

Operating System Labs

Yuanbin Wu
cs@ecnu

Announcement

- Project 1 due
 - 21:00, Oct. 8

Operating System Labs

- Introduction of I/O operations
- Project 1
 - Sorting

Operating System Labs

- Manipulate I/O
 - System call
 - File descriptor
 - No buffering
 - Standard library
 - FILE object
 - Buffering

Operating System Labs

- Manipulate I/O
 - System call
 - File descriptor
 - Standard library
 - FILE object
 - Buffer/non-buffer

I/O System Calls

- 5 basic system calls
 - `open()`, `read()`, `write()`, `lseek()`, `close()`
- I/O without buffering
- File sharing
 - understand file descriptor
- Other
 - `dup()`, `fcntl()`, `sync()`, `fsync()`, `ioctl()`

I/O System Calls

- File descriptor
 - Allocated when open a file
 - “ID” of the file **in the process** (unsigned int)
- Default
 - 0 (STDIN_FILENO): standard input
 - 1 (STDOUT_FILENO): standard output
 - 2 (STDERR_FILENO): standard error

I/O System Calls

- Open files:

```
# include <fcntl.h>
```

```
int open(const char *pathname, int o_flag, ... );  
// man 2 open
```

- Return value
 - Success: file descriptor
 - Failed: -1
- o_flag:
 - O_RDONLY, O_WRONLY, O_RDWR
 - Options:
 - O_APPEND, O_CREAT, O_TRUNC, ...

I/O System Calls

- Open files
 - File descriptors: the **smallest** one available

- Examples

```
int main (int argc, char **argv)
{
    int fd = open("foo", O_RDONLY);
    printf("%d", fd);
}
```

```
int main (int argc, char **argv)
{
    close(0);
    int fd = open("foo", O_RDONLY);
    printf("%d", fd);
}
```

I/O System Calls

- Open files
 - STDIN_FILENO, STDOUT_FILENO, STDERR_FILENO
 - opened by the OS when creating a process

I/O System Calls

- Close files

```
# include <unistd.h>  
  
int close(int fildes);
```

- Return

- Success: 0
- Failed: -1

I/O System Calls

- File Position

```
# include <unistd.h>
```

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

- “Current file offset”:

- An offset (in byte) to the beginning of the file

- whence:

- SEEK_SET, SEEK_CUR, SEEK_END

I/O System Calls

- Read files

```
# include <unistd.h>
```

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

- Start reading at “file offset”

- Return:

- Success: number of bytes read (0, if EOF)
- Failed: -1

- Return < size

- EOF
- Read from terminal (stdin), one line
- ...

I/O System Calls

- Write files

```
# include <unistd.h>
```

```
int write(int filedes, const void *buf, size_t nbytes);
```

Return:

- Success: number of bytes write
- Failed: -1

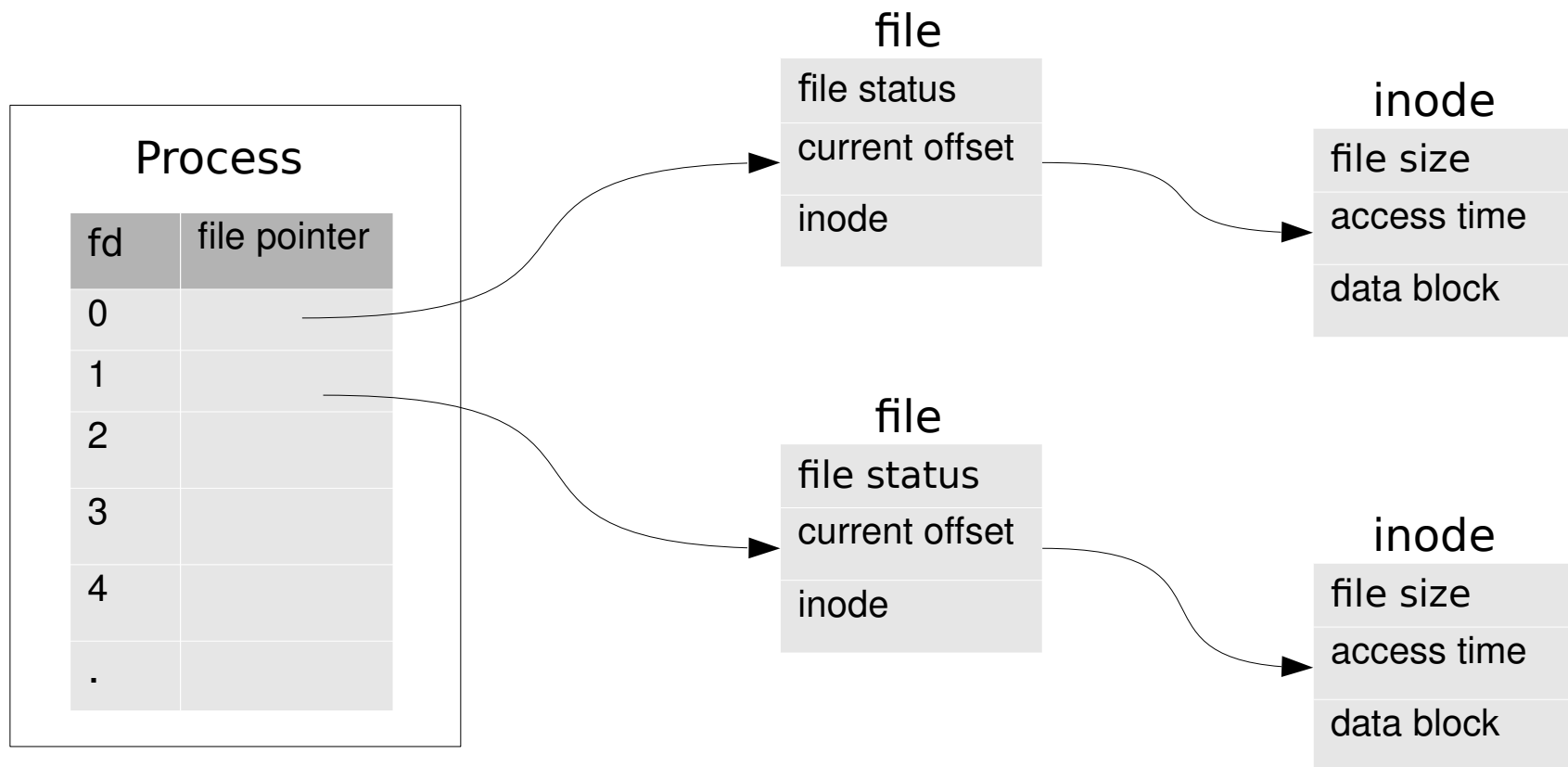
I/O System Calls

- I/O without buffer
 - No (user space) buffer
 - read(), write(): no buffer in **user space**
 - Do have buffer in **kernel space (by file system)**
 - Let's do some coding

- Buffering do matter!
 - printf, scanf in standard I/O library are buffered

I/O System Calls

- Revisit file descriptor



1. Each process has its own array of "struct file*"
2. Each file associates with only one "struct inode"
3. The "inode number" is a low-level id of a file


```

struct files_struct {
    int count;
    fd_set close_on_exec;
    fd_set open_fds;
    struct file * fd[NR_OPEN];
};

struct file {
    mode_t f_mode;
    loff_t f_pos;
    unsigned short f_flags;
    unsigned short f_count;
    unsigned long f_reada, f_ramax, f_raend, f_ralen, f_rawin;
    struct file *f_next, *f_prev;
    int f_owner;
    struct inode * f_inode;
    struct file_operations * f_op;
    unsigned long f_version;
    void *private_data;
};

struct ext2_inode {
    __u16 i_mode;      /* File type and access rights */
    __u16 i_uid;      /* Low 16 bits of Owner Uid */
    __u32 i_size;     /* Size in bytes */
    __u32 i_atime;    /* Access time */
    __u32 i_ctime;    /* Creation time */
    __u32 i_mtime;    /* Modification time */
    __u32 i_dtime;    /* Deletion Time */
    __u16 i_gid;      /* Low 16 bits of Group Id */
    __u16 i_links_count; /* Links count */
    __u32 i_blocks;   /* Blocks count */
    __u32 i_flags;    /* File flags */
    ...
    __u32 i_block[EXT2_N_BLOCKS]; /* Pointers to blocks */
    ...
};

```

A, B open the same file

Process A

fd	file pointer
0	
1	
2	
3	
4	
.	

file

file status
current offset
inode

inode

file size
access time
data block

Process B

fd	file pointer
0	
1	
2	
3	
4	
.	

file

file status
current offset
inode

I/O System Calls

- File sharing
 - Atomic operations
 - Example: `open("file", O_WRONLY | O_APPEND)`
 - Two process A, B run the same code, what will happen?

```
if (lseek(fd, 0, SEEK_END) < 0)
    perror("lseek");
```

```
if (write(fd, buf, 100) < 100)
    perro("write");
```

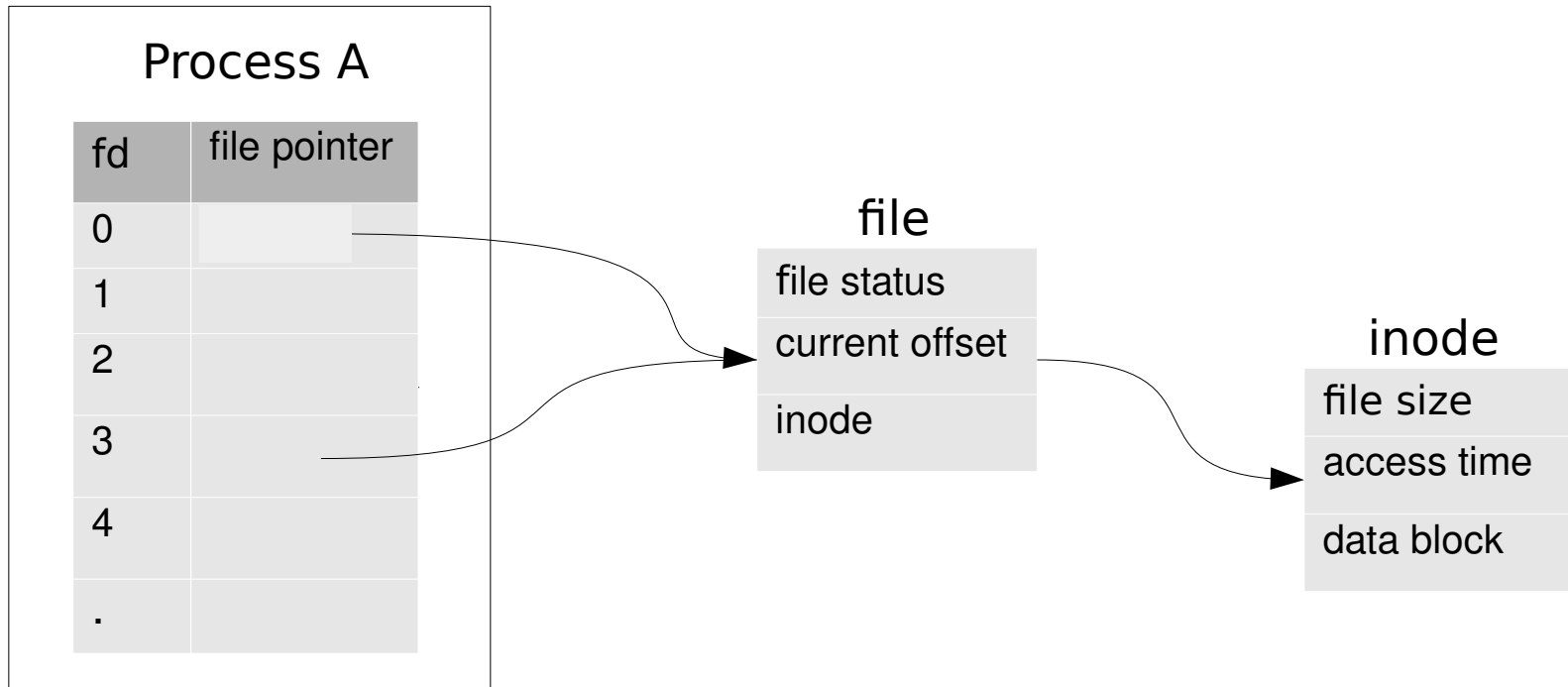
I/O System Calls

- Duplicate file descriptor

```
# include <unistd.h>  
  
int dup2(int fd, int fd2);
```

- set “fd2” point to the same file of “fd”
- Return
 - Success: fd
 - Failed: -1

```
// if fd 0 is open, close it first  
dup2(3, 0);
```



1. a file with multiple file descriptors
2. I/O redirection

I/O System Calls

- Other system calls
 - `sync()` / `fsync()`:
 - “delay write”
 - Flush kernel buffer
 - `fcntl()`: change file (opened) attributes
 - `ioctl()`: other methods

I/O System Calls

- Summary
 - File descriptor
 - open, close, read, write, lseek, dup
 - File sharing

Operating System Labs

- Manipulate I/O
 - System call
 - File descriptor
 - No buffering
 - Standard library
 - FILE object
 - Buffering

Standard I/O Library

- `#include <stdio.h>`
 - FILE object (structure)
 - Buffering
 - Formatted I/O

Standard I/O Library

- Recall:

```
#include <stdio.h>
void foo()
{
    printf("bar\n");
}
```

User application

```
printf()
fprintf()
malloc()
atoi()
```

Library Functions
(Glibc)

```
write(), reads(),
mmap()
```

System Calls

Kernel

Standard I/O Library

```
# include <fcntl.h>
```

```
int main (int argc, char **argv)
{
    int fd = open("foo", O_RDONLY);
}
```

```
# include <stdio.h>
```

```
int main (int argc, char **argv)
{
    FILE* fp = fopen("foo", "r");
}
```

- Stream and FILE object
 - A wrapper of file descriptor
 - More information:
 - **buffer**
 - error info
 - single-byte or multi-byte

Standard I/O Library

- FILE object
 - Opaque pointer
 - The implementation is hidden
 - Access the struct member through functions
 - Operations on FILE object
 - Get file descriptor: `fileno(FILE* f)`
 - Set buffer: `setbuf(FILE* f, char* buf)`

Standard I/O Library

- Buffering
 - stdio provide a “standard I/O buffer” (**user space**)
- Three types of buffering
 - Full buffered
 - Performs I/O when the buffer is full
 - Line buffered
 - Performs I/O when encounter a newline
 - Unbuffered
 - Performs I/O immediately, no buffer

Standard I/O Library

- Three types of buffering: cases
 - Standard error is unbuffered
 - A stream is line buffered if it refers to terminal device, otherwise full buffered
- Write “standard I/O buffer” to disc:

```
# include <stdio.h>  
  
int fflush(FILE *fp);
```

Standard I/O Library

- Open/Close streams

```
# include <stdio.h>
```

```
FILE *fopen(const char* path, const char * type);
```

```
FILE *fdopen(int fd, const char * type);
```

```
int fclose(FILE* fp);
```

- Type: “r”, “w”, “a”, “r+” . . .

- Return

- Failed: NULL

Standard I/O Library

- Character-at-a-time I/O

```
# include <stdio.h>
```

```
int getc(FILE *fp);
```

```
int fgetc(FILE *fp);
```

```
int putc(FILE *fp);
```

```
int fputc(FILE *fp);
```


Standard I/O Library

- Line-at-a-time I/O

```
# include <stdio.h>
```

```
char* fgets(char *buf, int n, FILE *fp);
```

```
char* gets(char *buf);    // should be never used
```

```
int fputs(char *str, FILE *fp);
```

```
int puts(char *str);
```

Standard I/O Library

- Direct I/O

```
# include <stdio.h>
```

```
size_t fread(void *ptr, size_t size, size_t, nobj, FILE *fp);  
size_t fwrite(void *ptr, size_t size, size_t, nobj, FILE *fp);
```

Standard I/O Library

- Standard I/O efficiency
 - Recall: buffering in system calls
 - Let's do some coding again

Standard I/O Library

- Formatted I/O
 - printf, fprintf, scanf

Standard I/O Library

- Summary
- `#include <stdio.h>`
 - FILE object (structure)
 - Buffering
 - Formatted I/O

Introduction of I/O Operations

- Summary
 - System call
 - File descriptor
 - No buffering
 - Standard library
 - FILE object
 - Buffering

Project 1

- Sorting

Announcement

- Project 1 due
 - 21:00, Oct. 8