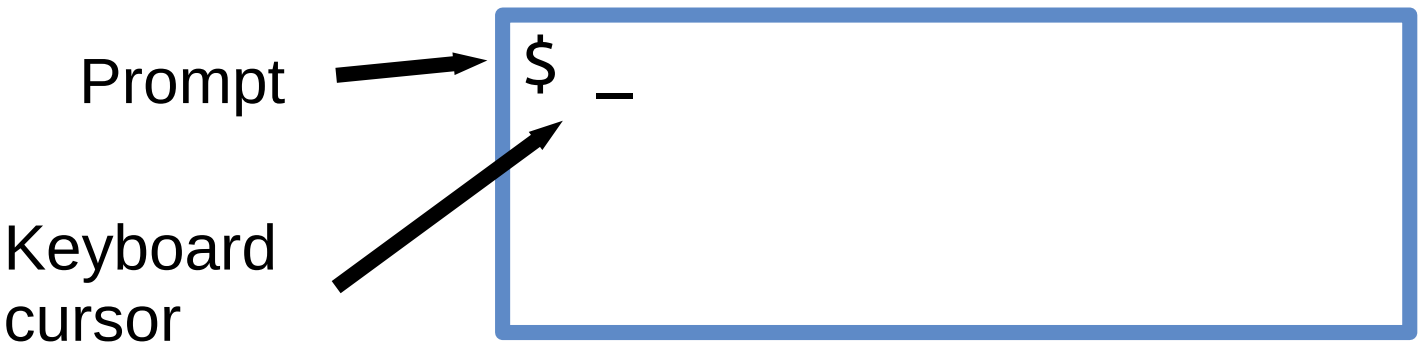
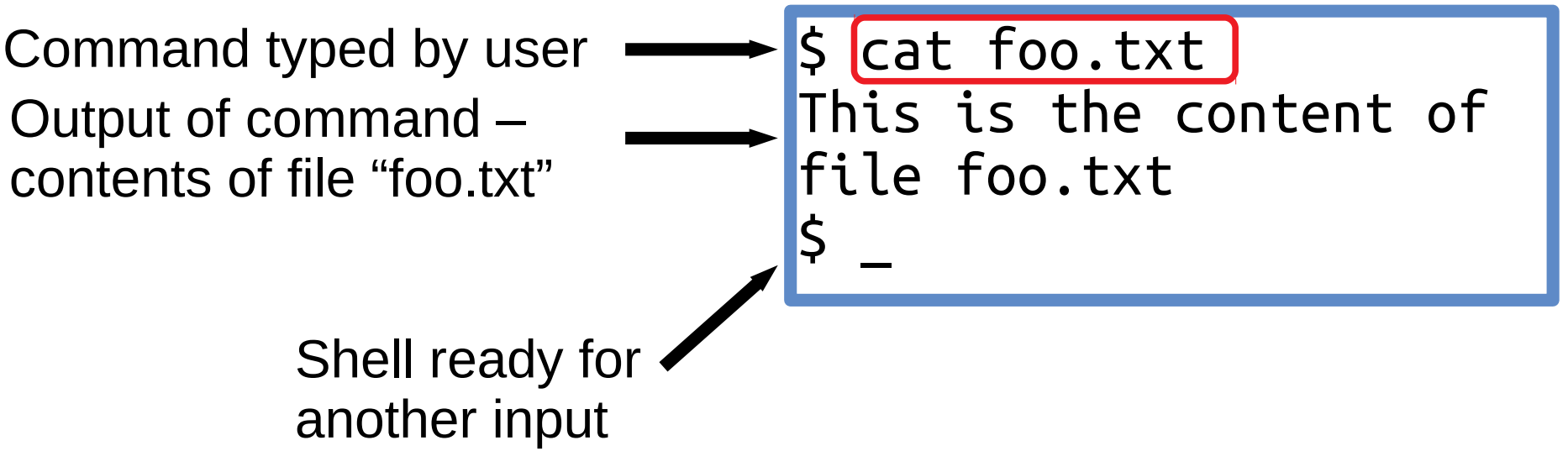
Operating System API
Case study: UNIX shell

Unix shell

- Provides interactive command execution
- Was part of OS kernel initially, now a normal program
- The shell interface looks like this:



Unix shell


Command typed by user
Output of command – contents of file “foo.txt”
Shell ready for another input
$ cat foo.txt
This is the content of file foo.txt
$ _

Unix shell: barebones code

```
while (1) {
  write (1, “$ “, 2);          // print “$ “
  readcommand(command, args);
  // ... spawn new process and wait for it
to finish
}
```

Unix shell: barebones code

```
while (1) {
  write (1, “$ “, 2);          // print “$ “
  readcommand(command, args);
  // ... spawn new process and wait for it
to finish
}
```

write() syscall:

write(fd, pointer, size) - write ‘size’ bytes pointed to by ‘pointer’ to file (or device) backed by file-descriptor ‘fd’.

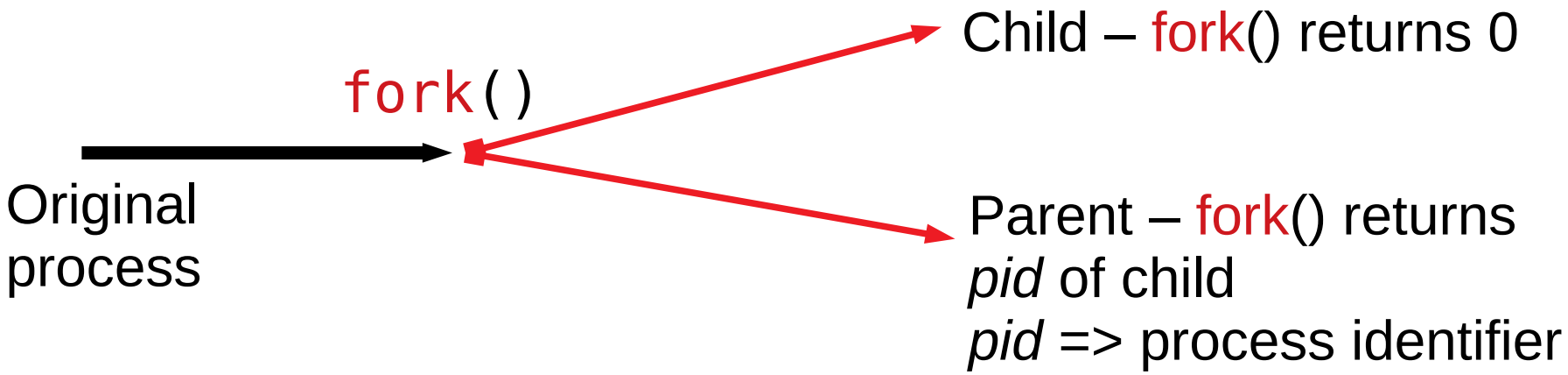
Unix I/O facilities

- Set of syscalls: read, write, open, close ...
- `fd = open(“filename”, ...);`
- ‘`fd`’ is the “file descriptor” for the file
  - The OS maintains a table of *open* file descriptors for each process.

```
while (1) {
  write (1, “$ “, 2);
  readcommand(command, args);
  if ((pid = fork()) == 0) // create ‘copy’
                           // of this
                           // process
    exec(command, args); // execute command
  else if (pid > 0)
    wait(0);
  else // handle error
}
```

Unix process management facilities

- Set of syscalls: fork, exec, wait, exit, ...
- `fork()` creates a replica of current process
  - Both processses then continue execution from the next statement.



Unix process management facilities

- Set of syscalls: fork, exec, wait, exit, ...
- `fork()` creates a replica of current process

```
if ((pid = fork()) == 0)
   exec(command, args);
else if (pid > 0)
   wait(0);
```

} Child part

} Parent part

Unix process management facilities

- Set of syscalls: fork, exec, wait, exit, ...
- `fork()` creates a replica of current process

```
if ((pid = fork()) == 0)
   exec(command, args);
else if (pid > 0)
   wait(0);
```

} Child part

} Parent part

- `exec()` executes program specified by `command` -- replacing the current process

Unix process management facilities

```
if ((pid = fork()) == 0)
   exec(command, args);
else if (pid > 0)
   wait(0);
```

} Child part

} Parent part

- `exec()` executes program specified by `command` -- replacing the current process
- `wait()` suspends the current process until the child calls `exit()`

Unix process management facilities

- `fork()` + `exec()` required for executing a new program
- Was somewhat simple to implement in those days
- Simple but enables other use cases
  - I/O redirection, pipes etc.
- Windows has `CreateProcess()` for the same job
  - 10 formal parameters
- Performance differences due to copy operation

More on Unix I/O facilities

- Each process has 3 OS provided file-descriptors *open* by default:
  - `stdin (0), stdout (1), stderr` (2)
- For programs started by shell:
  - `stdin` connected to keyboard
  - `stdout` connected to console
  - `stderr` (also) connected to console

Unix shell – I/O redirection

```
$ ls > tmp1
$ cat tmp1
Desktop Documents Downloads Music Pictures Videos
```

- ‘>’ redirects output (stdout) of ls to file tmp1
- Very useful construct – shell essentially acting as a programming environment
  - Similar functionality would otherwise require changes to the program

Unix shell – I/O redirection

- ‘`>`’ for redirecting `stdout`
- ‘`<`’ for redirecting `stdin`
- ‘`2>`’ for redirecting `stderr`

```
$ wc < tmp1 > tmp2
$ cat tmp2
1 6 50
```

Unix shell – I/O redirection implementation

```
if ((pid = fork()) == 0) {
    // close default stdin
    close(0);
    open(stdin_filename);
    // close default stdout
    close(1);
    open(stdout_filename);
    exec(command, args);
}
```

I/O redirection – Another example

```
$ sh < tests.sh > out
$ grep “fail” < out > fails
$ wc -l < fails
1
$ rm out fails
```

- Executes commands in `tests.sh`, saving output to file `out`
- Search for “fail” in `out`, save the results in file `fails`
- Count the number of lines in `fails`

Introducing “pipe”

```
$ sh < tests.sh > out
$ grep “fail” < out > fails
$ wc -l < fails
1
$ rm out fails
```

Same solution using pipe “|” construct:

```
$ sh < tests.sh | grep “fail” | wc -l
```

“pipe” -- overview

- Unidirectional of data (bytes) from one process to another
- Kernel manages the flow

“pipe” -- syscall

- Signature: `pipe(int[2])`
- Usage:

```
int pfd[2];
pipe(pfd);
```

- `pfd[0]` – read end of pipe
- `pfd[1]` – write end of pipe

“pipe” -- inter-process communication (IPC)

```
int pfd[2];
pipe(pfd);
if ((pid = fork()) == 0) {
   write(pfd[1], “Hello from child”, 16);
   exit(0);
} else (pid > 0) {
   sz = read(pfd[0], buf, 100); // blocks until
write is executed by child
   write(1, buf, sz);
   wait(0);
}
```

Questions?