Finish – Introduction to Process

Introduction to File Systems

David E. Culler
CS162 – Operating Systems and Systems Programming
Lecture 2’ & 3
Sept 5, 2014

Reading: A&D 3.1-3, 11.1-2
HW: 0 out, due 9/8
Recall: User/Kernal (Priviledged) Mode

User Mode
- syscall
- interrupt
- exception
- exec
- rtn
- rfi

Kernel Mode
- limited HW access
- full HW access

9/3/14
UCB CS162 Fa14 L2
Recall: Interrupt Vector

• Where else do you see this dispatch pattern?

```
intrpHandler_i () {
    ....
}
```
Simple B&B: User => Kernel

- How to return to system?
How to save registers and set up system stack?
Simple B&B: Switch User Process

• How to save registers and set up system stack?
Simple B&B: “resume”

- How to save registers and set up system stack?
What’s wrong with this simplistic address translation mechanism?
x86 – segments and stacks

Processor Registers

<table>
<thead>
<tr>
<th>CS</th>
<th>EIP</th>
</tr>
</thead>
<tbody>
<tr>
<td>SS</td>
<td>ESP</td>
</tr>
<tr>
<td>DS</td>
<td>EAX</td>
</tr>
<tr>
<td>ES</td>
<td>EBX</td>
</tr>
<tr>
<td></td>
<td>ECX</td>
</tr>
<tr>
<td></td>
<td>EDX</td>
</tr>
<tr>
<td></td>
<td>ESI</td>
</tr>
<tr>
<td></td>
<td>EDI</td>
</tr>
</tbody>
</table>

Start address, length and access rights associated with each segment
Virtual Address Translation

• Simpler, more useful schemes too!
• Give every process the illusion of its own BIG FLAT ADDRESS SPACE
  – Break it into pages
  – More on this later
Running Many Programs ???

• We have the basic mechanism to
  – switch between user processes and the kernel,
  – the kernel can switch among user processes,
  – Protect OS from user processes and processes from each other

• Questions ???
  • How do we decide which user process to run?
  • How do we represent user processes in the OS?
  • How do we pack up the process and set it aside?
  • How do we get a stack and heap for the kernel?
  • Aren’t we wasting are lot of memory?
  • ...

Process Control Block

• Kernel represents each process as a process control block (PCB)
  – Status (running, ready, blocked, ...)
  – Register state (when not ready)
  – Process ID (PID), User, Executable, Priority, ...
  – Execution time, ...
  – Memory space, translation, ...

• Kernel Scheduler maintains a data structure containing the PCBs

• Scheduling algorithm selects the next one to run
if ( readyProcesses(PCBs) ) {
    nextPCB = selectProcess(PCBs);
    run( nextPCB );
} else {
    run_idle_process();
}
Putting it together: web server

1. network socket read
2. copy arriving packet (DMA)
3. kernel copy
4. parse request
5. file read
6. disk request
7. disk data (DMA)
8. kernel copy
9. format reply
10. network socket write
11. kernel copy from user buffer into network buffer
12. format outgoing packet and DMA

syscall
interrupt
wait
interrupt

Hardware
Network Interface
Disk Interface

Server
request buffer
reply buffer

Kernel
network
read
wait

RTU
syscall
wait

 syscall
 syscall
 syscall
 syscall
4 OS concepts working together

• Privilege/User Mode
  – The hardware can operate in two modes, with only the “system” mode having the ability to access certain resources.

• Address Space
  – Programs execute in an *address space* that is distinct from the memory space of the physical machine

• Process
  – An instance of an executing program is a *process consisting of an address space and one or more threads of control*

• Protection
  – The OS and the hardware are protected from user programs and user programs are isolated from one another by *controlling the translation* from program virtual addresses to machine physical addresses
Introduction to File Systems

David E. Culler
CS162 – Operating Systems and Systems Programming
Lecture 3
Sept 5, 2014

Reading: A&D 3.1-3, 11.1-2
HW: 0 out, due 9/8
Objective of this lecture

• Show how Operating System functionality distributes across layers in the system.
• Introduce I/O & storage services – i.e., file systems
Reflecting on the process intro

• You said that applications request services from the operating system via `syscall`, but ...

• I’ve been writing all sort of useful applications and I never ever saw a “syscall” !!!

• That’s right.

• It was buried in the programming language runtime library (e.g., libc.a)

• … Layering
OS run-time library

Proc 1  Proc 2  Proc n

OS

Appln  login  Window Manager

OS library  OS library  OS library

OS
A Kind of Narrow Waist

Compilers

Word Processing

Web Browsers

Email

Web Servers

Databases

Application / Service

Portable OS Library

System Call Interface

Portable OS Kernel

Platform support, Device Drivers

User

System

Software

Portable OS

Hardware

x86

PowerPC

ARM

Ethernet (10/100/1000)

802.11 a/b/g/n

SCSI

IDE

Graphics

PCI
Key Unix I/O Design Concepts

• Uniformity
  – file operations, device I/O, and interprocess communication through open, read/write, close
  – Allows simple composition of programs
    • find | grep | wc ...

• Open before use
  – Provides opportunity for access control and arbitration
  – Sets up the underlying machinery, i.e., data structures

• Byte-oriented
  – Even if blocks are transferred, addressing is in bytes

• Kernel buffered reads
  – Streaming and block devices looks the same, read blocks yielding processor to other task

• Kernel buffered writes
  – Completion of out-going transfer decoupled from the application, allowing it to continue

• Explicit close
I/O & Storage Layers

Application / Service

High Level I/O

Low Level I/O

Syscall

File System

I/O Driver

streams

handles

registers

descriptors

Commands and Data Transfers

Disks, Flash, Controllers, DMA
The file system abstraction

- **File**
  - Named collection of data in a file system
  - File data
    - Text, binary, linearized objects
  - File Metadata: information about the file
    - Size, Modification Time, Owner, Security info
    - Basis for 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)
C high level File API – streams (review)

• Operate on “streams” - sequence of bytes, whether text or data, with a position

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

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

• Process has a ‘current working directory’

• Absolute Paths
  – /home/ff/cs152

• Relative paths
  – index.html, ./index.html - current WD
  – ../index.html - parent of current WD
  – ~, ~cs152 - home directory
C API Standard Streams

• 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
C high level File API – stream ops

#include <stdio.h>

// 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, ...);
C Stream API positioning

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

• Preserves high level abstraction of a uniform stream of objects
What’s below the surface ??

Application / Service

High Level I/O

Low Level I/O

Syscall

File System

I/O Driver

streams

handles

registers

descriptors

Commands and Data Transfers

Disks, Flash, Controllers, DMA

9/5/14
C Low level I/O

- Operations on File Descriptors – as OS object representing the state of a file
  - User has a “handle” on the descriptor

```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 (Appends, ...)

Bit vector of Permission Bits:
- User|Group|Other X R|W|X

C Low Level: standard descriptors

#include <unistd.h>

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

int fileno (FILE *stream)

FILE * fdopen (int filedes, const char *opentype)

• Crossing levels: File descriptors vs. streams
• Don’t mix them!
C Low Level Operations

ssize_t read (int filedes, void *buffer, size_t maxsize)
- returns bytes read, 0 => EOF, -1 => error
ssize_t write (int filedes, const void *buffer, size_t size)
- returns bytes written

off_t lseek (int filedes, off_t offset, int whence)

int fsync (int fildes) – wait for i/o to finish
void sync (void) – wait for ALL to finish

• When write returns, data is on its way to disk and can be read, but it may not actually be permanent!
And lots more!

- TTYs versus files
- Memory mapped files
- File Locking
- Asynchronous I/O
- Generic I/O Control Operations
- Duplicating descriptors

```c
int dup2 (int old, int new)
int dup (int old)
```
What’s below the surface ??

Application / Service

High Level I/O

Low Level I/O

Syscall

File System

I/O Driver

streams

handles

registers

descriptors

Commands and Data Transfers

Disks, Flash, Controllers, DMA
• Low level lib parameters are set up in registers and syscall instruction is issued
Internal OS File Descriptor

• Internal Data Structure describing everything about the file
  – Where it resides
  – Its status
  – How to access it
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;
}
```
Low Level Driver

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

```
struct file_operations {
    struct module *owner;
    loff_t (EINVAL) (struct file *, loff_t, int);
    ssize_t (S妥善) (struct file *, char __user *, size_t, loff_t *);
    ssize_t (S妥善) (struct file *, const char __user *, size_t, loff_t *);
    ssize_t (S妥善) (struct kiocb *, const struct iovec *, unsigned long, loff_t);
    ssize_t (S妥善) (struct kiocb *, const struct iovec *, unsigned long, loff_t);
    int (*readdir) (struct file *, void *, filldir_t);
    unsigned int (*poll) (struct file *, struct poll_table_struct *);
    int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
    int (*mmap) (struct file *, struct vm_area_struct *);
    int (*open) (struct inode *, struct file *);
    int (*flush) (struct file *, fl_owner_t id);
    int (*release) (struct inode *, struct file *);
    int (*fsync) (struct file *, struct dentry *, int datasync);
    int (*fasync) (int, struct file *, int);
    int (*flock) (struct file *, int, struct file_lock *);
    [...]}
```
So what happens when you `fgetc`?

Application / Service

- High Level I/O
- Low Level I/O
- Syscall
- File System
- I/O Driver

- streams
- handles
- registers

Commands and Data Transfers

Disks, Flash, Controllers, DMA