15213 Recitation Section C

Shimin Chen
Nov. 18, 2002

Outline

• Robust I/O package
• Chapter 11 practice problems
Important Dates

• Lab 6 *Malloc*: due on Thursday, Nov 21
• Lab 7 *Proxy*: due on Thursday, Dec 5
• Final Exam: Tuesday, Dec 17
Robust I/O: RIO

- **csapp.c** and **csapp.h**
- **Why?**
  - Handles interrupted system calls
  - Handles short counts
  - Good for network programming
- **Two parts:**
  - Unbuffered I/O
  - Buffered I/O
Rio: Unbuffered Input/Output

- Use Unix I/O
- No internal buffering
- Useful for reading/writing binary data to/from networks

```c
ssize_t rio_readn(int fd, void* usrbuf, size_t n)
    - reads n bytes from fd and put into usrbuf
    - only returns short count on EOF

ssize_t rio_writen(int fd, void* usrbuf, size_t n)
    - writes n bytes from usrbuf to fd
    - never returns short count
```
RIO: Buffered Input

- Internal buffers

```c
#define RIO_BUFSIZE 8192
typedef struct {
    int rio_fd;
    int rio_cnt;
    char *rio_bufptr;
    char rio_buf[RIO_BUFSIZE];
} rio_t;

void rio_readinitb(rio_t* rp, int fd);
ssize_t rio_readlineb(rio_t* rp,
                      void* usrbuf, size_t maxlen);
ssize_t rio_readnb(rio_t* rp,
                  void* usrbuf, size_t n);
```
Rio: Buffered Input

```c
void rio_readinitb(rio_t* rp, int fd);
  // called only once per open descriptor
  // associate fd with a read buffer rio_t structure pointed to by rp

ssize_t rio_readlineb(rio_t* rp, void* usrbuf,
                      size_t maxlen);
  // for reading text lines
  // read a line (until '\n') or maxlen-1 chars from file rp to usrbuf
  // terminate the text line with null (zero) character
  // returns number of chars read

ssize_t rio_readnb(rio_t* rp, void* usrbuf, size_t n);
  // reads n bytes from rp into usrbuf
  // Result string is NOT null-terminated!
  // Returns number of bytes read
```
ssize_t rio_readlineb(rio_t *rp, void *usrbuf, size_t maxlen)
{
    int   n, rc;
    char  c, *bufp = usrbuf;

    for (n = 1; n < maxlen; n++) {
        if ((rc = rio_read(rp, &c, 1)) == 1) {
            *bufp++ = c;
            if (c == '\n')
                break;
        } else if (rc == 0) {
            if (n == 1)
                return 0;       /* EOF, no data read */
            else
                break;         /* EOF, some data was read */
        } else
            return -1;       /* error */
    }

    *bufp = 0;
    return n;
}
Interleaving RIO Read Functions

- Do not interleave calls on the same fd between the buffered and unbuffered functions.
- Within each set it is ok.

<table>
<thead>
<tr>
<th>buffered</th>
<th>unbuffered</th>
</tr>
</thead>
<tbody>
<tr>
<td>rio_readinitb</td>
<td>rio_readn</td>
</tr>
<tr>
<td>rio_readlineb</td>
<td>rio_writen</td>
</tr>
<tr>
<td>rio_readnb</td>
<td></td>
</tr>
</tbody>
</table>

- Why?
Rio Error Checking

- RIO functions handle
  - Short counts
  - interrupted system calls

- All functions have upper case equivalents
  - `Rio_readn`, `Rio_writen`, `Rio_readlineb`, `Rio_readnb`, etc.
  - call `unix_error` if the function encounters an error

- But EPIPE errors!
  - for Lab 7, EPIPE should not terminate the process
Problems from Chapter 11

- 11.1 ~ 11.5
Problem 11.1

What is the output of the following program?

```c
#include "csapp.h"

int main()
{
    int fd1, fd2;
    fd1 = Open("foo.txt", O_RDONLY, 0);
    Close(fd1);
    fd2 = Open("baz.txt", O_RDONLY, 0);
    printf("fd2 = %d\n", fd2);
    exit(0);
}
```
Answer to 11.1

- **stdin** (descriptor 0)
  **stdout** (descriptor 1)
  **stderr** (descriptor 2)

- **open** always returns *lowest* unopened descriptor

- First **open** returns 3. **close** frees it.
- So second **open** also returns 3.
- Program prints: "**fd2 = 3**"
File Sharing

• Descriptor table
  – Each process has its own
  – Child inherits from parents

• File Table
  – set of all open files
  – Shared by all processes
  – Reference count of number of file descriptors pointing to each entry
  – File position

• V-node table
  – Contains information in the `stat` structure
  – Shared by all processes
Problem 11.2

Suppose that `fooban.txt` consists of the 6 ASCII characters "fooban". Then what is the output of the following program?

```c
#include "csapp.h"

int main()
{
    int fd1, fd2;
    char c;
    fd1 = Open("fooban.txt", O_RDONLY, 0);
    fd2 = Open("fooban.txt", O_RDONLY, 0);
    Read(fd1, &c, 1);
    Read(fd2, &c, 1);
    printf("c = %c\n", c);
    exit(0);
}
```
Answer to 11.2

The descriptors \texttt{fd1} and \texttt{fd2} each have their own open file table entry, so each descriptor has its own file position for \texttt{foobar.txt}. Thus, the read from \texttt{fd2} reads the first byte of \texttt{foobar.txt}, and the output is

\[
\texttt{c = f}
\]

and not

\[
\texttt{c = o}
\]

as you might have thought initially.
Problem 11.3

As before, suppose `foobar.txt` consists of 6 ASCII characters "foobar". Then what is the output of the following program?

```c
#include "csapp.h"

int main()
{
    int fd;
    char c;
    fd = Open("foobar.txt", O_RDONLY, 0);
    if (Fork() == 0)
        {Read(fd, &c, 1); exit(0);}
    Wait(NULL);
    Read(fd, &c, 1);
    printf("c = %c\n", c);
    exit(0);
}
```
Answer to 11.3

Child inherit’s the parent’s descriptor table. So child and parent share an open file table entry (refcount = 2). Hence they share a file position.

\[ c = o \]
Problem 11.4

• How would you use `dup2` to redirect standard input to descriptor 5?

• `int dup2(int oldfd, int newfd);`
  – copies descriptor table entry `oldfd` to descriptor table entry `newfd`
Answer to 11.4

dup2 (5, 0);

or

dup2 (5, STDIN_FILENO);
Problem 11.5

Assuming that `foobar.txt` consists of 6 ASCII characters “foobar”. Then what is the output of the following program?

```c
#include "csapp.h"

int main()
{
    int fd1, fd2;
    char c;
    fd1 = Open("foobar.txt", O_RDONLY, 0);
    fd2 = Open("foobar.txt", O_RDONLY, 0);
    Read(fd2, &c, 1);
    Dup2(fd2, fd1);
    Read(fd1, &c, 1);
    printf("c = %c\n", c);
    exit(0);
}
```
Answer to 11.5

We are redirecting $fd1$ to $fd2$. ($fd1$ now points to the same open file table entry as $fd2$). So the second `Read` uses the file position offset of $fd2$.

$$c = 0$$