【OS】Day44

Class	Operating System: Three Easy Pieces
Date	@February 28, 2022

【Ch39】Interlude: Files and Directories(2)

39.5 Reading and Writing, But Not Sequentially

Sometimes, it is useful to be able to read or write to a specific offset within a file. To do so, we will use the `lseek()` system call. Here is the function prototype:

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

- The first argument is a file descriptor
- The second argument is the offset, which positions the file offset to a particular location within the file.
- The third argument, called `whence` for historical reasons, determines exactly how the seek is performed.

From the man page:

```
If whence is SEEK_SET, the offset is set to offset bytes
If whence is SEEK_CUR, the offset is set to its current
  location plus offset bytes.
If whence is SEEK_END, the offset is set to the size of
  the file plus offset bytes.
```

For each file a process opens, the OS tracks a “current” offset, which determines where the next read or write will begin reading from or writing to within the file.

Thus, part of the abstraction of an open file is that it has a current offset, which is updated in two ways:

1. The first is when a read or write of N bytes takes place, N is added to the current offset. Thus each read or write implicitly updates the offset
2. The second is explicitly with `lseek`, which changes the offset as specified above

The offset is kept in the struct file we saw earlier.

```
struct file {
  int ref;
  char readable;
  char writable;
  struct inode *ip;
  uint off;
};
```

As we can see in the structure, the OS can use this to determine whether the opened file is readable or writable(or both), which underlying file it refers to(as pointed to by the `struct inode` pointer `ip`), and the current offset(`off`). There is also a reference count, which we will discuss further later.

These file structures represent all of the currently opened files in the system; together, they are sometimes referred to as the open file table.

The xv6 kernel just keeps these as an array as well, with one lock per entry, as shown below:

```
struct {
  struct spinlock lock;
  struct file file[NFILE];
} ftable;
```

Let's look at an example. First, let's track a process that opens a file(of size 300 bytes) and reads it by calling the read() system call repeatedly, each time reading 100 bytes.

System Calls	Return Code	Current Offset
`fd = open("file", O_RDONLY);`	3	0
`read(fd, buffer, 100);`	100	100
`read(fd, buffer, 100);`	100	200
`read(fd, buffer, 100);`	100	300
`read(fd, buffer, 100);`	0	300
`close(fd);`	0	–

There are a few things to be noted:

1. First, we can see how the current offset gets initialized to zero when the file is opened.
2. Next, we can see how it is incremented with each `read()` by the process
3. Finally, we can see how at the end, an attempted `read()` past the end of the file returns zero, thus indicating to the process that it has read the file in its entirety.

Second, let's trace a process that opens the same file twice and issues a read to each of them.

System Calls	Return Code	OFT[10] Current Offset	OFT[11] Current Offset
`fd1 = open("file", O_RDONLY);`	3	0	–
`fd2 = open("file", O_RDONLY);`	4	0	0
`read(fd1, buffer1, 100);`	100	100	0
`read(fd2, buffer2, 100);`	100	100	100
`close(fd1);`	0	–	100
`close(fd2);`	0	–	–

In this example, two file descriptors are allocated(3 and 4), and each refers to a different entry in the open file table. (in this example, entries 10 and 11, as shown in the table heading; OFT stands for Open File Table).

In one final example, a process uses `lseek()` to reposition the current offset before reading; in this case, only a single open file table entry is needed.

System Calls	Return Code	Current Offset
`fd = open("file", O_RDONLY);`	3	0
`lseek(fd, 200, SEEK_SET);`	200	200
`read(fd, buffer, 50);`	50	250
`close(fd);`	0	–

39.6 Shared File Table Entries: fork() and dup()

The mapping of file descriptor to an entry in the open file table is a one-to-one mapping. The file opened will have a unique entry in the open file table. Even if some other process reads the same file at the same time, each will have its own entry in the open file table. In this way, each logical reading or writing of a file is independent.

However, there are a few interesting cases where an entry in the open file table is shared. One of those cases occurs when a parent process creates a child process with `fork()`.

```
int main(int argc, char *argv[]) {
    int fd = open("file.txt", O_RDONLY);
    assert(fd >= 0);
    int rc = fork();
    if (rc == 0) {
        rc = lseek(fd, 10, SEEK_SET);
        printf("child: offset %d\n", rc);
    } else if (rc > 0) {
        (void) wait(NULL);
        printf("parent: offset %d\n",
               (int) lseek(fd, 0, SEEK_CUR));
    }
    return 0;
}
```

Figure 39.2: **Shared Parent/Child File Table Entries (`fork-seek.c`)**

The child adjusts the current offset via a call to `lseek()` and then exits. Finally the parent, after waiting for the child, checks the current offset and prints out its value.

When we run this program, we see the following output:

```
prompt> ./fork-seek
child: offset 10
parent: offset 10
prompt>
```

The following figure shows the relationships that connect each process's private descriptor array, the shared open file table entry, and the reference from it to the underlying file-system inode.

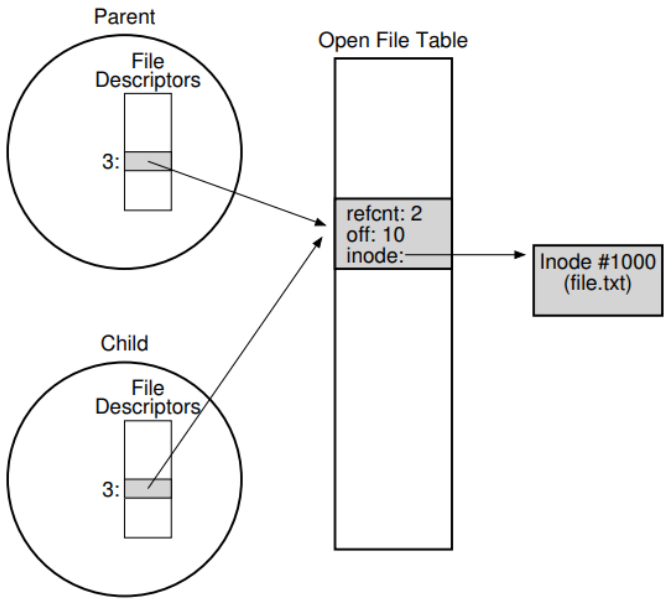

Figure 39.3: **Processes Sharing An Open File Table Entry**

Note that we finally make use of the reference count here. When a file table entry is shared its reference count is incremented; only when both processes close the file will the entry be removed.

Sharing open file table entries across parent and child is occasionally useful. For example, if we create a number of processes that are cooperatively working on a task, they can write to the same output file without any extra coordination.

One other interesting, and perhaps more useful, case for sharing occurs with the `dup()` system call.

The `dup()` system call allows a process to create a new file descriptor that refers to the same underlying open file as an existing descriptor.

```
int main(int argc, char *argv[]) {
    int fd = open("README", O_RDONLY);
    assert(fd >= 0);
    int fd2 = dup(fd);
    // now fd and fd2 can be used interchangeably
    return 0;
}
```

Figure 39.4: **Shared File Table Entry With dup() (dup.c)**

The `dup()` system call is useful when writing a UNIX shell and performing operations like output redirection.