Shun/翔海 (@shun4midx)

SP Final Notes

(At least I hope it's my final time taking this course xd)

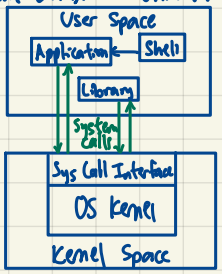
FILE I/O 1 (UNIT 3)

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)



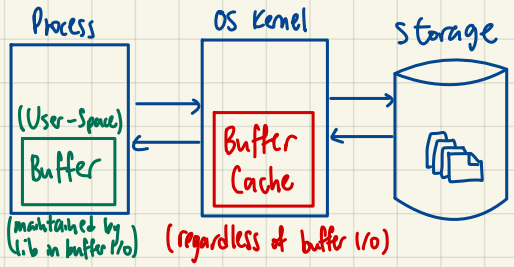
User Mode
e.g. printf()

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)

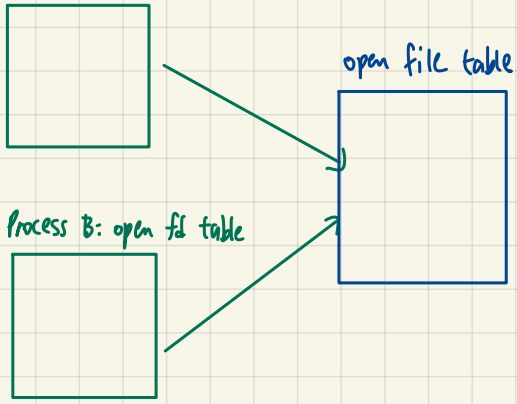
↳ Is a nonnegative integer ∈ [0, OPEN_MAX-1], OPEN_MAX = max # files a process can open at once

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

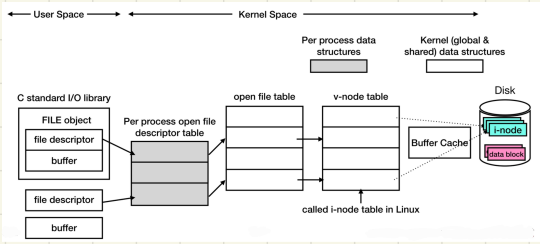
↳ Fds are per-process — different processes may have the same fd.

↳ 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

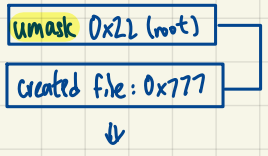
↳ O_TRUNC: Truncate file size to 0 (may ease contents)

↳ 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 disc 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

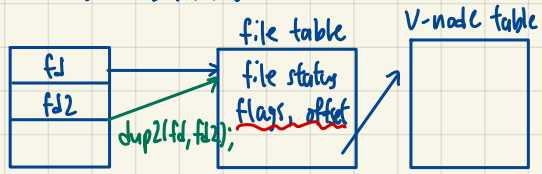
DUPLICATING FILE DESCRIPTORS

☆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 c)

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() easily cause r.c. (O_TRUNC.)

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);

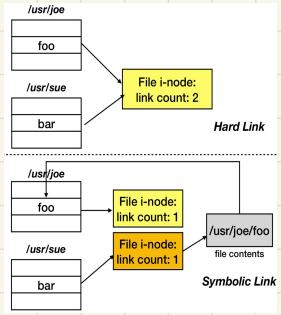
Link
- Create a hard link to same i-node
- Increments st_node in i-node
- Returns error if path dir exists
- Both paths must be on same filesystem
- Normal users CANNOT hard link dirs

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);
```

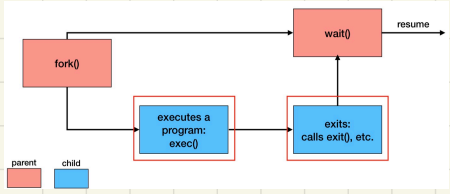
Every proc has a unique PID, the kernel allocates a new PID upon proc creation, and reuses the PID after proc termination.

```
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

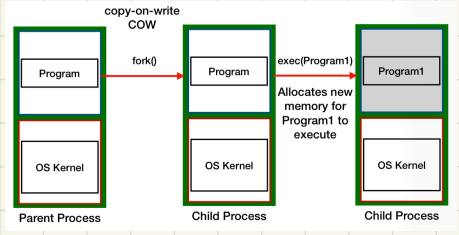
— Two or more competing actions are waiting for the other to finish, so neither does.

Example of deadlock (child first)

```
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);
}
```

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 within 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

ATEXIT

```
int atexit(void(*func)(void));
```

Registers func to be called at proc exit

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
☆ 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]

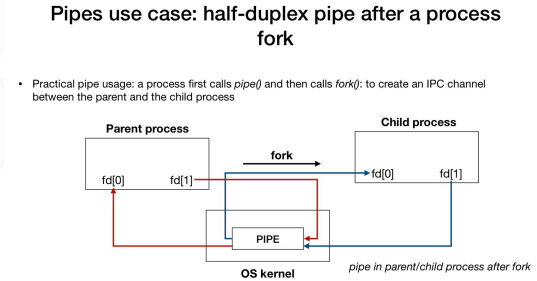
↳ Output of fd[1] = Input of fd[0]

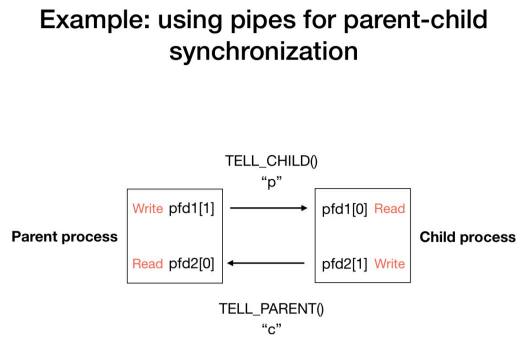
↳ File type of fd[0], fd[1] are FIFO

☆ 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

EXAM REMARKS FROM THE MIDTERM

Shun/翔海 (@shun4midx)

Shun will fill this in on Wed

1 day before the final :)

SIGNALS (UNIT 9.1) — BEFORE NONLOCAL JUMPS

Shun/翔海 (@shuntmidx)

OVERVIEW OF SIGNALS

Signal: Notification/Event sent to user processes to notify them of an event; events may generate signals (e.g. kill(1), divide-by-zero, ctrl+C from the terminal)
↳ Receive/send at any time/order w/o waiting for each other
↳ Handled one at a time by a signal handler
↳ A program can use its own or the default sig han.

SIGNAL TYPES AND FORMAT

Defined by a name (beginning w/ SIG) and number (+ve int) (consts in <signal.h>)

Key Signals

Terminal-generated signals:
↳ SIGINT(2): Sent when ctrl+C was pressed (interrupt)
↳ SIGKILL(9): Sent to terminate the process

Signals from Hardware Exceptions:
↳ SIGFPE(8): Divided-by-zero
↳ SIGSEGV(11): Illegal memory access (print segmentation fault)

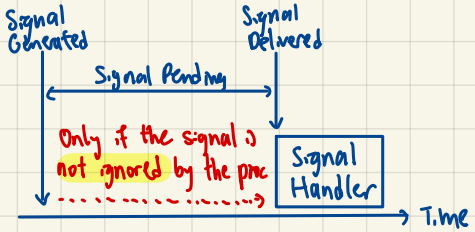
Signals from Software Conditions:
↳ SIGPIPE(13): Process that writes to a pipe w/o reader
↳ SIGALRM(14): Expiration of an alarm clock

SIGNAL DISPOSITION/ACTION

When a signal occurs, the kernel can do one of 3 things, which is called the disposition of the signal/action linked to the signal: (disposition = action assoc)
- Ignore the signal
- Catch the signal w/ process's own signal handler
- Perform the default action in the C library

SIGNAL TERMINOLOGY AND SEMANTICS

i.e. we check status by if an action is taken



A signal is pending: signal is generated for a proc but the action is not yet taken
A signal is delivered to the proc: Action is taken

A proc can block signal delivery if the signal is generated and the action ≠ ignore, i.e. not default/caught, and is pending until the proc unblocks or action → ignore.
(not ignored, just "held on pending")

Signal Mask: Each proc has one, defines the set of signals currently blocked from delivery to the proc ⇝ get/change w/ sigprocmask()

EXCEPTIONS The signals SIGKILL and SIGSTOP CANNOT BE BLOCKED/IGNORED/CAUGHT by the program's signal handler:
↳ SIGKILL: default action = terminate proc; SIGSTOP: default action = stop proc

≥1 SIGNALS? POSIX.1 doesn't specify the order of signal delivery, signals can be queued (but Linux doesn't queue! Only 1 is delivered to proc)

SIGNAL()

★Ret old disposition of signum, or SIG_ERR

```
typedef void (*sighandler_t)(int);   // data type of sig handlers.
sighandler_t signal(int signum, sighandler_t handler);
```
↳ Sets disposition of signum to handler (i.e. it "sends" the signal)
↳ handler: function ptr of sig han func, or SIG_IGN (ignore sig), SIG_DFL (default act)
↳ SIGKILL/SIGSTOP ⇒ ERROR!!

```
#def SIG_ERR (void(*)())-1, SIG_DFL:0, SIG_IGN:1
```

SIGNAL SETS

★sigset_t: rep ≥1 signals

```
typedef struct {unsigned long sig[_NSIG_WORDS];}
```
Don't manipulate directly! Use the 5 helper funcs.
★Ret -1: err, 0: OK (or member false), 1: member true

```
int sigemptyset(sigset_t* set);   // declare set
int sigfillset(sigset_t* set);    // assign set
int sigaddset(sigset_t* set, int signo);   // add to set
int sigdelset(sigset_t* set, int signo);   // rm from set
int sismember(const sigset_t* set, int signo);
```

```
int sigpending(sigset_t* set);   // ret pending sigs or err if invalid memory addr
```

SIGPROCMASK

★Ret 0: OK, -1: error

```
sigprocmask(int how, const sigset_t* set, sigset_t* oset);
```
Function depends on how = ?
↳ SIG_BLOCK: New sig mask ∪ set (sig incl block; union)
↳ SIG_UNBLOCK: New sig mask ← rm set (remove)
↳ SIG_SETMASK: New sig mask ← set

oset ≠ NULL: prev sig mask → oset
set = NULL: ignore how (sm unchanged)
≥1 signal delivered before ret

★get mask = sigprocmask(0, NULL, &sm);

KILL+RAISE

★Ret 0: OK, -1: error

```
int kill(pid_t pid, int signo);
int raise(int signo);   // = kill(getpid(), signo)
```
↳ kill() sends signo to { pid>0: PID pid; pid==-1: all proc prob won't test!? }
↳ Need either superuser or real/effective UID sender = ∨ receiver
↳ signo==0 ⇒ Ret -1, errno ← ESRCH (★can fault here)
↳ Nonatomic test for if proc exists w/ signo==0 for kill

ALARM+PAUSE

called by proc to catch SIGALRM usually

```
unsigned int alarm(unsigned int seconds);
```
↳ If secs==0, quit all pending alarms
↳ After expiring, SIGALRM is sent to calling proc
↳ Only one alarm per proc = new secs replaces old.
↳ Ret 0 if no prev alrm, secs left if have

```
int pause(void);   // ret -1, errno ← EINTR
```
↳ Suspends calling proc until a sig is delivered

SIGACTION

★Ret 0: OK, -1: error

```
int sigaction(int signo, const struct sigaction* act, struct sigaction* oact);
```
↳ Can get and/or modify action/disposition of a signal
↳ act ≠ NULL: New action for signo ← act
↳ oact ≠ NULL: Prev/curr (depends on "actual") action for signo → oact

```
struct sigaction {                          // for act/oact
  void     (*sa_handler)(int);              /* addr of signal handler, */
                                            /* or SIG_IGN, or SIG_DFL */
  sigset_t sa_mask;                         /* additional signals to block */
  int      sa_flags;                        /* signal options, Figure 10.16 */
                                            // I assume
  /* alternate handler */
  void     (*sa_sigaction)(int, siginfo_t *, void *);   // given in exam, combine by OR
};
```
↳ void func for signal handler

only tells you it existed at that moment, may have exited immediately after the check...

JUMP AND SIGNALS (UNIT 9.2+10)

Shun/翔海 (@shuntm.idx)

FORK VS EXEC – SIGNALS

	fork child	exec
Signal Mask	Inherited	Same
Pending alarms and signals	NOT inherited, signals are emptied, alarms are cleared	Same
Signal action/disposition	Inherited	If NOT ignored, then reset to default. Otherwise, ignore

REENTRANT FUNCTIONS

Def: Funcs that can be safely interrupted and called again before the first call completes, w/o causing erroneous behavior
- ↳ Safe to use in sig handlers
- ↳ Usually sig han can't tell where the proc was exec before signal was caught, solves this!

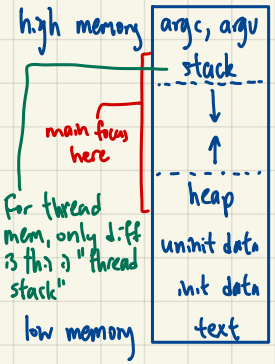
NONreen e.g.: strtoupper, malloc, free
Reen e.g.: chmod/chdir, dup, lseek

HOW TO MAKE A FUNC REEN

① Don't hold static/global var/data, don't ret a ptr to static memory
② For sig han, alloc memory in advance (don't malloc, free)
③ Call non-reentrant funcs

PROCESS MEMORY LAYOUT

reading order XD



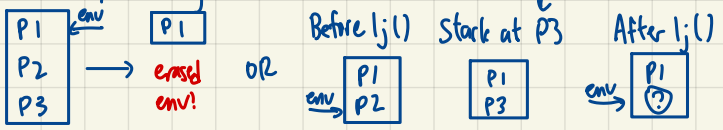
Stack frame: When a func is called, a stack frame is reserved for it, keeping the ret addr, arguments, saved registers

Nonlocal jumps enable you to transfer to an arbitrary program location not necessarily in the same func.

POTENTIAL PROBLEMS OF NONLOCAL JUMPS (I suggest working through slides examples)

We should only longjmp to the environment of a function that has been called but not yet completed!!

Otherwise, we may have:



SETJMP AND LONGJMP — BASIC SEMANTICS

`int setjmp(jmp_buf env); void longjmp(jmp_buf env, int val);`
- ↳ setjmp saves the calling environment into env, with env usually global (so longjmp uses)
- ↳ setjmp() rets 0 if called directly, rets nonzero if returning from a call to longjmp
- ↳ longjmp() uses env to restore the environment to the state during setjmp, i.e. setjmp saves the curr stack ptr (SP) and longjmp restores this SP value

TYPES OF VARIABLES UNDER SETJMP/LONGJMP

Automatic Variable: Var alloced and dealloced automatically when prgm enters/leaves var scope
Register Variable: Var w/ val stored in a register for faster access (request only, no guarantee)
- ↳ E.g. "register int val;"

When returning from a call to longjmp():
- Global/static/volatile vars retain values
- Automatic/register vars → undefined, could be restored via setjmp

USAGE OF SETJMP/LONGJMP (please check slide examples)

May test in exams to "find "problems" w/ the code :D

Usually if we want an alarm+pause and don't want the race cond of blocking forever cuz alarm went off, we can add setjmp(env_alrm) and have sig han w/ longjmp(env_alrm,1)

SIGSETJMP + SIGLONGJMP

`int sigsetjmp(sigjmp_buf env, int savemask);`
`void siglongjmp(sigjmp_buf env, int val);`
Same as normal setjmp/longjmp except if savemask≠0, then it saves curr sig mask to env

SIGSUSPEND ☆Ret -1, errno←EINTR

`int sigsuspend(const sigset_t* sigmask);`
- ↳ Replace curr sig mask w/ sigmask, then suspends proc until either:
  - ① A signal is delivered ⇒ ret when sig han ret
  - ② A signal terminates the proc ⇒ doesn't return
- ↳ When ret, restore sig mask to OG value

ABORT ☆Never ret

`void abort(void);` ←Kinda proc-related as a function
- ↳ Causes abnormal prgm termination by unblocking SIGABRT and raising SIGABRT
- ↳ EVEN IF SIGABRT is ignored, abort() will still terminate the proc IF IT CAN RETURN
- ↳ If it can't return (handler calls exit, or longjmp/siglongjmp), then no termination
- ↳ Default disposition of SIGABRT is terminate

SLEEP ☆Ret 0 or #unslept secs

`unsigned int sleep(unsigned int seconds);`
- ↳ Process suspends until seconds secs pass or a signal is caught ⇒ ret unslept secs (+sig han ret to quit)

THREADS (PART 1) — UNIT II.1

Shun/翔海 (@shunthmidx)

PROCESS VS THREADS

Process: Program in execution; an entity scheduled by the OS kernel for exec on CPUs
↳ Each proc has its own addr space, open files, pending alarms/signals, sig hans

Multithreading: Multiple tasks in the env of one process

Lightweight: Threads of the same proc share resources (threads: "lightweight procs")

Main Takeaway: Threads have shorter creation and termination time, but default share same addr space as others of same proc

MULTITHREADING MODEL

↳ User Threads: Supported by user space software w/o OS kernel support
↳ Kernel Threads: Managed directly by OS ker

MANY-TO-ONE MODEL (RARE NOW)

Multiple user threads → Single kernel thread
↳ Kernel dk about multiple user threads
↳ All thread management in user space
- Run-time system: func that do thread management + scheduling
- Thread table: Store per-thread states

(per proc)

Pros: No change needed in the kernel
Cons: If one thread has slow sys call, all others are likely blocked; cannot run parallel on multiple CPUs

THREAD IDENTIFICATION

pthread_t is used to represent thread ID, and we CANNOT USE "==" to compare.

```
pthread_t pthread_self(void);   // Ret own thread ID
int pthread_equal(pt_t t1, pt_t t2);
// Ret 0 if ≠, nonzero otherwise
```

kinda like getpid();

THREAD CREATION

☆Ret 0 if OK

```
int pthread_create(pt_t* tidp, pt_attr_t attr, void*(*start_rtn)(void*), void* arg);
```

Creates new thread w/ start routine start_rtn
↳ tidp: mem buf for new thread ID
↳ attr: Thread attributes (default = NULL)
↳ start_rtn: start routine addr for new thread
↳ arg: argument for start_rtn
(Terminology, like parent vs child proc)

Master thread calls pthread_create() to create worker(peer) threads to do work in parallel

☆Like fork(), there is no guarantee for the order of execution (for master/worker)

THREAD TERMINATION

These terminate a proc + all its threads:
- Thread in a proc that calls exit() (or _Exit/_exit)
- Sig sent to thread w/ def act term proc
- Ret from main() in main thread

How to exit w/o term proc:
- Ret in thread's start routine
- Thread calls pthread_exit()
- Cancelled by thread in same proc

```
void pthread_exit(void* rval_ptr);
```

① Terminates calling thread
② Rets return code/exit status as in rval_ptr avail to all threads in same proc via pthread_join

☆Other proc resources (e.g. fd) are NOT released if other threads are alive, only released when last thread terminates

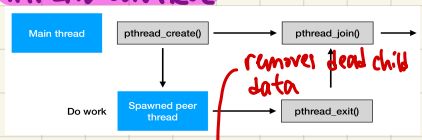
```
int pthread_join(pt_t thread, void** rval_ptr);   // ☆Ret 0: ok, errno: oth/w (with hex ret val!)
```

Even if rval_ptr = NULL

Blocks calling thread until "thread" exits

If rval_ptr ≠ NULL: Copies return code/exit status of target thread to rval_ptr
↳ Is ret code if thread rets from start routine or calls pthread_exit()
↳ Set to PTHREAD_CANCELED if cancelled

THREAD CONTROL



Default: Threads created w/ joinable attribute
↳ I.e. can be reaped/killed by other threads and its mem resources (e.g. stack/exit status) are only freed when being reaped

A thread is set to detached if we make it unable to be joined ⇒ its resources are auto reaped when terminated

THREAD DETACH

☆Ret 0 or errno

```
int pthread_detach(pt_t tid);
```

Detaches thread tid ⇒ undefined behavior if pthread_join() is called to wait for detached threads/attempting to detach a detached thread

THREAD ATTRIBUTES

Ret 0 or errno

```
int pthread_attr_init(pt_attr_t* attr);
int pthread_attr_destroy(pt_attr_t* attr);
int pt_attr_getdetachstate(attr, int* detachstate);
int pt_attr_setdetachstate(attr, int* detachstate);
```

init() sets attr to default values, destroy() sets attr to invalid values
get/set's "detachstate" specifies how to start the thread: PT_CREATE_DETACHED / PT_CREATE_JOINABLE

OTHER THREAD ATTRIBUTES

guardsize: guard buf size @ end of thread stack
stackaddr: lowest addr of thread stack
stacksize: min size of thread stack

ONE-TO-ONE MODEL (USED IN LINUX)

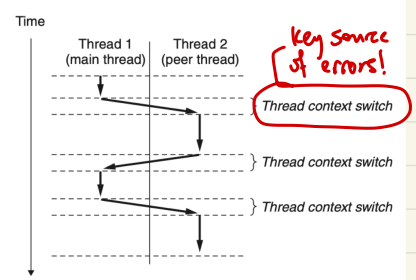
Each user thread ⇔ one kernel thread
↳ Creation of a user thread requires creation of a kernel thread

Pros: Multiple threads can parallel on multiple CPUs, do not block each other
Cons: Higher cost than many-to-one due to thread creation/destruction

POSIX THREADS PROPERTIES

Threads of the same process share:	Each thread has:
- Text, global data - Heap - Open file descriptors - Environment variables - PID, PPID - UID, EUID, GID, EGID - File record locks - Signal handlers - Signals/pending alarms	- Thread ID - Stack - Signal mask —> POSIX defines pthread_sigmask for pthreads - An errno variable - Scheduling properties - Thread-specific data (Chap. 12.6)

EXECUTION



THREADS (PART 2) — UNIT11.2+UNIT12.1

Shun/翔海 (@shuntm:idx)

THREAD CANCELLATION

Ret 0 or errno

`int pthread_cancel(pt_t tid);`

Limited to same proc threads, equiv to `pthread_exit(PTHREAD_CANCELED)`

To check if a thread is canceled, use `pt_join()`, see if it's `PT_CANCELED`

Canceled ⇒ terminated OR at cancellation point

`int pt_setcancelstate(int state, int* oldstate);`

Cancelability State: Decides whether a thread can be canceled (i.e. resp to cancelation request)

↳ `PT_CANCEL_ENABLE` / `PT_CANCEL_DISABLE` (queue until ENABLE)

`int pt_setcanceltype(int type, int* oldtype);`

Cancelability Type: How canc req are handled when received

↳ `PT_CANC_ASYNCHRONOUS` = can be canceled anytime

↳ `PT_CANC_DEFERRED` (DEFAULT): Canc req deferred until the thread next calls a function that is a canc point (a function)

CANCELLATION POINT

A thread is canceled when it calls a function that is a canc point and:

① Cancibility state: `PT_CANC_ENABLE`

② Cancibility type: `PT_CANC_DEFERRED`

③ A canc req is pending

`void pthread_testcancel(void);`

Creates canc point in the calling thread

Thread is canceled after called if ①+②+③, o/w no effect

THREAD CLEANUP HANDLERS

`void pt_cleanup_push(rtn, arg);`

`void pt_cleanup_pop(int execute);`

push() registers a thread cleanup handler rtn

rtn is called w/ arg if any: (CANNOT be triggered by join)

① Thread calls `pthread_exit()`

② Responds to canc req (i.e. is canceled)

③ Calls pop() w/ execute ≠ 0

PROCESS VS THREAD COMPARISON

Process primitive	Thread primitive	Description
fork()	pthread_create()	Create a new flow of control
exit()	pthread_exit()	Exit from an existing flow of control
waitpid()	pthread_join()	Get exit status flow of control
atexit()	pthread_cleanup_push()	Register function to be called at exit from flow of control
getpid()	pthread_self()	Gt ID for flow of control
abort()	pthread_cancel()	Request abnormal termination of flow of control

THREAD SYNCHRONIZATION

"var++" is NOT ATOMIC, on a CPU

it's ① Read var from mem to register

② Increment in register

③ Write register to mem

Race cond: 1 2 1' / 3 2' / 3' → var++

PTHREAD MUTEX

Ensures only one thread can access shared resources at a time (kinda like a lock!)

If a mutex is locked, other threads tying to set it are blocked until the mutex is released. (The reqs are lined in a FIFO (first in first out) queue)

`int pthread_mutex_init(mutex, attr);` `int pthread_mutex_destroy(mutex);` (UNINITIALIZES)

Init mutex by attr=NULL (default) or `mutex=PT_MUTEX_INITER`

`int pthread_mutex_lock(mutex);` (or trylock, unlock)

If a mtx is locked, then lock() ⇒ block caller until unlock,

trylock() ⇒ return EBUSY immediately, unlock() a mutex that caller doesn't hold ⇒ undefined behavior

ATTRIBUTES

`int pt_mutexattr_getpshared(attr);`

`int pt_mutexattr_setpshared(attr, pshared);`

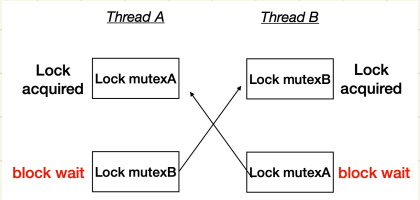
`PT_PROC_PRIVATE` (Default): Only op on threads created in the same proc as initializer (one proc)

`PT_PROC_SHARED`: Any proc w/ access to the mutex object can access (i.e. shared between procs)

DEADLOCK AVOIDANCE (MUTEX)

Cases of deadlock: (lock() ⇒ block, wait)

- One thread tries to lock same mutex twice
- One thread tries to lock mtxs in the opposite order from another thread



Solution: No deadlock if all threads employ the same lock ordering (If so, then we just need to make sure we add lock and unlock at beg/end of code)

lock lock unlock unlock ← avoid spurious wakeup

If the program cannot enforce the lock order, use `pthread_mutex_trylock()` ⇒ if the lock is acquired, proceed; otherwise, try to lock it again

READER-WRITER LOCKS

(i.e. shared-exclusive locks)

Have 3 states:

① Locked in read (shared) mode (multiple threads)

② Locked in write (exclusive) mode (one thread at a time)

③ Unlocked mode

☆ More parallelism than mutexes if reading >>> modifying

`int pthread_rwlock_init(rwlock, attr);`

`int pthread_rwlock_destroy(rwlock);`

Same as mutexes, can init w/ attr=NULL or rwlock=`PT_RWLOCK_INITIALIZER`

Same for rdlock, wrlock, unlock, tryrdlock, trywrlock (ret EBUSY if cannot acquire to avoid deadlock)

Lock already set	Request read lock	Request write lock
None	OK	OK
Read	OK	Block
Write	Block	Block

CONDITION VARIABLES

Enable threads to block and test the condition atomically under the protection of a mutex until cond is satisfied (must acq mtx to change/eval state of cond)

POSIX CV support wait, notify sig/broadcast

- Wait: Mutex locked by thread is released and the thread blocks+waits on CV
- Notify: A thread that changes cond notifies CV to unblock waiting threads

`int pt_cond_init(cond, attr);` (init with `PT_COND_INITIALIZER`)

`int pt_cond_destroy(cond);`

`int pt_condattr_init(attr);` `int pt_condattr_destroy(attr);`

`int pt_condattr_getshared(attr, pshared);` `int pt_condattr_setshared(attr, pshared);`

Similarly has `PT_PROC_PRIVATE` / `PT_PROC_SHARED` for process-private vs process-shared

☆ Ret 0 or errno, nothing about cond

`int pt_cond_wait(cond, mutex);` (ATOMIC)

`int pt_cond_timedwait(cond, mutex, tsptr);`

wait() blocks caller+waits for CV cond to notify, must be called w/ mtx locked by caller or else undef behavior

If thread blocked by cond, then mtx unlocked+func blocks (what we want so we can detect cond change) (so we can sleep if (else cannot sleep holding mtx))

When wait() returns (i.e. wait is satisfied), mtx is locked again (caller may be unable to acquire mtx upon return)

"wait = atmic unlock+sleep (wait) → wake+lock"

timedwait() = wait() + ret error if time > tsptr

USE CASE OF POSIX CV (THREAD 1)

```
pthread_cond_t ready = PTHREAD_COND_INITIALIZER;
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
int cond = 0;
.. ..
pthread_mutex_lock(&lock);
while (!cond)
  pthread_cond_wait(&ready, &lock);
pthread_mutex_unlock(&lock);
```

blocks the

THREADS (PART 3) — UNIT 12.2+ PROCESS ENVIRONMENT (UNIT 13)

Shun/翔海 (@shuntmidx)

CONDITION VARIABLES CONTINUED

int pt_cond_signal(cond); "signals" a CV
int pt_cond_broadcast(cond); "bcasts" a CV
signal() unblocks (wakes up) ≥1 thd blocked on cond (picked by OS scheduler, o/w wait)
bcast() unblocks ALL thd blocked on cond
No effect if no thds are curr blocked on cond
Can be called by thd NOT holding mtx passed to pt_cond_wait()
☆Ret 0 or errno, nothing about cond

THREAD 2 IN EXAMPLE

```
pthread_mutex_lock(&lock);
pthread_cond_signal(&ready);
pthread_mutex_unlock(&lock);
```

SPURIOUS WAKEUPS

pt_cond_wait() ret values DO NOT IMPLY ANYTHING ABOUT COND ⇒ we may accidentally have cond==true (⇒ thd continue spurious) ⇒ wakeup
⇒ Need to check cond val!!

EXAMPLES

① Thd blocked in (Thd☆) pt_cond_wait can ret from call w/o call to pt_cond_signal/bcast
② (Thd☆) rets from a call to signal/bcast, but after reacquiring mutex, the predicate is no longer true

CASE ② EXAMPLE

- Spurious wakeup in the example (events in time order) — > assume thread 1 is blocked by pthread_cond_wait():
  - 1. Thread 2: unlocks mutex m
  - 2. Thread 3: locks mutex m
  - 3. Thread 3: changes a to 1 and unlocks mutex m
  - 4. Thread 2: signals the condition v
  - 5. Thread 1: returns from pthread_cond_wait(&v, &m)
  - **6. Thread 1: while(a) is true, calls pthread_cond_wait(&v, &m)**

```
Thread 1:
//wait for the "a != 0" condition
pthread_mutex_lock(&m);
while (a) ← infinite loop (needs if(a))
    pthread_cond_wait(&v, &m);
//do something if needed
pthread_mutex_unlock(&m);

Thread 2:
//notify waiting thread
pthread_mutex_lock(&m);
① → a = 0;
③ → pthread_mutex_unlock(&m);
    pthread_cond_signal(&v);
Then a supposed to be 0 but is 1
Thread 3:
② [ pthread_mutex_lock(&m);
    a = 1;
    pthread_mutex_unlock(&m);
```

THREADS AND FORK()

Thd calls fork() ⇒ only 1 thd in child
↳ Child inhrt all mtx, rwlock, CV
⇒ Mtx lock in parent thds = locked in child
BUT child only has 1 thd, so locks could be held by non-forked thds (⇒ deadlock in child)
∴ Need fork handlers to unlock locks

int pt_atfork(prepare, parent, child); (any can be NULL if unneeded)
Fork han execs when fork() is called
prepare: called before fork() creates child
parent: called in parent after child is created but before fork() returned
child: called in child before fork() ret
(e.g. prep: lock, par/child: unlock)

SIGWAIT() VS SIGSUSPEND()

sigwait is atmic, and doesn't call sig han (it intercepts delivered sigs)

THREAD-SAFE VS REENTRANT

Thread-safe: can be called by ≥1 thds at the same time (Unrelated to reentrant = "interrupt itself to run itself: OK")
E.g. Thd-safe / Non-reen: signal handler() {pt_mtx_lock(); smth; pt_mtx_unlock();}
NON-REEN if "smth" requires sig han

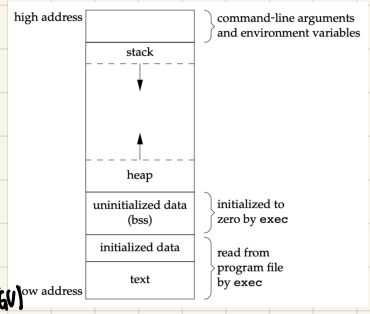
THREADS AND SIGNALS

Each thd has its own sig mask inhrtd from thd that calls pt_create(), and all thds of the proc share (t can edit)
disposition/action assoc w/ a signal
Signals related to hardware (e.g. ÷0) are delivered to the thd that caused it, o/w delivered to a random thd
☆Impossible to sigwait() SIGSTOP SIGKILL
☆≥1 thd blocked from sigwait() for one signal ⇒ only one thd will ret from delivery

int pthread_sigmask(how, set, oset); int sigwait(set, sig);
int pthread_kill(thread, signo); ⇒ Sends signo to thread
pt_sigmask is just like sigprocmask(), but rets errno directly.
↳ how = SIG_BLOCK, SIG_SETMASK, SIG_UNBLOCK (set is ignored for how)
↳ oset≠NULL ⇒ oset ← old mask (set==NULL & oset≠NULL ⇒ get curr mask)
(to eliminate undef behavior, make sure V sig ∈ set, block sig)
sigwait sends caller thread to wait() until a sig ∈ set is sent
↳ It atmicly unblocks set to "temp allow" sigs ∈ set: delivered
↳ Captures sigs w/ sig han ⇒ OG han not called
↳ Before ret signo in loc ref by sig, ① pending set -= sig ② Restore thread sig mask

MEMORY ADDRESSING (BIG PICTURE)

↳ Each proc has its own addr space
↳ Addr space = array of contiguous addr
↳ Memory mapping: The OS kernel connects a Proc Addr Space (PAS) (code-space to RAM) memory w/ main memory
↳ An addr ∈ either mapped or unmapped, and accessing unmapped addr = seg fault (SIGSEGV)



Any two proc in a prgm have the same set of addr but diff mappings
↳ Same memory of shared contents but different heap/stack

PAS are created in fork and exec but we can mmap for new PAS

The child get PAS mem maps like parent after fork via COW

MEMORY ALLOCATION FROM HEAP

malloc allocs from proc heap w/o sys calls
The heap is only expanded w/ sbrk(2) sys call

The heap contains record-keeping metadata to track alloc status and alloced mem buf
(keeps track but isn't memory for malloc)

MMAP

void* mmap(addr, len, prot, flags, fd, off);
Creates new mem map in PAS (not heap), similar to malloc mem buf. New mem region is file- (or device) or main mem-backed

Users can specify or let OS ker decide the addr to create a mem region.
Child inhrts mem regions created by mmap (for fork)
☆With mmap, a file becomes memory and not just copied to memory

addr = starting address for the new mapping (if NULL or used, the kernel picks on its own)
len = length of mapping (must be >0)
prot = desired memory protection for the mem map, MUST NOT BE MORE ACCESS THAN open() MODE OF FILE (e.g. PROT_WRITE for a read file) ⇒ SIGSEGV
Its values can be PROT_NONE or bitwise_or_of(PROT_READ, PROT_WRITE, PROT_EXEC)
fd = fd of file obj to be mapped, off = byte offset of the object
flags = Choose 1 of { MAP_SHARED = Visible to other procs, updates underlying files; MAP_PRIVATE = Priv COW map, not visible to other procs, doesn't update underlying files }
+ Bitwise OR: MAP_FIXED = Ker create mem region exactly at addr (For portability, don't use this)
Similar to normal malloc but MAP_ANONYMOUS = Allocs mem buf w/o file backing, contents init to 0
Since it's mmap, ITS MEM CAN BE SHARED ACROSS PROCS, unlike malloc

MUNMAP

int munmap(addr, len);
Deletes (unmaps) mem-mapped region, mem-mapped regions are auto unmapped when proc is term tho.

MPROTECT

int mprotect(addr, len, prot);
Changes perms on a mem region in PAS by setting it to prot

DAEMON PROCESS (UNIT 13) + PROGRAM LOADING/DYNAMIC LINKING (UNIT 14)

Shun/翔海 (@shuntmidx)

DAEMON PROCESS

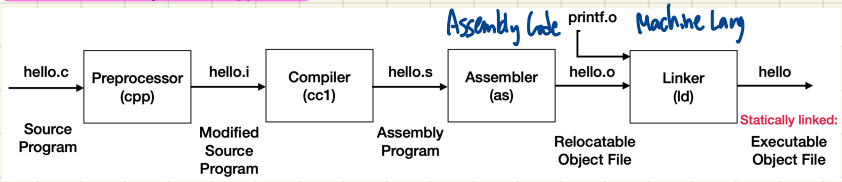
Daemon = Proc run in bg + during the entire sys uptime, + no logging to term
↳ Usually for admin tasks + servings to other procs

Usually, parent of daemon=init (PID 1)
Often run w/ root privileges and are not assoc w/ user session or terminal I/O (so no sig interactions)

There are 6 rules to code a daemon:
① File Creation Mask (umb mask=1 ⇒turn off)
⇒ umask(0) to enable all perms
② Fork and Exit
↳ Fork + Parent Exit Immediately
↳ If started from shell, the shell thinks cmd ended ⇒ returns control to user
↳ Daemon inhrt PGID but new PID ⇒ ensure not grp leader
③ Create a New Session
↳ Call setsid() ⇒ Becomes session leader, new leader of new proc group, no control to terminal
④ Working Directory changed to root to avoid sys maintenance
⑤ Close File Descriptors since it inhrts open fd from its parent
⑥ Standard Streams: Attach fd 0,1,2 to /dev/null

☆ Use syslog not terminal to log

GCC PROGRAM COMPILATION



LINKING

Collecting + combining various pieces of code into a single object file
↳ Static Linking: Performed at compile time
↳ Dynamic Linking: Performed at program load time

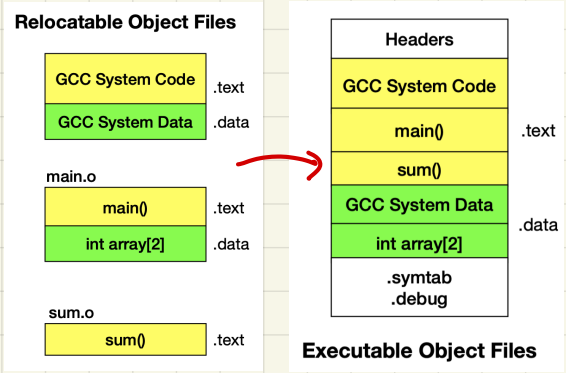
The linker makes sure the executable can access symbols
Consider obj file m,
↳ Global symbols: symbols defined by m that other obj files ref
↳ Global externals: symbols referenced by m but undefined by m
↳ Local symbols: defined + refed exclusively by m (e.g. local static var)

LOADING ELF EXECUTE OBJECT FILES

An ELF exec obj file includes all info needed to load prgm to mem + run it

When a proc calls exec(), the kernel invokes the loader to load the executable into the PAS. The prgm begins execution at entry point _start (setups + call main())

STATIC LINKING



DYNAMIC LINKING

Relies on shared libraries (end w/ .so), links them during program load time + runtime

It's performed by the dyn linker NOT kernel

⇒ Less memory, default for gcc
☆ Allows for libraries to be modified independently