# ECS 150 - System Calls

*Prof. Joël Porquet-Lupine*

UC Davis - 2020/2021

# C Standard Library

## C program example

```c
#include <fcntl.h>
#include <stdio.h>
...
int main(int argc, char *argv[])
{
    int fd;
    char buf[4];

    if (argc < 2)
        exit(EXIT_FAILURE);

    memset(buf, 0, sizeof(buf));

    fd = open(argv[1], O_RDONLY);
    if (fd < 0) {
        perror("open");
        exit(EXIT_FAILURE);
    }
    read(fd, buf, sizeof(buf));
    close(fd);

    printf("Executable detection... ");
    if (buf[0] == 0x7F && buf[1] == 0x45
        && buf[2] == 0x4C && buf[3] == 0x46)
        printf("ELF");
    else if (buf[0] == '#' && buf[1] == '!')
        printf("script");
    else
        printf("Not an executable!");
    printf("\n");

    return 0;
}
```

exec_detector.c

### Execution

```
$ make exec_detector
cc -o exec_detector exec_detector.c
```

```
$ ./exec_detector
$ ./exec_detector /path/to/nofile
open: No such file or directory
```

```
$ ./exec_detector exec_detector
Executable detection... ELF
```

```
$ ./exec_detector exec_detector.c
Executable detection... Not an executable!
```

```
$ ./exec_detector /bin/firefox
Executable detection... script
```

```
$ cat /bin/firefox
#!/bin/sh
exec /usr/lib/firefox/firefox "$@"
```

```
$ ./exec_detector /usr/lib/firefox/firefox
Executable detection... ELF
```

# C Standard Library

## Library functions

```c
#include <fcntl.h>
#include <stdio.h>
...
    memset(buf, 0, sizeof(buf));
...
    fd = open(argv[1], O_RDONLY);
    if (fd < 0) {
        perror("open");
        exit(EXIT_FAILURE);
    }
    read(fd, buf, sizeof(buf));
    close(fd);
...
    printf("Executable detection... ");
...
    printf("\n");
```

exec_detector.c

### Declaration

- Access via inclusion of headers
  - Function prototypes
  - Global variables (e.g. `errno`)
  - Type definitions (e.g. `size_t`)
  - Macros (e.g., `O_RDONLY`)

### Definition

- Actual code via library
- Linked at compile-time
- Dynamically loaded at runtime

### Categories of functions

1. No privileged operation to perform
2. Always needs to request privileged operation from OS
   - Known as **system call**, or *syscall*
3. Sometimes needs to request privileged operation from OS

# C Standard Library

## 1. Regular functions

### Usage example

```
    char buf[4];
...
    memset(buf, 0, sizeof(buf));
```

exec_detector.c

- `memset()` only needs to have access to array `buf`
  - `buf` is already part of the memory (defined in stack)
  - No need for special operations

### Implementation example

- Generic C implementation

```
void *memset(void *s, int c, size_t count)
{
    char *xs = s;

    while (count--)
        *xs++ = c;

    return s;
}
```

# C Standard Library

## 2. (Always) privileged functions

### Usage example

```
    fd = open(argv[1], O_RDONLY);
...
    read(fd, buf, sizeof(buf));
    close(fd);
```

exec_detector.c

- Need for special privileges: e.g.,
  - Verify that file exists on a physical medium accessible by computer (e.g., hard-drive, SD card, network, etc.)
  - Check that current user has permission to open it with specified mode
  - Actually read file's data from physical medium
- Sensitive operations passed to OS
  - Accessible via *syscalls*

### Implementation example

- Specific to Linux

```
#include <unistd.h>
#include <sys/syscall.h>  // Syscall code numbers

ssize_t read(int fd, void *buf,
             size_t count)
{
    long r = syscall(SYS_read,
                     fd, buf, count);
    if (r < 0) {
        errno = -r;
        return -1;
    }
    return r;
}
```

- Specific to `x86_64` processors

```
static __inline long __syscall3(long n, long a1,
                                long a2, long a3)
{
    unsigned long ret;
    __asm__ __volatile__ ("syscall" : "=a"(ret)
                          : "a"(n), "D"(a1),
                            "S"(a2), "d"(a3)
                          : "rcx", "r11", "memory");
    return ret;
}
```

# C Standard Library

## 3. (Sometimes) privileged functions

### Usage example

```
    printf("Executable detection... ");
...
    printf("\n");
```
exec_detector.c

- `printf()` prints to `stdout` which is internally buffered
  - Flushed when buffer is full, or when encountering `\n`
- Flushing requires to OS to actually write the characters
  - Use of (syscall) function `write()`

### Printf vs write

```
    printf("Hello ");
    sleep(2);
    printf("world!\n");

    write(STDOUT_FILENO, "Hello ", 6);
    sleep(2);
    write(STDOUT_FILENO, "world!\n ", 7);
```
printf_write.c

```
$ ./printf_write
 <wait 2 sec>
Hello world!
Hello <wait 2 sec>world!
```

# System calls

## Definition

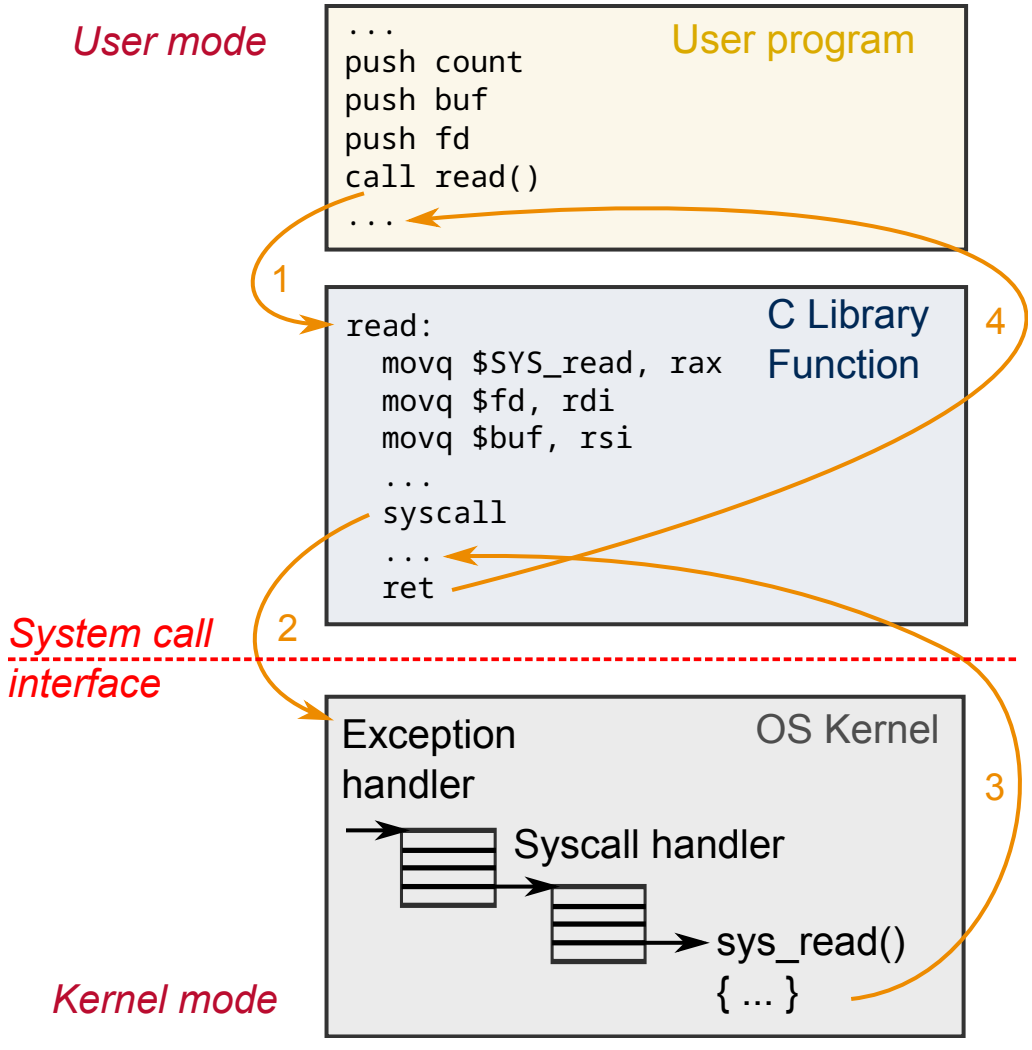
- Specific CPU instruction
- Immediate transfer of control to *kernel* code

## Purpose

- Secure API between user applications and OS kernel

## Main categories

- Process management
- Files and directories
- Pipes
- Signals
- Memory management

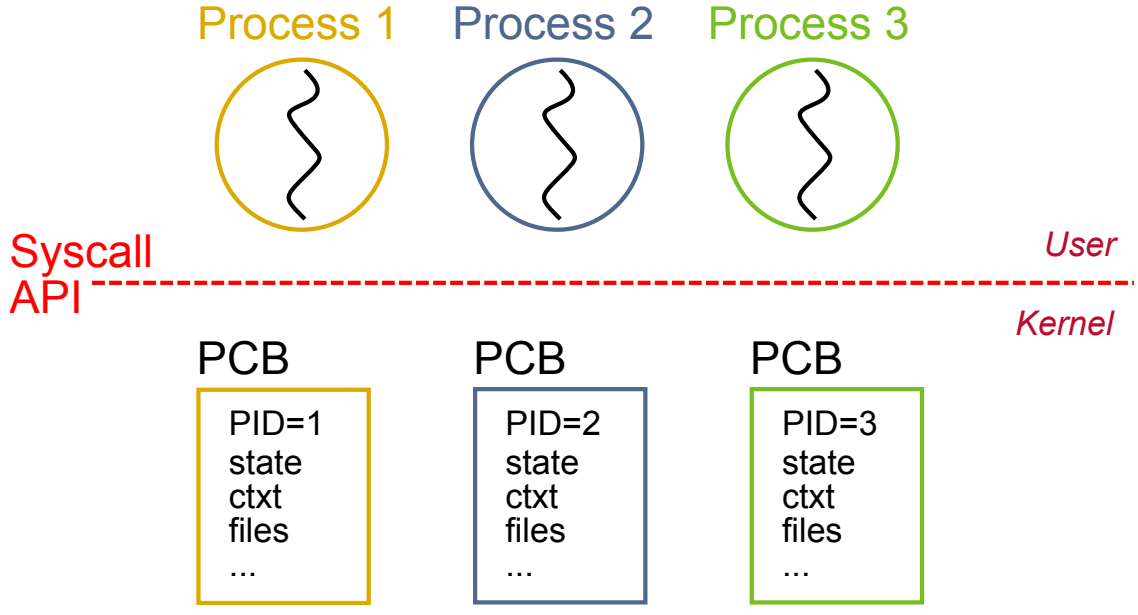


# Process management

## Definition of a process

**A process is a program in execution**

- Each process is identified by its *Process ID* (PID)
- Each process runs its own memory space
- Each process is represented in the OS by a *Process Control Block* (PCB)
  - Data structure storing information about process
  - PID, state, CPU register copies for context switching, open files, etc.



# Process management

## Main related functions/syscalls

- Process creation and execution
  - `fork()`: Create a new (clone) process
  - `exec()`: Change executed program within running process
- Process termination
  - `exit()`: End running process
  - `wait()/waitpid()`: Wait for a child process and collect exit code
- Process identification
  - `getpid()`: Get process PID
  - `getppid()`: Get parent process PID

# Process management

## fork()

- Running process gets cloned into a *child* process
- Child gets an (almost) identical copy of *parent*
  - Same open files, command line arguments, memory, stack, etc.
- Child "resumes" at the `fork()` as well

### Example

```c
int a = 42;

int main(int argc, char *argv[])
{
    int b = 23;

    printf("Hello world!\n");
    fork();
    printf("My favorite number is %d.\n",
           argc + a + b);

    return 0;
}
```

fork_101.c

```
$ ./fork_101
Hello world!
My favorite number is 66.
My favorite number is 66.
```

# Process management

## Distinguishing between parent and child

- `fork()` returns a value
  - *PID of the child* to the parent
  - *zero* to the child
  - `-1` to the parent in case of error

## Example

```c
int main(void)
{
    pid_t pid;

    pid = fork();
    if (pid > 0)
        printf("I'm the parent!\n");
    else if (pid == 0)
        printf("I'm the child!\n");
    else
        printf("I'm the initial process! "
               "But something went wrong...\n");

    printf("I'm here now, bye!\n");
    return 0;
}
```

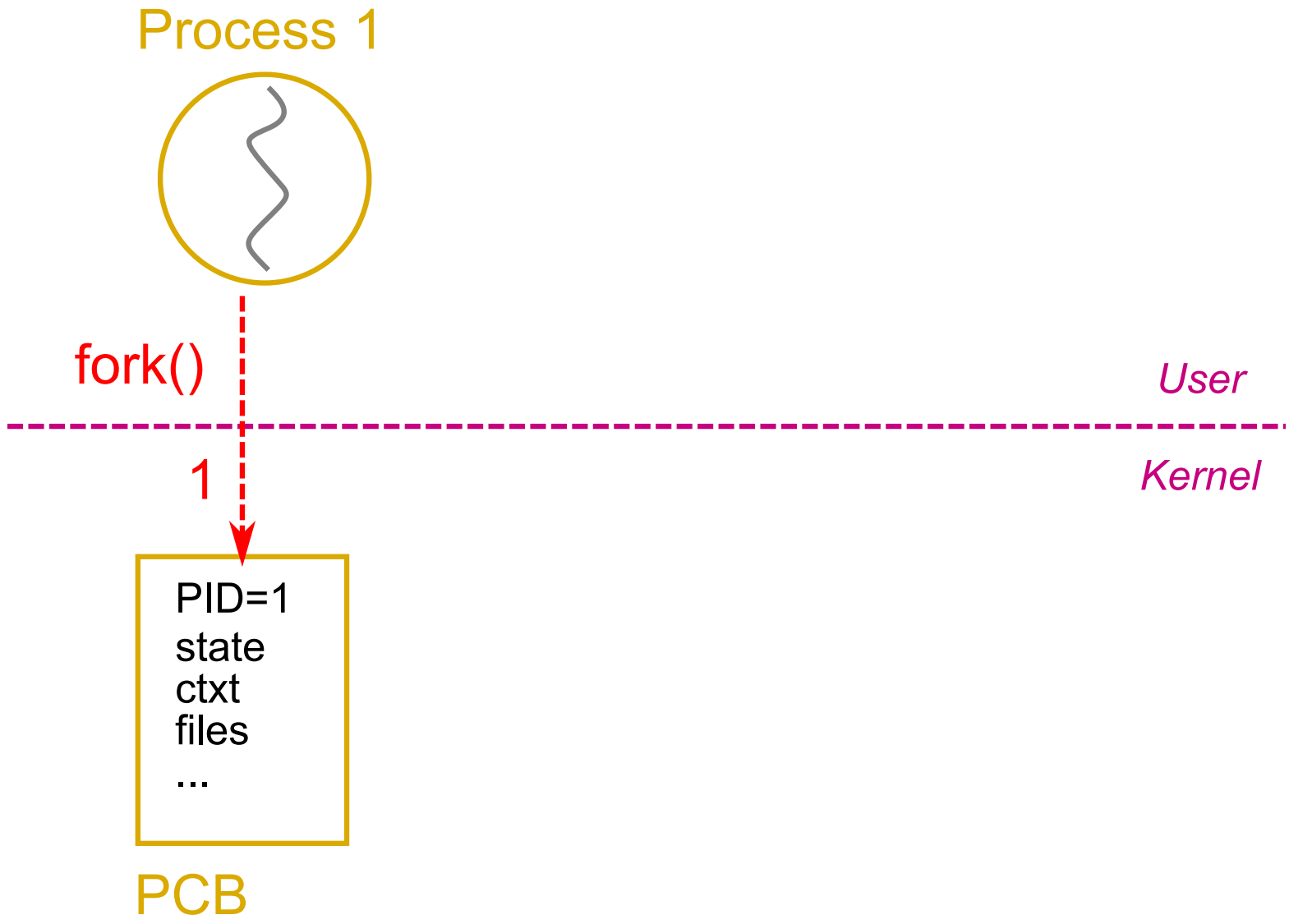
fork_201.c

```
$ ./fork_201
I'm the parent!
I'm here now, bye!
I'm the child!
I'm here now, bye!
```

- This exact output is not guaranteed
- Parent and child are independent processes
- Scheduling up to OS

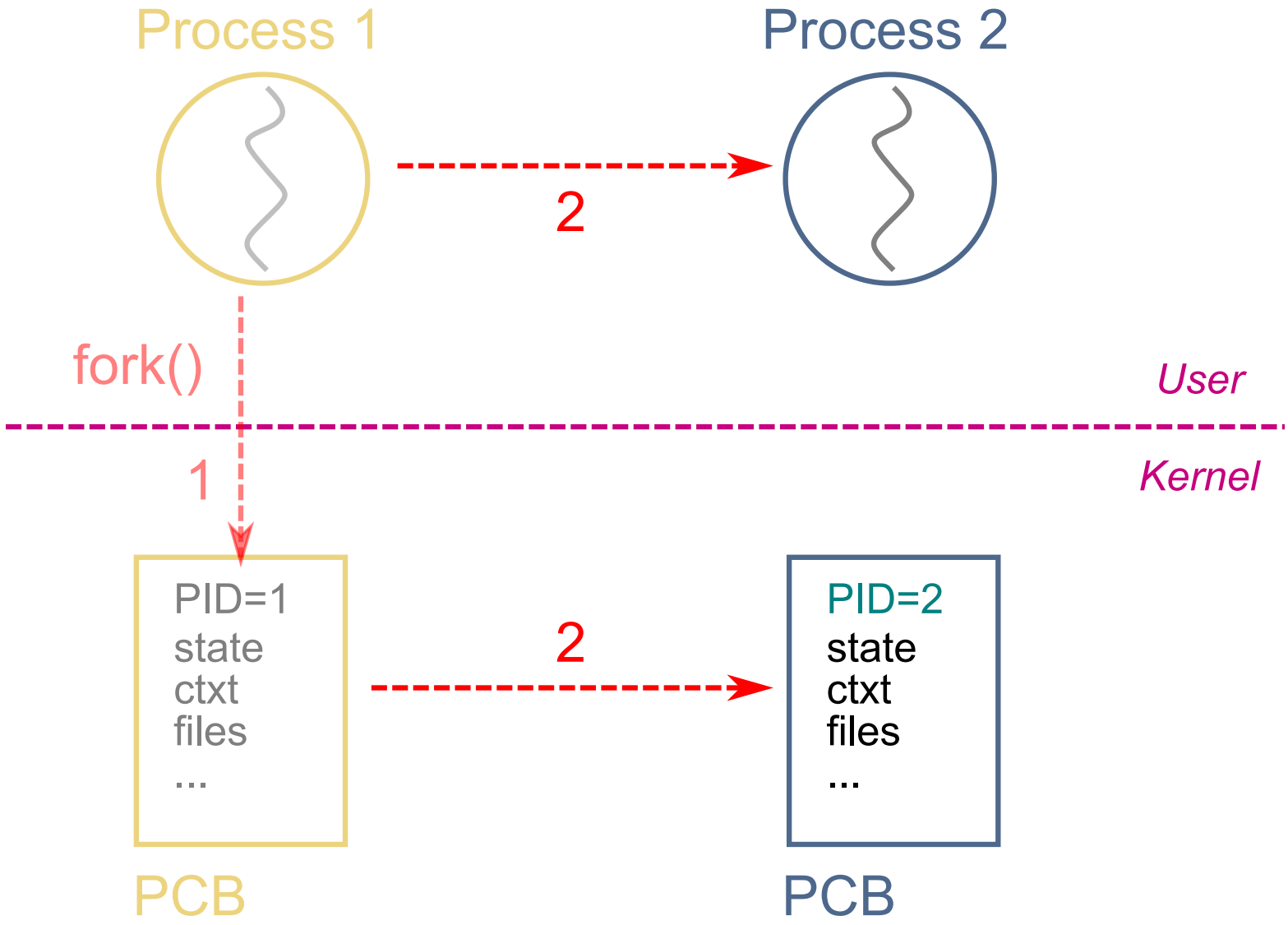
# Process management

## Fork illustrated (1/3)



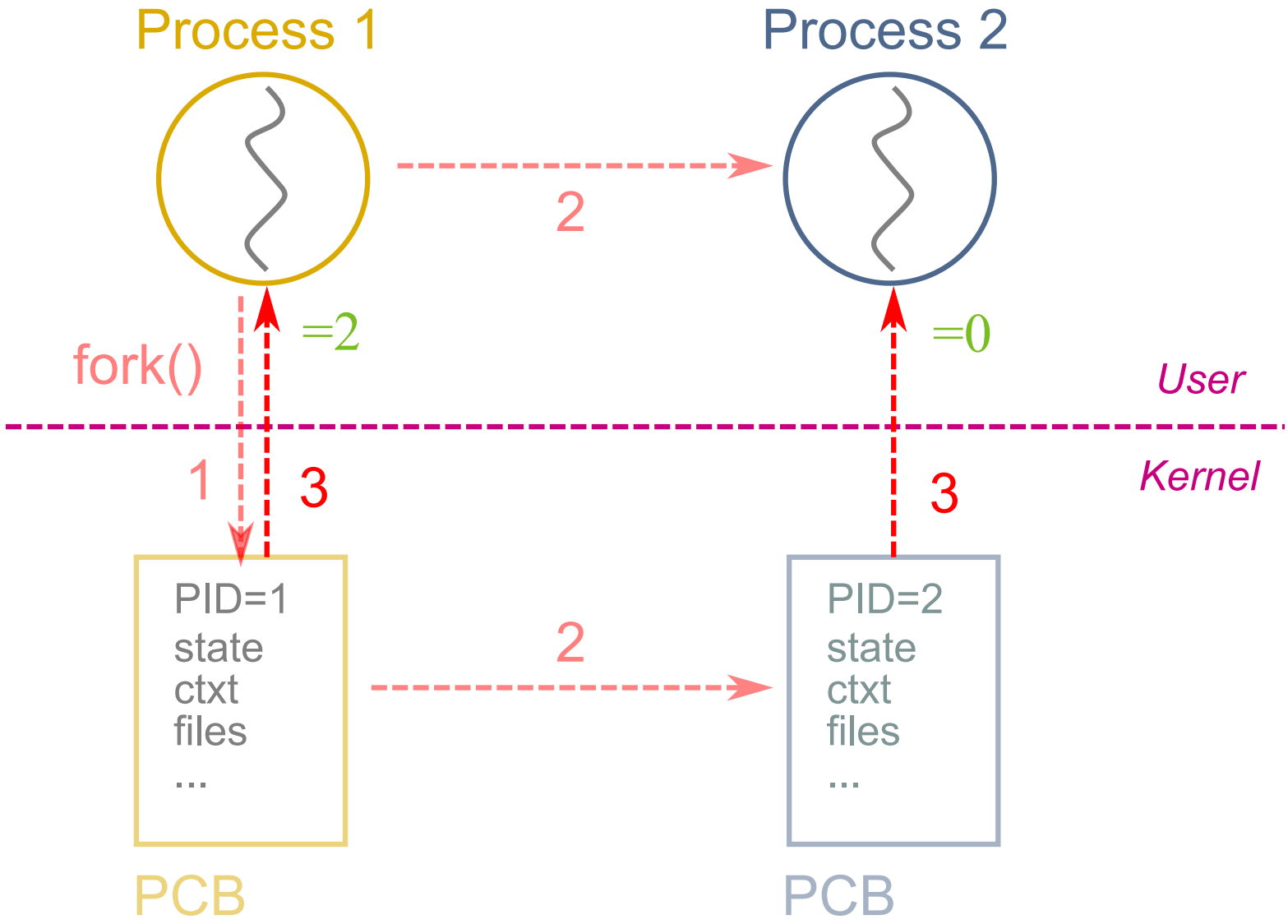
# Process management

## Fork illustrated (2/3)



# Process management

## Fork illustrated (3/3)



# ECS 150 - System Calls

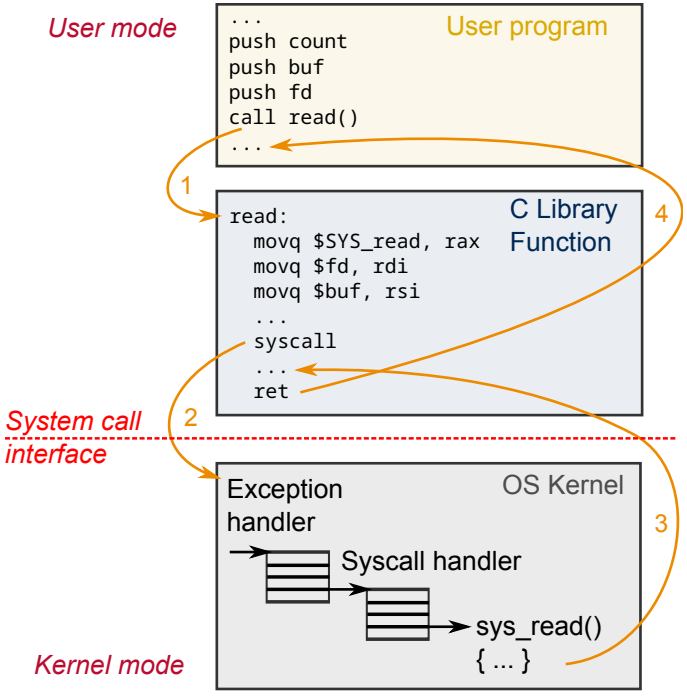
*Prof. Joël Porquet-Lupine*

UC Davis - 2020/2021

# Recap

## C standard library

- Non-privileged functions
  - E.g., `memset()`
- Always/sometimes privileged functions
  - E.g., `read()`/`printf()`
  - Require **system call**




## Syscall categories

- **Process management**
  - *A process is a program in execution*
  - PCB data structure
- Files and directories
- Pipes
- Signals
- Memory management

## `fork()`

```
pid_t fork(void);
```

- Clones *parent* process into *child* process
- Both return from fork call
- Return value distinguishes between parent and child

# Process management

## exec()

- Current *process* starts executing another *program*
- Family of functions, with slight variations
  - `exec[lv]p?e?()` (see man page for details)

## Example

```c
int main(void)
{
    char *cmd = "/bin/echo";
    char *args[] = { cmd, "ECS150", NULL};
    int ret;

    printf("Hi!\n");

    ret = execv(cmd, args);

    printf("Execv returned %d\n", ret);

    return 0;
}
```

execv.c

```
$ ./execv
Hi!
ECS150
```

- Call to `exec()` functions never returns if it succeeds!
- Otherwise returns `-1` and continues

# Process management

## exit()

- Termination of the current process
- Ability to return an *exit value*

### Example

- Exit at any time during execution

```
if (error)
    exit(1);
```

- Or return from `main()`

```
int main(void) {
    ...
    return 0;
}
```

- Libc transparently exits

```
exit(main(argc, argv));
```

### Usage

```
$ ls /
...
$ echo $?
0
```

```
$ ls /nodir
ls: cannot access '/nodir': No such file
or directory
$ echo $?
2
```

```
$ if [[ ! $(ls /nodir >& /dev/null) ]]; then echo "Expected..."; fi
Expected...
```

# Process management

## wait()/waitpid()

- `wait()` makes parent wait for *any* of its children to exit
  - Parent is blocked from execution in the meantime
- `waitpid()` enables more advanced options, such as
  - Specify PID of child to wait for
  - Don't block even if no children has returned

### Example

```c
pid = fork();
if (pid != 0) {
    /* Parent */
    int status;
    wait(&status);
    /* == waitpid(pid, &status, 0) */
    printf("Child returned %d\n",
           WEXITSTATUS(status));
} else {
    /* Child */
    printf("Will exit soon!\n");
    exit(42);
}
```

wait.c

```
$ ./wait
Will exit soon!
Child returned 42
```

- Output order guaranteed
- Scheduling constrained by blocking call

# Process management

## Putting it together: fork() + exec() + wait()

```c
int main(void)
{
    pid_t pid;
    char *cmd = "/bin/echo";
    char *args[] = { cmd, "ECS150", NULL};

    pid = fork();
    if (pid == 0) {
        /* Child */
        execv(cmd, args);
        perror("execv");
        exit(1);
    } else if (pid > 0) {
        /* Parent */
        int status;
        waitpid(pid, &status, 0);
        printf("Child returned %d\n",
               WEXITSTATUS(status));
    } else {
        perror("fork");
        exit(1);
    }

    return 0;
}
```

fork_exec_wait.c

```
$ ./fork_exec_wait
ECS150
Child returned 0
```

### system()

- Somewhat equivalent to what function `system()` does internally

```c
system("/bin/echo ECS150");
```

- But with a lot more control!

# Process management

## The shell

- A **shell** is a user interface to run commands in the terminal
- Typically makes heavy use of process-related functions

### Naive pseudo-implementation

```
while (1) {                             /* Repeat forever */
    char **command;

    display_prompt();                   /* Display prompt in terminal */
    read_command(&command);             /* Read input from terminal */

    if (!fork()) {                      /* Fork off child process */
        exec(command);                  /* Execute command */
        perror("execv");                /* Coming back here is an error */
        exit(1);
    } else {
        /* Parent */
        waitpid(-1, &status, 0);        /* Wait for child to exit */
    }
}
```

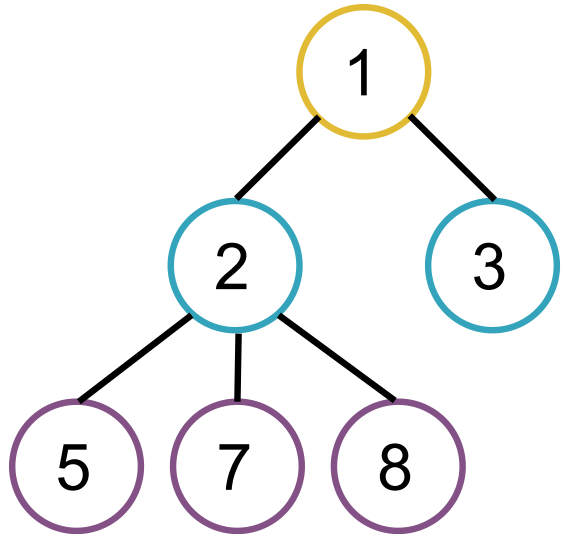
### Extra features

Background jobs (`&`), redirections (`<` and `>`) for connecting `stdin` or `stdout` of the child to files (instead of the terminal), pipes (`|`) for connecting `stdin` or `stdout` of the child to other processes, and many more.

# Process management

## getpid()/getppid()

- Notion of a (family) tree of processes
  - Only one parent per process
  - But possibly multiple children
- In Unix, `init` (PID=1) is ultimate ancestor
  - Only process created from scratch by kernel (and not by forking)
- `getpid()` returns process' PID
- `getppid()` return its parent's PID




## Example

```c
int main(void)
{
    if (fork() > 0)
        /* Forces parent to wait for child
         * to force scheduling order */
        wait(NULL);

    printf("My PID is %d\n", getpid());
    printf("My parent's PID is %d\n", getppid());

    return 0;
}
```

getpid.c

```
$ ./getpid
My PID is 406782
My parent's PID is 406781
My PID is 406781
My parent's PID is 162474
```

```
$ echo $$
162474
```

# System calls

Process management

Files and directories

Pipes

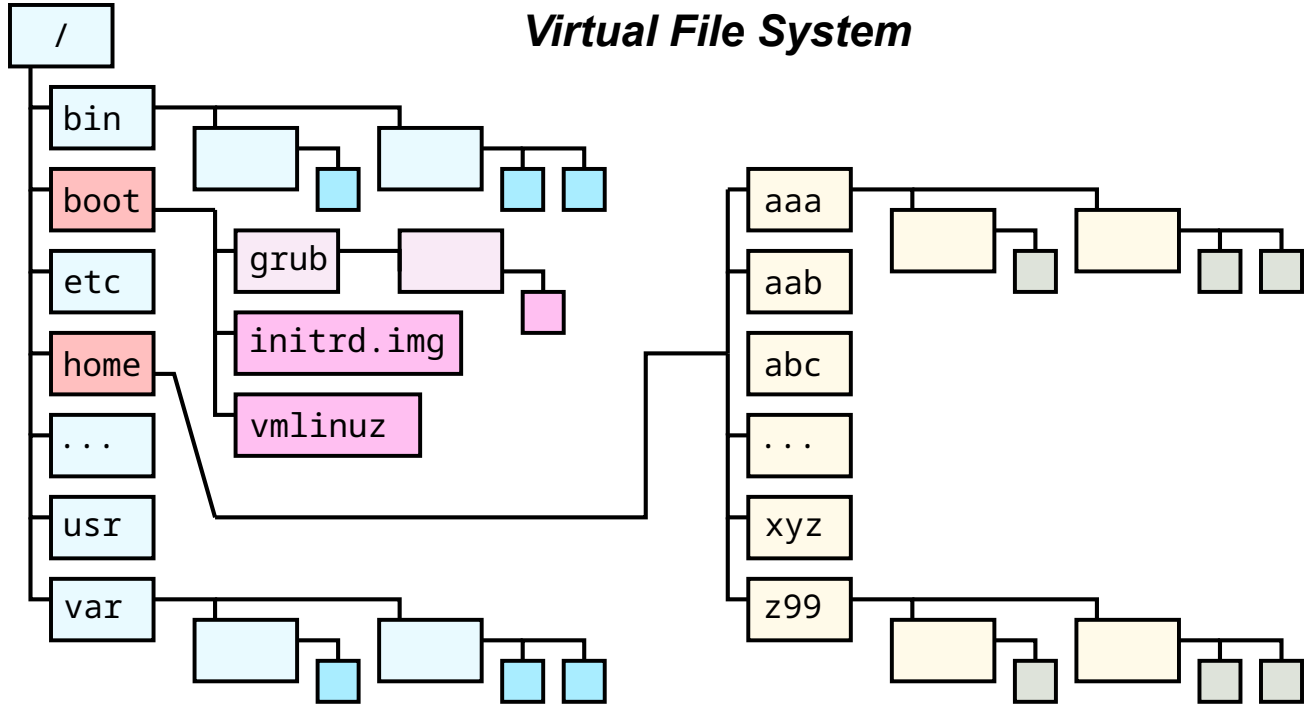
Signals

Memory management

# Files and directories

## Concepts

- Files and directories in tree-like structure called **Virtual File System**
  - Internal nodes are directories, leaf nodes are files
- Every directory contains a list of filenames
- Every file contains an array of bytes




- VFS can aggregate files and directories from various physical media (local hard-drive, remote network share, etc.)

# Files and directories

## Main related functions/syscalls

- File interaction
  - `open()`: open (create) file and return file descriptor
  - `close()`: close file descriptor
  - `read()`: read from file
  - `write()`: write to file
  - `lseek()`: move file offset
- File descriptor management
  - `dup()`/`dup2()`: duplicate file descriptor
- File characteristics
  - `stat()`/`fstat()`: get file status
- Directory traversal
  - `getcwd()`: get current working directory
  - `chdir()`: change directory
  - `opendir()`: open directory
  - `closedir()`: close directory
  - `readdir()`: read directory

# Files and directories

## Basic file interaction

- `open()` returns a **file descriptor** (*FD*), used for all interactions with file
  - Closed by `close()` when done with file
- `read()`/`write()` operations are sequential, tracked by file offset
- `lseek()` can manipulate current file offset

### Example

```c
#include <fcntl.h>
...
int main(void)
...
    int fd;
...
    fd = open("file_101.c", O_RDONLY);

    read(fd, &c, 1);
    printf("%c\n", c);
    read(fd, &c, 1);
    printf("%c\n", c);

    lseek(fd, -2, SEEK_END);
    read(fd, &c, 1);
    printf("%c\n", c);

    close(fd);
```

file_101.c

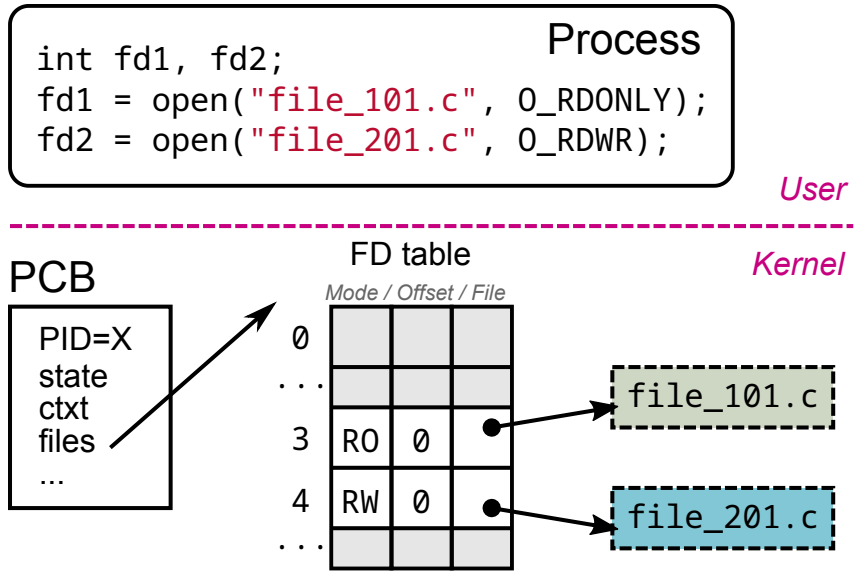
```
$ ./file_101
#
i
}
```

# Files and directories

## File descriptors

### Definition

- Table of *open files* per process
  - Part of PCB
  - (Duplicated upon forking)
- FDs are simple indexes in the table




### Example

```c
int fd1, fd2;

fd1 = open("file_101.c", O_RDONLY);
fd2 = open("file_201.c", O_RDWR);

printf("fd1 = %d\n", fd1);
printf("fd2 = %d\n", fd2);
```

file_201.c

```
$ ./file_201
fd1 = 3
fd2 = 4
...
```

### Allocation

- `open()` always returns *first available* FD

```c
close(fd1);
fd1 = open("file_201.c", O_WRONLY);
printf("fd1 = %d\n", fd1);
```

file_201.c

```
$ ./file_201
...
fd1 = 3
```

# ECS 150 - System Calls

*Prof. Joël Porquet-Lupine*

UC Davis - 2020/2021

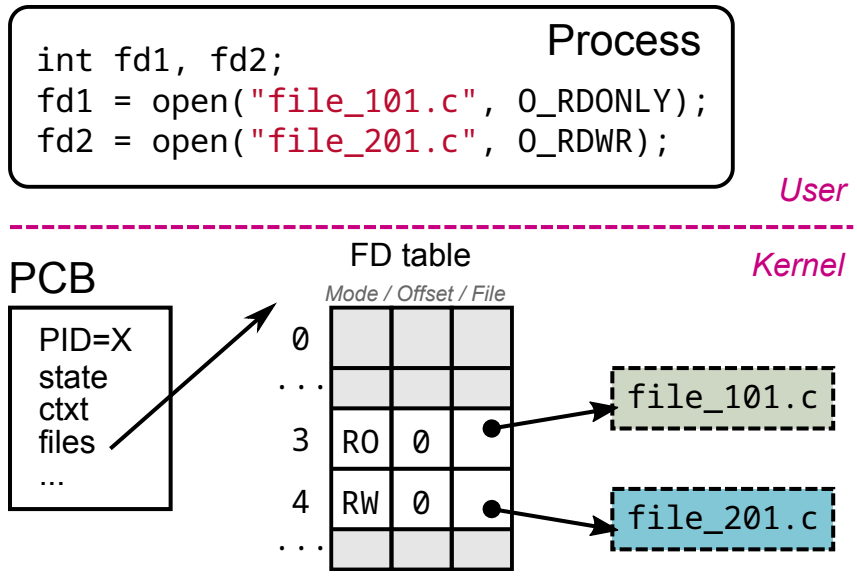
# Recap

## Process management

- `fork()`
  - Clone process
- `exec()`
  - Execute different program inside current process
- `wait()`
  - Parent for children processes to terminate
  - Collect return value

## Files and directories

- File descriptors

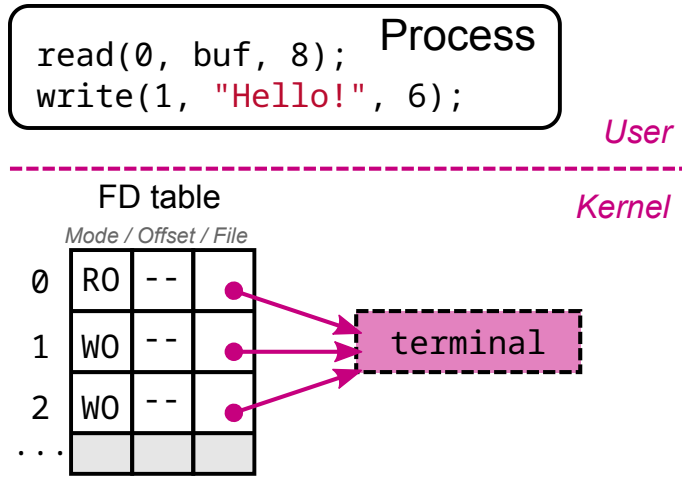


# Files and directories

## Standard streams

Initially, three open file descriptors per process

- `0`: standard input (`STDIN_FILENO`)
- `1`: standard output (`STDOUT_FILENO`)
- `2`: standard error (`STDERR_FILENO`)




### Example

```c
write(STDOUT_FILENO, "Hello ", 6);
write(1, "Hello ", 6);
fprintf(stdout, "Hello ");
printf("Hello\n");

write(STDERR_FILENO, "World ", 6);
write(2, "World ", 6);
fprintf(stderr, "World\n");
```

standard_fds.c

```
$ ./standard_fds
Hello Hello Hello Hello
World World World
```

## Redirections

- Can connect standard streams to other targets than the terminal

```
$ ./standard_fds > /dev/null
World World World
$ ./standard_fds 2> /dev/null | tr H J
Jello Jello Jello Jello
```

```
$ ./standard_fds >& myfile.txt
$ cat myfile.txt
Hello Hello World World World
Hello Hello
```

# Files and directories

## File descriptor manipulation

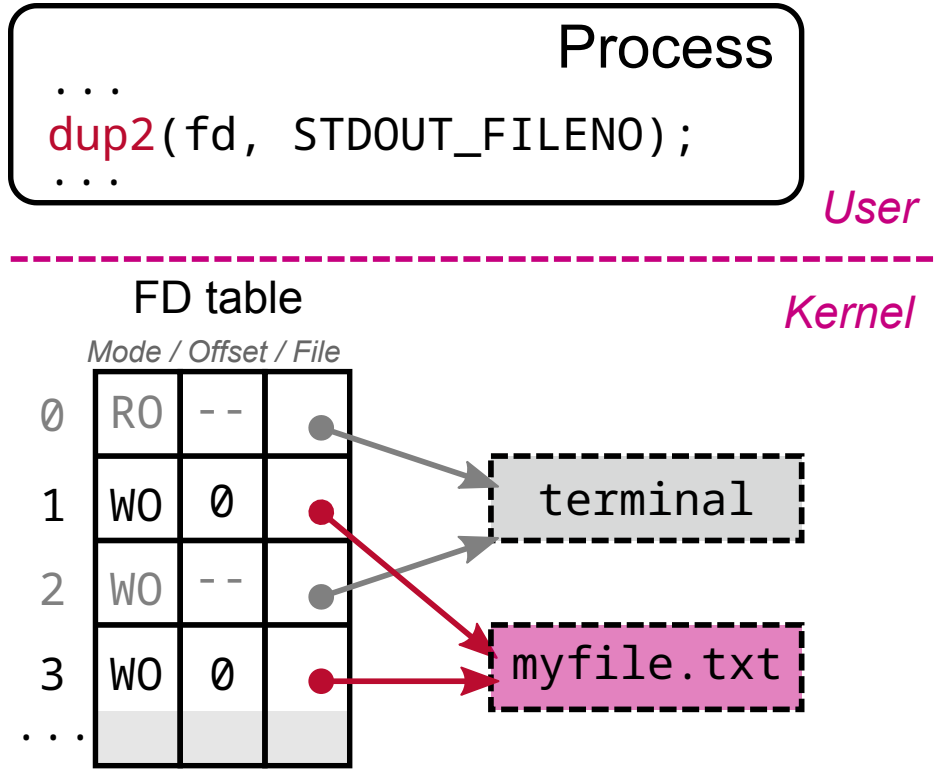
- `dup2()` replaces an open file descriptor with another
  - Avoid using *deprecated* `dup()`

### Example

```c
int main(void)
{
    int fd;

    printf("Hello #1\n");

    fd = open("myfile.txt",
              O_WRONLY | O_CREAT,
              0644);
    dup2(fd, STDOUT_FILENO);
    close(fd);

    printf("Hello #2\n");

    return 0;
}
```

dup2.c

```
$ ./dup2
Hello #1
$ cat myfile.txt
Hello #2
```



# Files and directories

## File characteristics

- `stat()` returns information about a file, from a filename
- Same with `fstat()`, but from an FD

### Example

```
struct stat {
    dev_t     st_dev;         /* ID of device containing file */
    ino_t     st_ino;         /* Inode number */
    mode_t    st_mode;        /* File type and mode */
    nlink_t   st_nlink;       /* Number of hard links */
    uid_t     st_uid;         /* User ID of owner */
    gid_t     st_gid;         /* Group ID of owner */
    dev_t     st_rdev;        /* Device ID (if special file) */
    off_t     st_size;        /* Total size, in bytes */
    blksize_t st_blksize;     /* Preferred blocksize for I/O */
    blkcnt_t  st_blocks;      /* Number of 512B blocks allocated */
    struct timespec st_atim;  /* Time of last access */
    struct timespec st_mtim;  /* Time of last modification */
    struct timespec st_ctim;  /* Time of last status change */
};
```

```
int main(int argc, char *argv[])
{
    struct stat sb;

    stat(argv[1], &sb);

    printf("File type:          ");
    switch (sb.st_mode & S_IFMT) {
        case S_IFDIR:  printf("directory\n");    break;
        case S_IFREG:  printf("regular file\n"); break;
        default:       printf("other\n");        break;
    }
    printf("Mode:               %lo (octal)\n",
           (unsigned long) sb.st_mode);
    printf("File size:          %lld bytes\n",
           (long long) sb.st_size);
    printf("Last file access:   %s",
           ctime(&sb.st_atime));

    return 0;
}
```

stat.c

```
$ ./stat stat.c
File type:          regular file
Mode:               100644 (octal)
File size:          644 bytes
Last file access:   Fri Sep 18 16:32:02 2020
$ ./stat .
File type:          directory
Mode:               40755 (octal)
File size:          4096 bytes
Last file access:   Fri Sep 18 11:59:49 2020
```

# Files and directories

## Directory traversal

- `getcwd()`/`chdir()` to access the *current working directory*
- `opendir()`/`closedir()`/`readdir()` to access the entries of a directory

### Example

```c
int main(int argc, char *argv[])
{
    char cwd[PATH_MAX];
    DIR *dirp;
    struct dirent *dp;

    getcwd(cwd, sizeof(cwd));
    printf("Change CWD from '%s' to '%s'\n",
           cwd, argv[1]);
    chdir(argv[1]);

    dirp = opendir(".");
    while ((dp = readdir(dirp)) != NULL)
        printf("Entry: %s\n", dp->d_name);
    closedir(dirp);

    return 0;
}
```

dir_scan.c

```
$ pwd
/home/jporquet
$ ./dir_scan /
Change CWD from '/home/jporquet' to '/'
Entry: ..
Entry: lib
Entry: home
Entry: etc
Entry: root
[...]
Entry: proc
Entry: tmp
Entry: dev
Entry: bin
Entry: sbin
Entry: mnt
Entry: .
Entry: usr
```

# System calls

Process management

Files and directories

Pipes

Signals

Memory management

# Pipe

## Definition

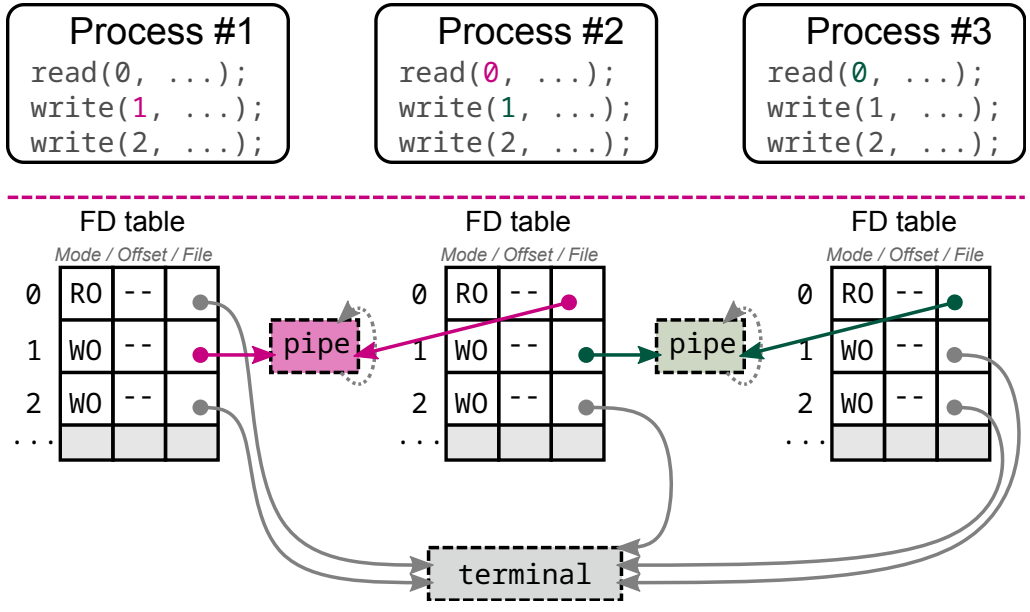
- Inter-process communication (IPC)
- Pipeline of processes chained via their standard streams
  - `stdout` of one process connected to `stdin` of next process

## Example

```
$ du -sh * | sort -h -r | head -3
1.2G    ecs150
555M    ecs36c
386M    ecs30
```

## Details

- Internally implemented as anonymous files
  - Circular memory buffer of fixed size
  - Accessible via file descriptors and regular read/write transfers
- Processes run *concurrently*, implicitly synchronized by communication



# Pipe

## `pipe()`

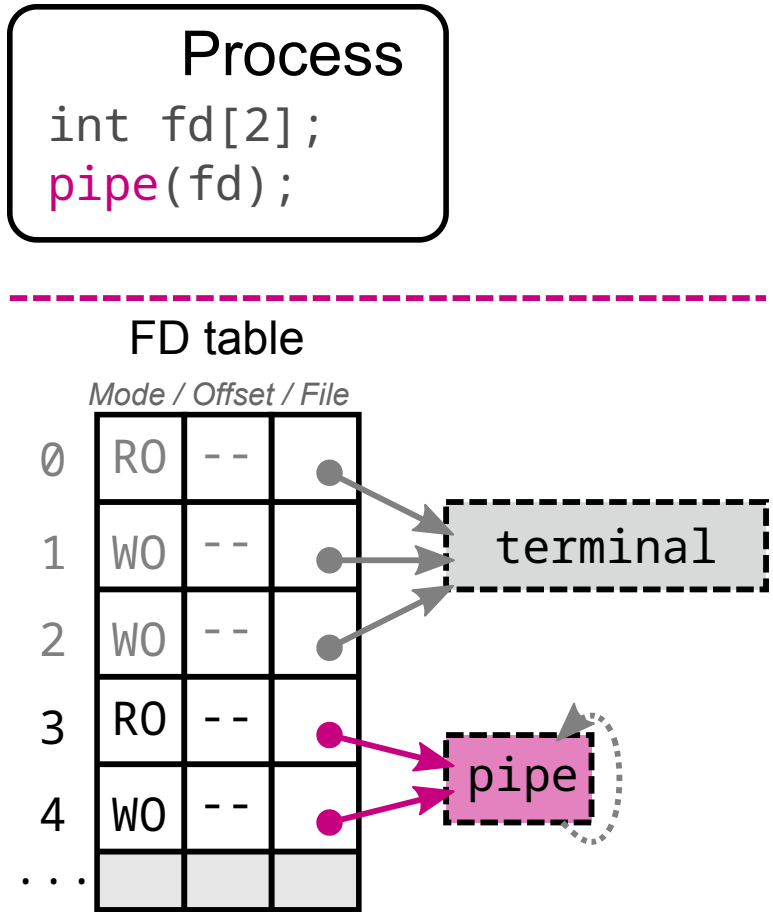
- Create a pipe and return two file descriptors via array
  - [0] for reading access, [1] for writing access

### Example

```c
int main(void)
{
    int fd[2];
    char send[7] = "Hello!";
    char recv[7];

    pipe(fd);
    printf("fd[0] = %d\n", fd[0]);
    printf("fd[1] = %d\n", fd[1]);

    write(fd[1], send, 7);

    read(fd[0], recv, 7);
    puts(recv);

    return 0;
}
```

pipe.c

```
$ ./pipe
fd[0] = 3
fd[1] = 4
Hello!
```



# Pipe

## Process pipeline example

- Pseudo-code setting up `process1 | process2`

```
void pipeline(char *process1, char *process2)
{
    int fd[2];

    pipe(fd);
    if (fork() != 0) {  /* Parent */
        /* No need for read access */
        close(fd[0]);
        /* Replace stdout with pipe */
        dup2(fd[1], STDOUT_FILENO);
        /* Close now unused FD */
        close(fd[1]);
        /* Parent becomes process1 */
        exec(process1);
    } else {            /* Child */
        /* No need for write access */
        close(fd[1]);
        /* Replace stdin with pipe */
        dup(fd[0], STDIN_FILENO);
        /* Close now unused FD */
        close(fd[0]);
        /* Child becomes process2 */
        exec(process2);
    }
}
```

# System calls

Process management

Files and directories

Pipes

Signals

Memory management

# Signals

## Definition

- Form of inter-process communication (IPC)
- Software notification system
  - From process' own actions: e.g., `SIGSEGV` *(Segmentation fault)*
  - From external events: e.g., `SIGINT` *(Ctrl-C)*
- About 30 different signals (see `man 7 signal`)

### Default action

In case process does not define specific signal handling

- Terminate process: e.g., `SIGINT`, `SIGKILL`*
- Terminate process and generate core dump: e.g., `SIGSEGV`
- Ignore signal: e.g., `SIGCHLD`
- Stop process: e.g., `SIGSTOP`*
- Continue process: e.g., `SIGCONT`

### Handling or ignoring

- Possible to change default action (but not for all signals *!)
- Ignore signals
  - Mask of blocked signals
- Set signal handlers
  - Function to be run upon signal delivery

# Signals

## Main related functions/syscalls

- Sending signals
  - `raise()`: Send signal to self
  - `kill()`: Send signal to other process
  - `alarm()/setitimer()`: Set timer for self
    - Receive signal (`SIGALRM`/`SIGVTALRM`) when timer is up
- Blocking signals
  - `sigprocmask()`: Examine or change signal mask
  - `sigpending()`: Examine pending blocked signals
- Receiving signals
  - `sigaction()`: Map signal handler to signal
    - Also `signal()` but usage not recommended
  - `pause()`: Suspend self until signal is received

# Signals

## Example

```c
void alarm_handler(int signum)
{
    printf("\nBeep, beep, beep!\n");
}

int main(void)
{
    struct sigaction sa;
    sigset_t ss;

    /* Ignore Ctrl-C */
    sigemptyset(&ss);
    sigaddset(&ss, SIGINT);
    sigprocmask(SIG_BLOCK, &ss, NULL);

    /* Set up handler for alarm */
    sa.sa_handler = alarm_handler;
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = 0;
    sigaction(SIGALRM, &sa, NULL);

    /* Configure alarm */
    printf("Alarm in 5 seconds...\n");
    alarm(5);

    /* Wait until signal is received */
    pause();

    /* Bye, ungrateful world... */
    raise(SIGKILL);

    return 0;
}
```

signal.c

```
$ ./signal
Alarm in 5 seconds...
 ^C^C^C^C
Beep, beep, beep!
zsh: killed     ./signal
```

# System calls

Process management

Files and directories

Pipes

Signals

Memory management

# Memory

## Division of labor

### User C library

- `malloc()`/`free()` for dynamic memory allocation
  - **Heap** memory segment (at the end of **data** segment)
- Fine-granularity management only
  - When heap is full, syscall to kernel to request for more

### OS/Kernel

- Memory management at "page" level
- Allocation of big chunks (many pages) to user library

## Related functions/syscalls

### `sbrk()`/`brk()`

- Increase size of data segment
- Old way of allocating heap space
- Legacy function now

### `mmap()`

- Map pages of memory in process' address space
- Can also map a file's contents
- Extremely versatile and powerful function