12 Lecture: Filesystem Wrapup for Real

Outline:

Announcements
Huffman Codes
    The assignment
From Email: Huffman
    Huffman Codes
    Logistics: extracting the codes
    Reminder: Setting and clearing bits
From last time
Our story so far
Onwards: lseek(2)
Other calls: dup, dup2
    dup() and dup2()
    Uses for dup() and dup2()
Onwards: Structure of the Filesystem
    File Types
    Implementation
    Structure of the filesystem
    More examples: Links, symbolic and otherwise
    Directory Reading Functions

12.1 Announcements

- Coming attractions:

Event	Subject	Due Date			Notes
asgn3	hencode/hdecode	Fri	Nov 3	23:59	
lab05	mypwd	Mon	Nov 6	23:59	
asgn4	mytar	Mon	Nov 27	23:59	
asgn5	mytalk	Fri	Dec 1	23:59	
lab07	forkit	Mon	Dec 4	23:59	
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

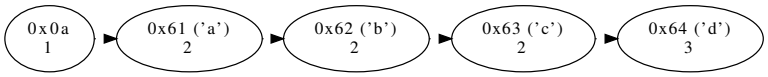
David A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the Institute of Radio Engineers*, 40(9):1098–1101, September 1952.

Building the tree

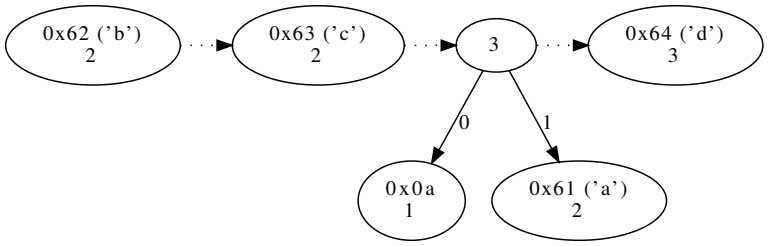
```
% cat test
aabbccddd
%
```

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

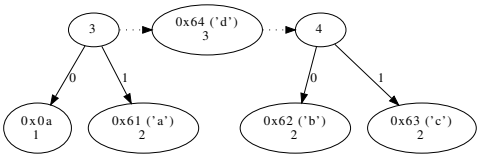
1. Building the tree: Stage 0



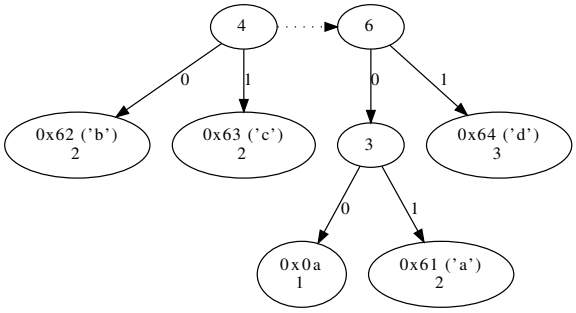
2. Building the tree: Stage 1



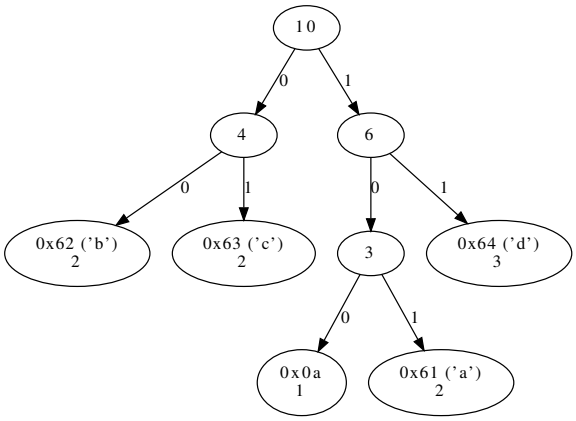
3. Building the tree: Stage 2



4. Building the tree: Stage 3



5. Building the tree: Stage 4



File Format

Num - 1	**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	0000 0000 00	0000 0001 01	0110 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
```

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: $\wedge$

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.

```
#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

12.5 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);
```

12.6 Onwards: lseek(2)

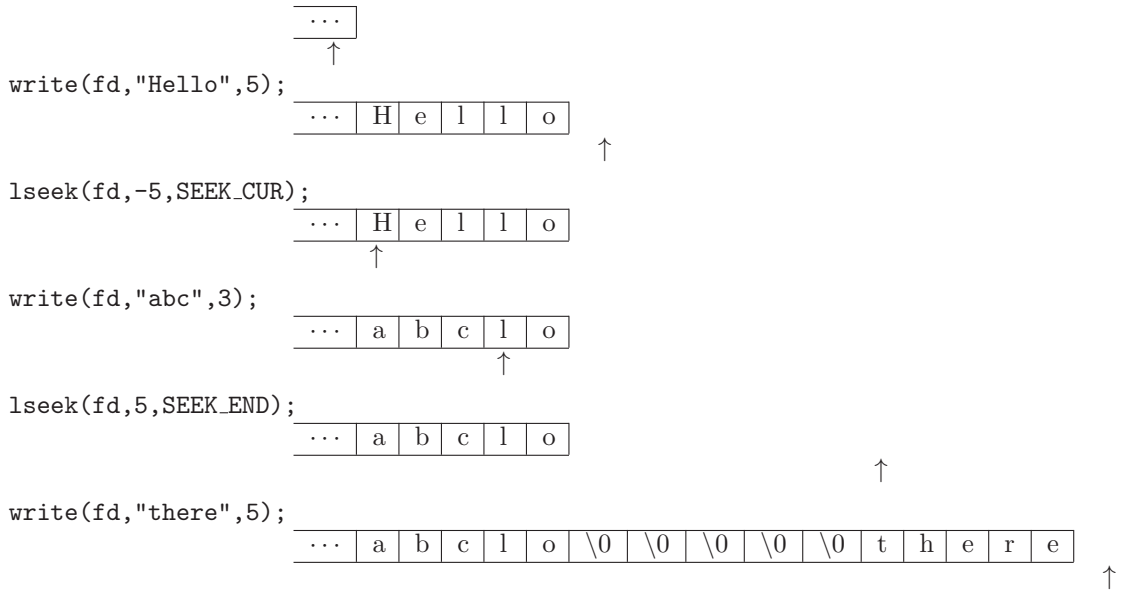
Move the file pointer (or find out where it is). This movement has no effect on the file until the next write.

`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

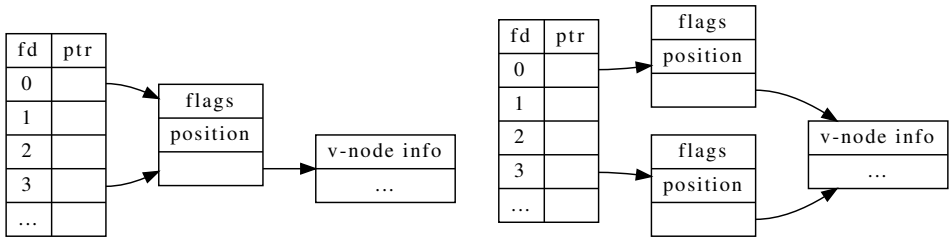
Wrapping up the system calls that affect a process' view of IO.

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

```
/*
 * 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

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:
  - 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

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

```
/*
 * 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

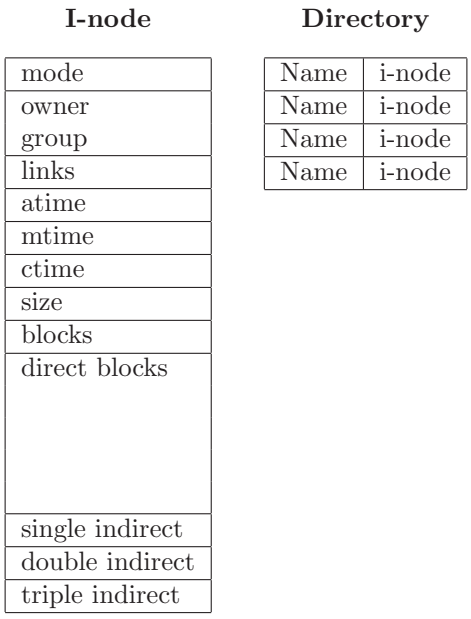


Figure 62: Structures for Directories and I-nodes

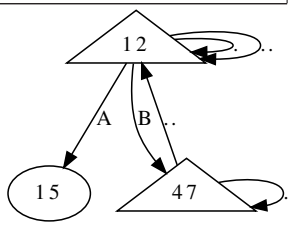
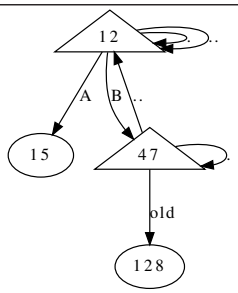
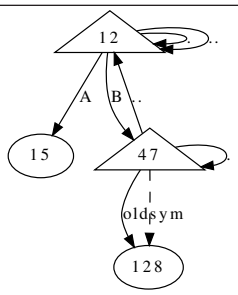
- hard links can only be made within a filesystem.
- only root may make a hard link to a directory

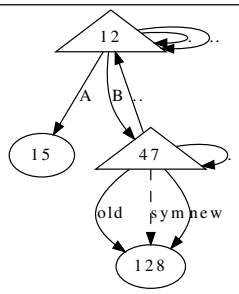
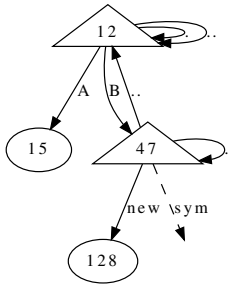
- Symbolic links:
  - like a link, but links to a name, not an i-node.
  - More fragile than a hard link, but can work across filesystems and to directories.

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.

<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>
</table>

<table>
<tr><th>Command</th><th>/
(inode 12)</th><th>/B
(inode 47)</th><th>Resulting Tree
Structure</th></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)