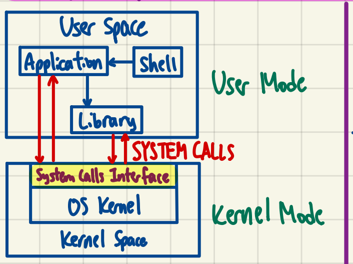
UNIT 2: UNIX+OS CONCEPTS

UNIT 3: FILE I/O (PART I)

WEEK 2 Shun/翔海 (@shun4midx)

OVERALL STRUCTURE — SYSTEM VS FUNCTION CALLS

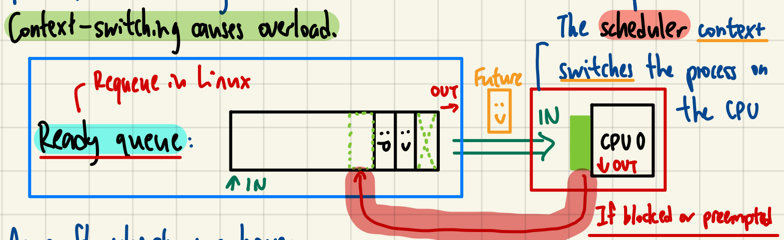


	System Call	Function Call
Description	Application calls to functions provided by OS kernel	Program calls to predefined functions (eg: defined in library)
Behavior	Causes switches from user to kernel (space/mode)	Cause no space/mode switches
Speed	Slow	Fast

PROCESS MANAGEMENT (Time-sharing: Processes can "share" CPU time)

Context-switch: The process of storing the execution context of a process, and restoring the execution context of a new process

Context-switching causes overload.



As a flowchart, we have:

New process created to run → The process is ready to run → The process is running → Exit → Clean up by OS

The process CANNOT RUN (Do context switching)

FILE/DIRECTORY MANAGEMENT

In Unix, everything is a file.

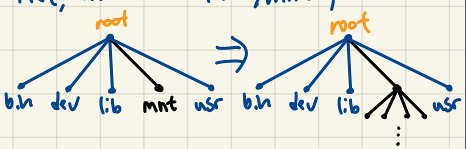
Root directory: Top-most dir. "/"

Home directory: Directory when you log in

Absolute path: /x/y/z (begin at root) Relative path: x/y/z

THE MOUNT COMMAND

Here, we mount in "/mnt",



SYNTAX

s=mkdir(name, code) create dir

s=rmdir(name) remove dir

s=link(name1, name2) create new name2→name1

s=unlink(name) remove dir entry

s=mount(special, name, flag) mount

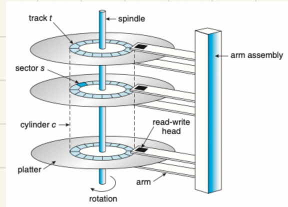
s=umount(special) unmount

Remark:

Pipe ("|") sends the output of one process as the input for another.

FILE I/O (BUFFERED VS UNBUFFERED)

Disk structure:



Buffered (standard) I/Os: Functions accumulate results in intermediate (within a process) buffers, not making system calls each time (e.g. fread/fwrite)

Unbuffered I/Os: Functions invoke system calls to the kernel each time (e.g. read(), write() in Unix)

P ⇄ OSK ⇄ S

Of course, buffered is faster due to no system calls. Unbuffered would require updating process, OS kernel, and storage as needed. Even with buffer cache in OS kernel, it takes a long time. Besides, buffer cache needs to be maintained carefully alongside storage when write() is called.

SYNTAX — FILE DESCRIPTORS

File descriptor: A nonnegative integer ∈ [0, OPEN_MAX−1], where OPEN_MAX = max files a process can open at once

↳ They are per-process, so diff processes may share file descriptors

In <unistd.h>, we have 0=STDIN_FILENO, 1=STDOUT_FILENO, 2=STDERR_FILENO by POSIX.1 standard. More are stored in a file table.

```
char buf[100];
E.g. while(n=read(STDIN_FILENO, buf, 100))!=0) {
        write(STDOUT, buf, n);}
```

FILE I/O CODE SYNTAX (Can end w/ exit(0);)

```
#include <fcntl.h>
int open(const char* path, int oflag, ..., /*mode_t mode*/);
int openat(int fd, const char* path, int oflag, ..., /*mode_t mode*/);
int close(int fd);
```

path: absolute path ⇒ susceptible to TOCTTOU attack

fd: base of relative path (file dir). If AT_FDCWD, then fd is the curr working dir

(Rmk: obsolete but creat(const char* path, mode_t mode); exists, it creates a file if DNE)

DIFFERENT OFLAGS

Must: O_RDONLY, O_WRONLY, O_RDWR (RD=read, WR=write)

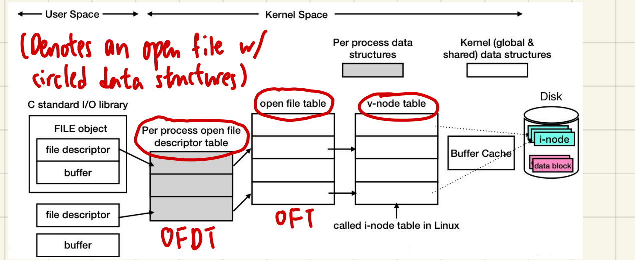
Optional: O_APPEND (append at EOF), O_TRUNC (trunc file size to 0), O_CREAT (create if DNE), O_NONBLOCK, O_SYNC,

(Can OR w/ must) O_DSYNC (data sync), O_RSYNC

Mode: S_I [4 char] ← order: R(read), W(write), X(execute) ← choose 1 or all three + USR, GRP, OTH (other) ← abbrev to fit length

UNIX KERNEL SUPPORT FOR FILE/IO

(Denotes an open file w/ circled data structures)



OFDT: One entry per file descriptor (file desc flag, ptr to OFT entry)

OFT: File status flag, curr file offset, ptr to V-node table

V-node (i-node table): V-node info, ptr to i-node

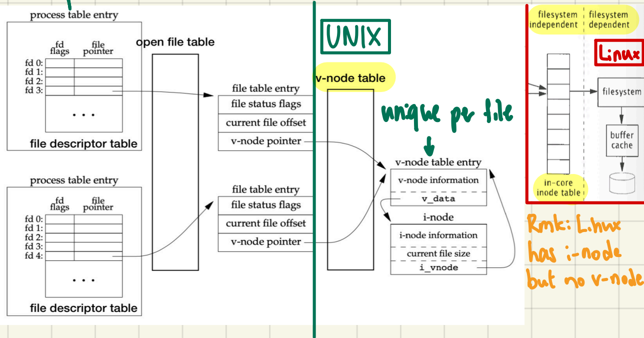
V-node: In-memory data structure for each open file

↳ Info: File type, ptr to func that operate the file

i-node: Stored physically on storage device and in memory

↳ Contains metadata (owner, size, device, protection info, ...)

↳ OS kernel reads i-node from disk to memory when file is opened



Rmk: Linux has i-node but no v-node