CS 3113
(some) File I/O Details + Pipes

Resource Sharing Challenges

- With modern OSes, we have the opportunity to be performing many tasks (executing many processes) at once
- In fact, your Linux instance has many processes executing right now (try the **top** command; and … ^c will get you out of it)
- Any time two processes try to access the same resource at the same time, the potential exists for things going very wrong
- In some cases, the OS automatically addresses the resource contention; in other cases, the processes must take special action through the OS to ensure that problems do not occur

File System Example

Program pseudo-code:

```
1. If file FOO does not exist then:
2.   Open the file for writing, creating it
3.   Place data in the file
4.   Close the file
```

- Each of these steps involves a system call
- Execution of this process can be interrupted at any time, allowing another process to do work

Creating a File if it Does not Exist

Listing 5-1: Incorrect code to exclusively open a file

from **fileio/bad_exclusive_open.c**

```
fd = open(argv[1], O_WRONLY);       /* Open 1: check if file exists */
    if (fd != -1) {                     /* Open succeeded */
        printf("[PID %ld] File \"%s\" already exists\n",
                (long) getpid(), argv[1]);
        close(fd);
    } else {
        if (errno != ENOENT) {          /* Failed for unexpected reason */
            errExit("open");
        } else {
            /* WINDOW FOR FAILURE */
            fd = open(argv[1], O_WRONLY | O_CREAT, S_IRUSR | S_IWUSR);
            if (fd == -1)
                errExit("open");

            printf("[PID %ld] Created file \"%s\" exclusively\n",
                    (long) getpid(), argv[1]);          /* MAY NOT BE TRUE! */
        }
    }
```

from **fileio/bad_exclusive_open.c**

Multiple Processes

- Both accessing the same resource
- **Race condition**: outcome depends on who arrives first
- Bug if A is interrupted after processing the first open & B jumps in
- Will also see the term **synchronization problem**

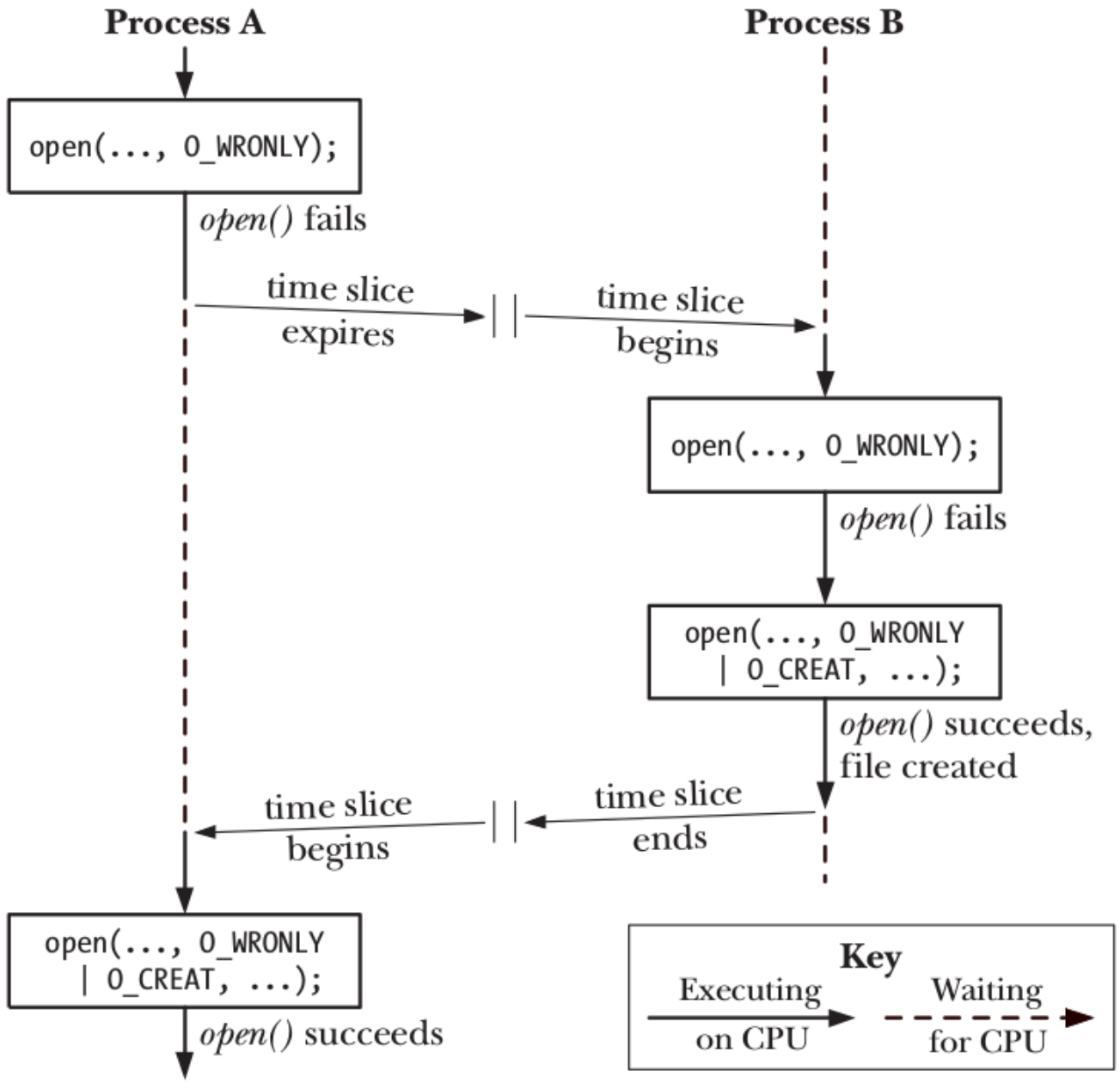

Figure 5-1: Failing to exclusively create a file

Atomicity

- There are many situations where access to a common resource involves a sequence of operations that cannot be interrupted.
  - We want to treat these operations **atomically** (i.e., that they cannot be broken apart)
- These are called **critical sections**

- Because this issue comes up in many different ways in an OS, we will find a range of context-specific solutions to this problem

Atomicity for the File Existence/Creation Operation

Solution: use a single open():

```
int fid = open(argv[1], O_WRONLY | O_CREAT | O_EXCL,
                        S_IRUSR | S_IWUSR);
```

- The system call **is** executed atomically
- fid < 0: failure on file creation
  - Already exists (so another process owns the file), or other failure, e.g. directory doesn't exist
- fid >= 0: file was created successfully

A Note of Caution

- For file system system calls, atomicity is a function of **both** the OS and the file system
- Local files are managed by your OS, so if it handles EXCL correctly (any POSIX OS will), then you are okay
- For files located on remote servers (e.g., Samba, NFS, ZFS), atomicity must also be enforced on those servers & they have each made their own decisions

… so tread cautiously

File Descriptors vs File Pointers

- File descriptor:
  - int type that references a table of open streams
  - Can reference files, pipes or sockets (more on the middle soon; latter is for inter-process communication)
  - Access through system calls: open(), read(), write(), close() …
- File pointer
  - FILE type defined in stdio.h (it is a struct)
  - Includes the file descriptor, but adds buffering and other features
  - Access through the stdio library: fopen(), fread(), fwrite(), fclose(), fprintf(), fscanf()
  - When working with files, this is the preferred interface

File Pointer Example

```
#include <stdio.h>

int main(int argc, char** argv)
{
  FILE* fp = fopen(argv[1], "w");
  if(fp == NULL){
    printf("Error opening file.\n");
  }else{
    fprintf(fp, "Foo bar: %s\n", argv[1]);
    fclose(fp);
  }
}
```

Another File Open Function …

```
FILE *freopen(const char *path, const char *mode, FILE *stream);
```

- Opens the specified file and associates it with the <stream> FILE
- If <stream> is already an open file, then it is closed first
- Returns <stream> if successful

Useful for substituting a file for the stdin stream

Flushing Streams

- Because FILE streams are buffered, a `fprintf()` does not necessarily affect the file immediately
- Instead, the bytes are dropped into a buffer; at some point the library will decide to move the bytes from the buffer to the file
- `fflush(fp)` will immediately force all bytes in the buffer to the file

Creating a New Process: the Basics

System call: fork()

- Defined in unistd.h
- Creates a duplicate of the calling process: copy of all of the data
  - All of the file descriptors are copied!
- Differences between the two processes include:
  - The child process has a new process id (PID)
  - In the child process, fork() returns zero
  - In the parent process, fork() returns the PID of the child

fork() demo…

File Descriptors to Files (or Streams)

What do we need to know about an open file (or other stream)?

File Descriptors to Files (or Streams)

What do we need to know about an open file (or other stream)?

- Type of the data & its location. If a file:
  - Where on the storage device?
  - What are the access permissions?
  - File size
  - Timestamps (access, creation)
- What part of the file are we accessing now? (the offset)
- How are we accessing the file (including read vs write, and append, creation)

File Descriptors to Files (or Streams): Three Levels of Representation

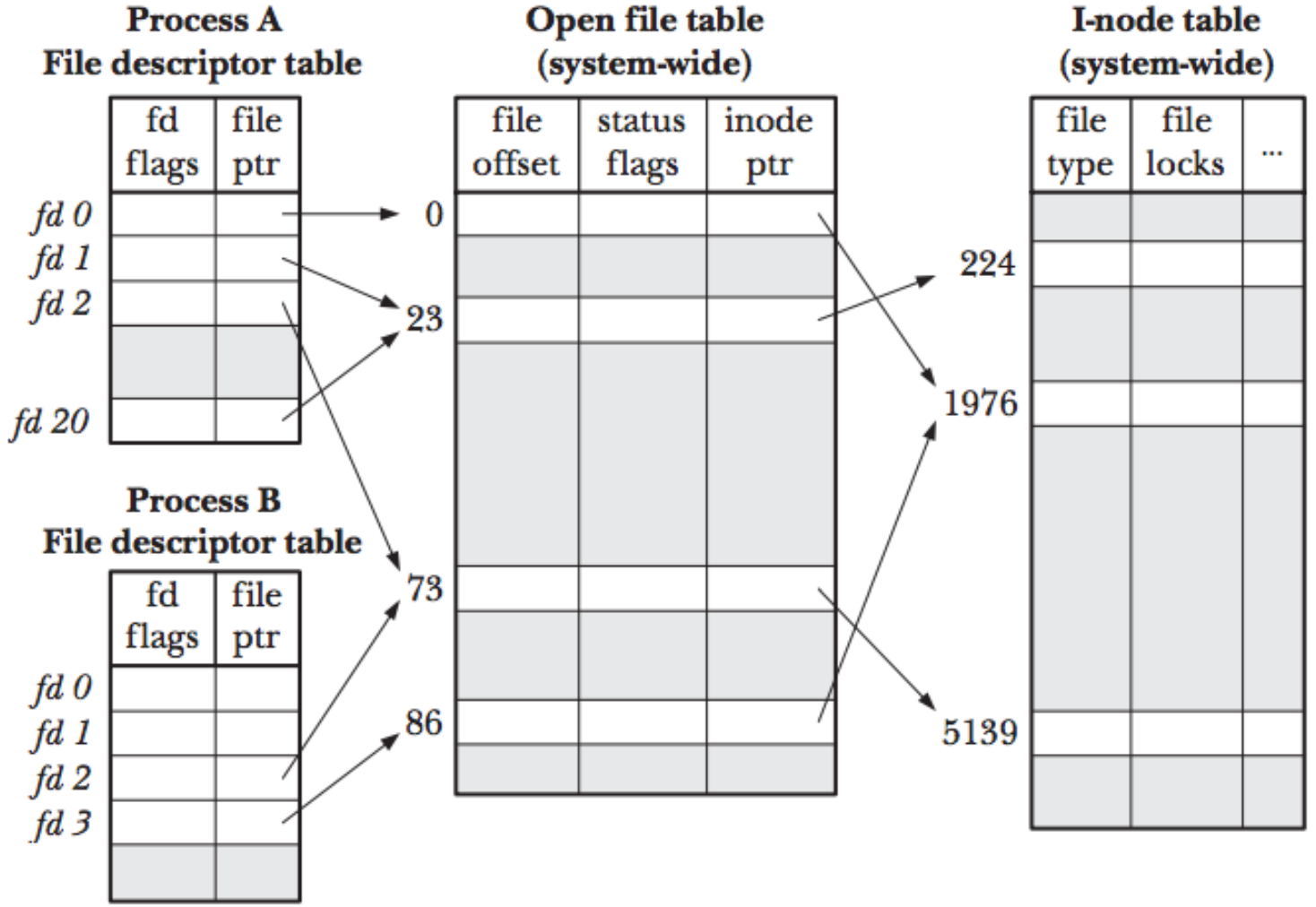

Figure 5-2: Relationship between file descriptors, open file descriptions, and i-nodes

Copying a File Descriptor

- In some cases, it is useful for a process to be able to refer to the same file/stream using two different file descriptors
  - For example, if we want output written to both stdout and stderr to appear on stderr
- Allocate the first available fd & configure it to point to the same resource as oldfd:

```
newfd = dup(oldfd)
```

- Close newfd (if it is open) and allocate it to point to oldfd:

```
newfd = dup2(oldfd, newfd)
```

Reading and Writing

```
char buf[20]
int n = read(0, buf, 5)
```

- Attempts to read the next 5 bytes from stdin (assuming there are 5 bytes before the EOF)
- Moves the offset by 5 to the right (again, if there are 5)
- Returns the number actually read
- Blocks (by default) if there are no bytes to read

pread() / pwrite()

```
char buf[20];
int n = pread(0, buf, 5, 200);
```

- Remember the current offset
- Change the offset to 200
- Read 5 bytes (if the exist) into buf
- Change the offset back to the original

- These operations are all done atomically!
  - Multiple threads/processes can all access a w/r file through the same fd and yet not interfere with one-another

Pipes

- Simple form of inter-process communication (IPC)
- Generally, one process has access to the input to the pipe, while another process has access to the output of the pipe
- Unidirectional. If we need bidirectional communication, then we need a 2nd pipe or a different form of IPC
- Pipes provide a buffer for written data until the other process can read them
- Pipes are identified using file descriptors, so the standard FD operators can be used: read(), write(), close()…

Typical Use

- Parent process creates the pipe: this results in both an input and an output file descriptor
- Parent forks a child process, which also has access to the pipe
- Parent and child each close one of their input/output pipe file descriptors
- The writer then uses write() to send bytes into the pipe
- The reader then uses read() to pull bytes out of the pipe

The bytes can be anything!