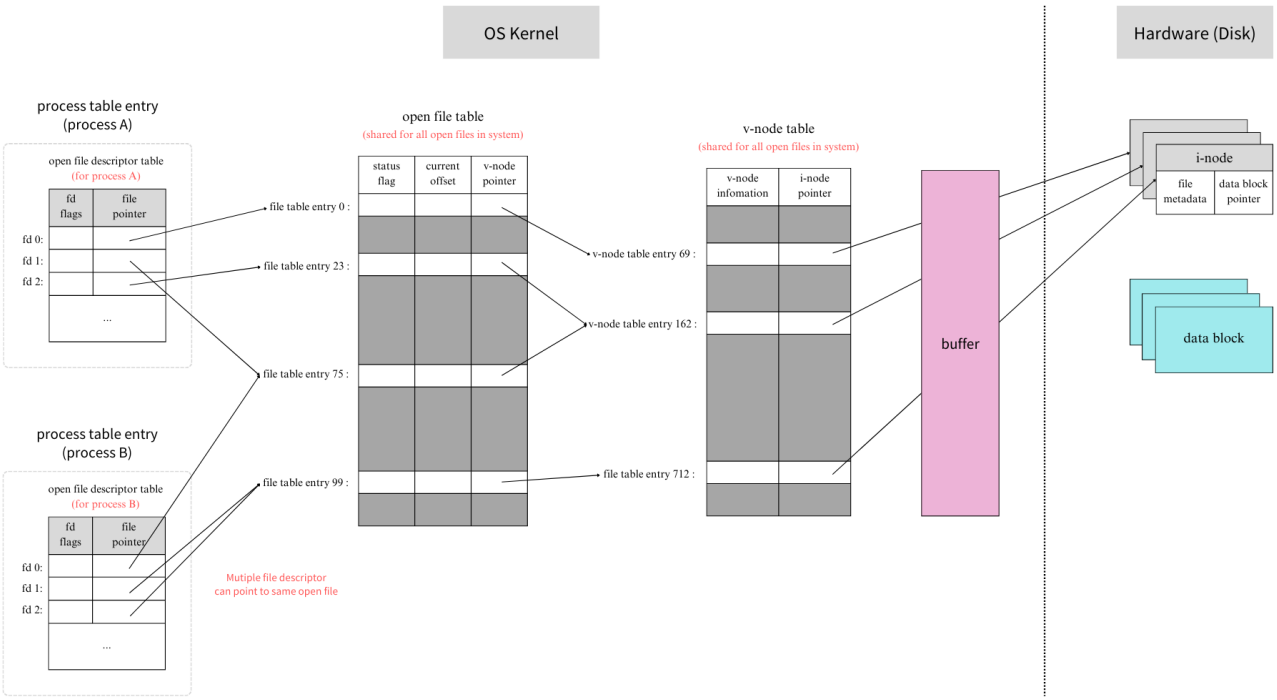

Machine Bootstrapping:
1. machine power on
2. firmware: *software* from 硬碟 to RAM
3. boot loader: *OS* from 硬碟 to RAM
4. OS: *init program* from filesystem to RAM
5. OS: go to user space, run *init process*

COW: sharing same memory address, until someone modify, then copy memory
Race Condition: 2↑ process modify shared data, and result affect by actions' order
Deadlock: multiple process waiting each other's action to continue, thus neither continue.
stdout buffer: connect to terminal: line buffered ; otherwise: fully buffered

Synchronization

```
TELL_WAIT();
if ((pid = fork()) < 0) {
        err_sys("fork error");
} else if (pid == 0) {
        TELL_PARENT(getppid());
        WAIT_PARENT();
        /* child continues */
        exit(0);
} else {
        TELL_CHILD(pid);
        WAIT_CHILD();
        /* parent continues */
        exit(0);
}
```

fork v.s. exec

	fork	exec
locks	diff	same
environment	inherit	change by *env*
pid	diff	same
memory	COW	replace

Zombie P: process is terminated, but parent didn't wait it → process in zombie state
*too many will cause no pid avaliable
Orphan P: process still running, but parent terminate → set parent to pid 1
Pipes:
- data is removed when read out
- not a real file, half-duplex, only work between parent-child.
- block: read from empty, write to full
- write end closed: `read` return 0
- read end closed: `write` trigger SIGPIPE

FIFOs:
- named pipe
- file type: FIFO (no content)
- work between any process
- `open` with O_RDONLY for read, O_WRONLY for write
- `open` without O_NONBLOCK: block until another end open

with: r-end: return ; w-end: error if no reader

function	effect
`atexit`	register function to be exit handler *same function can register multiple times *exit handler is called in reverse order
`exec`	replace process with another program `argv[] = {"ls", "-l", NULL}; execve("/bin/ls", argv, envp);`
`exit`	clean (call exit handler) then `_exit`
`waitpid`	`wait` but for specific *pid* *pid*: **=0**: child gid = caller's gid ; **>0**: child = pid **=-1**: any child ; **<-1**: child gid = pid
`wait`	wait for child's state change **block:** all child running **return status value:** some child state change **return error:** no child
`pipe`	make pipe. fd[0] read ; fd[1] write
`_exit`	terminate process
`fork`	create child. return: **child**: 0 ; **parent**: child pid
`getpid`	get pid. ; `getppid`: get parent pid

function	effect
`access`	check real user access. *mode*: R_OK, W_OK, X_OK
`create`	open or create w-only file.
`dup2`	`close` *fd2*, then copy *fd* to *fd2*.
`dup`	copy *fd* to lowest unopened fd.
`fdatasync`	write file data to disk, return after finish
`fnctl`	**F_GETFL**: get status flag ; **F_SETFL**: set status flag
`fsync`	write data & metadata to disk, return after finish
`getcwd`	store cwd's path to buf
`link`	create (h-link) dir entry → i-node, link count+1
`lseek`	modify offset. **SEEK_SET**:0 ; **SEEK_CUR**:current offset ; **SEEK_END**:EOF
`pread,pwrite`	r, w from specific offset. doesn't change offset.
`readlink`	read content of s-link
`rmdir`	remove dir. failed if not empty

function	effect	function	effect
`stat,lstat`	get file *stat*	`symlink`	create s-link → file(不一定要存在)
`truncate`	多的砍掉、不足補0	`sync`	queuing buffer data and return immediately
`umask`	turn off specified permission. per process.	`unlink`	remove dir entry → i-node, link count-1
`vfork`	`fork` but parent blocking until child `exec` or `exit`.	`utime`	change access, modification time

```
fd_set readSet;
FD_ZERO(&readSet);
FD_SET(fd1, &readSet);
FD_SET(fd2, &readSet);
while (1) {
        fd_set tmpReadSet = readSet;
        int readyfds = select(maxfd + 1, &tmpReadSet, NULL, NULL,
&timeout);
        if (readySockets > 0) {
                if (FD_ISSET(fd1, &tmpReadSet))
                        // deal with fd1
                if (FD_ISSET(socket2, &tmpReadSet))
                        // deal with fd1
        }
}
FD_CLR(fd1, &readSet);
```

File Properties

permission bit: 4=`s`(set-uid) 2=`s`(set-gid) 1=`t`(sticky) | 4=`r` 2=`w` 1=`x`
set-uid bit: set process's effective uid to file owner's real uid
sticky bit: to remove, rename file in dir:
user has `w` for dir **and** (user owns file **or** user owns dir **or** superuser)
ownership of file: when `open` `create`, set to effective ID of process
===
file size: 檔案的實際大小為 `st_size`, 占用的空間為 `st_blksize` * `st_blocks`
===
i-node : fixed size entry, contains file's info
i-node link count: # dir entries point to i-node
dir entry: contains filename and i-node location
hard-link: link between dir entry and i-node
no file type, can't connect to dir, only same filesystem
symbolic-link: actual file contains pathname of another file
===
access time: 被 `read`/`write` 的時間
modification time: 檔案內容被更改的時間
change-status time: i-node 被更改的時間
===
working dir: per process. decide by `/etc/passwd`
device special file: each represent a device. major: 管理該 device 的 driver ; minor: 用來區別同個 major 底下的不同 device
device type: block: random access data blocks ; character: sequential access data bytes ; network: use networking subsystem, not d-s-file

File Lock

shared read lock: multiple readers but no writers
exclusice write lock: single writer and no reader
lock release: process terminate, fd close

```
// Set flock
struct flock lock;
lock.l_type = <F_RDLCK, F_WRLCK, F_UNLCK>
lock.l_whence = <SEEK_SET, SEEK_CUR, SEEK_END>
lock.l_start = <offset relative to whence>
lock.l_len = <length to lock, 0 for entire>
// Acquire the lock
if (fcntl(file_descriptor, F_SETLK, &lock) == -1)
        perror("Failed to acquire lock");
// Release the lock
lock.l_type = F_UNLCK;
if (fcntl(file_descriptor, F_SETLK, &lock) == -1)
        perror("Failed to release lock");
```

File I/O

offset: # bytes from head of file. 紀錄現在讀寫到哪，等等就從哪開始讀寫
init as 0 when open. might exceed file size → truncate file at next write.
delayed write: `write` 的時候，先將資料存在 buffer，需要釋出空間時再將這些資料排入輸出隊列，等到該資料排到隊首時才真正的寫入磁碟
fast system calls: return in fixed time ; **slow:** return forever until event happened
multiple I/O: blocking: any fd might block
unblocking(polling): read if has data, otherwise return and keep reading
I/O multiplexing: only read when data is ready