Creating the Process Abstraction

Sam Kumar
CS 162: Operating Systems and System Programming
Lecture 7
https://inst.eecs.berkeley.edu/~cs162/su20

Read: A&D 3.5-6,
A fork() in the road
Recall: Endianness

- For a byte-address machine, which end of a machine-recognized object (e.g., int) does its byte-address refer to?
  - Big Endian: address is the most-significant bits
  - Little Endian: address is the least-significant bits

```c
int main(int argc, char *argv[]) {
    int val = 0x12345678;
    int i;
    printf("val = \%x\n", val);
    for (i = 0; i < sizeof(val); i++) {
        printf("val[%d] = \%x\n", i, ((uint8_t *) &val)[i]);
    }
}
```

(base) CullerMac19:code09 culler$ ./endian
val = 12345678
val[0] = 78
val[1] = 56
val[2] = 34
val[3] = 12
Recall: What About Richer Objects?

• Consider `word_count_t` of Homework 0 and 1 ...
• Each element contains:
  • An int
  • A `pointer` to a string (of some length)
  • A `pointer` to the next element
• `fprintf_words` writes these as a sequence of lines (character strings with \n) to a file stream
• What if you wanted to write the whole list as a binary object (and read it back as one)?
  • How do you represent the string?
  • Does it make any sense to write the pointer?
Recall: Remote Procedure Calls

Client (caller)

\[ r = f(v1, v2); \]

Server (callee)

\[ \text{res}_t \ f(a1, a2) \]

Machine A

Machine B

Packet Handler

Network

Stub

bundle args

send

receive

unbundle args

unbundle ret vals

bundle args

send

receive

unbundle ret vals

return

return

call

call
Recall: DNS

Hostname Resolution Request

Hostname Resolution Response (IP)

DNS Server

169.229.131.81

128.32.61.103

128.32.139.48

128.32.139.48
HTTP

• Application protocol for The Web
  • Retrieve a specific object, upload data, etc.
• Runs on top of TCP (sockets)
• Like any protocol, stipulates:
  • **Syntax**: Content sent over socket connection
  • **Semantics**: Meaning of a message
    • Valid replies and actions taken upon message receipt
• Arguably a primitive form of RPC
  • Parsing text, Hardcoded operations
  • No registry of available functions
  • No formal marshal/unmarshal
  • REST calls get a lot closer ...
HTTP “RPC”

GET /search.html

200 OK
< HTML Content>
HTTP “RPC”

POST /users/oksi/photos
<image content>

201 Created
HTTP Messages

• Text-based: We just send character strings over our TCP socket connection
• To make a request, browser might write something like the following on a socket:

```
GET /hello.html HTTP/1.0\r\nHost: 128.32.4.8:8000\r\nAccept: text/html\r\nUser-Agent: Chrome/45.0.2454.93\r\nAccept-Language: en-US,en;q=0.8\r\n```

7/1/2020

Kumar CS 162 at UC Berkeley, Summer 2020
HTTP Messages

• Text-based: We just send strings over our TCP socket connection
• We then read the following response from the web server:

HTTP/1.0 200 OK
Content-Type: text/html
Content-Length: 128

<html>
<body>
<h1>Hello World</h1>
<p>Hello, World!</p>
</body>
</html>
HTTP and State

• Remember this situation?
  ```c
  remoteFile *rf = remoteio_open(ctx, "oski.txt");
  remoteio_puts(ctx, rf, "Bear\n");
  remoteio_close(ctx, rf);
  ```

• Client fails: does file stay open forever?
• Server maintains state between requests from client
HTTP and State

• HTTP avoids this issue – **stateless protocol**

• Each request is self-contained
  • Treated independently of all other requests
  • Even previous requests from same client!

• So how do we get a **session**?
  • Client stores a unique ID locally – a **cookie**
  • Client adds this to each request so server can customize its response
REST Calls over HTTP?

HTTP GET
http://www.appdomain.com/dirs/mkdir?name=cs162&mode=tmp
RPC Locally

• Doesn’t need to be between different machines
• Could just be different address spaces (processes)

• Gives **location transparency**
  • Move service implementation to wherever is convenient
  • Client runs same old RPC code

• Much faster implementations available locally
  • (Local) Inter-process communication
  • We’ll see several techniques later on
Recall: Process

• Definition: execution environment with restricted rights
  • One or more threads executing in a single address space
  • Owns file descriptors, network connections

• Instance of a running program
  • When you run an executable, it runs in its own process
  • Application: one or more processes working together

• Protected from each other; OS protected from them

• In modern OSes, anything that runs outside of the kernel runs in a process
Today: How Does the OS Support the Process Abstraction?

• How does the kernel build the abstractions we have studied?
  • Dual-Mode Operation and Address Spaces?
  • Threads?
  • File I/O?

• What role does hardware play in serving syscalls/interrupts/traps?

• How is the kernel structured?

• And, along the way, getting you ready to tackle Project 1 ...
Today: How Does the OS Support the Process Abstraction?

• Support for threads and kernel structure
• Memory layout
• Support for process operations
• Support for I/O
• Influence of IPC/RPC on kernel structure
Today: How Does the OS Support the Process Abstraction?

• Support for threads and kernel structure
• Memory layout
• Support for process operations
• Support for I/O
• Influence of IPC/RPC on kernel structure
Recall: Kernel Stacks

- Interrupt handlers want a stack
- System call handlers want a stack
- Can't just use the user stack [why?]

- One Solution: two-stack model
  - Each thread has user stack and a kernel stack
  - Kernel stack stores users registers during an exception
  - Kernel stack used to execute exception handler in the kernel
Recall: Single and Multithreaded Processes

- Threads encapsulate concurrency
  - “Active” component
- Address space encapsulate protection:
  - “Passive” component
  - Keeps bugs from crashing the entire system
- Why have multiple threads per address space?
User/Kernel Threading Models

Almost all current implementations

Simple One-to-One Threading Model

Many-to-One

Many-to-Many
Thread State in the Kernel

• For every thread in a process, the kernel maintains:
  • The thread’s TCB
  • A kernel stack used for syscalls/interrupts/traps

• Additionally, some threads just do work in the kernel
  • Still has TCB
  • Still has kernel stack
  • But not part of any process, and never executes in user mode
**Pintos Thread**

- Single page (4 KiB)
  - Stack growing from the top (high addresses)
  - `struct thread` at the bottom (low addresses)

- `struct thread` defines the TCB structure in Pintos

![Diagram of Pintos Thread]

---

Pintos: thread.c
In Pintos, Processes are Single-Threaded

• Processes can contain exactly one thread, for simplicity
• Approach used by older systems
Pintos Thread

• Single page (4 KiB)
  • Stack growing from the top (high addresses)
  • `struct thread` at the bottom (low addresses)

• `struct thread` defines the TCB structure and PCB structure in Pintos

Pintos: `thread.c`
Linux “Task”

- 2 pages (8 KiB)
  - Stack and thread information on opposite sides
  - Containing stack and thread information + process descriptor

```
stack
  regs
sp
thread_info
  ...
  status
  flags
  *task
```

```
task_struct
  (process descr)
  state
  priority
  pid
  address space
  ...
  list elems
```
Multithreaded Processes (not in Pintos)

• Traditional implementation strategy:
  • One PCB (process struct) per process
  • Each PCB contains (or stores pointers to) each thread’s TCB

• Linux’s strategy:
  • One task_struct per thread
  • Threads belonging to the same process happen to share some resources
    • Like address space, file descriptor table, etc.

• To what extent does this actually matter?
Process Creation (Projects 0 and 1)

- Allocate and initialize Process object
- Allocate and initialize kernel thread mini-stack and associated Thread object
- Allocate and initialize page table for process
  - Referenced by process object
- Load code and static data into user pages
- Build initial User Stack
  - Initial register contents, argv, ...
- Schedule (post) process/thread for execution
- ...
- Eventually *switch* to user mode (switching to user stack and registers) ...
Aside: Polymorphic Linked Lists in C

- Many places in the kernel need to maintain a “list of X”
  - This is tricky in C, which has no polymorphism
  - Essentially adding an interface to a package
- In Linux and Pintos this is done by embedding a list_elem in the struct
  - Macros allow shift of view between object and list
  - You saw this in Homework 1
Kernel Structure So Far (1/3)
Kernel Structure So Far (2/3)
These two threads:
• Are used internally by the kernel
• Don’t correspond to any particular user thread or process
Today: How Does the OS Support the Process Abstraction?

• Support for threads and kernel structure
• Memory layout
• Support for process operations
• Support for I/O
• Influence of IPC/RPC on kernel structure
Recall: Process Control Block

- Kernel representation of each process
  - Status (running, ready, blocked)
  - Register state (if not running)
  - Thread control block(s)
  - Process ID
  - Execution time
  - Address space
  - Open files, etc.

How is this represented?
Recall: Address Space

- Program operates in an address space that is distinct from the physical memory space of the machine.
Understanding “Address Space”

- Page table is the primary mechanism
- Privilege Level determines which regions can be accessed
  - Which entries can be used
- System (PL=0) can access all, User (PL=3) only part
- Each process has its own address space
- The “System” part of all of them is the same

All system threads share the same system address space and same memory
Page Table Mapping (Rough Idea)

- Prog 1: Virtual Address Space 1
  - Code
  - Data
  - Heap
  - Stack
  - Translation Map 1

- Prog 2: Virtual Address Space 2
  - Code
  - Data
  - Heap
  - Stack
  - Translation Map 2

(user process view of memory)

Physical Address Space

- Code
- Static Data
- Heap
- Stack

Translation Map 1

Translation Map 2

0x000...

0xFFF...
User Process View of Memory

Process Virtual Address Space

Physical Memory

Processor registers

sp
ip

0x00000000
0xc0000000
0x08048000
0xffffffff

kernel
argv
stack
heap
user data
user code

Page Table

Page
Processor Mode (Privilege Level)

Process Virtual Address Space

Physical Memory

Page Table

Page

Processor registers

sp

ip

CPL: 3 - user

0x00000000

0x08048000

0x00000000

argv

stack

heap

user data

user code

kernel

0xffffffff

0xc0000000

argv

stack

heap

user data

user code

CPL: 3 - user

0x00000000

0x08048000

0x00000000

argv

stack

heap

user data

user code

kernel

0xffffffff

0xc0000000
Aside: x86 (32-bit) Page Table Entry

- Controls many aspects of access
- Later – discuss page table organization
  - For 32 (64?) bit VAS, how large? vs size of memory?
  - Used sparsely

Page Table

![Page Table Diagram](image)

Pintos: page_dir.c
User → Kernel

Process Virtual Address Space

Kernel:
- Stack (sp, ip)
- Heap
- User code
- User data
- Argument Vector (argv)

CPL: 3 - User

Physical Memory

Page Table
- u/s
User → Kernel

Process Virtual Address Space

Kernel:
- 0xffffffff
- kernel
- ker data
- ker code

User:
- argv
- stack
- heap
- user data
- user code

CPL: 0 - sys

Processor registers:
- sp
- ip

Physical Memory:
- Page Table
- u/s
- Page
Page Table Resides in Memory*

* In the simplest case. Actually more complex. More later.

Process Virtual Address Space

- kernel
  - ker data
  - ker code
- stack
  - argv
- heap
  - user data
  - user code

Processor registers

- sp
- ip

CPL: 0 - sys

PTBR:

Physical Memory

- Page Table
  - Page

In the simplest case. Actually more complex. More later.
Kernel Portion of Address Space

- Kernel memory is mapped into address space of every process
- Contains the kernel code
  - Loaded when the machine booted
- Explicitly mapped to physical memory
  - OS creates the page table
- Used to contain all kernel data structures
  - Lists of processes/threads
  - Page tables
  - Open file descriptions, sockets, ttys, ...
- Kernel stack for each thread
1 Kernel Code, Many Kernel Stacks

Process Virtual Address Space

Physical Memory

Processor registers

CPL: 0 - sys

PTBR:

Kernel Code, Many Kernel Stacks

0xffffffff

kernel
derdata
derkern
code

argv

stack

heap

user datan
n
user code

0xc0000000

argv

stack

heap

user data

user code

0x08048000

0x00000000

0x80804800

Page Table

u/s

Page

Physical Memory

Processor registers

CPL: 0 - sys

PTBR:

Kernel Code, Many Kernel Stacks

0xffffffff

kernel
derdata
derkern
code

argv

stack

heap

user datan
n
user code

0xc0000000

argv

stack

heap

user data

user code

0x08048000

0x00000000

0x80804800

Page Table

u/s

Page

Physical Memory

Processor registers

CPL: 0 - sys

PTBR:
How to Get to the Correct Kernel Stack?

• The hardware helps us out!
Recall: Where do User → Kernel Mode Transfers Go?

- Cannot let user programs specify the exact address!

- **Solution: Interrupt Vector**
  - OS kernel specifies a set of functions that are *entrypoints* to kernel mode
  - Appropriate function is chosen depending on the type of transition
    - Interrupt Number (i)
    - OS may do additional *dispatch*

```c
intrpHandler_i () {
    ...
}
```
Hardware Support for Switching Stacks

• Syscall/Intr (U → K)
  • PL 3 → 0;
  • TSS ← EFLAGS, CS:EIP;
  • SS:SP ← k-thread stack (TSS PL 0);
  • push (old) SS:ESP onto (new) k-stack
  • push (old) eflags, cs:eip, <err>
  • CS:EIP ← <k target handler>

• Then
  • Handler then saves other regs, etc
  • Does all its works, possibly choosing other threads, changing PTBR (CR3)
  • kernel thread has set up user GPRs

• iret (K → U)
  • PL 0 → 3;
  • Eflags, CS:EIP ← popped off k-stack
  • SS:SP ← user thread stack (TSS PL 3);
Recall: The Process

• Definition: execution environment with restricted rights
  • Address Space with One or More Threads
    • Page table per process!
  • Owns memory (mapped pages)
  • Owns file descriptors, file system context, …
  • Encapsulates one or more threads sharing process resources

• Application program executes as a process
  • Complex applications can fork/exec child processes [later]

• Why processes?
  • Protected from each other. OS Protected from them.
  • Execute concurrently [ trade-offs with threads? later ]
  • Basic unit OS deals with
Context Switch

• Diagram assumes single-threaded processes

• For multi-threaded process, substitute process → thread and PCB → TCB
Recall: Scheduling

• Scheduling: Mechanism for deciding which processes/threads receive the CPU

• Lots of different scheduling policies provide ...
  • Fairness or
  • Realtime guarantees or
  • Latency optimization or ...

if ( readyThreads(TCBs) ) {
    nextTCB = selectTCB(TCBs);
    run( nextTCB );
} else {
    run_idle_thread();
}
Scheduling: All About Queues

- TCBs move from queue to queue
- Scheduling: which order to remove from queue of “ready” threads
Announcements

• Homework 2 out, due Monday

• Project 0 due tonight tomorrow night

• Project 1 will be out soon

• Drop Deadline with Refund on Thursday

• Project Groups Due Friday
Today: How Does the OS Support the Process Abstraction?

• Support for threads and kernel structure
• Memory layout
• **Support for process operations**
• Support for I/O
• Influence of IPC/RPC on kernel structure
Operations on Process State

- exec()
- fork()
- open()/close()
- wait()
Recall: Running a Program

• Create OS “PCB”, address space, stack and heap
• Load instruction and data segments of executable file into memory
• “Transfer control to program”
• Provide services to program
• While protecting OS and program
Recall: Running a Program

- Create OS “PCB”, address space, stack and heap
- Load instruction and data segments of executable file into memory
- “Transfer control” to program
- Provide services to program
- While protecting OS and program

Program Source

Executable

Compiler and Linker

OS Loader

fork()

exec(...)

foo.c

Program Source

fork()

Compiler and Linker

OS Loader

fork()

exec(...)
How to `fork()` efficiently?

- Alias the pages
  - Same physical address!
  - If we stopped here, the data would be shared (not what we want)

- Mark PTEs read-only
  - If a process tries to write → trap to the OS

- On first write to a page after `fork()`, kernel copies the page, marks PTEs as writeable

- Illusion of separate memory, but really aliased until first write

Pintos doesn’t support `fork()`, just `CreateProcess()`
Recall: Open File Description is *Aliased*

- Process 1
  - File Descriptors: 3
  - Open File Description: File: foo.txt, Position: 300

- Process 2
  - File Descriptors: 3

```
read(3, buf, 100)
close(3)
read(3, buf, 100)
```
Solution: Reference Counting

User Space

Kernel Space

Not shown: Initially contains 0, 1, and 2 (stdin, stdout, stderr)

File Descriptors
3

Thread’s Regs
...

Address Space (Memory)

Open File Description
File: foo.txt
Position: 300
Reference Count: 2
Lock

File Descriptors
3

Thread’s Regs
...

Address Space (Memory)

Process 1

read(3, buf, 100)

Process 2

read(3, buf, 100)
**Solution: Reference Counting**

User Space

Kernel Space

---

**Process 1**

- Address Space (Memory)
- Thread’s Regs
- File Descriptors
  - 3

---

**Process 2**

- Address Space (Memory)
- Thread’s Regs
- File Descriptors
  - 3

---

Open File Description

- File: foo.txt
- Position: 300
- Reference Count: 1
- Lock

---

**Operations**

- `read(3, buf, 100)`
- `close(3)`

---

Not shown: Initially contains 0, 1, and 2 (stdin, stdout, stderr)
Solution: Reference Counting

read(3, buf, 100)
close(3)

Process 1

Address Space (Memory)

File Descriptors

Not shown: Initially contains 0, 1, and 2 (stdin, stdout, stderr)

Process 2

Address Space (Memory)

File Descriptors

Open File Description

File: foo.txt
Position: 300
Reference Count: 1
Lock

File: foo.txt
Position: 300
**Solution: Reference Counting**

```plaintext
read(3, buf, 100) close(3)
```

Process 1

- **Thread’s Regs**
- **Address Space (Memory)**
- **File Descriptors**

User Space

Kernel Space

Not shown: Initially contains 0, 1, and 2 (stdin, stdout, stderr)

Process 2

- **Thread’s Regs**
- **Address Space (Memory)**
- **File Descriptors**

Open File Description

- File: foo.txt
- Position: 300
- Reference Count: 1
- Lock

```
read(3, buf, 100) close(3)
```
Solution: Reference Counting

read(3, buf, 100)
close(3)

Process 1

Thread’s Regs
...

Address Space (Memory)

File Descriptors

Not shown: Initially contains 0, 1, and 2 (stdin, stdout, stderr)

Process 2

Thread’s Regs
...

Address Space (Memory)

File Descriptors

Open File Description
File: foo.txt
Position: 300
Reference Count: 0
Lock

read(3, buf, 100)
close(3)

read(3, buf, 100)
close(3)
Solution: Reference Counting

read(3, buf, 100)
close(3)

read(3, buf, 100)
close(3)

User Space
Kernel Space

Not shown: Initially contains 0, 1, and 2 (stdin, stdout, stderr)
What about `wait()`?

- The parent process needs to get the exit code
- The following events may happen in any order (or concurrently)
  - Parent process calls `wait()` (or `exit()`)
  - Child process calls `exit()`

- Where should the child put its exit code?
  - Needs to work even if the parent has exited
- Where should the parent search for the exit code?
  - Needs to work even if the child has exited already
Today: How Does the OS Support the Process Abstraction?

• Support for threads and kernel structure
• Memory layout
• Support for process operations
• Support for I/O
• Influence of IPC/RPC on kernel structure
Recall: I/O and Storage Layers

What we’ve covered so far...
- High Level I/O
  - Streams
  - File Descriptors
    - open(), read(), write(), close(), ...
  - Open File Descriptions
- Low Level I/O
  - Syscall
- File System
  - Files/Directories/Indexes
- I/O Driver
  - Commands and Data Transfers
    - Disks, Flash, Controllers, DMA

What we’ll peek at today
- Open File Descriptions
length = read(input_fd, buffer, BUFFER_SIZE);

ssize_t read(int, void *, size_t){
    marshal args into registers
    issue syscall
    register result of syscall to rtn value
};

void syscall_handler (struct intr_frame *f) {
    unmarshal call#, args from regs
    dispatch : handlers[call#](args)
    marshal results fo syscall ret
}

ssize_t vfs_read(struct file *file, char __user *buf, size_t count, loff_t *pos) {
    User Process/File System relationship
    call device driver to do the work
}

Exception U→K, interrupt processing

User App
User library

Application / Service
High Level I/O
Low Level I/O
Syscall
File System
I/O Driver

Device Driver
User library

Layers...
Low-Level Driver

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

```c
struct file_operations {
    struct module *owner;
    loff_t (==>seek) (struct file *, loff_t, int);
    ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
    ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
    ssize_t (*aio_read) (struct kiocb *, const struct iov的学生, unsigned long, loff_t);
    ssize_t (*aio_write) (struct kiocb *, const struct iov的学生, 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 (*mm_map) (struct file *, struct vm_area_struct *);
    int (*open) (struct inode *, struct file *);
    int (*flush) (struct file *, f_user_private_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 *);
};
```
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;
}

- Read up to "count" bytes from "file" starting from "pos" into "buf".
- Return error or number of bytes read.
File System: From Syscall to Driver

```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

Linux: fs/read_write.c
File System: From Syscall to Driver

 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**

```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;
}
```

- **Check whether we can write to buf (e.g., buf is in the user space range)**
- **unlikely(): hint to branch prediction this condition is unlikely**

Linux: fs/read_write.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;
}

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

Linux: fs/read_write.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;
}
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;
}
```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;
}
```

Update the number of bytes read by “current” task (for scheduling purposes)

Linux: fs/read_write.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;
}
Device Drivers

• 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
Recall: Inter-Process Communication (IPC)

• Mechanism to create communication channel between distinct processes
  • Same or different machines, same or different programming language...

• Requires serialization format understood by both

• Failure in one process isolated from the other
  • Sharing is done in a controlled way through IPC
  • Still have to be careful handling what is received via IPC

• Many uses and interaction patterns
  • Logging process, window management, ...
  • Potentially allows us to move some system functions outside of kernel to userspace
Device Driver
Top Half

Device Driver
Bottom Half

Device Hardware

Kernel I/O Subsystem

User Program
Today: How Does the OS Support the Process Abstraction?

• Support for threads and kernel structure
• Memory layout
• Support for process operations
• Support for I/O
• Influence of IPC/RPC on kernel structure
Recall: Using IPC to Simplify OS

- What if the file system is not local to the machine, but on the network?
- Is there a general mechanism for providing services to other processes?
  - Do the protocols we run on top of IPC generalize as well?
Microkernels

• Split OS into separate processes
  • Example: File System, Network Driver are processes outside of the kernel
• Pass messages among these components (e.g., via RPC) instead of system calls
Microkernels

• Microkernel itself provides only essential services
  • Communication
  • Address space management
  • Thread scheduling
  • Almost-direct access to hardware devices (for driver processes)
Why Microkernels?

**Pros**
- Failure Isolation
- Easier to update/replace parts
- Easier to distribute – build one OS that encompasses multiple machines

**Cons**
- More communication overhead and context switching
- Harder to implement?
Flashback: What is an OS?

• Always:
  • Memory Management
  • I/O Management
  • CPU Scheduling
  • Communications
  • Multitasking/multiprogramming

• Maybe:
  • File System?
  • Multimedia Support?
  • User Interface?
  • Web Browser?

(Not provided in a strict microkernel)
Influence of Microkernels

• Many operating systems provide some services externally, similar to a microkernel
  • OS X and Linux: Windowing (graphics and UI)

• Some currently monolithic OSes started as microkernels
  • Windows family originally had microkernel design
  • OS X: Hybrid of Mach microkernel and FreeBSD monolithic kernel
Conclusion

• We studied the structure of the kernel
  • Kernel thread backing every user thread

• We saw how the kernel organizes a process’ memory
  • Kernel memory mapped into each process’ virtual address space

• We saw how the kernel supports operations on processes
  • fork, wait, exec, open file descriptions...

• We saw how the kernel handles I/O
  • Device drivers

• We saw how IPC influences the structure of the kernel
  • Service provide by other processes
Bonus Material (If Time)
Don’t fork() in a process that already has multiple threads

Unless you plan to call exec() in the child process
fork() in Multithreaded Processes

• The child process always has just a single thread
  • The thread in which fork() was called

• The other threads just vanish
fork() in a Multithreaded Processes

Not shown: Initially contains 0, 1, and 2 (stdin, stdout, stderr)

- Only the thread that called fork() exists in the new process
Possible Problems with Multithreaded `fork()`

- When you call `fork()` in a multithreaded process, the other threads (the ones that didn’t call `fork()`) just vanish
  - What if one of these threads was holding a lock?
  - What if one of these threads was in the middle of modifying a data structure?
  - No cleanup happens!

- It’s safe if you call `exec()` in the child
  - Replacing the entire address space
Don’t carelessly mix low-level and high-level file I/O
Avoid Mixing FILE* and File Descriptors

char x[10];
char y[10];
FILE* f = fopen(“foo.txt”, “rb”);
int fd = fileno(f);
fwrite(x, 10, 1, f); // read 10 bytes from f
read(fd, y, 10); // assumes that this returns 10

• Which bytes from the file are read into y?
  A. Bytes 0 to 9
  B. Bytes 10 to 19
  C. None of these?
Avoid Mixing FILE* and File Descriptors

```c
char x[10];
char y[10];
FILE* f = fopen("foo.txt", "rb");
int fd = fileno(f);
fwrite(x, 10, 1, f); // read 10 bytes from f
read(fd, y, 10); // assumes that this returns 10
```

• Which bytes from the file are read into y?
  A. Bytes 0 to 9
  B. Bytes 10 to 19
  C. None of these?
Be careful with `fork()` and `FILE*`
Be Careful Using `fork()` with FILE*

```c
FILE* f = fopen("foo.txt", "w");
fwrite("a", 1, 1, f);
fork();
fclose(f);
```

After all processes exit, what is in `foo.txt`?

Could be either `a` or `aa`

• Usually `aa` based on what I’ve observed in Linux...

• Depends on whether this `fwrite` call flushes...
Be Careful Using `fork()` with `FILE*`

User Space

Kernel Space

Thread’s Regs

FILE* Buffer

File Descriptors

3

Process 1

- Open File Description is aliased
- But the `FILE*` buffer is copied!

Open File Description

File: foo.txt
Position: 0

Process 2

Thread’s Regs

FILE* Buffer

File Descriptors

3

Not shown: Initially contains 0, 1, and 2 (stdin, stdout, stderr)