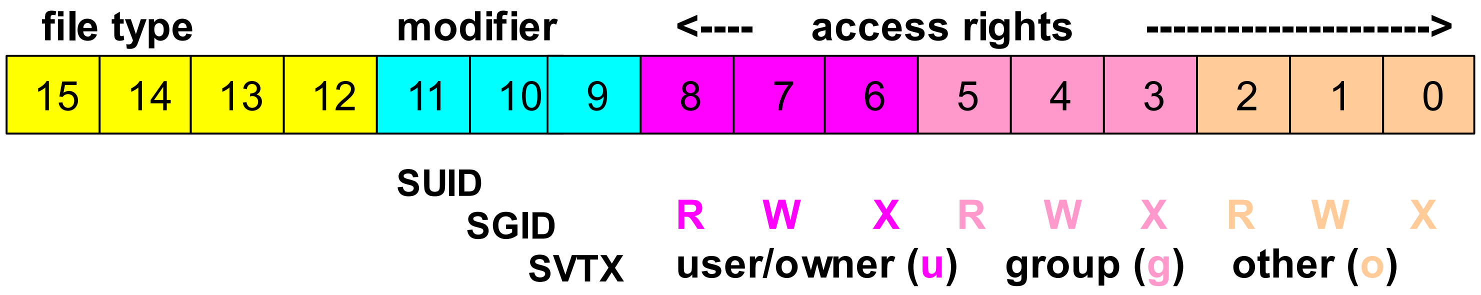
Input/Output – part 2

Last modification date: 05.11.2018

Low-level synchronous I/O

- **open** – R, W, RW modes, blocking vs unblocking access
- File attributes: **chmod**, **fchmod**, **fstat, lstat, open** (for a new file), **stat, umask**
- **close** – closing access (file session)
- **read/write** – sequential I/O operations
- File positioning for random access: **lseek**
- Signals and I/O operations
- Duplication of file descriptors: **dup**, **dup2**
- Descriptor tables, table of open files, table of i-nodes
- Testing „descriptor activity”: **select**

UNIX file attributes



File attribute	Value (octal)	Value (symbolic)
FIFO	0010000	S_IFIFO
Special character device	0020000	S_IFCHR
Directory	0040000	S_IFDIR
Special block device	0050000	S_IFBLK
Ordinary file	0100000	S_IFREG
Symbolic link	0120000	S_IFLNK
Socket	0140000	S_IFSOCK

Opening file session

- Opening access to an existing file (opening a new file session):

```
int  fd = open(const char *pathname, int oflag);
```

oflag determines access mode to a file with path **pathname**; logic sum of:

O_RDONLY	: read only access
O_WRONLY	: write only access
O_RDWR	: read-write access
O_TRUNC	: file truncated to 0 length before use
O_NONBLOCK, **O_NODELAY**	: non-blocking access mode
O_APPEND	: writing at the end of (appending) an existing file

Returned value **fd** depends on the result of the function call:

<0 : error exit (error code is specified with a global variable **errno**)

>=0 : unique (per process) **file descriptor**; file position is set at the file beginning, except for **O**_APPEND option, when it specifies the file end

Note: shell typically grants its children access to descriptors **0, 1, 2** – for standard input, output, error I/O operations (respectively)

Opening file session – cont.

- Opening access to possibly non-existing file:

```
int  fd = open(const char *pathname, int oflag, mode_t mode);
```

oflag determines access mode to a file with given path (**pathname**) as follows:

`O_CREAT` : creates a new file (if not existing) or opens existing file

`O_CREAT | O_EXCL` : creates a new file (fails if it exists)

`mode` determines access rights to a new file (`RWX-RWX-RWX`)

Effective access rights depend on current `umask` of the process executing `open`:

```
mode & ~umask
```

Typically `umask==0022`, which zeroes `W` bits for a group and „other users".

UID/GID for a new file == effective UID/GID of a process executing `open()`, i.e. if no `SUID/SGID` bit was used (nor `seteuid()/seteuid` executed) they are equal to effective EUID/EGID of a process which executed `exec()`; otherwise they are equal to UID/GID of the owner of the executable which created the process which called open.

Synchronous file I/O operations – cont.

- Retrieving attributes of a file with given path (`pathname`) or related to given file descriptor (`fd`):

```
int ret=stat(const char *pathname, struct stat *buf);

int ret=lstat(const char *pathname, struct stat *buf);

int ret=fstat(int fd, struct stat *buf);
```

If `ret==0`, then the function fill the structure pointed at by `buf` file attributes (for `lstat` – attributes of a symbolic link file, not the target file).

Important fields of `struct stat`:

`mode_t`	`st_mode`	: file attributes
`ino_t`	`st_ino`	: i-node number (file serial number, unique for a device)
`dev_t`	`st_dev`	: identifier of a device which stores the file
`nlink_t`	`st_nlink`	: the number of hard links to the file
`uid_t`	`st_uid`	: the user ID of the file's owner
`gid_t`	`st_gid`	: the group ID of the file
`off_t`	`st_size`	: the size of a regular file in bytes, for symbolic links - the length of the file name the link refers to
`time_t`	`st_mtime`	: the time of the last modification to the contents of the file

Synchronous file I/O operations – cont.

```
ssize_t ret = read(int fd, void *buf, size_t nbyte)
```

If **ret>0**, then `read()` stores in a buffer of `nbyte` bytes pointed at with `buf` exactly **ret<=nbyte** bytes, which were read from a file associated with the given descriptor `fd`; the reading started from the current file position at the moment of `open()` call. The file position is incremented by `ret`.

If **ret==0**, then no more data (end of file condition)

If **ret<0** (**ret==-1**), then the buffer has not been modified and the return code is specified with `errno`. Important `errno` values (see `read(2)`):

`EBADF` : invalid descriptor or file not opened for reading

`EINTR` : **read** operation was interrupted by a signal while it was blocked waiting for completion

`EAGAIN` : if the `O_NONBLOCK` flag is set for the file - `read()` can return immediately without reading any data and report this error

Synchronous file I/O operations – cont.

`ssize_t ret=write(int fd, const void *buf, size_t nbyte)`

If `ret>=0`, then exactly `ret<=nbyte` bytes was written from the buffer (of size `nbyte`) pointed at by `buf` to the file related to descriptor `fd`, starting from the file position which was current at the moment of `open()` call. The file position is incremented by `ret`.

If `ret<0` (`ret==-1`), then the write attempt was not successful, and the reason code is stored in `errno`. Important error codes (see `write(2)`):

EBADF : invalid descriptor or file not opened for writing

EAGAIN : normally `write()` blocks, until finished, but if the `O_NONBLOCK` flag is set for the file `write()` can return immediately without writing any data and report this error – if no write could have been performed immediately

EINTR : `write` operation was interrupted by a signal while it was blocked waiting for completion

EPIPE : trying to write to a pipe or FIFO that isn't open for reading by any process (the system also sends `SIGPIPE` signal)

Synchronous file I/O operations – cont.

Random access to a file

```
off_t  ret = lseek(int fd, off_t offset, int whence)
```

`lseek()` is used to change the file position of the file with descriptor `fd`

`whence` specifies how the `offset` should be interpreted:

`SEEK_SET`	: a count of characters from the beginning of the file
`SEEK_CUR`	: a count of characters from the current file position
`SEEK_END`	: a count of characters from the end of the file

The function returns (`ret`) the resulting file position, measured in bytes from the beginning of the file or `(off_t)-1` in case of failure (error code in `errno` - see `lseek(2)`).

Closing file session

`int ret = close(int fd)`

The function breaks association of the file descriptor **`fd`** with a file; returning 0 if successful. If **`ret<0`** **(`ret==-1`)**, then the error code is in **`errno`**. Important error codes (see **`close(2)`**):

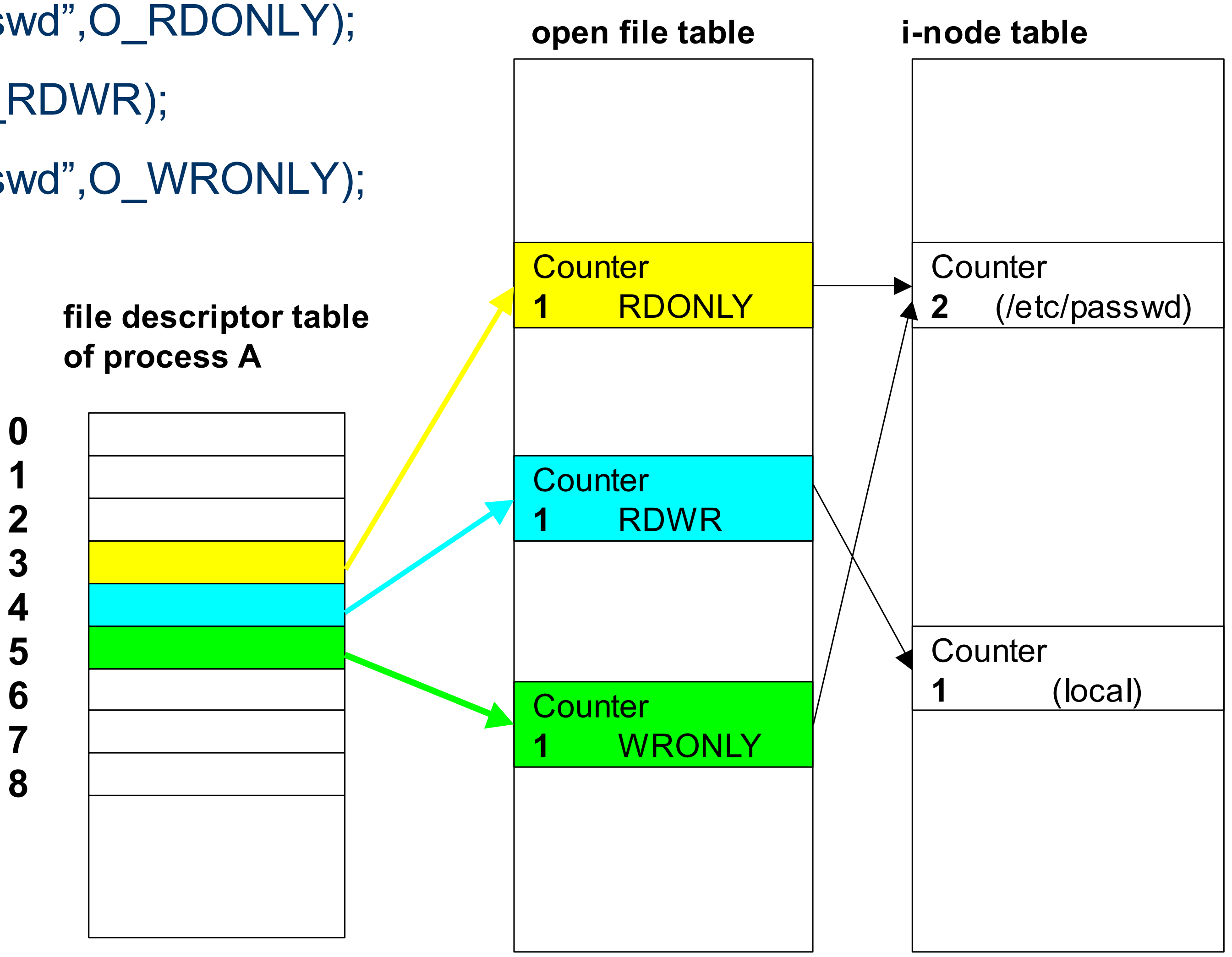
`EBADF`: invalid descriptor

`EINTR`: The close call was interrupted by a signal.

I/O operations – data structures (UNIX)

Example Process **A** executes:

```
fd1=open(”/etc/passwd”,O_RDONLY);
fd2=open(”local”,O_RDWR);
fd3=open(”/etc/passwd”,O_WRONLY);
```

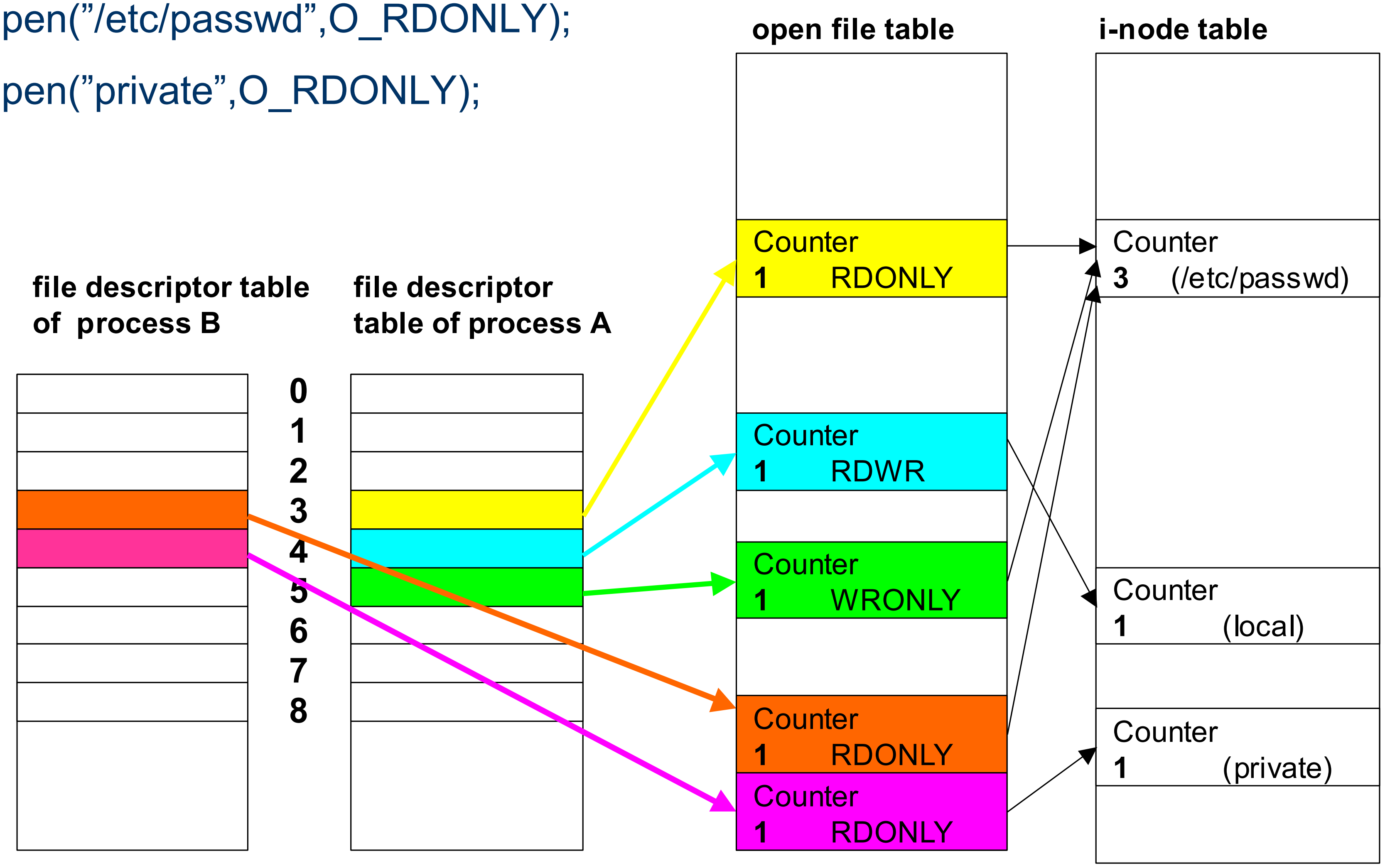


I/O operations – data structures (UNIX)

Przykład Process **B** executes:

fd1=open(”/etc/passwd”,O_RDONLY);

fd2=open(”private”,O_RDONLY);



Synchronous file I/O operations – cont.

`int dup (int old)`

This function copies descriptor `old` to the first available descriptor number (the first number not currently open).

`int dup2 (int old, int new)`

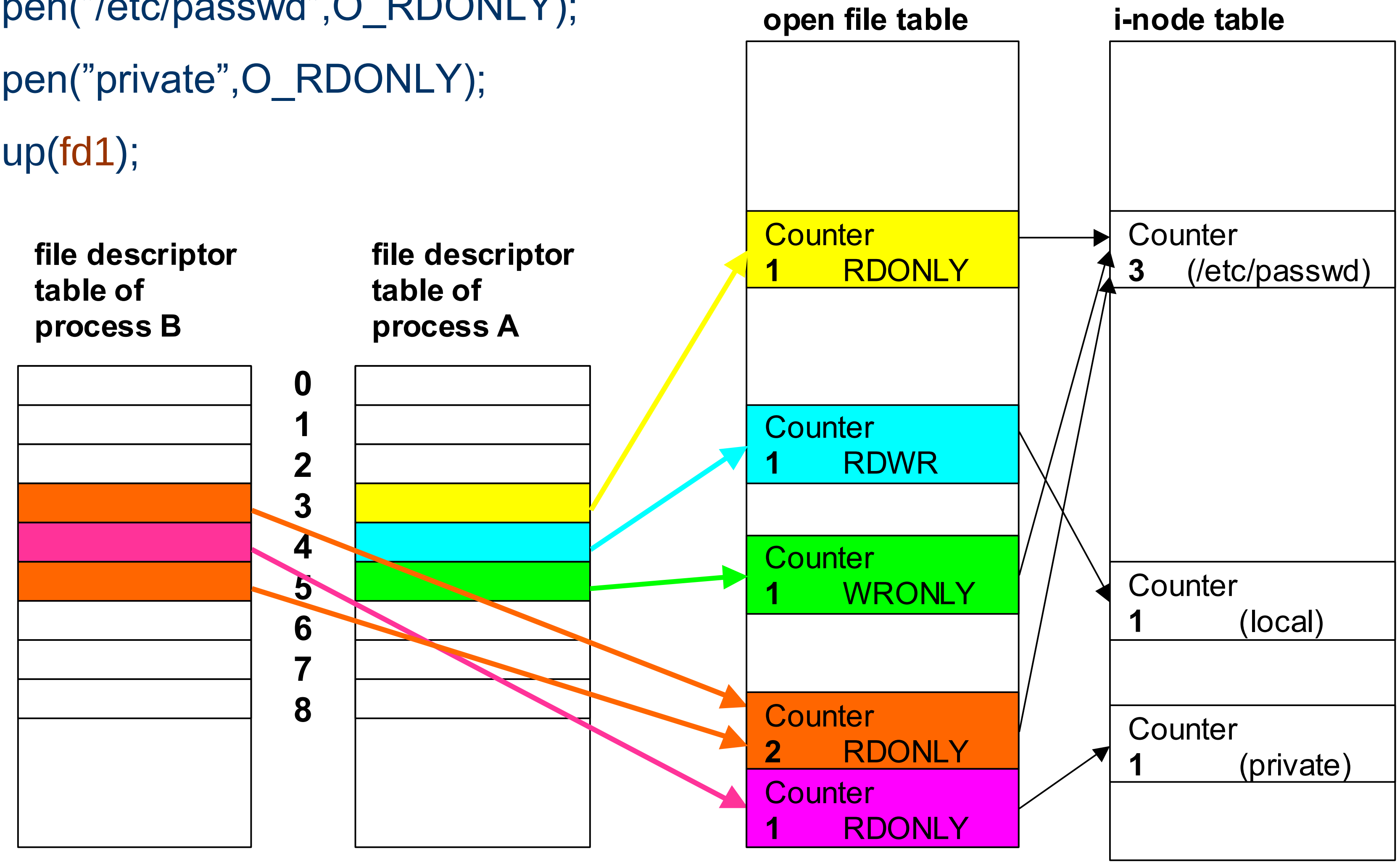
This function copies the descriptor `old` to descriptor number `new`

The functions return the new descriptor (>=0) or –1 if not successful (error code in `errno`).

I/O operations – data structures (UNIX)

Example Process **B** executes:

```
fd1=open(”/etc/passwd”,O_RDONLY);
fd2=open(”private”,O_RDONLY);
fd3=dup(fd1);
```



I/O operations – data structures (UNIX)

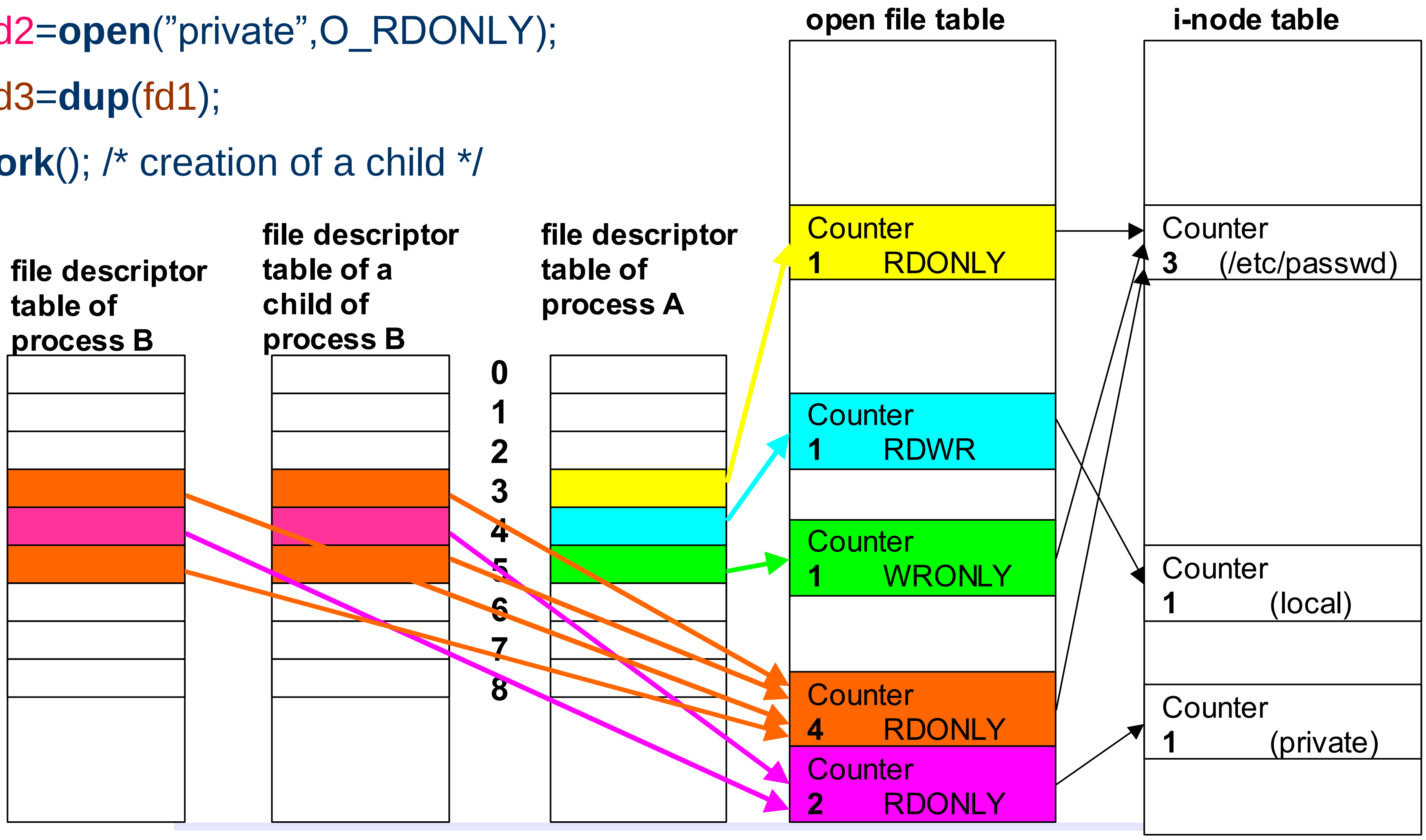
Example Process **B** executes:

fd1=**open**(”/etc/passwd”,O_RDONLY);

fd2=**open**(”private”,O_RDONLY);

fd3=**dup**(fd1);

fork(); /* creation of a child */



Synchronous file I/O operations – cont.

Summary:

- If a process opens several times the same file, then the descriptors point at different elements of the open file table, but the same element of i-node tables is used to refer to the file items.
- Two elements of a file descriptor table can point at the same element of the open file table, when `dup()` (or `dup2()`) was used to make one of the descriptors out of the other.
- Information on the current file position is stored in the open file table, so it is common to descriptors that point at the same element of the open file table.
- In traditional UNIX elements of two different file descriptor tables can point at the same open file table only when one process is a descendant of the other. In such a case changes of the file positions by one process are seen by the other process. In modern UNIX-like systems the file positions for these processes can be found disjoint.

Synchronous file I/O operations – cont.

```
int ret = select(int n, fd_set *readfds, fd_set *writefds,
         fd_set *exceptfds, struct timeval *timeout)
```

The function blocks the calling process until there is activity on any of the specified sets of file descriptors, or until the timeout period has expired

`readfds`	: mask of read descriptors
`writefds`	: mask of write descriptors
`exceptfds`	: mask of descriptors which can receive Out Of Band (OOB) data
`n`	: the number of descriptor (in mask) to check

`timeout`: if NULL => indefinite wait, otherwise a pointer at a timeout structure

`ret` **>0** the total number of ready file descriptors in all of the bit masks

Macros for bit mask manipulation

`FD_ZERO(fd_set *set);` - zeroes a mask pointed at by `set`

`FD_SET(int fd, fd_set *set);` - sets specified bit (nr `fd`) in a mask (pointed with `set`)

`FD_CLR(int fd, fd_set *set);` - clears specified bit in the mask

`FD_ISSET(int fd, fd_set *set);` - checks if the specified bit of the mask is set

Synchronous file I/O operations – cont.

```
/* copying data from 2 inputs (fd1,fd2) to the standard output (code excerpt) */
fd_set readfds;
int fd1= ..., fd2= ..., maxfd, ret, towrite;
. . .
  for(;;){
        maxfd=(fd1>fd2) ? fd1 : fd2;
        if(maxfd<0) break;  // no descriptor can be active (end of copying)
        FD_ZERO(&readfds);
        if(fd1>=0) FD_SET(fd1,&readfds);
        if(fd2>=0) FD_SET(fd2,&readfds);
        if(select(maxfd+1,&readfds,0,0,0) < 0){// blocking check of descriptors
           if(errno==EINTR) continue; else { perror("select"); ... }
        }
        if (fd1>=0 && FD_ISSET(fd1,&readfds)){// is descriptor ready?
           if( (towrite=read(fd1,buf,sizeof(buf))) < 0 ){ ; ... } // error ?
           if(towrite>0){
                p=buf;
                while(towrite>0){// note: write might not output all towrite  bytes
                   ret=write(STDOUT_FILENO,p,towrite); // in one call
                   if(ret<0){... } // error ?
                   towrite -= ret; p += ret;
                }
           } else {// towrite<=0
                close(fd1); fd1=-1;
                fprintf(stderr,„End of data 1\n");
           }
        }
        . . . // Similar code for fd2 descriptor
  }
```

Miscellaneous

`STDIN_FILENO, STDOUT_FILENO, STDERR_FILENO` - names for descriptor files beneath the standard streams: `stdin, stdout, stderr` (traditionally: 0, 1, 2)

```
FILE *fdopen(int fildes, const char *mode);
```

associates a stream with a file descriptor `fildes`. The mode of the stream (r/rb, w/wb, a/ab, r+/rb+, w+/wb+, a+/ab+) should be allowed by the file access mode of the open file description to which `fildes` refers.

```
int fileno(FILE *stream);
```

maps a stream pointer to a file descriptor

```
int fsync(int fildes);
```

Waits until data associated with the open file descriptor `fildes` is written to device. Note: `void sync(void);` waits for all descriptors to synchronize.