Übung zu Betriebssysteme

Aufgabe 4: Threadumschaltung

Wintersemester 2020/21

Bernhard Heinloth & Christian Eichler

Lehrstuhl für Informatik 4
Friedrich-Alexander-Universität Erlangen-Nürnberg

Lehrstuhl für Verteilte Systeme
und Betriebssysteme

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

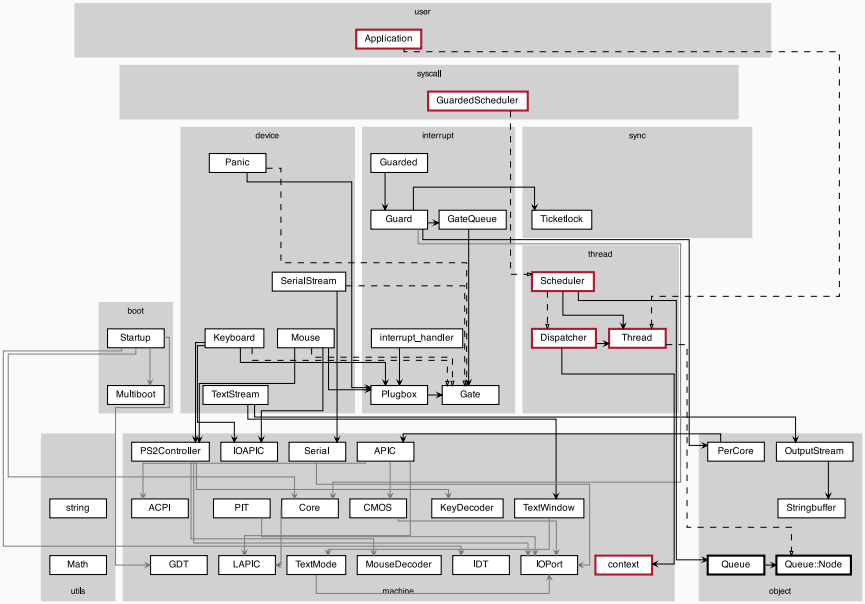
TECHNISCHE FAKULTÄT

Lernziele

- Auffrischen der Assemblerkenntnisse
- Verständnis der Abläufe beim Threadwechsel
- Unterscheiden von aktiven & passiven Objekten

Umsetzung

- Kontextwechsel
- Threadverwaltung
- Auslagerung von Anwendungen


user
Application
syscall
GuardedScheduler
device
interrupt
sync
Panic
Guarded
Guard
GateQueue
Ticketlock
thread
SerialStream
Scheduler
boot
Startup
Keyboard
Mouse
interrupt_handler
Dispatcher
Thread
Multiboot
TextStream
Plugbox
Gate
PS2Controller
IOAPIC
Serial
APIC
PerCore
OutputStream
string
ACPI
PIT
Core
CMOS
KeyDecoder
TextWindow
Stringbuffer
Math
GDT
LAPIC
TextMode
MouseDecoder
IDT
IOPort
context
Queue
Queue::Node
utils
machine
object

Threads

```
class Thread {
    StackPointer sp;
  public:
    Thread();
    virtual void action() = 0;
};
```

Adresse einer virtuellen Member-Funktion nicht (einfach) ermittelbar

```
class Foo : public Thread {
  void action() { ... }
};
```

```
class Bar : public Thread {
  void action() { ... }
};
```

```
Thread * x = &app;
x->action(); // Foo::action oder Bar::action ?
```

Threads

```
class Thread {
    StackPointer sp;
  public:
    Thread();
    virtual void action() = 0;
};
```

Adresse einer virtuellen Member-Funktion nicht (einfach) ermittelbar

```
void kickoff(Thread* t){
  t->action();
}
```

Threads: Stack aufsetzen

```
void * prepareContext(void * tos,
                      void (*kickoff)(void*),
                      void * param);
```

Präpariert einen Stack für den ersten Einsprung

- statisch übergebenen Speicher `tos` als Stack aufsetzen
- nach dem Einsprung soll `kickoff` mit `param` aufgerufen werden
- Stackpointer `kernel` in Thread entsprechend setzen
- in C++ (statt Assembler)
- Pointerarithmetik ist hilfreich

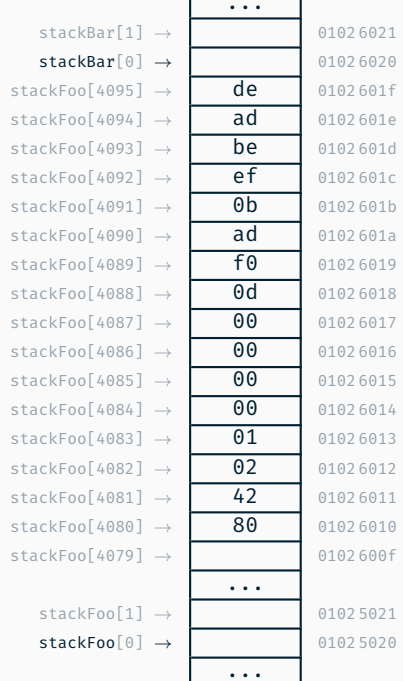
```
#define STACKSIZE 4096
char stackBar[STACKSIZE];
char stackFoo[STACKSIZE];

void* tos = stackFoo + STACKSIZE;
void** rsp = (void**) tos;

rsp--;
// rsp = &(stackFoo[4088])
*rsp = (void*) 0xdeadbeef0badf00d;

rsp--;
// rsp = &(stackFoo[4080])
extern Thread foo;
// &foo = 0x102 4280
*rsp = (void*) &foo;

// ...
```



Stapelüberlauf

- Die Stacks sind nur 4K groß
- Die Stacks liegen (oft) hintereinander
- Das gilt auch für die initialen Stacks beim Systemstart

→ Mechanismus zum Erkennen von Überlaufen hilfreich („Stack Canary“)

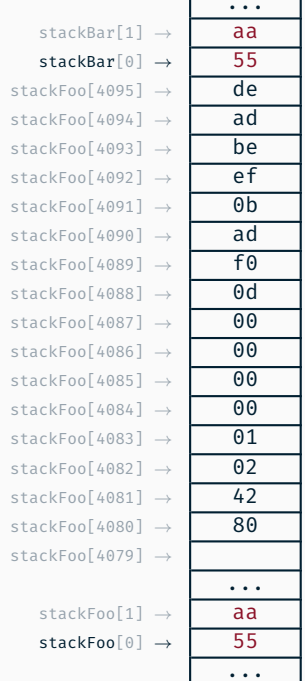
```
#define STACKSIZE 4096
char stackBar[STACKSIZE];
char stackFoo[STACKSIZE];

void* tos = stackFoo + STACKSIZE;
void** rsp = (void**) tos;

rsp--;
// rsp = &(stackFoo[4088])
*rsp = (void*) 0xdeadbeef0badf00d;

rsp--;
// rsp = &(stackFoo[4080])
extern Thread foo;
// &foo = 0x102 4280
*rsp = (void*) &foo;

// ...
```



Scheduler

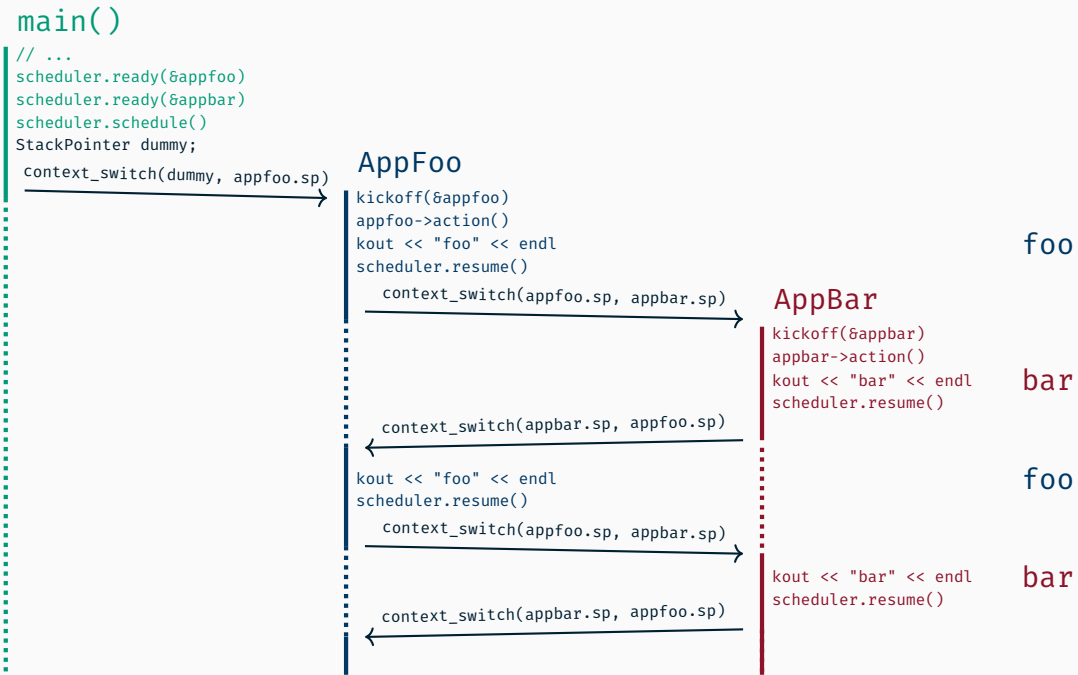
Scheduler verwaltet Koroutinen (Threads) zentral

```
class Scheduler {
    Queue<Thread> readylist;
  public:
    void ready(Thread *);
    void schedule();
    void resume();
    void exit();
    void kill(Thread *);
};
```

```
// Thread AppFoo
void AppFoo::action(){
  while (1){
    kout << "foo" << endl;
    scheduler.resume();
  }
}
```

```
// Thread AppBar
void AppBar::action(){
  while (1){
    kout << "bar" << endl;
    scheduler.resume();
  }
}
```

```
// main.cc
AppFoo appfoo;
AppBar appbar;
int main (){
  // ...
  scheduler.ready(&appfoo);
  scheduler.ready(&appbar);
  scheduler.schedule();
}
```


main()
// ...
scheduler.ready(&appfoo)
scheduler.ready(&appbar)
scheduler.schedule()
StackPointer dummy;
context_switch(dummy, appfoo.sp)
AppFoo
kickoff(&appfoo)
appfoo->action()
kout << "foo" << endl
scheduler.resume()
context_switch(appfoo.sp, appbar.sp)
AppBar
kickoff(&appbar)
appbar->action()
kout << "bar" << endl
scheduler.resume()
context_switch(appbar.sp, appfoo.sp)
kout << "foo" << endl
scheduler.resume()
context_switch(appfoo.sp, appbar.sp)
kout << "bar" << endl
scheduler.resume()
context_switch(appbar.sp, appfoo.sp)
foo
bar
foo
bar

E_0 (Anwendung)

$E_{\frac{1}{2}}$ (Epilog)

E_1 (IRQ/Prolog)

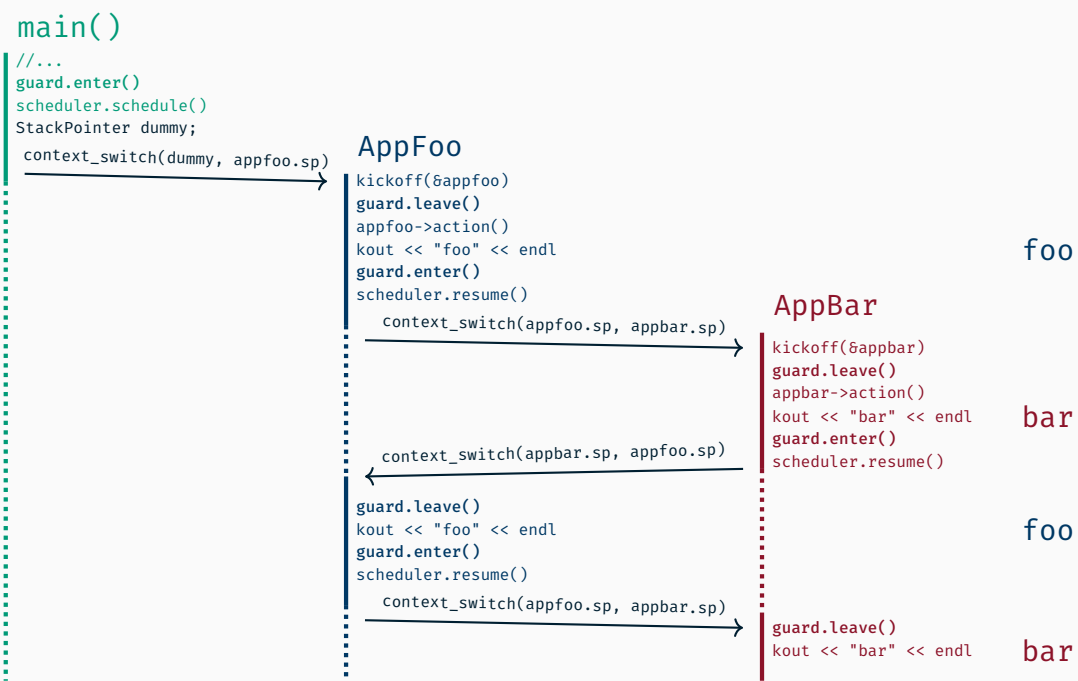
Scheduler

Scheduler verwaltet Koroutinen (Threads) zentral

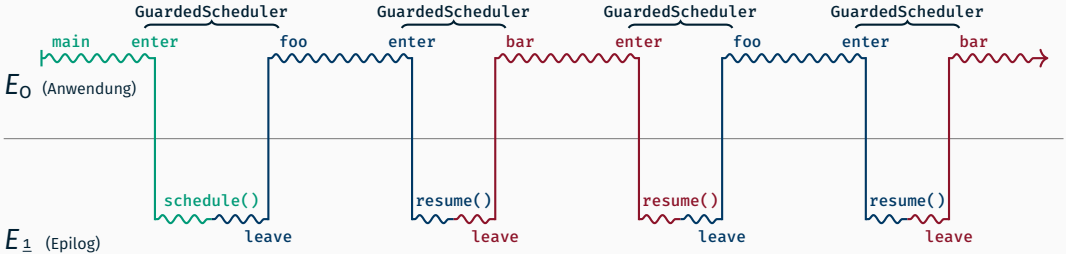
```
class Scheduler {
    Queue<Thread> readylist;
  public:
    void ready(Thread *);
    void schedule();
    void resume();
    void exit();
    void kill(Thread *);
};
```

OOStuBS immer synchrone Aufrufe
→ Konsistenz gesichert

MPStuBS Synchronisation notwendig


main()
//...
guard.enter()
scheduler.schedule()
StackPointer dummy;
context_switch(dummy, appfoo.sp)
AppFoo
kickoff(&appfoo)
guard.leave()
appfoo->action()
kout << "foo" << endl
guard.enter()
scheduler.resume()
context_switch(appfoo.sp, appbar.sp)
AppBar
kickoff(&appbar)
guard.leave()
appbar->action()
kout << "bar" << endl
guard.enter()
scheduler.resume()
context_switch(appbar.sp, appfoo.sp)
guard.leave()
kout << "foo" << endl
guard.enter()
scheduler.resume()
context_switch(appfoo.sp, appbar.sp)
guard.leave()
kout << "bar" << endl
foo
bar
foo
bar

- Scheduling auf der Epilogebene
  (➔ automatische Synchronisation der Ready-Liste)
- Kapselung in `GuardedScheduler`
  (➔ „Systemaufruf“ für Anwendungen)



E_1 (IRQ/Prolog)