Karlsruher Institut für Technologie

1. Operating Systems Tutorial

Formalities, OS and C Basics, ABIs, Linking

Péter Bohner | 2023-11-02

Inhaltsverzeichnis

Introduction

About Me

- Péter Bohner (peter.bohner@student.kit.edu)
- CS Bachelor, 5. Semester

Introduce yourselves

- Name?
- Field of Study?
- Am I your regular tutor?
- C/Systems programming/operating systems knowledge?
- Experience with Linux?
- Expectations for the tutorial?

Links

- Everything is in ILIAS
- My website `https://bohner.me/teaching/ws2023_ostut/`
- ATIS: `https://www.atis.informatik.kit.edu/513.php`
- Compiler Explorer: `https://godbolt.org/`
- Feedback: `https://forms.gle/9CwJSKidKibubran9`

Where should I ask my questions?

1. Ask in the tutorial
2. ILIAS-Forum
3. Tutor (email: `peter.bohner@student.kit.edu`, discord: `xzvf`)
4. Mail to Peter Maucher
5. Mail to prof

Lecture

- Exam: Three exercises (20P each) with mixed theory and practice (programming) Exact modalities unknown **different from past exams!**
- No separate exam for repeat takers, probably
- Exercises optional, but highly recommended
  - Complete short introductory tasks!
  - Tackle larger tasks if interested
- The contents of the exercises are part of the exam, not just the lecture!

Completing the exercises

- Submissions must compile and run on **Linux x86_64**
- Exercises with x86 assembly and linux syscalls

your host	recommended development environment
* x86_32	ATIS
Linux x86_64	your host :-)
*BSD x86_64	Linux VM/dual boot
Windows x86_64	WSL2 or Linux VM/dual boot
Windows ARM	ATIS
MacOS ARM	ATIS, Asahi Linux
MacOS x64	Linux VM/dual boot
Sonstiges	ATIS

Tasks and responsibilities of operating systems

Abstraction/Standardization

- Devices and how to talk to them differ greatly
  - Remember the Driver-CDs‘‘ shipped with motherboards back in the days so you could properly configure and run things?
- Strange user demands: Programs should run on more than one hardware configuration
  - Abstract away hardware details!

Tasks and responsibilities of operating systems

Resource management

- You want to print? Too bad, another program is already using that printer.
- You want to access the storage drive? Too bad, another program is already doing that.
- You want to get CPU time? Too bad, this while (true) loop is more important.

Tasks and responsibilities of operating systems

Security and Protection

- You are a good citizen and use a password manager. What could happen without your helpful OS? Other programs may read its memory!
- You copied a password to your clipboard? Ooops. You're on your own there. The Clipboard is not provided by the OS and mostly has no special protections.
- You write a cool little program that fills a buffer with a random value. Sadly you made a mistake and missed a bounds check. What happens? You crash, but your text editor doesn't suddenly have its memory overwritten!

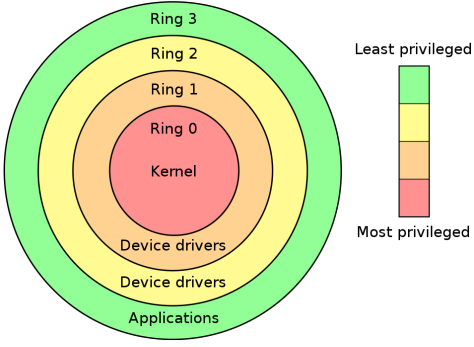
Tasks and responsibilities of operating systems

Provide an execution environment for applications

- What does that mean? Basically all of the above combined and more. Make a homely place where applications like to live!

Kernel / User mode differences

- What are the differences between a processor running in kernel or user mode?
- Why are both modes needed?



- In kernel mode you have full access to privileged instructions.

What instructions are privileged?

- Change control registers for memory mappings
  - Read other process's memory
- Disable / Enable interrupts
  - No preemption for you.
- Access platform devices (network card, storage, printer,...)
- Some nice registers: LGDT (Load Global Descriptor Table) or the LLDT, INVD (Invalidate cache), HLT (Halt processor!)
- Is **MOV** (Move) a priviledged instruction?
  - Yes, if moving to debug/control registers or privileged memory locations!

The C Programming Language

History

- Developed by Dennis Ritchie at Bell Labs in 1972
- successor to B
- one of the most used (systems-) programming languages

Properties

- imperative, procedural, not OO
- low level
- manual memory management
  - fun with pointers
  - CVE-Factory

Datatypes - Primitives

name	minimum size (bytes)	x86_64 Linux
char	1	1
short	2	2
int	2	4
long	4	8
long long	8	8
float	-	4
double	-	8
long double	-	16

- Modifier: `unsigned signed`, all types are signed by default
- except `char`, that is architecture dependent . . .
- except `char` which is implementation dependant

Datatypes - fixed size

```
#include<inttypes.h> // or #include<stdint.h>
int8_t a; uint16_t b;
int_least16_t c; int_fast16_t d;
intptr_t p; ssize_t ss; size_t s;
```

Datatypes - boolean

- does not really exist (C99 `_Bool` type)
- use 0 / 1 (0 = false, nonzero true)
- or `stdbool.h`

```
// #include<stdbool.h>
typedef unsigned char bool;
#define false 0
#define true 1
int main () {
    bool hey = true;
    return 0;
}
```

- Size of types/variables

```
int foo; printf("%d", sizeof(foo));
```

Datatypes - Arrays

```
int arr[5] = {1, 2, 3, 4, 5};
int arr2[] = {1, 2, 3, 4, 5};
int arr4[10] = {1,2};
int arr3[5];
```

- continuous memory area
- size is part of type, `sizeof(int[5])` works correctly
- How do you get the size of an array during runtime?
- That's the neat part: you don't :D
  - `sizeof(arr)` is `sizeof(int *)`
  - because arrays are just syntax sugar for pointers

Datatypes - Strings

```
char *string = "mystring";
char[] s2 = "another";
char[] s3 = {'H', 'i', '!', '\0'};
```

- strings are char arrays with NULL terminator at the end
- One byte longer than number of ASCII characters
- Uses ascii codes

Datatypes - Structs

```c
struct Person {
    int age;
    char *name;
};
typedef struct Person person_t;
int main(void) {
    struct Person peter = {.name = "Peter", .age = 20};
    person_t also_peter = peter;
    printf("%s is %d years old.\n", peter.name, peter.age);
}
```

- Product type
- Not a class: No hidden vtables/inheritance, no methods.

Locals and globals

- What is the difference between local and global variables?
- **globals**: Live while the program runs. Placed in data segments.
- **locals**: Are only accessible in scope of declaration. Live on stack or in registers

```
int uninitialized_global; // 0
int initialized_global = 0xCAFE; // 1
const int const_global = 0xBEEF; // 2
int func() {int local; /**/}
```

Where is each symbol stored?

1. .bss
2. .data
3. .rodata
4. stack (or register)

Pointers

```c
typedef struct {int age;char *name;} Person;
void foo(Person p) { p.age = 100; printf("%d ", p.age);}
void bar(Person *p) {p->age = 200; printf("%d ", p->age);}
int main(void) {
    Person p = {.name = "Me", .age = 20};
    printf("%d ", p.age);
    foo(p);
    printf("%d ", p.age);
    bar(&);
    printf("%d ", p.age);
}
```

output?

20 100 20 200 200

Pointers

- A variable that points to an arbitrary region of memory
- a `int *` points to an int, `void *` points to anything
- the element size of `void` is **undefined**
- Java references are like pointers, but without pointer arithmetic

Pointers

```c
void swap(int *a, int *b) {
    int tmp = *a; //dereference with *
    *a = *b; *b = tmp;
}
int main(void) {
    int x = 5, y = 10;
    printf("x = %d; y = %d\n", x, y);
    swap(&a, &b); //take address with &
    printf("x = %d; y = %d\n", x, y);
}
```

output?

```
x = 5; y = 10
x = 10; y = 5
```

Pointers

```c
typedef struct {char *name; int age; } Person;
void foo(Person *p) {
    printf("%s is %d years old\n", p->name; (*p).age); //-> is just syntax sugar
}
void bar(Person people[], size_t count) {
    for(size_t i = 0; i < count; i++) {
        printf("%s is %d years old\n", people[i].name, (people + i)->age); //-> as is []
    }
}
```

Operators applicable on int*

```
* & -> [] + -
```

Demo time

What is an ABI?

Application **B**inary **I**nterface

What does that specify?

- Interface of **binary** programs (i.e. after compilation)
- Instruction Set (e.g. x68, ARM)
- Calling convention (e.g. **cdecl** or **System V AMD64 ABI**)
- Basic data types and their size / alignment (int, sizeof(int))
- How to perform System Calls

ABI of a library

This tem is used when talking about binary compatibility of different library versions. -> Do you need to recompile your code against the new version?

Calling conventions

What's the usual calling convention on modern Linux?

System V AMD64 ABI

- Integer arguments in `rdi`, `rsi`, `rdx`, `rcx`, `r8`, `r9`, then stack
- FP arguments in `xmm0` to `xmm7`
- Integer return value in `rax`, `rdx`

`https://godbolt.org/z/68xexn`

Static vs Dynamic linking

What is the difference between static and dynamic linking?

- A static library is a collection of object files
  - The linked can treat it as normal code
- A dynamic library is loaded and linked at runtime

Pros and cons

1. static Linking
   - + Unused references can be elided
   - + Library calls just as fast as local ones
   - + No runtime overhead for loading and relocation
   - - Library can not be shared -> Memory overhead
2. dynamic linking
   - + Library (Code segment) can be shared

PIC

What does PIC stand for?

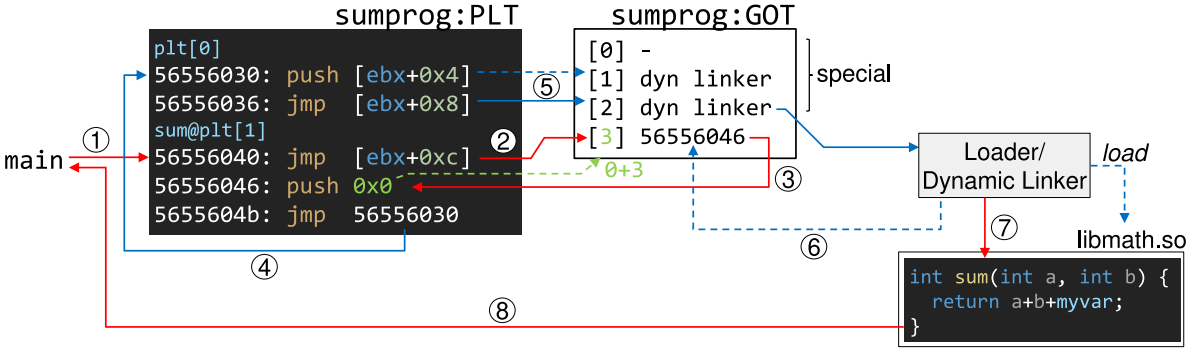
Position Independent Code

Independent to what?

It's position in the process' address space.

- Shared libraries are loaded somewhere in the address space of a process
- PIC uses relative addresses only -- and can therefore be loaded anywhere
- How do you find global symbols then?
  - GOT, the Global Offset Table

PIC


sumprog:PLT
plt[0]
56556030: push [ebx+0x4]
56556036: jmp [ebx+0x8]
sum@plt[1]
56556040: jmp [ebx+0xc]
56556046: push 0x0
5655604b: jmp 56556030
sumprog:GOT
[0] -
[1] dyn linker
[2] dyn linker
[3] 56556046
special
0+3
main
①
②
③
④
⑤
⑥
⑦
⑧
Loader/
Dynamic Linker
load
libmath.so
int sum(int a, int b) {
return a+b+myvar;
}

End

Are there any questions?

Leave anonymous feedback

https://forms.gle/9CwJSKidKibubran9

Bye!

Until next time, **Wed, 8.11.2023**!