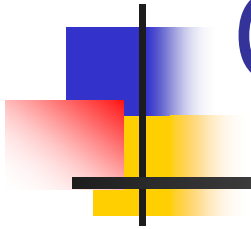


Chapter 3. File I/O

System Programming

http://www.cs.ccu.edu.tw/~pahsiung/courses/sp

熊博安

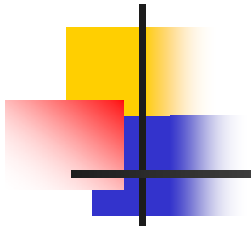
國立中正大學資訊工程學系

pahsiung@cs.ccu.edu.tw

(05)2720411 ext. 33119

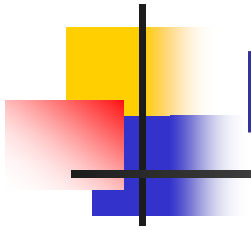
Class: EA-104

Office: EA-512



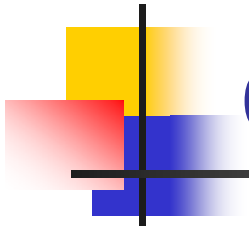
Introduction

- Discussion of unbuffered I/O
- 5 functions: (POSIX.1, XPG3, not ANSI)
  - open()
  - read() (a system call)
  - write() (a system call)
  - lseek()
  - close()
- Sharing of files: dup(), fcntl(), ioctl()



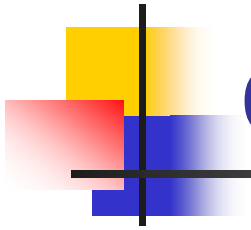
File Descriptors

- A nonnegative integer
- Represents a file
- Referenced by the kernel
- Returned by open() to process
- read(), write() use them
- stdin=0, stdout=1, stderr=2



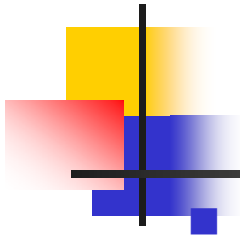
open()

- #include <fcntl.h>
- int open(const char **pathname*, int *oflag*, .../* , mode_t *mode* */);
- Returns: file descriptor if OK, -1 on error



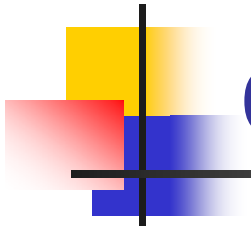
open()

- *pathname*: name of file to open
- *oflag* = OR of the following:
  - O_RDONLY: open for reading only
  - O_WRONLY: open for writing only
  - O_RDWR: open for reading & writing
- Only 1 of the above must be specified



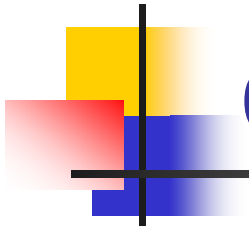
open()

- Other optional constants:
  - O_APPEND: append to end of file
  - O_CREAT: create if not existing, need mode
  - O_EXCL: gen error if file exist and O_CREAT
  - O_TRUNC: truncate length to 0, if exists
  - O_NOCTTY: do not allocate as controlling tty
  - O_NONBLOCK: nonblocking mode for FIFO or character special file
  - O_DSYNC: wait write complete (without waiting file attributes to be updated)
  - O_RSYNC: wait all pending write to complete before read operation
  - O_SYNC: wait write to complete with attributes



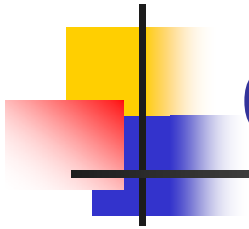
creat()

- #include <fcntl.h>
- int creat(const char *_pathname_,
  mode_t _mode_);
- Returns: file descriptor opened for write-only if OK, -1 on error



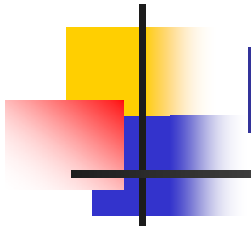
creat()

- Equivalent to:
  open(*pathname*, O_WRONLY |
  O_CREAT | O_TRUNC, *mode*);
- Used when open() could take only 0 (O_RDONLY), 1(O_WRONLY), and 2(O_RDWR).
- **Better**: open(*pathname*, O_RDWR | O_CREAT | O_TRUNC, *mode*);



close()

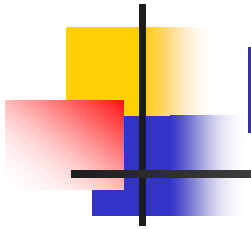
- #include <unistd.h>
- int close(int *filedes*);
- Returns: 0 if OK, -1 on error
- Releases record locks on the file
- All open files automatically closed by kernel, when a process terminates.
- Take advantage: no explicit close()



lseek()

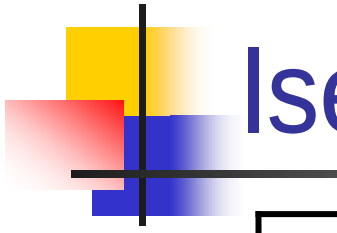
Current file offset:

- nonnegative integer
- #bytes from beginning of file
- read(), write() start from current file offset, incremented by #bytes read/written
- initialized to 0 when file is opened (unless O_APPEND is specified)



lseek()

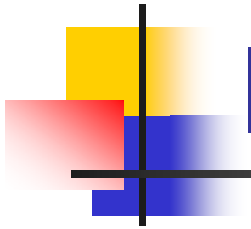
- #include <unistd.h>
- off_t lseek(int *filedes*, off_t *offset*,
  int *whence*);
- Returns: new file offset if OK, -1 on error



lseek()

whence	**New offset (bytes)**
SEEK_SET	file start + *offset*
SEEK_CUR	current offset + *offset*
SEEK_END	file size + *offset*

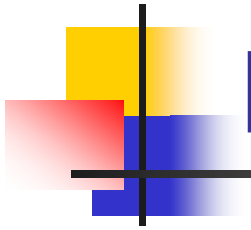
+/-



lseek()

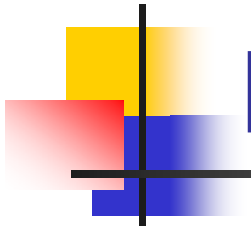
- To determine current offset:
  - off_t curpos;
  - currpos = lseek(fd, 0, SEEK_CUR);
- If fd is a FIFO, pipe, or socket:
  - currpos = –1
  - errno = ESPIPE
- Before System V: whence = 0(absolute), 1(relative to current), 2(relative to EOF)

Hard coded



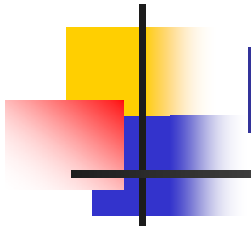
Program 3.1: seeking stdin

```
#include    “apue.h"

int
main(void)
{
  if (lseek(STDIN_FILENO, 0, SEEK_CUR) == -1)
      printf("cannot seek\n");
  else
      printf("seek OK\n");
  exit(0);
}
```



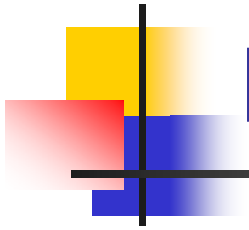
Program 3.1 (output)

- $ **a.out < /etc/motd**
- `seek OK`
- $ **cat < /etc/motd | a.out**
- `cannot seek`
- $ **a.out < /var/spool/cron/FIFO**
- `cannot seek`



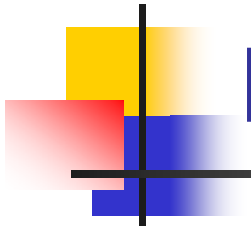
lseek()

- For regular files, offset is usually nonnegative
- For special files, offset can be negative
- Check -1 and not < 0
- lseek() only records current file offset in kernel, no I/O takes place
- offset > file size? OK! Hole in file! All 0!



Program 3.2: Create file with hole

```
#include    <fcntl.h>
#include    “apue.h"

char  buf1[] = "abcdefghij";
char  buf2[] = "ABCDEFGHIJ";

int
main(void)
{
  int       fd;

  if ( (fd = creat("file.hole", FILE_MODE)) < 0)
      err_sys("creat error");
```



Program 3.2: Create file with hole

```
    if (write(fd, buf1, 10) != 10)
        err_sys("buf1 write error");
    /* offset now = 10 */

    if (lseek(fd, 40, SEEK_SET) == -1)
        err_sys("lseek error");
    /* offset now = 40 */

    if (write(fd, buf2, 10) != 10)
        err_sys("buf2 write error");
    /* offset now = 50 */

    exit(0);
}
```

Program 3.2 (output)

- $ a.out
- ls –l file.hole

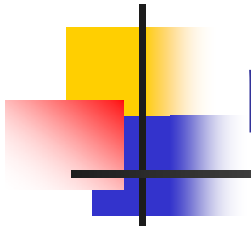
```
-rw-r--r– 1 stevens    50 Jul 31 0:50 file.hole
```

- $ od –c file.hole

byte offset in octal

```
0000000  a b c d e f g h i j  \0 \0 \0 \0 \0 \0
0000020  \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0
         \0 \0 \0
0000040  \0 \0 \0 \0 \0 \0 \0 \0 A B C D E F G H
0000060  I J
```

(–c: print as characters)



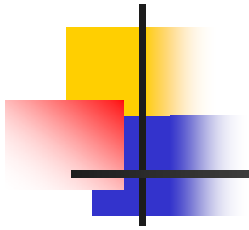
read()

- #include <unistd.h>
- ssize_t read(int *filedes*, void **buf*, size_t, *nbytes*);

signed

unsigned

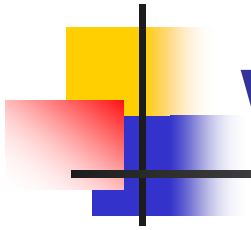
- Returns: #bytes read, 0 if EOF, -1 on error
- #bytes read may be < nbytes



read()

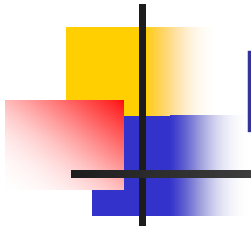
When is #bytes read < nbytes requested?

- EOF reached
- Terminal: 1 line at a time
- Network: buffer size limit
- Pipe or FIFO: pipe contents
- Magnetic tape: single record at a time
- Interrupted by signal: partially read



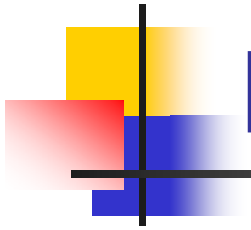
write()

- #include <unistd.h>
- ssize_t write(int filedes, const void *buff, size_t nbytes);
- Returns: #bytes written if OK, -1 on error
- File offset incremented by #bytes written



Program 3.4: stdout ← stdin

```
#include       “apue.h"

#define BUFFSIZE       4096

int
main(void)
{
   int          n;
   char buf[BUFFSIZE];

   while ( (n = read(STDIN_FILENO, buf, BUFFSIZE)) > 0)
        if (write(STDOUT_FILENO, buf, n) != n)
             err_sys("write error");

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

   exit(0);
}
```



Program 3.4 (details)

- stdin and stdout opened automatically by UNIX shell
- STDIN_FILENO (assumed as 0)
- STDOUT_FILENO (assumed as 1)
- stdin and stdout are not closed
- No difference between text and binary files

Timing results for reading with different buffer sizes on Linux

- 103,316,352-byte file
- 20 buffer sizes
- Linux ext2 filesystem
- 4,096-byte blocks
- 512 MB RAM

BUFFSIZE	User CPU (seconds)	System CPU (seconds)	Clock time (seconds)	#loops
1	124.89	161.65	288.64	103,316,352
2	63.10	80.96	145.81	51,658,176
4	31.84	40.00	72.75	25,829,088
8	15.17	21.01	36.85	12,914,544
16	7.86	10.27	18.76	6,457,272
32	4.13	5.01	9.76	3,228,636
64	2.11	2.48	6.76	1,614,318
128	1.01	1.27	6.82	807,159
256	0.56	0.62	6.80	403,579
512	0.27	0.41	7.03	201,789
1,024	0.17	0.23	7.84	100,894
2,048	0.05	0.19	6.82	50,447
4,096	0.03	0.16	6.86	25,223
8,192	0.01	0.18	6.67	12,611
16,384	0.02	0.18	6.87	6,305
32,768	0.00	0.16	6.70	3,152
65,536	0.02	0.19	6.92	1,576
131,072	0.00	0.16	6.84	788
262,144	0.01	0.25	7.30	394
524,288	0.00	0.22	7.35	198

Figure 3.5 Timing results for reading with different buffer sizes on Linux

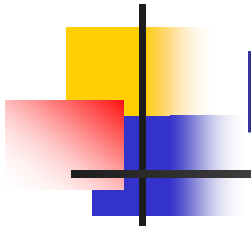
Edition 1 Example: I/O Efficiency (Buffer size?)

BUFFSIZE	User CPU (seconds)	System CPU (seconds)	Clock time (seconds)	#loops
1	23.8	397.9	423.4	1468802
2	12.3	202.0	215.2	734401
4	6.1	100.6	107.2	367201
8	3.0	50.7	54.0	183601
16	1.5	25.3	27.0	91801
32	0.7	12.8	13.7	45901
64	0.3	6.6	7.0	22951
128	0.2	3.3	3.6	11476
256	0.1	1.8	1.9	5738
512	0.0	1.0	1.1	2869
1024	0.0	0.6	0.6	1435
2048	0.0	0.4	0.4	718
4096	0.0	0.4	0.4	359
8192	0.0	0.3	0.3	180
16384	0.0	0.3	0.3	90
32768	0.0	0.3	0.3	45
65536	0.0	0.3	0.3	23
131072	0.0	0.3	0.3	12

8192-byte blocks

No effect on increasing beyond 8192 bytes

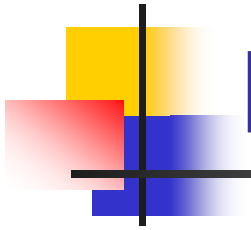
Hence, BUFFSIZE = 8192 bytes



File Sharing

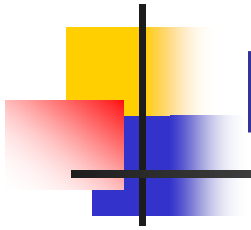
3 data structures used by kernel

- Process Table
- File Table
- V-node Table



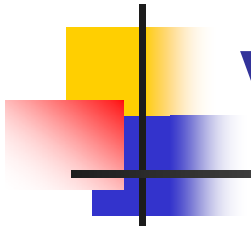
Process Table

- An entry for each process
  - A table of open file descriptors in each entry
    - File descriptor flag (FD_CLOEXEC)
    - A pointer to a file table entry



File Table

- An entry for each open file
  - file status flags: read, write, append, sync, nonblocking, etc.
  - current file offset
  - a pointer to a v-node table entry for the file



V-node Table

Each open file has a v-node structure

- type of file
- pointers to functions operating on file
- i-node for the file:
  - owner,
  - size,
  - device,
  - disk location pointers

File Sharing

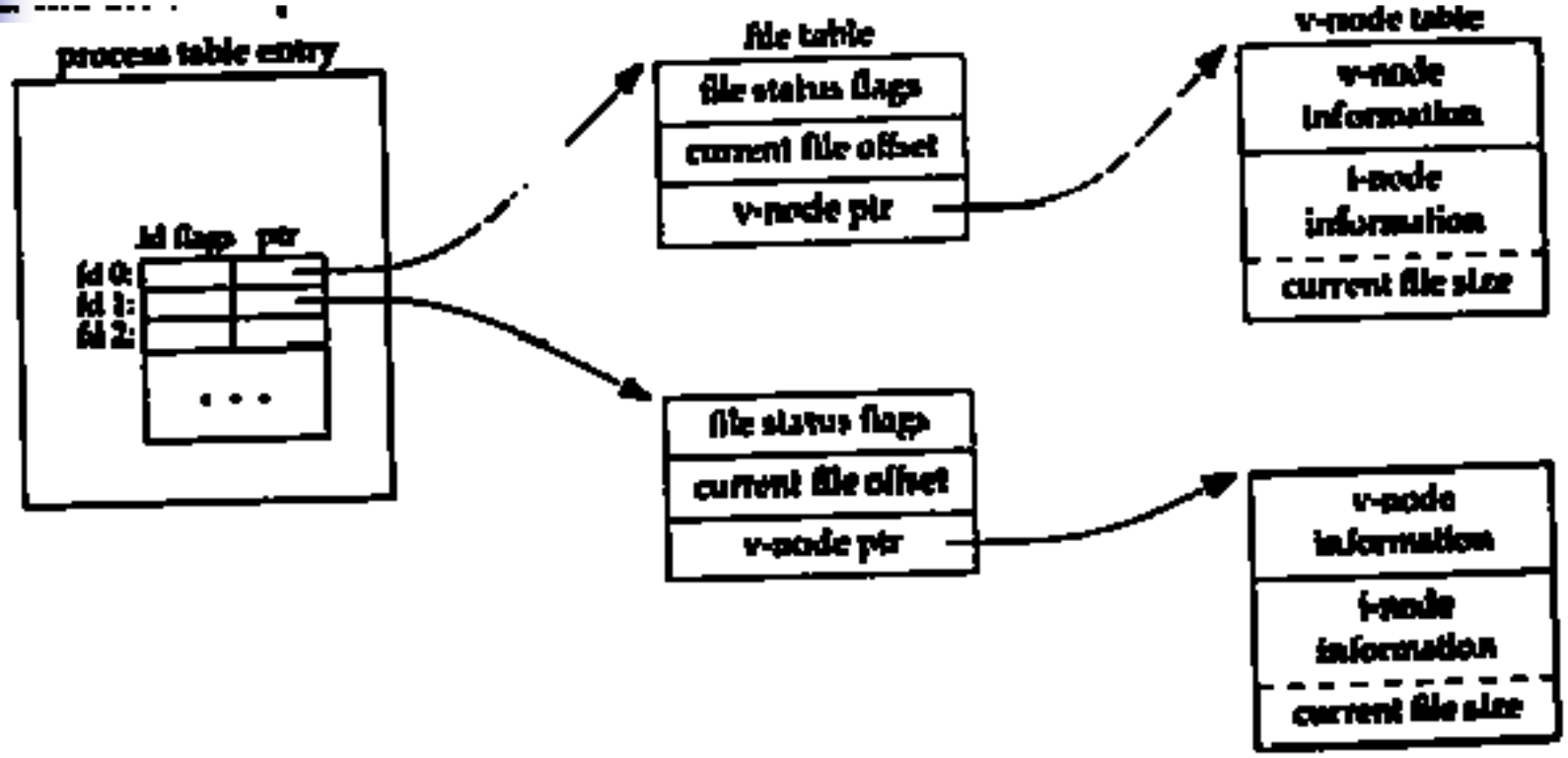
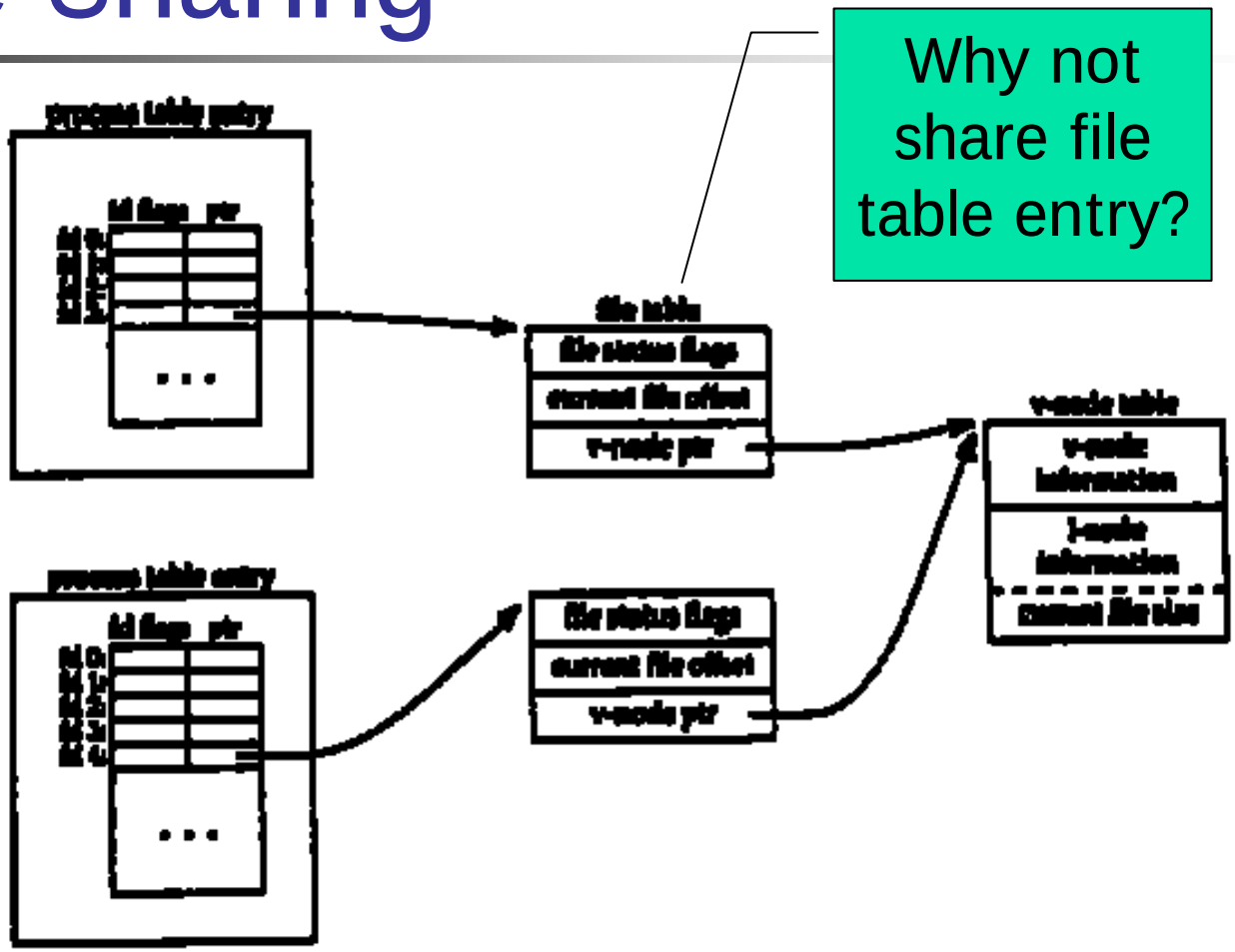
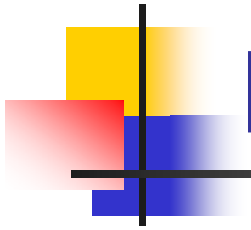


Figure 3.2 Kernel data structures for open files.

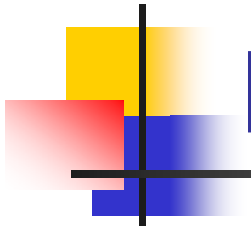
File Sharing





File Sharing

- After each write(), current file offset is incremented by #bytes written.
- If file size increases, current file size in i-node table is updated.
- If a file open() has O_APPEND flag, file status flags are updated and
  before each write, current file offset := current file size (from i-node table)



File Sharing

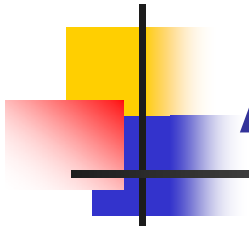
- lseek() only modifies current file offset. No I/O.
- lseek(fd, 0, SEEK_END);
  current file offset := current file size
- All above work fine for multiple processes reading same file.
- For write() of same file by multiple processes, unexpected results can occur

Atomic Operations (APPEND)

- O_APPEND option in open() was not available before, for appending to file:

```
if(lseek(fd, 0L, 2)<0)  /*position to EOF*/
        err_sys("lseek error");
if(write(fd, buff, 100) != 100)
        err_sys("write error");
```

2 actions

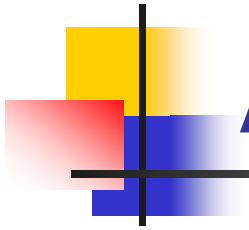


Atomic Operations (APPEND)

Scenario

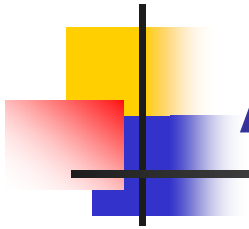
Processes A & B appending to same file

- Set file offset to EOF (1500) for A
- Kernel switches to Process B
- Set file offset to EOF (1500) for B
- Write by B, file size increased to 1600
- Kernel switches back to Process A
- Write by A, data of B is **overwritten**!!!



Atomic Operations (APPEND)

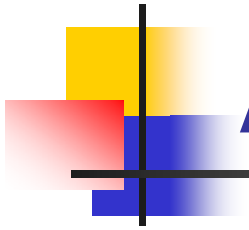
- What is the problem?
- Append is implemented as 2 actions:
  - position to EOF
  - write to file
- Kernel may switch from one process to another **between those two actions**
- Solution: let append be **ATOMIC**!!!



Atomic Operations (CREATE)

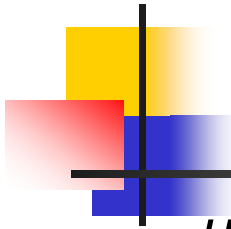
- open() with O_CREAT, O_EXCL fails if file exists
- This is an ATOMIC operation!
- Can we implement it as follows?

```
if ( (fd=open(pathname, O_WRONLY)<0)
   if (errno==ENOENT) {
        if ((fd=creat(pathname, mode)) < 0)
              err_sys("creat error");
   } else err_sys ("open error");
```



Atomic Operations

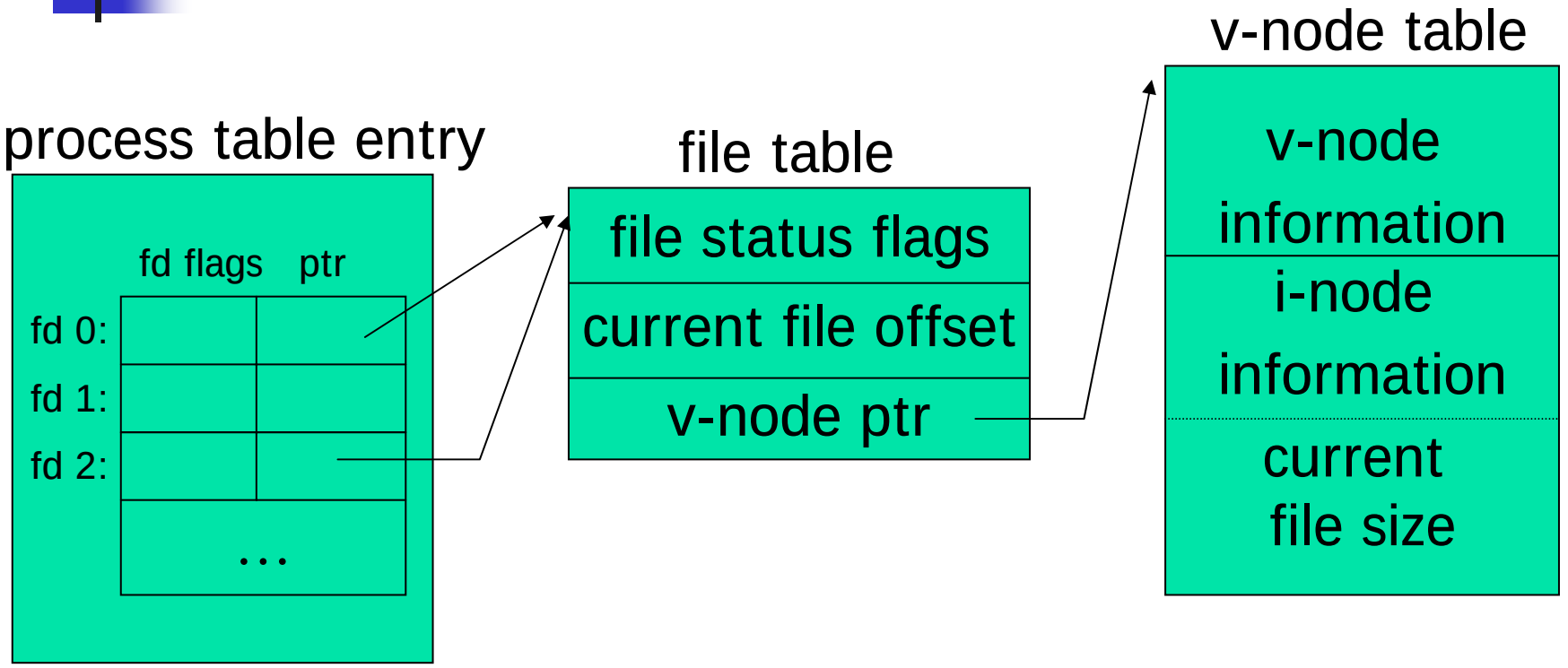
- What if the file is created by another process between open() and creat()?
- Any data written by the other process will be erased by the creat() of first process.
- Thus, we need ATOMIC file creation
- ATOMIC = Either all operations are performed or none (NO SUBSETS!)

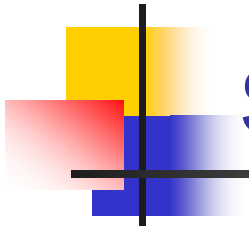


dup() and dup2()

- #include <unistd.h>
- int dup (int *filedes*);
- int dup2 (int *filedes*, int *filedes2*);
- Return: new file descriptor if OK, -1 on error
- Duplicate *filedes* and return as new file descriptor,
  - dup: lowest numbered file descriptor
  - dup2: specified as *filedes2*

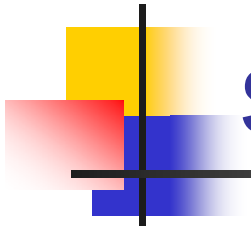
Kernel data structures on dup(1)





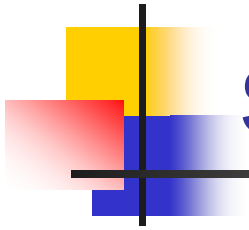
sync, fsync, fdatasync

- For efficient I/O, UNIX system has buffer cache or page cache
- Delayed write
  - Data is first written to buffer
- Kernel eventually writes all delayed-write blocks to disk
  - sync, fsync, fdatasync



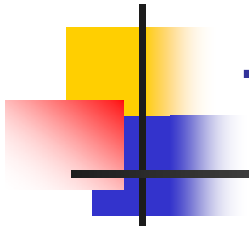
sync, fsync, fdatasync

- `#include <unistd.h>`
- `int fsync(int filedes);`
- `int fdatasync(int filedes);`
- `void sync(void);`
- Returns: 0 if OK, -1 on error



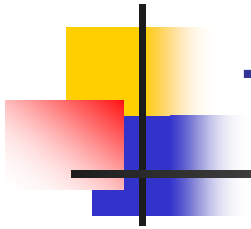
sync, fsync, fdatasync

- sync
  - Queues write requests for the modified blocks
  - Called by “update” daemon every 30 seconds
  - Function, command
- fsync
  - Single file, waits for disk writes to complete (data + attributes all updated)
- fdatasync
  - Single file, waits for data update only, not file attributes



fcntl()

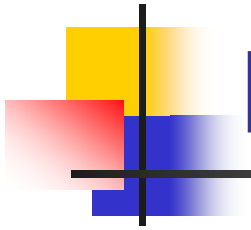
- #include <fcntl.h>
- int fcntl (int *filedes*, int *cmd*, ...
  /* int *arg* */);
- Returns: depends on cmd if OK,
  -1 on error



fcntl()

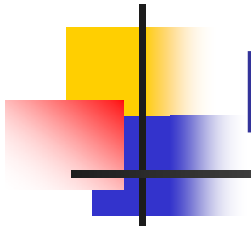
Change properties of an opened file

cmd	fcntl() results
F_DUPFD	duplicate an existing descriptor
F_GETFD or F_SETFD	get/set file descriptor flags
F_GETFL or F_SETFL	get/set file status flags
F_GETOWN or F_SETOWN	get/set async I/O owner
F_GETLK, F_SETLK, or F_SETLKW	get/set record locks



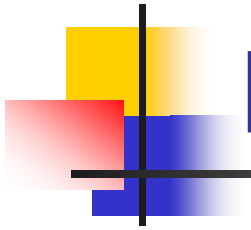
Program 3.10: print file flags

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

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

    if (argc != 2)
        err_quit("usage: a.out <descriptor#>");

    if ( (val = fcntl(atoi(argv[1]), F_GETFL, 0)) < 0)
        err_sys("fcntl error for fd %d", atoi(argv[1]));
```



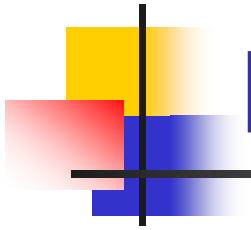
Program 3.10: print file flags

```
    switch(val & O_ACCMODE) {
    case O_RDONLY:      printf("read only"); break;
    case O_WRONLY:      printf("write only"); break;
    case O_RDWR:        printf("read write"); break;
    default:            err_dump("unknown access mode");
    }

    if (val & O_APPEND)             printf(", append");
    if (val & O_NONBLOCK)           printf(", nonblocking");
#if defined(O_SYNC)
    if (val & O_SYNC)               printf(", synchronous writes");
#endif
#if !defined(_POSIC_C_SOURCE) && defined(O_FSYNC)
    if (val & O_FSYNC)              printf(“, synchronous writes”);
#endif
    putchar('\n');
    exit(0);
}
```



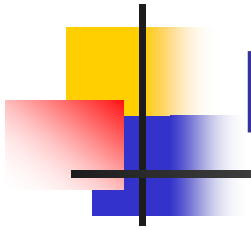
Program 3.10: (output)

- **$ a.out 0 < /dev/tty**
- read only
- **$ a.out 1 > temp.foo**
- **$ cat temp.foo**
- write only
- **$ a.out 2 2>> temp.foo**
- write only, append
- **$ a.out 5 5<>temp.foo**
- read write



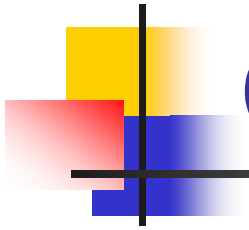
Program 3.5: set flags

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

void
set_fl(int fd, int flags) /* flags are file status flags to turn on */
{
    int             val;

    if ( (val = fcntl(fd, F_GETFL, 0)) < 0)
        err_sys("fcntl F_GETFL error");

    val |= flags;           /* turn on flags */

    if (fcntl(fd, F_SETFL, val) < 0)
        err_sys("fcntl F_SETFL error");
}
```



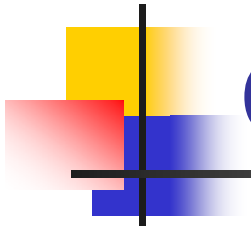
Program 3.5: clear flags

- How to clear flags for a file?
- val &= ~flags; /* turn flags off */



O_SYNC

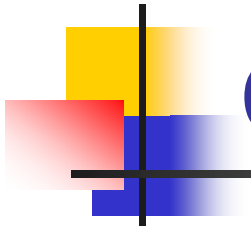
- set_fl(STDOUT_FILENO, O_SYNC);
- Add the above line to the start of Program 3.4
- Each write is sync'ed.



O_SYNC (copying 98.5 MB data)

	Operation	User CPU (s)	System CPU (s)	Clock time (s)
> /dev/null	read time from Fig 3.5 for BUFFSIZE=4096	0.03	0.16	6.86
> file	normal write to disk file	0.02	0.30	6.87
Linux does not honor O_SYNC	write to disk file with O_SYNC set	0.03	0.30	6.83
	write to disk followed by fdatasync	0.03	0.42	18.28
	write to disk followed by fsync	0.03	0.37	17.95
	write to disk with O_SYNC set followed by fsync	0.05	0.44	17.95

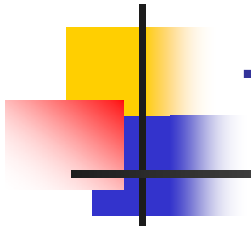
Linux system with ext2 filesystem, BUFFSIZE=4096



O_SYNC (copying 98.5 MB data)

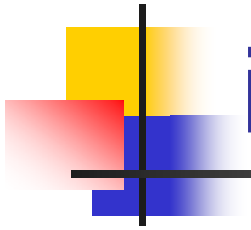
Operation	User CPU (s)	System CPU (s)	Clock time (s)
write to /dev/null	0.06	0.79	4.33
normal write to disk file	0.05	3.56	14.40
write to disk file with O_FSYNC set	0.13	9.53	22.48
write to disk followed by fsync	0.11	3.31	14.12
write to disk with O_FSYNC set followed by fsync	0.17	9.14	22.12

Mac OS X 10.3 system



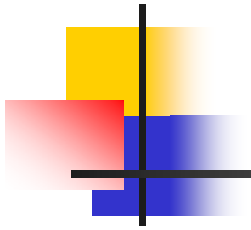
fcntl()

- Is fcntl() really needed?
- The O_SYNC example: we do not know which file is the stdout, we only know the descriptor STDOUT_FILENO
- Pipe: All we have with a pipe is a descriptor



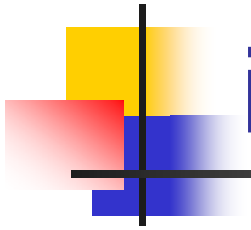
ioctl()

- #include <unistd.h> /* SVR4 */
- #include <sys/ioctl.h> /* BSD, Linux*/
- #include <stropts.h> /* XSI STREAMS */
- int ioctl(int *filedes*, int *request*, …);
- Returns: -1 on error, something else if OK
- POSIX.1 prototype



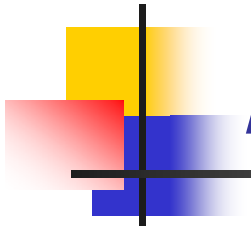
Ioctl commands (FreeBSD)

Category	Constant	Header	#ioctls
Disk labels	DIOxxx	<sys/disklabel.h>	6
File I/O	FIOxxx	<sys/filio.h>	9
Mag tape I/O	MTIOxxx	<sys/mtio.h>	11
Socket I/O	SIOxxx	<sys/sockio.h>	60
Terminal I/O	TIOxxx	<sys/ttycom.h>	44



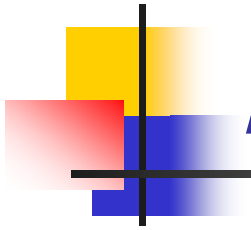
ioctl()

- It can be used on magnetic tapes to:
  - write end-of-file marks
  - rewind tape
  - space forward over #files or #records
- No other function (read, write, lseek) can be used for the above
- Other uses: Sections 14.4, 18.12, 19.7



/dev/fd

- Files in /dev/fd are named 0, 1, 2, …
- open(“/dev/fd/n”, mode) = dup(n)
- mode is mostly ignored or must be a subset of original mode for n
- Some systems:
  /dev/stdin = /dev/fd/0
  /dev/stdout = /dev/fd/1
  /dev/stderr = /dev/fd/2



/dev/fd

- filter file2 | cat file1 – file3 | lpr

  (– = standard input)

- Can also be given as:
- filter file2 | cat file1 /dev/fd/0 file2 | lpr
- /dev/fd is for uniformity and cleanliness