Beware of some German slides!

Betriebssysteme

Vorlesung im Herbstsemester 2008
Universität Mannheim

Kapitel 5c:
System Calls and
Signals (User Level Interrupts) in ULIX

Felix C. Freiling

Lehrstuhl für Praktische Informatik 1

Universität Mannheim

(Diese Folien sind nicht Teil des Basiskurses)

Overview

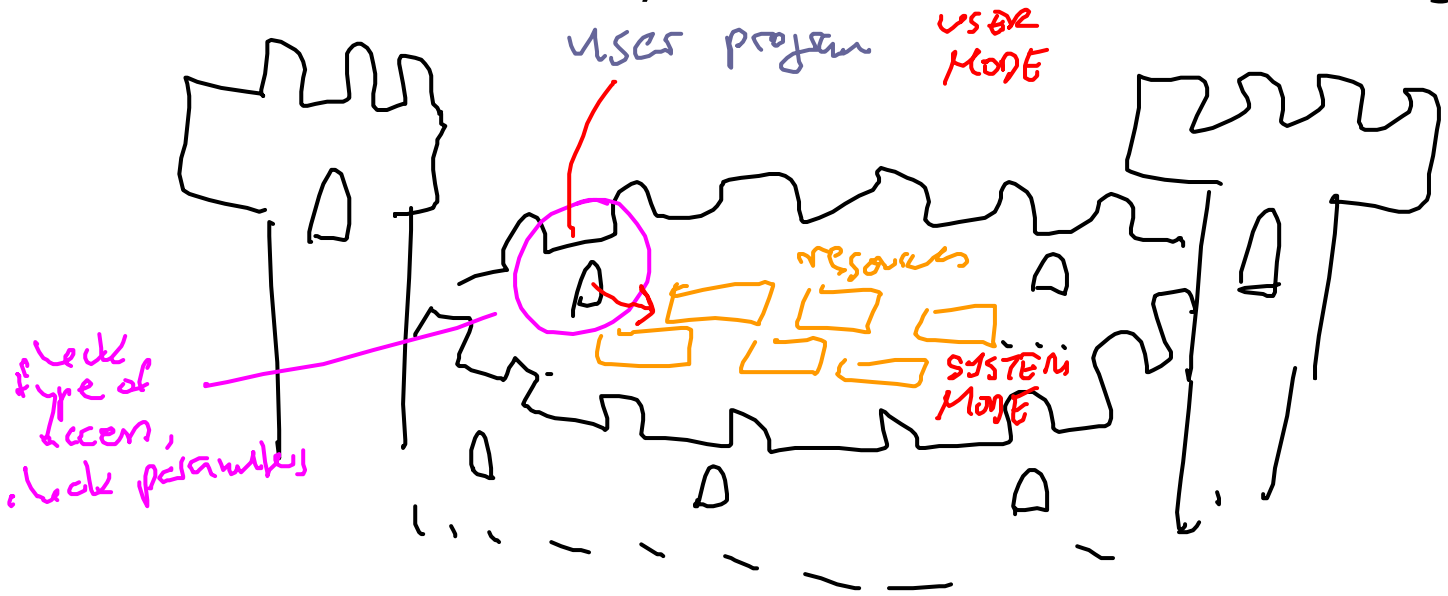
- System calls
- Interrupt handlers
- Signals (user level interrupts)
- Implementation of signals

User Space and Kernel Space

- User programs run in user mode
- Operating system code runs in system mode
  - No user code should run in system mode (protection!)
- User programs must be able to use services of the operating system
  - Examples: Start a new thread, create a semaphore
- System calls allow controlled transition from user mode to system mode

Reference Monitor

- Reference monitors offer controlled access to resources
  - Operating systems should be implemented as reference monitors
  - If check fails, access to resources is not granted



System Calls

- System calls define the interface of the operating system (seen as a reference monitor)
  - A system call is a controlled call of an operating system function
- System calls have a well-defined interface
  - Documented in “manual pages”
  - Can be called conveniently from programming languages

man fork

```
int main(void) {

   fork();

}
```

sonic.informatik.uni-mannheim.de - PuTTY

```
FORK(2)                    BSD System Calls

NAME
     fork -- create a new process

SYNOPSIS
     #include <sys/types.h>
     #include <unistd.h>

     pid_t
     fork(void);

DESCRIPTION
     Fork() causes creation of a new process.  The new process (child process)
     is an exact copy of the calling process (parent process) except for the
     following:

           o   The child process has a unique process ID.

           o   The child process has a different parent process ID (i.e., the
               process ID of the parent process).

           o   The child process has its own copy of the parent's descriptors.
:
```

Are System Calls C Functions?

- Compiler translates C program to executable machine code
- How is system call translated?

- Must do parameter checking
  - Must be done in system mode (protection!)
- Must perform transition to system mode
  - Only possible through interrupt

Low Level System Calling

- Hypothetical system call **`int foo(int x)`**
  - Assumption: Parameter x available in register R0
  - Convention: Return parameter should be in R0 too after call
- Possible realization:
  - Pass parameter to kernel via user stack
  - Pass id of system call to user stack
  - Trap into operating system
  - Retrieve return parameter from user stack to register

Machine Instructions

- Assembly code (ULIX assembler):

```
push r0    // parameter on stack
push FOO   // id of foo system call
trap 1     // trap interrupt level
pop r0     // retrieve return param.
```

System Calls as Library Functions

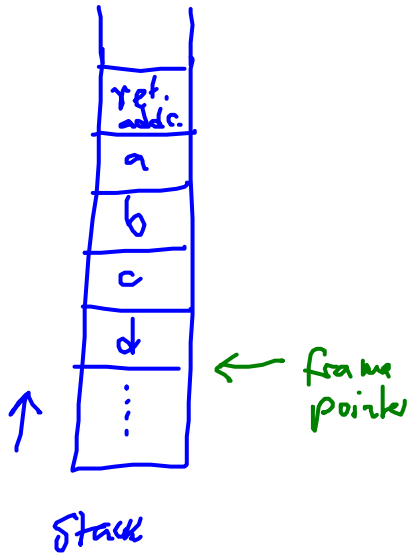
- Calling of TRAP is cumbersome
- Rather use pre-defined library functions with clean C function call interface:

```
int foo(int x) {
   // some magic with TRAP etc.
}
```

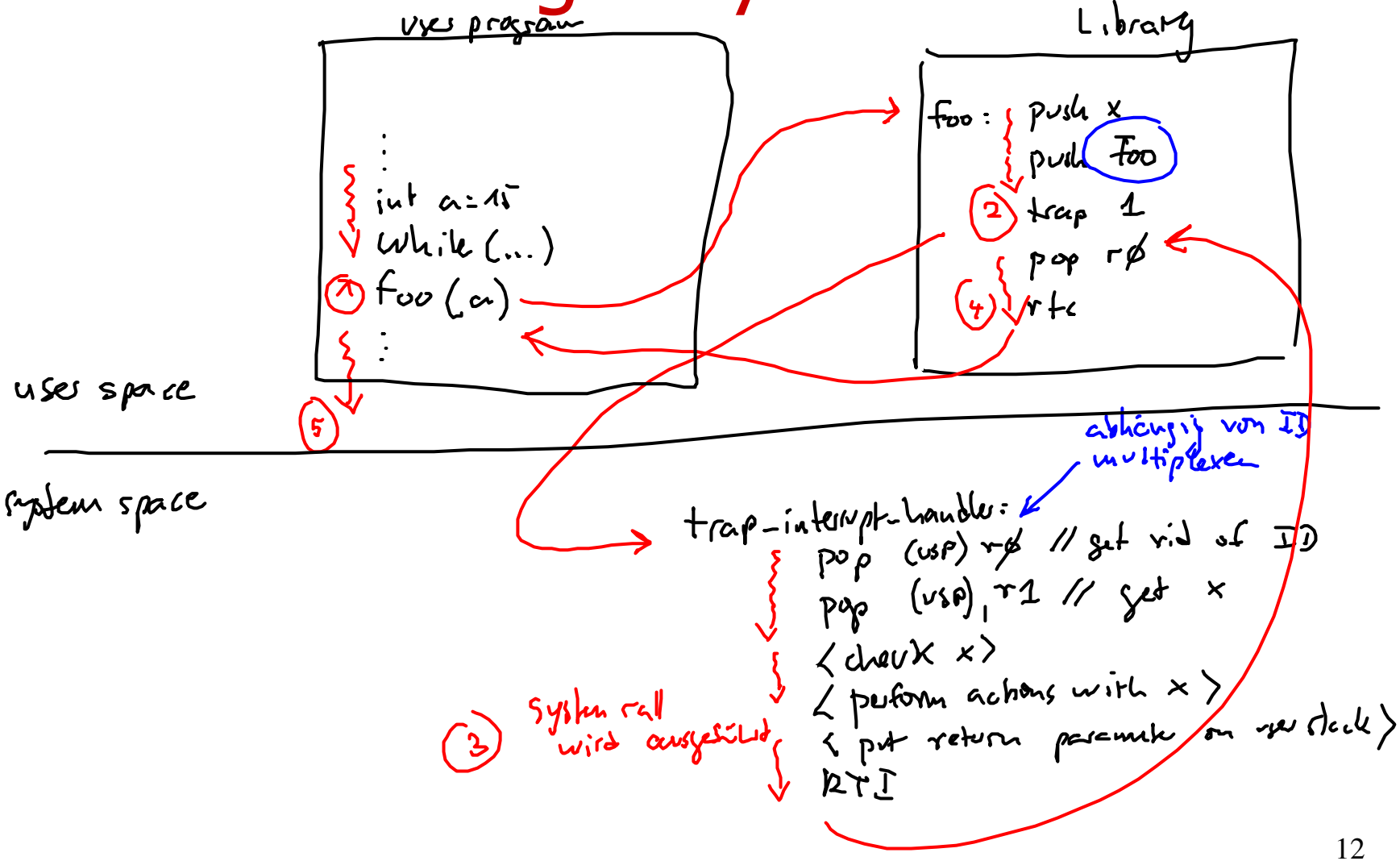
- Handling of parameters dependent on C conventions

C Calling Conventions

- Depends on C compiler and architecture
- ULIX style:
  - Push parameters to stack in reverse order
  - Then jump to subroutine
  - Return parameter is in register R0
- Example:
  - `void bar(int a, b, c, d) { ... }`
- Leads to:
  - `push d`
  - `push c`
  - `push b`
  - `push a`
  - `jsr bar`
  - `// return parameter is now in R0`
- Caller pushes to stack, callee pops stack and prepares R0



Calling a System Call



Comments

1. Call local library routine with parameters
2. Library routine prepared parameters (no check)
3. Calls TRAP
4. TRAP interrupt handler multiplexes different system calls
5. Checks parameters
6. Performs functionality
7. Prepared return values
8. Calls RTI
9. Prepares return values according to C conventions

Overview

- System calls
- **Interrupt handlers**
- Signals (user level interrupts)
- Implementation of signals

Interrupt Handlers in ULIX

- Interrupt handlers are "normal" assembler subroutines
  - Parameter passing via user stack
  - Return value via R0
- Default interrupt handler: panic
  - Calls undocumented ULIX machine instruction **`dump`**
  - Dumps processor context to the screen
    - Easy to implement in an emulator
    - Much more complicated in practice

number	class/example
0	none
1	TRAP
2	timer interrupt
3	I/O interrupt
4	MMU interrupt (page fault)
5	division by zero (non-maskable)
6	basic protection violation (non-maskable)
7	invalid machine instruction encoding (non-maskable)

Handlers

- Level 1: multiplex system calls
- Level 2: resign, assign
- Level 3: deblock thread waiting for DMA
- Level 4: handle page fault
- Level 5: terminate KLT (or dump?)
- Level 6: terminate KLT (or dump?)
- Level 7: dump
- All other (249) handlers default to **`panic`**

Overview

- System calls
- Interrupt handlers
- **Signals (user level interrupts)**
- Implementation of signals

Interrupts and User Space

- Interrupts and interrupt handlers happen in kernel space
  - Predefined functionality, carefully prepared
- Interrupts are transparent to user programs
  - System calls are like function calls, even if DMA happens (thread blocks)
- Sometimes we need possibility to execute functions “asynchronously” in user programs
- Example: Thread A wants to terminate thread B, but allow B to perform cleanup

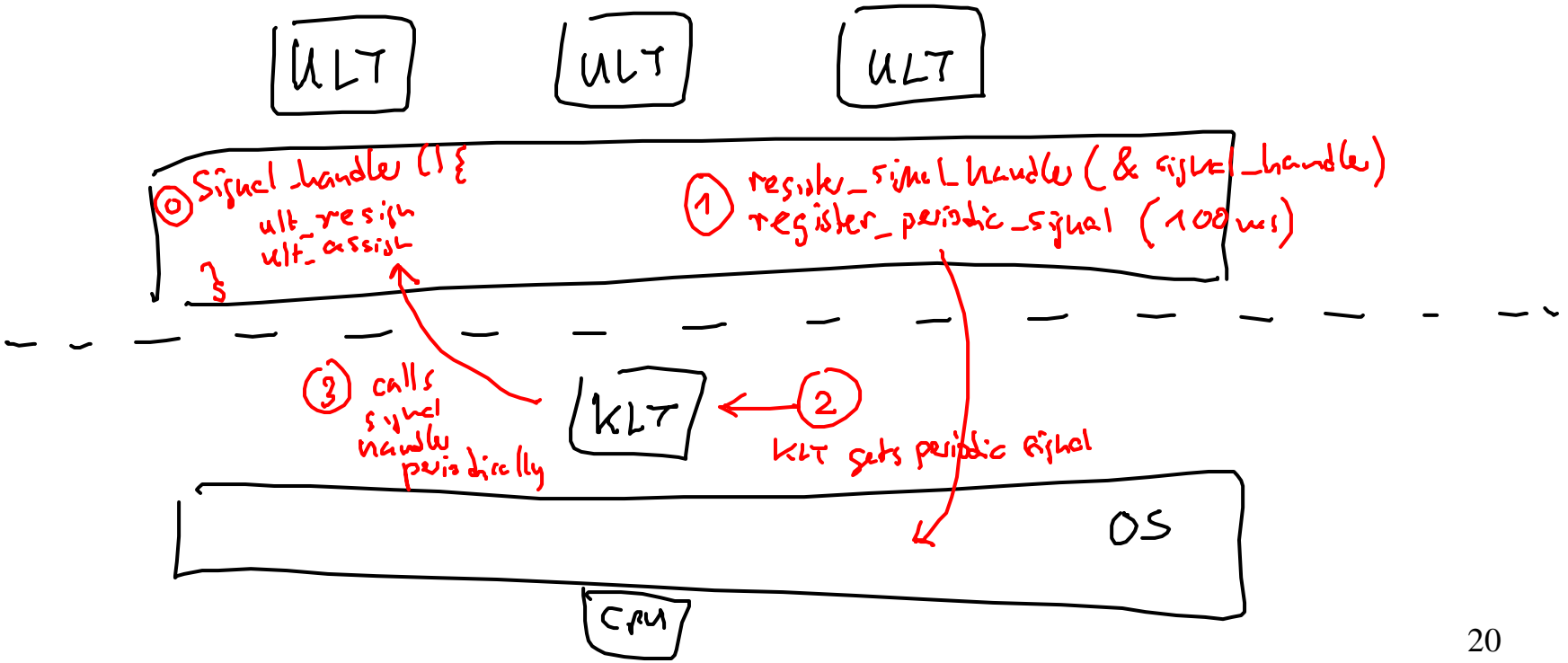
Signals (User Level Interrupts)

- Idea:
  - Different signal levels (0-7 in ULIX)
  - Threads can send other threads signals at a certain level
  - Threads can register signal handlers (C functions) for certain signal levels
  - Threads can request periodic signals for themselves
- System calls:
  - `register_signal(int level, void* handler)`
  - `send_signal(int tid, int level)`
  - `periodic_signal(int level, int millis)`

(void*) (handle(int))

Example

- User level threads package
- Periodic signal can "simulate" timer interrupt



Overview

- System calls
- Interrupt handlers
- Signals (user level interrupts)
- **Implementation of signals**

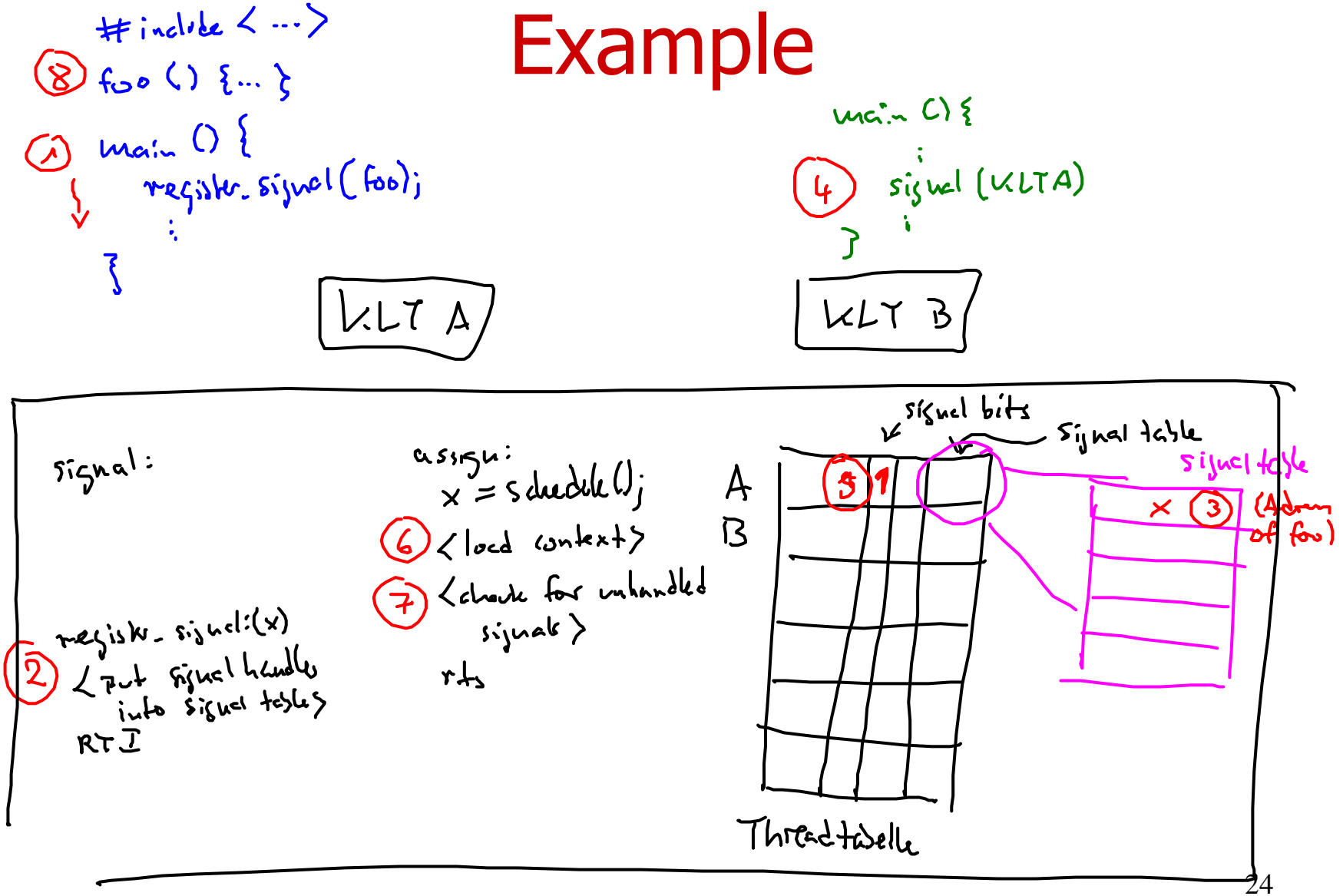
Signals vs. Interrupts

- High level concepts implemented using low level concepts
- (High level) signals must somehow be tied to (low level) interrupts
  - Example: periodic signal implemented using timer unit
- Problem: Signal handler can only be called when signalled thread is on CPU
  - Must invoke signal handler in a "deferred" manner

Signal Bits and Signal Table

- Idea: extend TCB with
  - signal bits (one for each signal level)
  - table of function addresses (signal table, one function address per level)
- When a signal handler is registered, store address in signal table
- When a signal is sent to thread, set the appropriate bit
- When a thread is scheduled, check for uncalled signal handlers before control passes back to user program of that thread

Example


#include <...>
8 foo () {... }
1 main () {
register_signal(foo);
:
}
VLT A
main C) {
:
4 signal (VLTA)
i
}
VLT B
signal:
assign:
x = schedule();
6 <load context>
7 <check for unhandled signals>
rts
register_signal:(x)
2 <put signal handler into signal table>
RTI
signal bits
signal table
A
B
5
1
signal table
x
3
(Adress of foo)
Thread table

Discussion

- Note that user program and signal handlers are executed in user mode
- There is no real-time guarantee on execution of signal handlers
  - Depends on when thread is scheduled again

Summary

- System calls
- Interrupt handlers
- Signals (user level interrupts)
- Implementation of signals

- Nothing implemented yet ...