- File Representation (File descriptor, File pointer)
- Redirection
- Inheritance

File Representation

Files are designated within C programs either by file pointers or by file descriptors.

I/O library functions	**UNIX I/O functions**
POSIX standard both are Same	
an ordinary function	is a request to the operating system for service
Use file pointers fopen, fscanf, fprintf, fread, fwrite, fclose	Use File descriptor open, read, write, close and ioctl
symbolic names standard input --stdin , standard output -- stdout standard error -- stderr,	symbolic names standard input --STDIN_FILENO , standard output--STDOUT_FILENO standard error -- STDERR_FILENO
defined in stdio.h	defined in unistd.h.
Many library functions (read and write) are, jackets for system calls. That is, they reformat the arguments in the appropriate system-dependent form and then call the underlying system call to perform the actual operation	

File descriptors

The open function associates a file or physical device with the logical handle

The file or physical device is specified by a character string.

The handle is an integer which is an index into a file descriptor table, it is specific to a process.

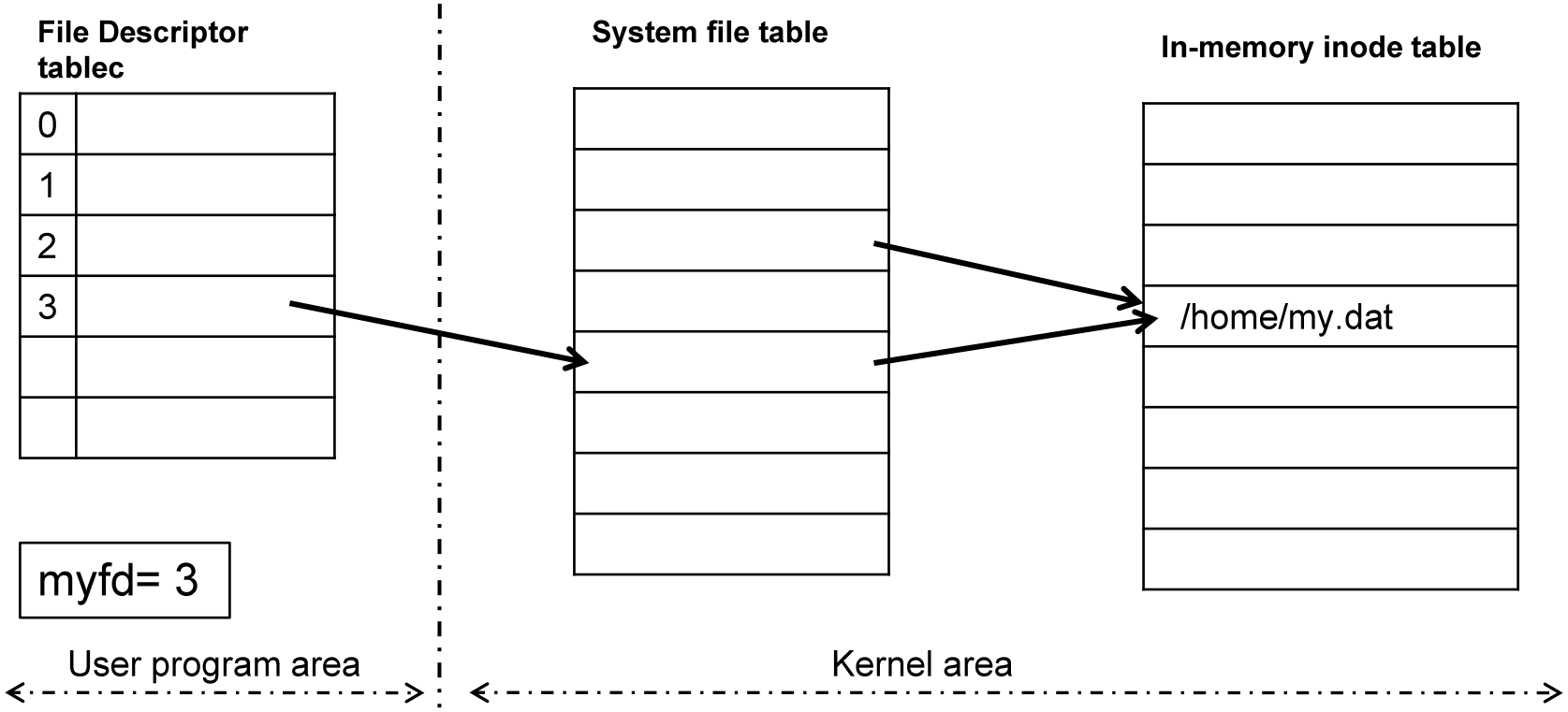
It contains an entry for each open file in the process.

The file descriptor table is part of the process user area, but the program cannot access it except through functions using the file descriptor.

Example: myfd = open("/home/my.dat", O_RDONLY);

The open function creates an entry in the file descriptor table that points to an entry in the system file table.

The open function returns the value 3, specifying that the file descriptor entry is in position three of the process file descriptor table.



schematic of the file descriptor table after a program execute the open()

The system file table, is shared by all the processes in the system, has an entry for each active open.

Each system file table entry contains the file offset, an indication of the access mode (i.e., read, write or read-write)

Also keep a count of the number of file descriptor table entries pointing to it.

Several system file table entries may correspond to the same physical file.

Each of these entries points to the same entry in the in-memory inode table.

The in-memory inode table contains an entry for each active file in the system.

When a program opens a particular physical file that is not currently open, the call creates an entry in this inode table for that file

As there are two entries in the system file table with pointers to the entry in the inode table, the figure shows that the file /home/my.dat had been opened before the code execute.

?? What happens when the process whose file descriptor table shown in figure executes the close(myfd) function?

The operating system deletes the fourth entry in the file descriptor table and the corresponding entry in the system file table.

If the operating system also deleted the inode table entry, it would leave other pointer hanging in the system file table.

Therefore, the inode table entry must have a count of the system file table entries that are pointing to it.

When a process executes the close function, the operating system decrements the count in the inode entry.

If the inode entry has a 0 count, the operating system deletes the inode entry from memory.

?? The system file table entry contains an offset that gives the current position in the file.
If two processes have each opened a file for reading, each process has its own offset into the file and reads the entire file independently of the other process.
What happens if each process opens the same file for write?
What would happen if the file offset were stored in the inode table instead of the system file table?

The writes are independent of each other.
Each user can write over what the other user has written because of the separate file offsets for each process.

If the offsets were stored in the inode table rather than in the system file table, the writes from different active opens would be consecutive.
Also, the processes that had opened a file for reading would only read parts of the file because the file offset they were using could be updated by other processes.

?? Suppose a process opens a file for reading and then forks a child process. Both the parent and
child can read from the file. How are reads by these two processes related? What about writes?

The child receives a copy of the parent's file descriptor table at the time of the fork.
The processes share a system file table entry and therefore also share the file offset.
The two processes read different parts of the file.
If no other processes have the file open, writes append to the end of the file and no data is lost on writes.

File pointers and Buffering

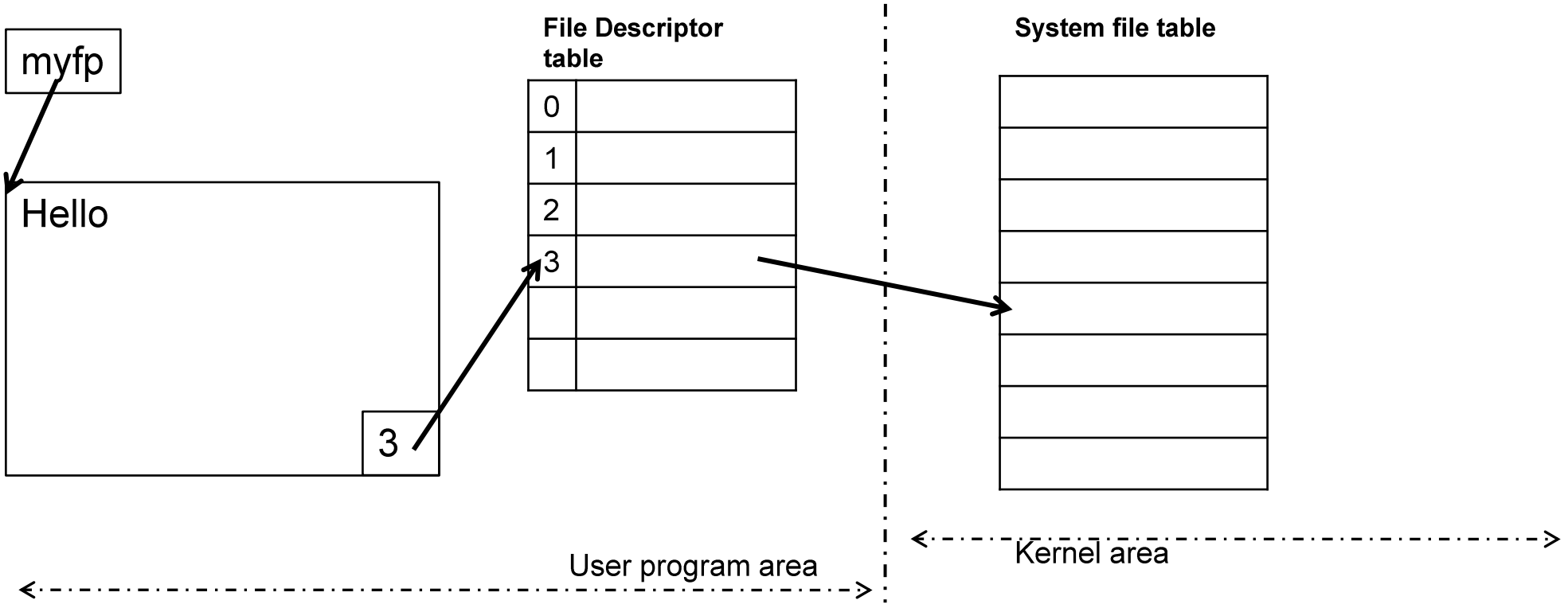
- The ISO C standard I/O library uses file pointers as handles for I/O.
- A file pointer points to a data structure called a FILE structure in the user area of the process.

Example: To open a file /home/my.dat for output and then writes a string to the file.

```
FILE *myfp;
if ((myfp = fopen("/home/ann/my.dat", "w")) == NULL)
    perror("Failed to open /home/my.dat");
else
    fprintf(myfp, “Hello");
```

File pointers and Buffering

- The FILE structure contains a buffer and a file descriptor value.
- The file descriptor value is the index of the entry in the file descriptor table that is actually used to output the file to disk
- file pointer is a handle to a handle



schematic of the FILE structure allocated by the fopen call

File pointers and Buffering

What happens when the program calls fprintf()

It depends on the type of file that was opened

Disk files

- usually fully buffered,

(fprintf does not actually write the message “Hello” to disk, but instead writes the bytes to a buffer in the FILE structure.)

When the buffer fills, the I/O subsystem calls write with the file descriptor,

So influenced by the time when a program executes fprintf and the time when the writing actually occurs.

Therefore on system crash the program appear to complete normally but its disk output could be incomplete

How to avoid the effects of buffering?

1. Use fflush call to forces buffered in the FILE structure to be written out.
2. call setvbuf to disable buffering.

File pointers and Buffering

What happens when the program calls fprintf()

- Terminal I/O works a little differently.

  Files associated with terminals are line buffered rather than fully buffered (except for standard error, which by default, is not buffered).

On output, line buffering means that the line is not written out until the buffer is full or until a newline symbol is encountered.

```
#include <stdio.h>
int main(void) {
fprintf(stdout, "a");
fprintf(stderr, "a has been written\n");
fprintf(stdout, "b");
fprintf(stderr, "b has been written\n");
fprintf(stdout, "\n");
return 0;
}
```

```
o/p
MN$ ./a.out
a has been written
b has been written
ab
```

standard error appear before the 'a' and 'b' because
standard output is line buffered, whereas standard error is not buffered.

File pointers and Buffering

```
#include <stdio.h>
int main(void) {
int i;
fprintf(stdout, "a");
scanf("%d", &i);
fprintf(stderr, "a has been written\n");
fprintf(stdout, "b");
fprintf(stderr, "b has been written\n");
fprintf(stdout, "\n");
return 0;
}
```

```
o/p
MN$ ./a.out
a
12
a has been written
b has been written
b
```

The scanf() flushes the buffer for stdout, so 'a' is displayed before the number is read.
After the number has been entered, 'b' still appears after the b has been written message

Inheritance of file descriptors

When fork creates a child, the child inherits a copy of most of the parent's environment and context, signal state, scheduling parameters and file descriptor table.

The implications of inheritance are not always obvious.

Because children receive a copy of their parent's file descriptor table at the time of the fork, the parent and children share the same file offsets for files that were opened by the parent prior to the fork.

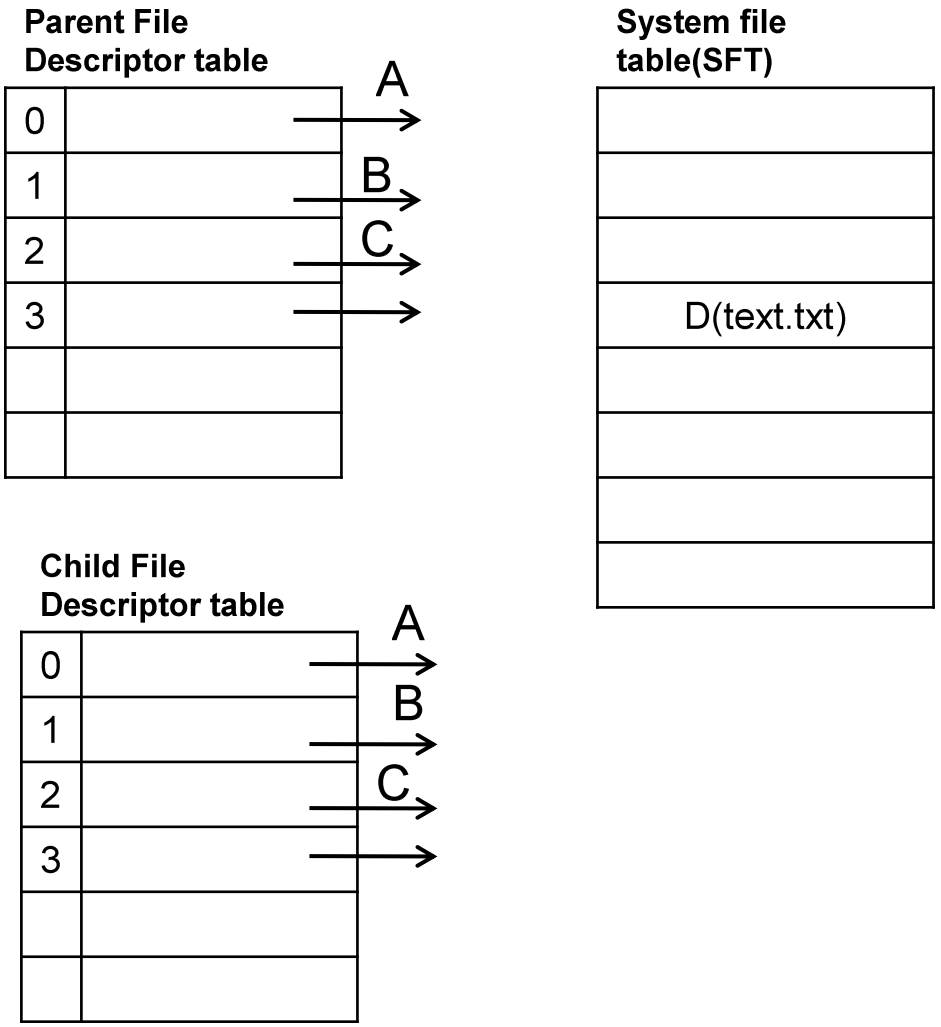
Inheritance of file descriptors

```
//child inherits the file descriptor

int main(void)
{
 char c='!';
  int fd;
  fd=open("test.txt",O_RDONLY);
  fork();
  read(fd,&c,1);
  printf("Process %ld got %c\n", (long)getpid(), c);
return 0;
}
```

The file descriptor table entries of the two processes point to the same entry in the system file table.
The parent and child therefore share the file offset, which is stored in the system file table.

Inheritance of file descriptors



parent opens file before forking, both parent and child share the system file table entry

Inheritance of file descriptors

?? If the first few bytes in the file text.txt are abcdefg.
What output would be generated

o/p

Process 9551 got a
Process 9552 got b

Since the two processes share the file offset,
the first one to read gets a and the second one to read gets b.

Inheritance of file descriptors

```
//child inherits the file descriptor, open after fork()
 int main(void)
 {
  char c='!';
   int fd;
   fork();
   fd=open("test.txt",O_RDONLY);
   read(fd,&c,1);
   printf("Process %ld got %c\n", (long)getpid(), c);
 return 0;
 }
```

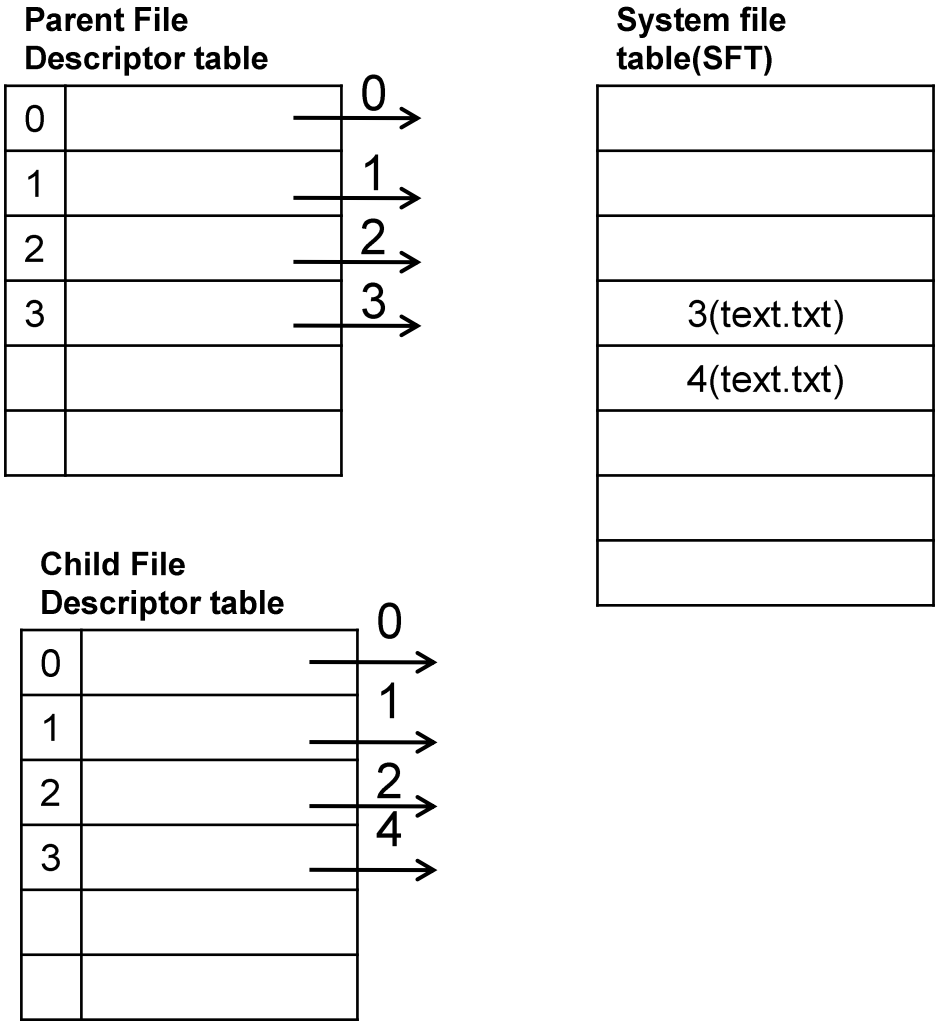
The file descriptor table entries of text.txt point to different system file table entries.

Consequently, the parent and child do not share the file offset.

The child does not inherit the file descriptor, because each process opens the file after the fork and each open creates a new entry in the system file table.

The parent and child still share system file table entries for standard input, standard output and standard error.

Inheritance of file descriptors



parent opens file before forking, both parent and child share the system file table entry

Inheritance of file descriptors

?? If the first few bytes in the file text.txt are abcdefg.
What output would be generated

o/p

Process 9551 got a
Process 9552 got a

Since the two processes use different file offsets,
each process reads the first byte of the file.

Inheritance of file descriptors

When a program closes a file, the entry in the file descriptor table is freed.

What about the corresponding entry in the system file table?

The system file table entry can only be freed if no more file descriptor table entries are pointing to it.

For this reason, each system file table entry contains a count of the number of file descriptor table entries that are pointing to it.

When a process closes a file, the operating system decrements the count and deletes the entry only when the count becomes 0.

Inheritance of file descriptors

How does fork affect the system file table?

Answer:

The system file table is in system space and is not duplicated by fork.

However, each entry in the system file table keeps a count of the number of file descriptor table entries pointing to it.

These counts must be adjusted to reflect the new file descriptor table created for the child.

Inheritance of file descriptors

What output would be generated

```
int main(void) {
printf("This is my output.");
fork();
return 0;
}
```

Because of buffering, the output of printf is likely to be written to the buffer corresponding to stdout, but not to the actual output device.

Since this buffer is part of the user space, it is duplicated by fork.

When the parent and the child each terminate, the return from main causes the buffers to be flushed as part of the cleanup.

The output appears as follows.

o/p:

This is my output.This is my output.

Inheritance of file descriptors

What output would be generated

```
int main(void) {
printf("This is my output.\n");
fork();
return 0;
}
```

The buffering of standard output is usually line buffering.

This means that the buffer is flushed when it contains a newline.

Since in this case a newline is output, the buffer is flushed before the fork and only one line of output will appear.

o/p:

This is my output.