System Programming: File Management

Raymond Namyst

Dept. of Computer Science

University of Bordeaux, France

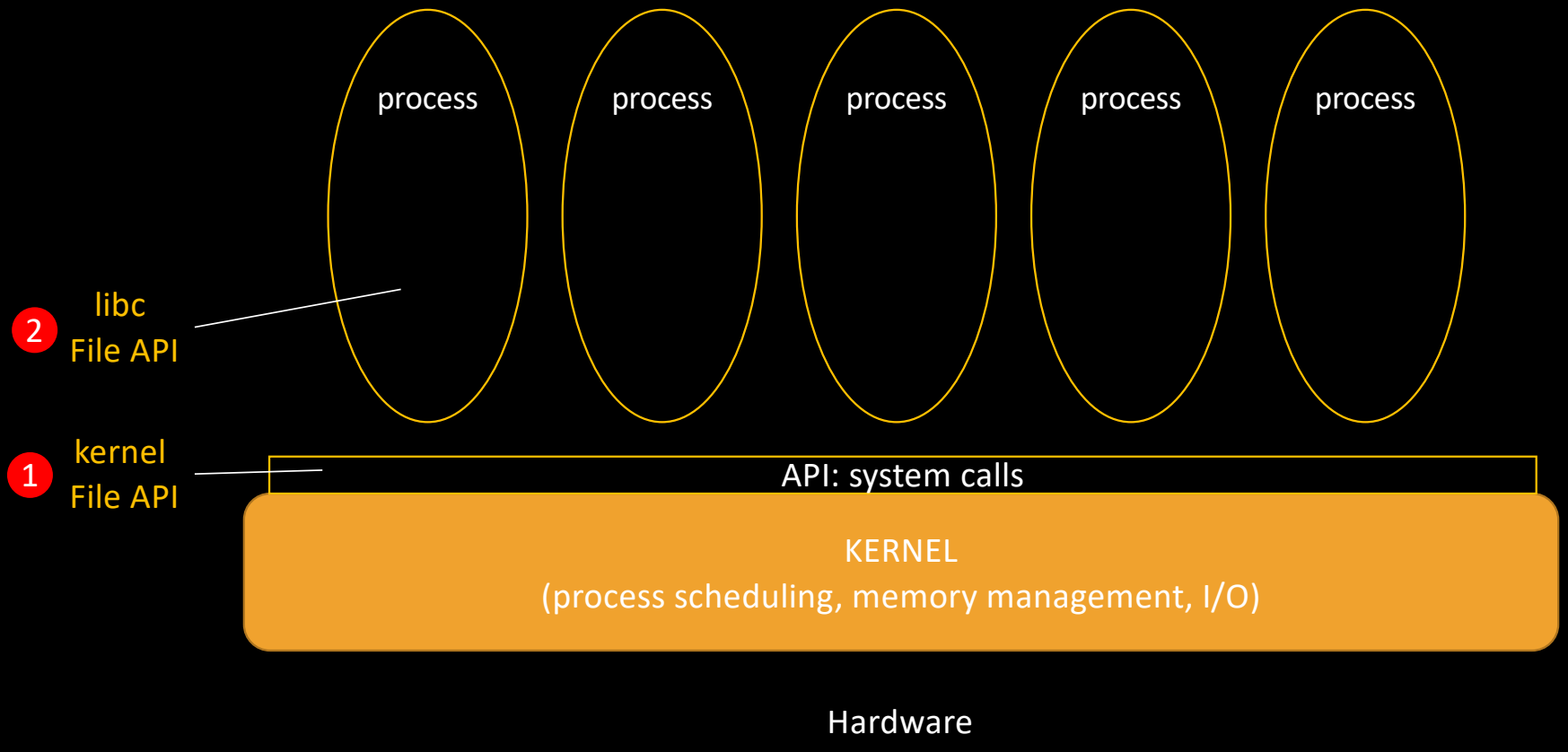
https://gforgeron.gitlab.io/progsys/

The concept of File

- A central concept in Unix
  - "*All is File*" philosophy
    - Regular disk files, but also
      - Terminal
      - Devices (mouse, keyboard)
      - Network sockets
      - Etc.

- User view of a Disk File
  - Contiguous series of bytes
  - Known length, but may expand/shrink dynamically
  - Access rights (`rwx`)
  - Can be referenced by multiple links (paths)

Two File Management APIs



The concept of File

- Before we can read from/write into a file, we must *open* it
  - Why can’t we just read directly ?
    - read (“/net/cremi/dupont/myfile.txt”, buffer1, …)
    - read (“/net/cremi/dupont/myfile.txt”, buffer2, …)
    - …

The concept of File

- Before we can read from/write into a file, we must *open* it
  - Why can't we just read directly ?
    - read ("/net/cremi/dupont/myfile.txt", buffer1, …)
    - read ("/net/cremi/dupont/myfile.txt", buffer2, …)
    - …
  - Partly for efficiency reasons
    - To access file "/net/cremi/dupont/myfile.txt", the OS must check
      - That there is a "net" entry in the "/" directory
      - That "/net" is a directory and that the user can traverse it (x)
      - That there is a "cremi" entry in the "/net" directory
      - That "/net/cremi" is a directory and that the user can traverse it (x)
      - …
      - That "/net/cremi/dupont/myfile.txt" is a file and that the user can read it (r)

Opening Files

- Before we can read from/write into a file, we must *open* it

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

  - Open performs the appropriate checks, and returns a *file descriptor*
    - This file descriptor is a *key* which will
      - Accelerate upcoming read/write operations
      - Maintain the “current position” in the file

Opening Files

- Before we can read from/write into a file, we must *open* it

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

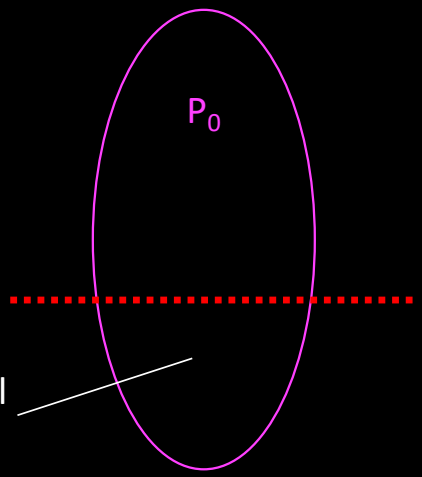
  - Open performs the appropriate checks, and returns a *file descriptor*
    - This file descriptor is a *key* which will
      - Accelerate upcoming read/write operations
      - Maintain the "current position" in the file
  - `oflag`:
    - `O_RDONLY`, `O_WRONLY` or `O_RDWR`
    - Optional: `O_CREAT`, `O_TRUNC`, `O_SYNC`, etc.
  - When a file is created, the third parameter sets access rights (octal notation)
    - 0750 = 111 101 000 = `rwxr-x---`
    - 0666 = 110 110 110 = `rw-rw-rw-`

Opening Files

- See ouverture.c…

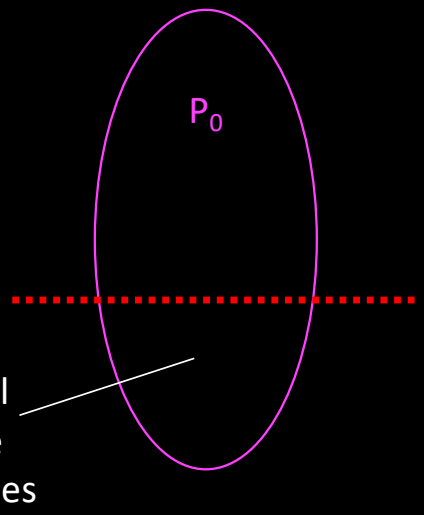
Side note about process representation

Processes can be represented this way:

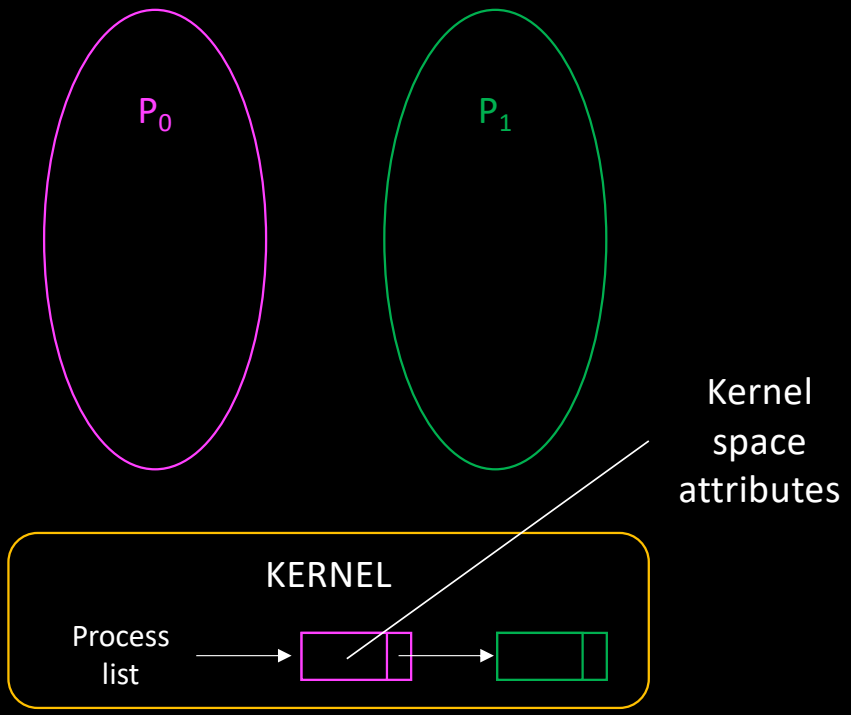


Side note about process representation

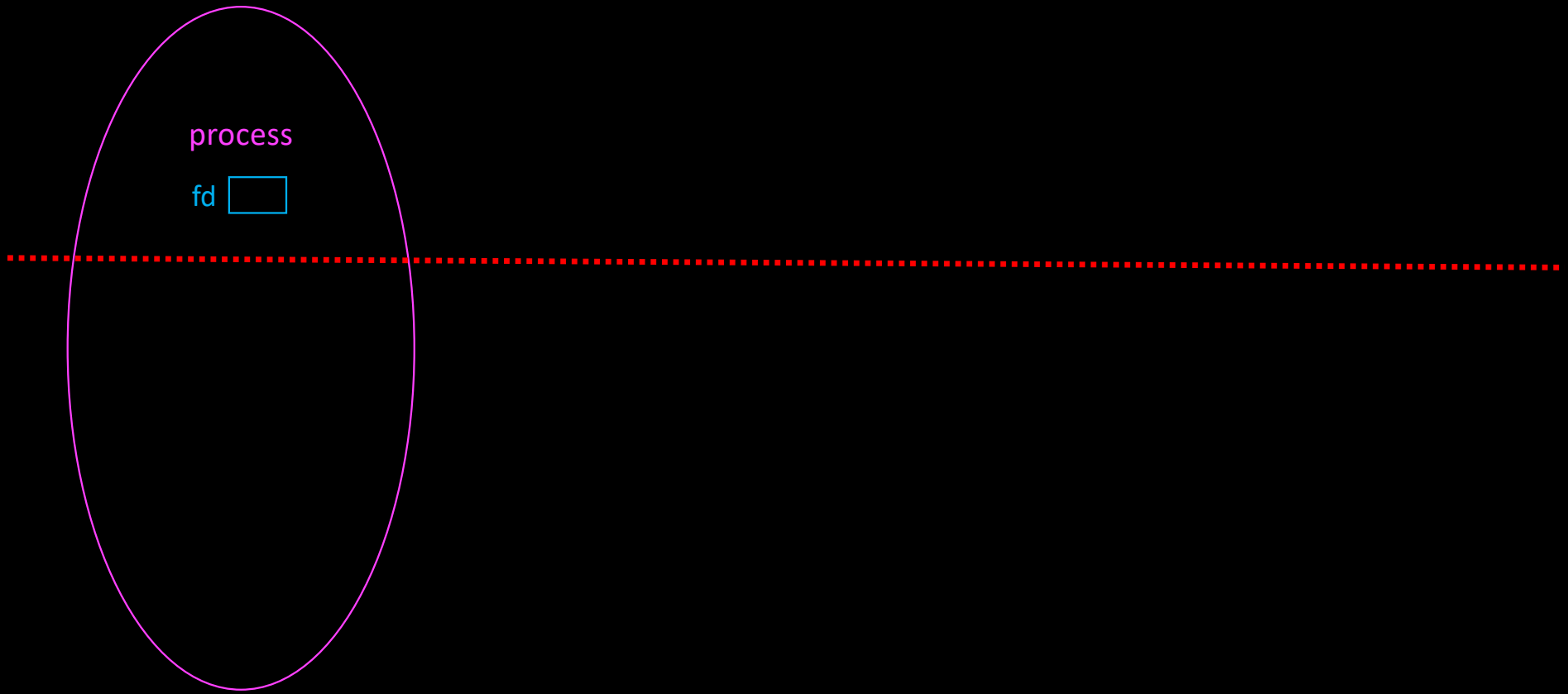
Processes can be represented this way:



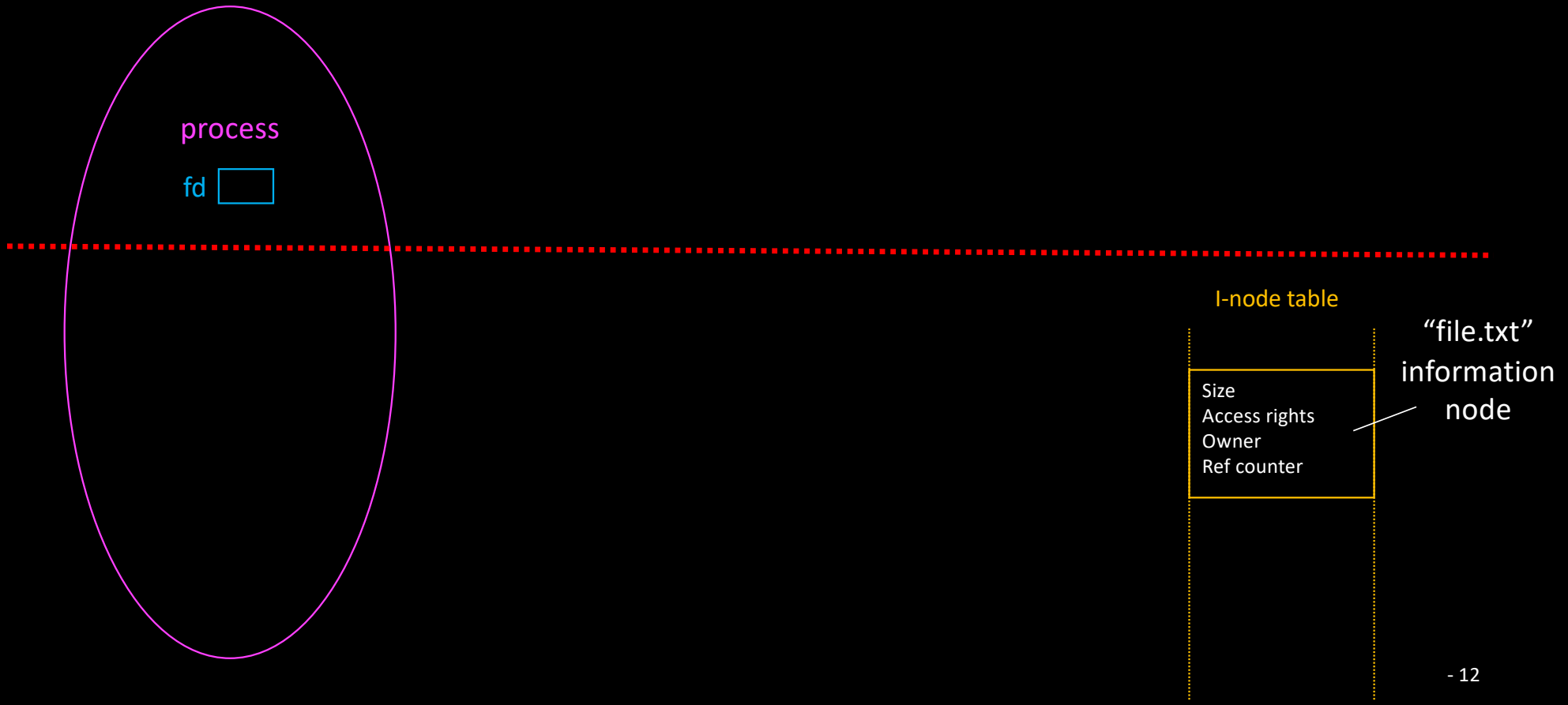
But reality is (obviously) more like:



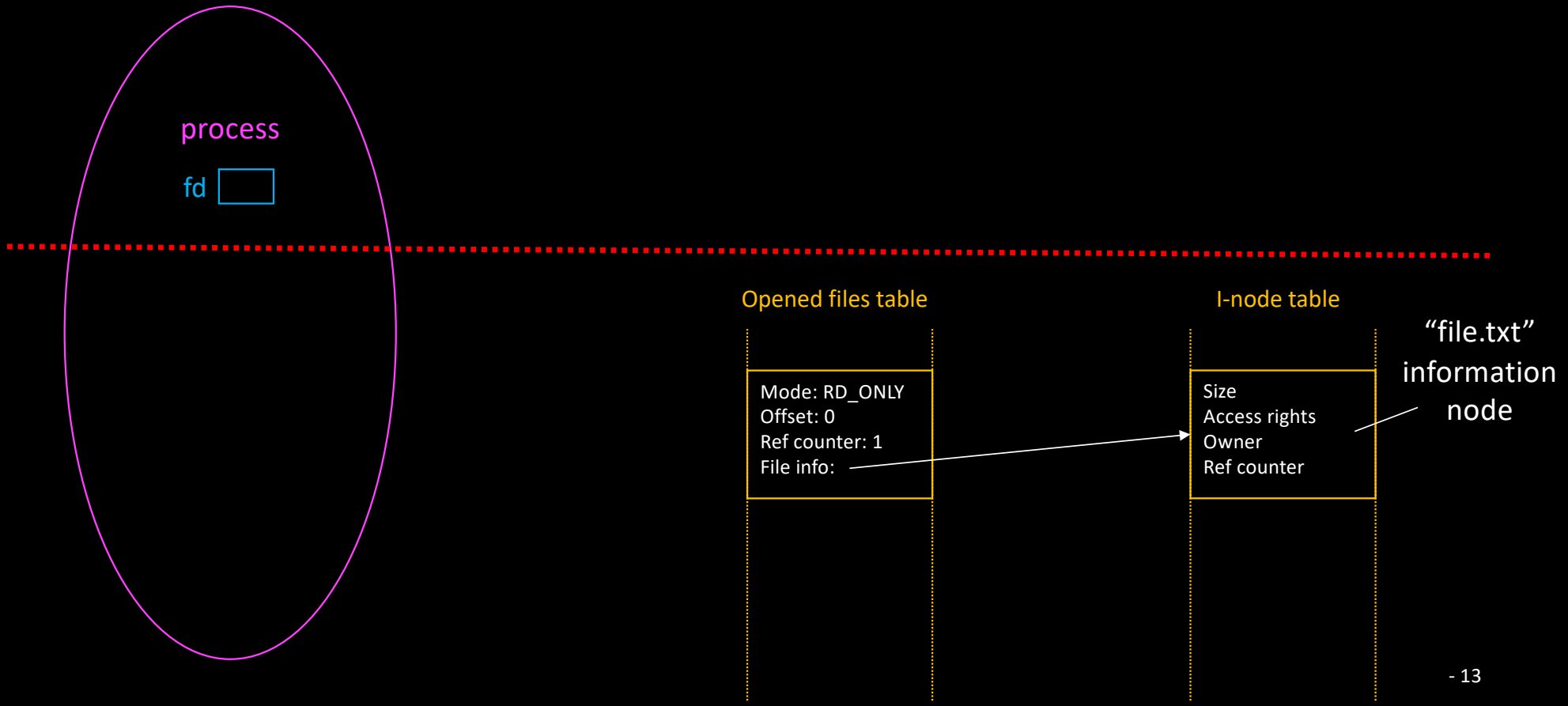
fd = open (“file.txt”, O_RDONLY);



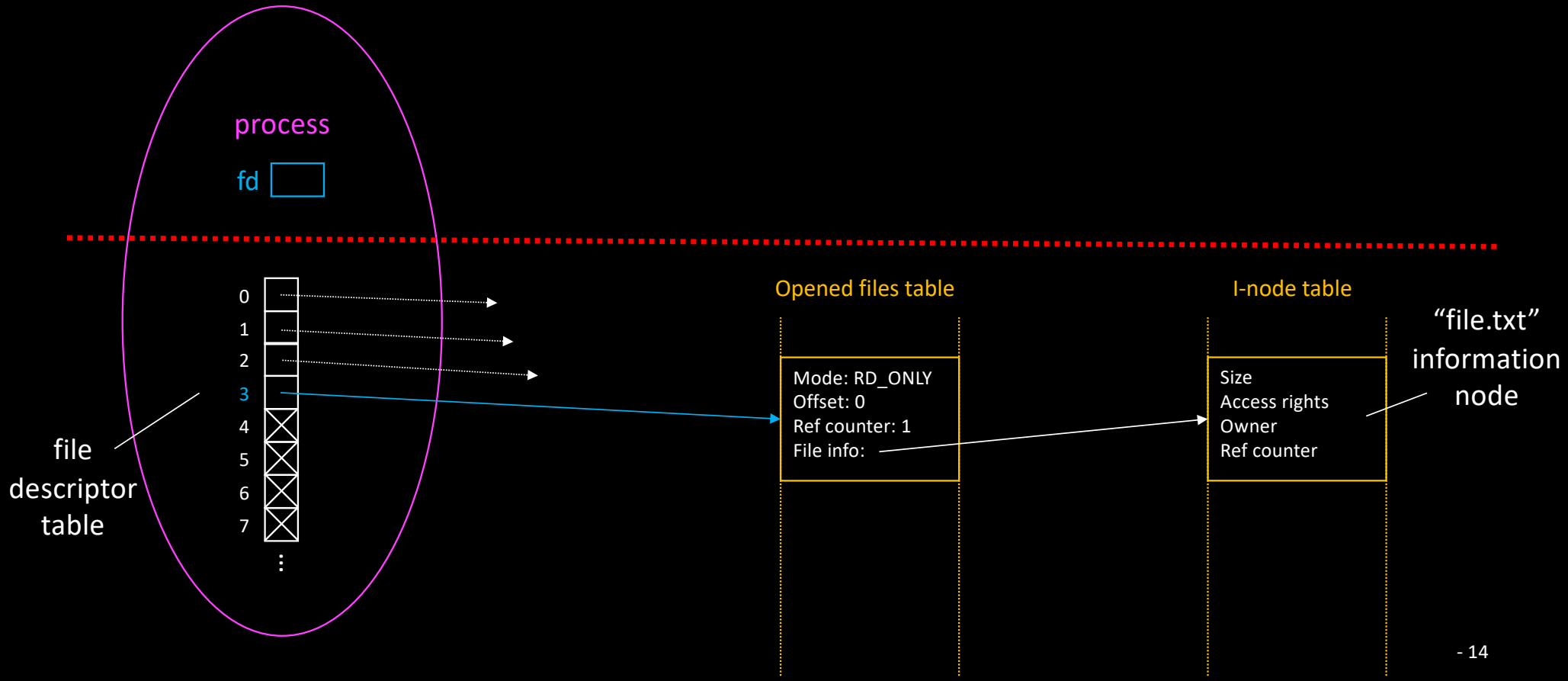
fd = open ("file.txt", O_RDONLY);



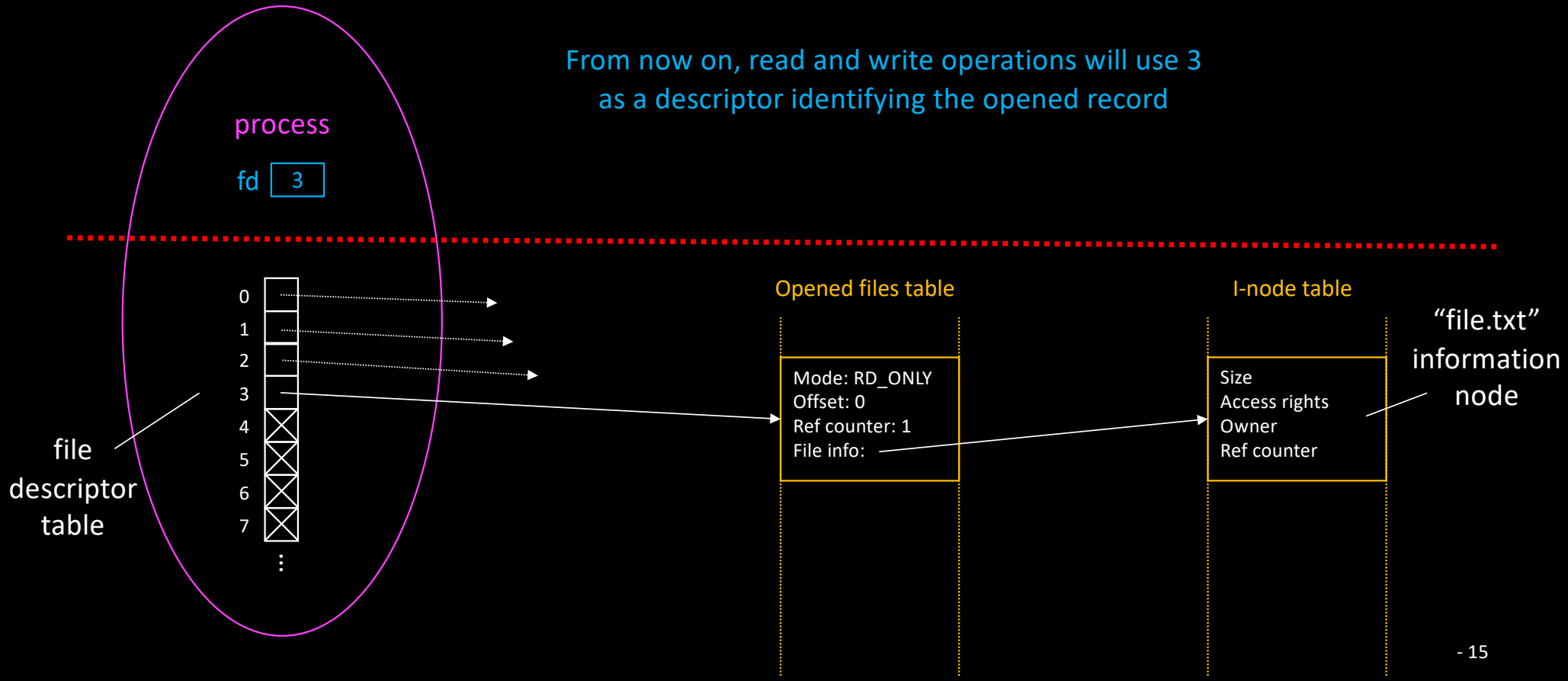
fd = open ("file.txt", O_RDONLY);



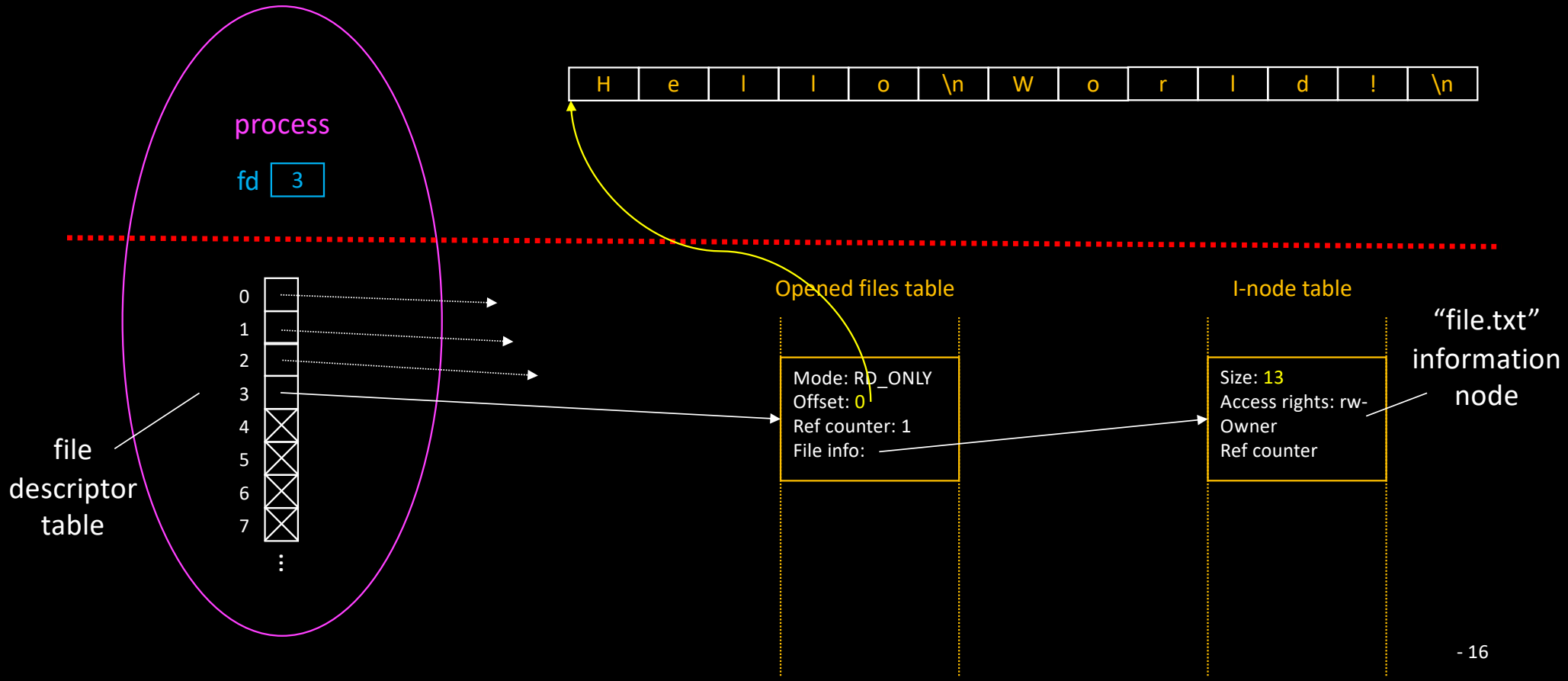
fd = open ("file.txt", O_RDONLY);



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



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



Reading from a File

- Reading from the file and writing to process memory

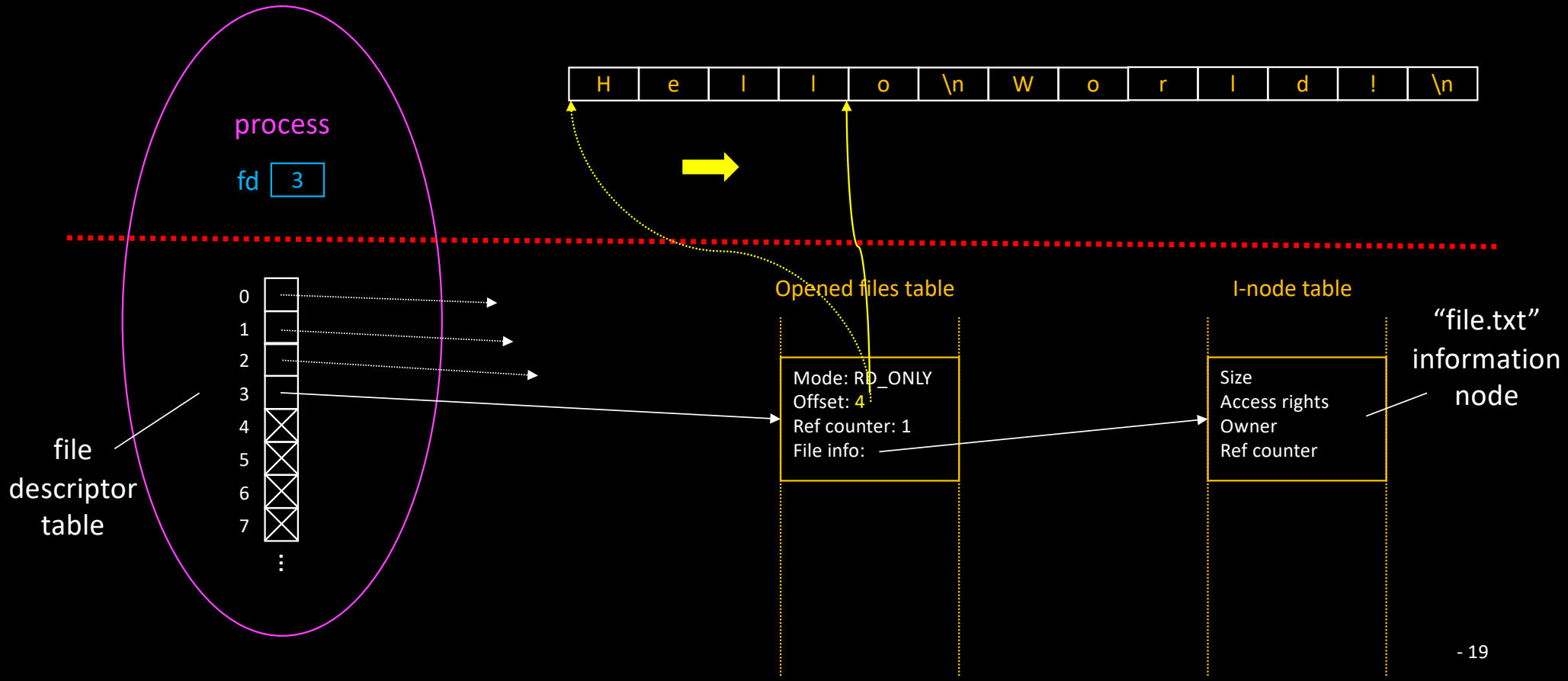
```
ssize_t read(int fildes, void *buf, size_t nbyte);
```

  - Return value:
    - -1 if error
    - Number of bytes read (and copied to buf)
      - If zero, then we have reached the End Of File

Reading from a File

- See lecture.c

read (fd, buffer, 4)



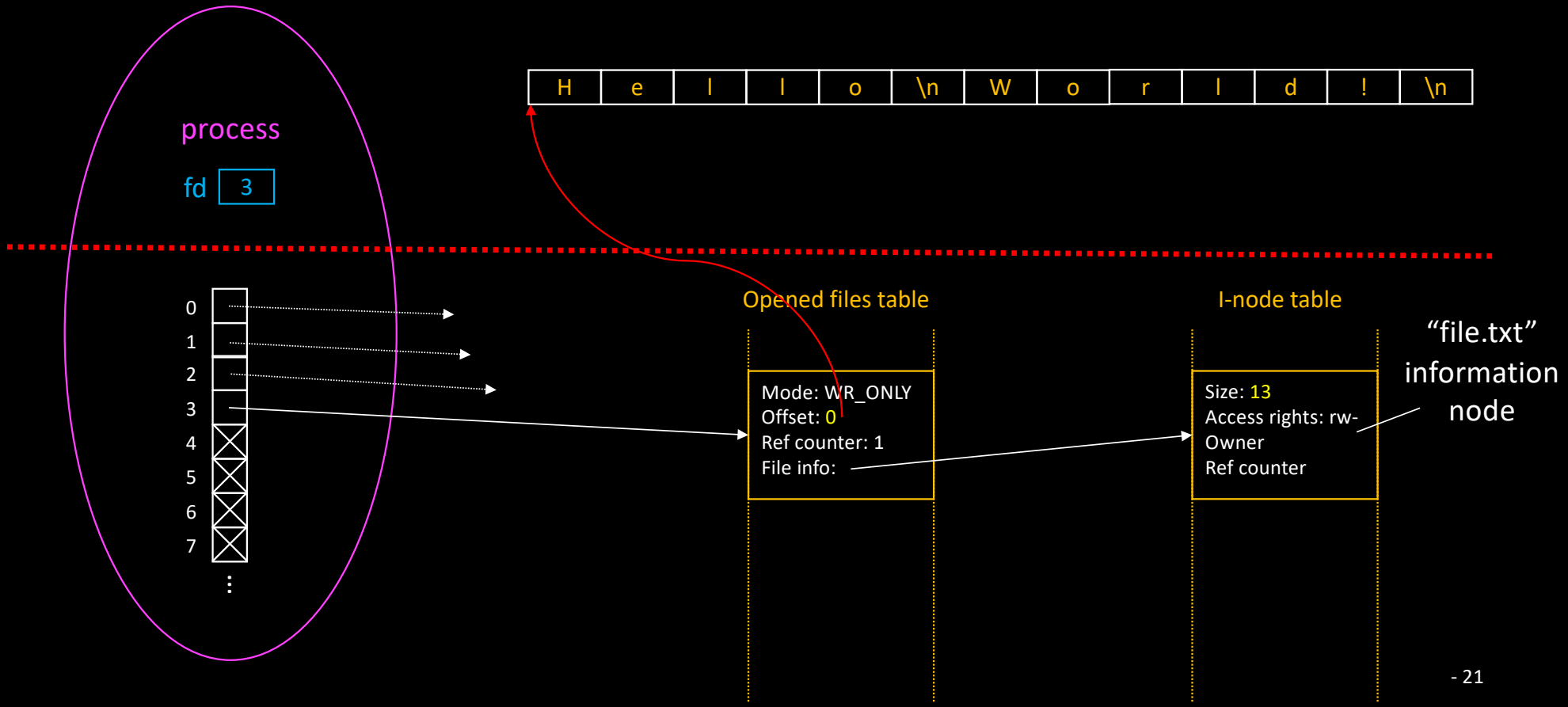
Writing into a File

- Reading from memory and writing to the file

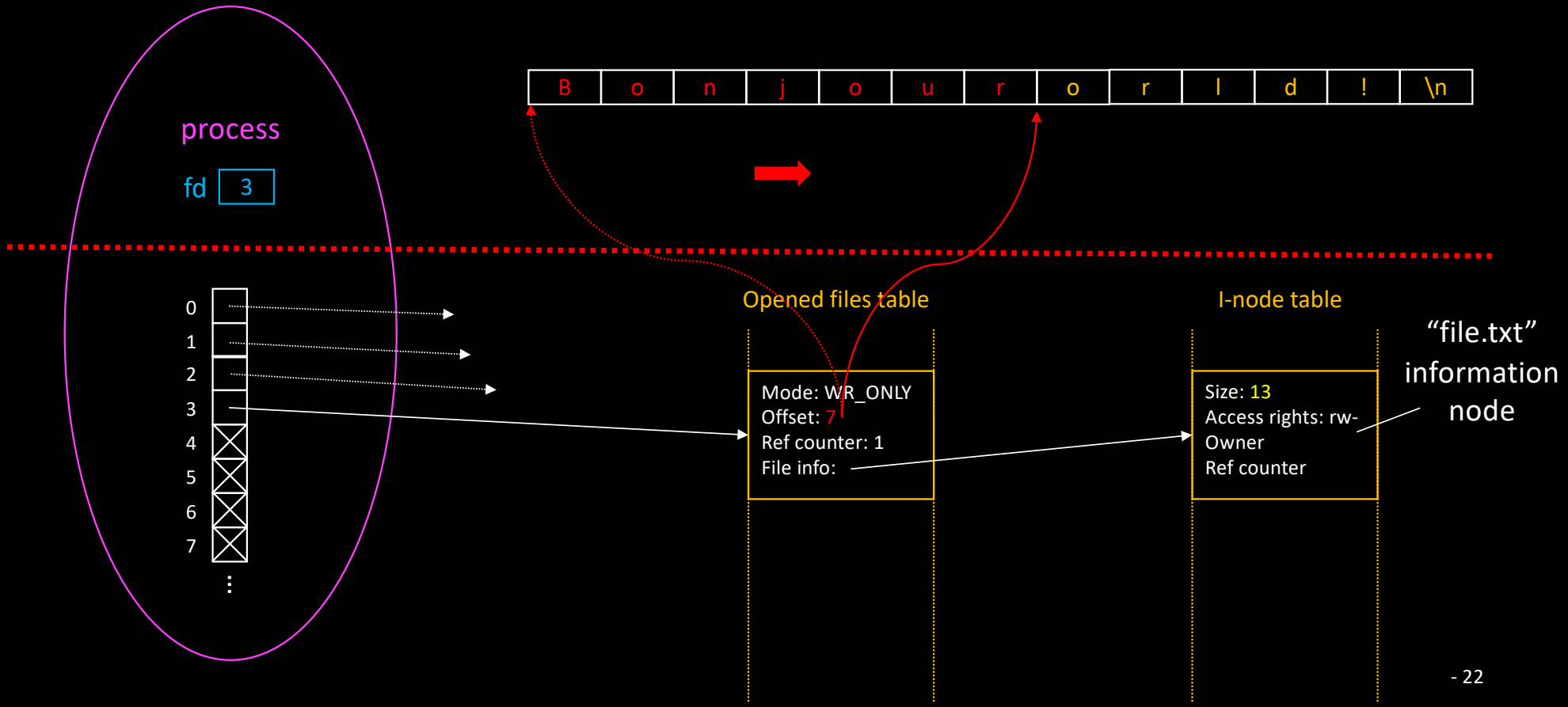
```
ssize_t write(int fildes, void *buf, size_t nbyte);
```

  - Return value:
    - -1 if error
    - Number of bytes written
      - Can be less than `nbyte`…
  - Writing beyond end-of-file automatically extends the file

fd = open ("file.txt", O_WRONLY | O_CREAT, 0666);



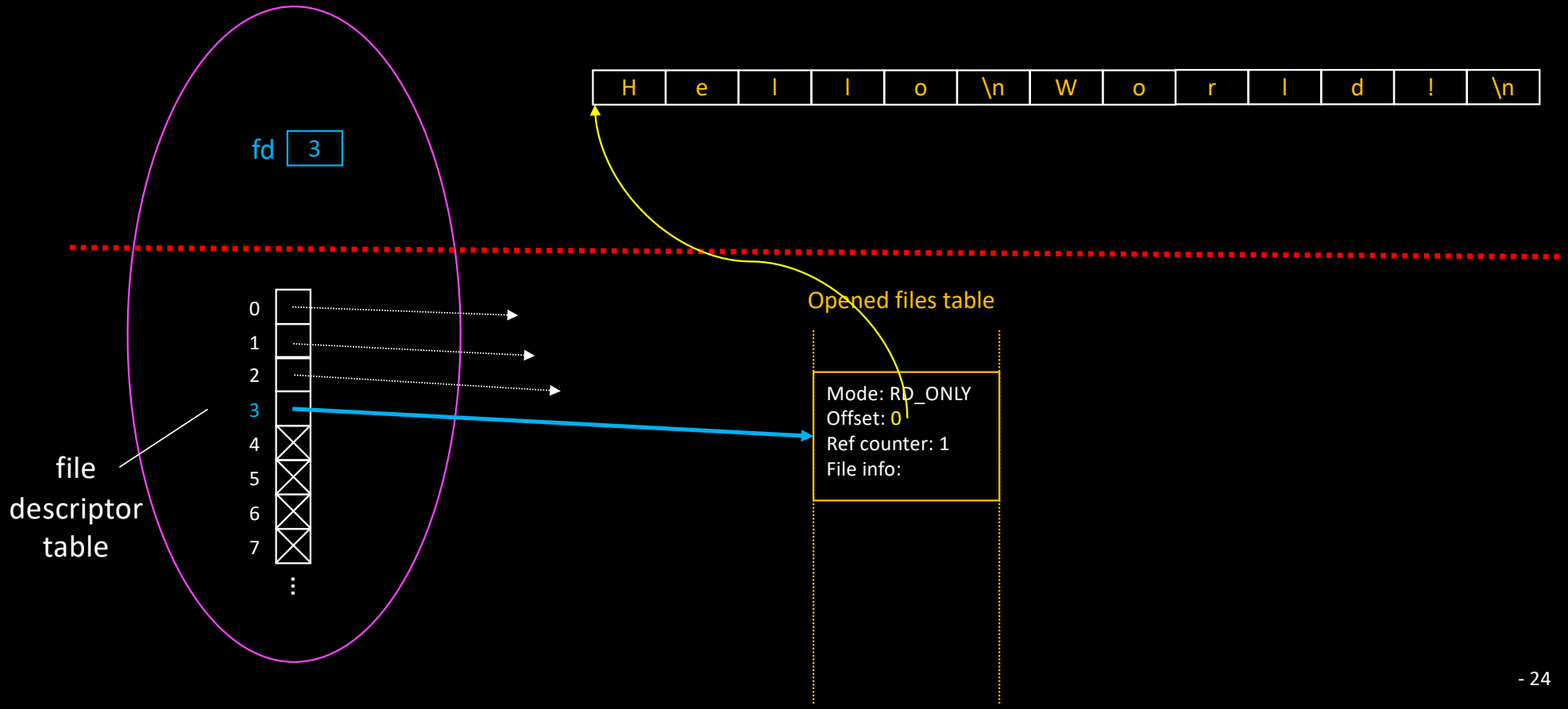
write (fd, “Bonjour”, 7);



Writing into a File

- See ecriture.c

Take away of previous lecture



Coping with structured data

- How to store integers into a file?
  - Say we have a function that generates a series of integers
- There are several ways
    - Ascii representation
    - Raw integers

Coping with structured data

- Writing the ascii representation into a File
  - Convert integer into string
    - E.g. using `sprintf`
  - Write string to file
    - Think about a separator…
- Pros
  - Readable like a text file
  - Portable
- Cons

Coping with structured data

- Writing the ascii representation into a File
  - Convert integer into string
    - E.g. using `sprintf`
  - Write string to file
    - Think about a separator…
- Pros
  - Readable like a text file
  - Portable
- Cons
  - Reading back the file
    - Finding separators
    - Convert string back to integer
  - No efficient (direct) access to the N^{th} integer

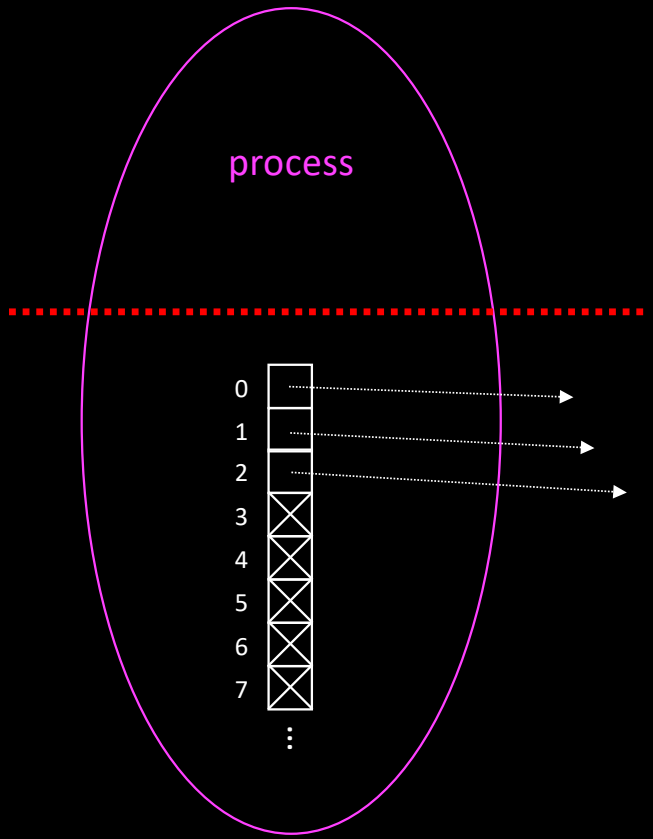
Coping with structured data

- Writing the binary representation into a File
  - `int i = …; write (fd, &i, sizeof(int));`
- Pros
  - Performance of both read/write operations
  - Efficient (direct) access to the N^{th} integer
    - Index files
- Cons

Coping with structured data

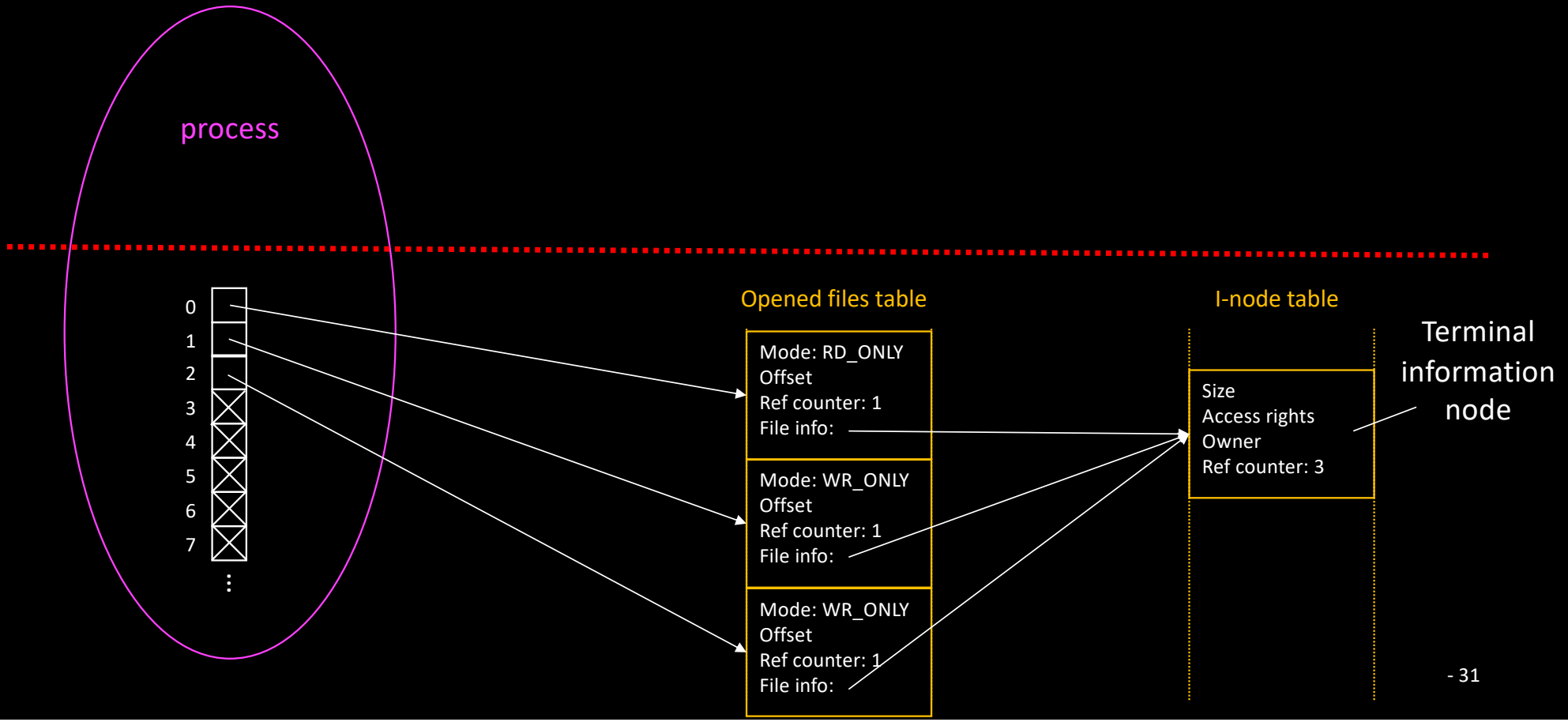
- Writing the binary representation into a File
  - `int i = …; write (fd, &i, sizeof(int));`
- Pros
  - Performance of both read/write operations
  - Efficient (direct) access to the N^{th} integer
    - Index files
- Cons
  - File readability
    - Hexdump ☺
  - Portability
    - Only works if file is generated AND accessed on the same processor architecture

Pre-opened descriptors

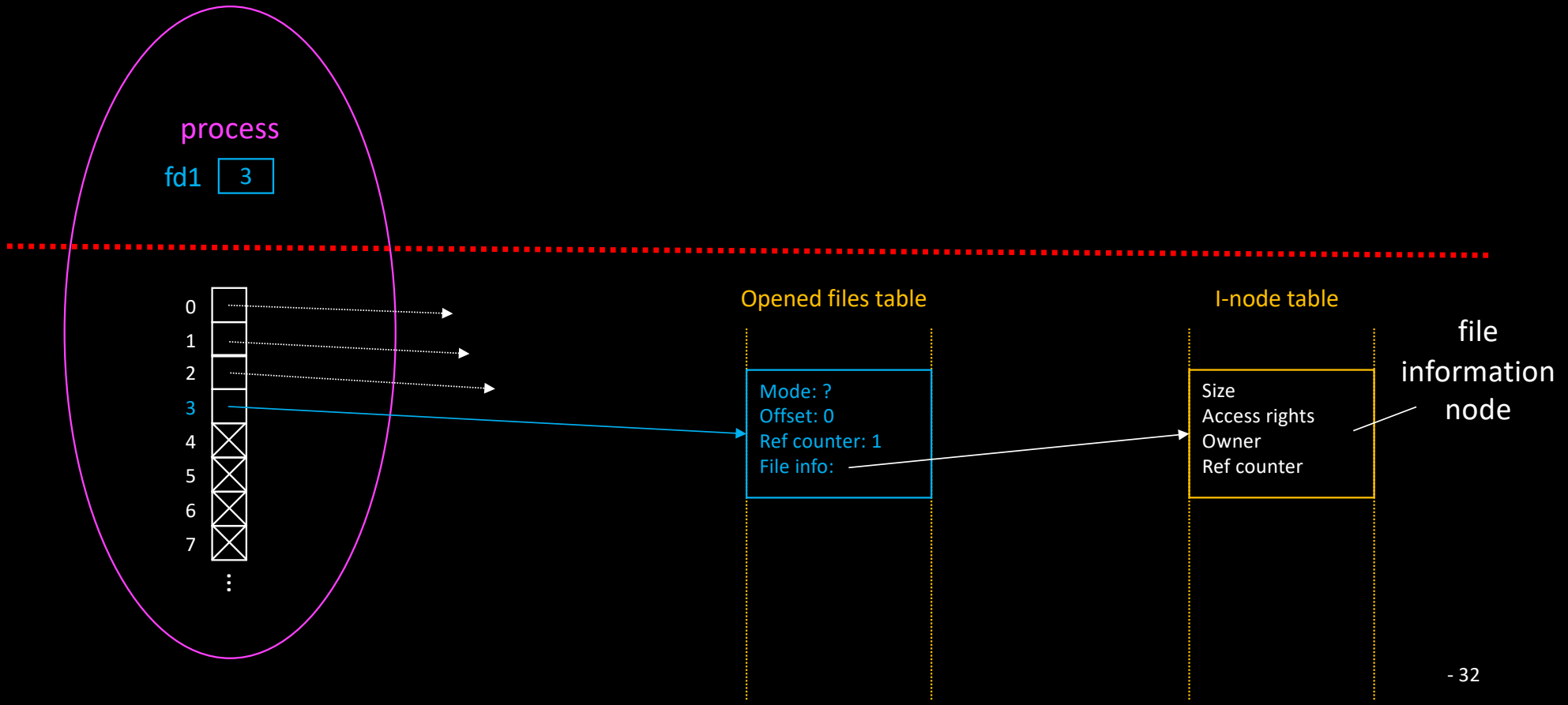


- Standard input/output
  - E.g. Terminal
    - `0: STDIN_FILENO`
    - `1: STDOUT_FILENO`
    - `2: STDERR_FILENO`

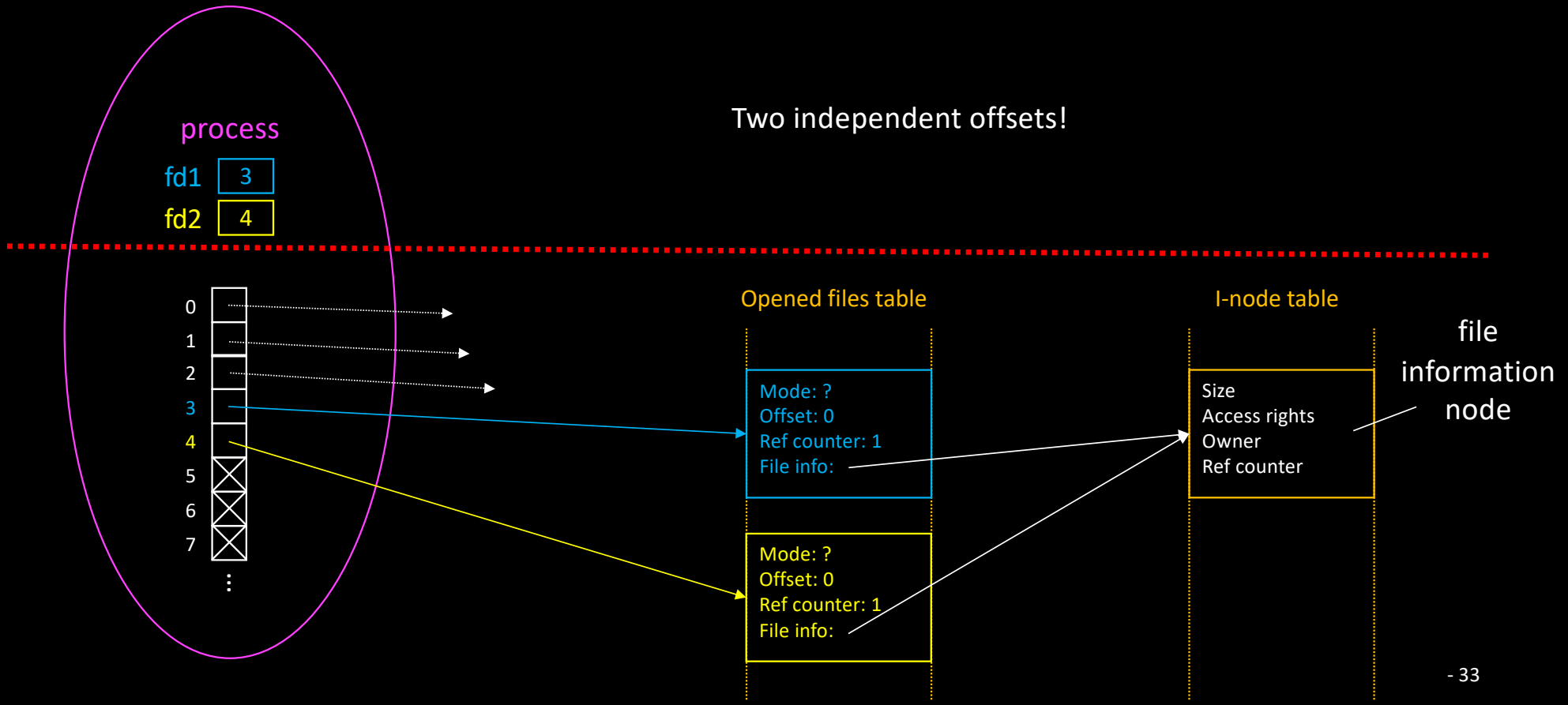
Pre-opened descriptors



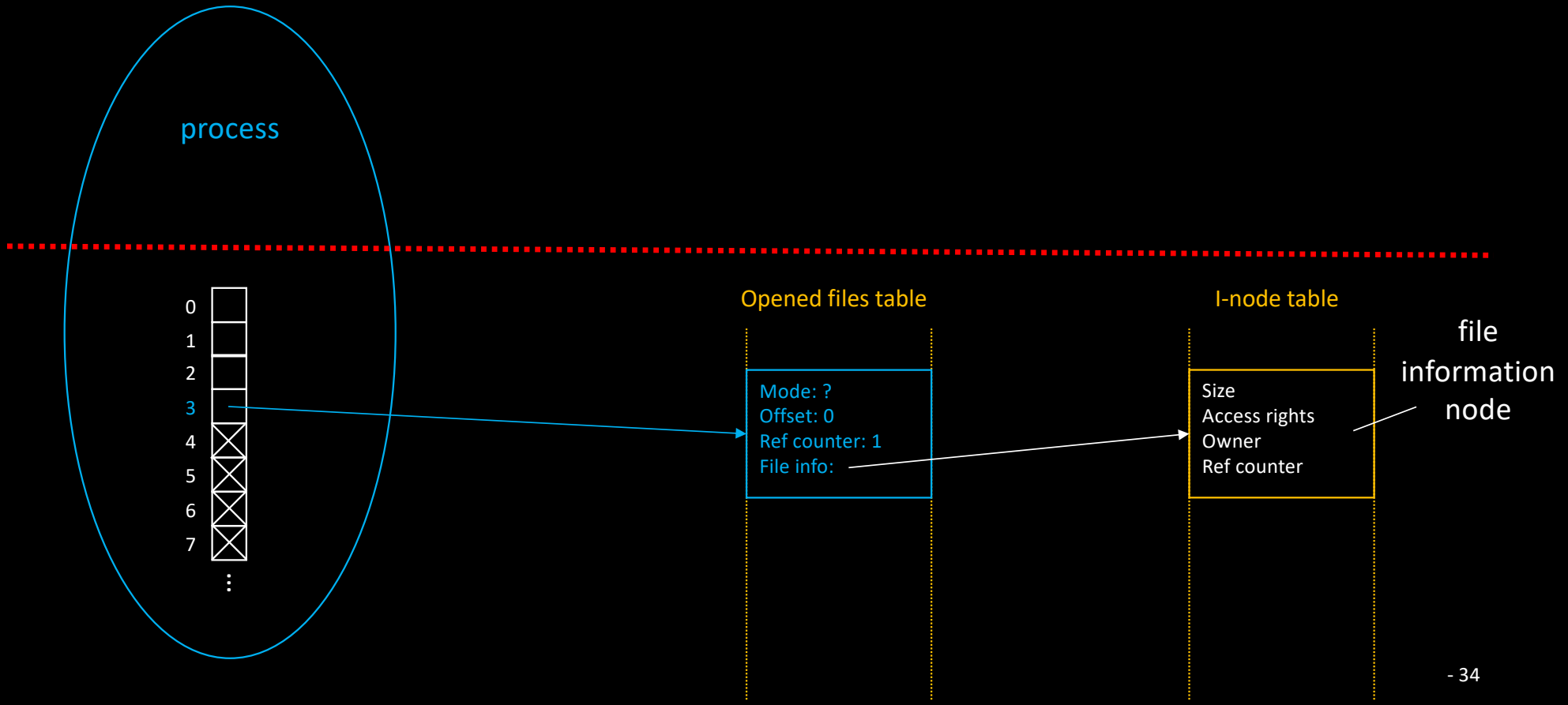
What happens when opening the same file multiple times?



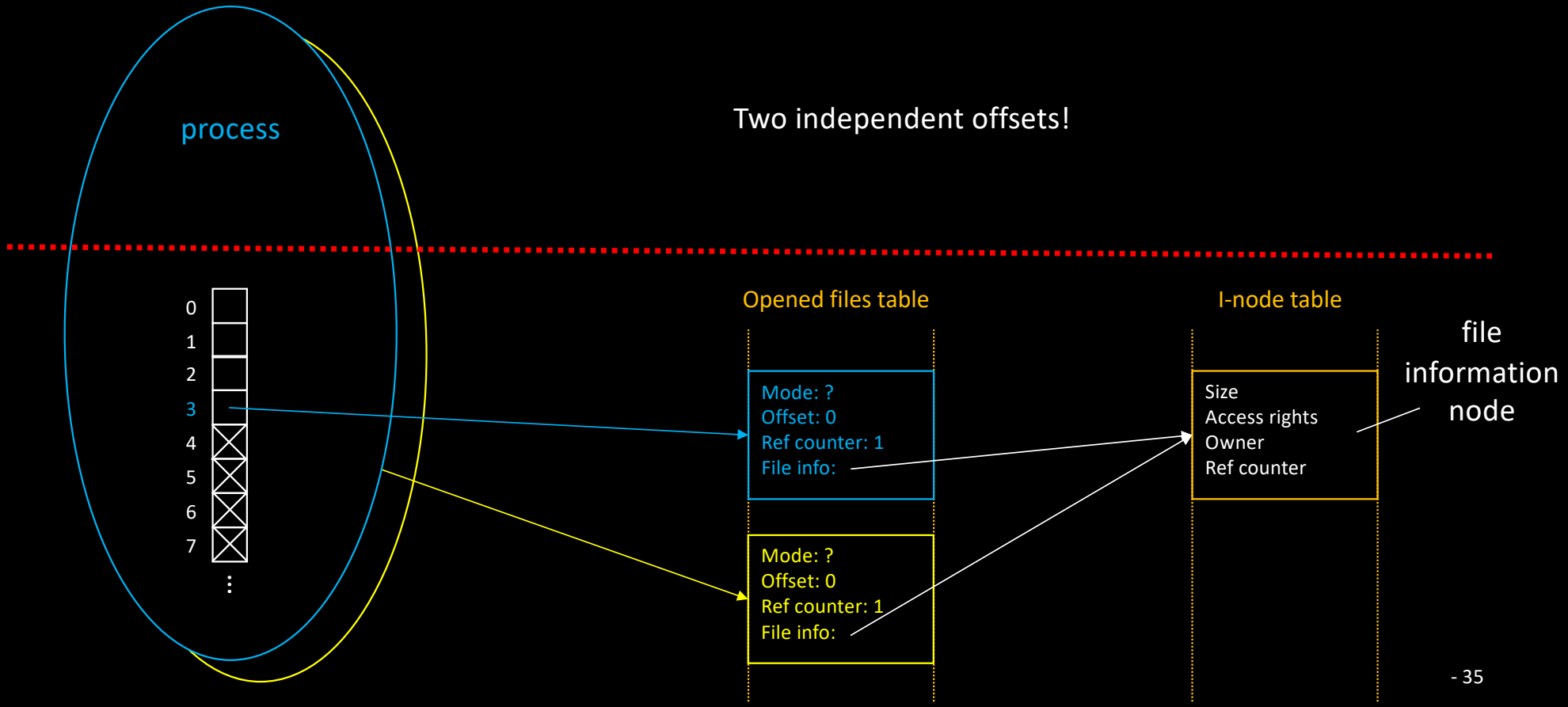
What happens when opening the same file multiple times?



What happens when different processes open the same file?

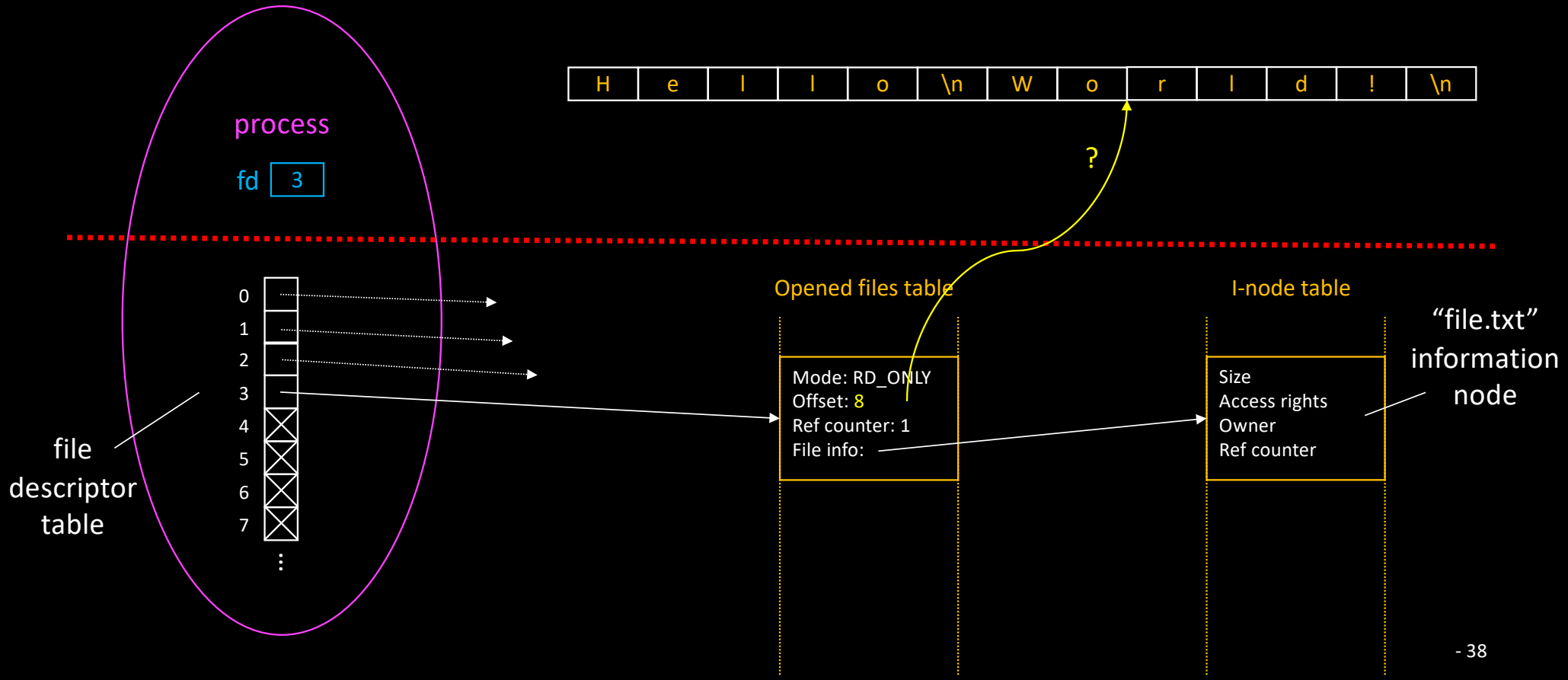


What happens when different processes open the same file?



Random access to files

What if we quickly need to jump to an arbitrary position?



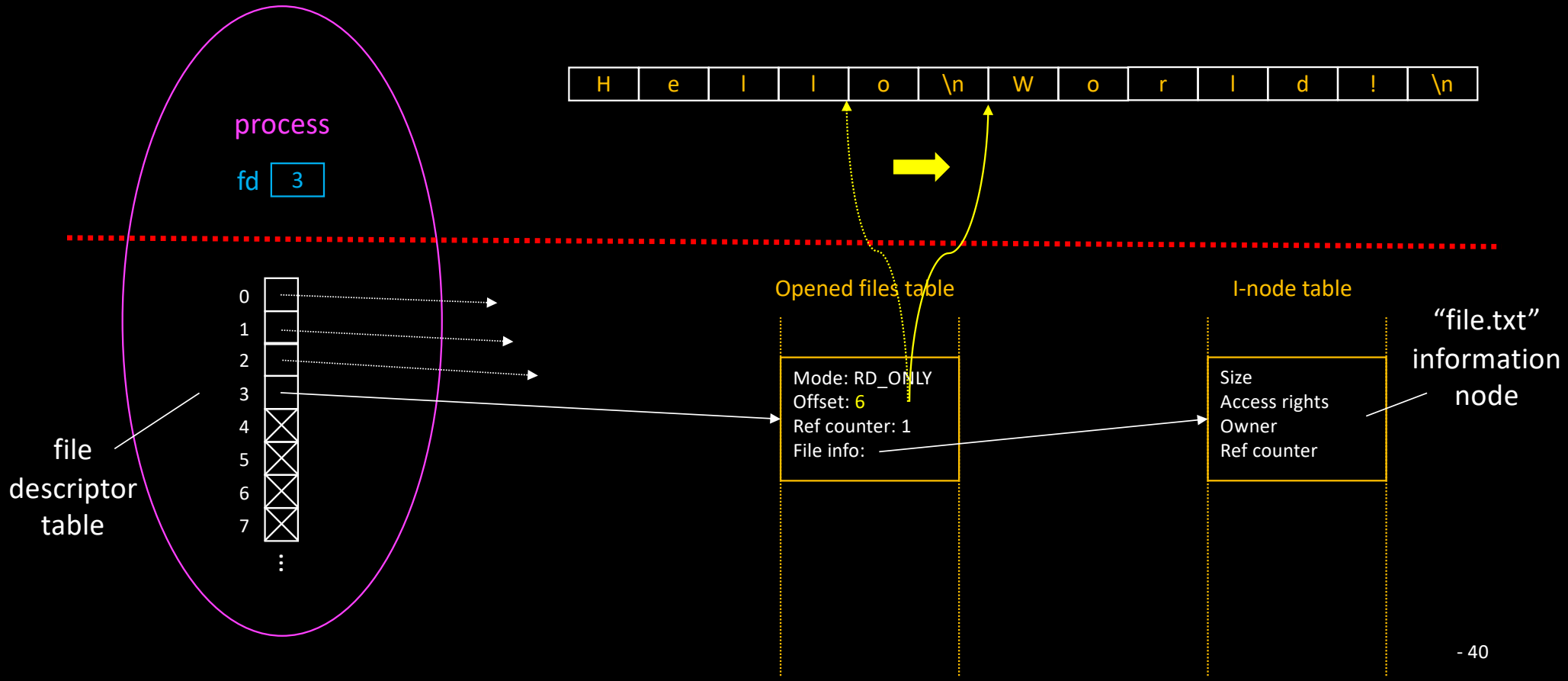
Changing current position

- Reading from the file and writing to the process memory

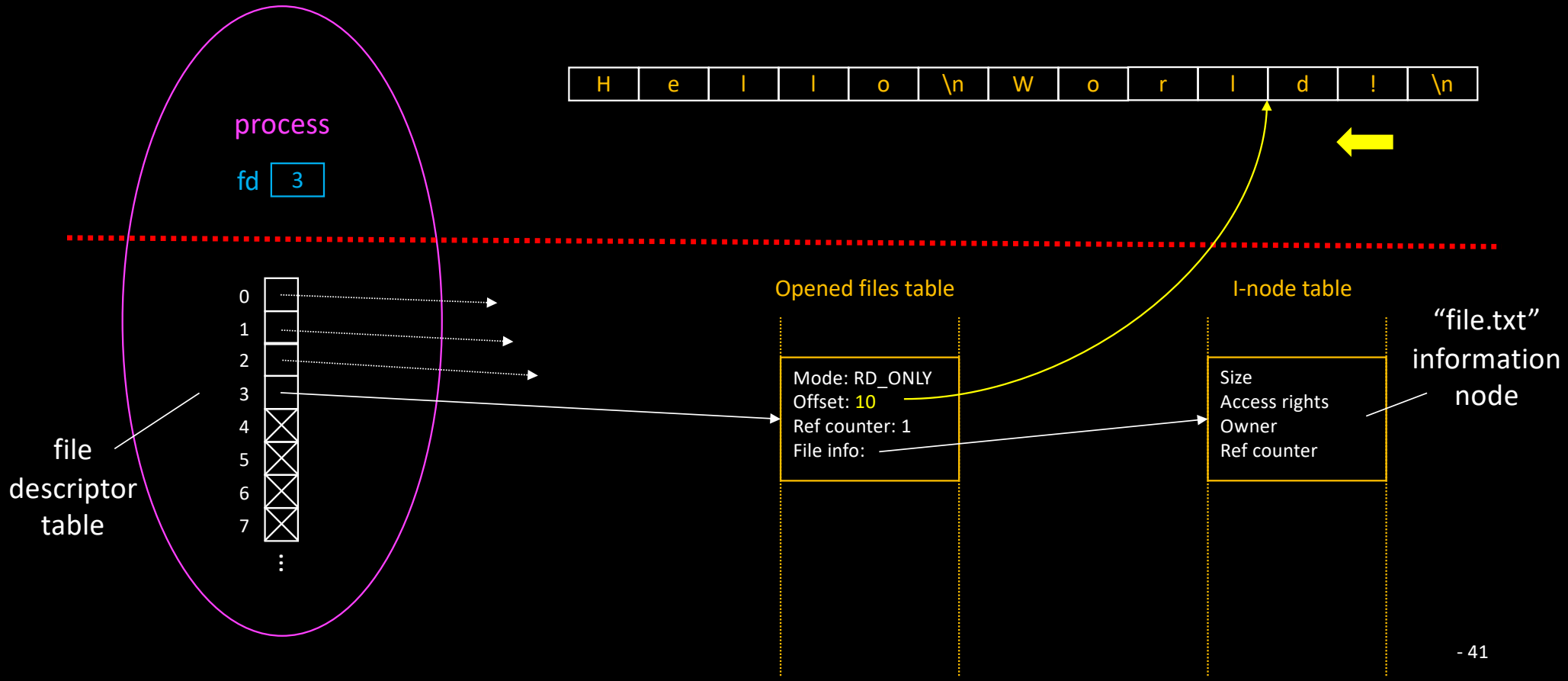
```
off_t lseek(int fildes, off_t offset, int whence);
```

  - `whence` can be:
    - SEEK_SET
    - SEEK_CUR
    - SEEK_END
  - Return value:
    - Absolute offset
      - Cannot be negative

lseek (fd, 2, SEEK_CUR)



lseek (fd, -3, SEEK_END)

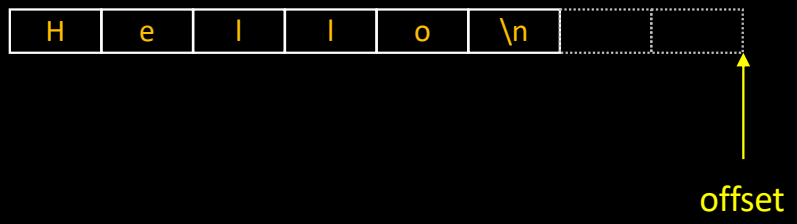


lseek

- seek.c, reverse.c

Changing current position

- Setting offset beyond end-of-file is possible
  - Remember : lseek performs no file access
- What happens upon read?
- What happens upon write?



lseek

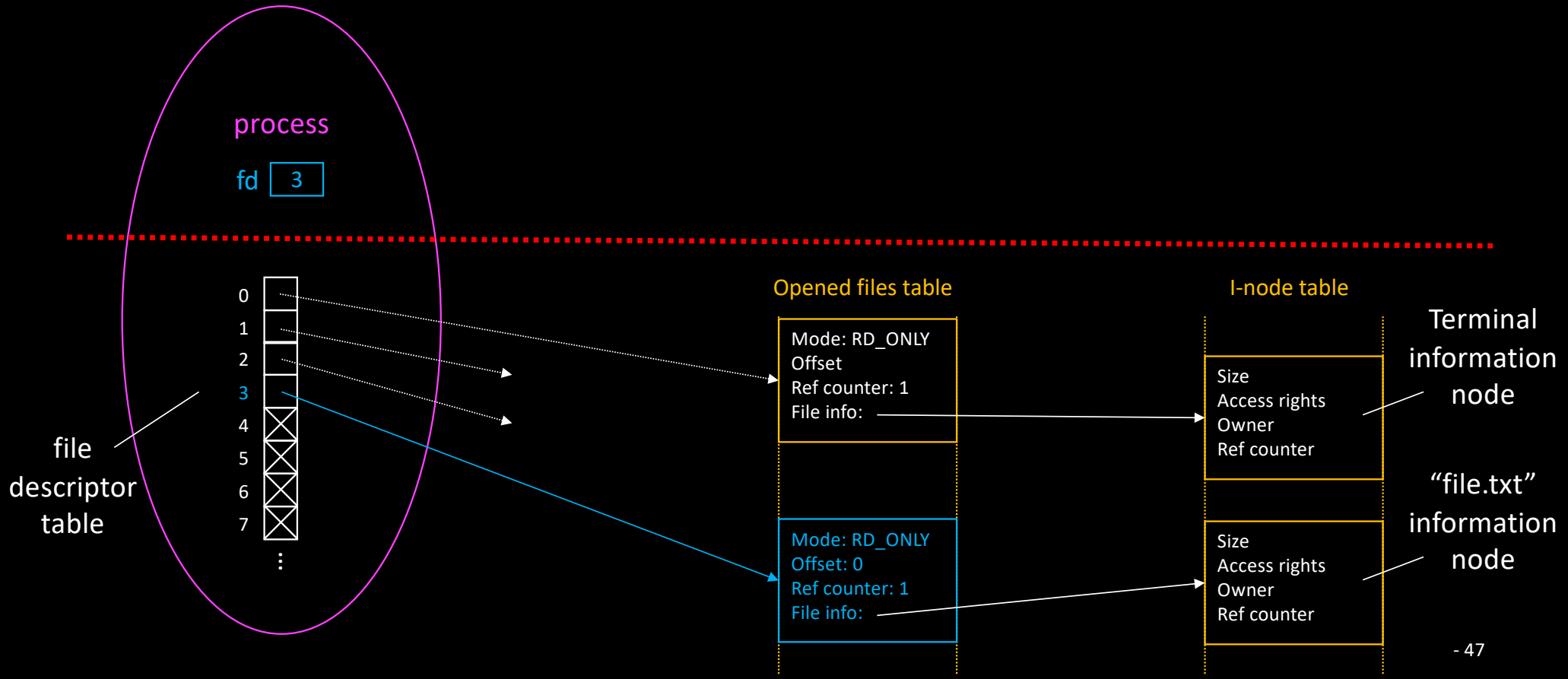
- create_big_file.c

I/O redirections

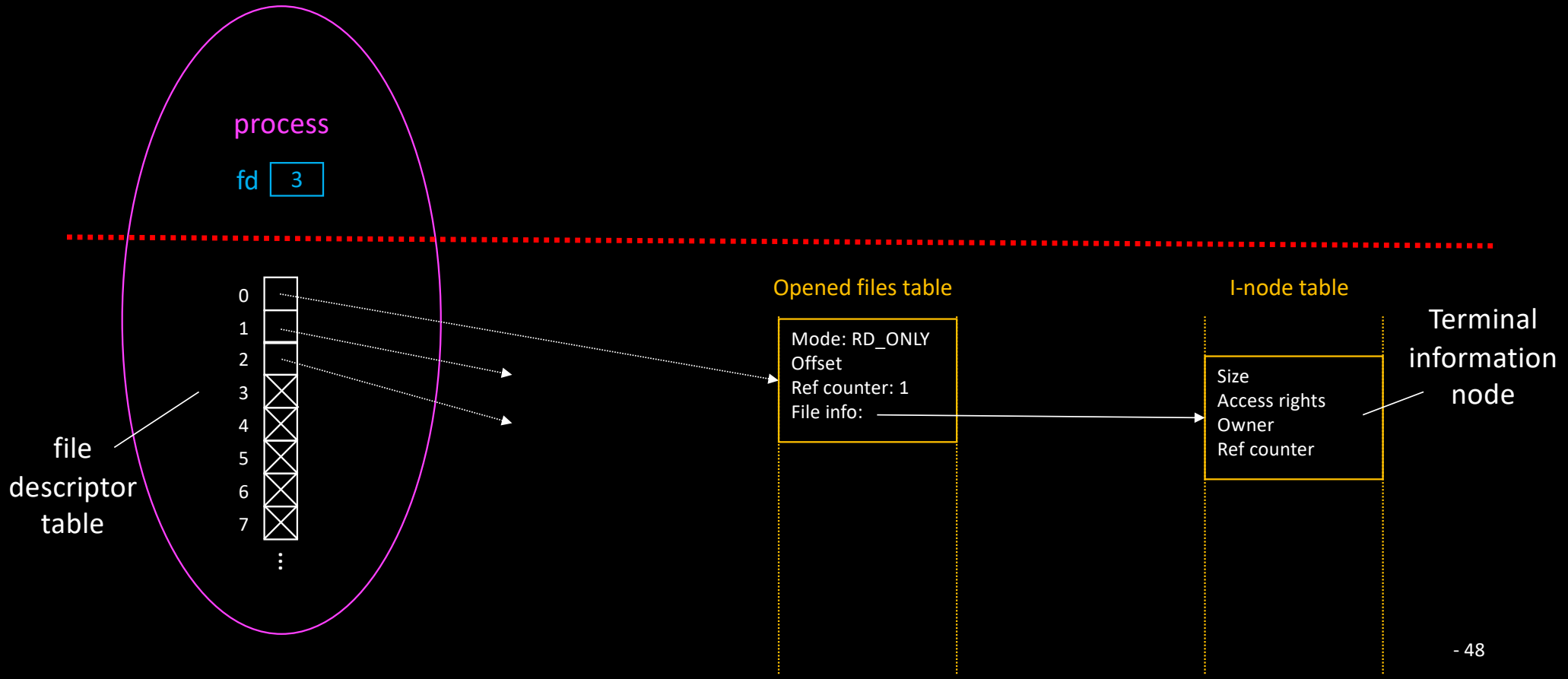
Back to pre-opened descriptors…

```
int main (int argc, char *argv[])
{
  close (STDIN_FILENO);
  int fd = open (“file.txt”, O_RDONLY);
  …
  read (STDIN_FILENO, …);  // What happens?
  …
}
```

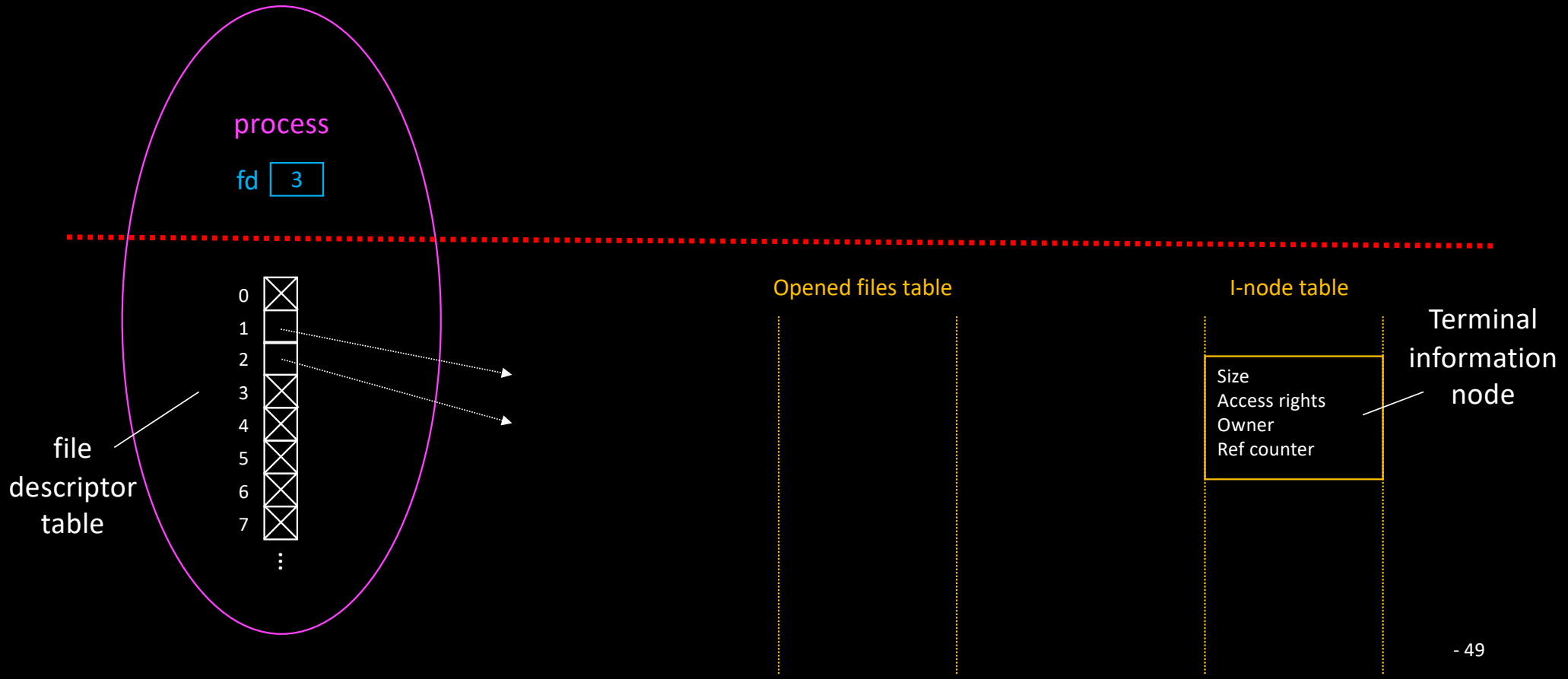
fd = open ("file.txt", O_RDONLY);



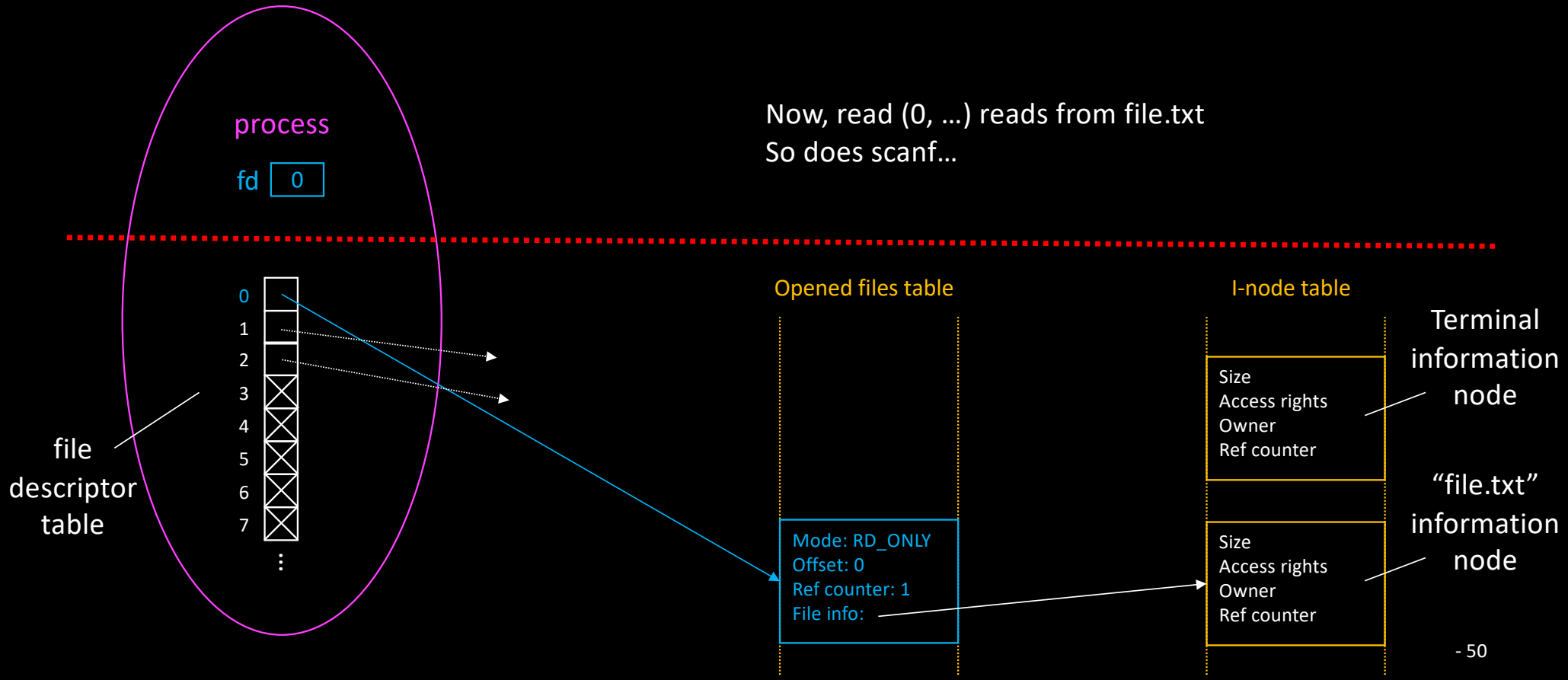
At launch time



close (0);


process
fd
3
0
1
2
3
4
5
6
7
file descriptor table
Opened files table
I-node table
Size
Access rights
Owner
Ref counter
Terminal information node

close (0); fd = open ("file.txt", O_RDONLY);



Back to pre-opened descriptors…

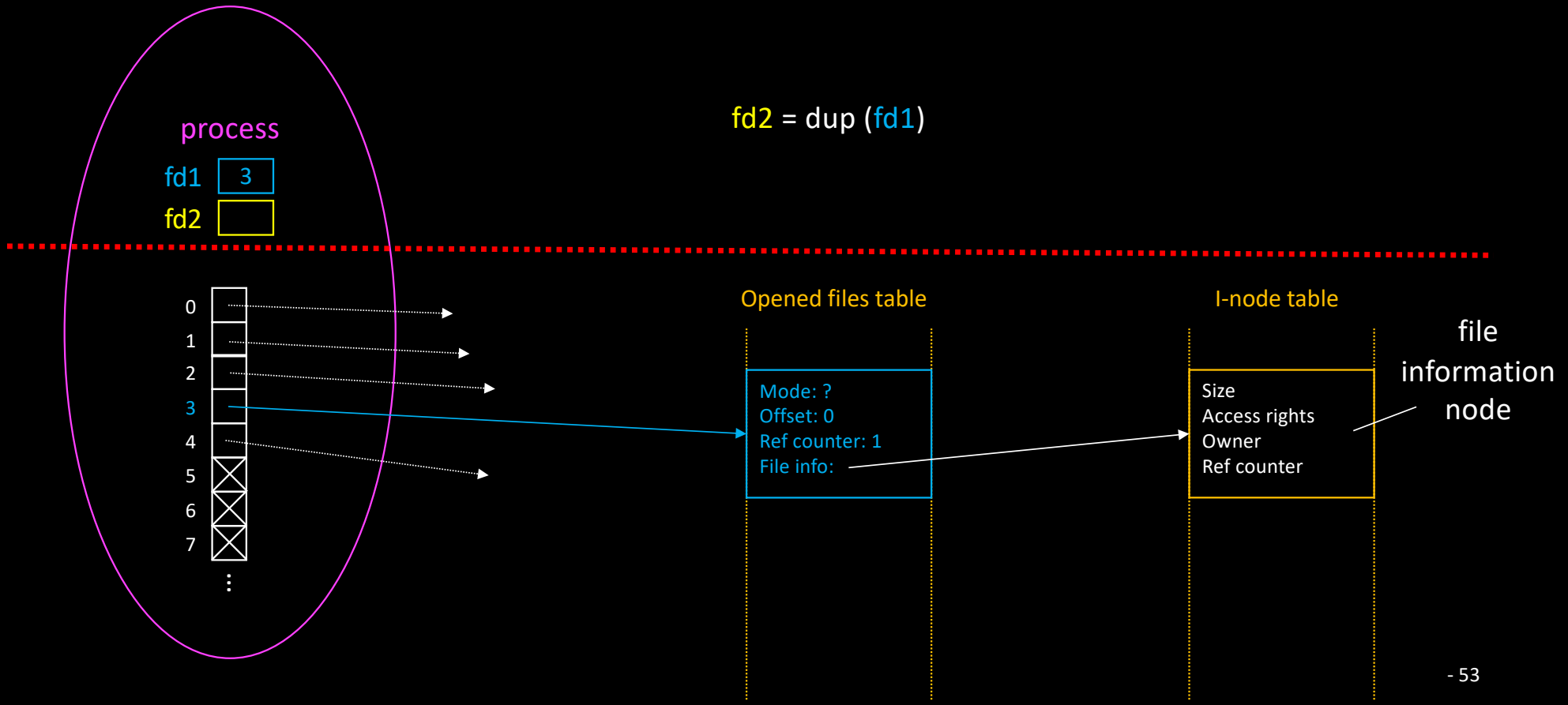
```
int main (int argc, char *argv[])
{
  close (STDIN_FILENO);
  int fd = open (“file.txt”, O_RDONLY);
  …
  // From now on, standard input is redirected to file.txt
  // ./prog < file.txt
  …
}
```

File descriptor manipulation

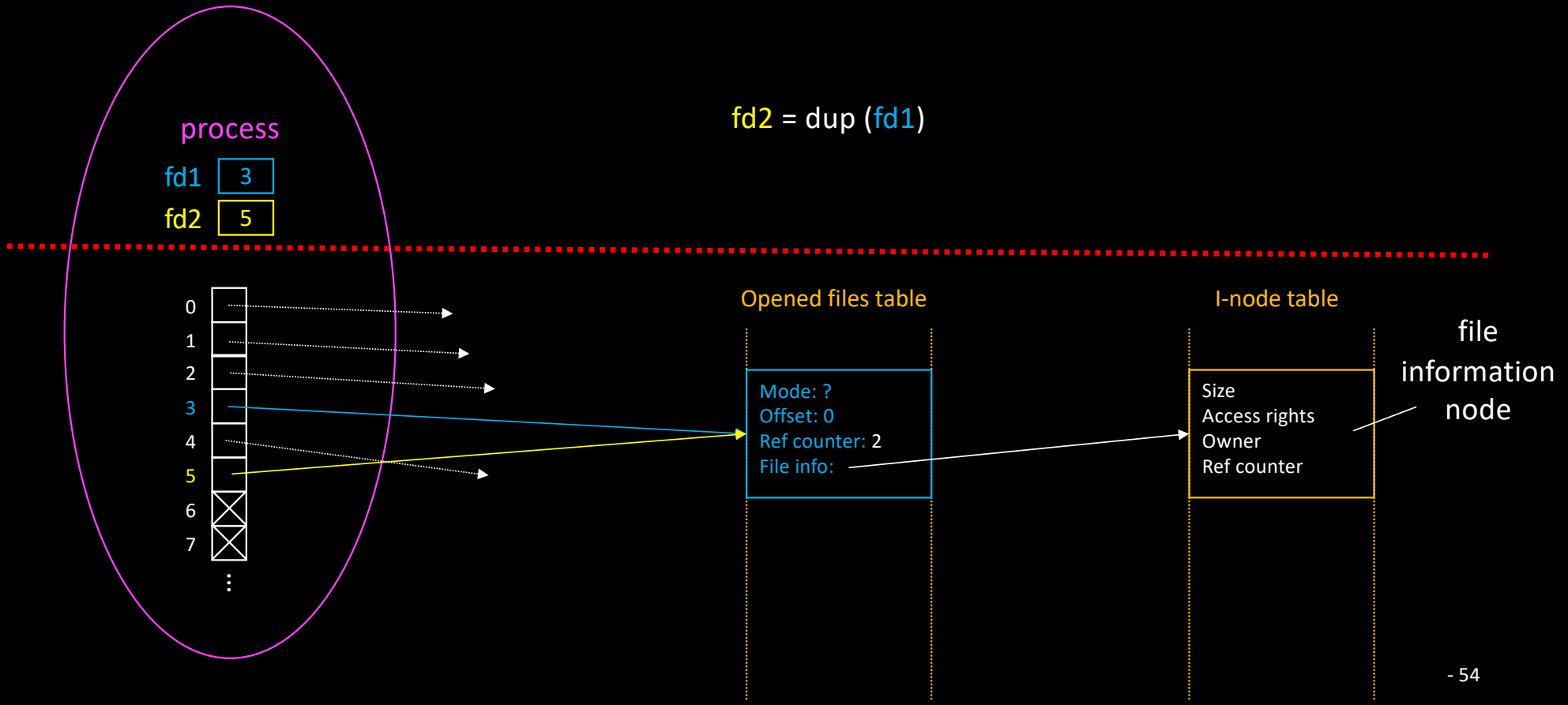
- Duplicating file descriptors
  - i.e. duplicating pointers in the file descriptor table

```
int dup (int fildes);

int dup2 (int src_fd, int dst_fd);
```

File descriptor manipulation



File descriptor manipulation

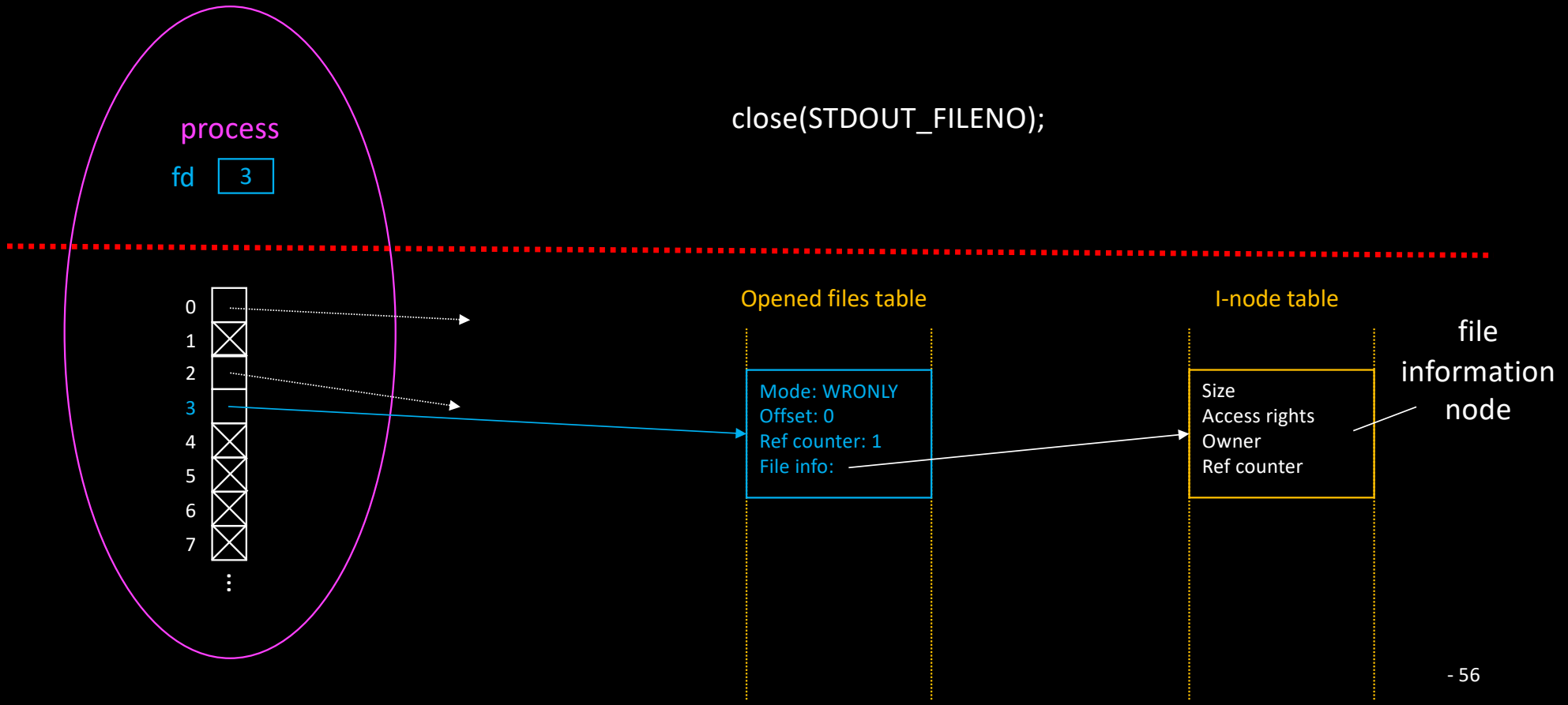


File descriptor manipulation

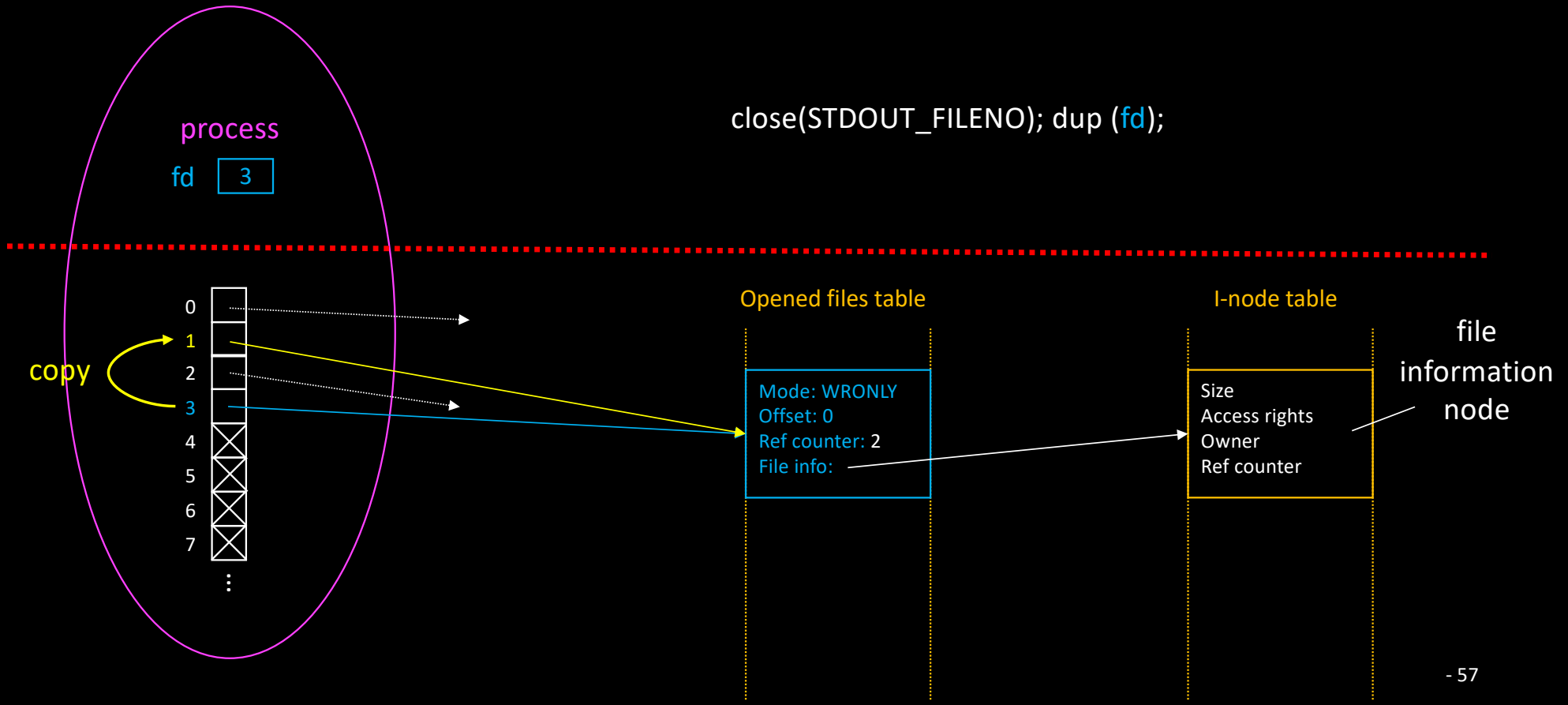
- Typical usage of dup

```
int fd = open (…);
…
close (STDOUT_FILENO);
dup (fd);
close (fd);
```

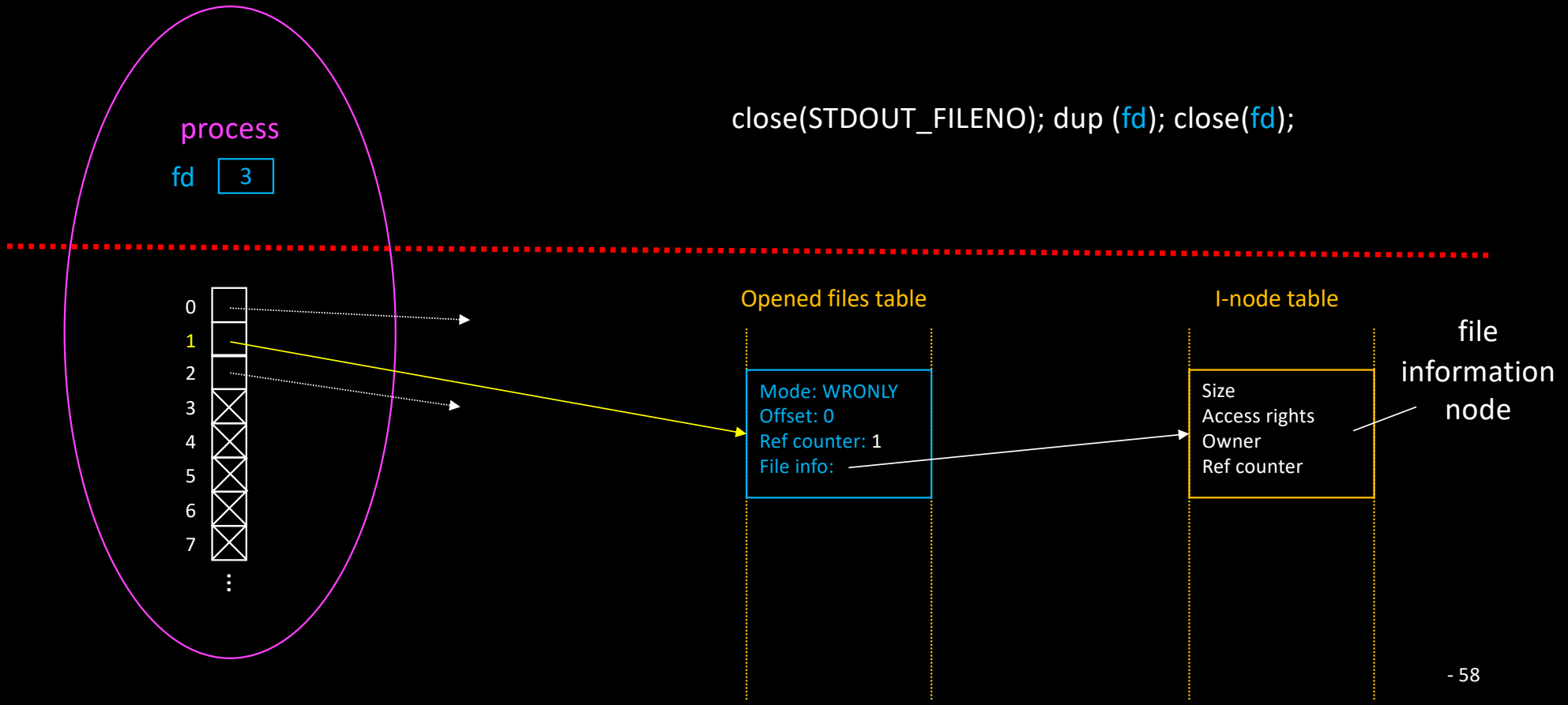
STDOUT redirection (with dup)



STDOUT redirection (with dup)



STDOUT redirection (with dup)



File descriptor manipulation

- Typical usage of dup

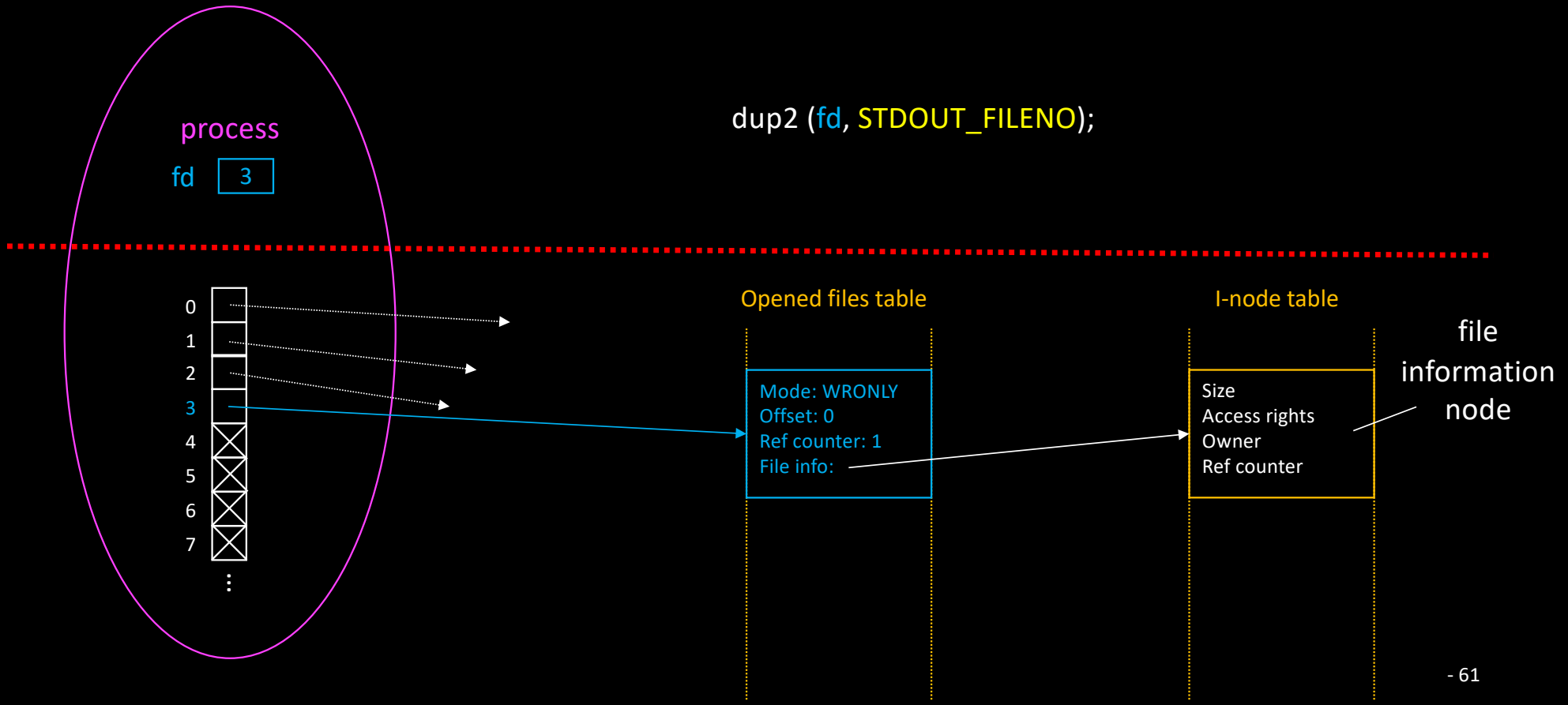
```
int fd = open (…);

… // what if STDIN_FILENO is closed here?

close (STDOUT_FILENO);
dup (fd);
close (fd);
```

File descriptor manipulation

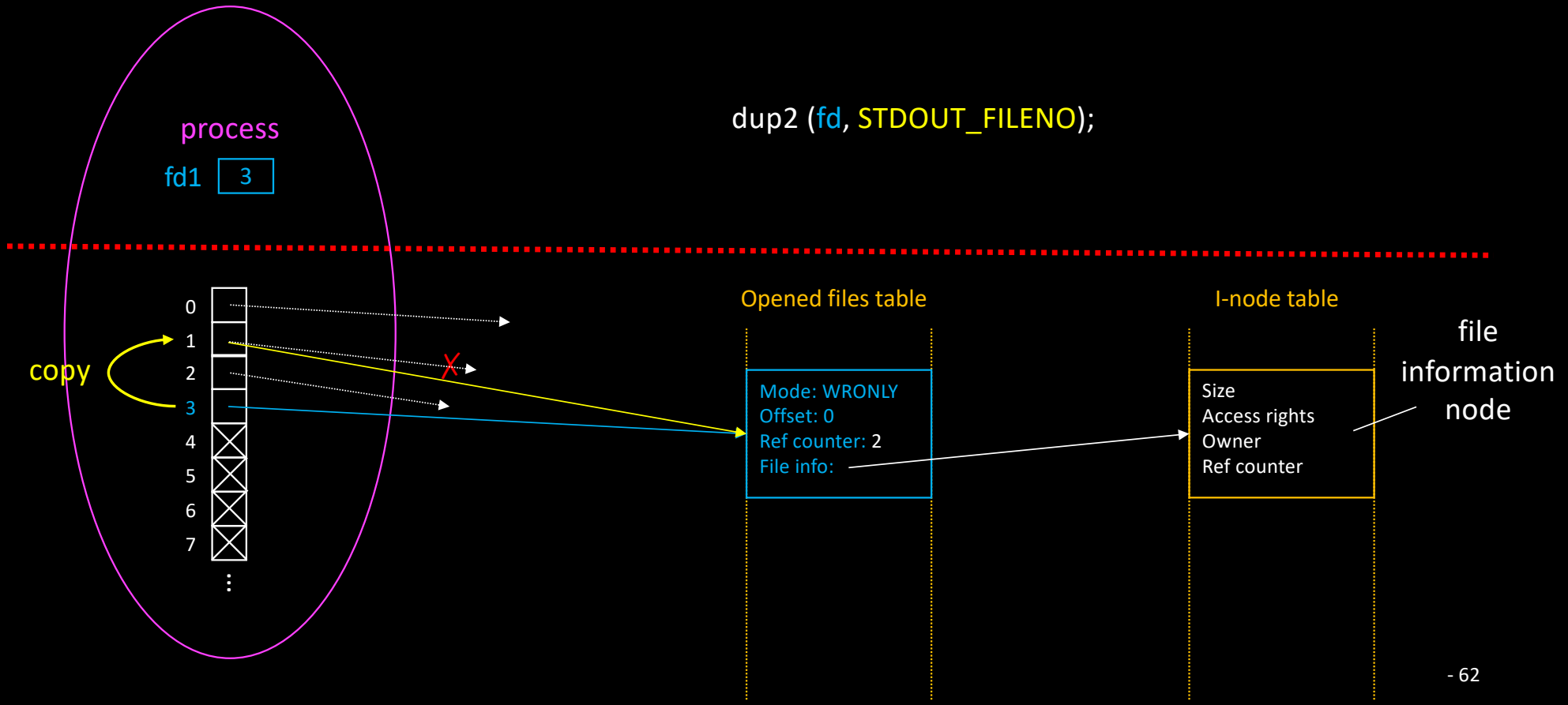
- We need a safer way to select the target descriptor

```
int fd = open (…);
…
dup2 (fd, STDOUT_FILENO);
// STDOUT is automatically closed
// before fd is copied
close (fd);
```

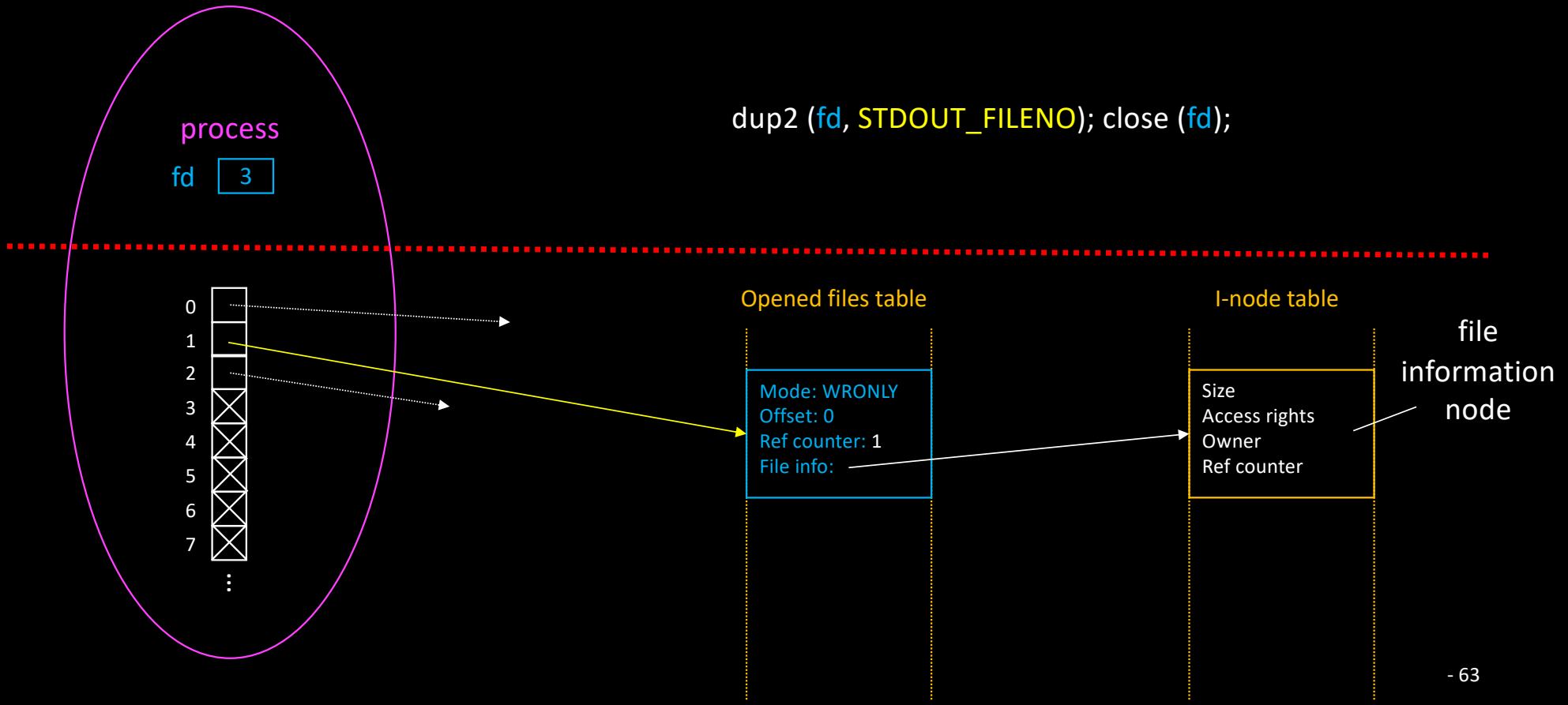
STDOUT redirection (with dup2)



STDOUT redirection (with dup2)



STDOUT redirection (with dup2)

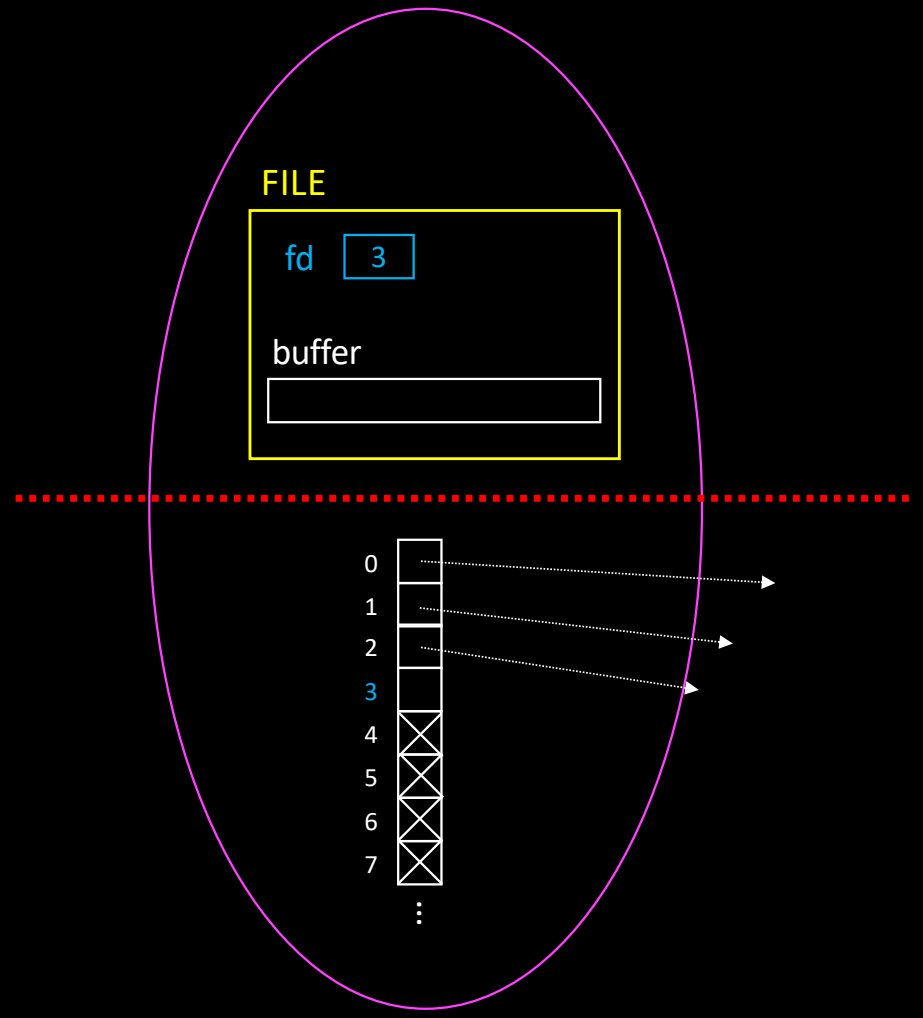


C standard file API

- Files are manipulated through FILE* handlers (≠ file descriptor)
  - FILE* fopen (…)
  - fread (…, FILE *f), fwrite (…, FILE *f), fprintf (FILE *f, …), fscanf (FILE *f, …), …
  - extern FILE *stdin, *stdout, *stderr;
- Implemented in user mode (libc)
  - fopen relies on open
  - fread relies on read
  - Etc.

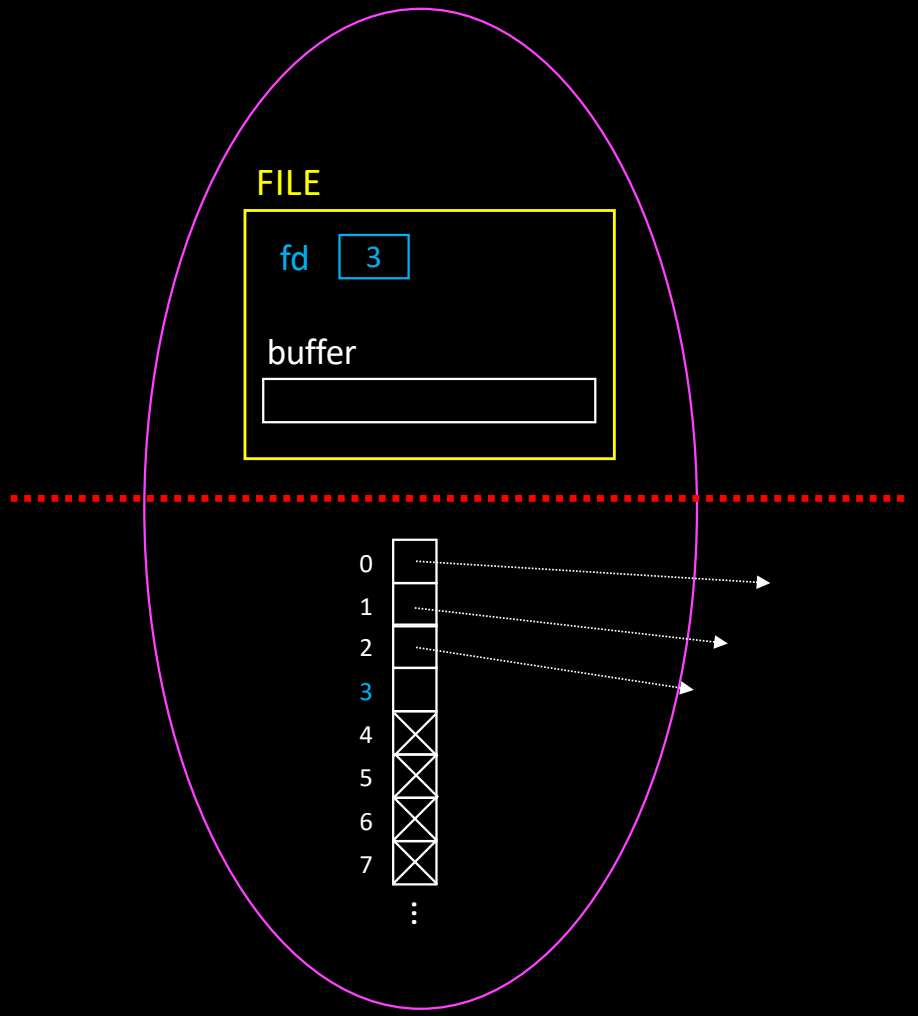
C standard file API

- One could think that these routines introduce overhead
  - But libc routines are usually (much) faster!
  - Reason?
    - The FILE struct contains a buffer
      - 1KB ~ 8KB
    - Read operations use prefetching
    - Write operation use buffering



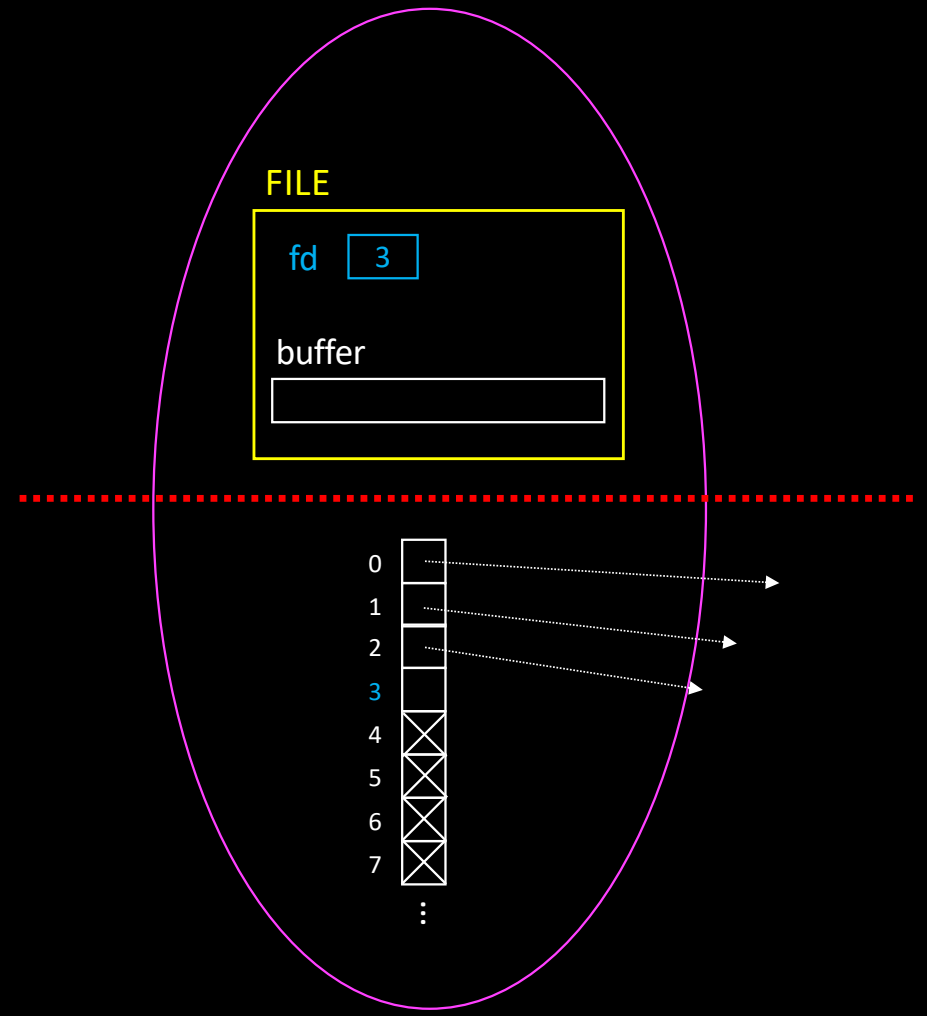
C standard file API

- Read prefetching
  - The first fread prefetches BUFSIZE bytes (if possible) using read
  - Next fread operations simply copy from buffer to destination
- As a result
  - fcopy performs a lot less *system calls* than copy



C standard file API

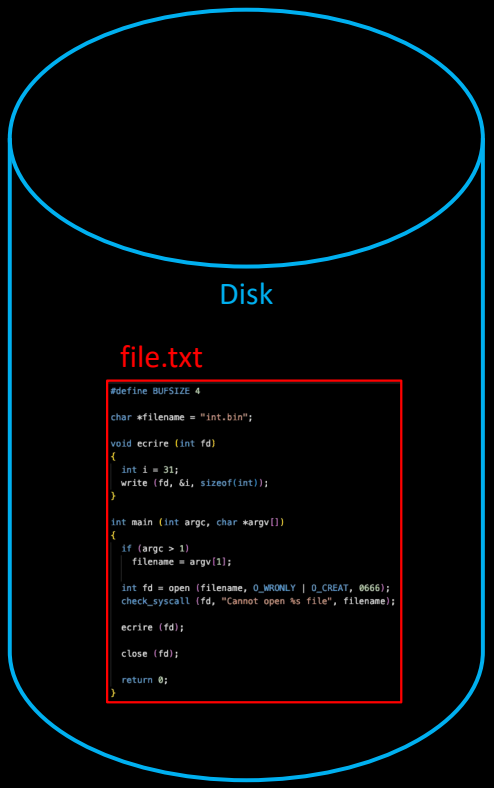
- Write buffering
  - fwrite copies data into buffer
  - When the buffer is full, it gets flushed to the kernel (write)
- Special buffering policies
  - stderr is unbuffered
  - stdout is line buffered
    - printf (“hello”) vs printf (“hello\n”)
    - fflush (FILE *f)




Allez sur wooclap.com et utilisez le code PSFOREVER
printf("Hello\nWorld!\n"); provoque
Résultats
1
Aucun appel système : les données sont juste stockées dans un tampon
6%
4
me à write
44%
28
3
Deux appels système à write
33%
21
4
printf est un appel système au même titre que write, donc cela ne provoque pas d'appel supplémentaire
17%
11
wooclap
100 %
64 / 192

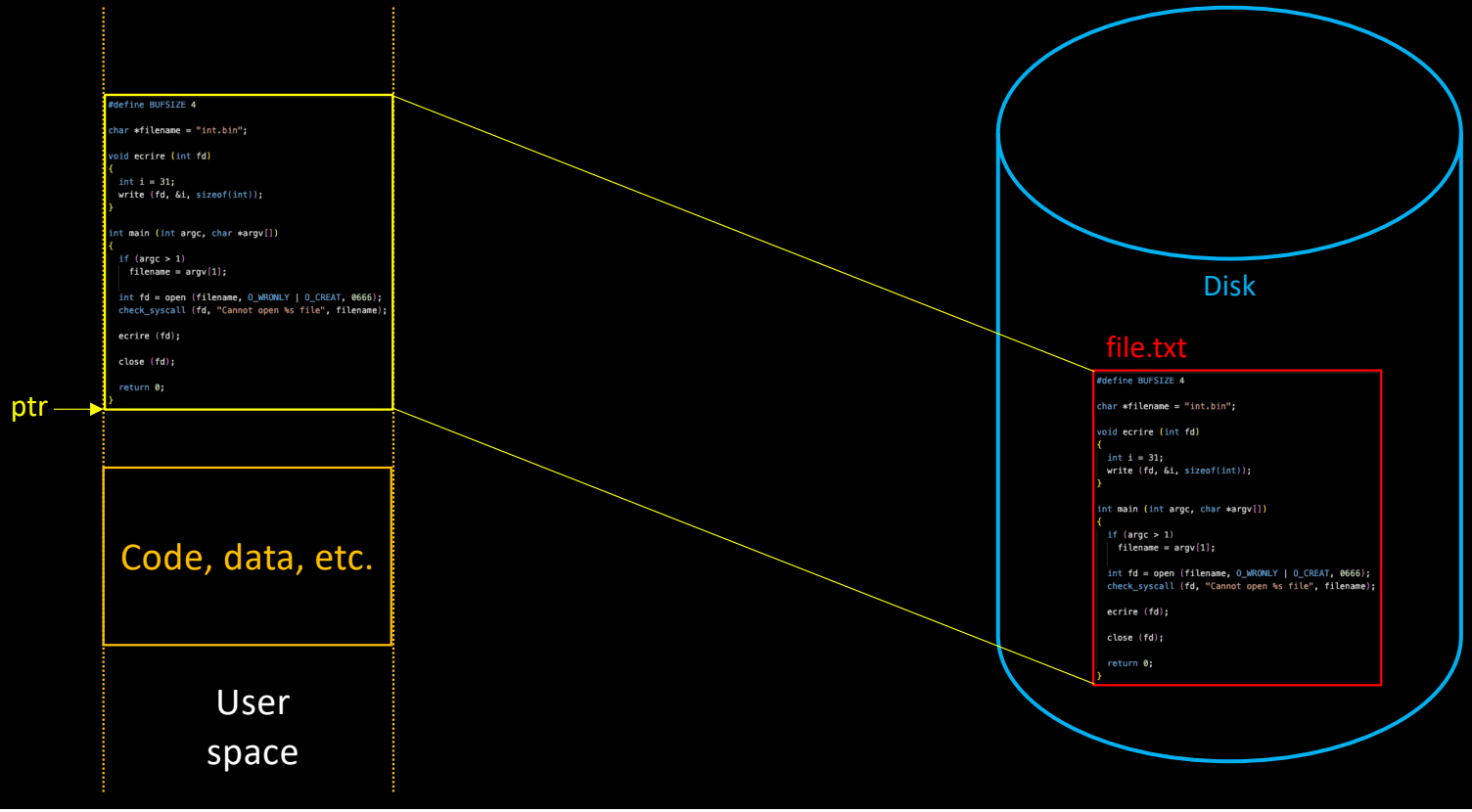
Memory-mapped files

- Accessing files as if they were in memory ?



Memory-mapped files

- Accessing files as if they were in memory ? It is called: *file mapping*



Memory-mapped files

- Accessing files as if they were in memory ? It's called *file mapping*
  - Processes can map a (portion of a) file in their address space

```
void *mmap(void *addr, size_t len, int prot, int flags, int fd, off_t offset);
```

Suggested address (or NULL)

access mode

file

  - Example:

```
char *addr = mmap (NULL, len, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
```

Memory-mapped files

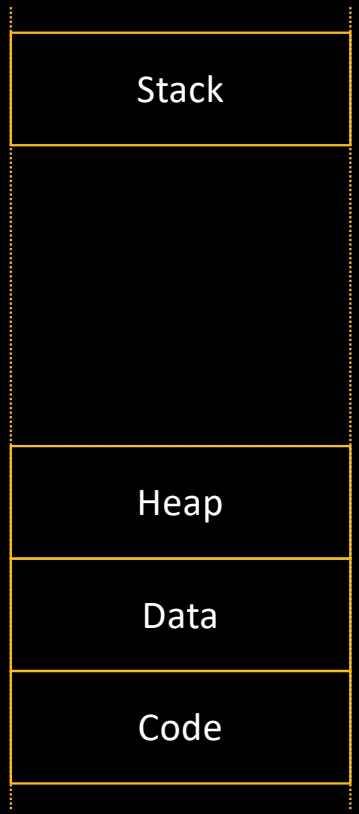
- Accessing files as if they were in memory ? It's called *file mapping*
  - Processes can map a (portion of a) file in their address space

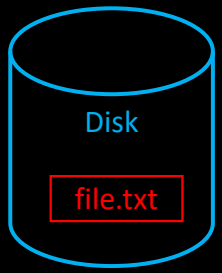
```
int fd = open ("file.txt", O_RDWR);

off_t len = lseek (fd, 0, SEEK_END);

char *region = mmap (NULL, len, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
```

  - Then, file contents can be accessed as an array

```
strcpy (region, "Hello World!");
```

mmap

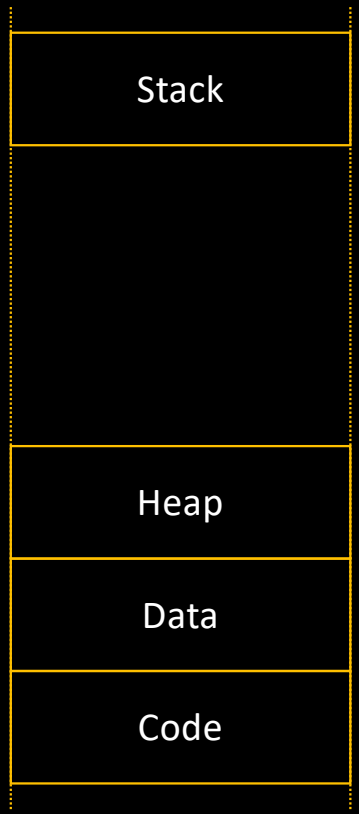


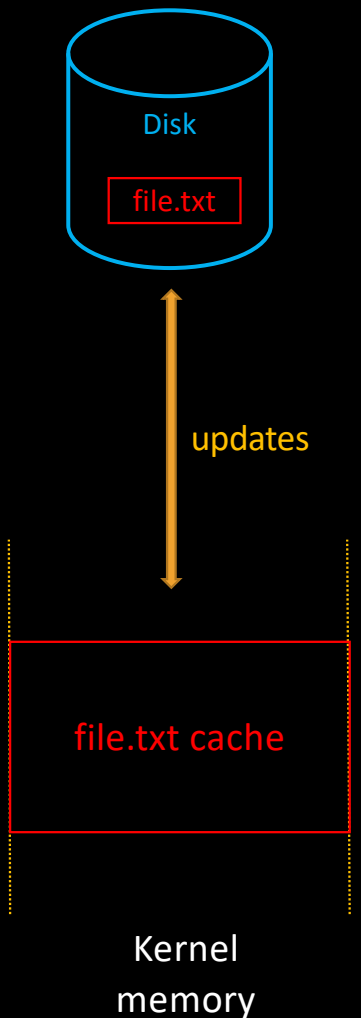


Kernel memory

- mmap takes a file descriptor as a parameter

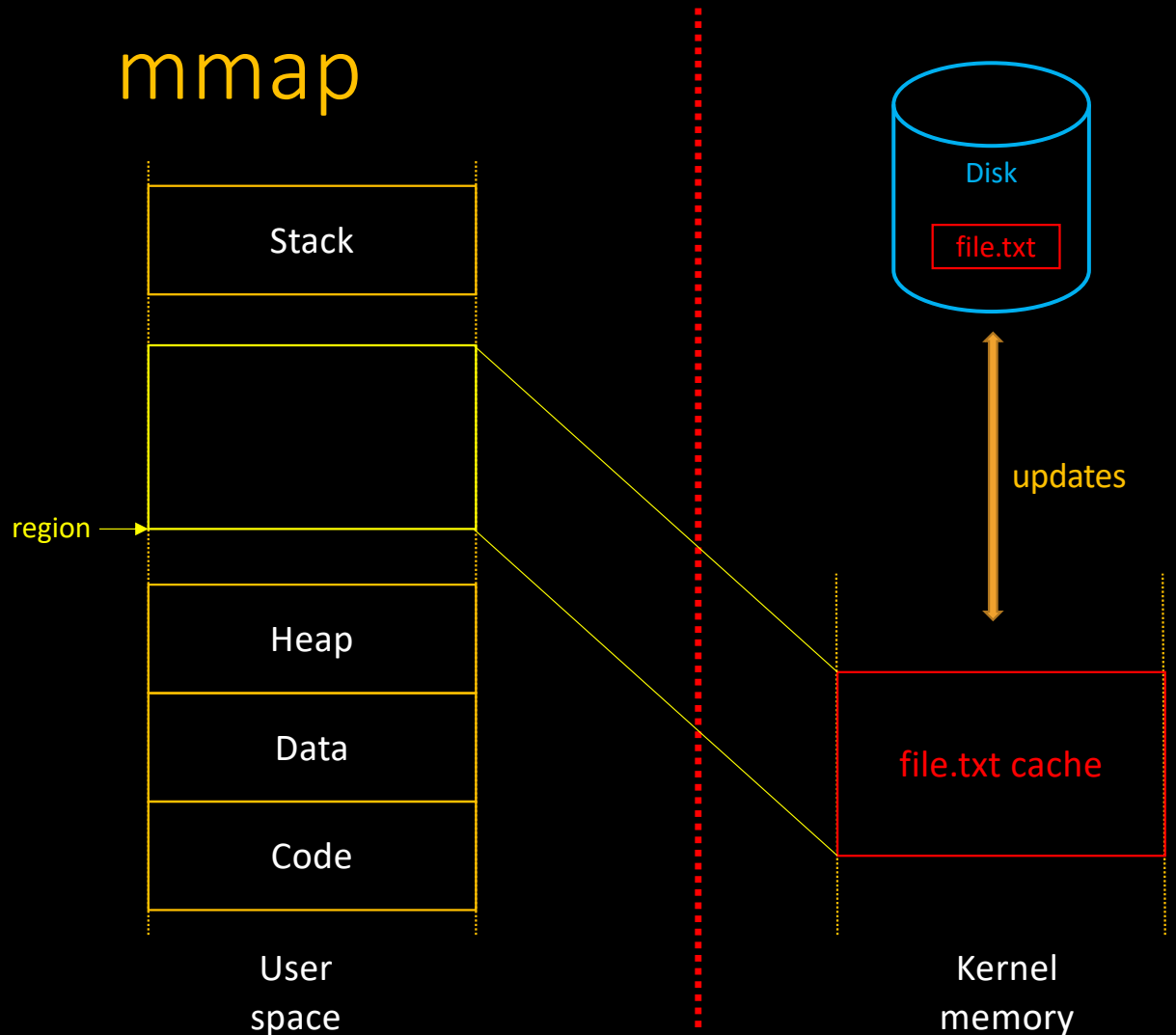
mmap





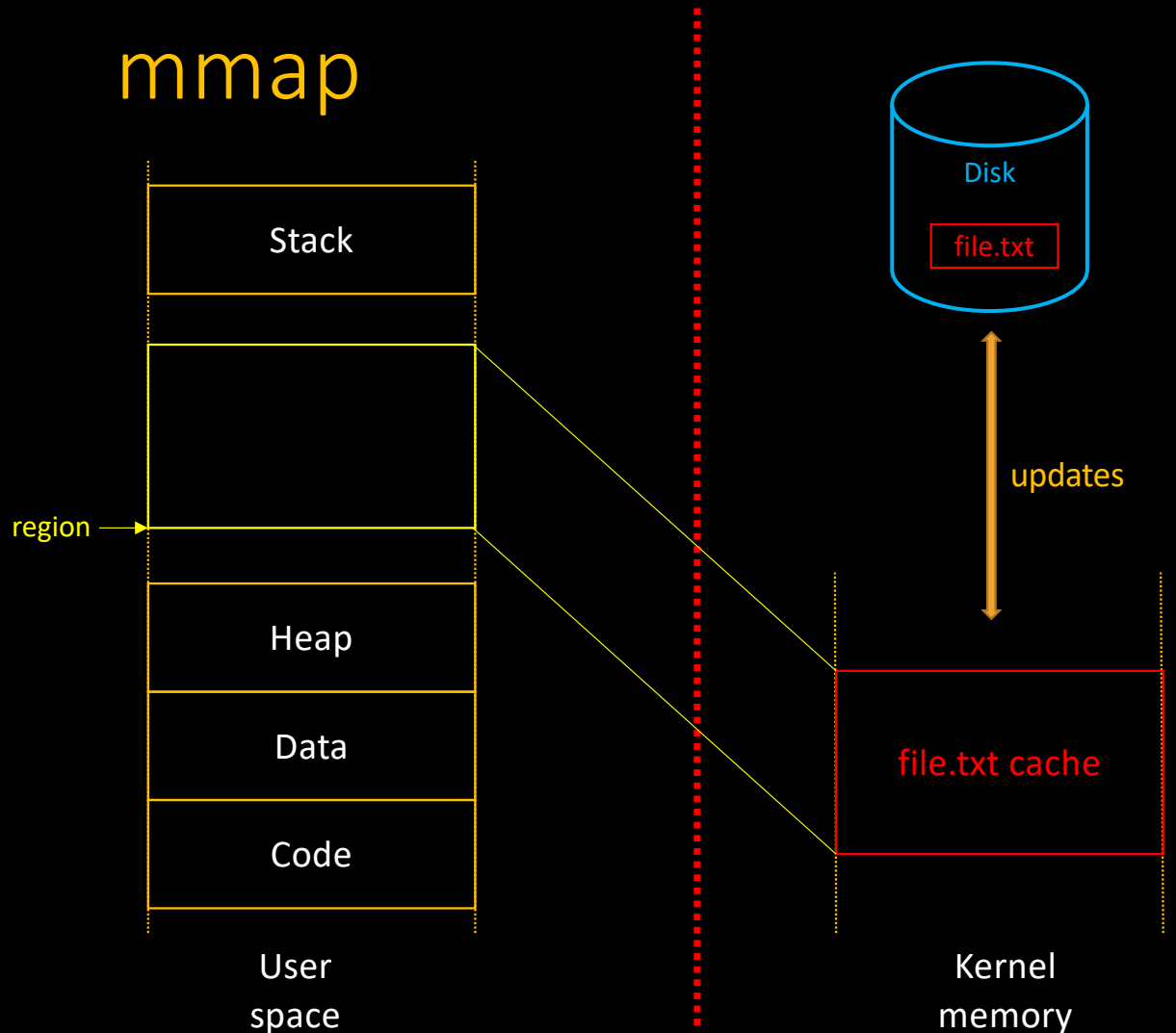
- (Part of) file.txt is loaded into kernel memory
  - Acts as a cache:
    - All read/write operations go through the cache
  - Disk file will be automatically updated…

mmap

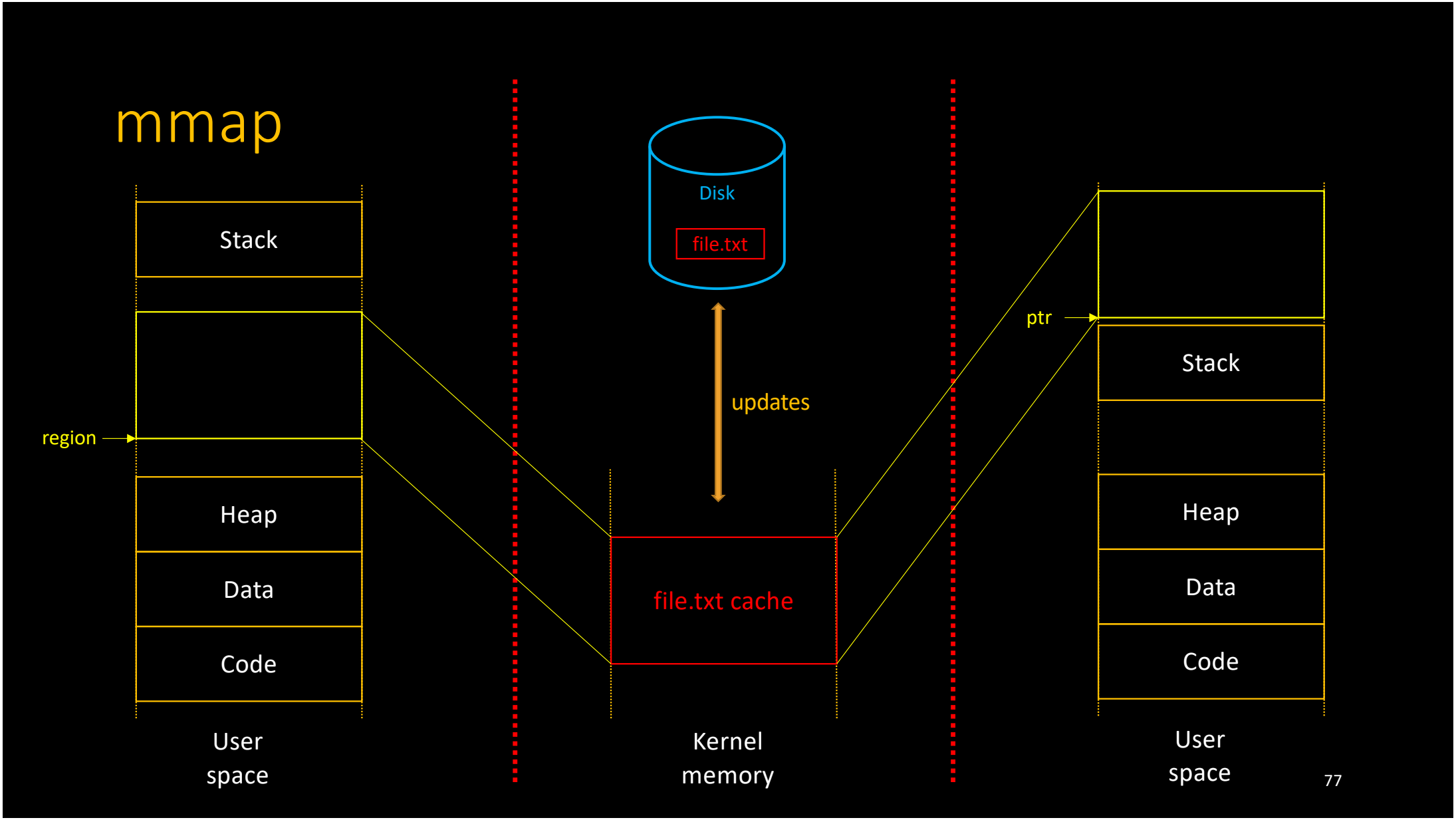


- Kernel cache is mapped directly in process' address space
  - No memory allocation
    - Purely virtual
  - CPU "move" operations directly operate on the kernel cache

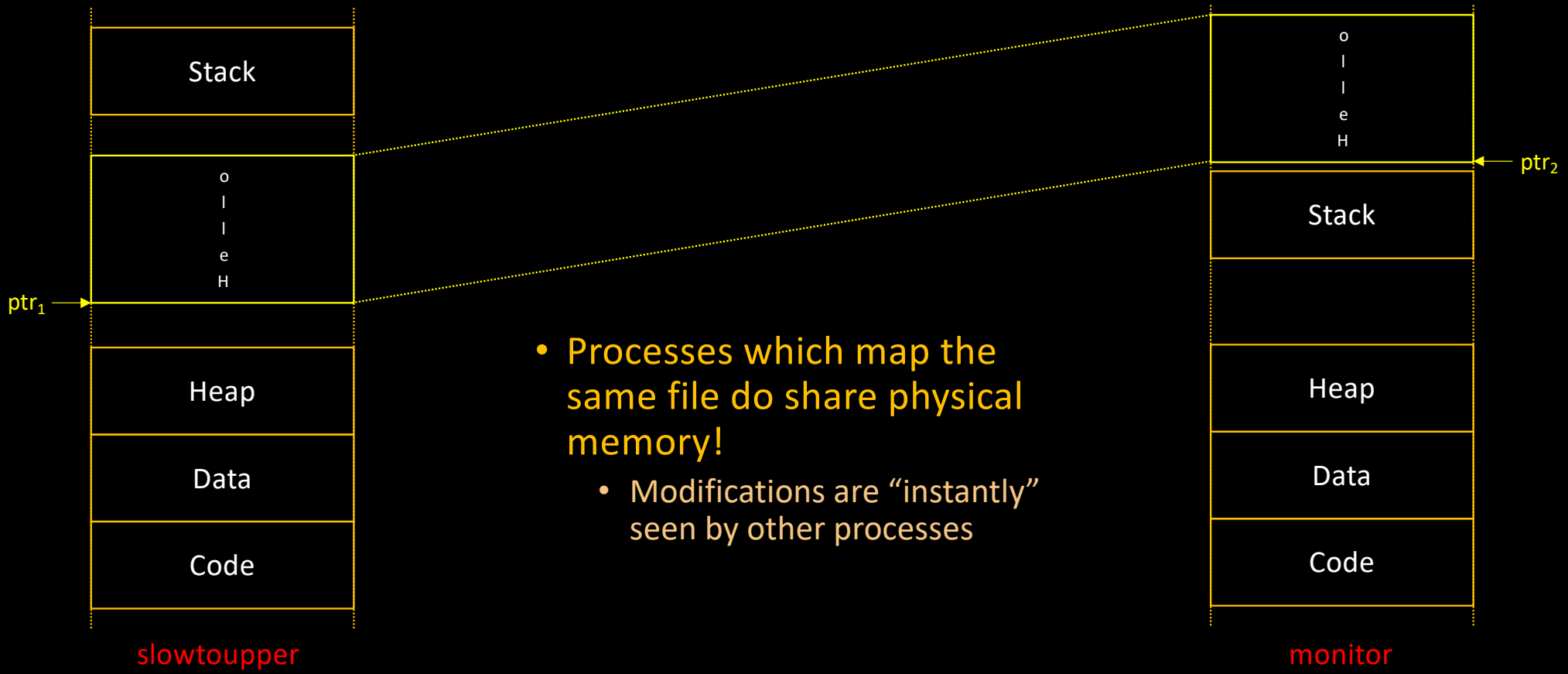
mmap



- In fact, *file mapping* is the most efficient way to cope with non-growing files
  - Direct memory access
  - No system calls!
  - See reverse.c, toupper.c

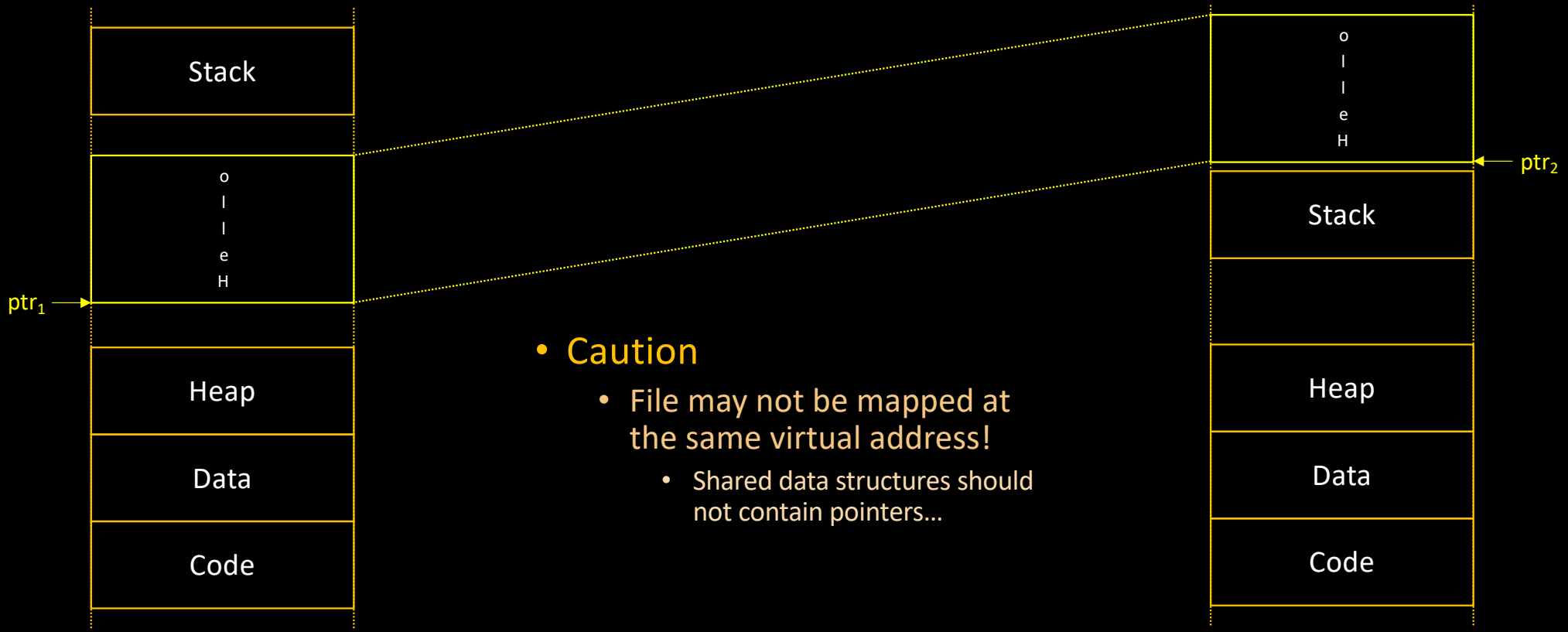

mmap
Disk
file.txt
Stack
region
updates
ptr
Stack
Heap
Data
Code
file.txt cache
Heap
Data
Code
User space
Kernel memory
User space

Communication between processes



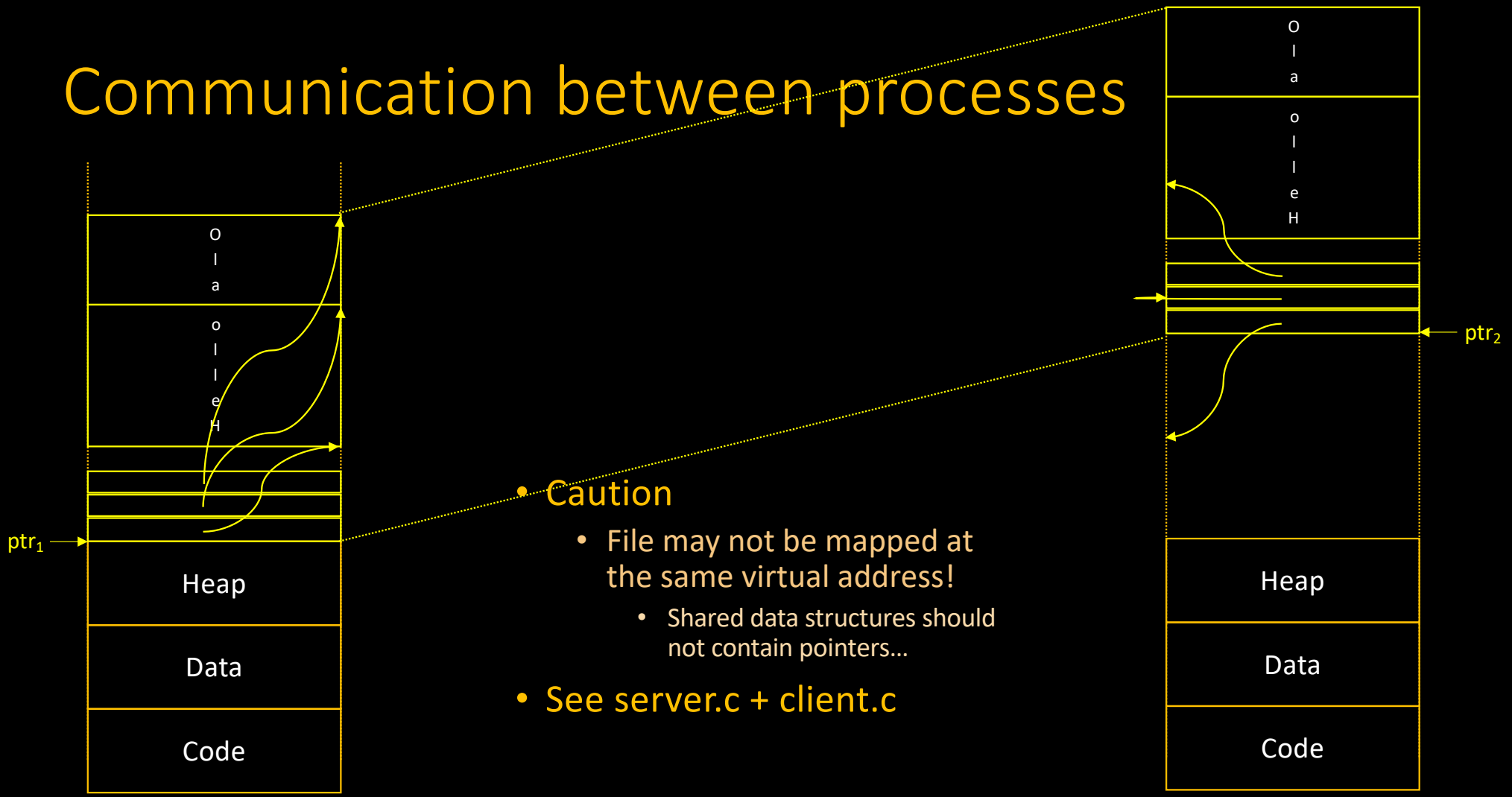
- Processes which map the same file do share physical memory!
  - Modifications are “instantly” seen by other processes

Communication between processes



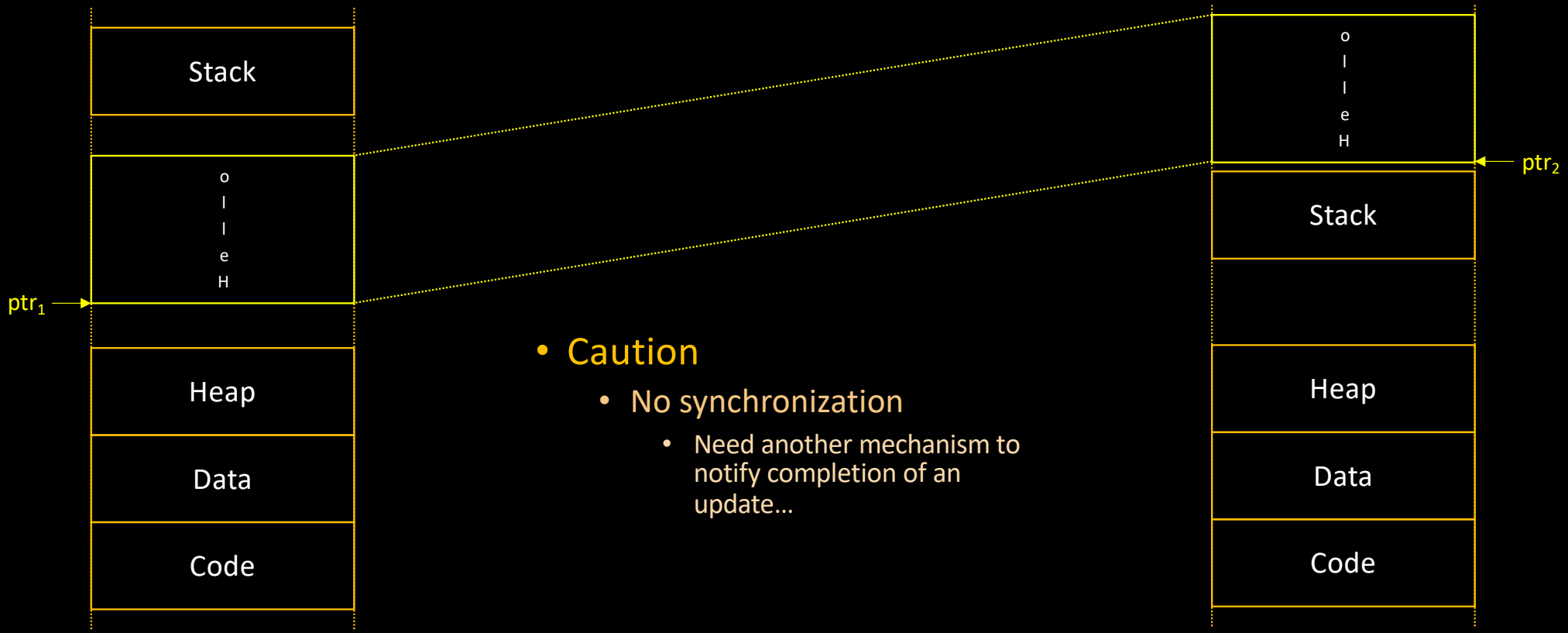
- Caution
  - File may not be mapped at the same virtual address!
    - Shared data structures should not contain pointers…

Communication between processes



- Caution
  - File may not be mapped at the same virtual address!
    - Shared data structures should not contain pointers…
- See server.c + client.c

Communication between processes



- Caution
  - No synchronization
    - Need another mechanism to notify completion of an update…

Additional resources
available on
http://gforgeron.gitlab.io/progsys/