates to the least significant  eight  bits  of
       the  return  code  of  the  child which terminated,
       which may have been set as the argument to  a  call
       to exit() or as the argument for a return statement
       in the main program.  This macro can only be evalu-
       ated if WIFEXITED returned non-zero.
```

`fork()`	creates a new process that is an exact image of the current one

`execl()`
`execlp()`
`execle()`
`execv()`
`execvp()`
	known collectively as "the execs", this family of functions overwrites the current process with a new program. For this assignment, look closely at `execlp()` and `execvp()`.

`kill()`	sends a signal to a process
`wait()` `waitpid()`	waits for a child process to terminate
`pipe()`	returns an array of two connected file descriptors, one for reading and the other for writing
`isatty()`	for determining whether a particular file descriptor is associated with a tty
`feof()`	Determines whether a FILE * stream has reached its end of file. This is useful for telling when a stdio function's return value of EOF means end-of-file or interrupted system call.

`sscanf()`
`strchar()`
`index()`
`strtok()`
`strpbrk()`
etc.
	The string functions, defined in `string.h` and `strings.h`, are helpful for parsing strings

`isspace()` etc.	One of many functions defined in `ctype.h` for text processing. Very useful.

Other useful things to remember in no particular order (read these and save time debugging):

- It can't be said too many times: stdout is buffered. Be sure to `fflush()` stdout after commands complete. There may be unwritten output still in the buffer.
- Remember that signal masks are inherited over a `fork()` and `exec()`. If you block `SIGINT` while launching your pipeline stages, remember to unblock it in each child before the `exec()`. If you forget, the children will be ever-after deaf to `SIGINT`.
- Keep a list (or at least a count) of child processes. This is the only way to know how many to wait for and which processes to kill (should you need to)
- Remember that `wait()` will get interrupted by the signal handler, so be sure to check its return value to be sure a child actually exited. If you don't, you risk losing count of your children.
- Setting up pipelines involves opening a lot of file descriptors. Be careful to remember to close these file descriptors when done. If you don't, the file descriptor table can fill up and prevent you from running any more commands.

  After each `fork()`, the child has a copy of all file descriptors open in the parent. All unneeded ones should be closed before the `exec()`.
- Be particularly careful to close the *parent's* copy of the write-end of a pipe. The process reading from the read-end will never get an end of file from the pipe until all open descriptors to the write end are closed. *If you forget to do this, pipelines like "*`ls | more`*" will hang forever.* (Even when the `ls` terminates, the parent still has an open descriptor to the pipe.)
- For what it's worth, my implementation—documentation and all—is approximately 1000 lines of C code. I expect mine is more verbose than yours will be. I have also implemented features not required here.

```
SYNOPSIS
       #include <string.h>

       char *strtok(char *s, const char *delim);

#include <string.h>
int strcasecmp(const char *s1, const char *s2);
char *strcat(char *dest, const char *src);
char *strchr(const char *s, int c);
int strcmp(const char *s1, const char *s2);
int strcoll(const char *s1, const char *s2);
char *strcpy(char *dest, const char *src);
size_t strcspn(const char *s, const char *reject);
```

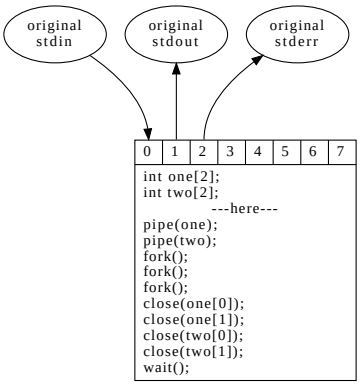


Figure 145: At the start of the program

```
char *strdup(const char *s);
char *strfry(char *string);
size_t strlen(const char *s);
char *strncat(char *dest, const char *src, size_t n);
int strncmp(const char *s1, const char *s2, size_t n);
char *strncpy(char *dest, const char *src, size_t n);
int strncasecmp(const char *s1, const char *s2, size_t n);
char *strpbrk(const char *s, const char *accept);
char *strrchr(const char *s, int c);
char *strsep(char **stringp, const char *delim);
size_t strspn(const char *s, const char *accept);
char *strstr(const char *haystack, const char *needle);
char *strtok(char *s, const char *delim);
size_t strxfrm(char *dest, const char *src, size_t n);
char *index(const char *s, int c);
char *rindex(const char *s, int c);
```

28.3.1 Shell Examples

- Fork **all** children: don't do them sequentially. (otherwise deadlock may result. (also consider the effect on signal handling))
- Example of closing pipes appropriately. Figures 145–149 show the launching of the pipe "`ls | sort -r | more`". The code for this program is given in Figure 150.

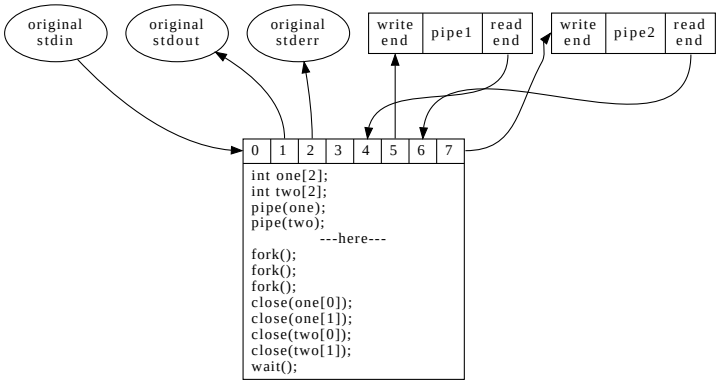


Figure 146: Before the `fork`s

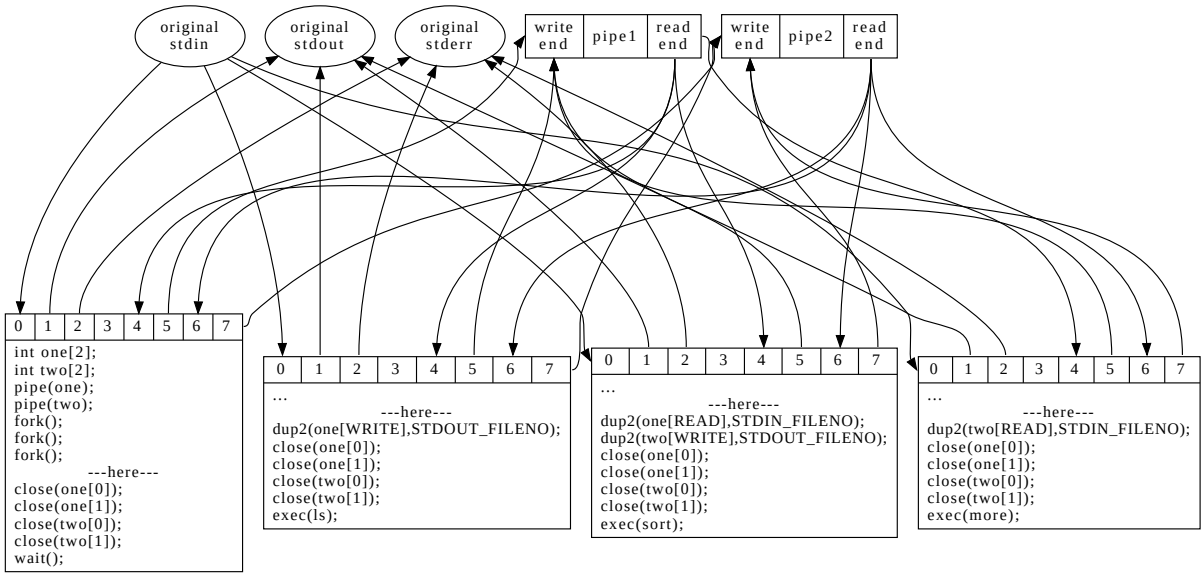


Figure 147: After the `fork`s

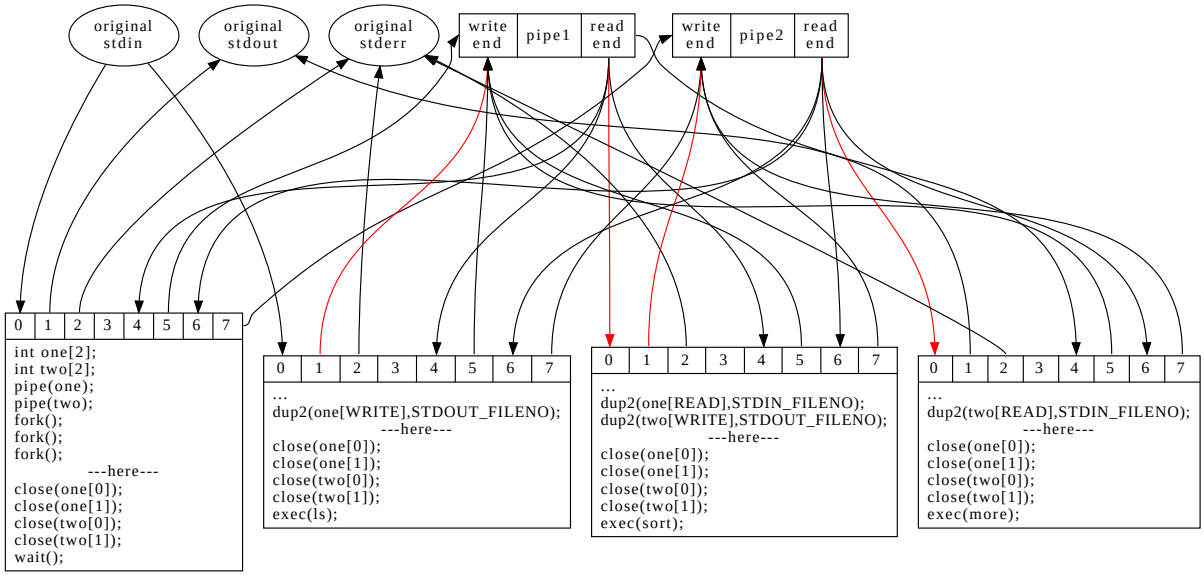


Figure 148: After the `dup()`s

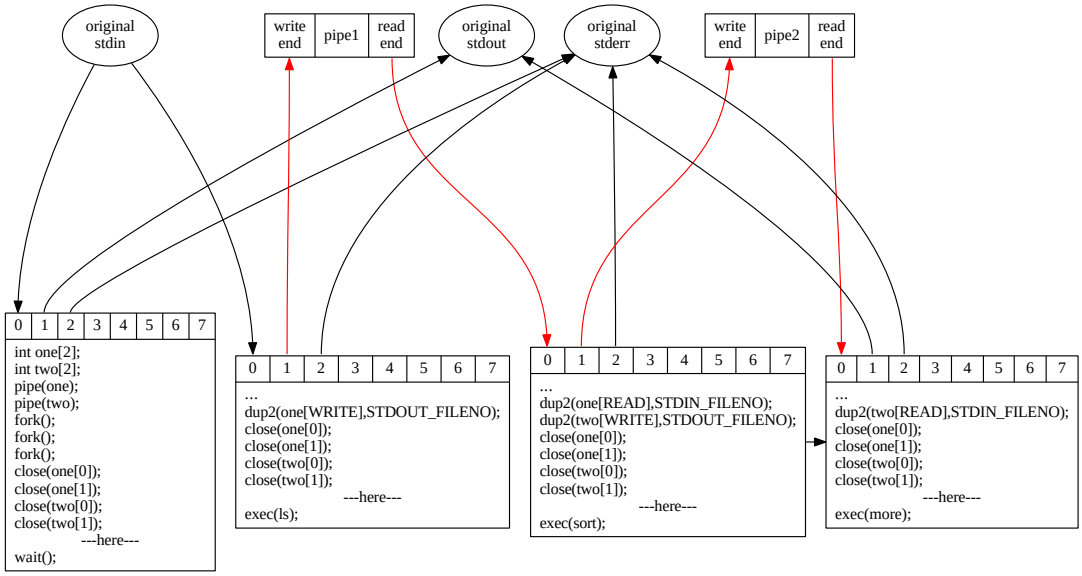


Figure 149: After the `close()`s

```
/*
 * Demonstration of building a three–stage pipeline
 * ls | sort –r | more
 */

#include <unistd.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>

#define READ_END  0
#define WRITE_END 1

int main(int argc, char *argv[]) {
  int one[2];
  int two[2];
  pid_t child1, child2, child3;

  if ( pipe(one) ) {
    perror("First pipe");
    exit(−1);
  }

  if ( pipe(two) ) {
    perror("Second pipe");
    exit(−1);
  }

  if ( ! (child1 = fork()) ) {
    /* child one stuff */
    /* dup appropriate pipe ends */
    if ( −1 == dup2(one[WRITE_END],STDOUT_FILENO)  ) {
      perror("dup2");
      exit(−1);
    }

    /* clean up */
    close(one[READ_END]);
    close(one[WRITE_END]);
    close(two[READ_END]);
    close(two[WRITE_END]);

    /* do the exec */
    execl("/bin/ls","ls",NULL);
    perror("/bin/ls");
    exit(−1);
  }

  if ( ! (child2 = fork()) ) {
    /* child two stuff */
    if ( −1 == dup2(one[READ_END],STDIN_FILENO) ) {
      perror("dup2");
      exit(−1);
    }
    if ( −1 == dup2(two[WRITE_END],STDOUT_FILENO) ) {
      perror("dup2");
      exit(−1);
    }

    /* clean up */
    close(one[READ_END]);
    close(one[WRITE_END]);
    close(two[READ_END]);
    close(two[WRITE_END]);

    /* do the exec */
    execl("/bin/sort","sort","-r",NULL);
    perror("/bin/sort");
    exit(−1);
  }

  if ( ! (child3 = fork()) ) {
    /* child three stuff */
    if ( −1 == dup2(two[READ_END],STDIN_FILENO) ) {
      perror("dup2");
      exit(−1);
    }

    /* clean up */
    close(one[READ_END]);
    close(one[WRITE_END]);
    close(two[READ_END]);
    close(two[WRITE_END]);

    /* do the exec */
    execl("/bin/more","more",NULL);
    perror("/bin/more");
    exit(−1);
  }

  /* parent stuff */

  /* clean up */
  close(one[READ_END]);
  close(one[WRITE_END]);
  close(two[READ_END]);
  close(two[WRITE_END]);

  if ( −1 == wait(NULL) ) {       /* wait for one child */
    perror("wait");
  }
  if ( −1 == wait(NULL) ) {       /* wait for the middle child */
    perror("wait");
  }
  if ( −1 == wait(NULL) ) {       /* wait for the last child */
    perror("wait");
  }

  exit(0);
}
```

Figure 150: Demonstration of a three stage pipeline

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

void usage(char *name){
  fprintf(stderr,"usage:  %s <stages>\n", name);
  exit(EXIT_FAILURE);
}

#define WRITE_END 1
#define READ_END  0
#define MSG "Boo\n"

void telephone(int id) {
  int c;
  printf("This is process %d (pid:  %d).\n",id,getpid());
  while ((EOF != (c=getchar())) )
    putchar(c);
}

int main(int argc, char *argv[]) {
  int num,i;
  char *end;
  int old[2], next[2];
  pid_t child;

  if ( argc != 2 )
    usage(argv[0]);

  num = strtol(argv[1],&end,0);
  if (num <= 0 || *end)
    usage(argv[0]);

  if ( pipe(old) ) {
    perror("old pipe");
    exit(EXIT_FAILURE);
  }
  write(old[WRITE_END],MSG,strlen(MSG));

  for(i=0; i<num; i++) {
    if ( i < num-1 ) {      /* create new pipe */
      if ( pipe(next) ) {
        perror("next pipe");
        exit(EXIT_FAILURE);
      }
    }

    if ( !(child = fork()) ) {
      /* child */
      if( -1 == dup2(old[READ_END], STDIN_FILENO) ) {
        perror("dup2 old");
      }
      if ( i < num-1 ) {
        if ( -1 == dup2(next[WRITE_END], STDOUT_FILENO)) {
          perror("dup2 new");
        }
      }
      close(old[0]);            /* clean house */
      close(old[1]);
      close(next[0]);
      close(next[1]);
      telephone(i);
      exit(EXIT_SUCCESS);
    }

    /* parent */
    /* close up old pipe */
    close(old[0]);
    close(old[1]);
    old[0]=next[0];
    old[1]=next[1];
  }

  while ( num-- ) {
    if ( -1 == wait(NULL)) {
      perror("wait");
    }
  }

  return 0;
}
```

Figure 151: A process version of the telephone game

28.4 Documentation

(Class portrait)

28.5 Final Review

28.5.1 Resources

- Most necessary system call prototypes will be listed on the exam. (Unnecessary ones may be listed as well.)

28.5.2 Style

The final style will be short-answer questions for breadth and some longer ones to demonstrate depth of knowledge. I try to emphasize understanding over rote knowledge, but bear in mind that in a field such as this two are hard to separate.

The midterm is not a bad style guide.

Question ideas?

28.5.3 Material: Pre-Midterm

- The C Programming Language (background noise)
  - The Basics: Type system, flow control, calling protocols, etc.
  - Structures and types
  - Memory management: malloc() and free() and traversing dynamic data structures.
  - Preprocessor controls, headers, etc.
- Unix Systems Programming
  - System architecture:
    - identity (users and groups)
    - programs
    - processes
    - the filesystem
    - IO (buffered vs. not and the issues involved)
    - ipc
    - system calls
  - The Unix Family: (Chapter 2 stuff)
  - Unbuffered IO – the basic system calls and how they work
    - open, close, creat, read, write, lseek
    - dup, dup2
    - fcntl() and ioctl()
  - The filesystem: Directories and structure
    - Structure of the filesystem (inodes and whatnot)
    - opendir(), readdir(), etc.
    - the stats
    - link, unlink, symlink, readlink
    - identity
    - How times work

28.5.4 Material: Post-Midterm

- Signals and Signal handling (Ch. 10)
  - types and where they come from
  - traditional vs. reliable (`signal()` vs. `sigaction()`)
  - Alarms (`alarm()` and `setitimer()`) (Not Fall 2024, cuz reasons)
- Networks
  - Lightly

- Processes: (Ch. 7 and 8)
  - Process life cycle
  - `fork()`, `exec()`, and `wait()`
  - the different execs
- Pipes: (Sec. 15.2)
  - `pipe()`
  - Ramifications of concurrency.

28.5.5 More specifically

Kernighan and Ritchie

Stevens Ch. 1,3,4 (lightly 2) Which bits are C and which bits are unix.

Stevens Ch. 10, 7, 8, 15.2

Other Lectures and Homework.

28.6 What's next: Life in the big system

It's all happening at once:

- resource management
- synchronization
- deadlock: Two (or more) processes locked in a mutual wait.

Where this might show up:

- waiting for signals
- waiting for input.
- locks (we didn't discuss this, but one can open a file exclusively.)

In particular, pipes: readers and writers.

28.7 Wrapping up

So what's the class all about anyhow?

- This is not a programming course, although it involves programming.
- This is not really even a unix course, although unix concepts are central to it.
- What you learn, mostly, is more about the inner workings of any complex computer system. System services need to be provided and managed, and at some level, someone has to "know" about the actual implementation.

  You also learn to distinguish the language from the operating system.

  These skills will serve beyond programming on unix to help with software development whereever you are working.
- *Pay attention to the man behind the curtain!*