- xv6
  - Chapter 1 Operating system interfaces 9
    - Introduction 9
      - Shell 10
        > The shell is an ordinary program that reads commands from the user and executes them.
      - System Calls 9
        > When a process needs to invoke a kernel service, it invokes a system call, one of the calls in the operating system's interface. The system call enters the kernel; the kernel performs the service and returns. Thus a process alternates between executing in user space and kernel space.
        - kernel space 9
        - user space 9
      - Design Interfaces 9
        > The trick in resolving this tension is to design interfaces that rely on a few mechanisms that can be combined to provide much generality.
    - 1.1 Processes and memory 10
      - Process 10
        > process consists of user-space memory
        - instructions 10
        - data 10
        - stack 10
        - per-process state private to the kernel 10
        - Switch among processes 10
          > Xv6 time-shares processes: it transparently switches the available CPUs among the set of processes waiting to execute.
      - System call examples
        - fork() 10
          > Fork creates a new process
          > child process
          > Fork returns in both the parent and the child
          - Parent 10
            > In the parent, fork returns the child's PID

- Child (10)

  > in the child, fork returns zero

- wait() (11)

  > 1. Returns the PID of an exited (or killed) child of the current process
  >
  > 2. opies the exit status of the child to the address passed to wait;

▼ exec() (12)

> The exec system call replaces the calling process's memory with a new memory image loaded from a file stored in the file system.

- Not return to program (12)

  > When exec succeeds, it does not return to the calling program; instead, the instructions loaded from the file start executing at the entry point declared in the ELF header

▼ Shell (12)

> shell is simple

▼ Why fork and exec are not combined

Read command

Fork

Parent waits

Child executes

> The shell exploits ***the separation in its implementation of I/O redirection***.

- Copy-on-write (12)

  > To avoid the wastefulness of creating a duplicate process and then immediately replacing it (with exec), operating kernels optimize the implementation of fork for this use case by using virtual memory techniques such as ***copy-on-write*** (see Section 4.6)

- Read command (12)

  > reads a line of input from the user with getcmd

- Fork (12)

  > fork, which creates a copy of the shell process

- Parent waits (12)

  > The parent calls wait

- Child executes (12)

  > while the child runs the command

- List of System Call (11)

> | System call | Description |
> |---|---|
> | int fork() | Create a process, return child's PID. |
> | int exit(int status) | Terminate the current process; status reported to wait(). No return. |
> | int wait(int *status) | Wait for a child to exit; exit status in *status; returns child PID. |
> | int kill(int pid) | Terminate process PID. Returns 0, or -1 for error. |
> | int getpid() | Return the current process's PID. |
> | int sleep(int n) | Pause for n clock ticks. |
> | int exec(char *file, char *argv[]) | Load a file and execute it with arguments; only returns if error. |
> | char *sbrk(int n) | Grow process's memory by n bytes. Returns start of new memory. |
> | int open(char *file, int flags) | Open a file; flags indicate read/write; returns an fd (file descriptor). |
> | int write(int fd, char *buf, int n) | Write n bytes from buf to file descriptor fd; returns n. |
> | int read(int fd, char *buf, int n) | Read n bytes into buf; returns number read; or 0 if end of file. |
> | int close(int fd) | Release open file fd. |
> | int dup(int fd) | Return a new file descriptor referring to the same file as fd. |
> | int pipe(int p[]) | Create a pipe, put read/write file descriptors in p[0] and p[1]. |
> | int chdir(char *dir) | Change the current directory. |
> | int mkdir(char *dir) | Create a new directory. |
> | int mknod(char *file, int, int) | Create a device file. |
> | int fstat(int fd, struct stat *st) | Place info about an open file into *st. |
> | int stat(char *file, struct stat *st) | Place info about a named file into *st. |
> | int link(char *file1, char *file2) | Create another name (file2) for the file file1. |
> | int unlink(char *file) | Remove a file. |

1.2 I/O and File descriptors

13

Definition

13

> A file descriptor is ***a small integer*** representing ***a kernel-managed object*** that a process may read from or write to.

> By convention, a process reads from file descriptor 0 (standard input), writes output to file descriptor 1 (standard output), and writes error messages to file descriptor 2 (standard error).

- **stdin** 13

  > file descriptor 0 (standard input)

- **stdout** 13

  > file descriptor 1 (standard output)

- **stderr** 13

  > file descriptor 2 (standard error)

How

13

> obtain a file descriptor

- **File, Dir and Device** 13

  > opening a file, directory, or device

- **Pipe** 13

  > creating a pipe

- **Duplicate** 13

  > duplicating an existing descriptor

read and write

13

> The read and write system calls read bytes from and write bytes to open files named by file descriptors.

- **read** 13

  > read(fd, buf, n)
  >
  > > 1. reads at most n bytes from the file descriptor fd
  > >
  > > 2. copies them into buf

- **write** 13

  > write(fd, buf, n)
  >
  > > 1. writes n bytes from buf to the file descriptor fd
  > >
  > > 2. returns the number of bytes written

- **close** 14

  > The close system call releases a file descriptor

redirection

14

> cat < input.txt

- New file descriptor

  > A newly allocated file descriptor is always the lowestnumbered unused descriptor of the current process.

14

14

```
char *argv[2];

argv[0] = "cat";
argv[1] = 0;
if(fork() == 0) {
  close(0);
  open("input.txt", O_RDONLY);
  exec("cat", argv);
}
```

```
case REDIR:
  rcmd = (struct redircmd*)cmd;
  close(rcmd->fd);
  if(open(rcmd->file, rcmd->mode) < 0){
    fprintf(2, "open %s failed\n", rcmd->file);
    exit(1);
  }
  runcmd(rcmd->cmd);
  break;
```

- close

  > child closes file descriptor 0

14

- open

  > 0 will be the smallest available file descriptor

14

- fork and exec are separate calls

  > The shell has a chance to redirect the child's I/O without disturbing the I/O setup of the main shell

14

hello world into a file

15

- fork

```
if(fork() == 0) {
  write(1, "hello ", 6);
  exit(0);
} else {
  wait(0);
  write(1, "world\n", 6);
}
```

15

dup

```
fd = dup(1);
write(1, "hello ", 6);
write(fd, "world\n", 6);
```

> dup system call duplicates an existing file descriptor

> Both file descriptors share an offset

15

- Error redirection 15
  - ls existing-file non-existing-file > tmp1 2>&1.
  - Both the name
  - and the error message
  - will show up in the file tmp1

1.3 Pipes 15

- a small kernel buffer 15
- processes as a pair of file descriptors 15
  - reading 15
  - writing 15
- 16

```
int p[2];
char *argv[2];

argv[0] = "wc";
argv[1] = 0;

pipe(p);
if(fork() == 0) {
  close(0);
  dup(p[0]);
  close(p[0]);
  close(p[1]);
  exec("/bin/wc", argv);
} else {
  close(p[0]);
  write(p[1], "hello world\n", 12);
  close(p[1]);
}
```

- 1.4 File system 17
- 1.5 Real world 19