The microkernel OS Escape

Nils Asmussen

FOSDEM’14

Outline

Outline

1. Introduction
2. Tasks
3. Memory
4. VFS
5. Demo

Motivation

Beginning

- Writing an OS alone? That's way too much work!
- Port of UNIX32V to ECO32 during my studies
- Started with Escape in October 2008

Goals

- Learn about operating systems and related topics
- Experiment: What works well and what doesn't?
- What problems occur and how can they be solved?

Overview

Basic Properties

- UNIX-like microkernel OS
- Open source, available on `github.com/Nils-TUD/Escape`
- The kernel and the GUI are written in `C++`, the rest in `C`
- Runs on x86, ECO32 and MMIX
- Besides `libgcc` and `libsupc++`, no third party components

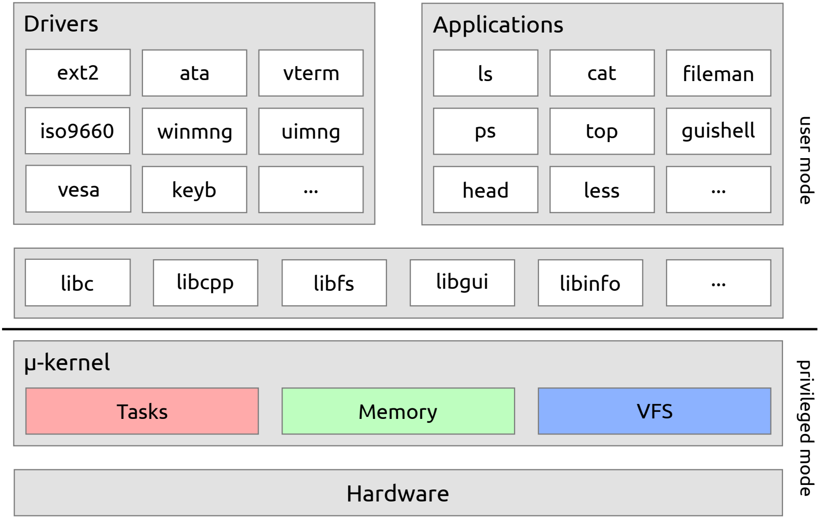
ECO32

MIPS-like, 32-bit big-endian RISC architecture, developed by Prof. Geisse for lectures and research

MMIX

64-bit big-endian RISC architecture of Donald Knuth as a successor for MIX (the abstract machine from TAOCP)

Overview



Drivers

ext2 ata vterm

iso9660 winmng uimng

vesa keyb ...

Applications

ls cat fileman

ps top guishell

head less ...

user mode

libc libcpp libfs libgui libinfo ...

µ-kernel

Tasks Memory VFS

Hardware

privileged mode

Outline

1. Introduction
2. **Tasks**
3. Memory
4. VFS
5. Demo

Processes and Threads

Process

- Virtual address space
- File-descriptors
- Mountspace
- Threads (at least one)
- ...

Thread

- User- and kernelstack
- State (running, ready, blocked, ...)
- Scheduled by a round-robin scheduler with priorities
- Signals
- ...

Processes and Threads

Synchronization

- Process-local semaphores
- Global semaphores, named by a path to a file
- Userspace builds other synchronization primitives on top
    - "User-semaphores" as a combination of atomic operations and process-local semaphores
    - Readers-writer-lock
    - ...

Priority Management

- Kernel adjusts thread priorites dynamically based on compute-intensity
- High CPU usage → downgrade, low CPU usage → upgrade

Outline

1. Introduction
2. Tasks
3. Memory
4. VFS
5. Demo

Memory Management

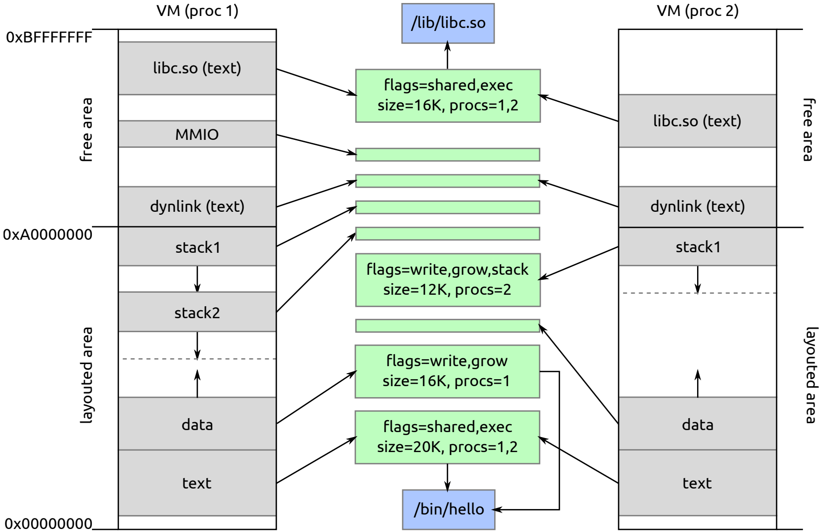
Physical Memory

- Most of the memory is managed by a stack for fast alloc/free of single frames
- A small part handled by a bitmap for contiguous phys. memory

Virtual Memory

- Upper part is for the kernel and shared among all processes
- Lower part is managed by a region-based concept
- `mmap`-like interface for the userspace

Virtual Memory Management



Outline

1. Introduction
2. Tasks
3. Memory
4. VFS
5. Demo

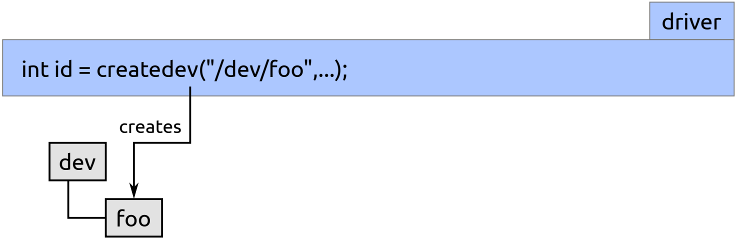
Basics

- The kernel provides the virtual file system
- System-calls: `open`, `read`, `mkdir`, `mount`, ...
- It's used for:
    1. Provide information about the state of the system
    2. Unique names for synchronization and shared memory
    3. Access userspace filesystems
    4. Access devices

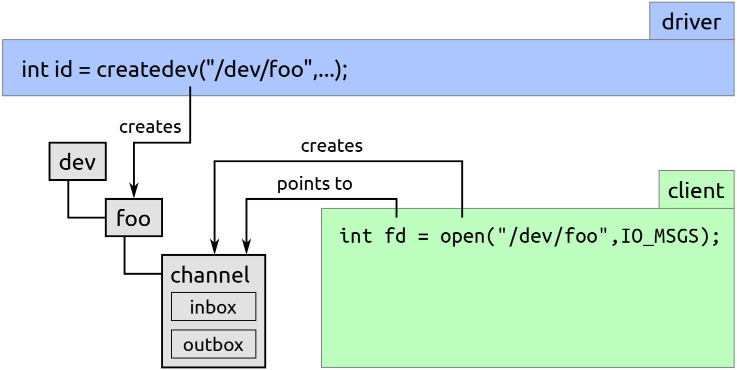
Drivers and Devices

- Drivers are ordinary user-programs
- They create devices via the system-call `createdev`
- These are usually put into /dev
- Devices can also be used to implement on-demand-generated files (such as /system/fs/$fs)
- The communication with devices works via asynchronous message passing

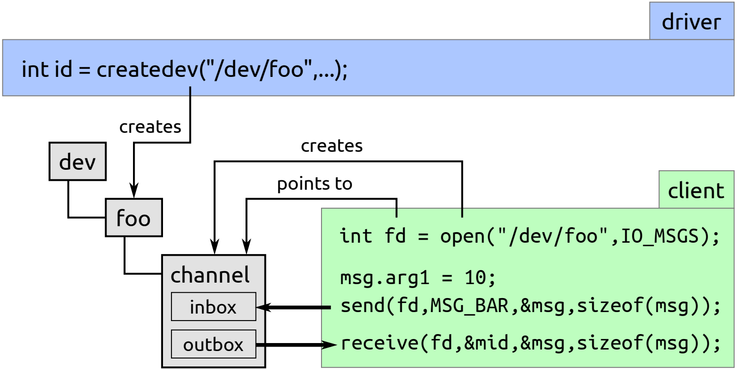
IPC between Client and Driver



IPC between Client and Driver



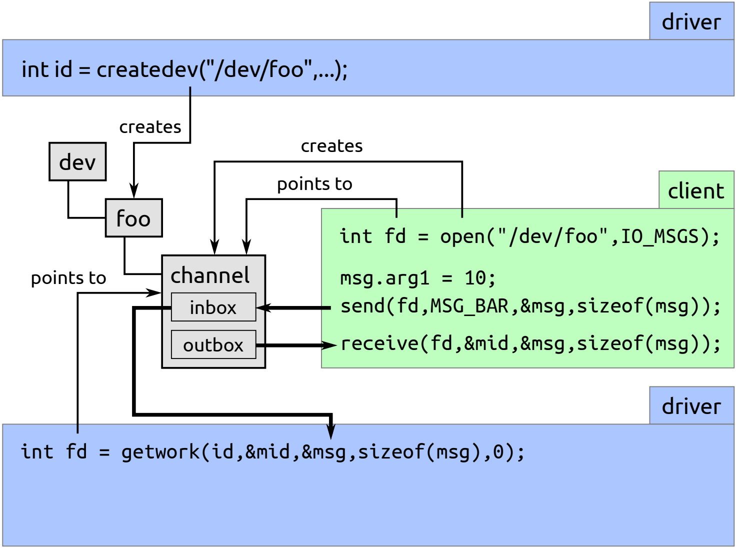
IPC between Client and Driver



```
int id = createdev("/dev/foo",...);
```

```
int fd = open("/dev/foo",IO_MSGS);
msg.arg1 = 10;
send(fd,MSG_BAR,&msg,sizeof(msg));
receive(fd,&mid,&msg,sizeof(msg));
```

IPC between Client and Driver



driver

```
int id = createdev("/dev/foo",...);
```

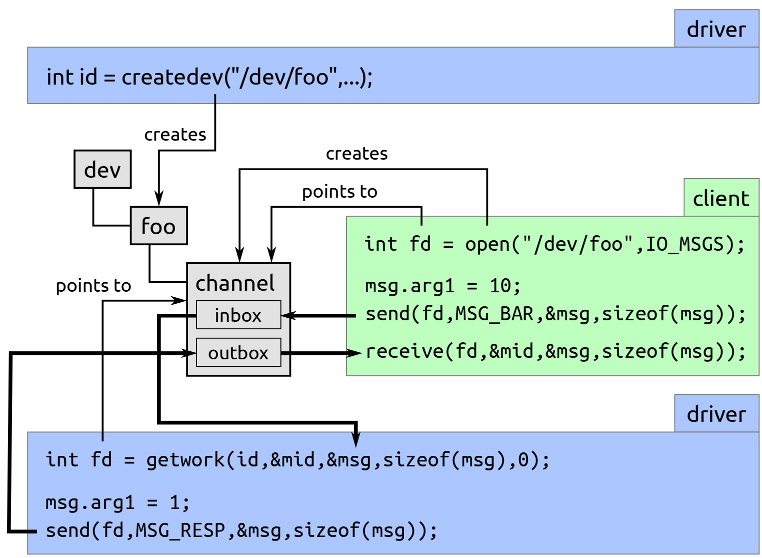
client

```
int fd = open("/dev/foo",IO_MSGS);
msg.arg1 = 10;
send(fd,MSG_BAR,&msg,sizeof(msg));
receive(fd,&mid,&msg,sizeof(msg));
```

driver

```
int fd = getwork(id,&mid,&msg,sizeof(msg),0);
```

IPC between Client and Driver



```
int id = createdev("/dev/foo",...);
```

```
int fd = open("/dev/foo",IO_MSGS);
msg.arg1 = 10;
send(fd,MSG_BAR,&msg,sizeof(msg));
receive(fd,&mid,&msg,sizeof(msg));
```

```
int fd = getwork(id,&mid,&msg,sizeof(msg),0);
msg.arg1 = 1;
send(fd,MSG_RESP,&msg,sizeof(msg));
```

Integrating devices into the read-write-pattern

- As in UNIX: Devices should be accessable like files
- Messages: `DEV_OPEN`, `DEV_READ`, `DEV_WRITE`, `DEV_CLOSE`
- Devices may support a subset of these message
- If using `open`/`read`/`write`/`close`, the kernel handles the communication
- Transparent for apps whether it is a virtual file, file in userspace fs or device

Achieving higher throughput

- Copying everything twice hurts for large amounts of data
- `sharebuf` establishes shmem between client and driver
- Easy to use: just call `sharebuf` once and use this as the buffer
- Clients don't need to care whether a driver supports it or not
- Drivers need just react on a specific message, do an `mmap` and check in `read`/`write` whether the shared memory should be used

Mounting

Concept

- Every process has a mountspace, that is inherited to childs
- `clonems` gives your process its own copy
- Mountspace is a list of (*path*,*fs-con*) pairs
- Kernel translates fs-system-calls into messages to *fs-con*

Mounting

Concept

- Every process has a mountspace, that is inherited to childs
- `clonems` gives your process its own copy
- Mountspace is a list of (*path*,*fs-con*) pairs
- Kernel translates fs-system-calls into messages to *fs-con*

Example

```
// assuming that an ext2-instance has been started
// to create /dev/ext2-hda1
int fd = open("/dev/ext2-hda1", ...);
mount(fd, "/mnt/hda1");
```

Outline

1. Introduction
2. Tasks
3. Memory
4. VFS
5. Demo

The End

Get the code, ISO images, etc. on:
`https://github.com/Nils-TUD/Escape`

Thanks for your attention!