Operating Systems

Tutorial 1

Michael Tänzer

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

26th October 2010

Outline I

About Me

Me, Myself and I

- Student of Computer Science (Diploma)
- 9^{th} semester
- Interests:
  - Operating systems
  - Cryptography and security
  - Telematics

Distro Wars

Now the fun part

- Your given name
- The Operating System and Distribution/Version you use most of the time
- For how long you've been using it

Differences between user and kernel mode

Why are they needed?

User Mode

- Only unprivileged instructions may be executed
- Otherwise Exception is thrown

Kernel Mode

- All instructions may be executed
- Entered if an interrupt, trap or exception occurs

Needed to protect applications from each other and make sure everyone gets their fair share

Typical examples for privileged instructions

Why are they privileged?

- Manipulate control registers for memory address translation
- Disable or enable interrupts
- Access privileged plattform devices

If they weren't user software could

- manipulate critical system state
- elevate its privileges
- overcome protection
- ruin isolation
- impact stability
- impact trustworthiness
- ...

Bit Operators

C	Description
&	Bitwise $\wedge$
|	Bitwise $\vee$
~	Bitwise $\neg$
<<	Left shift
>>	Right shift (filled with zeros)
^	Bitwise XOR

Why might it be necessary to set or clear a single bit of an integer value?

- Most hardware has control registers where each bit has a special meaning
- If one only needs to store one bit, why use more space? Bit maps, bit fields, etc.

Exercises

- How can you set the `i`th bit of a given integer `value`?

```
result = value | (1 << i);
```

- How can you clear the `i`th bit of a given integer `value`?

```
result = value & ~(1 << i);
```

- How can you retrieve the contents of bit 2 to bit 5 of a given integer `value`?

```
result = value & (0xf << 2);
```

Visualize the stack before, during and after the execution of `foo()`

```
double foo(int *p){
    int x; double y;
    x = *p;
    /* do something useful */
    return y;
}

double bar(){
    double d; int i = 42;
    d = foo(&i);
    return d;
}
```

Some claim parameter passing via registers or non-stack memory is either too special or too difficult I

Registers

- – Only a few available
- – Caller and callee have to agree on a usage convention
- \+ Fast

Heap

- Dynamically allocate memory for the parameters an pass the address to the function
- \+ “Unlimited”
- \+ Reentrant
- – Slow

Some claim parameter passing via registers or non-stack memory is either too special or too difficult II

Code Segment

- Arguments are stored in the caller's code right before or after the `call` instruction or the code of the callee
- \+ "Unlimited"
- – Arguments must be skipped at runtime ⇒ additional `jump`
- – Non-reentrant code
- – If code segment is only executable you can't write to it

Fixed Memory Location

- \+ "Unlimited"
- – Non-reentrant code

Function vs. Macro

- Code from macros is duplicated, code for functions only exists once (exception: inlining)

$\Rightarrow$ The compiled binary is usually larger but more efficient (due to less overhead for the function calls at runtime)

- Macros are sensitive to side effects
- Macros are untyped (they don't receive parameters of a well-defined type)

Policy vs. Mechanism

- Policies specify how a certain task should be fulfilled, optimised for certain goals.
- Mechanisms are used to realise those policies
- Example:
  - Task: Get from the Mensa to the ‘Info-Bau’
  - Goals: Short, fast
  - Mechanisms: Turn left/right, slow down, etc.
  - Policies: Minimum deviation from linear distance, minimal number of obstacles

Example II

- Assume you are the leader of a supermarket
- Task: How many cash points should be occupied?
- Goal: Minimal personnel costs, minimal waiting time for the customers
- Mechanisms:
  - Bell to indicate another cash point should be occupied
  - Icon on the cash display to indicate this cash point should be closed
  - Light barriers
  - Buzzers
- Policies: Have a minimum and maximum queue length, let the customers hit a buzzer if they wait too long

Enumerate the major tasks of an operating system

- Abstraction/Standardisation
- Resource Management
- Security/Protection
- Providing an execution environment for applications

What is a Shell?

The shell is a CLI (Command Line Interface) for the Operating System

- In Linux the shell runs as a normal user programme
- Usually a command starts another programme which does the work
- The shell provides mechanisms to combine commands (pipes, control flow statements)
- And manipulate (environment) variables
- Allows manipulation of multiple files through wild cards
- Sophisticated shells (e. g. `bash`) also have convenience features like tab completion for almost everything

Basic Commands

`cd`	Change the working directory
`ls`	List the contents of a directory (like `dir` in Windows)
`mkdir`	Make a new directory
`rmdir`	Remove directory (only works if the directory is empty)
`cp`	Copy file or directory
`mv`	Move/Rename file or directory
`rm`	Delete a file
`cat`	Concatenate inputs to output
`less`	Scroll through input
`grep`	Print the line containing a specified pattern in the input
`xargs`	Execute a command with the input as parameters

How can you get help for a programme?

No, ‘Use Google’ is not the answer I’ve hoped for

- The command `man` shows the (hopefully complete and understandable) manual
- Sometimes more than one programme has the same name (e. g. there is the command, system call and function `exit`) then you also have to specify which section you want to see
- `man man` shows how to use `man` and which sections are available

How can you search for a file?

- Assume you want to edit a file named `syscall.c`, but you have forgotten in which subdirectory of our project it resides. What can you do?
- ```
  $ find ./ -name 'syscall.c'
  ./kern/arch/mips/mips/syscall.c
  $
  ```
  ```

Combine commands to get complex things done

- Assume you have a directory that contains multiple C source files in multiple subdirectories. How can you search for all occurrences of the macro `FOOBAR` in these C files?
- `$ find ./ -name '*.c'| xargs grep 'FOOBAR'`
  `./foo/bar.c:#define FOOBAR 42`
- `$ grep -r --include='*.c' 'FOOBAR'./`
  `./foo/bar.c:#define FOOBAR 42`

Creating Binaries

- What steps are necessary to create an executable program from multiple C source files?
  1. Compilation – that's `gcc`'s job, each file is compiled on its own ⇒ multiple object files (*.o)
     - Object files contain machine code and not yet resolved symbols which reference to 'things' in other object files
  2. Linkage – that's what `ld` does, it resolves the references and puts everything together ⇒ one binary
- Usually `gcc` will also call `ld` but you could give the `-c` option
- Manual linking is a little nightmare

What is `make`?

- `make` is an automated build tool (an ancestor of `ant`)
- The build process is separated into steps called targets
- Targets can depend on other targets
- Reduces overhead:
  - Targets will only run once per invocation of `make`
  - Sources will only be compiled if they were modified since the last build
- Executes `gcc` and other tools to do the dirty work
- If something doesn't work try a `make clean` followed by `make` to get a clean build

Example Makefile

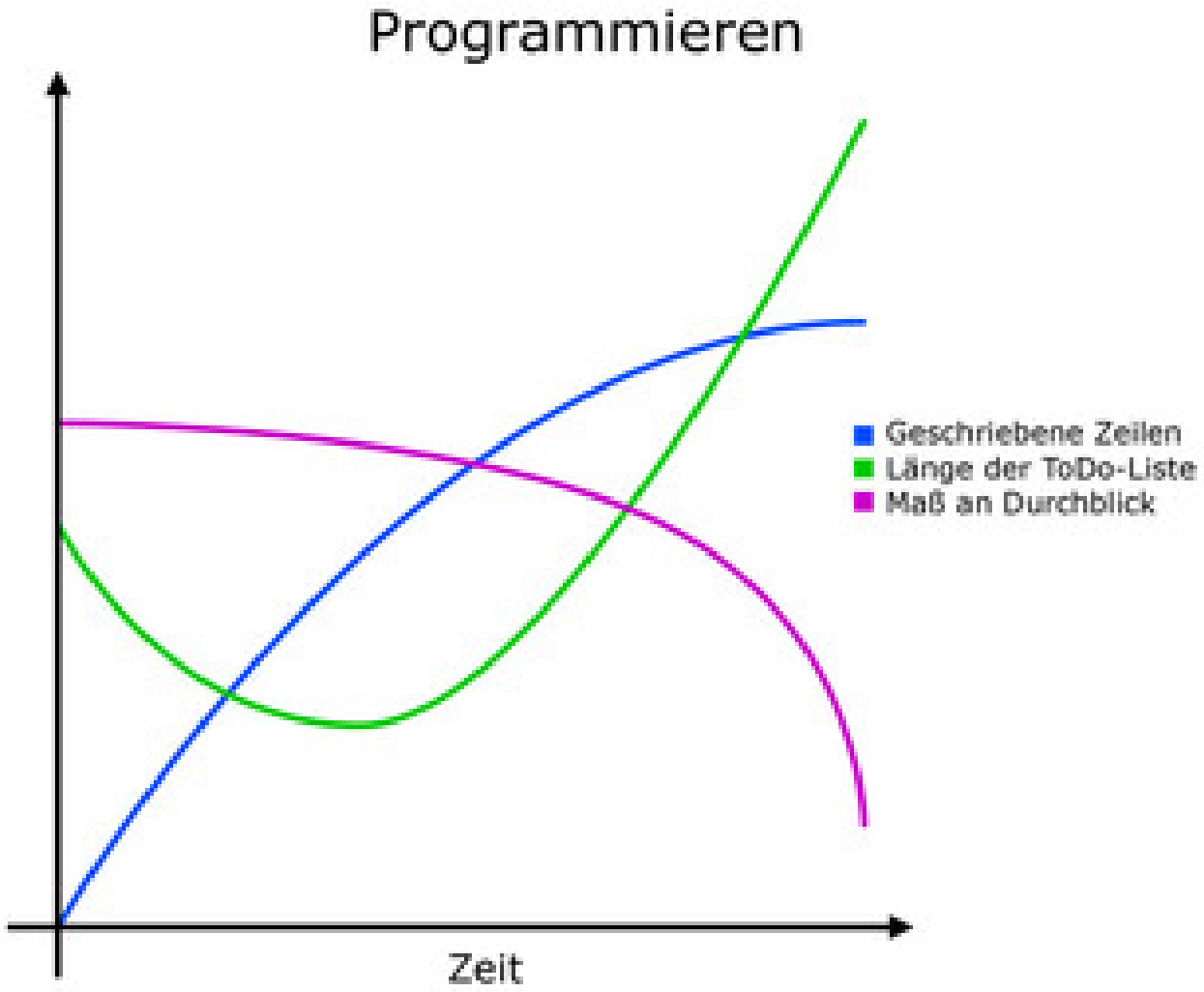
```
coffee: water powder filter
        mv machine/water machine/filter/ && \
        mv machine/filter/water machine/coffee
water:
        touch machine/water

powder: filter
        touch machine/filter/powder
filter:
        mkdir machine/filter
clean:
        rm -r machine/*
```

Questions & Comments

Any questions or comments?

The End


Programmieren
Geschriebene Zeilen
Länge der ToDo-Liste
Maß an Durchblick
Zeit