9 Lecture: Unbuffered IO, cont.

Outline:

9.1 Announcements

- Coming attractions:

Event	Subject	Due Date	Notes

  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

9.2 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

9.3 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.

9.3.1 Looking at system files

real vs. shadow or yp

shadow/yp on hornet and drseuss

9.4 Files and Directories

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

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

9.4.1 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.

9.4.2 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()

9.5 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

9.6 From last time: Files and the filesystem

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

9.7 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);
```

9.7.1 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

9.7.2 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)`

9.7.3 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.
```

9.7.4 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..

9.7.5 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..

9.8 Performance: Buffered vs. Unbuffered

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

```
% 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
```

```
#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 38: 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 39: cp based on unbuffered IO

9.9 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);
```

9.10 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

···

`write(fd,"Hello",5);`

···	H	e	l	l	o

`lseek(fd,-5,SEEK_CUR);`

···	H	e	l	l	o

`write(fd,"abc",3);`

···	a	b	c	l	o

`lseek(fd,5,SEEK_END);`

···	a	b	c	l	o

`write(fd,"there",5);`

···	a	b	c	l	o	\0	\0	\0	\0	\0	t	h	e	r	e

9.11 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

9.12 If there's time: Lab03/Asgn3

9.12.1 The assignment

Discussion of the assignment:

Write two programs to compress and uncompress files:

Usage:

hencode infile [outfile]

hdecode [(infile | -) [outfile]]

9.13 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.
```

9.13.1 Huffman Codes

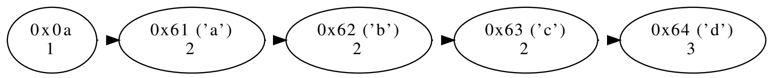
David A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the Institute of Radio Engineers*, 40(9):1098–1101, September 1952.

Building the tree

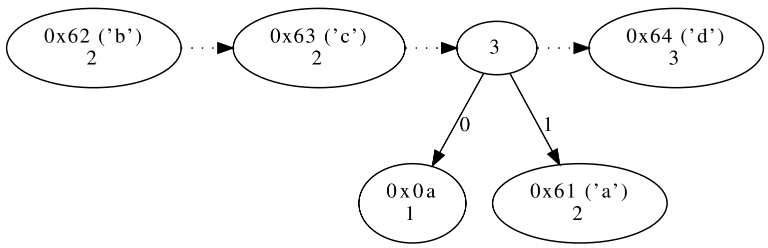
```
% cat test
aabbccddd
%
```

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

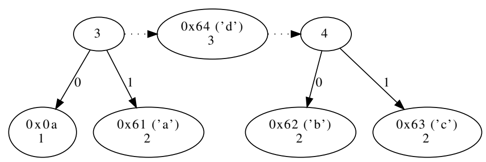
1. Building the tree: Stage 0



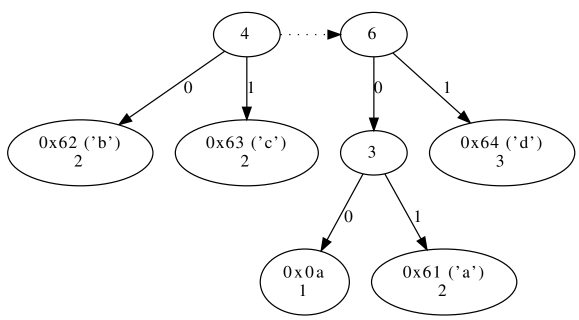
2. Building the tree: Stage 1



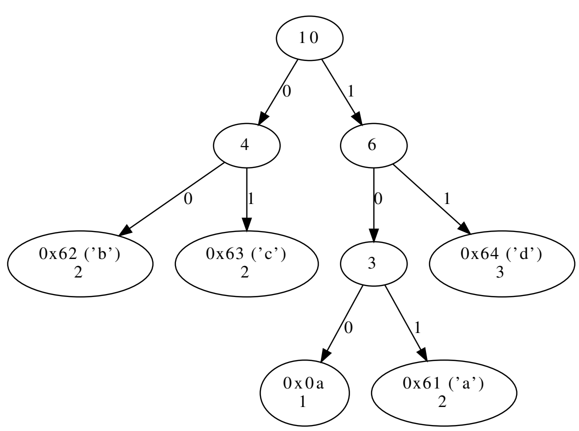
3. Building the tree: Stage 2



4. Building the tree: Stage 3



5. Building the tree: Stage 4



File Format

Num - 1	**c1**	**count of c1**	**c2**	**count of c2**	…	**cn**	**count of cn**
(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
```

9.13.2 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:

  $$word \lor mask$$

  Bitwise, anything ORed with 1 is set and ORed with 0 is itself:

  $$\begin{array}{cccccc} & a & b & c & d & e \\ \text{OR} & 1 & 0 & 1 & 1 & 0 \\ \hline & 1 & b & 1 & 1 & e \end{array}$$

- To Clear Bits:

  Bitwise-AND the word with the inverse of mask:

  $$word \land \overline{mask}$$

  Bitwise, anything ANDed with 1 is itself and ANDed with 0 is 0:

  $$\begin{array}{cccccc} & a & b & c & d & e \\ \text{AND} & 1 & 0 & 1 & 1 & 0 \\ \hline & a & 0 & c & d & 0 \end{array}$$

- In C:

Bitwise OR:	|
Bitwise AND:	&
Bitwise NOT:	~
Bitwise XOR:	∧

An example in Figure 41.

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

Figure 40: printing the bits in a byte.