Open

```
int open(const char *pathname, int flags);
int open(const char *pathname, int flags,
                               mode_t mode);
```

Starts a open session on the file given in pathname.
(absolute or relative)

Returns a handle, called file descriptor(fd), to the open session of the file.

Use two arguments to open an existing file.

Use three arguments to create & open a file.

Second Argument – Flags

Specifies the flags for the open session effective in current session only. (some times refered to as file status flags).

Should have one of:

O_RDONLY - Input operations only.

O_WRONLY - Output operations only.

O_RDWR - Input and output operations.

The above flags require appropriate access permissions on the file.

Optional flags :

O_APPEND - All output operations at the current end of the file.

O_TRUNC - Delete file contents on open and set file size to zero.
O_RDWR or O_WRONLY is required.

O_CREAT - Create the file if it does not exist.
Requires the third argument mode.

O_EXCL - Can be used along with O_CREAT .
If the file already exists, open fails.

Logic Option

Opening an existing file

```
int fd1 = open("myfile1",          O_RDONLY);          Input only
int fd2 = open("./dir1/myfile3",   O_RDWR);            Input/Output
int fd3 = open("/home/srini/file1",
                                   O _WRONLY | O_APPEND);
                                   All Output at the current end of the file
int fd4 = open("myfile1",      O_WRONLY  |  O_TRUNC);
                                        Output, existing contents deleted
int fd5 = open("/dev/tty3",        O_RDONLY);          Input only
                                   Device special files can also be opened
int fd6 = open("softlink-to-f1",O_WRONLY);             Output only
                  Opens file "f1" to which the soft link softlink-to-f1 points
```

Two independent open sessions on "myfile1" in current process with differing flags :

fd1 - O_RDONLY

fd4 - O_WRONLY | O_APPEND

Logic Option

Creating Files

int fd = open ("newfile",O_CREAT | O_RDWR,
S_IRWXU);

If “newfile” doesn't exist, it will be created.

If “newfile” already exists, it will be opened for Input/Output with its contents retained.

Creating files Exclusively

Int fd = open ("newfile",

O_CREAT | O_EXCL | O_RDWR,

S_IRWXU);

If “newfile” doesn't exist, it will be created.

If “newfile” already exists, open will fail with errno set to EEXIST.

Exclusively means – Make sure the file created afresh.

Fail if the file already exists.

This makes sure the contents of the existing file are not modified in this open session.

Logic Option

Third Argument – Mode

int fd = open ("newfile", O_CREAT | O_RDWR,
S_IRWXU | S_IRGRP | S_IWGRP | S_IROTH);

Third argument specifies the requested access permissions on the file to be created.

S_IRWXU - Owner read, write and execute permissions.

S_IRUSR - Owner read permission.

S_IWUSR - Owner write permission.

S_IXUSR - Owner execute permission.

Similarly, S_IRWXG S_IRGRP S_IWGRP S_IXGRP
S_IRWXO S_IROTH S_IWOTH S_IXOTH

are defined for the group and others.

Logic Option

Third Argument – Mode Continued

S_ISUID - set user ID on execution

S_ISGID - set group ID on execution

S_ISVTX - sticky bit

Permissions of the created file.

int fd = open ("newfile", O_CREAT | O_RDONLY,
S_IRWXU | S_IRGRP | S_IWGRP | S_IRWXO);

Assume current umask = 033 = **000 011 011**

Requested Permissions are **rwx rw- rwx**

File permissions are **rwx r-- r--**

But, flags for the current open session has O_RDONLY.

Only input operation allowed on file descriptor fd, even though the created file will have rwx access for the owner.

Permissions of the created file.

int fd = open ("newfile", O_CREAT | O_RDWR,
S_IRWXU | S_IRGRP | S_IWGRP | S_IRWXO);

Assume current umask = 033 =	**000**	**011**	**011**
Requested Permissions are	**rwx**	**rw-**	**rwx**
	111	**110**	**111**
~umask =	**111**	**100**	**100**
Requested Permissions & ~umask=	**111**	**100**	**100**
File permissions are	**rwx**	**r--**	**r--**

Logic Option

Creat system call

int creat(const char *pathname, mode_t mode);

creat() is equivalent of 3 argument open() with default flags
O_CREAT|O_WRONLY|O_TRUNC

creat("newfile1", S_IRWXU | S_IROTH);
is same as
open("newfile", O_CREAT | O_WRONLY | O_TRUNC,
S_IRWXU | S_IROTH):

File Table

- Kernel maintains one global File table.
- When any process opens a file, one entry in this table is allocated.
- Important info contained in the entries includes

  Flags - Open session flags.

  (O_CREAT and O_EXCL will not be included.)

  Offset - Position in the file at which next Input/Output will commence. Offset of the first byte is 0.

O_RDWR | O_APPEND | O_NONBLOCK , 346

Logic Option

File Table

Process P1:

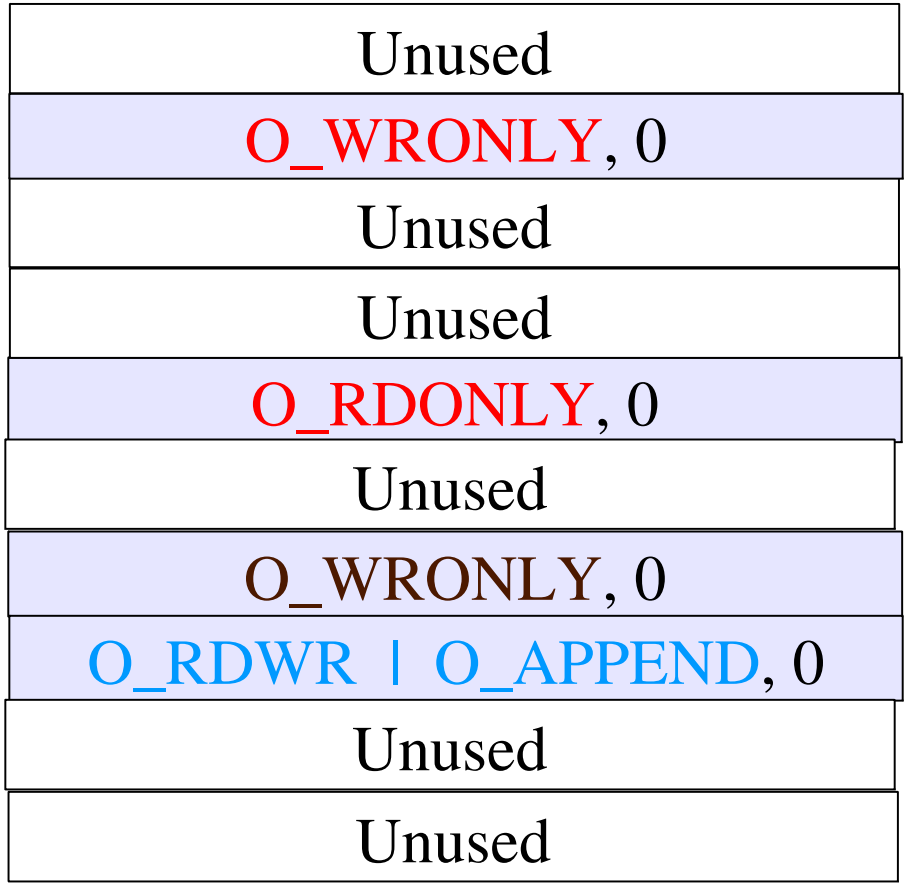
```
int fd = open("f1",
   O_RDWR |O_APPEND);
```

Process P2:

```
int fd1 = open("f2",
   O_RDONLY);

int fd2 = open("f3",
   O_WRONLY);
```

Process P3:

```
int fd = open("f4",
   O_WRONLY |
   O_TRUNC);
```



Logic Option

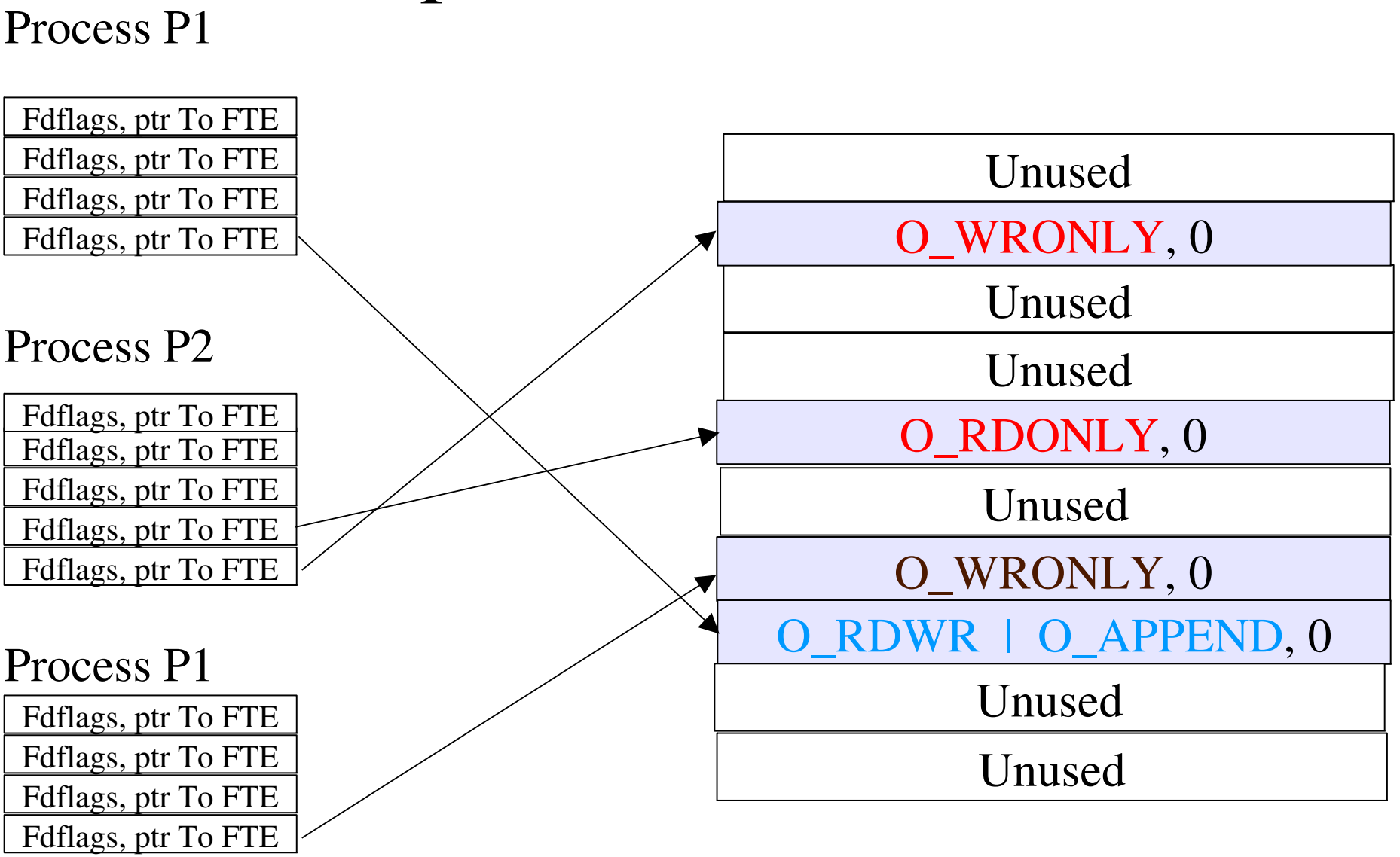
Per Process File Descriptor Table

- Kernel maintains a File Descriptor Table for each of the process on the system.
- Every succesful open() has one FDT entry allocated.
- The file descriptor returned by open() - an index into FDT.
- Each entry has fd flags and a pointer to File Table Entry.
- Indecies 0, 1 and 2 reserved for stdin, stdout and stderr.
- FDT for process P2 in the previous slide:

0	FD Flags, Pointer to FTE of stdin device.
1	FD Flags, Pointer to FTE of stdout device.
2	FD Flags, Pointer to FTE of stderr device.
3	FD Flags, Pointer to FTE of file “f2”
4	FD Flags, Pointer to FTE of file “f3”

More of FD Flags after exec() system call.

File Descriptor Tables & File Table



Closing an open session

int close(int fd);

- The open session of the file is terminated.
- The file descriptor is no more a handle to open file session.
- The entry in FDT indexed by fd is free for reuse.
- The corresponding File Table may also be released.

  More on this after dup() and fork() syscalls.

- The existence of the file will not be effected by close().

  Exception to this in combination with unlink() system call.

  Calling unlink() first and then close() could delete file.

  More on this after unlink() system call.

Closing Continued..

Process P2:

int fd1 = open("f2", O_RDONLY);

int fd2 = open("f3", O_WRONLY);

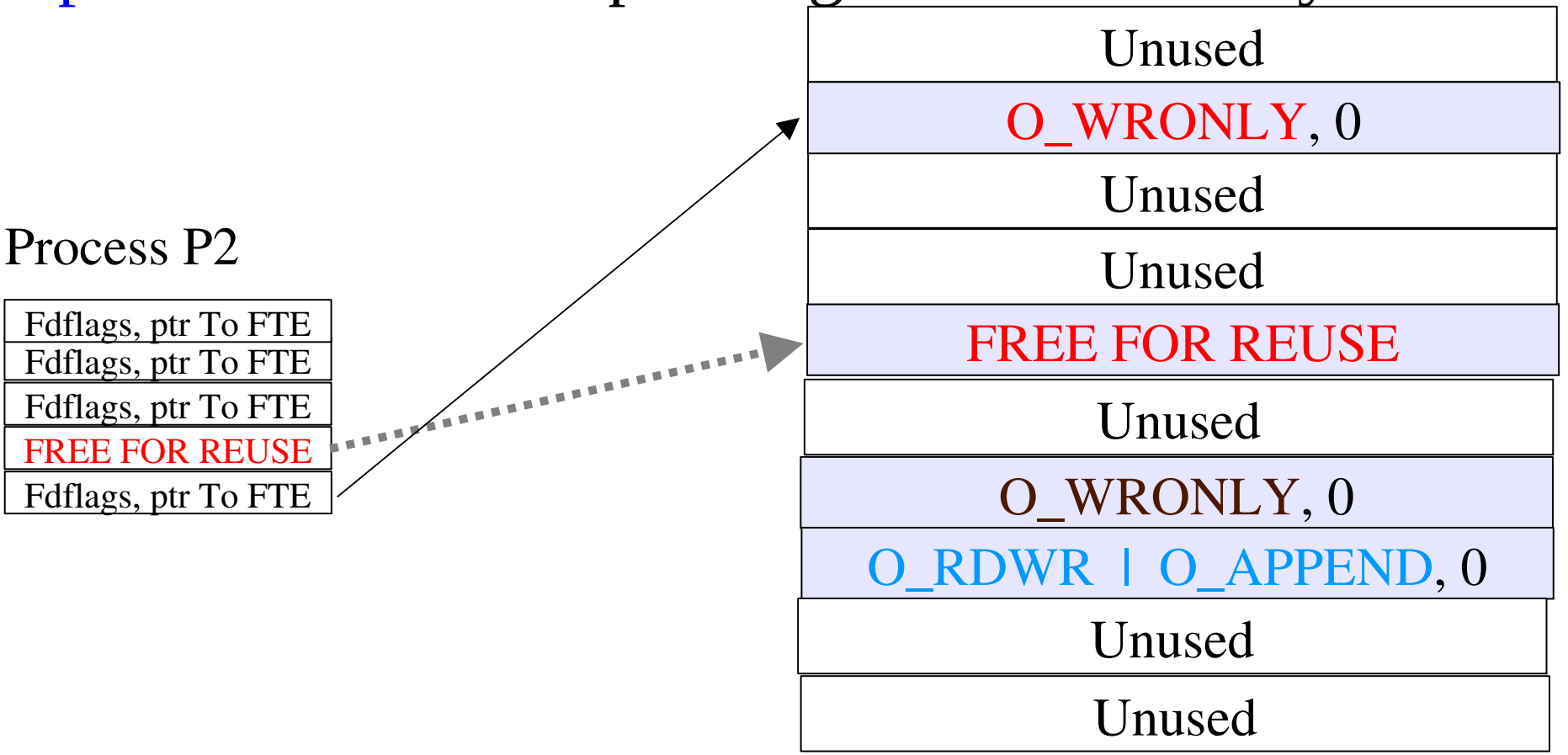
```
/* Some code .....
   ..............*/
close(fd1);
```

FDT of Process P2.

0	FD Flags, Pointer to FTE of stdin device.
1	FD Flags, Pointer to FTE of stdout device.
2	FD Flags, Pointer to FTE of stderr device.
3	FREE FOR REUSE
4	FD Flags, Pointer to FTE of file "f3"

Effect of close on File Table

- File Table entry is also freed.
- Since FDT entry index by fd is freed, it no longer points to the corresponding File Table Entry.



File Descriptor Reuse

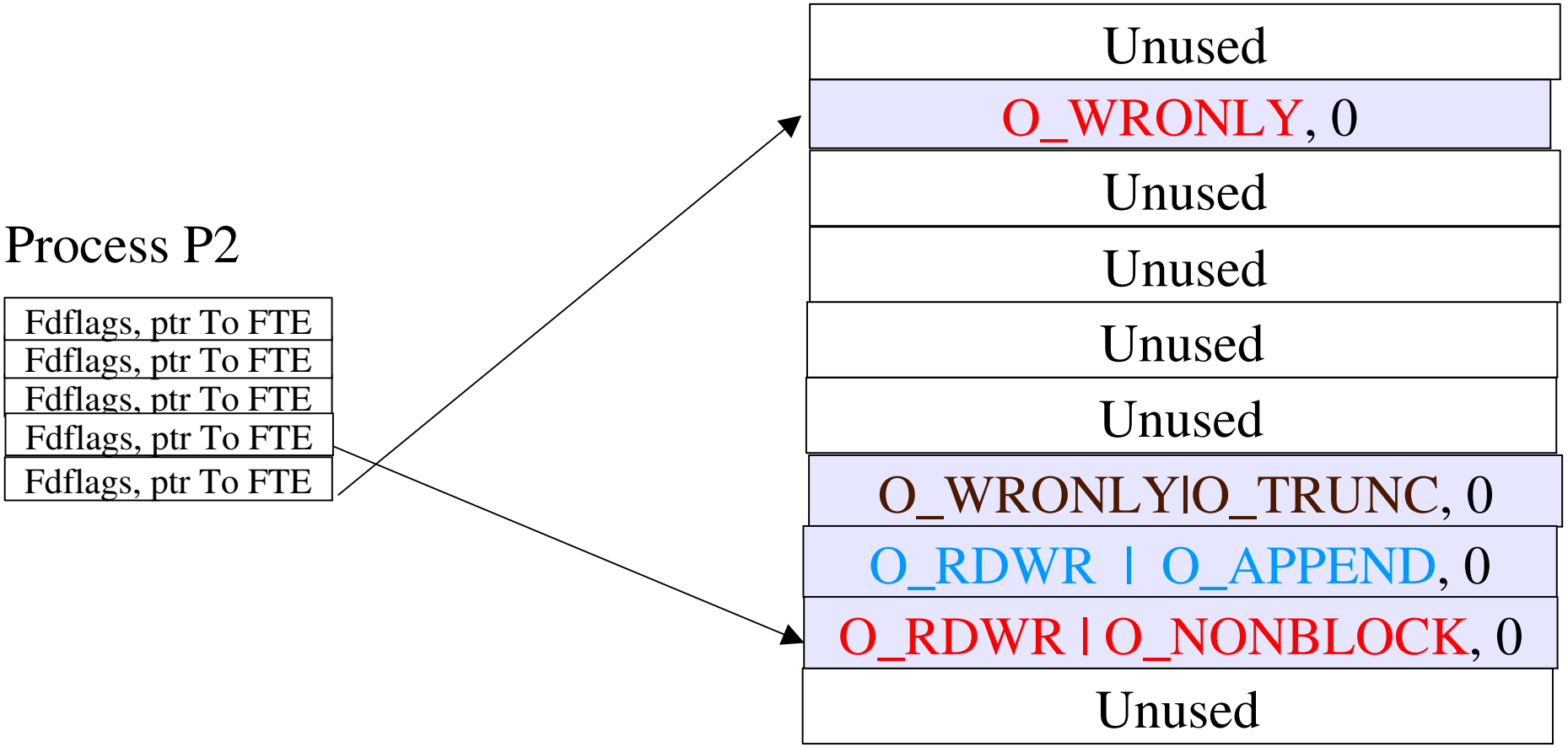
Process P2:

```
int fd1 = open("f2", O_RDONLY);
int fd2 = open("f3", O_WRONLY);
/* Some code .....*/
close(fd1);
int fd3 = open("f4", O_RDWR | O_NONBLOCK);
/* open() will return the least unused descriptor in FDT. */
```

FDT of Process P2.

0	FD Flags, Pointer to FTE of stdin device.
1	FD Flags, Pointer to FTE of stdout device.
2	FD Flags, Pointer to FTE of stderr device.
3	FD Flags, Pointer to FTE of file "f4"
4	FD Flags, Pointer to FTE of file "f3"

File Descriptor Reuse and File Table



Input From Files – Request

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

- size_t - unsigned integral, ssize_t - signed integral.
- Requests the kernel to perform an input operation.
- From the file identified by open session handle fd.
- Requests to read count number of bytes.
- Data will be read into the memory location given by the pointer **buf**.
- Starting at the position in the file given by current offset of the open session.
- Atleast count number of bytes of space should be allocated to buf before calling read().

Input From Files – Result

Ssize_t read(int fd, void *buf, size_t count);

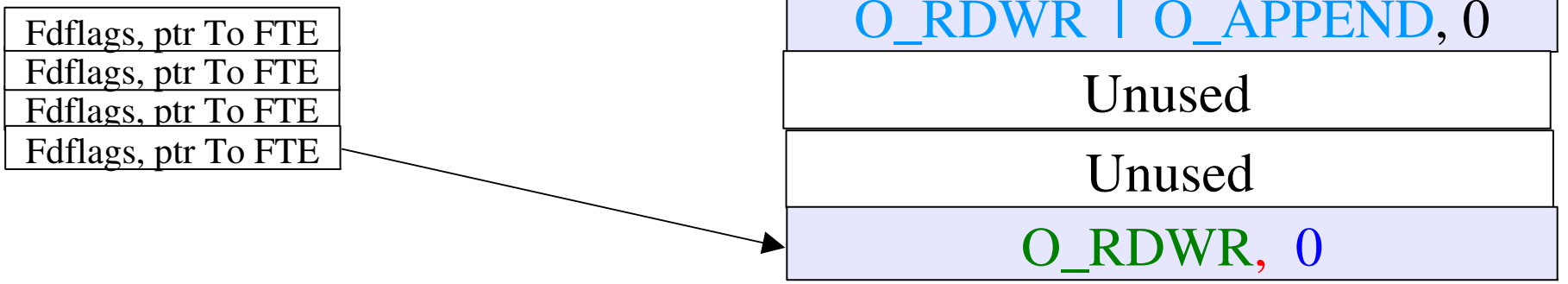
- With ssize_t being signed integral, return value can be:
  - > 0 -- Gives the number of bytes actually read into buf. Can be less than count, the requested number of bytes to read.
  - = 0 -- Reached the end of file. No data to read further. Offset equals the size of the file which indicates one position beyond the current end of file
  - < 0 -- Indicates error in reading and errno will be set.
- Offset of the open session will be advanvced by number of bytes actually read.

Input Operations and FT

int fd = open("f1", O_RDWR);

/* On open offset is zero */

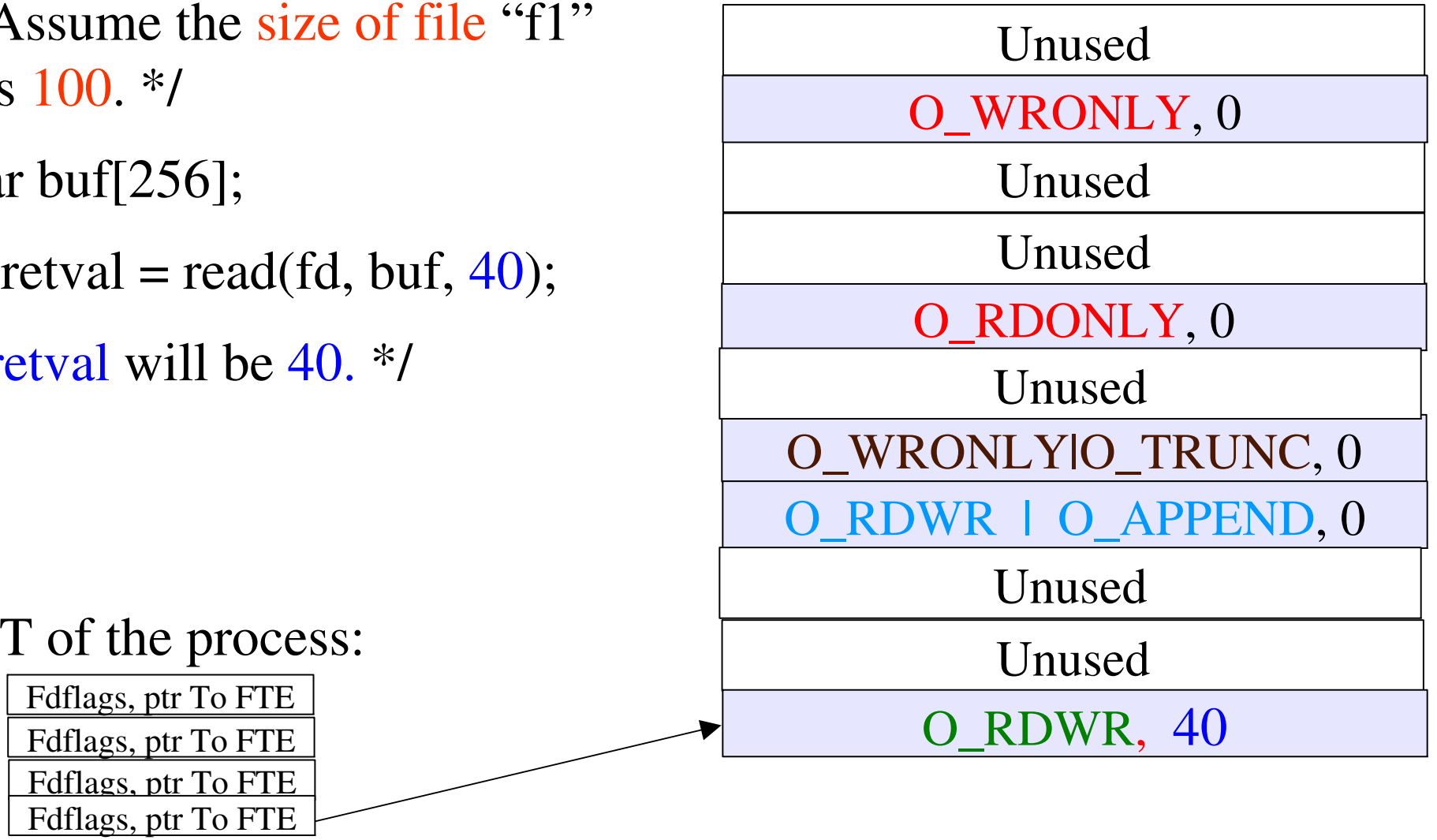
FDT of the process :



Logic Option

Input Operations and FT

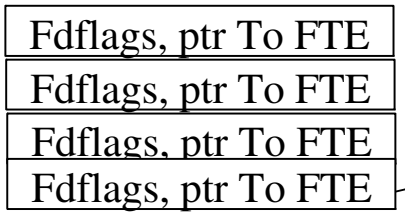
```
int fd = open("f1",
    O_RDWR);

/* Assume the size of file "f1"
    is 100. */

char buf[256];

int retval = read(fd, buf, 40);

/* retval will be 40. */
```



Input Operations and FT

```
int fd = open("f1",
    O_RDWR);

char buf[256];

int retval = read(fd, buf, 40);

int retval = read(fd, buf, 50);

/* retval will be 50. */
```

FDT of the process :



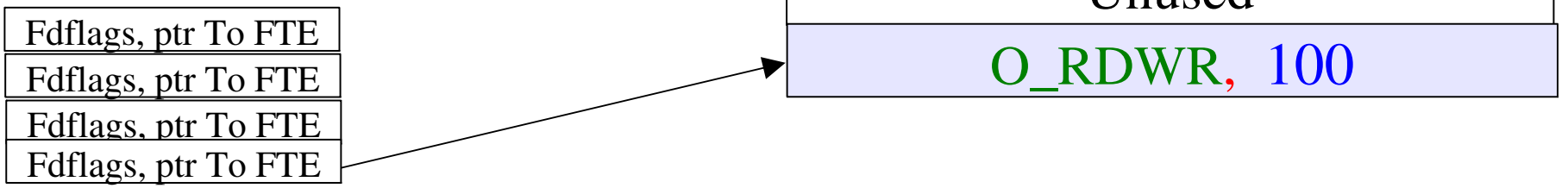
Unused
O_WRONLY, 0
Unused
Unused
O_RDONLY, 0
Unused
O_WRONLY|O_TRUNC, 0
O_RDWR | O_APPEND, 0
Unused
Unused
O_RDWR, 90

Logic Option

Input Operations and FT

```
int fd = open("f1",
    O_RDWR);

char buf[256];

int retval = read(fd, buf, 40);

int retval = read(fd, buf, 50);

int retval = read(fd, buf, 50);

/* retval will be 10. */
```

FDT of the process :



Logic Option

Output To Files – Request

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

- size_t - unsigned integral, ssize_t - signed integral.
- Requests the kernel to perform an output operation.
- To the file identified by open session handle fd.
- Requests to write count number of bytes.
- Data to be written is available in the memory location given by the pointer buf.
- Starting at the position in the file given by current offset of the open session.

Output To Files – Result

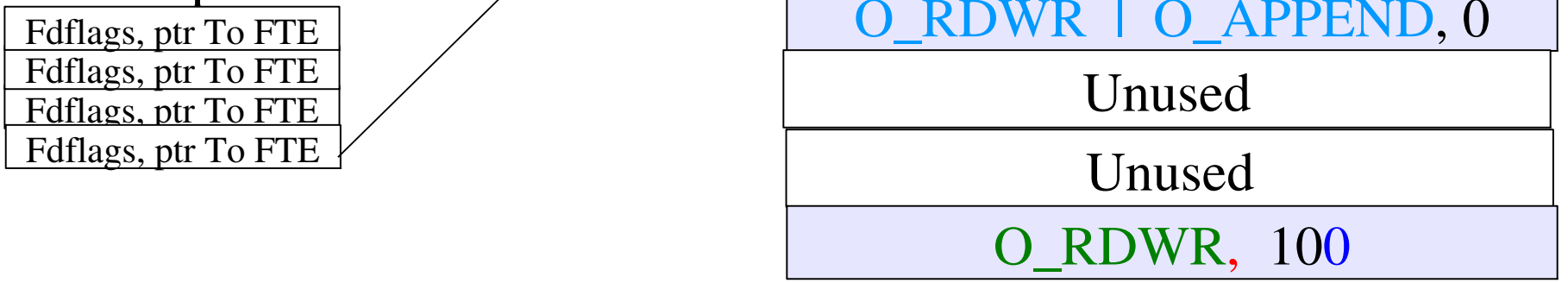
ssize_t write(int fd, const void *buf, size_t count);

- With ssize_t being signed integral, return value can be:
  - > 0 -- Gives the number of bytes actually written to file. Can be less than count, the requested number of bytes to write.
  - < 0 -- Indicates error in wrtiing and errno will be set.
  - = 0 -- This will happen only on regulur files with count equal to zero. Since write() is supposed to insert data to the file, it writes past the current end of file, if needed, increasing the file size.
- Offset of the open session will be advanvced by number of bytes actually written to the file. .

Output Operation and FT

```
int fd = open("f1",
   O_WRONLY);

/* On open offset is zero

 and assume the file  exists
   with size 100.  */
```

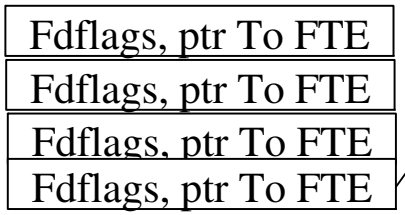
FDT of the process:



Output Operation and FT

```
int fd = open("f1",
    O_WRONLY);

char buf[256];

int retval = write(fd, buf, 40);

/* retval will be 40.

Over writes first 40 bytes.  */
```

FDT of the process :



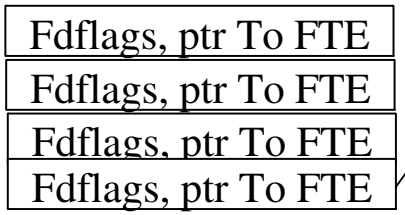
Unused
O_WRONLY, 0
Unused
O_WRONLY, 40
O_RDONLY, 0
Unused
O_WRONLY|O_TRUNC, 0
O_RDWR | O_APPEND, 0
Unused
Unused
O_RDWR, 100

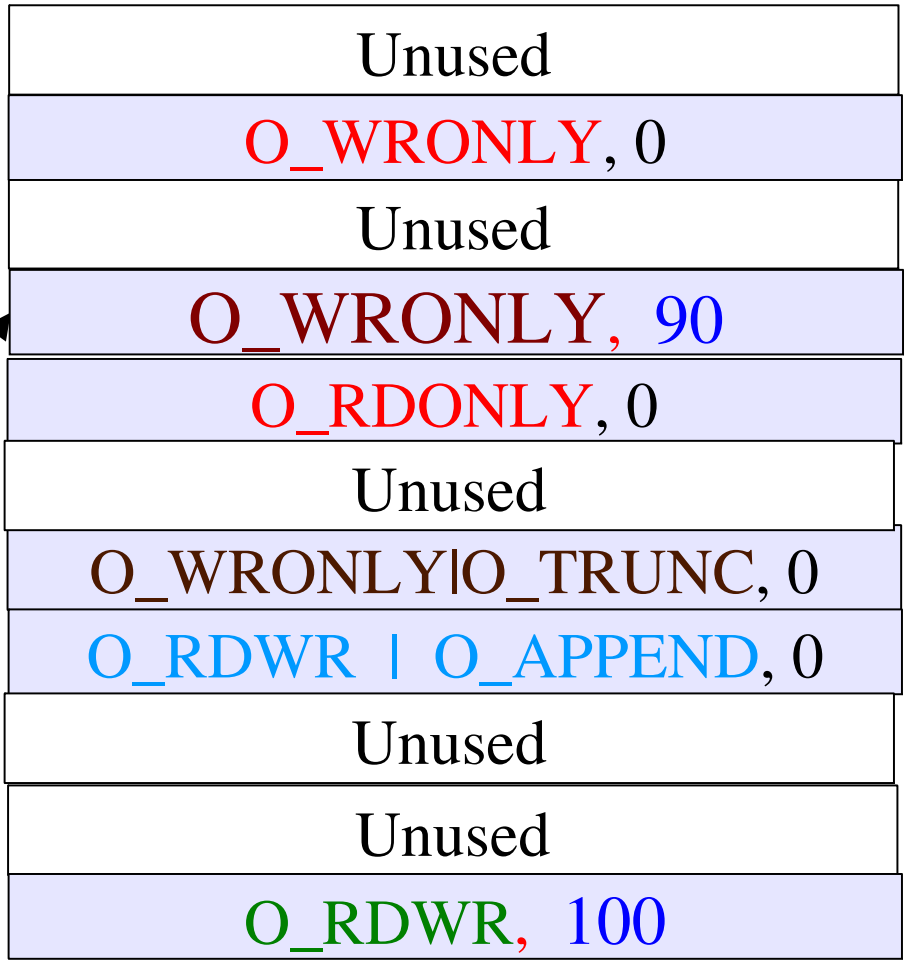
Logic Option

Output Operation and FT

```
int fd = open("f1",
    O_WRONLY);

char buf[256];

int retval = write(fd, buf, 40);

int retval = write(fd, buf, 50);

/* retval will be 50.

Over writes next 50 bytes. */
```

FDT of the process :



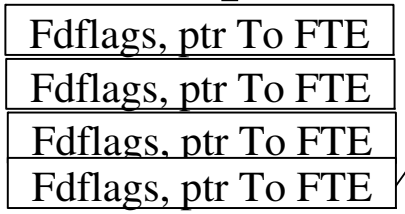


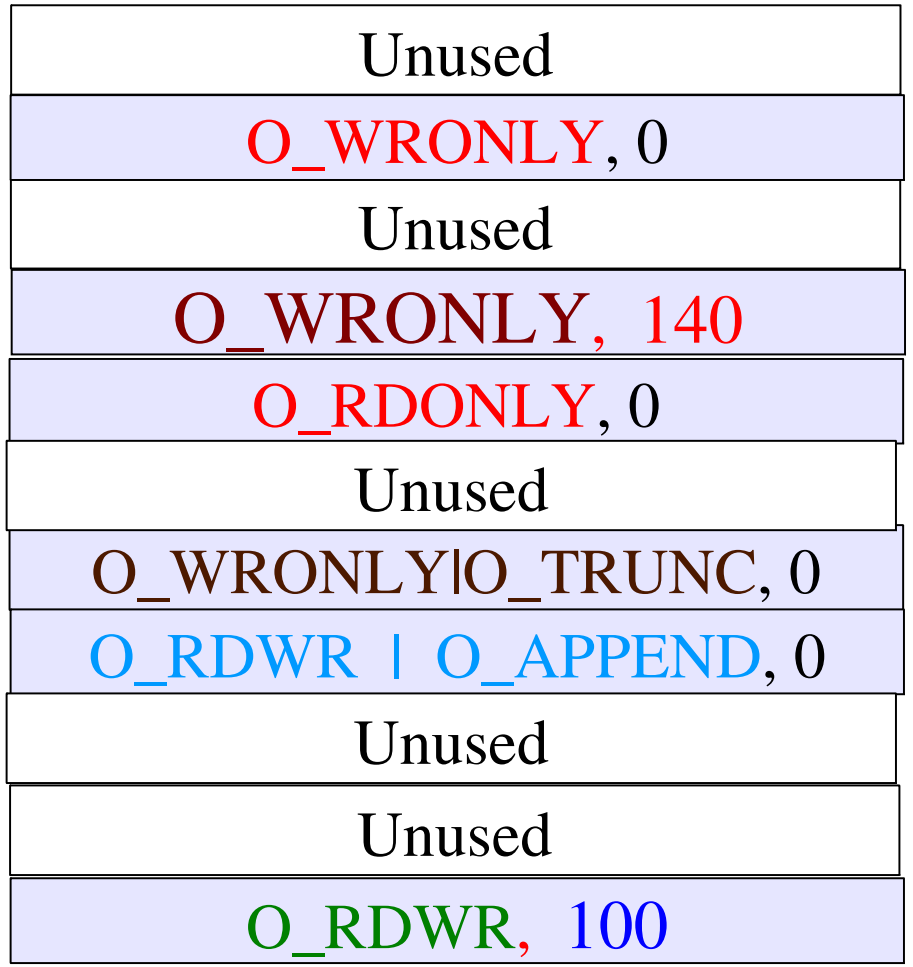
Logic Option

Output Operation and FT

```
int fd = open("f1",
    O_WRONLY);

char buf[256];

int retval = write(fd, buf, 40);

int retval = write(fd, buf, 50);

int retval = write(fd, buf, 50);

/* retval will be 50.

Over writes last 10 bytes of
    file and appends 40 bytes. */
```

FDT of the process:

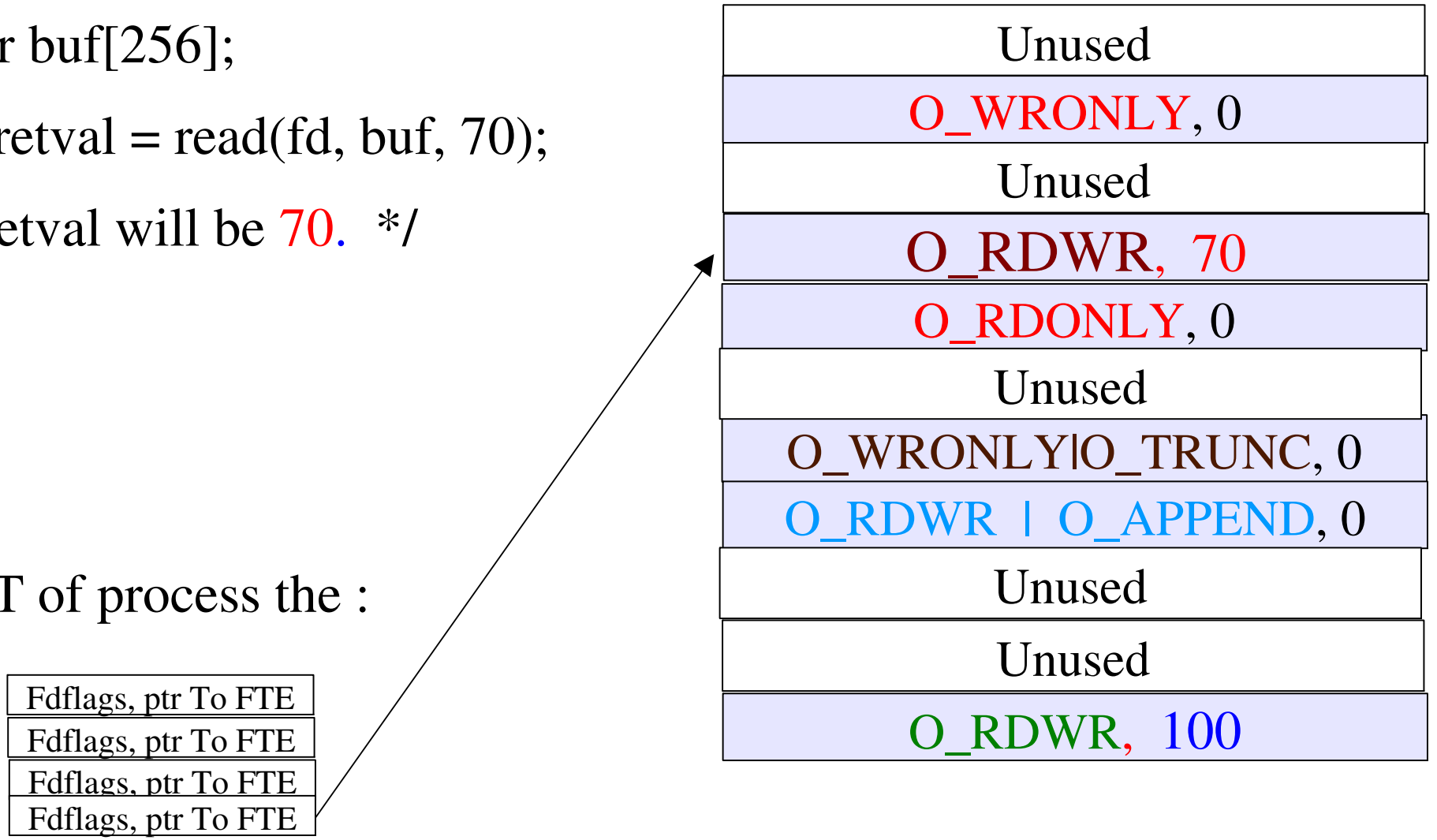




Logic Option

Input/Output Operation and FT

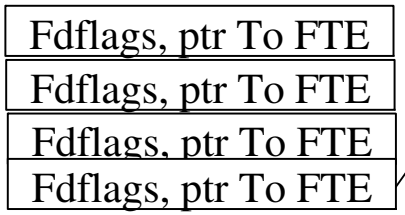
```
int fd = open("f1", O_RDWR);
/* assume file size 100 */
char buf[256];
int retval = read(fd, buf, 70);
/* retval will be 70.  */
```

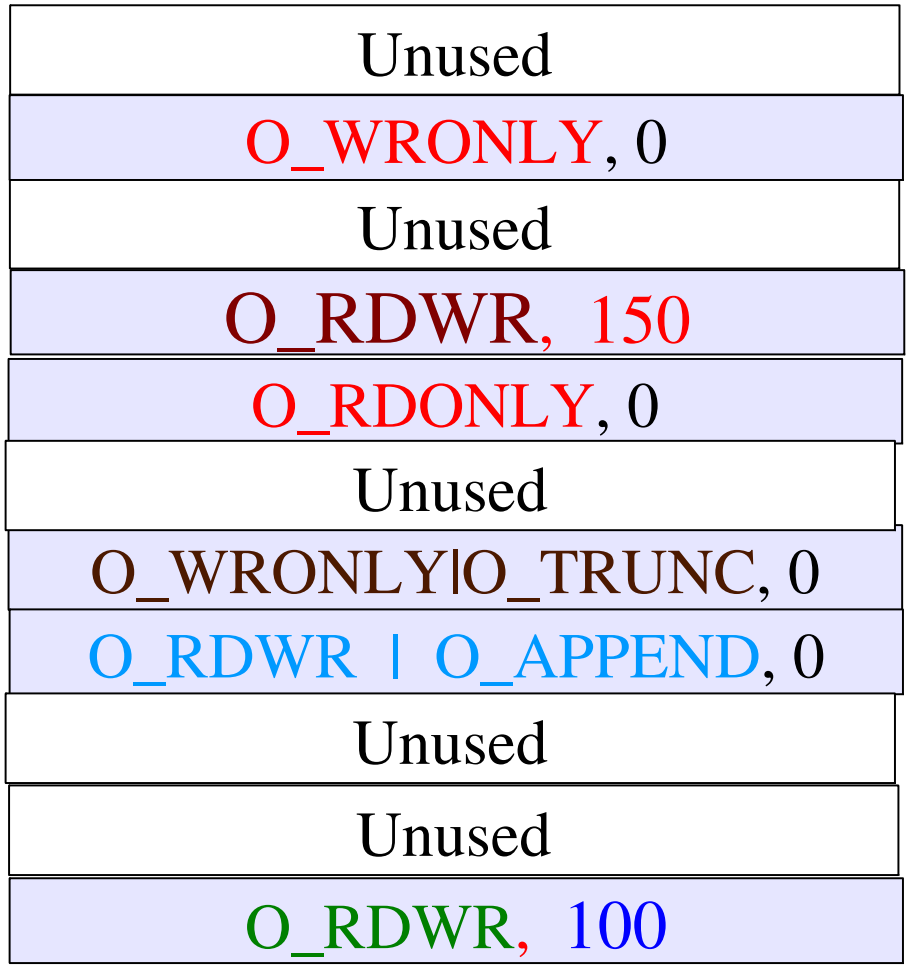


Input/Output Operation and FT

```
int fd = open("f1", O_RDWR);

/* assume file size 100 */

char buf[256];

int retval = read(fd, buf, 70);

/* retval will be 70.  */

retval = write(fd, buf, 80);

/* overwrites last 30 bytes and
   appends 50 more bytes.*/
```

FDT of process:





Logic Option

Duplicating file descriptors

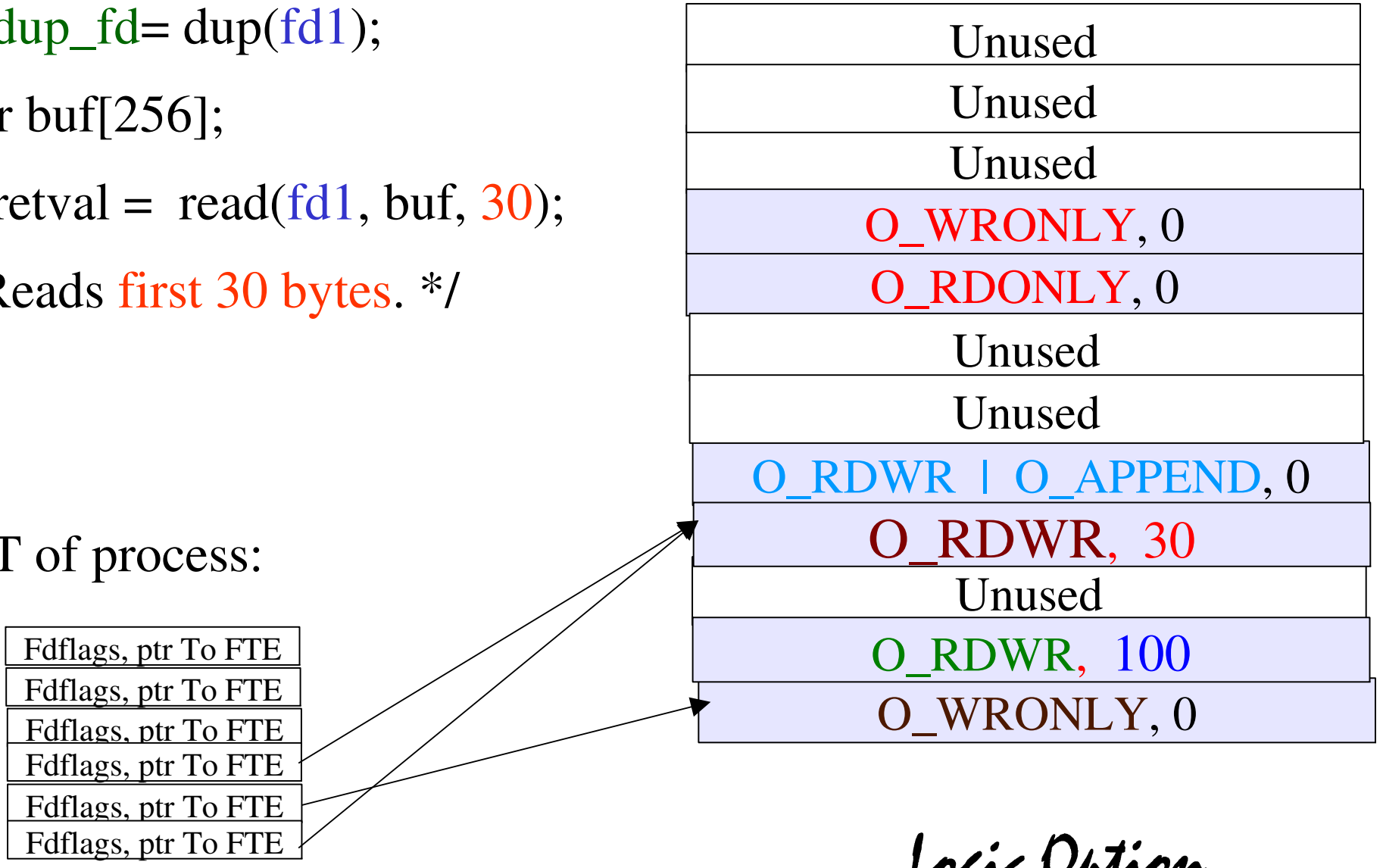
int dup(int oldfd);

- Will create a duplicate descriptor for file open session identified by the file descriptor oldfd.
- The least unused descriptor is made a duplicate of oldfd.
- Returns the duplicate descriptor.
- FDT entries indexed by the original and duplicate descriptors both point to the same file table entry.

  Open session flags and offset are shared by both the descriptors.
- As a result Input/Output operations on either descriptor update the same offset in the file.

Duplicate descriptors and FT

```
int fd1 = open("f1", O_RDWR);
int fd2 = open("f2", O_WRONLY);
int dup_fd= dup(fd1);
char buf[256];
int retval =  read(fd1, buf, 30);
/* Reads first 30 bytes. */
```

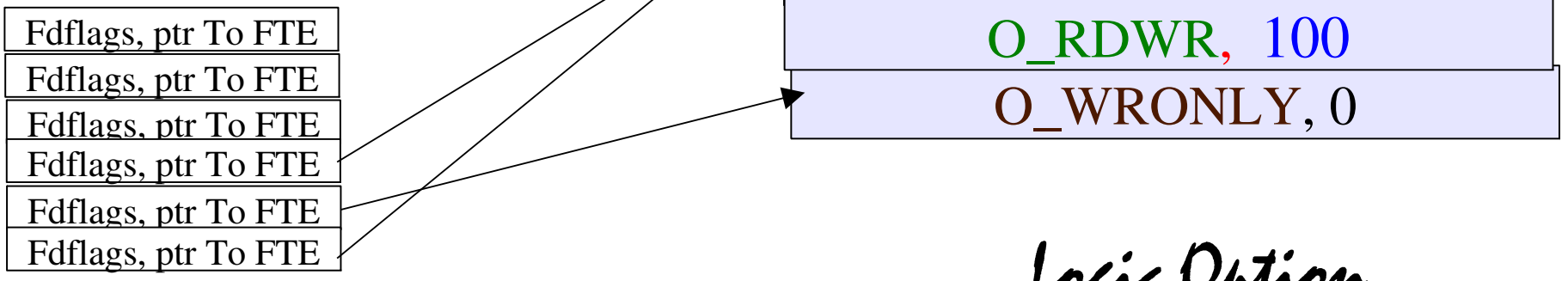


Logic Option

Duplicate descriptors and FT

```
int fd1 = open("f1", O_RDWR);

int fd2 = open("f2", O_WRONLY);

int dup_fd= dup(fd1);

char buf[256];

int retval =  read(fd1, buf, 30);

retval     =  write(dup_fd, buf, 40);

/* writes next 40 bytes */
```

FDT of process:

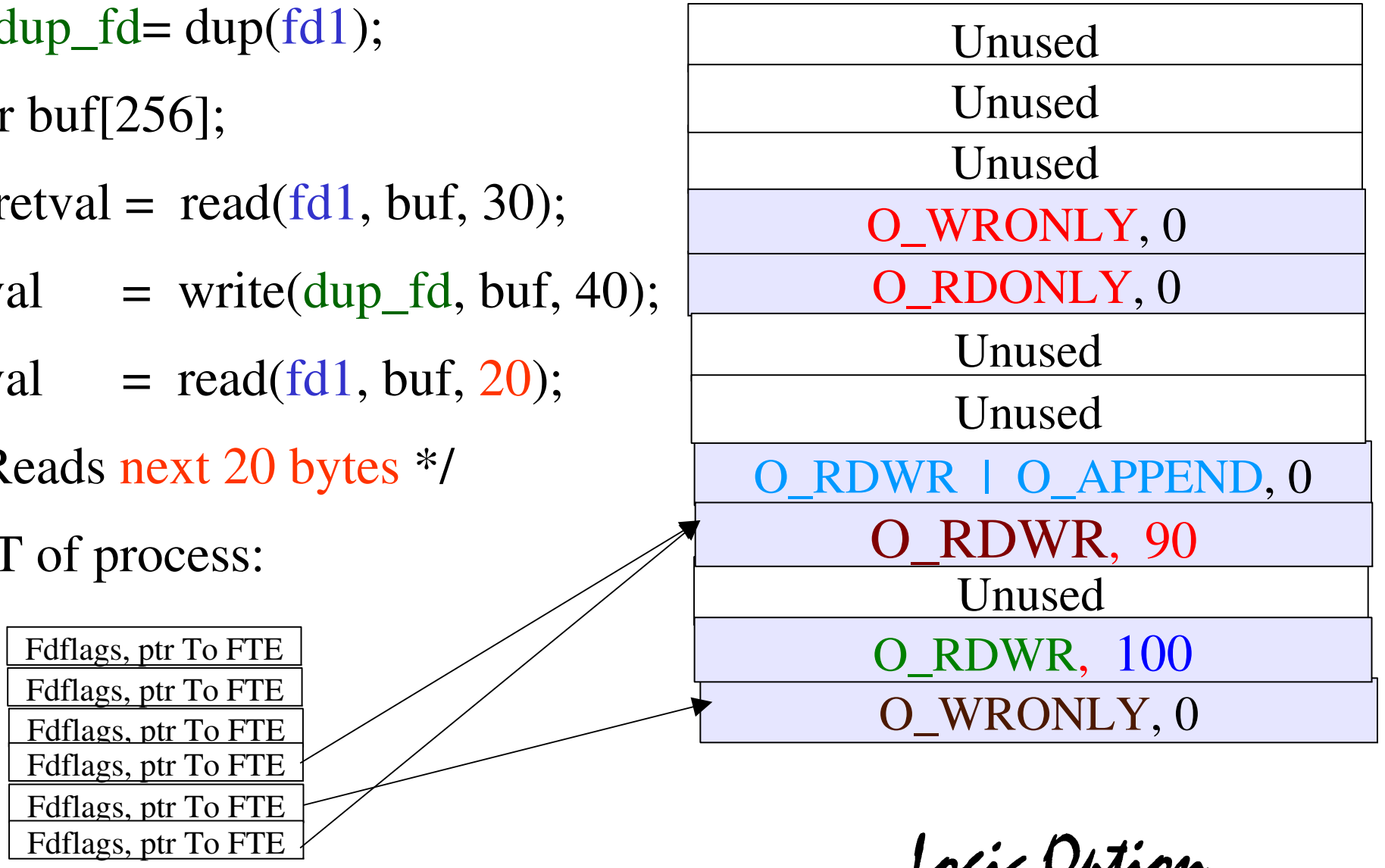


Logic Option

Duplicate descriptors and FT

```
int fd1 = open("f1", O_RDWR);
int fd2 = open("f2", O_WRONLY);
int dup_fd= dup(fd1);
char buf[256];
int retval =  read(fd1, buf, 30);
retval     =  write(dup_fd, buf, 40);
retval     =  read(fd1, buf, 20);
/* Reads next 20 bytes */
```

FDT of process:

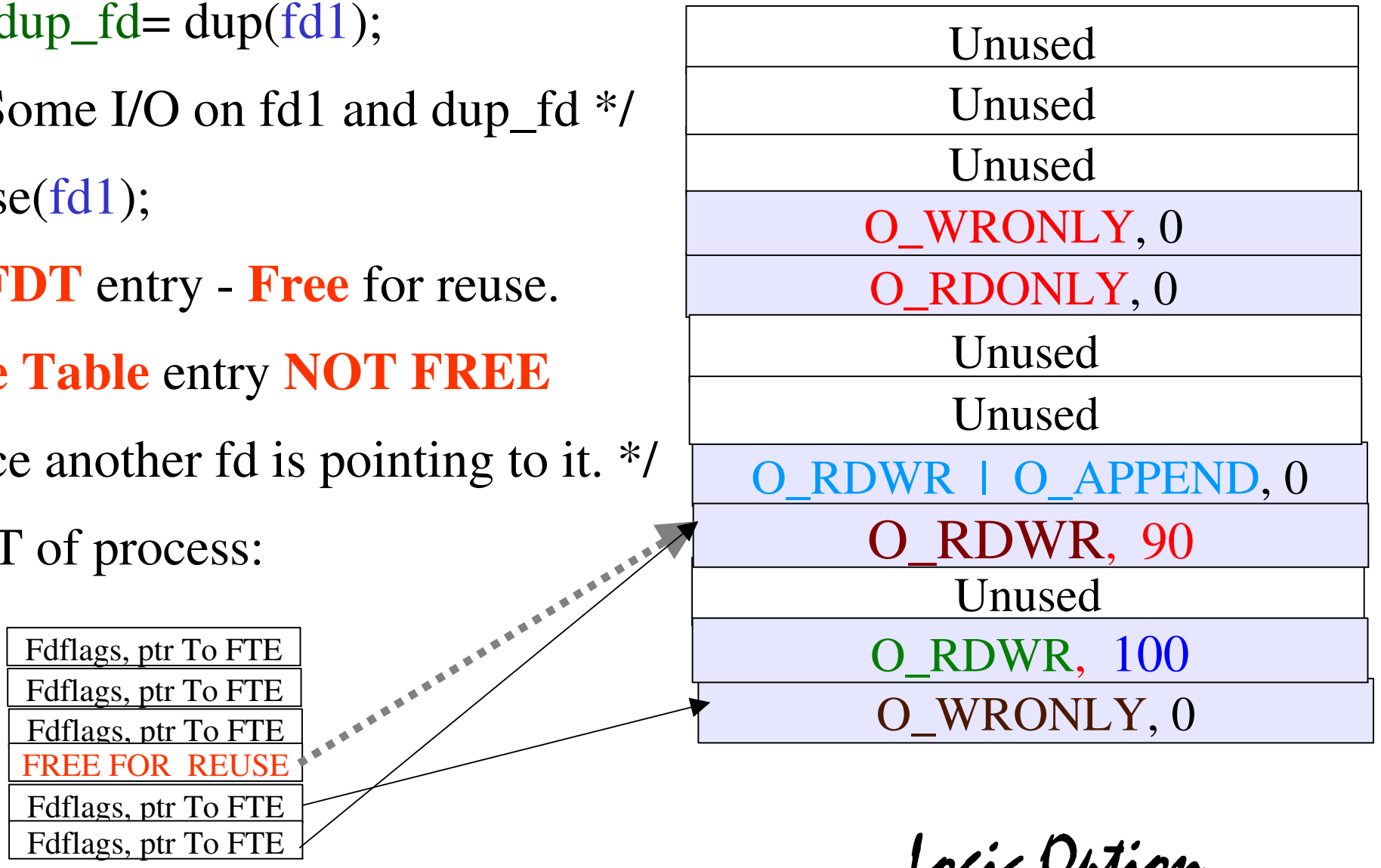


Logic Option

Closing duplicate descriptor

```
int fd1 = open("f1", O_RDWR);

int fd2 = open("f2", O_WRONLY);

int dup_fd= dup(fd1);

/* Some I/O on fd1 and dup_fd */

close(fd1);

/* FDT entry - Free for reuse.

File Table entry NOT FREE

since another fd is pointing to it. */
```

FDT of process:



Logic Option

Attributes of a file system entry

int stat(const char *path_name, struct stat *buf);

- Attributes of a file system entry are stored in the inode.
- Attributes stored in the inode of path_name will be copied to the structure pointed to buf.
- The structure stat is system defined.
- Follows symbolic links.

```
struct stat stat_buf1, stat_buf2;

stat("./dir1/file1", &stat_buf1);
                        /* Attributes of ./dir1/file1  retreived*/

stat("sf", &stat_buf2);
            /* sf is a soft link to file2. Attributes of file2 retreived. */
```

Logic Option

Attributes Contd...

```
int lstat(const char *file_name, struct stat *buf);
struct stat stat_buf1, stat_buf2;
stat("sf", &stat_buf1);
        /* sf is a soft link to file2. Attributes of sf retreived. */
int fstat(int filedes, struct stat *buf);
int fd = open("file1", O_WRONLY);
fstat(fd, &stat_buf2);
/* Attributes of file file1 retreived. */
```

Stat structure...

```
struct stat {
        dev_t     st_dev;     /* device */
        ino_t     st_ino;     /* inode */
        mode_t    st_mode;    /* protection */
        nlink_t   st_nlink;   /* number of hard links */
        uid_t     st_uid;     /* user ID of owner */
        gid_t     st_gid;     /* group ID of owner */
        dev_t     st_rdev;    /* device type (if inode device) */
        off_t     st_size;    /* total size, in bytes */
        blksize_t st_blksize; /* blocksize for filesystem I/O */
        blkcnt_t  st_blocks;  /* number of 512 byte blocks allocated */
        time_t    st_atime;   /* time of last access */
        time_t    st_mtime;   /* time of last modification */
        time_t    st_ctime;   /* time of last change */
          };
```

Access checking

int access(const char *pathname, int mode);

- mode - combination of R_OK, W_OK, X_OK and F_OK
- Determines if the real user id of the current process is allowed all the accesses given in mode on the file pathname.
- Returns zero if all accesses in mode are allowed, else -1.
- Search access required on all ancestor directories in the pathname.

int retval = access("/home/kumar/dir1/f1", R_OK | W_OK);

Succeeds if search access on :

/, /home, /home/kumar and /home/kumar/dir1

read and write access on : /home/kumar/dir1/f1

Logic Option

Access checking

int retval = access("f1", R_OK | W_OK);

Succeeds if read and write allowed on "f1" which resides in the current directory.

If current dir is "/tmp/dir1" obviously search allowed on /, /tmp, /tmp/dir1

retval = access("f2", F_OK);

Succeeds if file "f2" exists in the current directory.

retval = access("dir1", R_OK | X_OK);

Succeeds if read and execute allowed on directory "dir1" which resides in the current directory.

Changing access permissions

int chmod(const char *path, mode_t mode);

- Changes the security mask of path to the values given in mode.
- Mode specifies the exact permissions to be set on the file. umask has no effect, it is used only at file creation.
- Only the owner can modify the permissions.
- With super user privileges permissions of any file can be modified.

```
int retval = chmod("f1", S_IRUSR | S_IWUSR |
                         S_IRGRP | S_IWGRP |
                         S_ISUID );
```

Logic Option

Changing ownership

int chown(const char *path, uid_t owner, gid_t group);

- uid_t and gid_t are unsigned integral type.
- Sets the owner and group ids of path to the given values.
- Either owner or group can be -1, which will be ignored.
- Requires super user privileges to change the ownership.
- The owner of path can change the group ownership to one of the groups to which the owner belongs.

Deleting files

int unlink(const char *pathname);

Directory entry deleted. pathname can't be opened further.

Decrements link count bye one.

if link count equal to zero

then if no process has the file open currently

then Free inode

Release the data blocks of file if any.

else When the last process that has the

file open closes it

Free inode

Release the data blocks of file if any.

Logic Option

Deleting files

Prog1.c

```
main()
{
int fd = open( “myfile”,O_RDWR);
      sleep(500);  /* wait 500 secs */
      close(fd);
}
```

Prog2.c

```
main()
{
unlink(“myfile”);
/*Assume link count 0*/
}
```

```
# Assume link count of “myfile” is one
# and no other process has it open.
```

- Shell prompt> prog1 & # Execute prog2 next quickly.
- Shell prompt> prog2 # Directory entry only deleted. After

```
# 500 seconds sleep prog1 closes
# “myfile” then the inode and the data
# blocks of “myfile” are released.
```

Logic Option