Recall: Processes

- How to manage process state?
  - How to create a process?
  - How to exit from a process?

- Remember: Everything outside of the kernel is running in a process!
  - Including the shell! (Homework 2)

- Processes are created and managed... by processes!

Bootstrapping

- If processes are created by other processes, how does the first process start?

- First process is started by the kernel
  - Often configured as an argument to the kernel before the kernel boots
  - Often called the "init" process

- After this, all processes on the system are created by other processes

Process Management API

- exit – terminate a process
- fork – copy the current process
- exec – change the program being run by the current process
- wait – wait for a process to finish
- kill – send a signal (interrupt-like notification) to another process
- sigaction – set handlers for signals
Process Management API

- exit – terminate a process
- fork – copy the current process
- exec – change the program being run by the current process
- wait – wait for a process to finish
- kill – send a signal (interrupt-like notification) to another process
- sigaction – set handlers for signals

pid.c

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>

int main(int argc, char *argv[]) {
    /* get current processes PID */
    pid_t pid = getpid();
    printf("My pid: %d\n", pid);
    exit(0);
}
```

Q: What if we let main return without ever calling exit?
- The OS Library calls exit() for us!
- The entrypoint of the executable is in the OS library
- OS library calls main
- If main returns, OS library calls exit
- You’ll see this in Project 0: init.c

Creating Processes

- pid_t fork() – copy the current process
  - New process has different pid
  - New process contains a single thread
- Return value from fork(): pid (like an integer)
  - When > 0:
    » Running in (original) Parent process
    » return value is pid of new child
  - When = 0:
    » Running in new Child process
  - When < 0:
    » Error! Must handle somehow
    » Running in original process
- State of original process duplicated in both Parent and Child!
  - Address Space (Memory), File Descriptors (covered later), etc…
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

int main(int argc, char *argv[]) {
    pid_t cpid, mypid;
    pid_t pid = getpid();  /* get current processes PID */
    printf("Parent pid: \d\n", pid);
    cpid = fork();
    if (cpid > 0) {  /* Parent Process */
        mypid = getpid();
        printf("\ [%d] parent of \ [%d]\n", mypid, cpid);
    } else if (cpid == 0) {  /* Child Process */
        mypid = getpid();
        printf("\ [%d] child\n", mypid);
    } else {
        perror("Fork failed");
    }
}

Recall: a process consists of one or more threads executing in an address space
• Here, each process has a single thread
• These threads execute concurrently

What does this print?
Would adding the calls to sleep() matter?
### Process Management API

- **exit** – terminate a process
- **fork** – copy the current process
- **exec** – change the program being run by the current process
- **wait** – wait for a process to finish
- **kill** – send a signal (interrupt-like notification) to another process
- **sigaction** – set handlers for signals

---

### Starting new Program: variants of exec

```c
int cpid = fork();
if (cpid > 0) {
    /* Parent Process */
    tcpid = wait(&status);
} else if (cpid == 0) {
    /* Child Process */
    char *args[] = {"ls", "-l", NULL};
    execv("/bin/ls", args);
}
```

/* execv doesn't return when it works. 
So, if we got here, it failed! */

```c
perror("execv");
exit(1);
```

---

### fork2.c – parent waits for child to finish

```c
int status;
pid_t tcpid;

int cpid = fork();
if (cpid > 0) {
    /* Parent Process */
    mypid = getpid();
    printf("[%d] parent of [%d]\n", mypid, cpid);
    tcpid = wait(&status);
    printf("[%d] bye %d(%d)\n", mypid, tcpid, status);
} else if (cpid == 0) {
    /* Child Process */
    mypid = getpid();
    printf("[%d] child\n", mypid);
    exit(42);
}
```

---

### Process Management: The Shell pattern

```
main() {
    child
    child = fork();
    if (child == 0) {
        exec(...);
        exit(...);
    } else {
        wait(&stat);
        exit(...);
    }
    parent
    parent = fork();
    if (parent == 0) {
        exec(...);
        exit(...);
    } else {
        wait(&stat);
        exit(...);
    }
    fork
    fork = fork();
    if (fork == 0) {
        exec(...);
        exit(...);
    } else {
        wait(&stat);
        exit(...);
    }
    exec
    exec = fork();
    if (exec == 0) {
        exec(...);
        exit(...);
    } else {
        exit(...);
    }
}```
**Administrivia**

- Kubiatowicz Office Hours (Starting next week)
  - 2pm-3pm, Monday & Wednesday
- TOMORROW (Friday) is Drop Deadline! VERY HARD TO DROP LATER!
- Recommendation: Read assigned readings before lecture
- Starting next week, we will be adhering to strict slip-day policies for non-DSP students
  - Slip days are no-questions asked (or justification needed) extensions
  - Anything beyond this requires documentation (i.e. doctor’s note, etc)
  - If you run out of slip days, assignments will be discounted 10%/day
- You get 4 slip days for homework and 5 slip days for group projects
  - No project extensions on design documents, since we need to keep design reviews on track
- You should be going to sections – Important information covered in section
  - Any section will do until groups assigned
- Get finding groups of 4 people ASAP
  - Priority for same section; if cannot make this work, keep same TA
  - Remember: Your TA needs to see you in section!
- Midterm 1 will be on 2/16 from 7-9pm

**Process Management API**

- **exit** – terminate a process
- **fork** – copy the current process
- **exec** – change the program being run by the current process
- **wait** – wait for a process to finish
- **kill** – send a signal (interrupt-like notification) to another process
- **sigaction** – set handlers for signals

**inf_loop.c**

```c
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <signal.h>

void signal_callback_handler(int signum)
{
    printf("Caught signal!\n");
    exit(1);
}

int main()
{
    struct sigaction sa;
    sa.sa_flags = 0;
    sigemptyset(&sa.sa_mask);
    sa.sa_handler = signal_callback_handler;
    sigaction(SIGINT, &sa, NULL);
    while (1) {}  
}
```

Q: What would happen if the process receives a SIGINT signal, but does not register a signal handler?
A: The process dies!

For each signal, there is a default handler defined by the system

**Common POSIX Signals**

- **SIGINT** – control-C
- **SIGTERM** – default for kill shell command
- **SIGSTOP** – terminate/stop process
  - Can't be changed with sigaction
  - Why?
- **SIGKILL**

```c
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <signal.h>

void signal_callback_handler(int signum)
{
    printf("Caught signal!\n");
    exit(1);
}

int main()
{
    struct sigaction sa;
    sa.sa_flags = 0;
    sigemptyset(&sa.sa_mask);
    sa.sa_handler = signal_callback_handler;
    sigaction(SIGINT, &sa, NULL);
    while (1) {}  
}
```
Recall: UNIX System Structure

User Mode
- Applications (the users)
- Standard Libs
  - shells and commands
  - compilers and interpreters
  - system libraries

Kernel Mode
- system-call interface to the kernel
  - signals terminal handling
  - character I/O system terminal drivers
  - file system swapping block I/O system disk and tape drivers
  - CPU scheduling
  - page replacement
  - demand paging
  - virtual memory

Hardware
- terminal controllers
- device controllers
- memory controllers
- terminals
- disks and tapes
- physical memory

A Kind of Narrow Waist

- Compilers
- Word Processing
- Web Servers
- Databases
- Email
- Web Browsers
- Email
- Word Processing
- Portable OS Library
- System Call Interface
- Portable OS Kernel
- Platform support, Device Drivers
- x86
- PowerPC
- ARM
- Ethernet (1Gbs/10Gbs)
- 802.11 a/g/n/ac
- SCSI
- Thunderbolt
- Graphics
- PCI

Recall: OS Library (libc) Issues Syscalls

- OS Library: Code linked into the user-level application that provides a clean or more functional API to the user than just the raw syscalls
- Most of this code runs at user level, but makes syscalls (which run at kernel level)

Unix/POSIX Idea: Everything is a “File”

- Identical interface for:
  - Files on disk
  - Devices (terminals, printers, etc.)
  - Regular files on disk
  - Networking (sockets)
  - Local interprocess communication (pipes, sockets)
- Based on the system calls `open()`, `read()`, `write()`, and `close()`
- Additional: `ioctl()` for custom configuration that doesn’t quite fit
- Note that the “Everything is a File” idea was a radical idea when proposed
  - Dennis Ritchie and Ken Thompson described this idea in their seminal paper on UNIX called “The UNIX Time-Sharing System” from 1974
  - I posted this on the resources page if you are curious
Aside: POSIX interfaces

- POSIX: Portable Operating System Interface (for uniX?)
  - Interface for application programmers (mostly)
  - Defines the term "Unix," derived from AT&T Unix
  - Created to bring order to many Unix-derived OSes, so applications are portable
    » Partially available on non-Unix OSes, like Windows
  - Requires standard system call interface

The File System Abstraction

- File
  - Named collection of data in a file system
  - POSIX File data: sequence of bytes
    » Could be text, binary, serialized objects, …
  - File Metadata: information about the file
    » Size, Modification Time, Owner, Security info, Access control

- Directory
  - "Folder" containing files & directories
  - Hierarchical (graphical) naming
    » Path through the directory graph
    » Uniquely identifies a file or directory
      » /home/ff/cs162/public_html/fa14/index.html
  - Links and Volumes (later)

Connecting Processes, File Systems, and Users

- Every process has a current working directory (CWD)
  - Can be set with system call:
    int chdir(const char *path); //change CWD

  » Absolute paths ignore CWD
  - /home/oski/cs162

  » Relative paths are relative to CWD
  - index.html, ..index.html
    » Refers to index.html in current working directory
  - ..index.html
    » Refers to index.html in parent of current working directory
  - ~/index.html, ~cs162/index.html
    » Refers to index.html in the home directory

I/O and Storage Layers

- Application / Service

  Streams (buffered I/O)
  File Descriptors
    open(), read(), write(), close(), …
  Open File Descriptions
  Files/Directories/Indexes
  Commands and Data Transfers
  Disks, Flash, Controllers, DMA
### C High-Level File API – Streams

- Operates on “streams” – unformatted sequences of bytes (with text or binary data), with a position:

```
#include <stdio.h>
FILE *fopen( const char *filename, const char *mode );
int fclose( FILE *fp );
```

<table>
<thead>
<tr>
<th>Mode</th>
<th>Text</th>
<th>Binary</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>r</td>
<td>r</td>
<td>r</td>
<td>Open existing file for reading</td>
</tr>
<tr>
<td>w</td>
<td>w</td>
<td>w</td>
<td>Open for writing; created if does not exist</td>
</tr>
<tr>
<td>a</td>
<td>a</td>
<td>a</td>
<td>Open for appending; created if does not exist</td>
</tr>
<tr>
<td>r+</td>
<td>r+</td>
<td>r+</td>
<td>Open existing file for reading &amp; writing</td>
</tr>
<tr>
<td>w+</td>
<td>w+</td>
<td>w+</td>
<td>Open for reading &amp; writing; truncated to zero if exists, create otherwise</td>
</tr>
<tr>
<td>a+</td>
<td>a+</td>
<td>a+</td>
<td>Open for reading &amp; writing. Created if does not exist. Read from beginning, write as append</td>
</tr>
</tbody>
</table>

- Open stream represented by pointer to a `FILE` data structure
- Error reported by returning a NULL pointer

```
#include <stdio.h>
FILE *fopen( const char *filename, const char *mode );
int fclose( FILE *fp );
```

### C API Standard Streams – stdio.h

- Three predefined streams are opened implicitly when the program is executed.
  - `FILE *stdin` – normal source of input, can be redirected
  - `FILE *stdout` – normal source of output, can too
  - `FILE *stderr` – diagnostics and errors

- `STDIN / STDOUT enable composition in Unix`

- All can be redirected
  - `cat hello.txt | grep “World!”`
  - `cat's stdout goes to grep's stdin`

### C Streams: Char-by-Char I/O

```
// character oriented
int fputc( int c, FILE *fp ); // rtn c or EOF on err
int fputs( const char *s, FILE *fp ); // rtn > 0 or EOF
int fgetc( FILE *fp );
char *fgets( char *buf, int n, FILE *fp );
```

```
// block oriented
size_t fread(void *ptr, size_t size_of_elements,
             size_t number_of_elements, FILE *a_file);
size_t fwrite(const void *ptr, size_t size_of_elements,
              size_t number_of_elements, FILE *a_file);
```

```
// formatted
int fprintf(FILE *restrict stream, const char *restrict format, ...);
int fscanf(FILE *restrict stream, const char *restrict format, ...);
```

```
int main(void) {
    FILE* input = fopen("input.txt", "r");
    FILE* output = fopen("output.txt", "w");
    int c;
    c = fgetc(input);
    while (c != EOF) {
        fputc(output, c);
        c = fgetc(input);
    }
    fclose(input);
    fclose(output);
}
```
C High-Level File API

// character oriented
int fputc( int c, FILE *fp );  // rtn c or EOF on err
int fputs( const char *s, FILE *fp );  // rtn > 0 or EOF
int fgets( FILE * fp );
char *fgets( char *buf, int n, FILE *fp );

// block oriented
size_t fread(void *ptr, size_t size_of_elements,
size_t number_of_elements, FILE *a_file);
size_t fwrite(const void *ptr, size_t size_of_elements,
size_t number_of_elements, FILE *a_file);

// formatted
int fprintf(FILE *restrict stream, const char *restrict format, ...);
int fscanf(FILE *restrict stream, const char *restrict format, ...);

Aside: Check your Errors!

• Systems programmers should always be paranoid!
  – Otherwise you get intermittently buggy code
• We should really be writing things like:
  FILE* input = fopen("input.txt", "r");
  if (input == NULL) {
    // Prints our string and error msg.
    perror("Failed to open input file")
  }

• Be thorough about checking return values!
  – Want failures to be systematically caught and dealt with
• I may be a bit loose with error checking for examples in class (to keep short)
  – Do as I say, not as I show in class!

C Streams: Block-by-Block I/O

#define BUFFER_SIZE 1024
int main(void) {
    FILE* input = fopen("input.txt", "r");
    FILE* output = fopen("output.txt", "w");
    char buffer[BUFFER_SIZE];
    size_t length;
    length = fread(buffer, BUFFER_SIZE, sizeof(char), input);
    while (length > 0) {
        fwrite(buffer, length, sizeof(char), output);
        length = fread(buffer, BUFFER_SIZE, sizeof(char), input);
    }
    fclose(input);
    fclose(output);
}

C High-Level File API: Positioning The Pointer

int fseek(FILE *stream, long int offset, int whence);
long int ftell (FILE *stream)
void rewind (FILE *stream)

• For fseek(), the offset is interpreted based on the whence argument
  (constants in stdio.h):
  - SEEK_SET: Then offset interpreted from beginning (position 0)
  - SEEK_END: Then offset interpreted backwards from end of file
  - SEEK_CUR: Then offset interpreted from current position

• Overall preserves high-level abstraction of a uniform stream of objects
I/O and Storage Layers

Application / Service

High Level I/O
- Streams (buffered I/O)
- File Descriptors
  - open(), read(), write(), close(), ...
- Open File Descriptions
- Files/Directories/Indexes

Low Level I/O
- Syscall
- File System
- Commands and Data Transfers
  - Disks, Flash, Controllers, DMA

Low-Level File I/O: The RAW system-call interface

```c
#include <fcntl.h>
#include <unistd.h>
#include <sys/types.h>

int open (const char *filename, int flags [, mode_t mode])
int creat (const char *filename, mode_t mode)
int close (int filedes)
```

Bit vector of:
- Access modes (Rd, Wr, ...)
- Open Flags (Create, ...)
- Operating modes (Append, ...)

- Integer return from open() is a file descriptor
  - Error indicated by return < 0: the global errno variable set with error (see man pages)
- Operations on file descriptors:
  - Open system call created an open file description entry in system-wide table of open files
  - Open file description object in the kernel represents an instance of an open file
  - Why give user an integer instead of a pointer to the file description in kernel?

C Low-Level (pre-opened) Standard Descriptors

```c
#include <unistd.h>

#define STDIN_FILENO - macro has value 0
#define STDOUT_FILENO - macro has value 1
#define STDERR_FILENO - macro has value 2

int fileno (FILE *stream)
FILE *fdopen (int filedes, const char *opentype)
```

Low-Level File API

- Read data from open file using file descriptor:
  ```c
  ssize_t read (int filedes, void *buffer, size_t maxsize)
  ```
  - Reads up to maxsize bytes – might actually read less!
  - returns bytes read, 0 => EOF, -1 => error
- Write data to open file using file descriptor:
  ```c
  ssize_t write (int filedes, const void *buffer, size_t size)
  ```
  - returns number of bytes written
- Reposition file offset within kernel (this is independent of any position held by high-level FILE descriptor for this file!)
  ```c
  off_t lseek (int filedes, off_t offset, int whence)
  ```
Example: lowio.c

```c
int main() {
    char buf[1000];
    int fd = open("lowio.c", O_RDONLY, S_IRUSR | S_IWUSR);
    ssize_t rd = read(fd, buf, sizeof(buf));
    int err = close(fd);
    ssize_t wr = write(STDOUT_FILENO, buf, rd);
}
```

- How many bytes does this program read?

POSIX I/O: Design Patterns

- Open before use
  - Access control check, setup happens here
- Byte-oriented
  - Least common denominator
  - OS responsible for hiding the fact that real devices may not work this way (e.g. hard drive stores data in blocks)
- Explicit close

POSIX I/O: Kernel Buffering

- Reads are buffered inside kernel
  - Part of making everything byte-oriented
  - Process is blocked while waiting for device
  - Let other processes run while gathering result
- Writes are buffered inside kernel
  - Complete in background (more later on)
  - Return to user when data is "handed off" to kernel
- This buffering is part of global buffer management and caching for block devices (such as disks)
  - Items typically cached in quanta of disk block sizes
  - We will have many interesting things to say about this buffering when we dive into the kernel

Low-Level I/O: Other Operations

- Operations specific to terminals, devices, networking, …
  - e.g., ioctl
- Duplicating descriptors
  - int dup2(int old, int new);
  - int dup(int old);
- Pipes – channel
  - int pipe(int pipefd[2]);
  - Writes to pipefd[1] can be read from pipefd[0]
- File Locking
- Memory-Mapping Files
- Asynchronous I/O
High-Level vs. Low-Level File API

High-Level Operation:

```c
size_t fread(...) {
    Do some work like a normal fn...
    asm code ... syscall # into %eax
    put args into registers %ebx, ...
    special trap instruction
    get return values from regs
    Do some more work like a normal fn...
};
```

Kernel:

- Get args from regs
- Dispatch to system function
- Do the work to read from the file
- Store return value in %eax

Low-Level Operation:

```c
ssize_t read(...) {
    asm code ... syscall # into %eax
    put args into registers %ebx, ...
    special trap instruction
    get return values from regs
};
```

Kernel:

- Get args from regs
- Dispatch to system function
- Do the work to read from the file
- Store return value in %eax

- Streams are buffered in user memory:
  ```c
  printf("Beginning of line ");
  sleep(10); // sleep for 10 seconds
  printf("and end of line\n");
  ```
  Prints out everything at once

- Operations on file descriptors are visible immediately:
  ```c
  write(STDOUT_FILENO, "Beginning of line ", 18);
  sleep(10);
  write("and end of line \n", 16);
  ```
  Outputs "Beginning of line" 10 seconds earlier than "and end of line"

What’s below the surface ??

- Application / Service
- I/O Driver
- File System
- Syscall
- Low Level I/O
- High Level I/O
- Streams
- Handles
- Registers
- Descriptions
- Commands and Data Transfers
- Disks, Flash, Controllers, DMA

Recall: SYSCALL

- Low level lib parameters are set up in registers and syscall instruction is issued
  - A type of synchronous exception that enters well-defined entry points into kernel
What's below the surface ??

- File descriptor number - an int
- Application / Service
  - High Level I/O
  - Low Level I/O
  - Syscall
  - File System
  - I/O Driver
- commands and data transfers
  - Disks, Flash, Controllers, DMA

What's in an Open File Description?

Inside Kernel!
For our purposes, the two most important things are:
- Where to find the file data on disk
- The current position within the file

```c
ssize_t vfs_read(struct file *file, char __user *buf, size_t count, loff_t *pos)
{
    ssize_t ret;
    if (!(file->f_mode & FMODE_READ)) return -EBADF;
    if (!file->f_op || (!file->f_op->read && !file->f_op->aio_read)) return -EINVAL;
    if (unlikely(!access_ok(VERIFY_WRITE, buf, count))) return -EFAULT;
    ret = rw_verify_area(READ, file, pos, count);
    if (ret >= 0) {
        count = ret;
        if (file->f_op->read)
            ret = file->f_op->read(file, buf, count, pos);
        else
            ret = do_sync_read(file, buf, count, pos);
        if (ret > 0) {
            fsnotify_access(file->f_path.dentry);
            add_rchar(current, ret);
        }
        inc_syscr(current);
    }
    return ret;
}
```

File System: from syscall to driver

In fs/read_write.c

```c
ssize_t vfs_read(struct file *file, char __user *buf, size_t count, loff_t *pos)
{
    ssize_t ret;
    if (!(file->f_mode & FMODE_READ)) return -EBADF;
    if (!file->f_op || (!file->f_op->read && !file->f_op->aio_read)) return -EINVAL;
    if (unlikely(!access_ok(VERIFY_WRITE, buf, count))) return -EFAULT;
    ret = rw_verify_area(READ, file, pos, count);
    if (ret >= 0) {
        count = ret;
        if (file->f_op->read)
            ret = file->f_op->read(file, buf, count, pos);
        else
            ret = do_sync_read(file, buf, count, pos);
        if (ret > 0) {
            fsnotify_access(file->f_path.dentry);
            add_rchar(current, ret);
        }
        inc_syscr(current);
    }
    return ret;
}
```

Make sure we are allowed to read this file.
```c
ssize_t vfs_read(struct file *file, char __user *buf, size_t count, loff_t *pos) {
    ssize_t ret;
    if (!file->f_mode & MODE_READ) return -EBADF;
    if (!file->f_op || (!file->f_op->read && !file->f_op->aio_read))
        return -EINVAL;
    if (unlikely(!access_ok(VERIFY_WRITE, buf, count))) return -EFAULT;
    ret = rw_verify_area(READ, file, pos, count);
    if (ret > 0) {
        count = ret;
        if (file->f_op->read)
            ret = file->f_op->read(file, buf, count, pos);
        else
            ret = do_sync_read(file, buf, count, pos);
        if (ret > 0) {
            fsnotify_access(file->f_path.dentry);
            add_rchar(current, ret);
        }
        inc_syscr(current);
    }
    return ret;
}
```

Check if file has read methods

Check whether we read from a valid range in the file.

Check whether we can write to buf (e.g., buf is in the user space range)

If driver provide a read function (f_op->read) use it; otherwise use do_sync_read()
### File System: from syscall to driver

**In fs/read_write.c**

```c
ssize_t vfs_read(struct file *file, char __user *buf, size_t count, loff_t *pos)
{
    ssize_t ret;
    if (!file->f_mode & FMODE_READ)
        return -EBADF;
    if (!file->f_op || (!file->f_op->read && !file->f_op->aio_read))
        return -EINVAL;
    if (unlikely(!access_ok(VERIFY_WRITE, buf, count)))
        return -EFAULT;
    ret = rw_verify_area(READ, file, pos, count);
    if (ret >= 0)
    {
        count = ret;
        if (file->f_op->read)
            ret = file->f_op->read(file, buf, count, pos);
        else
            ret = do_sync_read(file, buf, count, pos);
        if (ret > 0)
        {
            fsnotify_access(file->f_path.dentry);
            add_rchar(current, ret);
        }
        inc_syscr(current);
    }
    return ret;
}
```

**Notes:**
- Notify the parent of this file that the file was read (see [http://www.fieldses.org/~bfields/kernel/vfs.txt](http://www.fieldses.org/~bfields/kernel/vfs.txt))
- Update the number of bytes read by 'current' task (for scheduling purposes)

---

### Lower Level Driver

- Associated with particular hardware device
- Registers / Unregisters itself with the kernel
- Handler functions for each of the file operations

**struct file_operations**

```c
struct file_operations
{
    loff_t (*llseek)(struct file *, loff_t, int);
    ssize_t (*read)(struct file *, char *, size_t, loff_t);
    ssize_t (*write)(struct file *, const char __user *, size_t, loff_t);
    ssize_t (*fsync)(struct file *, int);
    ssize_t (*release)(struct file *);
    int (*fadvise64)(struct file *, int, loff_t, int);
    int (*unmap)(struct file *, unsigned long, size_t);
    int (*flush)(struct file *, int);
    int (*f getattr)(struct file *, struct fsuid_t, loff_t, size_t);
    int (*f setattr)(struct file *, struct fsuid_t, loff_t, size_t);
    int (*f getxattr)(struct file *, const char *, size_t, int);
    int (*f setxattr)(struct file *, const char *, size_t, int);
    int (*f remount_ino)(struct file *, umode_t);
    int (*f chmod)(struct file *, umode_t);
    int (*f chown)(struct file *, umode_t);
    int (*f fstat64)(struct file *, struct stat64_t *);
    int (*f fstat)(struct file *, struct stat *);
    int (*f fstatfs)(struct file *, struct statfs *);
    int (*f getdents64)(struct file *, struct fsid_t, int, loff_t, size_t);
    int (*f getdents)(struct file *, struct fsid_t, int, loff_t, size_t);
}
```

**Notes:**
- Update the number of read syscalls by 'current' task (for scheduling purposes)
**Device Drivers**

- **Device Driver**: Device-specific code in the kernel that interacts directly with the device hardware
  - Supports a standard, internal interface
  - Same kernel I/O system can interact easily with different device drivers
  - Special device-specific configuration supported with the `ioctl()` system call
- **Device Drivers** typically divided into two pieces:
  - Top half: accessed in call path from system calls
    - Implements a set of standard, cross-device calls like `open()`, `close()`, `read()`, `write()`, `ioctl()`, `strategy()`
    - This is the kernel's interface to the device driver
    - Top half will start I/O to device, may put thread to sleep until finished
  - Bottom half: run as interrupt routine
    - Gets input or transfers next block of output
    - May wake sleeping threads if I/O now complete

**Life Cycle of An I/O Request**

**Communication between processes**

- Can we view files as communication channels?
  - `write(wfd, wbuf, wlen);`
  - `n = read(rfd, rbuf, rmax);`

- Producer and Consumer of a file may be distinct processes
  - May be separated in time (or not)
- However, what if data written once and consumed once?
  - Don’t we want something more like a queue?
  - Can still look like File I/O!

**Communication Across the world looks like file IO!**

- Connected queues over the Internet
  - But what’s the analog of open?
  - What is the namespace?
  - How are they connected in time?
**Request Response Protocol**

Client (issues requests)  
write(rqfd, rqbuf, buflen);

Server (performs operations)  
n = read(rfd, rbuf, rmax);

writes

requests  
wait

write(wfd, respbuf, len);
n = read(resfd, resbuf, resmax);

responses

**Request Response Protocol: Across Network**

Client (issues requests)  
write(rqfd, rqbuf, buflen);

Server (performs operations)  
n = read(rfd, rbuf, rmax);

writes

requests  
wait

write(wfd, respbuf, len);
n = read(resfd, resbuf, resmax);

responses

**The Socket Abstraction: Endpoint for Communication**

- **Key Idea:** Communication across the world looks like File I/O

write(wfd, wbuf, wlen);

n = read(rfd, rbuf, rmax);

- Sockets: Endpoint for Communication
  - Queues to temporarily hold results
- Connection: Two Sockets Connected Over the network ⇒ IPC over network!
  - How to open()?
  - What is the namespace?
  - How are they connected in time?

**Sockets: More Details**

- **Socket:** An abstraction for one endpoint of a network connection
  - Another mechanism for inter-process communication
  - Most operating systems (Linux, Mac OS X, Windows) provide this, even if they don’t copy rest of UNIX I/O
  - Standardized by POSIX
- First introduced in 4.2 BSD (Berkeley Standard Distribution) Unix
  - This release had some huge benefits (and excitement from potential users)
  - Runners waiting at release time to get release on tape and take to businesses
- Same abstraction for any kind of network
  - Local (within same machine)
  - The Internet (TCP/IP, UDP/IP)
  - Things “no one” uses anymore (OSI, Appletalk, IPX, …)
Sockets: More Details

- Looks just like a file with a file descriptor
  - Corresponds to a network connection (two queues)
  - write adds to output queue (queue of data destined for other side)
  - read removes from input queue (queue of data destined for this side)
  - Some operations do not work, e.g. lseek

- How can we use sockets to support real applications?
  - A bidirectional byte stream isn’t useful on its own…
  - May need messaging facility to partition stream into chunks
  - May need RPC facility to translate one environment to another and provide the abstraction of a function call over the network

Simple Example: Echo Server

Client (issues requests)  Server (services requests)

```c
void client(int sockfd) {
  int n;
  char sndbuf[MAXIN]; char rcvbuf[MAXOUT];
  while (1) {
    fgets(sndbuf,MAXIN,stdin); /* prompt */
    write(sockfd, sndbuf, strlen(sndbuf)+1);
    /* send (including null terminator) */
    memset(rcvbuf,0,MAXOUT);
    n=read(sockfd, rcvbuf, MAXOUT);
    /* receive */
    write(STDOUT_FILENO, rcvbuf, n);
    /* echo */
  }
}
```

```c
void server(int consockfd) {
  char reqbuf[MAXREQ];
  int n;
  while (1) {
    memset(reqbuf,0,MAXREQ);
    len = read(consockfd, reqbuf,MAXREQ);
    /* Recv */
    if (n <= 0) return;
    write(STDOUT_FILENO, reqbuf, n); /* echo */
    write(consockfd, reqbuf, n); /* echo */
  }
}
```
What Assumptions are we Making?

- Reliable
  - Write to a file => Read it back. Nothing is lost.
  - Write to a (TCP) socket => Read from the other side, same.

- In order (sequential stream)
  - Write X then write Y => read gets X then read gets Y

- When ready?
  - File read gets whatever is there at the time
    » Actually need to loop and read until we receive the terminator ("\0")
  - Assumes writing already took place
  - Blocks if nothing has arrived yet

Socket Creation

- File systems provide a collection of permanent objects in a structured name space:
  - Processes open, read/write/close them
  - Files exist independently of processes
  - Easy to name what file to open()

- Pipes: one-way communication between processes on same (physical) machine
  - Single queue
  - Created transiently by a call to pipe()
  - Passed from parent to children (descriptors inherited from parent process)

- Sockets: two-way communication between processes on same or different machine
  - Two queues (one in each direction)
  - Processes can be on separate machines: no common ancestor
  - How do we name the objects we are opening?
  - How do these completely independent programs know that the other wants to “talk” to them?

Namespaces for Communication over IP

- Hostname
  - www.eecs.berkeley.edu

- IP address
  - 128.32.244.172 (IPv4, 32-bit Integer)
  - 2607:f140:0:81::f (IPv6, 128-bit Integer)

- Port Number
  - 0-1023 are “well known” or “system” ports
    » Superuser privileges to bind to one
  - 1024 – 49151 are “registered” ports (registry)
    » Assigned by IANA for specific services
  - 49152–65535 (2^{15}+2^{14} to 2^{16}–1) are “dynamic” or “private”
    » Automatically allocated as “ephemeral ports”

Connection Setup over TCP/IP

- Special kind of socket: server socket
  - Has file descriptor
  - Can’t read or write
- Two operations:
  1. listen(): Start allowing clients to connect
  2. accept(): Create a new socket for a particular client
Connection Setup over TCP/IP

**Client Side**
- Connection request:
  1. Client IP addr
  2. Client Port
  3. Protocol (TCP/IP)

**Server Side**
- Server Listening:
  1. Server IP addr
  2. well-known port
  3. Protocol (TCP/IP)

Connection request:
- 1. Client IP addr
- 2. Client Port
- 3. Protocol (TCP/IP)

- New socket

5-Tuple identifies each connection:
1. Source IP Address
2. Destination IP Address
3. Source Port Number
4. Destination Port Number
5. Protocol (always TCP here)

- Often, Client Port "randomly" assigned
  - Done by OS during client socket setup
- Server Port often "well known"
  - 80 (web), 443 (secure web), 25 (sendmail), etc
  - Well-known ports from 0—1023

Conclusion (I)

- System Call Interface is "narrow waist" between user programs and kernel
- Streaming IO: modeled as a stream of bytes
  - Most streaming I/O functions start with "f" (like "fread")
  - Data buffered automatically by C-library functions
- Low-level I/O:
  - File descriptors are integers
  - Low-level I/O supported directly at system call level
- STDIN / STDOUT enable composition in Unix
  - Use of pipe symbols connects STDOUT and STDIN
  » find | grep | wc ...

Conclusion (II)

- Device Driver: Device-specific code in the kernel that interacts directly with the device hardware
  - Supports a standard, internal interface
  - Same kernel I/O system can interact easily with different device drivers
- File abstraction works for inter-processes communication (local or Internet)
- Socket: an abstraction of a network I/O queue
  - Mechanism for inter-process communication