Operating Systems

Tutorial 1

Michael Tänzer

os-tut@nhng.de
http://os-tut.nhng.de

2nd November 2010

Outline

1. Review
2. System Structures
   - Single- vs. Multi-Programming
   - Monolithic vs. Microkernel
3. System Calls: The User/Kernel Boundary
   - Kernel Invocation
   - Kernel and User Space Isolation
4. I/O Techniques
   - Polling I/O
   - DMA

True or False

- If you are the leader of a supermarket minimizing personnel costs is a policy.
- `(0xFF - 1) == (0xFF & (~1))`
- `trap` is a privileged instruction

True or False

- If you are the leader of a supermarket minimizing personnel costs is a policy.
- `(0xFF - 1) == (0xFF & (~1))`
- `trap` is a privileged instruction

True or False

- If you are the leader of a supermarket minimizing personnel costs is a policy.
- `(0xFF - 1) == (0xFF & (~1))`
- `trap` is a privileged instruction

True or False

- If you are the leader of a supermarket minimizing personnel costs is a policy.
- `(0xFF - 1) == (0xFF & (~1))`
- `trap` is a privileged instruction

Single- vs. Multi-Programming Systems

What is the advantage of the latter?

Single- vs. Multi-Programming Systems

What is the advantage of the latter?

Single-Programming

- Only one application may run at a time

Multi-Programming

- Multiple applications may run simultaneously
- OS has to save and restore the processing state
- The CPU can do something useful while another programme waits for I/O
- Only makes sense when doing interrupt driven I/O

Single- vs. Multi-Programming Systems

What is the advantage of the latter?

Single-Programming

- Only one application may run at a time

Multi-Programming

- Multiple applications may run simultaneously
- OS has to save and restore the processing state
- The CPU can do something useful while another programme waits for I/O
- Only makes sense when doing interrupt driven I/O

- Multi-Programming $\neq$ Multi-Tasking
- But Multi-Tasking $\Rightarrow$ Multi-Programming

Systems based on a monolithic vs. on a μ-kernel

Strengths and weaknesses

Systems based on a monolithic vs. on a μ-kernel

Strengths and weaknesses

Monolithic Kernel

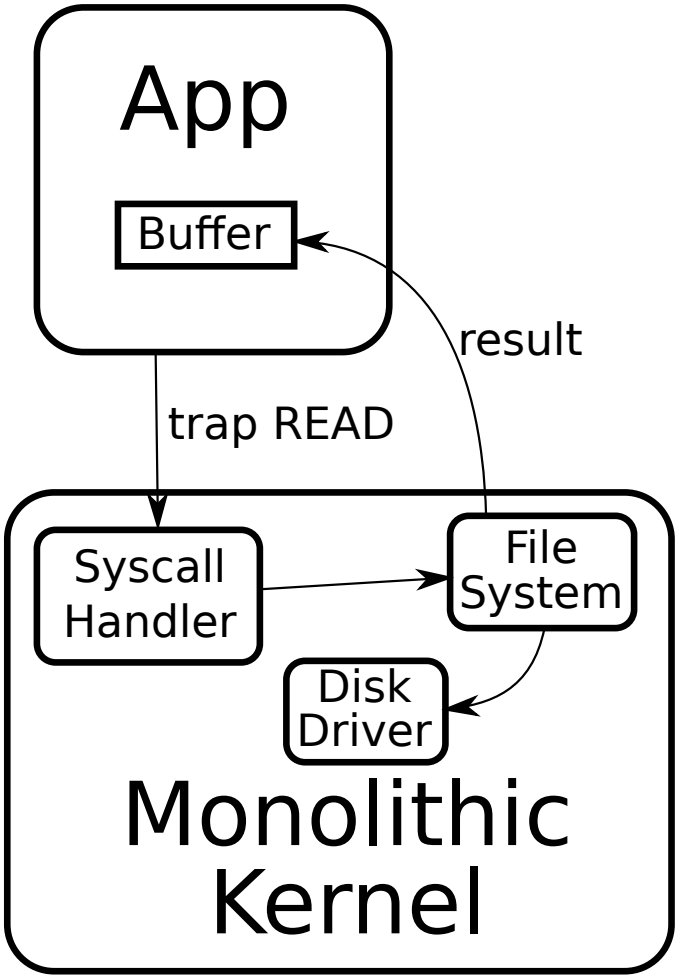
- One large binary
- \+ Easy and fast service invocation through function calls
- – Complex interdependencies $\Rightarrow$ difficult to extend
- – No isolation $\Rightarrow$ bug in one component affects entire kernel

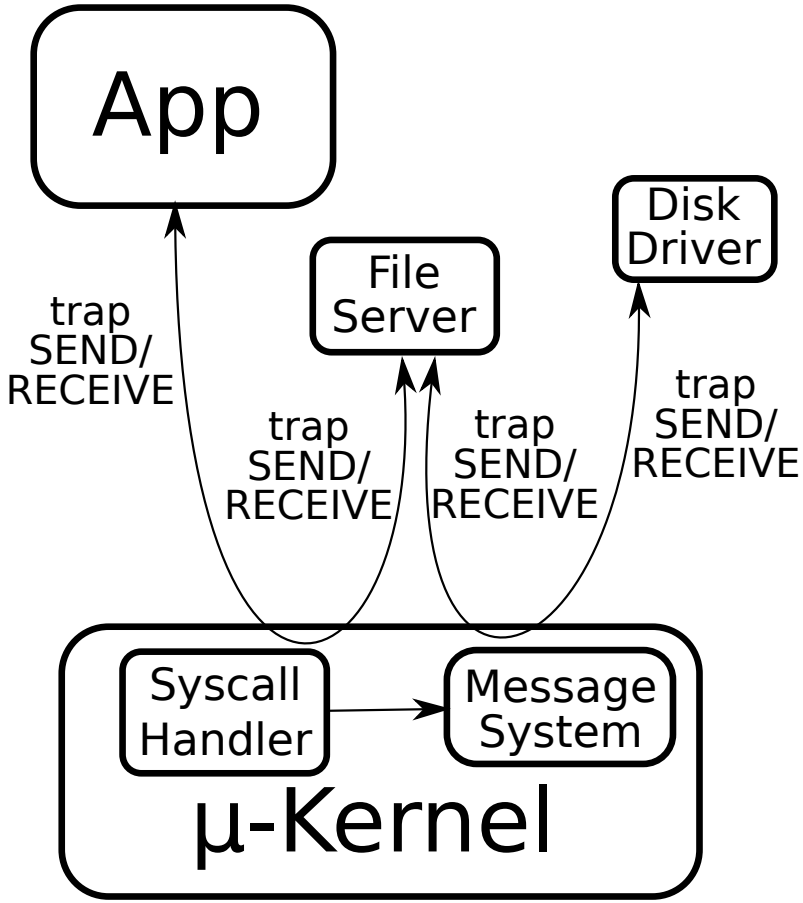
Microkernel

- Small kernel is host for servers running at user level
- \+ Each server offers a well-defined API $\Rightarrow$ better structure
- \+ Malfunctions in one component can't affect others
- – Higher communication overhead

Overhead of an implementation of `read` on a monlithic opposed to a μ-kernel

Overhead of an implementation of `read` on a monlithic opposed to a μ-kernel





Which events can lead to invocation of the kernel?

Which events can lead to invocation of the kernel?

Exceptions Generated by the CPU to notify the OS of error conditions (e. g. division by zero, page fault). Side effects of executed instructions

Interrupts Generated outside the processor, to notify the OS of events (e. g. a hardware device has finished retrieving/storing data). Used to replace polling and improve throughput

System Calls a user process needs some service provided by the OS

How is the `trap` instruction related to system calls?

How is the `trap` instruction related to system calls?

- `trap` leads to a ‘software interrupt’ which causes the kernel to run
- It is used to implement system calls
- In principle you could also implement syscalls using exceptions

Syscalls enable the transition from user to kernel level

Why do we have to be very careful when designing this transition?

Syscalls enable the transition from user to kernel level

Why do we have to be very careful when designing this transition?

- The parameters of the system call are provided by user-level software
- Many applications are buggy or even malicious

$\Rightarrow$ The kernel has to validate the parameters very carefully

- E. g. the kernel has to check whether the address of a buffer provided by the application points to the applications address space, otherwise it could overwrite or disclose kernel data structures

Does using a kernel stack instead of the application's stack during syscalls improve protection?

Does using a kernel stack instead of the application's stack during syscalls improve protection?

- Kernel-internal information could still be in the memory that once was the stack when the syscall returns (only popped not overwritten)
- In multi-processor environments another thread could even modify the data on the stack while the kernel is using it
- The application's stack might not provide enough space (e. g. too close to a page boundary that hasn't been mapped yet). On some architectures this leads to a system halt/crash

What is the responsibility of the CPU for device-to-memory data transfers in each of the three primary I/O models?

What is the responsibility of the CPU for device-to-memory data transfers in each of the three primary I/O models?

Programmed I/O (Polling)

Coordinate the entire data transfer

1. Issue an I/O command to the device
2. Check if the device is ready, if it is not repeat this step
3. Fetch the data directly from the device registers

What is the responsibility of the CPU for device-to-memory data transfers in each of the three primary I/O models?

Interrupt-Driven I/O

1. Initiate the I/O transaction
2. Do something else while waiting for the interrupt from the device
3. When the interrupt occurs fetch the data directly from the device registers

What is the responsibility of the CPU for device-to-memory data transfers in each of the three primary I/O models?

DMA

1. Initiate the transaction and provide a location in physical memory to the DMA controller (may be integrated in the device)
2. Do something else while waiting for the interrupt from the DMA controller
3. When the interrupt occurs all data is already in memory for further processing

Calculate the CPU overhead with polling for the following devices

Specs

- CPU frequency: 400 MHz
- One polling operation costs 400 cycles
- Hard disk: 32 bit per poll at 8 MB/s

What is an interrupt vector and an interrupt service routine (ISR)?

What is an interrupt vector and an interrupt service routine (ISR)?

Interrupt Vector The entry address of an interrupt handler (MIPS and many Microcontrollers) or the index into an array of such addresses (interrupt vector table, found e. g. in x86 systems)

ISR System routine that handles an interrupt (and syscalls)

Write pseudo-code which expresses the functionality of the DMA engine for device-to-memory data transfer

The destination address `baseAddr` and `blockSize` are given as parameter

Write pseudo-code which expresses the functionality of the DMA engine for device-to-memory data transfer

The destination address `baseAddr` and `blockSize` are given as parameter

```
// Device has already been configured when
// setting up DMA transfer
for count = 0 to blockSize - 1 do
    data = deviceRead(count);
    memoryWrite(baseAddr + count, data);
od
generateInterrupt();
```

Why is it preferable to use a dedicated I/O bus, thus separating devices from the CPU and memory bus?

Why is it preferable to use a dedicated I/O bus, thus separating devices from the CPU and memory bus?

- The DMA engine has to access the bus twice for each loop iteration
- When the DMA controller accesses the bus the CPU has to wait for it to become available again to fetch data or instructions from memory
- When separating the I/O bus from the memory bus the DMA controller only needs one access to the memory bus per iteration

$\Rightarrow$ Less traffic on the memory bus, less wait cycles for the CPU

Questions & Comments

Any questions or comments?

The End

A Unix saleslady, Lenore,
Enjoys work, but she likes the beach more.
She found a good way
To combine work and play:
She sells C shells by the seashore.

A very intelligent turtle
Found programming Unix a hurdle
The system, you see,
Ran as slow as did he,
And that's not saying much for the turtle.