TECHNISCHE UNIVERSITÄT DRESDEN

Fakultät Informatik Institut für Systemarchitektur, Professur für Betriebssysteme

OPERATING-SYSTEM CONSTRUCTION

Material based on slides by Olaf Spinczyk, Universität Osnabrück

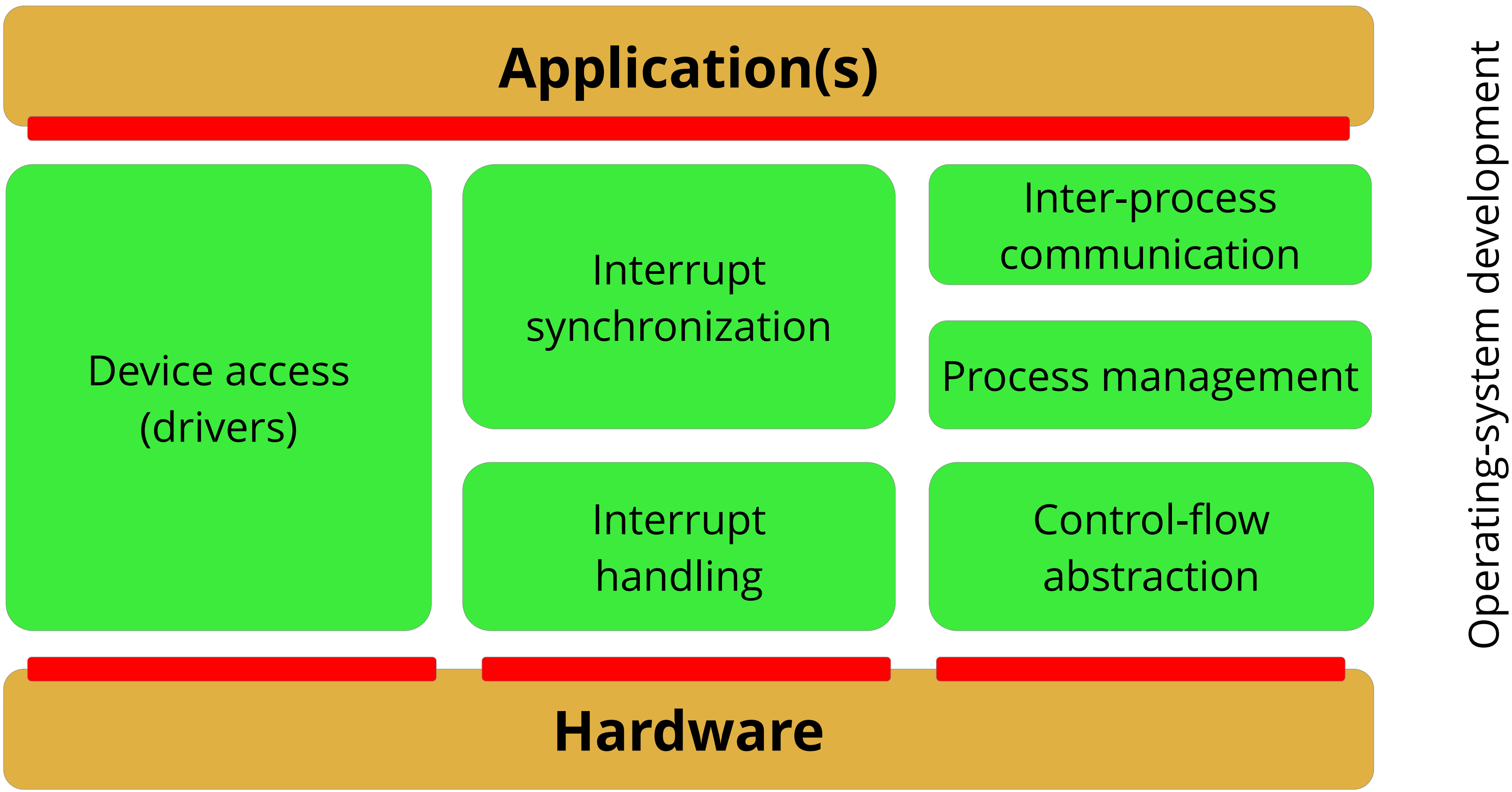
Coroutines and Threads

https://tud.de/inf/os/studium/vorlesungen/betriebssystembau

HORST SCHIRMEIER

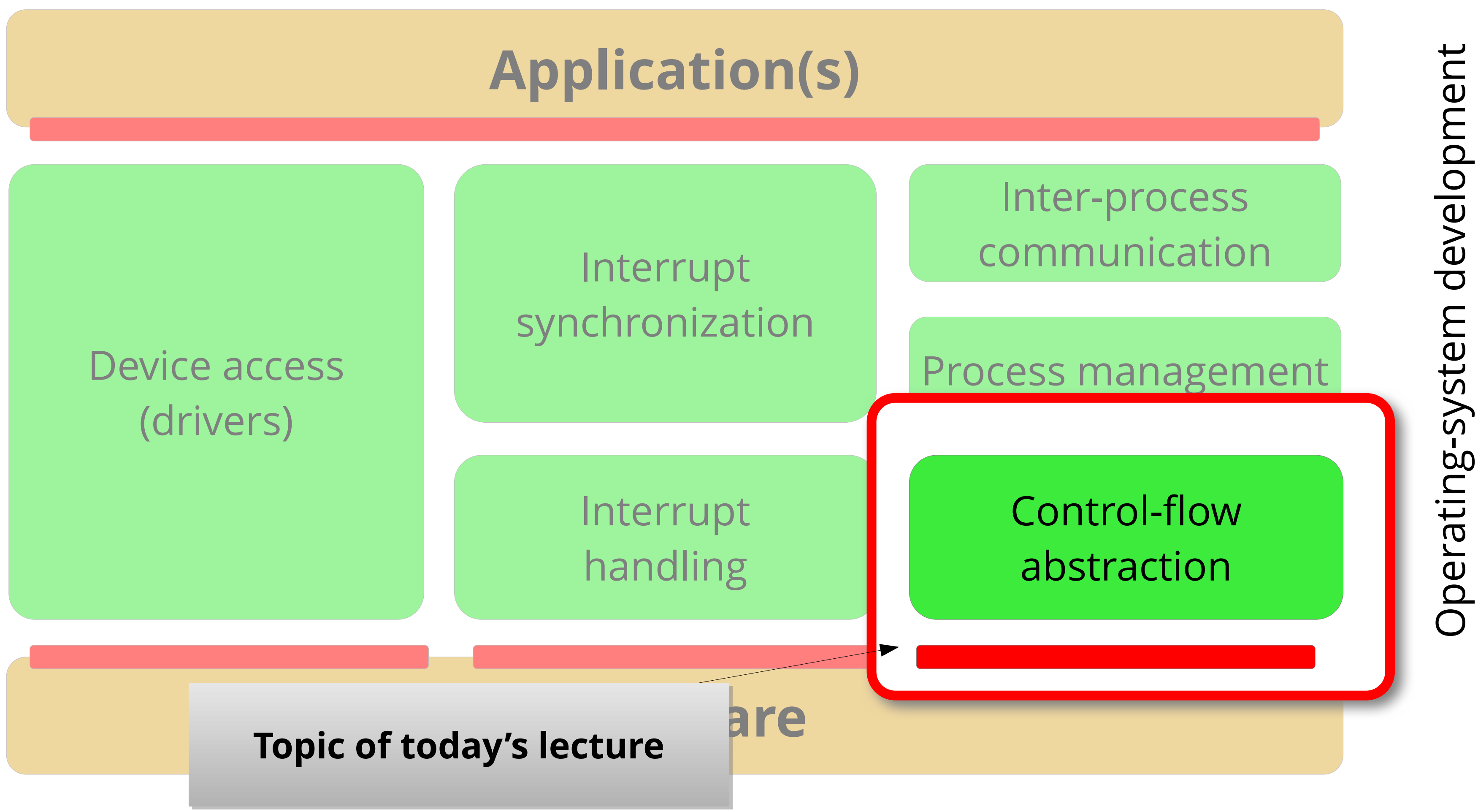
Overview: Lectures

Structure of the “OO-StuBS” operating system:



Overview: Lectures

Structure of the “OO-StuBS” operating system:



Agenda

- Motivation: Quasi Parallelism
  - Experiments
- Basic Terminology
  - Routine and Control Flow
  - Coroutine, Control Flow and Thread
  - Asymmetric and Symmetric Continuation Model
- Implementing Coroutines
  - Continuations
  - Elementary Operations
- Preview
  - Coroutines as a Basis for Multithreading
- Summary

Agenda

- **Motivation: Quasi Parallelism**
  - Experiments
- Basic Terminology
  - Routine and Control Flow
  - Coroutine, Control Flow and Thread
  - Asymmetric and Symmetric Continuation Model
- Implementing Coroutines
  - Continuations
  - Elementary Operations
- Preview
  - Coroutines as a Basis for Multithreading
- Summary

Motivation: Quasi Parallelism

- Given: Functions *f* and *g*
- Goal: *f* and *g* shall run in overlapping/alternating fashion

```
void f() {
    printf("f:1\n");❶


    printf("f:2\n");❸


    printf("f:3\n");❺

}
```

```
int main() {
    ?
}
```

```
void g() {
❷   printf("g:A\n");


❹   printf("g:B\n");


❻   printf("g:C\n");

}
```

Motivation: Quasi Parallelism – Experiment 1

```c
void f() {
    printf("f:1\n");


    printf("f:2\n");


    printf("f:3\n");

}
```

```c
void g() {
    printf("g:A\n");


    printf("g:B\n");


    printf("g:C\n");

}
```

```c
int main() {
  f();
  g();
}
```

```
$ gcc experiment1.c
$ ./a.out
f:1
f:2
f:3
g:A
g:B
g:C
```

Of course, it doesn't work this way ...

Motivation: Quasi Parallelism – Experiment 2

```
void f() {
    printf("f:1\n");
    g();

    printf("f:2\n");
    g();

    printf("f:3\n");
    g();
}
```

```
int main() {
  f();

}
```

```
void g() {
    printf("g:A\n");


    printf("g:B\n");


    printf("g:C\n");

}
```

```
$ gcc experiment2.c
$ ./a.out
f:1
g:A
g:B
g:C
f:2
g:A
…
```

This way neither ...

Motivation: Quasi Parallelism – Experiment 3

```
void f() {
    printf("f:1\n");
    g();

    printf("f:2\n");
    g();

    printf("f:3\n");
    g();
}
```

```
int main() {
  f();

}
```

```
void g() {
    printf("g:A\n");
    f();

    printf("g:B\n");
    f();

    printf("g:C\n");
    f();
}
```

```
$ gcc experiment3.c
$ ./a.out
f:1
g:A
f:1
g:A
…
Segmentation fault
```

Definitely not **this way!**

Motivation: Quasi Parallelism – Experiment 4

```c
void f_start() {
    printf("f:1\n");
    f = &&l1; goto *g;

l1: printf("f:2\n");
    f = &&l2; goto *g;

l2: printf("f:3\n");
    goto *g;
}
```

How about this way?

```c
void g_start() {
    printf("g:A\n");
    g = &&l1; goto *f;

l1: printf("g:B\n");
    g = &&l2; goto *f;

l2: printf("g:C\n");
    exit(0);
}
```

Works!

```c
void (*volatile f)();
void (*volatile g)();

int main() {
    f = f_start;
    g = g_start;
    f();
}
```

```
$ gcc experiment4.c
$ ./a.out
f:1
g:A
f:2
g:B
f:3
g:C
```

Motivation: Quasi Parallelism – Experiment 4

```c
void f_start() {
    printf("f:1\n");
    f = &&l1; goto *g;

l1: printf("f:2\n");
    f = &&l2; goto *g;

l2: printf("f:3\n");
    goto *g;
}
```

How about this way?

```c
void g_start() {
    printf("g:A\n");
    g = &&l1; goto *f;

l1: printf("g:B\n");
    g = &&l2; goto *f;

l2: printf("g:C\n");
    exit(0);
}
```

```c
void (*volatile f)();
void (*volatile g)();

int main() {
    f = f_start;
    g = g_start;
    f();
}
```

Works!

```
$ gcc experiment4.c
$ ./a.out
f:1
g:A
f:2
g:B
f:3
g:C
```

Please don't try this at home!

Quasi Parallelism: First Conclusions (1)

- Quasi parallelism between two function executions cannot be achieved by function calls
  - simple function calls (experiments 1 and 2)
    → always run to completion
  - recursive function calls (experiment 3)
    → ditto, thus infinite recursion and stack overflow

Quasi Parallelism: First Conclusions (2)

- We need functions that can be **left "during execution"** and **re-entered** again
  - roughly like in experiment 4
    - program counter (PC) is saved, and restored with goto
  - ... but without the accompanying problems
    - Direct jumps from and to functions is undefined in C!
      (goto via pointers is a GCC "feature")
    - State consists of more than the PC – what about registers, stack, ...?

Agenda

- Motivation: Quasi Parallelism
  - Experiments
- **Basic Terminology**
  - Routine and Control Flow
  - Coroutine, Control Flow and Thread
  - Asymmetric and Symmetric Continuation Model
- Implementing Coroutines
  - Continuations
  - Elementary Operations
- Preview
  - Coroutines as a Basis for Multithreading
- Summary

Basic Terminology: Routine, Control Flow

- **Routine:** a finite sequence of instructions
  - e.g. function ***f***
  - Language mechanism/abstraction in almost all programming languages
  - is executed by a (routine) control flow
- **(Routine) Control flow:** a (routine) execution
  - Execution and and control flow are synonyms
  - e.g. the execution <f> of function ***f***
    - starts after activation with the first instruction of ***f***

Routines and executions have a **schema-instance relationship**.
For precise distinction, we show executions in angle brackets:

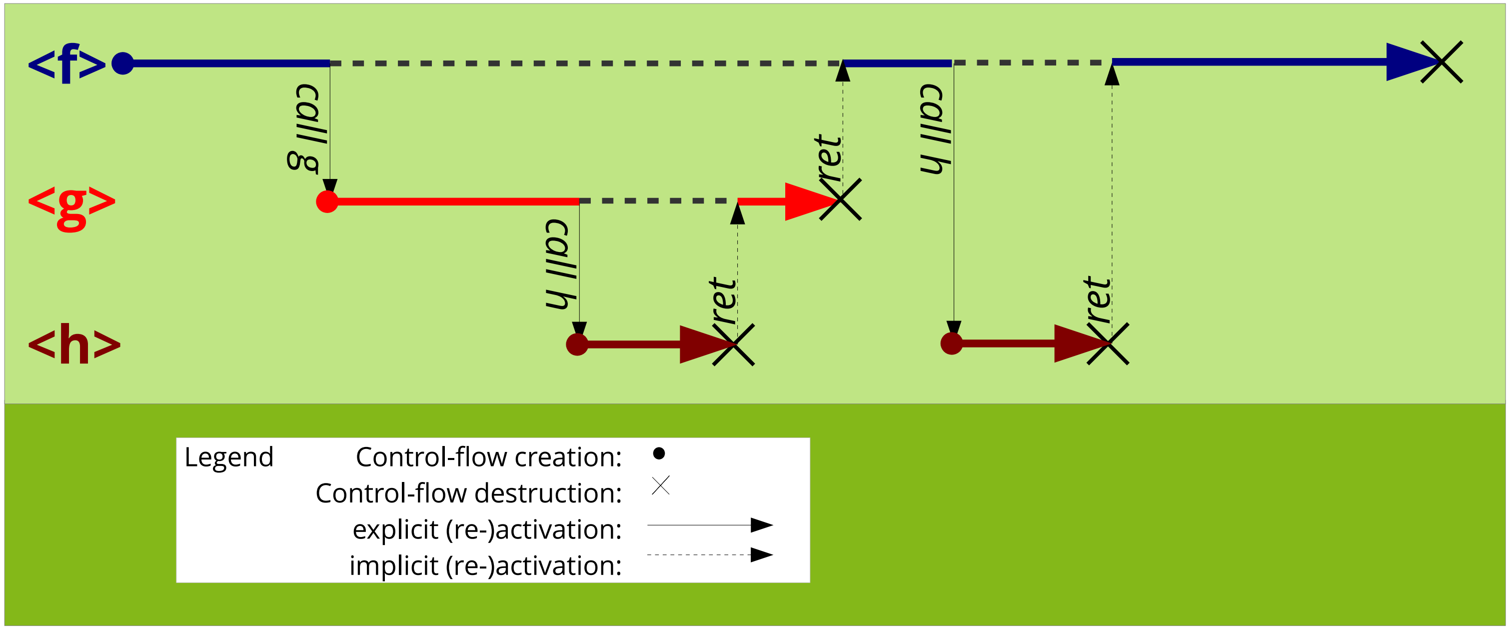
<f>, <f'>, <f''> denote **executions of function *f***.

Basic Terminology: Routine, Control Flow

- Routine control flows are created, managed and destroyed with specific **primitives**:
  - <f> ***call*** g (Execution <f> reaches instruction *call g*)
    - **creates** new execution <g> of *g*
    - **suspends** execution <f>
    - **activates** execution <g> (first instruction is executed)
  - <g> ***ret*** (Execution <g> reaches instruction *ret*)
    - **destroys** execution <g>
    - **reactivates** execution of the creating/calling control flow

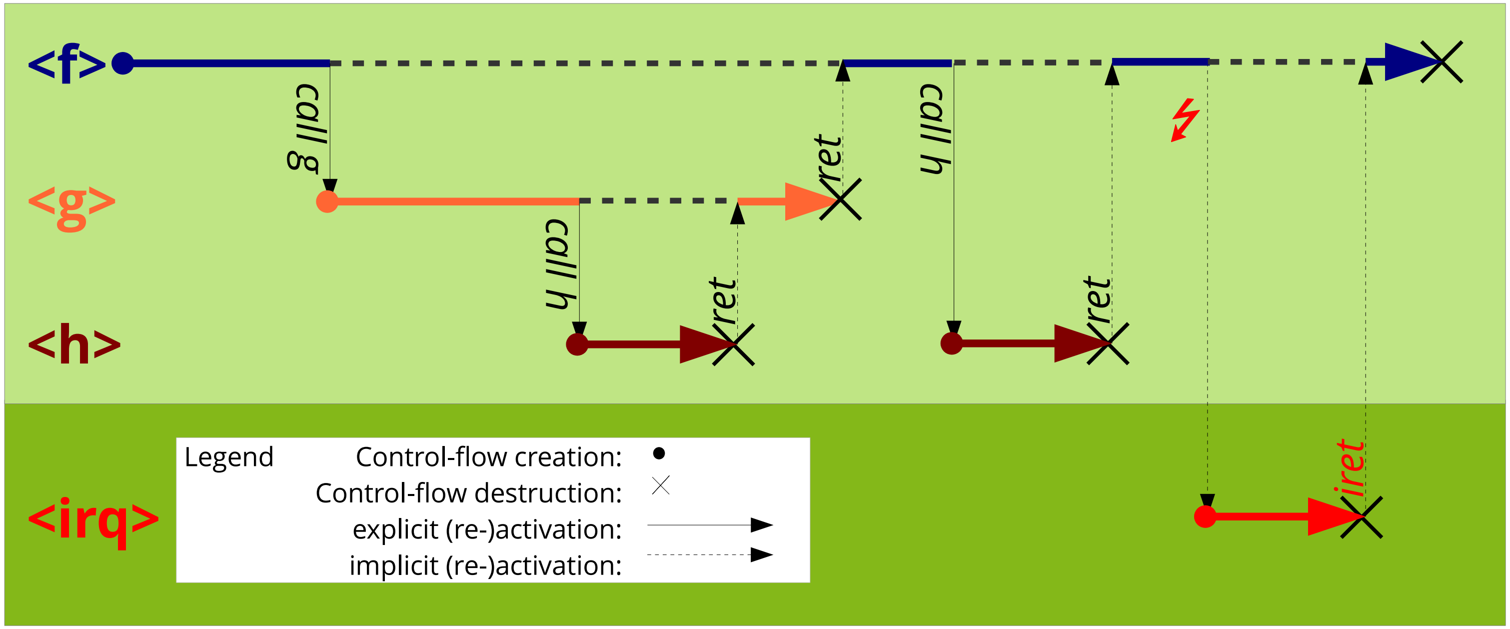
Routines → Asymmetric Continuation Model

- Routine control flows form a **continuation hierarchy**
  - Parent/child relationship between creator and created
- Activated control flows are continued following **LIFO**.
  - The most recently activated control flow always terminates first.
  - Parent is only resumed after child terminates



Routines → Asymmetric Continuation Model

- This also holds for **interrupts**
  - <f>ϟ ***irq*** like *call*, but implicit
  - <irq> ***iret*** like *ret*
- Interrupts can be understood as **implicitly** created and activated routine executions.



Basic Terminology: Coroutine

- **Coroutine:** generalized routine
  - **additionally** allows: explicit suspend/resume
  - Supported by several programming languages
    - e.g. Mono/C#, **C++20**, D, Go, **Rust**, Haskell, JavaScript, Python, …
  - is executed by a coroutine control flow
- **Coroutine control flow:** a coroutine execution
  - Control flow with own, independent state
    - Stack, registers
    - In principle an independent **thread** – **more on that later**

> Coroutines and coroutine control flows **also have a schema-instance relationship**.
>
> In the literature this distinction is unusual. Coroutine control flows are often also called “coroutines”.

Basic Terminology: Coroutine

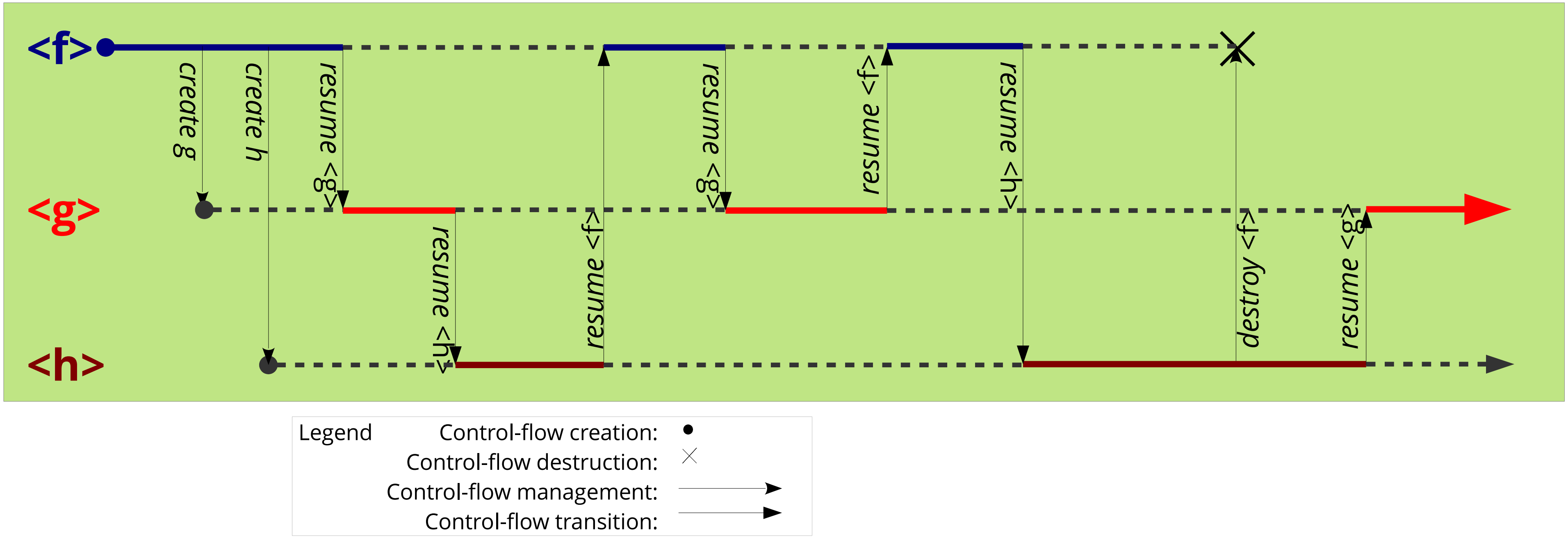
- Coroutine control flows are created, managed and destroyed by additional **primitives**:
  - ***create*** *g*
    - **creates** new coroutine execution <g> of *g*
  - <f> ***resume*** <g>
    - **suspends** coroutine execution <f>
    - **(re-)activates** coroutine execution <g>
  - ***destroy*** <g>
    - **destroys** coroutine execution <g>

Difference to routine control flows:

Activation and re-activation are **temporally decoupled** from creation and destruction.

Coroutines → Symmetric Continuation Model

- Coroutine control flows form a **continuation sequence**
  - Coroutine state is conserved across suspensions/activations
- All coroutine control flows are **equitable**
  - Cooperative multitasking
  - Continuation order is arbitrary



Coroutines and Threads

- Coroutine control flows are often also called
  - cooperative **threads**
  - **fibers**
- In principle this is true, however the terms originate from different worlds
  - Coroutine support is historically (rather) a **language concept**
  - Multithreading is historically (rather) an **operating-system concept**
  - The boundaries are blurred …
    - Language concept – (runtime) library mechanism – OS concept
- Here (in OSC) we understand coroutines as a technical means
  - to **implement** multithreading in the OS
  - in particular later also non-cooperative threads

Agenda

- Motivation: Quasi Parallelism
  - Experiments
- Basic Terminology
  - Routine and Control Flow
  - Coroutine, Control Flow and Thread
  - Asymmetric and Symmetric Continuation Model
- **Implementing Coroutines**
  - Continuations
  - Elementary Operations
- Preview
  - Coroutines as a Basis for Multithreading
- Summary

Implementation: Continuations

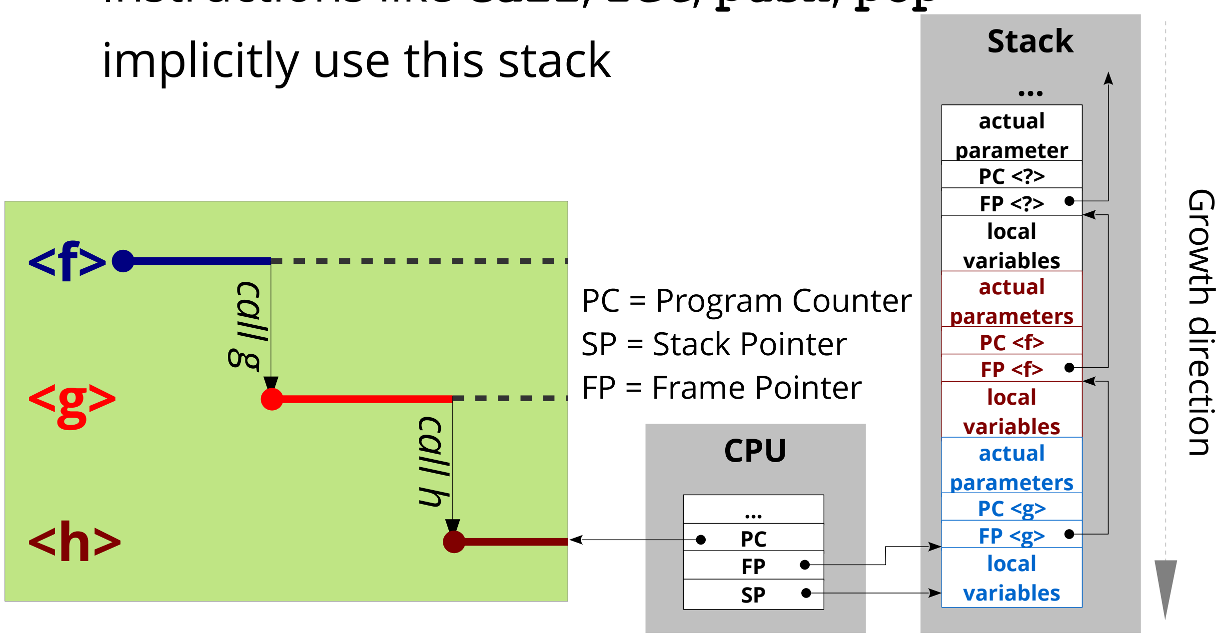
- **Continuation:** Rest / remainder of an execution
  - An object that represents a suspended control flow
    - Program counter, registers, local variables, ...
    - in short: complete control-flow state
  - Needed to reactivate the control flow

Continuations were originally conceived of in the context of *denotational semantics*.

Languages like Haskell or Scheme support continuations as central language concepts.

Routines → Asymmetric Continuation Model

- Routine continuations are instantiated on **the stack**
  - in the form of **stack frames**, created and destroyed by
    - **compiler** (and CPU) with ***call***, ***ret***
    - **wrapper function** (and CPU) at **interrupt**, ***iret***
  - Stack is provided by the hardware (CPU stack)
    - Instructions like `call`, `ret`, `push`, `pop` implicitly use this stack



Each routine control flow has a **stack frame**.

It contains the **continuation of the calling function**.

Coroutines → Symmetric Continuation Model

- A coroutine control flow needs an own stack
  - for local variables: they are part of its state
  - for subroutine calls: we don't want to do without them
  - During execution, this stack is the CPU stack.

Thus, **coroutine control flows can create routine control flows** on their stack, and activate them!

Coroutines → Symmetric Continuation Model

- A coroutine control flow needs an own stack
  - for local variables: they are part of its state
  - for subroutine calls: we don't want to do without them
  - During execution, this stack is the CPU stack.
- **Approach:** Coroutine continuations are instantiated as **stack frames on their stack**.
  - A control-flow context is represented by the stack.
  - The top-most stack element always contains the continuation.
  - **A control-flow switch corresponds to a stack switch and "return".**

In principle, this approach **implements coroutine continuations using routine continuations**.

Implementation: *resume*

- **Task:** Switch the coroutine control flow

```
// Stack-pointer type (the stack is an array of void*)
typedef void** SP;

extern "C" void resume( SP& from_sp, SP& to_sp ) {
  /* current stack frame is the continuation of the
     to-be-suspended control flow (caller of resume) */

  < save CPU stack pointer in from_sp >
  < load CPU stack pointer from to_sp >

  /* current stack frame is the continuation of the
     to-be-(re)activated control flow */

} // return
```

Implementation: *resume*

- **Task:** Switch the coroutine control flow

```
// Stack-pointer type (the stack is an array of void*)
typedef void** SP;

extern "C" void resume( SP& from_sp, SP& to_sp ) {
  /* current stack frame is the continuation of the
     to-be-suspended control flow (caller of resume) */

  < save CPU stack pointer in from_sp >
  < load CPU stack pointer from to_sp >

  /* curren
     to-be-

} // return
```

Problem: non-volatile registers

The stack frame does **not contain any non-volatile registers**, because the caller expects them not to be modified.

However, we return **to a different caller**.

Implementation: *resume*

- **Problem:** non-volatile registers
  - Stack frame does not contain any non-volatile registers
  - so they must be explicitly saved and restored
- Implementation variants
  - Save non-volatile registers to a special data structure
  - or save them as “local variables” on the stack:

```
extern "C" void resume( SP& from_sp, SP& to_sp ) {
  /* current stack frame is the continuation of the
     to-be-suspended control flow (caller of resume) */
  < push non-volatile registers on the stack >
  < save CPU stack pointer in from_sp >
  < load CPU stack pointer from to_sp >
  < pop non-volatile registers from the stack >
  /* current stack frame is the continuation of the
     to-be-(re)activated control flow */

} // return
```

Implementation: *resume*

- *resume* is architecture specific
  - Stack-frame structure
  - Non-volatile registers
  - Stack growth direction
- And we have to touch registers → **Assembler**

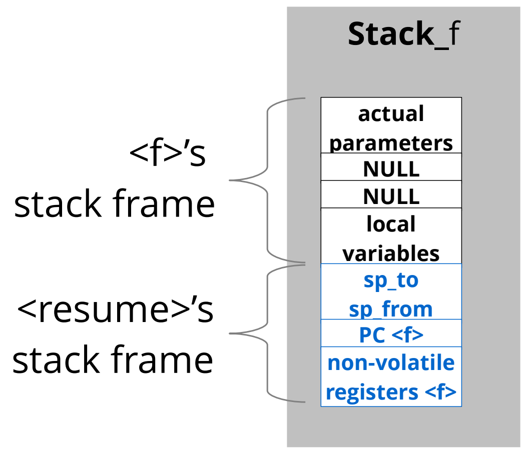
Example Motorola 68000

```
// extern "C" void resume( SP& sp_from, SP& sp_to )
resume:
  move.l  4(sp), a0                 // a0 = &sp_from
  move.l  8(sp), a1                 // a1 = &sp_to
  movem.l d2-d7/a2-a6, -(sp)        // nv registers to stack
  move.l  sp, (a0)                  // sp_from = sp
  move.l  (a1), sp                  // sp = sp_to
  movem.l (sp)+, d2-d7/a2-a6        // load nv regs. from stack
  rts                               // “return”
```

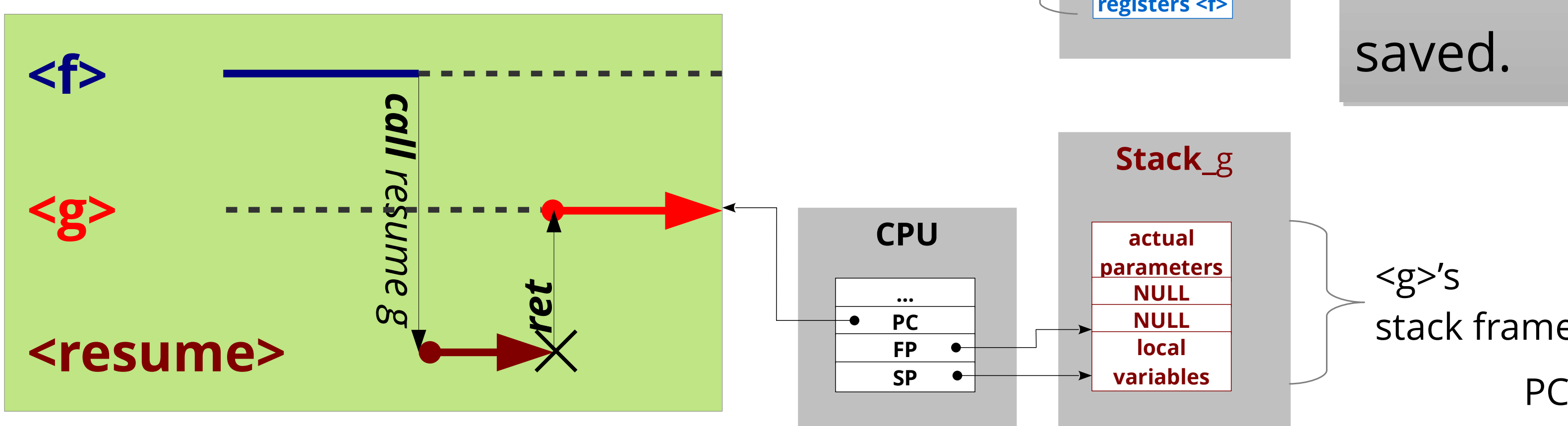
Example: *resume* usage

- Coroutine control flow <f> handed over to <g>
  - <f> is suspended, <g> is active

<f> called *resume* as a routine. This call created a stack frame on <f>'s stack.



<resume>'s stack frame describes <f>'s continuation. Additionally, the non-volatile registers were saved.



PC = Program Counter
SP = Stack Pointer
FP = Frame Pointer

Implementation: *create*

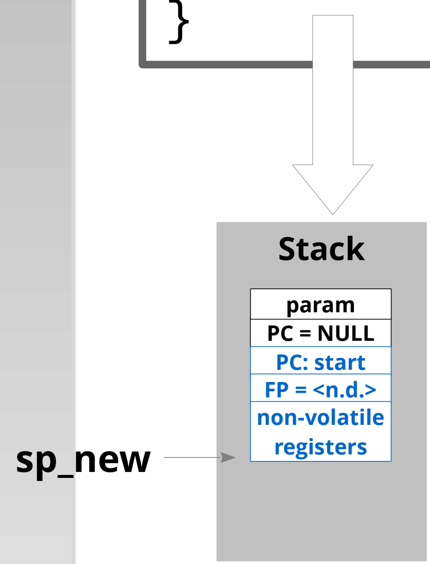
- **Task:** Create coroutine control flow <start>
  - We need
    - **Stack memory** (somewhere, **global**) `static void *stack_start[ 256 ];`
    - a **stack pointer** `SP sp_start = &stack_start[ 256 ];`
    - a **start function** `void start( void* param ) {...}`
    - **parameters** for the start function
  - We create the coroutine control flow in suspended state
    - Stack represents the context
    - Execution should not start until *resume* is called
- **Approach:** ***create*** generates two stack frames
  - "as if" the start function had already called *resume* before:
    - the start function's frame (created by a "virtual caller")
    - *resume*'s frame (contains start function's continuation)
  - First *resume* "returns" to the begin of the start function

Implementation: *create*

Example Motorola 68000

```
void create( SP& sp_new, void (*start)(void*), void* param) {
   *(--sp_new) = param; // start-function parameter
   *(--sp_new) = 0;     // (non-existent) caller’s return addr.

   *(--sp_new) = start; // start() address
   sp_new -= 11;        // n-v. registers (values don’t matter)
}
```



Because we “return” to a function’s first instruction, the frame structures are very simple. At this continuation point, a function has

- not yet put any **local variables** (or a frame pointer) on the stack
- not yet put **parameters** (for *resume*) on the stack, and
- no assumptions regarding **values of non-volatile registers**.

Implementation: *destroy*

- **Task:** destroy coroutine control flow
- **Approach:** deallocate control-flow context
  - corresponds to freeing the context variable (stack pointer)
  - Stack memory can be used otherwise afterwards.

At last, that's not really complicated.

Agenda

- Motivation: Quasi Parallelism
  - Experiments
- Basic Terminology
  - Routine and Control Flow
  - Coroutine, Control Flow and Thread
  - Asymmetric and Symmetric Continuation Model
- Implementing Coroutines
  - Continuations
  - Elementary Operations
- **Preview**
  - Coroutines as a Basis for Multithreading
- Summary

Next Up: Kernel-Level Threads

- Coroutines are (originally) a language concept
  - Multitasking on language level
  - We just “retrofitted” C with this
  - Context switches need no system privileges
    (do not necessarily involve the OS kernel)
- Prerequisite for multitasking is, however: **Cooperation**
  - Applications must be **implemented as coroutines**
  - Applications must **know** each other
  - Applications must **activate** each other

> For unrestricted multiprogramming, these prerequisites are **unrealistic**!

Next Up: Kernel-Level Threads

- **Alternative:** Perceive "cooperation capability" as an operating-system responsibility
- **Approach:** Run applications "unnoticed" as independent threads
  - **OS** takes care of **creating** coroutine control flows
    - Each application is called as a routine from an **OS coroutine**
    - consequently, indirectly every application is implemented as a coroutine
  - **OS** takes care of **suspending** running coroutine control flows
    - so that applications do not have to be cooperative
    - necessitates a **preemption mechanism**
  - **OS** takes care of **selecting** the next coroutine control flow
    - so that applications do not have to know each other
    - necessitates a **scheduler**

Next Up: Kernel-Level Threads

- **Alternative:** Perceive “cooperation capability” as an operating-system responsibility
- **Approach:** Run applications “unnoticed” as independent threads
  - **OS** takes
    - Each ne
    - conse as a coroutine
  - **OS** takes care of **suspending** running coroutine control flows
    - so that applications do not have to be cooperative
    - necessitates a **preemption mechanism**
  - **OS** takes care of **selecting** the next coroutine control flow
    - so that applications do not have to know each other
    - necessitates a **scheduler**

More on that in the exercise + lab and in the next lecture

Agenda

- Motivation: Quasi Parallelism
  - Experiments
- Basic Terminology
  - Routine and Control Flow
  - Coroutine, Control Flow and Thread
  - Asymmetric and Symmetric Continuation Model
- Implementing Coroutines
  - Continuations
  - Elementary Operations
- Preview
  - Coroutines as a Basis for Multithreading
- **Summary**

Summary

- Our goal was to enable “quasi parallelism”
  - Run functions “alternatingly”, in “little” steps
    - Suspension and reactivation of function executions
    - New term: **Continuation**
- **Routines** → asymmetric continuation model
  - Execution in LIFO order (and thereby not “quasi parallel”)
  - CPU and compiler provide primitives
- **Coroutines** → symmetric continuation model
  - Execution in arbitrary order
    - necessitates own context: registers, stack
  - Primitives generally not provided by CPU/compiler
- Threads are OS-managed coroutines