Shun/翔海 (@shun4midx)

SP Midterm Notes

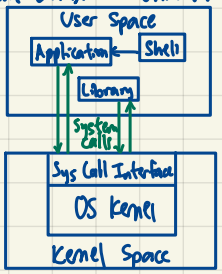
FILE I/O 1 (UNIT 3)

Shun/翔海 (@shun4midx)

PRE-REQUISITES "Everything is a file"

Who created Unix: Bell Labs 1969 (Ken Thompson & Dennis Ritchie)

What Linux is: Unix-like kernel, Linus Torvalds (1991)



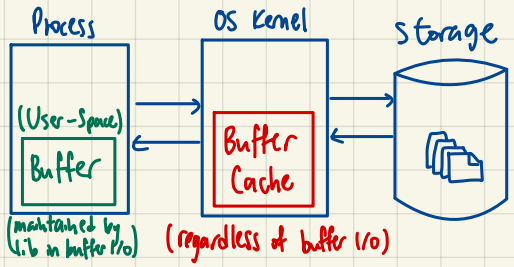
User Mode
e.g. printf()

Kernel Mode
e.g. write()

↳ System calls cause switches from the user space to kernel space

BUFFERED VS UNBUFFERED I/O

Buffered I/O includes a buffer maintained by the library in the process, hence being faster by decreasing slow system calls



FILE DESCRIPTORS

The Unix kernel refers to an open file using a file descriptor (fd)

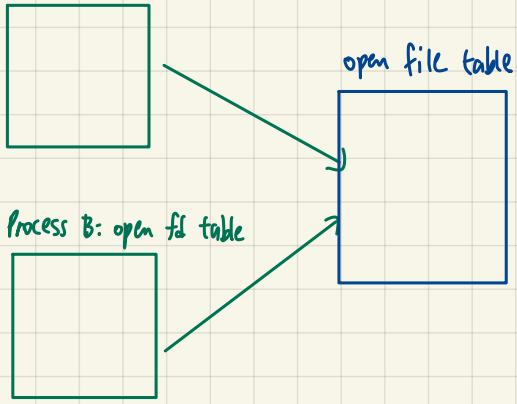
↳ Is a nonnegative integer ∈ [0, OPEN_MAX-1], OPEN_MAX = max # files a process can open at once

↳ The kernel manages a file descriptor table for each process and allocates the file desc.

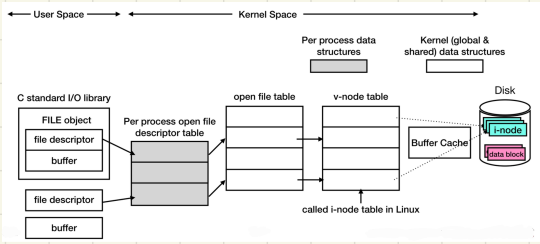
↳ Fds are per-process — different processes may have the same fd.

↳ Common fds: STDIN_FILENO=0, STDOUT_FILENO=1, STDERR_FILENO=2

E.g. Per process, kernel data structures



FILE I/O KERNEL STRUCTURE



Open fd table: One entry per fd, contains fd flag

Open file table: Contains file status flag, current file offset

V-node info: file type, ptrs to func to operate

I-node: Both on storage device and in memory, contains metadata

OPEN AND OPENAT ☆Returns lowest-numbered not opened fd

int open(const char* path, int oflag, mode_t mode); (oflag: specify access modes)

int openat(int fd, const char* path, int oflag, mode_t mode);

"AT_FDCWD = path relative to curr working dir

☆ open can be used to get fd

DIFFERENT OFLAGS

↳ O_RDONLY: read only

↳ O_WRONLY: write only

↳ O_RDWR: read + write

Sometimes OR'd with file creation/status flags:

↳ O_APPEND: Append to EOF for each write

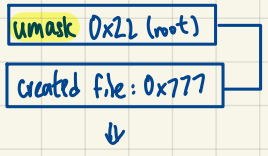
↳ O_TRUNC: Truncate file size to 0 (may ease contents)

↳ O_CREAT: Create if file does not exist

↳ O_NONBLOCK: Non-blocking mode

↳ O_SYNC, O_DSYNC, O_RSYNC: Data sync, wait for I/O completion

MODE CONSTANTS AND MASKING (IF flag | O_CREAT)



Resulting perm: 0x755 (XOR)

Constants in the form:

S_I[ ][ ]

↳ cap so that it's total length is 7, [USR, GRP, OTH]

[RWX], avail perms in that order (read, write, exec)

E.g. S_IRUSR, S_IRWXO

CLOSE ☆Returns 0 if successful, 1 if error

int close(int fd);

↳ When a process terminates, all its open files are closed automatically by the kernel

CREAT ☆Returns fd if success, -1 otherwise

int creat(const char* path, mode_t mode);

↳ Created and opened write-only

↳ Equiv to open(path, O_WRONLY | O_CREAT | O_TRUNC, mode)

FILE I/O 2 (UNIT 4)

Shun/舜海 (@shuntmidx)

FILE OFFSET

Every open file in Unix has an associated file offset.
It's init to 0 when opened unless O_APPEND.

READ AND WRITE

```
ssize_t read(int fd, void* buf, size_t nbytes);
```
☆ Returns # bytes read, or -1 on error, 0 if EOF (e.g. #bytes left < nbytes)

```
ssize_t write(int fd, void* buf, size_t nbytes);
```
☆ Returns # bytes written, or -1 if error

Write can surpass EOF, but read returns 0 already, so O_APPEND setting offset to EOF is bad for read.

LSEEK ☆Returns new file offset (of open file), or -1 if error

```
off_t lseek(int fd, off_t offset, int whence);
```

whence can be set to three values:
- ↳ SEEK_SET: file offset ← offset bits + beginning
- ↳ SEEK_CUR: file offset ← offset bits + curr val
- ↳ SEEK_END: file offset ← EOF + offset bits

☆No actual I/O takes place, the updated offset is only used in the next read/write operation

UNIX TIME COMMAND (strictly in user space / strictly in kernel space)

"time ls" ⇒ real time, user time, sys time

READ-AHEAD + DELAYED-WRITE

Uses buffer cache in the kernel to optimize disk I/Os.
- ↳ Detects sequential reads ⇒ stores in cache (Buffered data) so we access from kernel not disk later on, similar with writing to disk later

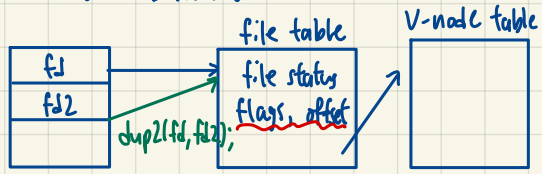
DUPLICATING FILE DESCRIPTORS

☆Returns new fd, or -1 if error

```
int dup(int fd);
int dup2(int fd, int fd2);
```

dup: Assigns lowest avail fd ← new fd
dup2: Assigns fd2 ← fd (if fd2 was used, it's closed first of)

Done by having both fds refer to the same open file table entry
⇒ If offset is modified by lseek, it does for both fds



☆They connect to the same file table!!

/dev/fd

Unix provides a dir /dev/fd whose entries are named 0, 1, 2, etc...

Namely, these two are equivalent

```
int fd = open("/dev/fd/0", mode);
int fd = dup(0);
```

ERROR HANDLING

Under header file <errno.h>

FCNTL ☆Returns -1 on error

```
int fcntl(int fd, int cmd, ... arg);
```
- ↳ cmd = F_GETFL ⇒ get file status
- ↳ cmd = F_SETFL ⇒ set file status

Different types of file statuses:
O_RDONLY, O_WRONLY, O_RDWR, O_EXEC,
O_SEARCH ← open dir for searching only
or O_APPEND, O_NONBLOCK, O_□SYNC

MASKS IN FCNTL (How do I turn on/off flags?)

Sample code to add a flag "flag" to a file status

```
int val;
if ((val = fcntl(fd, F_GETFL, 0)) < 0)
    err_sys("fcntl F_GETFL error");
val |= flag  /* turn on flag */
if (fcntl(fd, F_SETFL, val) < 0)
    err_sys("fcntl F_SETFL error");
```

Get curr file status

val &= ~flags to turn off flags

Set to new flagged file status

ATOMIC OPERATIONS

Race Condition: When multiple processes try to do something with shared data and the final outcome depends on the order in which the processes run

☆lseek() + read()/write() or creat() (O_TRUNC.) easily cause r.c.

Atomic Operation: Effectively executed as a single step (exec w/o other proc reading or changing the state during the operation)

PREAD AND PWRITE —— ATOMIC READ/WRITE (Avoid offset updated mid-step)

Basically lseek() + read() or write(), offset counted from start of file

```
ssize pread(int fd, void* buf, size_t nbytes, off_t offset);
ssize pwrite(int fd, const void* buf, size_t nbytes, off_t offset);
```

EXAMPLES OF RACE CONDITIONS (Consider two processes running the same snippet)

Snippet 1 (Both write at same loc...)

```
if (lseek(fd, 0, SEEK_END) < 0)
    err_sys("lseek error");
if (write(fd, buf, 100) != 100)
    err_sys("write error");
```

Snippet 2 (file not found twice = same file created twice)

```
if ((fd = open(path, O_WRONLY)) < 0) {
    if (errno == ENOENT) {
        if ((fd = creat(path, mode)) < 0)
            err_sys("creat error");
    } else {
        err_sys("open error");
    }
}
```

FILE/RECORD (BYTE RANGE) LOCKING AND MULTIPLEXING (UNIT 5)

Shun/翔海 (@shun4midx)

ADVISORY VS MANDATORY LOCKING

Advisory lock: A cooperative locking scheme, only works if everyone checks the locking status. Otherwise, it can still read/write a locked file ← ofc problematic

☆ We only learn about advisory lock

Mandatory lock is an enforced lock by the OS, but it is rarely used.

FLOCK ☆Returns 0 if success, -1 otherwise

int flock(int fd, int operation);

↳ Applies/removes a lock on an open file

The operation is one of the following:

- LOCK_SH: Places a shared lock (>1 proc can hold the lock at a given time) (e.g. read)
- LOCK_EX: Places an exclusive lock (only 1 proc can access the lock at a given time) (e.g. write)
- LOCK_UN: Removes a lock held by proc

FCNTL RECORD LOCKING

int fcntl(int fd, int cmd, ..., struct flock *flockptr); (contains locking info)

It supports three cmds for record locking: (updates flockptr)

- F_GETLK: Determine if lock can be placed on the file
  - ↳ Yes ⇒ returns F_UNLCK in l_type field
  - ↳ No ⇒ returns details about the lock preventing placement
- F_SETLK: Sets lock as flockptr, if fail, return -1 and update errno
- F_SETLKW: F_SETLK but caller blocks until lock is released.

Obviously, this is vulnerable to race conditions...

We have this struct:

```
struct flock {
    short l_type;   (F_RD, WR, UN LCK)
    short l_whence; SEEK_ (SET, CUR, END)
    off_t l_start;  offset
    off_t l_len;    # bytes to lock, 0 ⇒ to EOF
    pid_t l_pid;    filled in by F_GETLK
}
```

RELEASE OF LOCKS

☆ forked child does not inherit locks!!

When a process terminates, all its locks are released (can already foresee issues with dup'ed fd...)

Locks are assoc w/ a proc and a file, so when a fd is closed, any locks on the file ref by fd are released

BLOCKING VS NONBLOCKING I/O

Fast func calls: Take a known amount of time to finish, not blocking

Slow func calls: Wait for an indefinite amount of time to finish

Blocking I/O: I/O funcs that do not return until their action is completed (slow func calls)

Nonblocking I/O: I/O operation issued and returned ASAP w/o waiting (may return w/ error)

I/O MULTIPLEXING — SELECT AND POLL (Do not waste resources)

SELECT ☆ Returns # ready fds, 0 on timeout, -1 on error

int select(int nfds, fd_set *readfds, ␣ writefds, ␣ exceptfds, struct timeval *timeout);

↳ Descriptor sets (readfds, writefds, exceptfds) are stored in fd_set data type

↳ The descriptor sets are updated to show which fds are ready for each category

↳ Returned int = Σ ready fds

↳ nfds is max FD_SETSIZE

↳ As timeout is in microseconds, if we set nfds to 0, desc set to NULL, we get a high-precision timer

POLL ☆ Returns # ready fds, 0 on timeout, -1 on error

int poll(struct pollfd fdarray[], nfds_t nfds, int timeout); where

```
struct pollfd {
    int fd;       fd to check, <0 to ignore
    short events; events of interest on fd
    short revents; events that occured on fd
}
```

☆ >3 condition types supported

☆ Masks

MULTIPLEXING WITH SELECT() AND POLL()

The point of either is it waits and informs you when your call on fd is ready.

Code Sample (Blocks until Readable):

```
struct pollfd fds[1];                          ☆ poll()
fds[0].fd = sockfd; fds[0].events = POLLIN;
poll(fds, 1 (nfds), -1); //-1 = wait forever
if (fds[0].revents & POLLIN) - Poll mask
    recv(sockfd, buf, sizeof(buf), 0);
```

☆ select()

```
fd_set rfds; FD_ZERO(&rfds);
FD_SET(sockfd, &rfds);
select(sockfd+1, &rfds, NULL, NULL, NULL);
if (FD_ISSET(sockfd, &rfds)) - ISSET updated?
    recv(sockfd, buf, sizeof(buf), 0);
```

COMPARING I/O MODELS (Multiplexing is the best)

	Blocking I/O	Nonblocking (polling)	I/O Multiplexing
How to Use?	FD w/o O_NONBLOCK	FD w/ O_NONBLOCK	select()/poll()
Handle >1 fd?	No	Yes	Yes
CPU Usage	Low	High	Low
Responsiveness	Slow	Fast	Fast

FILES AND DIRECTORIES I (UNIT 6 PAGE 1)

Shun/翔海 (@shuntmidx)

STAT, FSTAT, LSTAT ☆Return 0 if OK, -1 if error

```
int stat(const char* pathname, struct stat* buf);
int fstat(int fd, struct stat* buf);
int lstat(const char* pathname, struct stat* buf);
```

struct stat is obtained from a file's i-node

buf holds file info, diff betwn stat and lstat is if pathname is a symbolic link, then lstat returns info about sym link but NOT the file ref by sym link

FILE TYPES

example of macro: #define S_ISDIR(mode) ((mode) & S_IFMT) == S_IFDIR

Type of file mask

Macro	Type of File
S_ISREG()	Regular File: Text, binary, etc
S_ISDIR()	Directory File: Contains name of other files and ptrs to info of these files
S_ISCHR()	Char special files: e.g. tty, audio
S_ISBLK()	Block special files: e.g. disk
S_ISFIFO()	FIFO: named pipes
S_ISLNK()	Sym Links: File that points to another file
S_ISSOCK()	Sockets: file used for network communication

ACCESS() ☆Returns 0 if OK, -1 on error

```
int access(const char* pathname, int mode);
```

↳ Checks if real user/group has access to the file (e.g. SGID/SUID can check)

↳ mode = F_OK (if a file exits) by default, but can be bitwise OR any flags R_OK, W_OK, X_OK (r, w, x perm)

UMASK()

File mode creation mask ⇒ mask on = turn off in st_mode

```
mode_t umask(mode_t cmask);
```

updates file mode creation mask, and returns value before update

Mask bit	Meaning
0400	user-read
0200	user-write
0100	user-execute
0040	group-read
0020	group-write
0010	group-execute
0004	other-read
0002	other-write
0001	other-execute

Example: default = 0666

mask = 0022 //write

new = 0644 //no write

ACCESS PERMS & UID/GID

There are 9 perm bits per file, divided into three categories: read(r)[4], write(w)[2], execute(x)[1] and the following users: file owner/user (u), group (g), other (o), all users (a: ugo) for each file, resulting in the st_mode mask of: S_I[ ][ ] (R,W,X; order of MSB to LSB; USR, GRP, OTH)

⇒ shell > chmod u=rwx, g=rx, o=r file == shell > chmod 754 file (-rwx-rx-r)

Unique UID but can share GID

DIFFERENT ID TYPES

who we really are

- Real User/Group ID (UID/GID): Taken from password file /etc/passwd when a user logs in (maps usernames to IDs) E.g. file, oracle:x:1021:1020:Oracle_user:... (UID, GID)
- Effective User/Group ID: Used for process's access permission checks
- Supplementary Group IDs: Unix maintains this list to track the groups the process belongs to ⇒ used for access permission checks
- Saved-Set User/Group ID: Contains copies of effective user/group ID

☆ Every file has a user and group owner: seen in st_uid and st_gid from struct stat

FILE ACCESS TEST (if it passes, perform access permission check)

straightforward: just check rwx alignty or not for file or dir

1. If effective user ID = 0 (0 ⇒ superuser, so access to all files)
2. If effective UID == file UID (st_uid): Perm check on user
3. If effective GID or a supp GID == file GID (st_gid): check grp
4. Perform perm check on other

☆ Rmk: To del a file, we only need dir perm, not file perm.

SET-UID (SUID) + SET-GID (SGID)

E.g. -rwsr-xr-x, st_mode mask: S_ISUID, S_ISGID

SUID: Set effective UID of a process to be the owner: st_uid

SGID: Set effective GID of a process to be the group owner: st_gid

☆ E.g. allows normal users to change their password by changing the password file, even if it should only be written by root

OWNERSHIP OF A FILE CREATED BY A PROCESS (open or creat)

UID = effective UID of proc, GID = effective GID of proc OR GID of parent dir

STICKY BIT

Aka "saved-text bit", used only in dirs. It is denoted with st_mode flag S_ISVTX, and at the end of perms displays a letter "t". drwxrwxrwt

Files inside the dir can only be removed/renamed by file owner, dir owner, or superuser

E.g.: /tmp. Everyone has write perm but cannot del/rename ∴ sticky bit

CHMOD, FCHMOD ☆Ret (OK? 0:-1)

```
int chmod(const char* pathname, mode_t mode);
int fchmod(int fd, mode_t mode);
```

☆ Change file access perms for an existing file

☆ mode = bitwise OR of mode consts (S_ISVTX, S_IRUSR, etc)

CHOWN, FCHOWN, LCHOWN ☆Ret (OK? 0:-1)

```
int chown/lchown(const char* pathname, uid_t owner, gid_t group);
int fchown(int fd, uid_t owner, gid_t group);
```

☆ Changes a file's UID and GID

☆ fchown changes an already opened file

☆ lchown changes the ownership of sym link not file

If owner or group = -1, the ID is unchanged

Rmk: The process that uses chmod or chown must be by superowner or eff UID == st_uid

FILE SIZE (st_size)

Regular file: 0~MAX, Dir: Mult of 16 or 512, Sym links: length of pathname

FILE TRUNCATION ☆Ret (OK? 0:-1)

```
int truncate(const char* pathname, off_t length);
int ftruncate(int fd, off_t length);
```

☆ fills in 0 if have remaining space

FILES AND DIRECTORIES 1.5 (UNIT 6 PAGE 2) + FILES AND DIRECTORIES 2 (UNIT 7)

Shun/翔海 (@shunthmidx)

I-NODE AND I-NODE LINKS

In struct stat, st_nlink = link count and st_ino = i-node number

st_nlink keeps track of #dir entries that points to the i-node (each is a hard link)

The file (spec. by i-node) can only be deleted from the file sys if st_nlink = 0

E.g. mv file1 file2, there is no need to physically move the contents in disk blocks, only needs a new dir entry pointing to the existing i-node then unlink the old dir entry.

LINK() AND UNLINK() (OK? 0: -1)

int link(existingpath, newpath);
int unlink(pathname);

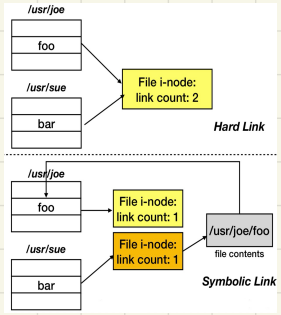
Link
- Create a hard link to same i-node
- Increments st_node in i-node
- Returns error if path dir exists
- Both paths must be on same filesystem
- Normal users CANNOT hard link dirs

Unlink
- Removes a dir entry (del name in filesys)
- Decrements st_node in i-node
- st_node == 0 && NOT OPEN ⇒ del file data
- Needs w+x perm on dir (not actual file)
- Sticky bit rules applied to shared dir

HARD LINK VS SYM LINK

Normal users can create/del sym links, which are indirect ptrs to files, w/o needing paths on the same filesys (Only superuser can hard link dirs)

Example:
mkdir foo
touch foo/a
ln (-s) ../foo foo/testdir (sym)
To remove sym link, "rm foo/testdir"



	Hard Link	Symbolic Link
File type (in the stat structure)	No actual file	S_IFLINK
Can link to directory	No (in most systems)	Yes
Can create links across file system boundary	No (in most systems)	Yes
Link Implementation	A directory entry pointing to the i-node of the linked target	A file containing the path (in file content) to the linked target

SYMLINK AND READLINK

int symlink(actualpath, sympath); OK?0:-1
Creates sym link sympath → actualpath
Doesn't require actualpath to be a file

★Ret #bytes read, -1 if error
int readlink(pathname, buf, bufsize);
Reads content in sym link pathname, not null terminated

RENAME ★Ret (OK? 0: -1)

int rename(oldname, newname);
Any hard links or open file desc are unaffected. The user must have w+x perm in dir to execute.

FILE TIMES

There are three types of time in stat:
- st_atim: Access time (e.g. read)
- st_mtim: Modification time (e.g. write)
- st_ctim: Change-status time (e.g. chmod, chown)

★ st_ctim is when i-node was last modified
★ rename() is atomic ⇒ updates ctim not mtim
Syntax: ls -l --time=atime/ctime, ls -lu/-lc

UTIME AND UTIMES ★OK? 0: -1

int utime(pathname, const struct utimbuf* times);
int utimes(pathname, const struct interval times[2]);
utime (to the nearest second)
struct utimbuf {time_t actime, time_t modtime;}
Changes access and modification times to times'

utimes (to the nearest microsecond)
struct timeval {time_t tv_sec; long tv_usec;}
Access time = times[0], modif time = times[1]

FILE DIRECTORIES

mkdir(pathname); = New dir (creates two links: ., ..)
rmdir(pathname); = Del dir if empty
chdir(pathname); = Change curr proc working dir
fchdir(fd); = Change curr proc working dir
getcwd(buf, size); = Gets curr working dir and stores in buf (env/sys calls)

TRAVERSING FILE HIERARCHIES

```
struct dirent {
    ino_t d_ino;
    char d_name[NAME_MAX+1];
}
```

DIR* opendir(pathname); ★Ret ptr of dir
struct dirent* readdir(DIR* dp); ★Ret ptr or NULL at end of dir or error
struct dirent* rewinddir(DIR* dp); → reset to beginning
struct dirent* closedir(DIR* dp); OK? 0: -1
TL;DR, these commands used in C code can trav a file hierarchy. Don't need to memo but need to be able to identify in code.

FLUSHING BUFFERS

Buffered I/O uses internal memory to reduce sys calls, there are three main types of modes of flushing (writing to kernel)
- Fully buffered: Regular files or pipes (flush when full)
- Line buffered: Terminals (flush on newline)
- Unbuffered: stderr (flush immediately)

★ We update stdio → kernel → Disk usually
fflush(FILE* fp); ★ Stdio → Kernel
↳ Manually flush user-space buffer to kernel (: If a proc exits abnormally, data in user-space buffers may be lost)
void sync(void); ★ Kernel → Disk
↳ Queues ∀ files in file sys, their modified block buffers in the kernel for writing to the disk
int fsync(int fd); ★ Kernel → Disk
↳ Synchronize data and metadata (e.g. i-node) of file
int fdatasync(int fd); ★ Kernel → Disk
↳ Synchronize the data of the file, and metadata if needed
⇒ fsync and fdatasync wait for disk write before ret.

PROCESS CONTROL I (UNIT 8)

Shun/翔海 (@shuntmidx)

PROCESSES (OVERALL PICTURE)

Program: An exec file stored on a disk
Process: An executing instance of a program
↳ shell < ps = snapshot of curr procs
↳ shell < top = display procs

PROCESS IDENTIFIERS = PID

☆ Returns proc ID of calling process

`int getpid(void);`

Every proc has a unique PID, the kernel allocates a new PID upon proc creation, and reuses the PID after proc termination.

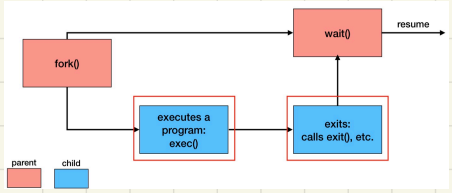
`int getppid(void); //Get parent proc PID`

SPECIAL PROCESSES

PID = 0: Swapper or scheduler/idle proc
↳ Kernel proc, i.e. no program on the disk corr to this process (no shell process either)
PID = 1: Init process
↳ User proc w/ superuser privilege
↳ Runs init program (/etc/init) by reading sys-dependent init files
↳ Inits system services and login proc
PID = 2: Pagedaemon / kthreadd (Unix/Linux)
↳ Kernel process, responsible for paging virtual memory system

☆ TLDR: PID 0, 1, 2 are occupied

FORK PROCESS CREATION



FORK FUNCTION

☆ Ret 0 if child, PID of child if parent, -1 otherwise

`pid_t fork(void);` Creates child process when called

We cannot control the order of which proc runs first, so we need e.g. sleep() for sync if necessary

VFORK

(So does fork() too)

`pid_t vfork(void);` Uses copy-on-write (COW), similar to the i-node link thing.

Diff from fork:
↳ Child proc runs in same address space as parent
↳ Child runs first (∵ parent is blocked)
↳ Parent is blocked until child calls exec/exit

vfork() is undefined if the child modified data, makes function calls, or ret w/o calling exec/exit

DEADLOCK — Two or more competing actions are waiting for the other to finish, so neither does.

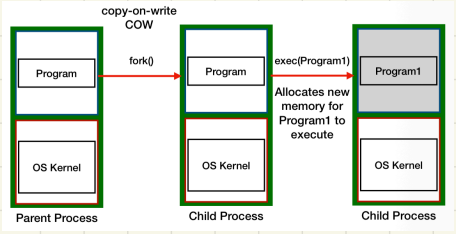
Example of deadlock

```
int
main(void)
{
    int     var;        /* automatic variable on the stack */
    pid_t   pid;

    var = 88;
    printf("before vfork\n");   /* we don't flush stdio */
    if ((pid = vfork()) < 0) {
        err_sys("vfork error");
    } else if (pid == 0) {      /* child */
        while (var < 100);      /* parent's variables */
        _exit(0);               /* child terminates */
    }
    /* Parent continues here. */
    var = 100;
    exit(0);
}
```

child first

FORK + EXEC



EXEC FUNCTIONS

☆ Ret -1 if error, NO RET IF SUCCESS

The real sys call is execve(), others are simply wrappers of the same thing.

exec replaces curr proc w/ new alloc memory but same PID to execute

Variants: (l and v, p and e)
l: "list", so exec("str1", ..., NULL)
v: "vector", args[] = {"str1", ..., NULL}; Then exec(cmd, args)
e: Environment variables, envp[] = { }; Then execl_, envp);
p: Path, we only have "filename" not "pathname", so it's searched for with the PATH env var, and ran there.

ENVIRONMENT VARIABLES

Named values that affect runtime behavior
Examples of env var we can set:
HOME, PATH, SHELL, USER, LOGNAME

FD_CLOEXEC FLAG

Fd flag can be set/get by fcntl:
↳ fcntl(fd, F_SETFD, val)
↳ fcntl(fd, F_GETFD, val)

FD_CLOEXEC = close when execl(), i.e. when we call execl(), we close fd so that running that process doesn't affect the current fd

FORK VS EXEC

Property after op	fork()	execl()
Open file descriptors (e.g. offset)	Inherited from parent	FD_CLOEXEC ⇒ closed otherwise ⇒ unchanged
Parent and proc PID	Child proc ⇒ new PID	Same PID
IDs: Real/Effective UID/GIDs	Inherited from par	SUID/SGID ⇒ effective UID/GID = SUID/SGID
Dirs: Curr working dir, root dir	Inherited from par	Same d.rs
Masks	Inherited from par	Same masks
File locks	NO INHERITANCE	Same locks
Pending alarms/signals	NO INHERITANCE	Same
Environment	Inherited from par	Could be changes by input env

WAIT AND WAITPID ☆ Ret PID if OK, 0, or -1 on error

`pid_t wait(int* statloc);`

`pid_t waitpid(int pid, int* statloc, int options);`

They wait for a state change in any child (e.g. stopped or resumed), and can retrieve its exit status value via statloc

Wait/waitpid can:
- Block: All children are running
- Return immediately w/ status value: Child changed state
- Return immediately w/ error: Have no child procs

WAIT VS WAITPID

wait() blocks the caller until status change in a child, whereas waitpid() has this as default but modifiable depending on PID value

☆ pid == -1: wait for any child
☆ pid > 0: wait for child w/ PID pid
pid < -1: wait for child w/ PGID |pid| (process group ID)
pid == 0: wait for child w/ PGID = calling PID

☆ options = WNOHANG ⇒ nonblocking

PROCESS CONTROL 2 (UNIT 8)

Shun/翔海 (@shun4midx)

ORPHANS AND ZOMBIES

Zombie process = child terminated but parent didn't wait yet

Orphan process = parent terminated but child running still

ORPHAN PROCESSES

Unix ensures each proc has a parent proc by setting parent = init proc (PID 1)

Notably, init's children never become zombies, child terminate ⇒ init wait()

in order to get termination status

ZOMBIE PROCESSES

We can make zombies like so:

```
if (fork() == 0) {
    exit(0); // child exit
} else {
    while (1) {
        sleep(1); // don't wait
    }
}
```

DOUBLE FORK ⇒ NO ZOMBIES

Create a child, fork again and immediately exit ⇒ orphan grandchild

∴ Grandchild adopted by init, which auto-calls wait()

⇒ No zombie :D

PIPES (UNNAMED PIPES)

int pipe(int fd[2]); ⇒ Provides IPC channels

↳ Read end : fd[0], write end : fd[1]

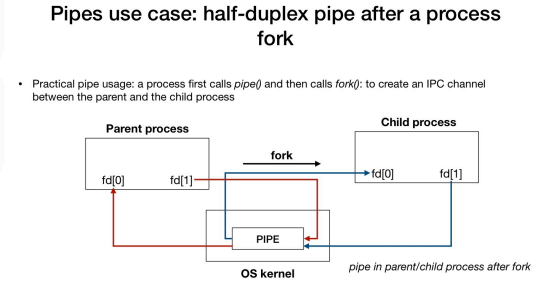
↳ Output of fd[1] = Input of fd[0]

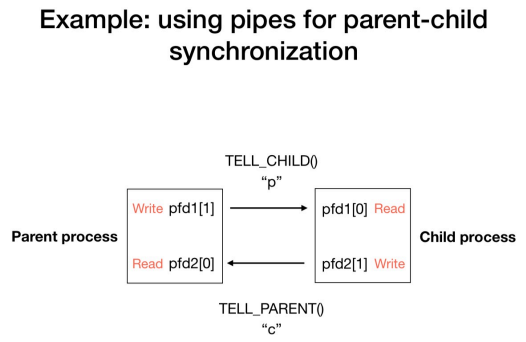
↳ File type of fd[0], fd[1] are FIFO

☆ Blocking when pipe is empty/full

☆ close(read) = SIGPIPE, close(write) = EOF (0)

☆ Atomic write = no interleaving other writes (happens only if data size <= PIPE_BUF)





FIFO (NAMED PIPES)

Can check with macro S_ISFIFO

Also provide IPC channels betwn processes

☆ The data exchanged via FIFO is kept internally w/o writing it to the file sys.

int mkfifo(const char* path, mode_t mode);

↳ path = name of FIFO, mode = umask

↳ FIFO must be opened on both ends before data can be passed (need O_RDONLY for read end, O_WRONLY for write end)

Use FIFO instead of pipe for unrelated processes

☆ made w/ O_NONBLOCK or w/o O_NONBLOCK

↳ For w/ O_NONBLOCK, open for O_WRONLY may ret error if no process has FIFO open for reading