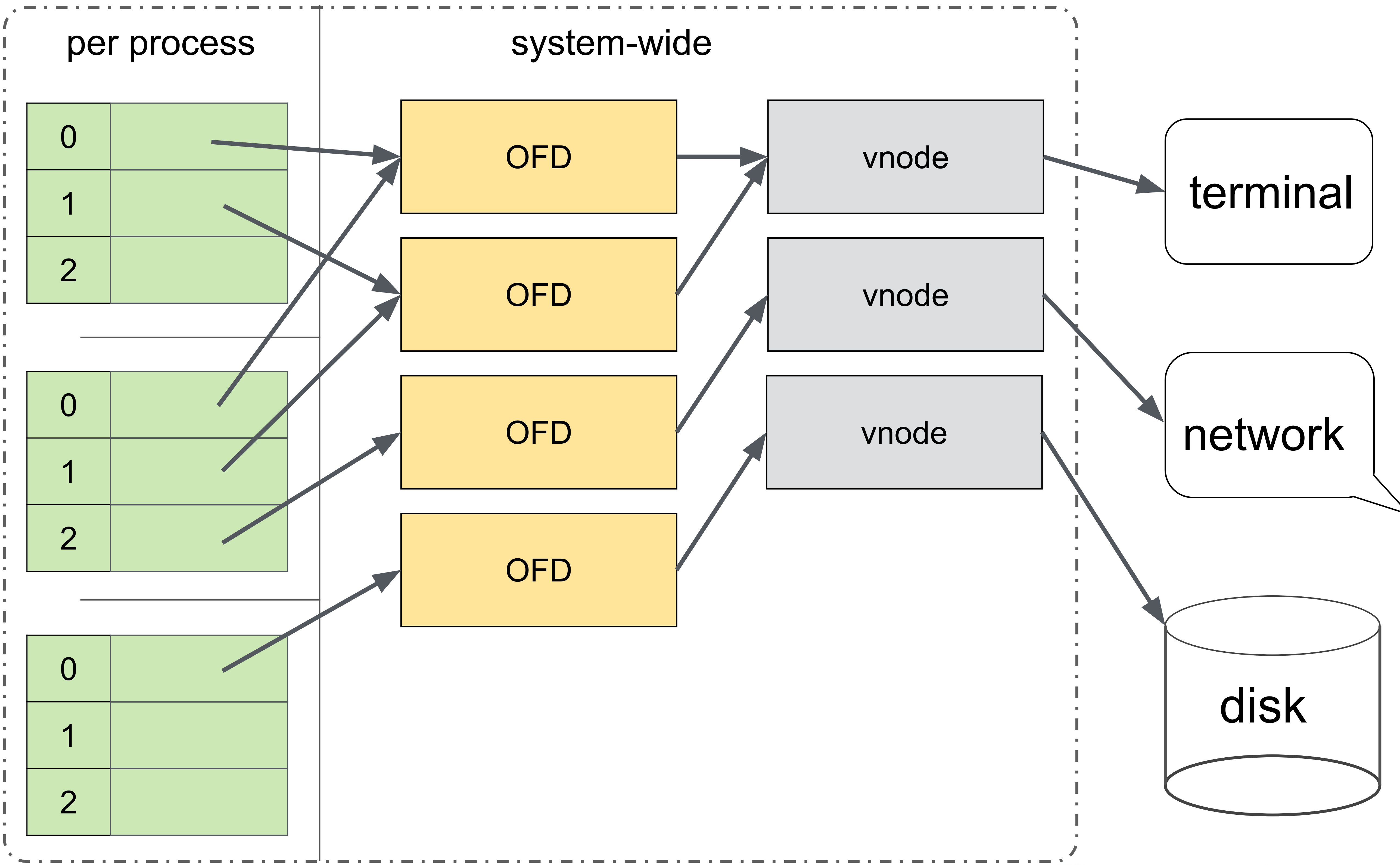
Input/Output

CS 351: Systems Programming
Melanie Cornelius

Slides and course content obtained with permission from Prof. Michael Lee, <lee@iit.edu>

IIT College of Science
ILLINOIS INSTITUTE OF TECHNOLOGY

kernel space



FDs *obscure* kernel I/O & FS implementation details from the user, and enable an *elegant, abstract* I/O API

Some mini-quizzes

§ System-level I/O API

```
int      open ( const char *path, int oflag, ... );
int      fstat( int fd, struct stat *buf );
int      dup  ( int fd );
int      dup2 ( int fd1, int fd2 );
int      close( int fd );
off_t    lseek( int fd, off_t offset, int whence );
ssize_t read  ( int fd, void *buf, size_t nbytes );
ssize_t write( int fd, const void *buf, size_t nbytes );
```

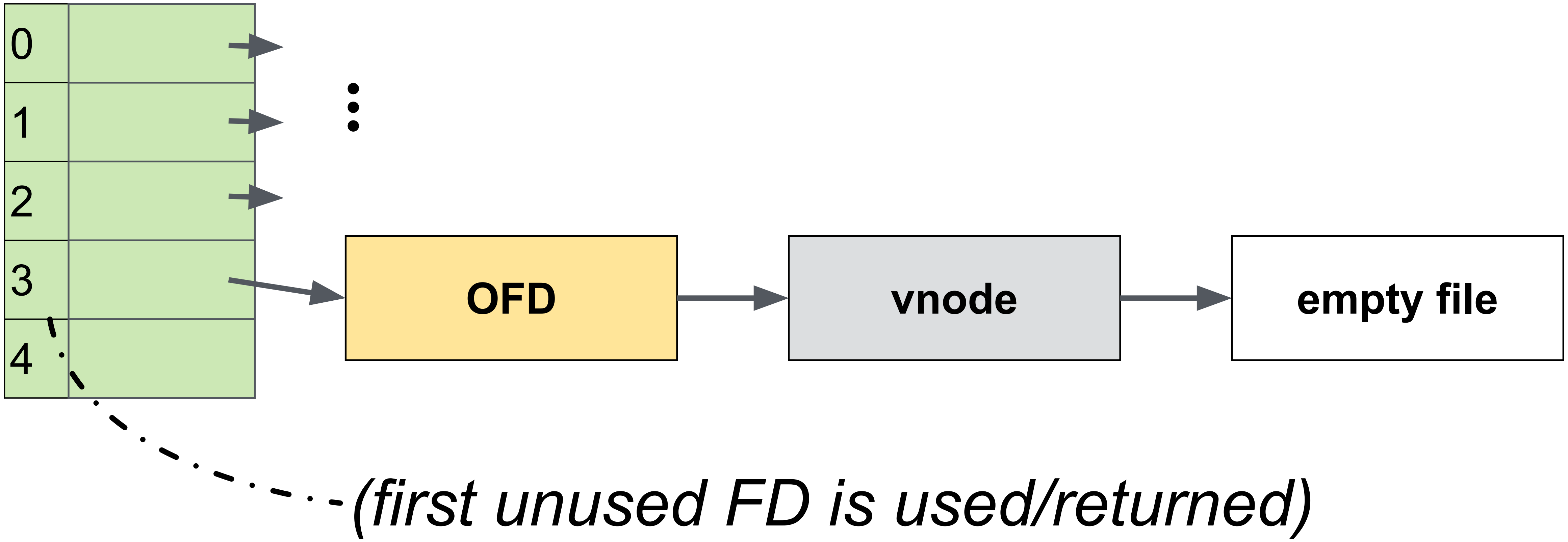
```
int open(const char *path,
         int oflag, ...);
```

- loads *vnode* for file at path (if not already loaded)
- creates & inits a new OFD
- returns a **FD** referring to the new OFD

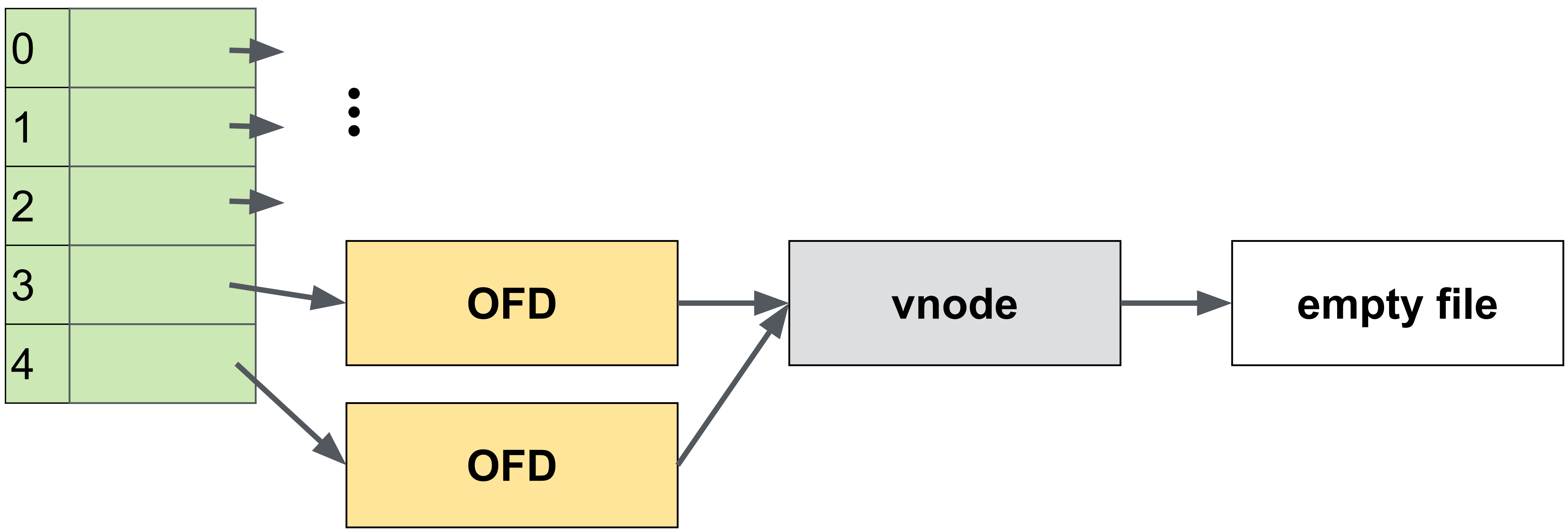
```
int open(const char *path,
         int oflag, ...);
```

- `oflag` is an *or*-ing of `O_RDONLY`, `O_WRONLY`, `O_RDWR`, `O_CREAT`, `O_TRUNC`, etc.
- if `O_CREAT`, must specify access permissions of new file (“rwx” flags)

```
int fd1 = open("foo.txt", O_CREAT | O_TRUNC | O_RDWR, 0644);
```



```c
int fd1 = open("foo.txt", O_CREAT | O_TRUNC | O_RDWR, 0644);
int fd2 = open("foo.txt", O_RDONLY);
```

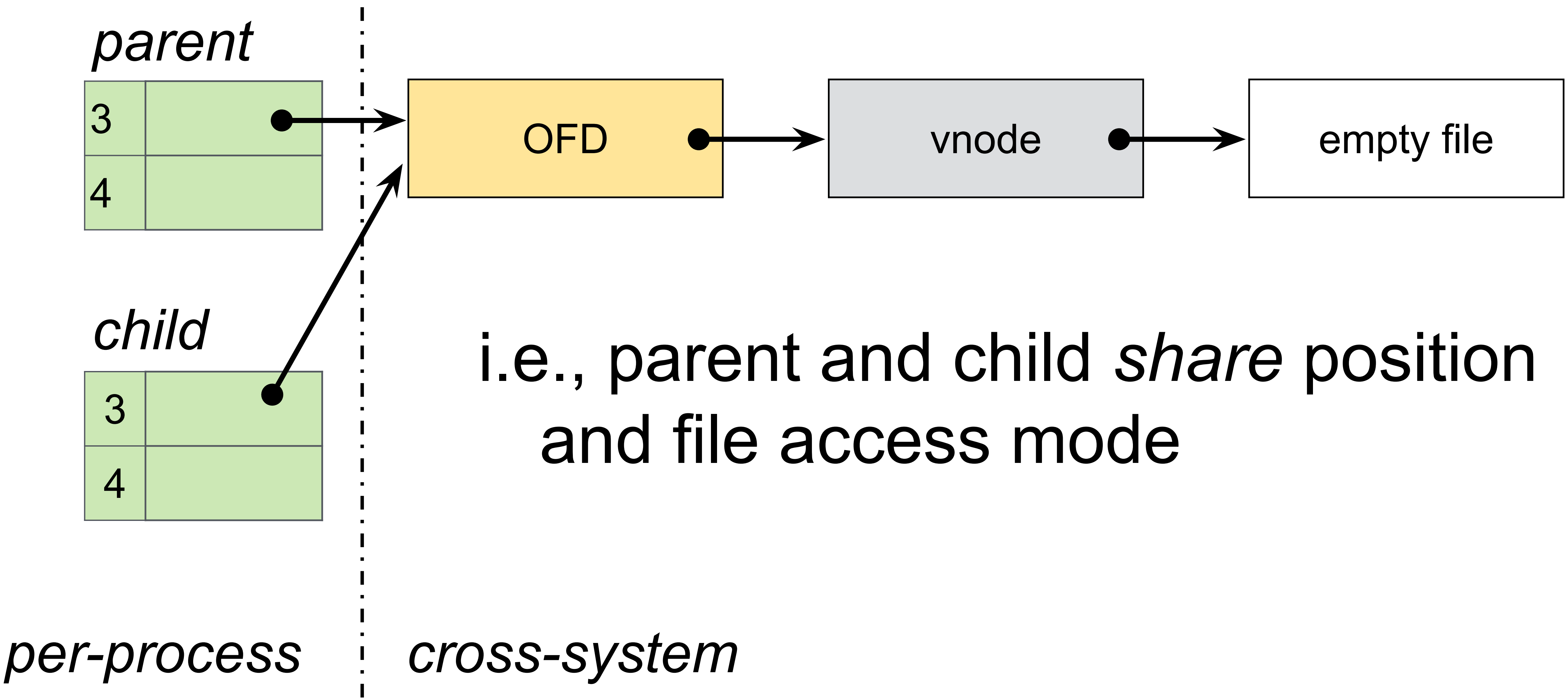


```c
int fd1 = open("foo.txt", O_CREAT | O_TRUNC | O_RDWR, 0644);

struct stat stat;

/* query file metadata */
fstat(fd1, &stat);

printf("Inode # : %lu\n", stat.st_ino);
printf("Size    : %lu\n", stat.st_size);
printf("Links   : %lu\n", stat.st_nlink);
```

```
Inode # : 19603149
Size    : 0
Links   : 1
```

a process inherits its parent's open files across a fork, and *retains them post*-exec!

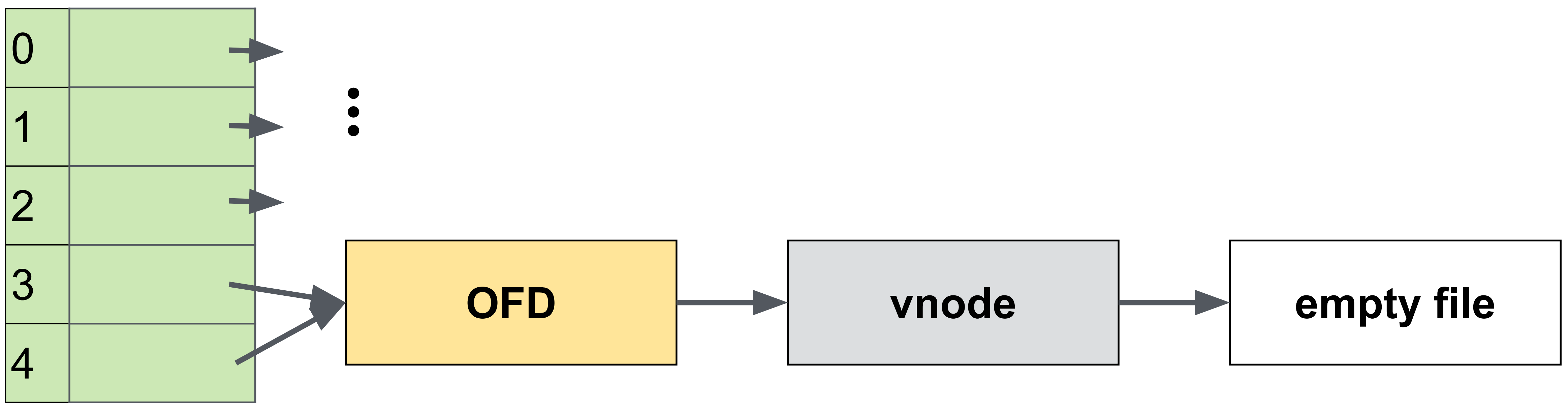
```c
int fd1 = open("foo.txt", O_CREAT | O_TRUNC | O_RDWR, 0644);
fork();
```



i.e., parent and child *share* position and file access mode sharing an OFD can be very handy — e.g., for coordinating output to terminal

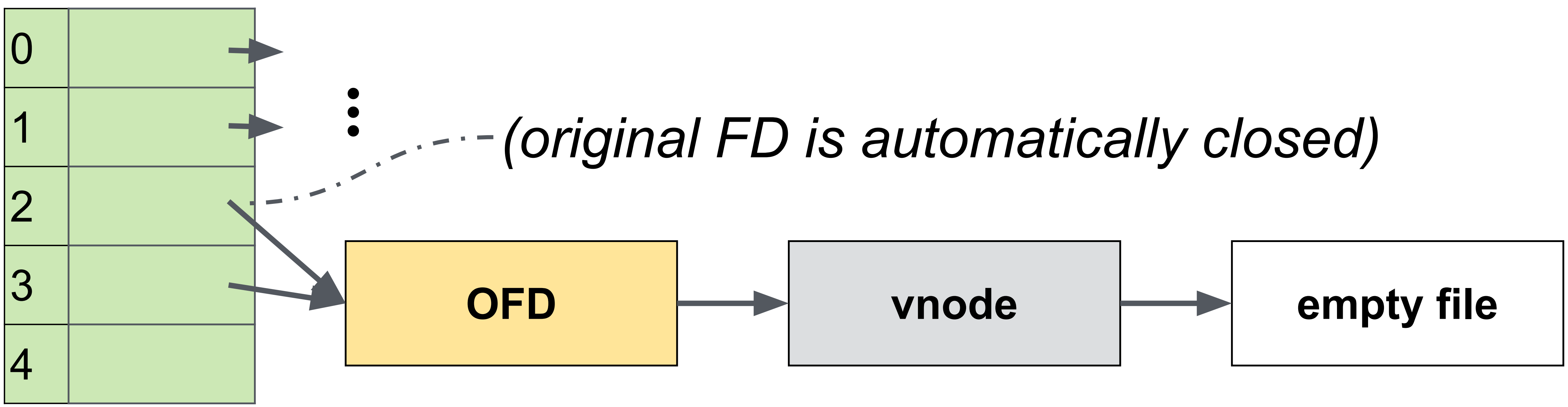
can also explicitly “share” position from separate FDs using dup syscalls

```
int fd1 = open("foo.txt", O_CREAT | O_TRUNC | O_RDWR, 0644);
int fd2 = dup(fd1);
```



i.e., reading/writing FD 4 is equivalent to doing so with FD 3

```c
int fd1 = open("foo.txt", O_CREAT | O_TRUNC | O_RDWR, 0644);
dup2(fd1, 2); /* second arg is "destination" fd */
```

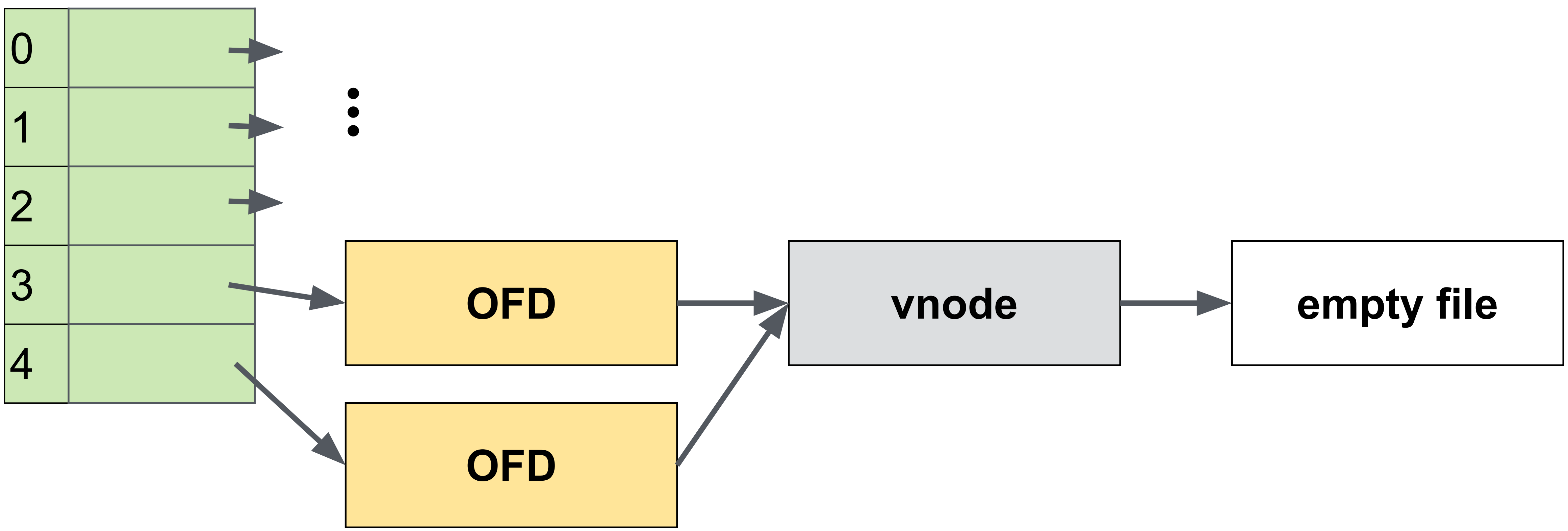


i.e., reading/writing FD 2 (*stderr*) is equivalent to doing so with FD 3

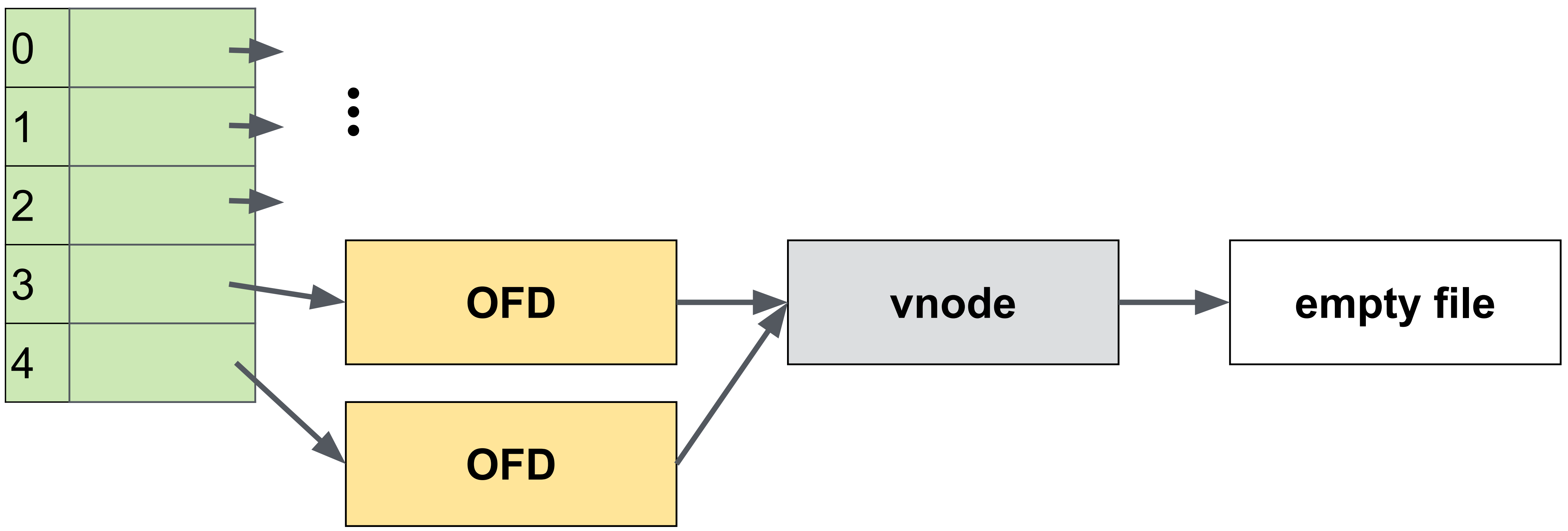
```c
int close(int fd);
```

- delete OFD pointer in file table for fd
- if the OFD has no referring FDs (in any process), deallocate it

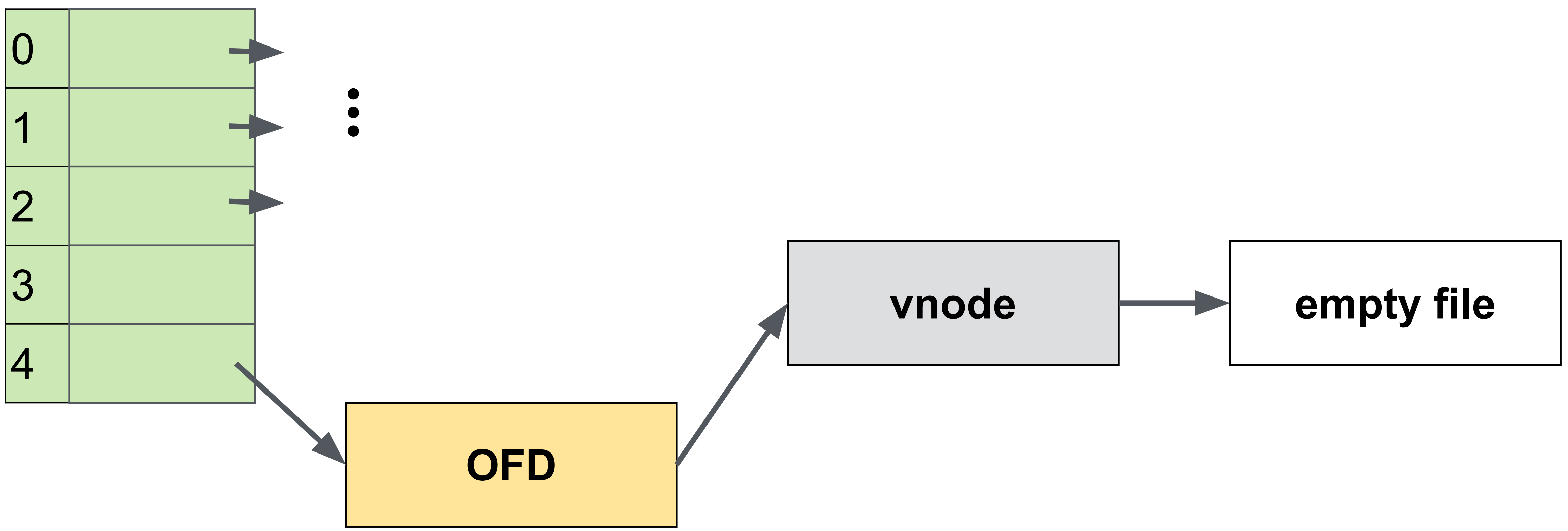
```
int fd1 = open("foo.txt", O_CREAT | O_TRUNC | O_RDWR, 0644);
int fd2 = open("foo.txt", O_RDONLY);
```



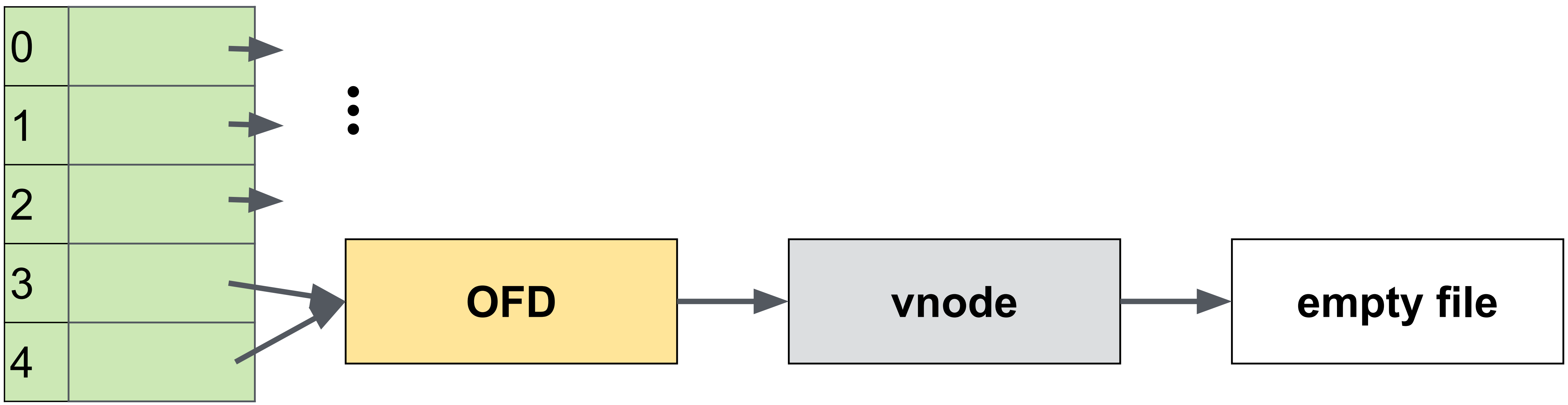
```c
int fd1 = open("foo.txt", O_CREAT | O_TRUNC | O_RDWR, 0644);
int fd2 = open("foo.txt", O_RDONLY);
close(fd1);
```



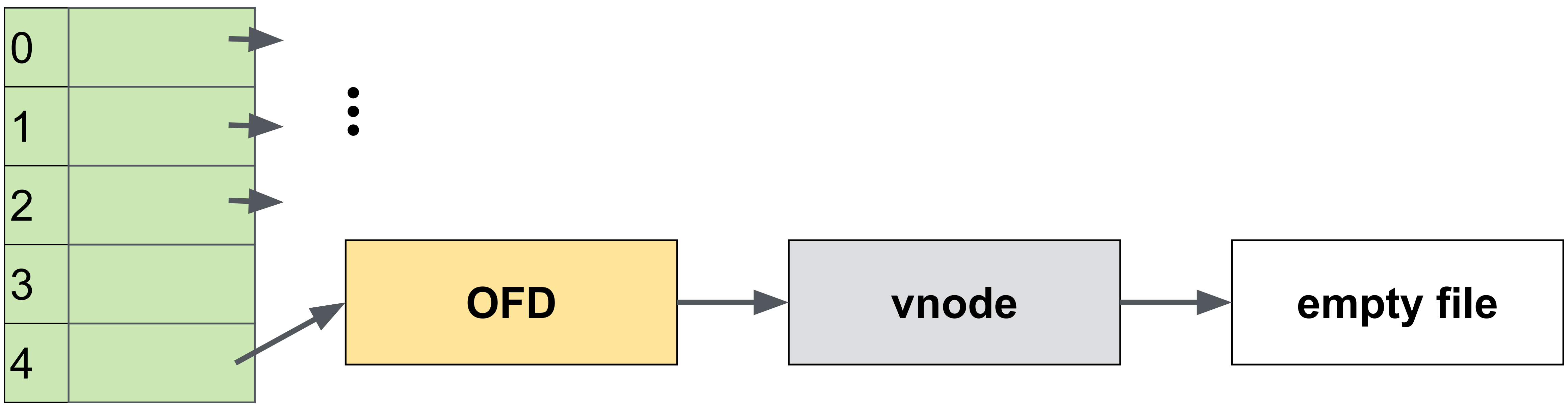
```c
int fd1 = open("foo.txt", O_CREAT | O_TRUNC | O_RDWR, 0644);
int fd2 = open("foo.txt", O_RDONLY);
close(fd1);
close(fd2);
```



```c
int fd1 = open("foo.txt", O_CREAT | O_TRUNC | O_RDWR, 0644);
int fd2 = dup(fd1);
close(fd1);
```



```
int fd1 = open("foo.txt", O_CREAT | O_TRUNC | O_RDWR, 0644);
int fd2 = dup(fd1);
close(fd1);
close(fd2);
```



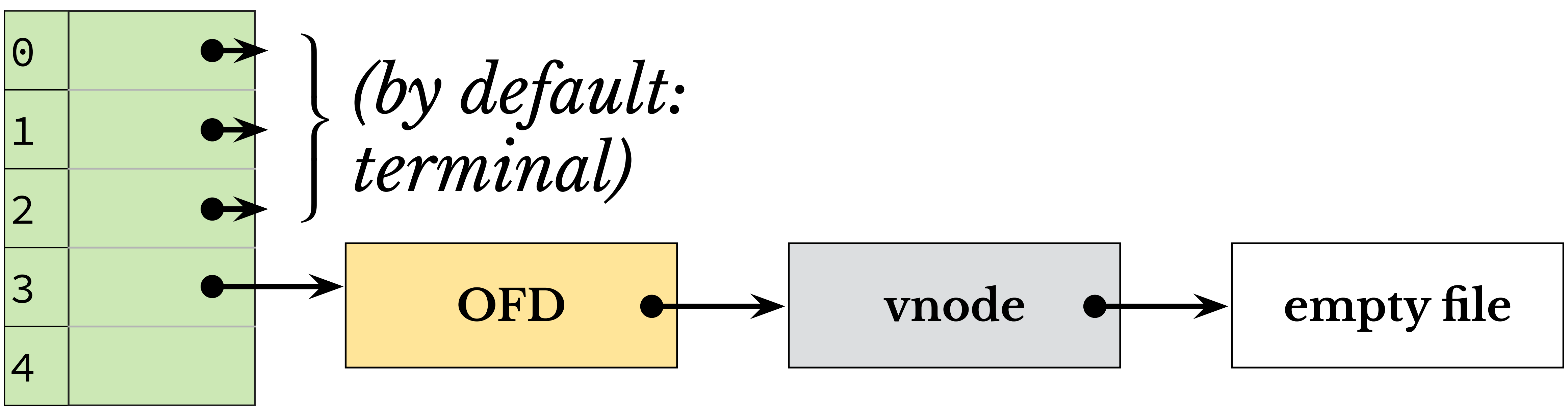
application: *input/output redirection*

- leverage FD usage conventions
- 0 = *stdin*, 1 = *stdout*, 2 = *stderr*
  - recall: FD says nothing about the actual file/device it refers to!

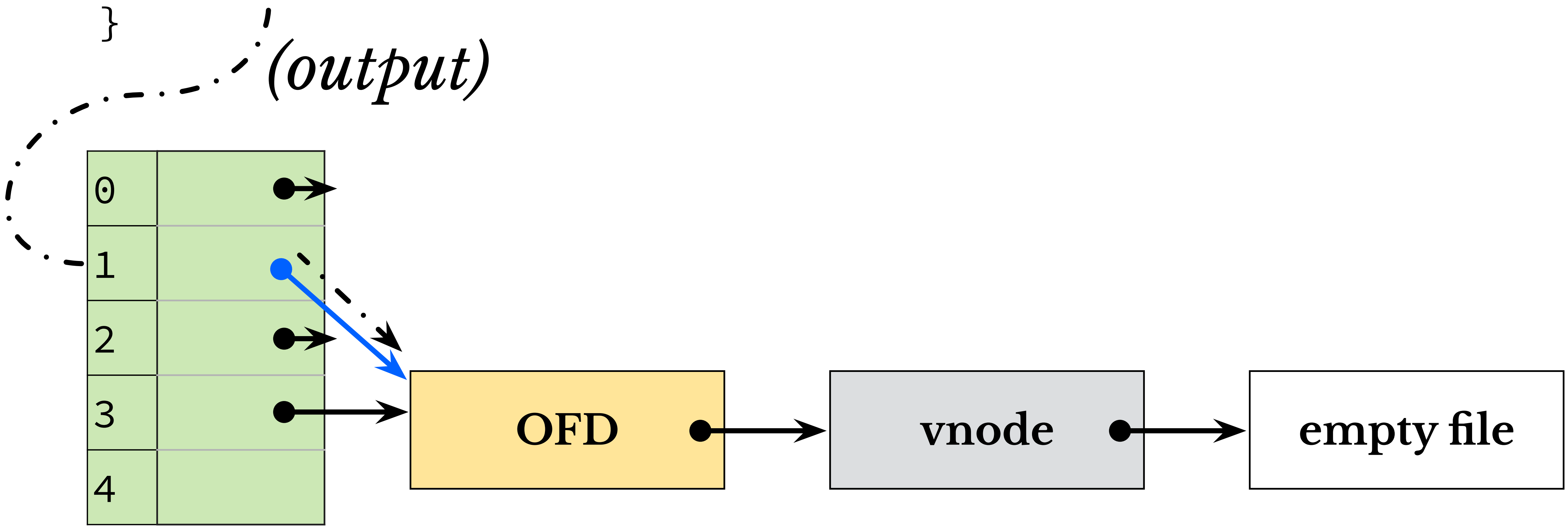
```c
int main(int argc, char *argv[]) {
    int fd = open("foo.txt", O_CREAT|O_TRUNC|O_RDWR, 0644);
    dup2(fd, 1);
    printf("Arg: %s\n", argv[1]);
}
```

```
$ ./a.out hello!
$ ls -l foo.txt
-rw-r--r-- 1 lee staff 12 Feb 19 20:36 foo.txt
$ cat foo.txt
Arg: hello!
```

```c
int main(int argc, char *argv[]) {
    int fd = open("foo.txt", O_CREAT|O_TRUNC|O_RDWR, 0644);
    dup2(fd, 1);
    printf("Arg: %s\n", argv[1]); /* printf prints to stdout */
}
```



```
int main(int argc, char *argv[]) {
    int fd = open("foo.txt", O_CREAT|O_TRUNC|O_RDWR, 0644);
    dup2(fd, 1);
    printf("Arg: %s\n", argv[1]); /* printf prints to stdout */
}
```



```c
int main() {
    int fd = open("foo.txt", O_CREAT|O_TRUNC|O_RDWR, 0644);
    if (fork() == 0) {
        dup2(fd, 1);
        execlp("echo", "echo", "hello!", NULL);
    }
    close(fd);
}
```

```
$ ./a.out
$ cat foo.txt
hello!
```

```
int main() {
    int fd = open("foo.txt", O_CREAT|O_TRUNC|O_RDWR, 0644);
    if (fork() == 0) {
        dup2(fd, 1);
        execlp("echo", "echo", "hello!", NULL);
    }
    close(fd);
}
```

illustrates a powerful technique that requires separating `fork` & `exec`

- original program sets up *new process environment* before `exec`-ing

```
ssize_t read(int fd, void *buf,
             size_t nbytes);
```

- reads up to `nbytes` bytes *from* open file at `fd` *into* `buf`
- returns # bytes read (or -1 for error)

```
ssize_t write(int fd, const void *buf,
              size_t nbytes);
```

- writes *into* open file at fd *from* buf up to `nbytes` bytes
- returns # bytes written (or -1 for error)

“up to `nbytes` bytes”

i.e., *short counts* can occur

— process asks OS to write *k* bytes, but only *j* < *k* bytes are actually written why?

reads:

- EOF, unreadable FD, “slow” file, interrupt, etc.

writes:

- out of space, unwritable FD, “slow” file, interrupt, etc.

`read/write` are the lowest index I/O calls

— kernel objective is to support *maximum performance* & *minimum latency*

e.g., if reading from slow network, return to process asap and allow it to decide to read again or do something else

(but usually, short counts are a royal pain)

```
ssize_t robust_read(int fd, void *buf, size_t n) {
    size_t  nleft = n;
    ssize_t nread;
    char *p = buf;

    while (nleft > 0) {
        if ((nread = read(fd, p, nleft)) < 0)
            return -1; /* error in read */
        else if (nread == 0)
            break;     /* read returns 0 on EOF */
        nleft -= nread;
        p += nread;
    }

    return (n - nleft);
}
```

(yuck)

good news: short counts only occur on EOF for reads on regular files but there’s another concern…

```c
char buf[10];
int fd, x, y, z;

fd = open("data.txt", O_RDONLY);

read(fd, buf, 2);  buf[2] = 0;
x = atoi(buf);

read(fd, buf, 2);  buf[2] = 0;
y = atoi(buf);

read(fd, buf, 2);  buf[2] = 0;
z = atoi(buf);

printf("%d %d %d", x, y, z);
```

data.txt

```
102030
```

```
10 20 30
```

```c
char buf[10];
int fd, x, y, z;

fd = open("data.txt", O_RDONLY);

read(fd, buf, 2);  buf[2] = 0;
x = atoi(buf);

read(fd, buf, 2);  buf[2] = 0;
y = atoi(buf);

read(fd, buf, 2);  buf[2] = 0;
z = atoi(buf);

printf("%d %d %d", x, y, z);
```

data.txt

```
102030
```

one syscall per integer read = inefficient!!!

```
fd = open("data.txt", O_RDONLY);

read(fd, buf, 2);  buf[2] = 0;
x = atoi(buf);

read(fd, buf, 2);  buf[2] = 0;
y = atoi(buf);

read(fd, buf, 2);  buf[2] = 0;
z = atoi(buf);

printf("%d %d %d", x, y, z);
```

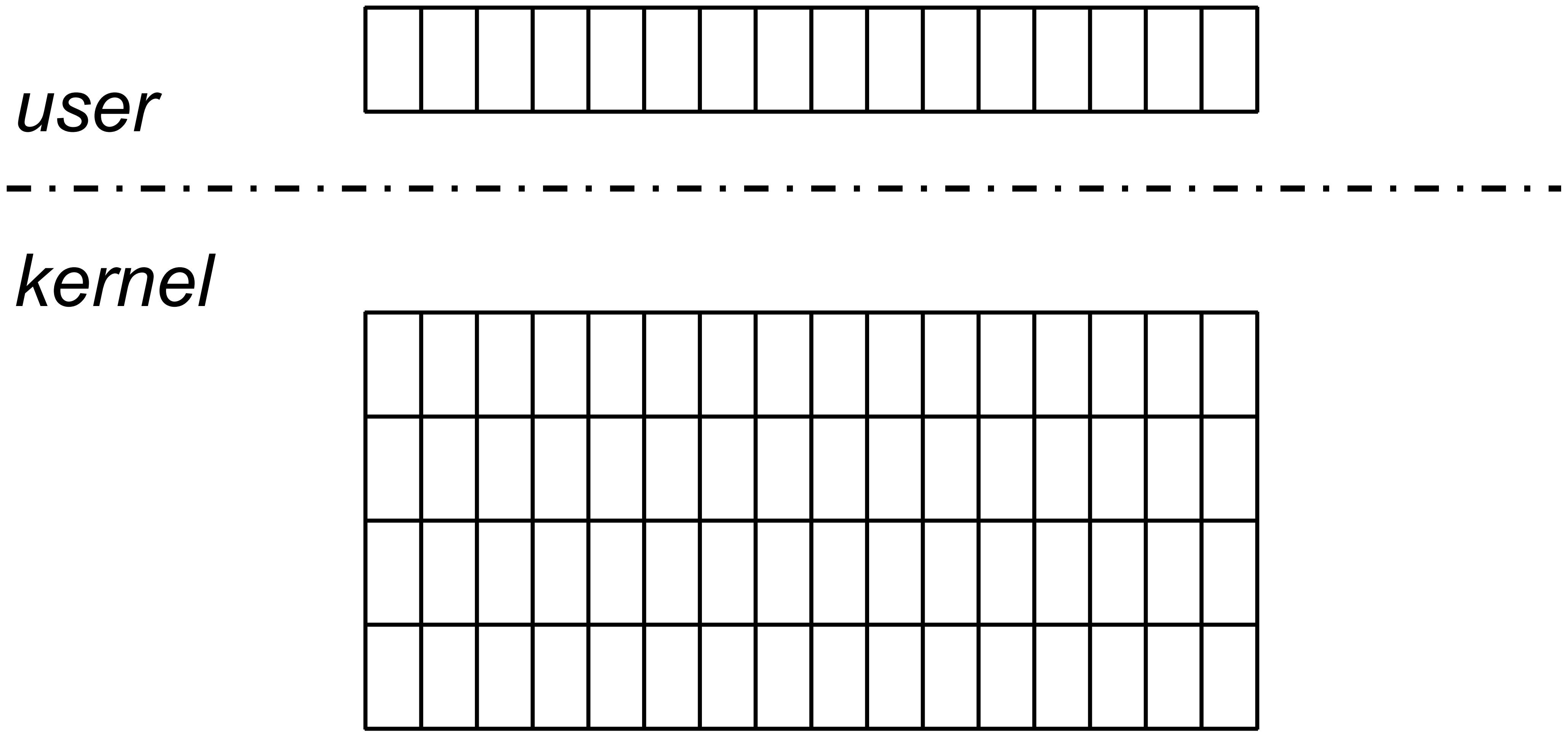
data.txt

```
102030
```

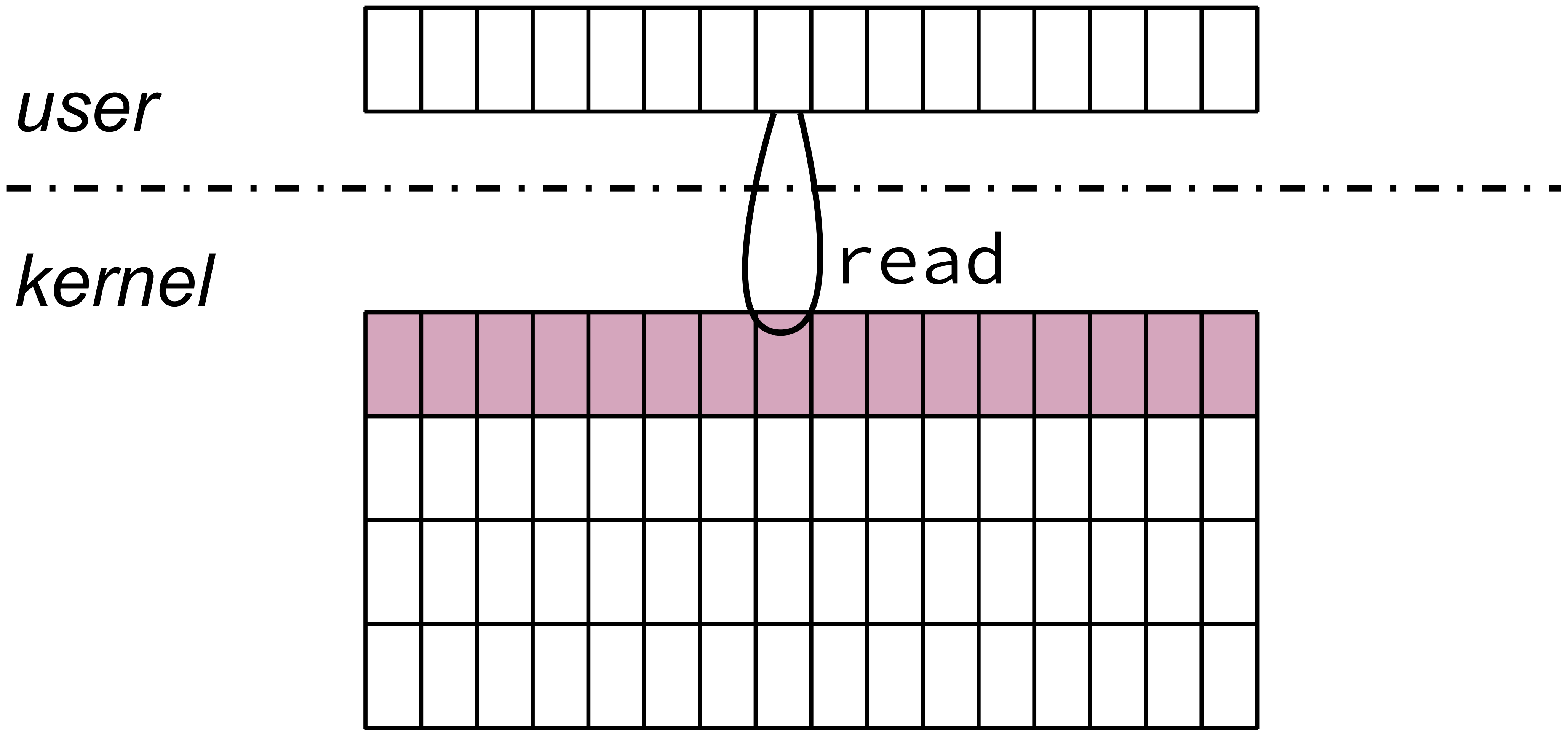
```
$ strace ./a.out
execve("./a.out", ["./a.out"], [/* 67 vars */]) = 0
...
open("data.txt", O_RDONLY)                = 3
read(3, "10", 2)                          = 2
read(3, "20", 2)                          = 2
read(3, "30", 2)                          = 2
write(1, "10 20 30", 8)                   = 8
...
```

solution: *buffering*

step 1: read more bytes than we need into a separate *backing buffer*



step 1: read more bytes than we need into a separate *backing buffer*

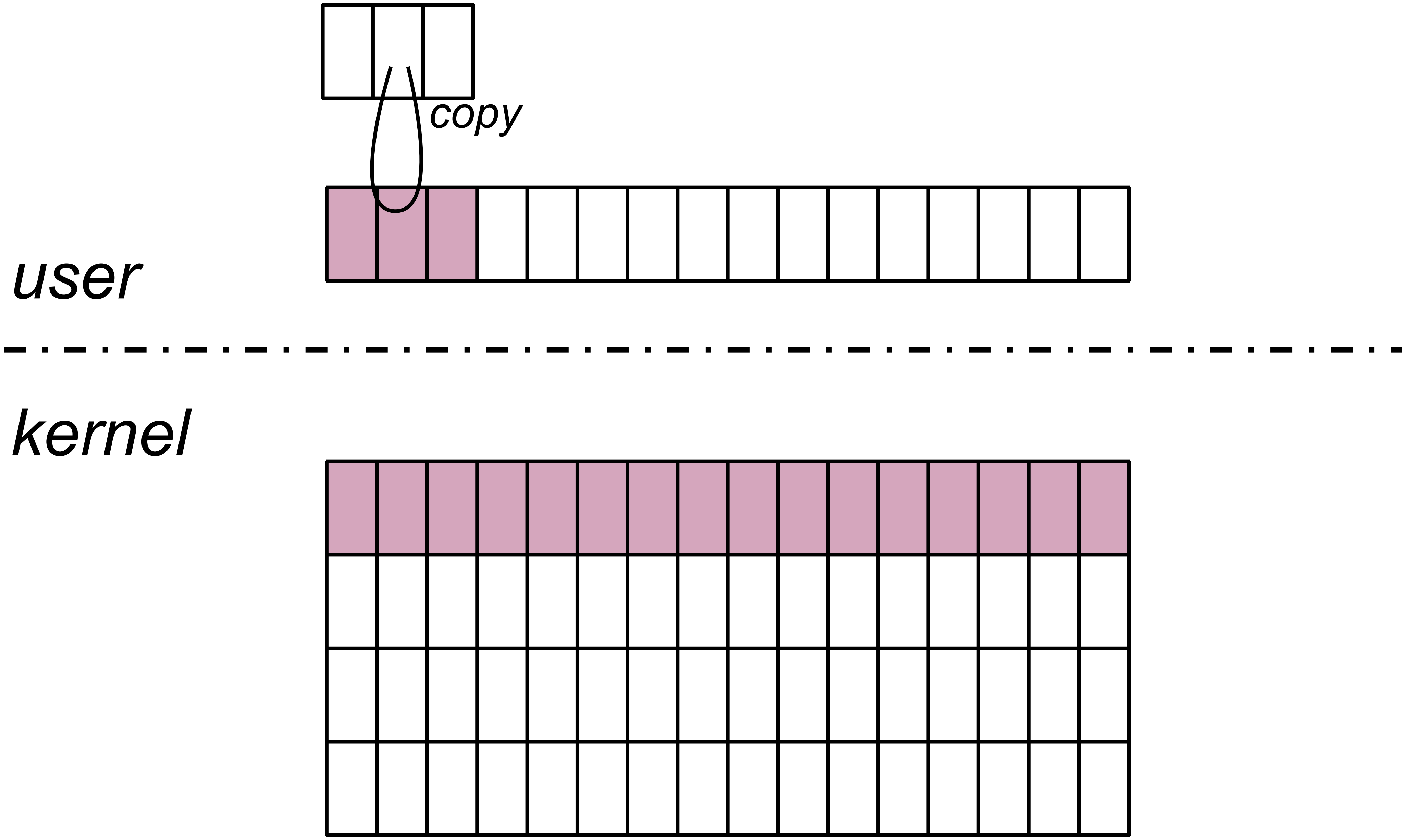


```c
char buf[10], bbuf[80];
int fd, x, y, z;

fd = open("data.txt", O_RDONLY);
read(fd, bbuf, sizeof(bbuf));
```

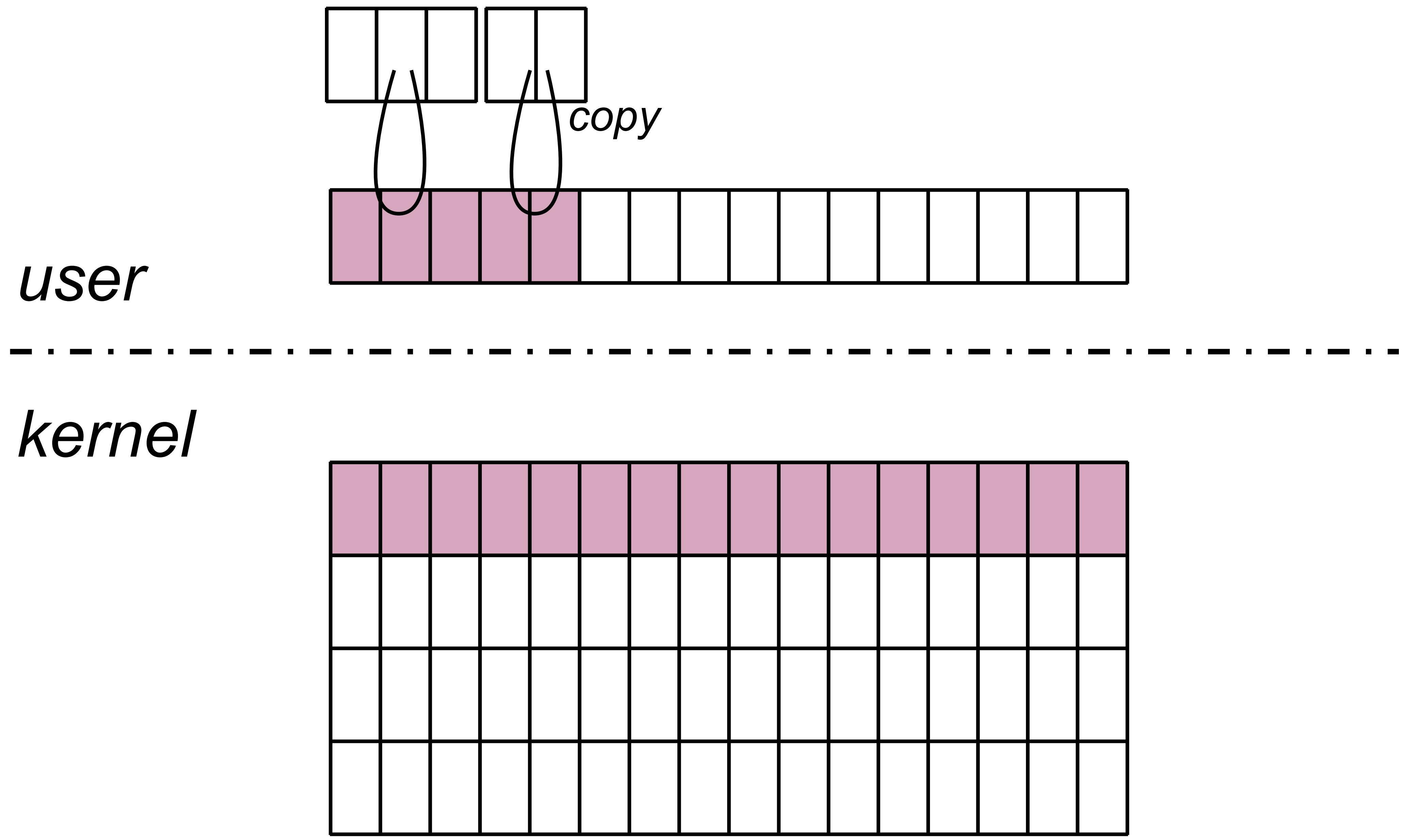
data.txt

```
102030
```

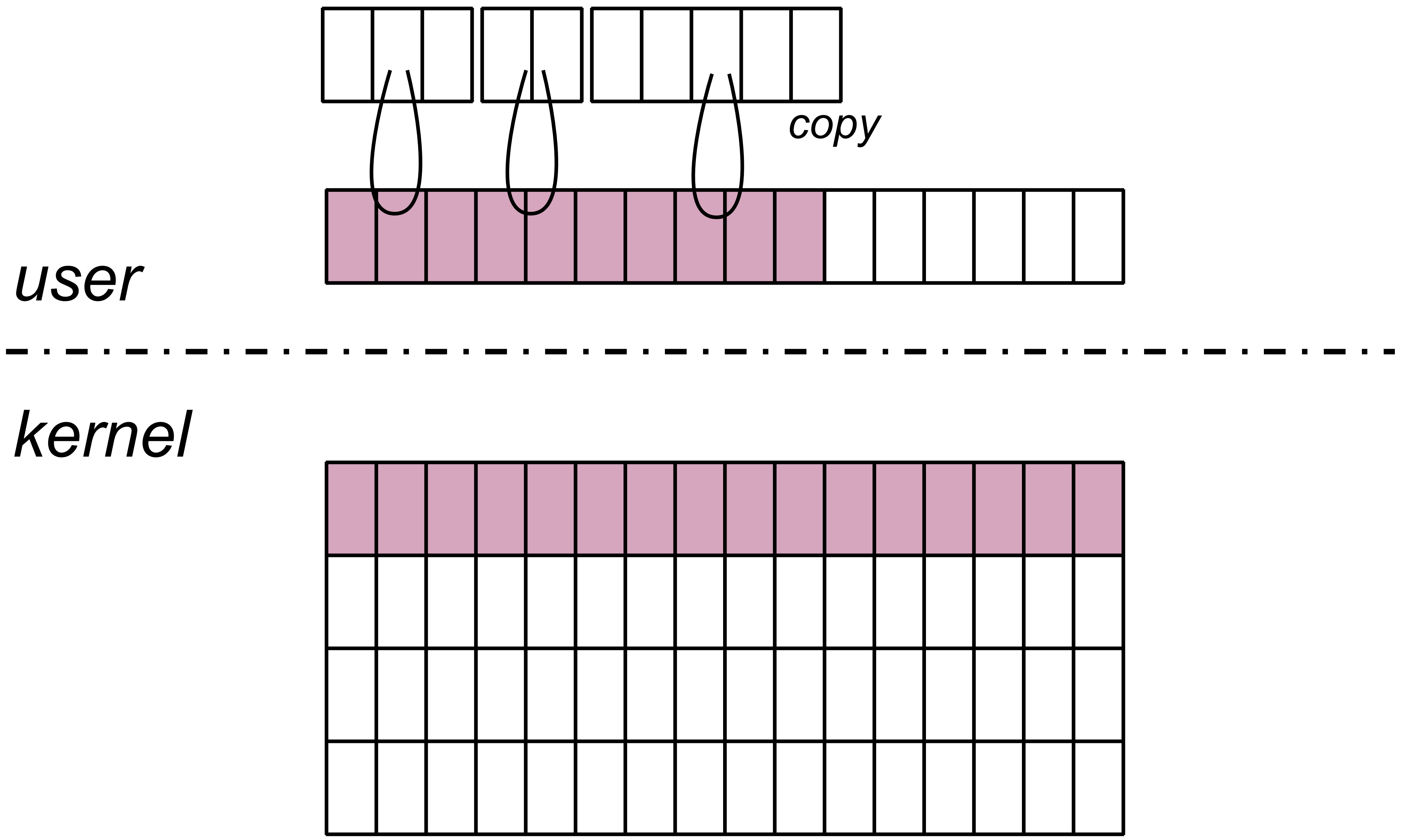
step 2: avoid syscalls and process future "reads" from that buffer



step 2: avoid syscalls and process future "reads" from that buffer



step 2: avoid syscalls and process future "reads" from that buffer



```
char buf[10], bbuf[80];
int fd, x, y, z;

fd = open("data.txt", O_RDONLY);
read(fd, bbuf, sizeof(bbuf));

buf[2] = 0;
memcpy(buf, bbuf, 2);
x = atoi(buf);
```

data.txt

```
102030
```

```c
char buf[10], bbuf[80];
int fd, x, y, z;

fd = open("data.txt", O_RDONLY);
read(fd, bbuf, sizeof(bbuf));

buf[2] = 0;
memcpy(buf, bbuf, 2);
x = atoi(buf);

memcpy(buf, bbuf+2, 2);
y = atoi(buf);
```

data.txt

```
102030
```

```c
char buf[10], bbuf[80];
int fd, x, y, z;

fd = open("data.txt", O_RDONLY);
read(fd, bbuf, sizeof(bbuf));

buf[2] = 0;
memcpy(buf, bbuf, 2);
x = atoi(buf);

memcpy(buf, bbuf+2, 2);
y = atoi(buf);

memcpy(buf, bbuf+4, 2);
z = atoi(buf);
```

data.txt

```
102030
```

```c
fd = open("data.txt", O_RDONLY);
read(fd, bbuf, sizeof(bbuf));

buf[2] = 0;
memcpy(buf, bbuf, 2);
x = atoi(buf);

memcpy(buf, bbuf+2, 2);
y = atoi(buf);

memcpy(buf, bbuf+4, 2);
z = atoi(buf);
```

data.txt

```
102030
```

```
$ strace ./a.out
execve("./a.out", ["./a.out"], [/* 67 vars */]) = 0
...
open("data.txt", O_RDONLY)                 = 3
read(3, "102030\n", 80)                    = 7
write(1, "10 20 30", 8)                    = 8
...
```

to generalize, bundle together:

(1) FD
(2) backing buffer
(3) num unused bytes
(4) pointer to next byte

```c
typedef struct {
    int  fd;          /* (1) wrapped FD             */
    char buf[100];    /* (2) backing buffer         */
    int  count;       /* (3) num unused bytes       */
    char *nextp;      /* (4) pointer to next byte */
} bufio_t;


void bufio_init(bufio_t *bp, int fd) {
    bp->fd    = fd;
    bp->count = 0;
    bp->nextp = bp->buf;
}
```

```
ssize_t bufio_read(bufio_t *bp, char *buf, size_t n) {
    int ncpy;

    /* fill backing buffer if empty */
    if (bp->count <= 0) {
        bp->count = read(bp->fd, bp->buf, sizeof(bp->buf));
        if (bp->count <= 0)
            return bp->count;     /* EOF or read error */
        else
            bp->nextp = bp->buf; /* re-init buf position */
    }

    /* copy from backing buffer to user buffer */
    ncpy = (bp->count < n)? bp->count : n;
    memcpy(buf, bp->nextp, ncpy);
    bp->nextp += ncpy;
    bp->count -= ncpy;

    return ncpy;
}
```

```
char buf[10];
int fd, x, y, z;
bufio_t bbuf;

fd = open("data.txt", O_RDONLY);
bufio_init(&bbuf, fd);

buf[2] = 0;
bufio_read(&bbuf, buf, 2);
x = atoi(buf);

bufio_read(&bbuf, buf, 2);
y = atoi(buf);

bufio_read(&bbuf, buf, 2);
z = atoi(buf);
```

data.txt

```
102030
```

open is now a distraction… we never use the FD directly (except to initialize buffer) next step: hide syscalls from user — wrap `open` together with buffer initialization

```
bufio_t *buf_open(const char *path) {
    bufio_t *buf = malloc(sizeof(bufio_t));
    int fd = open(path, O_RDWR);
    bufio_init(buf, fd);
    return buf;
}


int main() {
    bufio_t *bbuf = buf_open("data.txt");
    char buf[10];
    int x, y, z;

    bufio_read(bbuf, buf, 2);
    ...
}
```

Stop!

`<stdio.h>` does all this for us!

fclose fdopen feof ferror
fflush fgetc fgetln fgetpos
fgets **fopen** **fprintf** fputc fputs
fread freopen **fscanf** fseek
fsetpos **fwrite** getc mktemp
perror **printf** putc putchar puts
remove **rewind** **scanf** sprintf
sscanf strerror tmpfile ungetc
vfprintf vprintf vscanf ...

… all use buffered I/O

stdio functions operate on *stream* objects

i.e., buffered wrappers on FDs

```
FILE* fopen ( const char *filename, const char *mode );
FILE* fdopen   ( int fd, const char *mode );
int   fclose   ( FILE *stream );
int   fseek ( FILE *stream, long offset, int whence );
size_t   fread ( void *ptr, size_t size, size_t nitems, FILE *stream );
size_t   fwrite   ( void *ptr, size_t size, size_t nitems, FILE *stream );

int   fprintf  ( FILE *stream, const char *format, ... );
int   fscanf   ( FILE *stream, const char *format, ... );
char* fgets ( char *str, int size, FILE *stream );
```

```c
int x, y, z;
FILE *infile = fopen("data.txt", "r");

fscanf(infile, "%2d", &x);
fscanf(infile, "%2d", &y);
fscanf(infile, "%2d", &z);

printf("%d %d %d", x, y, z);


fclose(infile); /* or memory leak! */
```

```
$ strace ./a.out
execve("./a.out", ["./a.out"], [/* 67 vars */]) = 0
...
open("data.txt", O_RDONLY)                = 3
read(3, "102030\n", 4096)                 = 7
write(1, "10 20 30", 8)                   = 8
close(3)                                  = 0
...
```

```
printf("h");
printf("e");
printf("l");
printf("l");
printf("o");
```

```
$ strace ./a.out
...
write(1, "hello", 5)                    = 5
...
```

(writes are buffered too!)

stream buffer can *absorb* multiple writes before being flushed to underlying file flush happens on:

- buffer being filled
- (normal) process termination
- newline, in a line-buffered stream
- explicitly, with fflush

```c
int main() {
    printf("h");
    printf("e");
    printf("l");
    printf("l");
    printf("o");
    fork();
}
```

```
$ ./a.out
hellohello
```

@#$%^&*!!!

```c
int n, fd = open("fox.txt", O_RDONLY);
char buf[10];

n = read(fd, buf, sizeof(buf));
write(1, buf, n);
if (fork() == 0) {
    n = read(fd, buf, sizeof(buf));
    write(1, buf, n);
    exit(0);
}
wait(NULL);
n = read(fd, buf, sizeof(buf));
write(1, buf, n);
```

fox.txt

```
the quick brown
fox jumps over
the lazy dog
```

```
$ ./a.out
The quick brown fox jumps over
```

```
int n;
FILE *stream = fopen("fox.txt", "r");
char buf[10];

n = fread(buf, 1, sizeof(buf), stream);
write(1, buf, n);
if (fork() == 0) {
    n = fread(buf, 1, sizeof(buf), stream);
    write(1, buf, n);
    exit(0);
}
wait(NULL);
n = fread(buf, 1, sizeof(buf), stream);
write(1, buf, n);
```

fox.txt

```
the quick brown
fox jumps over
the lazy dog
```

```
$ ./a.out
The quick brown fox brown fox
```

@#$%^&*!!!

things gets even more confusing when we perform *both* input & output

```
int fd = open("fox.txt", O_RDWR);
char buf[10];

/* output followed by input */
write(fd, "a playful ", 10);
read(fd, buf, sizeof(buf));

write(1, buf, sizeof(buf));
```

fox.txt

```
a playful brown
fox jumps over
the lazy dog
```

```
$ ./a.out
brown fox
```

```
int fd = open("fox.txt", O_RDWR);
FILE *stream = fdopen(fd, "r+");
char buf[10];

/* output followed by input */
fwrite("a playful ", 1, 10, stream);
read(fd, buf, sizeof(buf));

write(1, buf, sizeof(buf));
```

fox.txt

```
the quick a
playful jumps
over the lazy
dog
```

```
$ ./a.out
the quick
```

@#$%^&*!!!

```c
int fd = open("fox.txt", O_RDWR);
char buf[10];

/* input followed by output */
read(fd, buf, sizeof(buf));
write(fd, "green cat ", 10);



write(1, buf, sizeof(buf));
```

fox.txt

```
the quick green
cat jumps over
the lazy dog
```

```
$ ./a.out
the quick
```

```c
FILE *stream = fopen("fox.txt", "r+");
char buf[10];

/* input followed by output */
fread(buf, 1, sizeof(buf), stream);
fwrite("green cat ", 1, 10, stream);

write(1, buf, sizeof(buf));
```

fox.txt

```
the quick brown
fox jumps over
the lazy dog
green cat
```

```
$ ./a.out
the quick
```

@#$%^&*!!!

When a file is opened with update mode ..., both input and output may be performed on the associated stream. However, output shall not be directly followed by input without an intervening call to the **`fflush`** function or to a file positioning function ..., and input shall not be directly followed by output without an intervening call to a file positioning function, unless the input operation encounters end- of-file.

ISO C99 standard, 7.19.5.3 (par 6)

input shall not be directly followed by output without an intervening call to a file positioning function

but not all files support "file positioning functions"! (e.g., no seeks on *character devices*) lessons:

- buffered stdio functions help minimize system overhead & simplify I/O
- use whenever possible!

lessons:

- but need to beware of glitches
- don’t mix buffered & unbuffered I/O
- and not appropriate for some devices (e.g., network)
  - use low-level, robust I/O for these

TO DO:

Read CH 6 CS:APP

How is Lab 01 going? (Due Sunday!)

Lab 02 out tomorrow! (Due Oct 29)