Lab 3 Report

1 比较同步/异步读取文件用时

测试代码如下：
syncread.c：

```
#include "apue.h"
#include <stdio.h>
#include <fcntl.h>
#include <time.h>

#define BSZ 4096000

int
main(int argc, char *argv[])
{
  int fd, n;
  char buffer[BSZ];

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

  if ((fd = open(argv[1], O_RDONLY)) < 0)
    err_sys("open");

  clock_t begin_time = clock();
  while ((n = read(fd, buffer, BSZ)) > 0) ;
  clock_t end_time = clock();

  if (n < 0)
      err_sys("read");

  close(fd);

  printf("Synchronous read time: %f seconds\n", (double)(end_time - begin_time) /
CLOCKS_PER_SEC);

  return 0;
}
```

aioread.c：

```
#include "apue.h"
#include <ctype.h>
#include <fcntl.h>
#include <aio.h>
```

```c
#include <errno.h>
#include <time.h>
#include <stdio.h>

#define BSZ 4096000
#define NBUF 8

enum rwop {
  UNUSED = 0,
  READ_PENDING = 1
};

struct buf {
  enum rwop     op;
  int           last;
  struct aiocb  aiocb;
  unsigned char data[BSZ];
};

struct buf bufs[NBUF];

int
main(int argc, char* argv[])
{
  int         ifd, i, n, err, numop;
  struct stat     sbuf;
  const struct aiocb  *aiolist[NBUF];
  off_t       off = 0;

  if (argc != 2)
    err_quit("usage: %s <filename>\n", argv[0]);
  if ((ifd = open(argv[1], O_RDONLY)) < 0)
    err_sys("can't open %s", argv[1]);
  if (fstat(ifd, &sbuf) < 0)
    err_sys("fstat failed");

  /* initialize the buffers */
  for (i = 0; i < NBUF; i++) {
    bufs[i].op = UNUSED;
    bufs[i].aiocb.aio_fildes = ifd;
    bufs[i].aiocb.aio_buf = bufs[i].data;
    bufs[i].aiocb.aio_sigevent.sigev_notify = SIGEV_NONE;
    aiolist[i] = NULL;
  }

  numop = 0;
  clock_t begin_time = clock();
  for (;;) {
    for (i = 0; i < NBUF; i++) {
      switch (bufs[i].op) {
      case UNUSED:
        /*
```

```
          * Read from the input file if more data
          * remains unread.
          */
         if (off < sbuf.st_size) {
           bufs[i].op = READ_PENDING;
           bufs[i].aiocb.aio_offset = off;
           off += BSZ;
           if (off >= sbuf.st_size)
             bufs[i].last = 1;
           bufs[i].aiocb.aio_nbytes = BSZ;
           if (aio_read(&bufs[i].aiocb) < 0)
             err_sys("aio_read failed");
           aiolist[i] = &bufs[i].aiocb;
           numop++;
         }
         break;

       case READ_PENDING:
         if ((err = aio_error(&bufs[i].aiocb)) == EINPROGRESS)
           continue;
         if (err != 0) {
           if (err == -1)
             err_sys("aio_error failed");
           else
             err_exit(err, "read failed");
         }
         /*
          * A read is complete.
          */
         if ((n = aio_return(&bufs[i].aiocb)) < 0)
           err_sys("aio_return failed");
         if (n != BSZ && !bufs[i].last)
           err_quit("short read (%d/%d)", n, BSZ);
         aiolist[i] = NULL;
         bufs[i].op = UNUSED;
         numop--;
         break;
       }
     }
     if (numop == 0) {
       if (off >= sbuf.st_size)
         break;
     } else {
       if (aio_suspend(aiolist, NBUF, NULL) < 0)
         err_sys("aio_suspend failed");
     }
   }
   clock_t end_time = clock();
   printf("Asynchronous read time: %f seconds\n", (double)(end_time - begin_time) /
CLOCKS_PER_SEC);

   if (aio_fsync(O_SYNC, &bufs[0].aiocb) < 0)
```

```
    err_sys("aio_fsync failed");
  exit(0);
}
```

运行结果如下：

```
@NekoYellow → /workspaces/codespaces-blank/apue.3e/advio $ ./syncread in.txt
Synchronous read time: 0.030559 seconds
@NekoYellow → /workspaces/codespaces-blank/apue.3e/advio $ ./aioread in.txt
Asynchronous read time: 0.038038 seconds
@NekoYellow → /workspaces/codespaces-blank/apue.3e/advio $ wc in.txt
    409194    409195 419424000 in.txt
```

分析：

异步读用时大于同步读用时，不过两者用时随着buffer size的增大逐渐接近。

理论上异步的`aio_read`效率应该优于`read`，但是为了处理加入了很多辅助代码，这部分的开销可能会超过aio带来的优化。如果输入文件足够大，aio用时应该少与普通read。

2 文件翻译

分别用`read`和`aio_read`完成文件的读取、翻译和写入。理论上与调用`read`需要阻塞直到读取完成相比，使用`aio_read`可以有效利用读取的时间，对当前读到的内容先进行处理，节省总时间开销。但是在linux上运行结果显示aio用时仍然较多：

```
@NekoYellow → /workspaces/codespaces-blank/apue.3e/advio $ make
gcc -ansi -I../include -Wall -DLINUX -D_GNU_SOURCE  rot1.c -o rot1  -L../lib -lapue -lrt
gcc -ansi -I../include -Wall -DLINUX -D_GNU_SOURCE  rot2.c -o rot2  -L../lib -lapue -lrt
@NekoYellow → /workspaces/codespaces-blank/apue.3e/advio $ ./rot1 in.txt out.txt
Sync translation time: 2.729554 seconds
@NekoYellow → /workspaces/codespaces-blank/apue.3e/advio $ ./rot2 in.txt out.txt
Async translation time: 3.012946 seconds
```

（在Mac上得到了合理的结果）

```
(base) [yyh@14MB:~/Desktop/sysprog-labs/lab3 on main]
% make
gcc     test_read.c   -o test_read
gcc     rot1.c   -o rot1
gcc     rot2.c   -o rot2
(base) [yyh@14MB:~/Desktop/sysprog-labs/lab3 on main]
% ./rot1 in.txt out.txt
Sync translation time: 2.637191 seconds
(base) [yyh@14MB:~/Desktop/sysprog-labs/lab3 on main]
% ./rot2 in.txt out.txt
Async translation time: 2.527681 seconds
```

3 高性能io服务器

io服务器最基本的功能即监听一组端口（形式与文件一致，用fd表示，可接受客户端连接），通过这些端口接受和处理io请求。

阻塞的同步io`read`显然不能满足高性能的需求，因为服务器在处理一个端口的io时，特别是如果这个连接的客户端一直不发数据，服务器端线程会一直阻塞在`read`上不返回，也无法接受其他客户端连接。

Linux系统中提供了epoll来实现非阻塞的io多路复用。epoll的实现原理如下：内核中会保存一副文件描述符集合（使用支持快速增删查改的数据结构，如平衡树），内核通过异步io事件唤醒（比轮询高效）来找到就绪的fd，并且会将有io事件的fd返回给用户。

一种可能的实现如下：

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/epoll.h>
#include <netinet/in.h>
#include <fcntl.h>
#include <errno.h>

#define MAX_EVENTS 1024
#define BUFFER_SIZE 4096
#define PORT 8080

void err_sys(const char *msg) {
    perror(msg);
    exit(1);
}

int set_nonblocking(int fd) {
    int flags = fcntl(fd, F_GETFL, 0);
    if (flags == -1) return -1;
    return fcntl(fd, F_SETFL, flags | O_NONBLOCK);
}

int main() {
    int listen_fd, epoll_fd;
    struct epoll_event ev, events[MAX_EVENTS];
    /* 创建监听socket */
    listen_fd = socket(AF_INET, SOCK_STREAM, 0);
    if (listen_fd == -1)
        err_sys("socket");

    /* 设置socket为非阻塞模式 */
    if (set_nonblocking(listen_fd) == -1)
        err_sys("set_nonblocking");

    struct sockaddr_in server_addr;
    memset(&server_addr, 0, sizeof(server_addr));
    server_addr.sin_family = AF_INET;
    server_addr.sin_addr.s_addr = INADDR_ANY;
    server_addr.sin_port = htons(PORT);

    /* 绑定地址和端口 */
    if (bind(listen_fd, (struct sockaddr *)&server_addr, sizeof(server_addr)) == -1)
        err_sys("bind");
```

```
    /* 开始监听 */
    if (listen(listen_fd, SOMAXCONN) == -1)
        err_sys("listen");

    /* 创建epoll实例 */
    epoll_fd = epoll_create1(0);
    if (epoll_fd == -1)
        err_sys("epoll_create1");

    /* 将监听socket添加到epoll实例 */
    ev.events = EPOLLIN;
    ev.data.fd = listen_fd;
    if (epoll_ctl(epoll_fd, EPOLL_CTL_ADD, listen_fd, &ev) == -1)
        err_sys("epoll_ctl: listen_fd");

    printf("Server listening on port %d\n", PORT);

    while (1) {
        int nfds = epoll_wait(epoll_fd, events, MAX_EVENTS, -1);
        if (nfds == -1) {
            if (errno == EINTR)
                continue;
            err_sys("epoll_wait");
        }

        int n;
        for (n = 0; n < nfds; ++n) {
            if (events[n].data.fd == listen_fd) {
                /* 接受新的连接 */
                while (1) {
                    int conn_fd = accept(listen_fd, NULL, NULL);
                    if (conn_fd == -1) {
                        if (errno == EAGAIN || errno == EWOULDBLOCK)
                            break; /* 没有更多的连接需要接受 */
                        else
                            err_sys("accept");
                    }

                    /* 设置新连接为非阻塞模式 */
                    if (set_nonblocking(conn_fd) == -1)
                        err_sys("set_nonblocking");

                    ev.events = EPOLLIN | EPOLLET;
                    ev.data.fd = conn_fd;
                    if (epoll_ctl(epoll_fd, EPOLL_CTL_ADD, conn_fd, &ev) == -1)
                        err_sys("epoll_ctl: conn_fd");
                }
            } else {
                /* 处理现有连接的I/O */
                int conn_fd = events[n].data.fd;
                char buffer[BUFFER_SIZE];
                ssize_t count;
```

```
                if (events[n].events & EPOLLIN) {
                    count = read(conn_fd, buffer, sizeof(buffer));
                    if (count == -1) {
                        if (errno != EAGAIN)
                            err_sys("read");
                    } else if (count == 0) {
                        /* 客户端关闭连接 */
                        close(conn_fd);
                    } else {
                        /* 回显数据 */
                        write(conn_fd, buffer, count);
                    }
                }
            }
        }
    }

    close(listen_fd);
    close(epoll_fd);
    return 0;
}
```