Universität Bamberg

Operating Systems Engineering

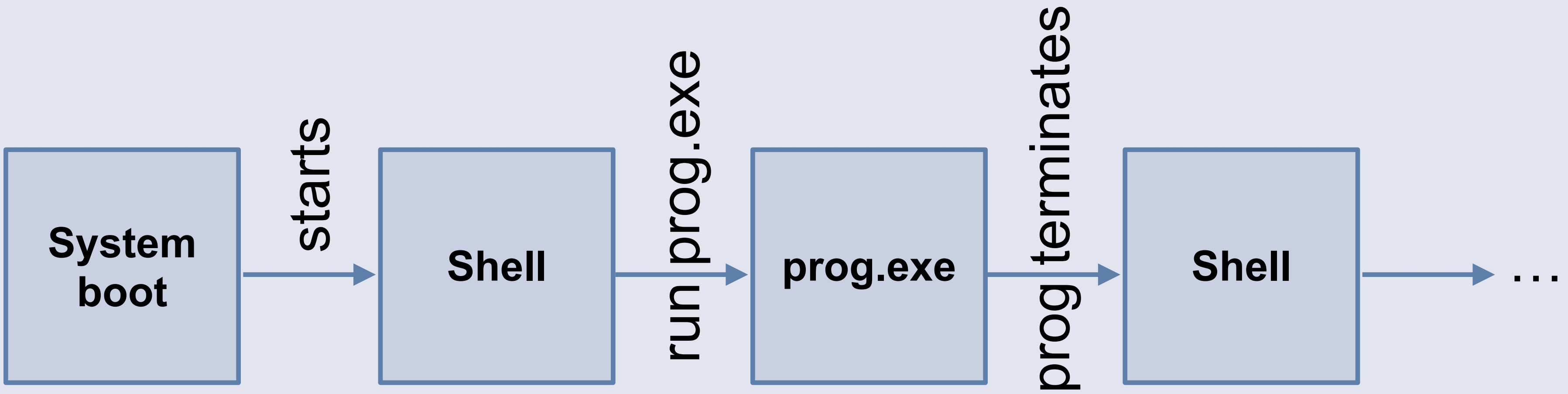
Lecture 5: Multiple processes

Michael Engel (michael.engel@uni-bamberg.de)
Lehrstuhl für Praktische Informatik, insb. Systemnahe Programmierung
https://www.uni-bamberg.de/sysnap

Switching processes

Universität Bamberg

- **With a separate address space for processes…**
  - we can now easily ***exchange*** the process in memory!

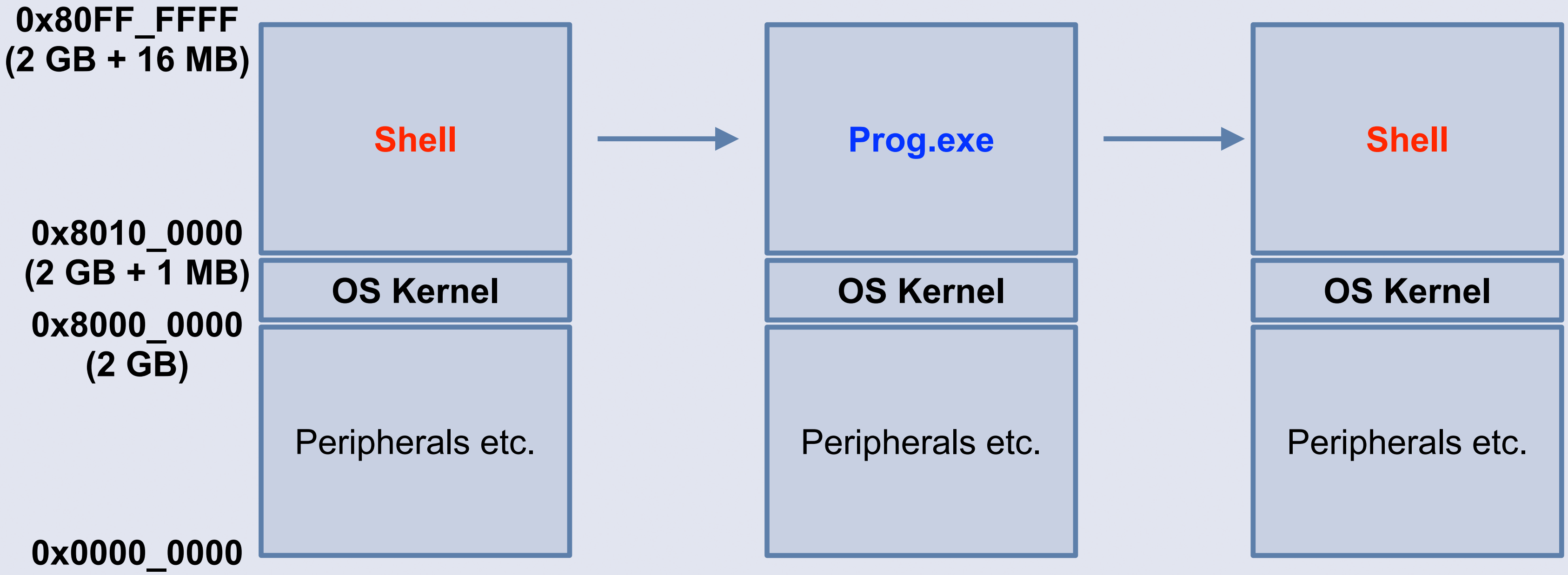


- The OS kernel initially starts a given first process (shell)
- The shell then accepts commands
  - e.g. `run prog.exe` loads and runs the given program
  - to do this, the shell asks the kernel to ***execute*** the new program

Switching processes: memory view

Universität Bamberg

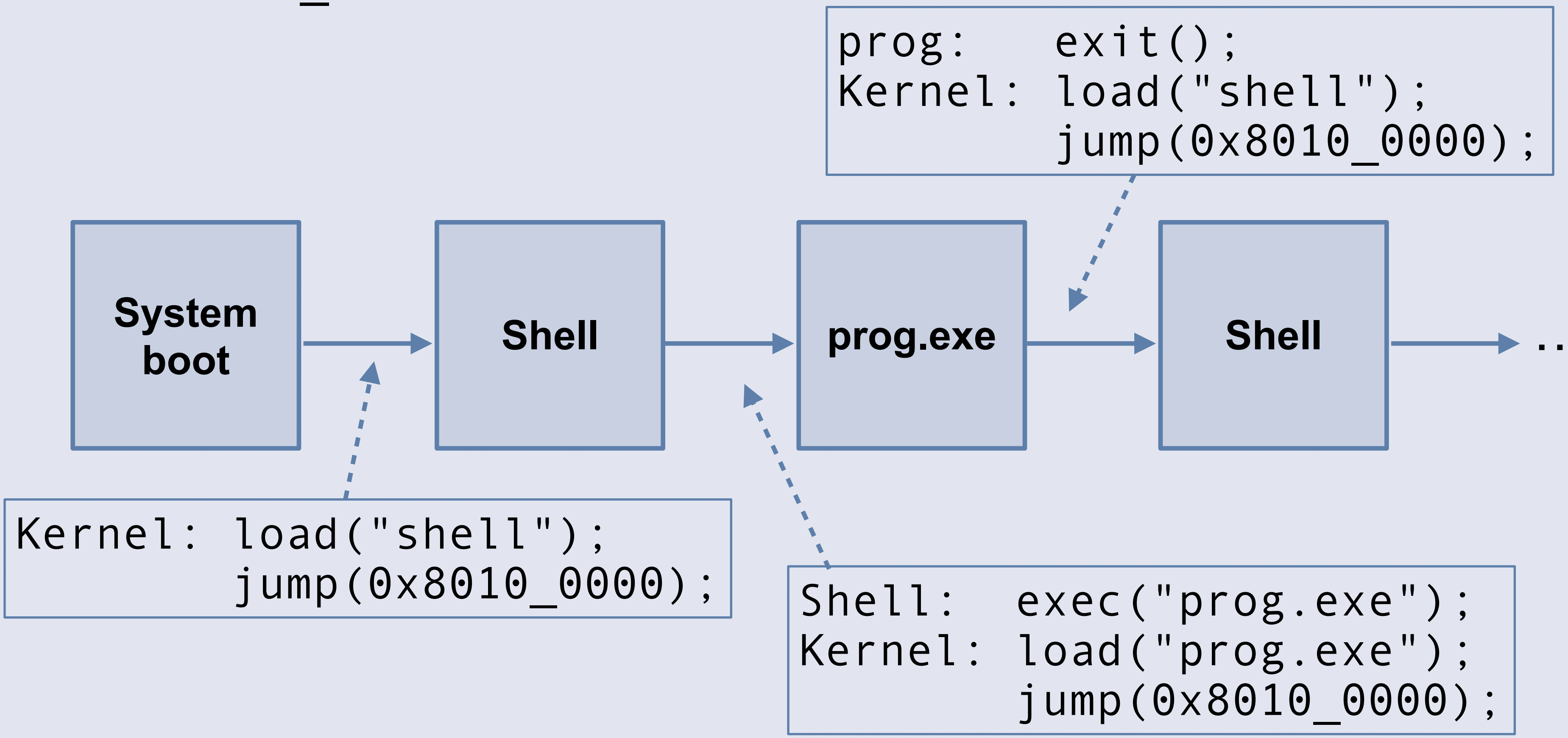
- **The OS has to replace the application memory range with the new application**
  - Kernel code and data remains in its memory range



Managing process switching

Universität Bamberg

- **With a separate address space for processes…**
  - we could try to load different programs into memory at 0x8010_0000 one after the other



Take it easy

Universität Bamberg

- **Let's start a bit more simple…**
  - Just build two programs:
    - one prints an 'a' (syscall 2) and exits (syscall 42)
    - the other prints a 'b' (syscall 2) and exits (syscall 42)
- ***Important:***
  - Both programs *run to completion*
  - Both programs are linked to the same address: 0x8010_0000
- Build a kernel that switches between the two

user1.c

```
int main(void) {
    syscall(2, 'a');
    syscall(42, 0);
    return 0;
}
```

user2.c

```
int main(void) {
    syscall(2, 'b');
    syscall(42, 0);
    return 0;
}
```

Let’s do it like Linus Torvalds…

Universität Bamberg

This is similar to what Linus Torvalds did in his first experiments for what would become Linux (from [1])

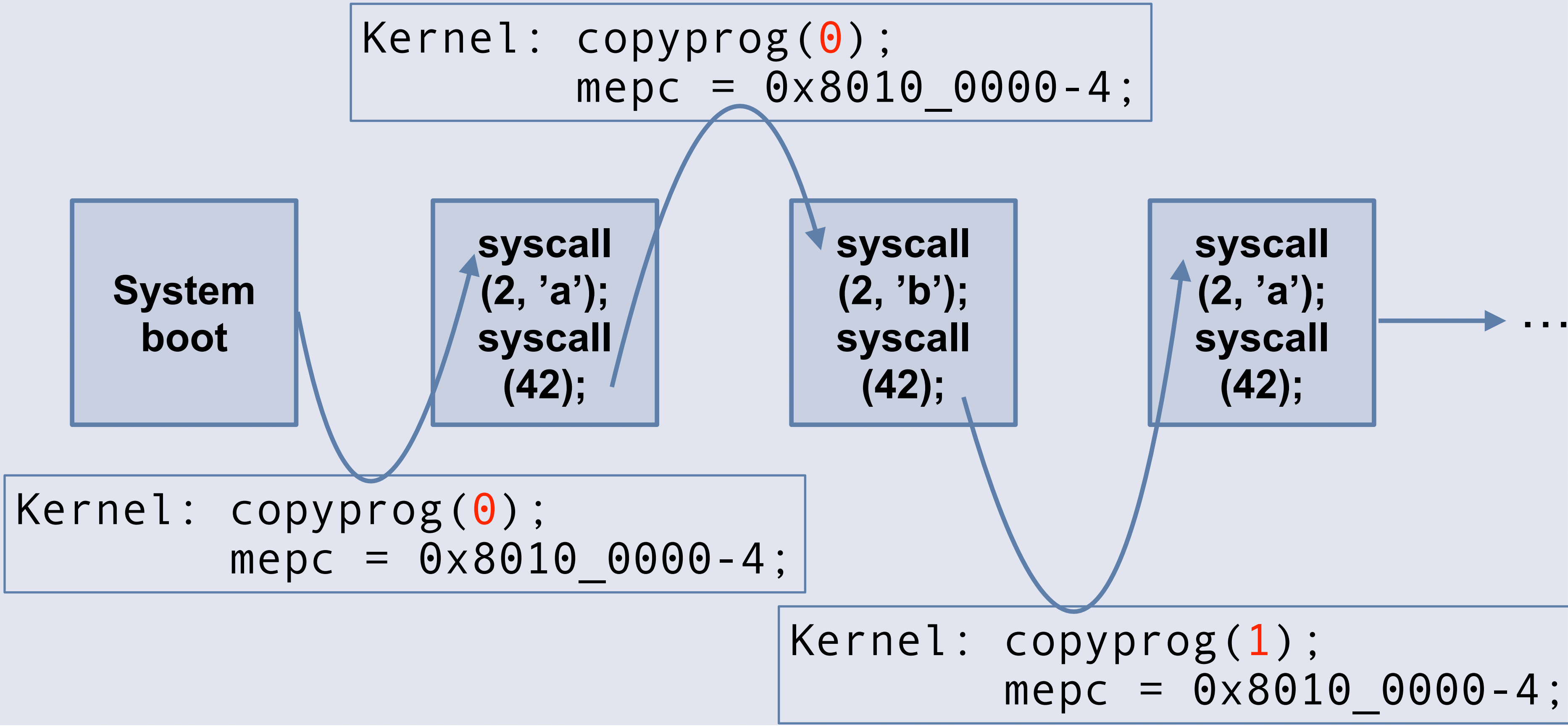
I wanted to have two independent threads. One thread would read from the modem and then display on the screen. The other thread would read from the keyboard and write out to the modem. And there would be two pipes going both ways. This is called task-switching, and a 386 had hardware to support this process. I thought it was a cool idea.

My earliest test program was written to use one thread to write the letter A to the screen. The other thread wrote the letter B. (I know, it sounds unimpressive.) And I programmed this to happen a number of times a second. With the timer interrupt, I wrote it so that the screen would fill with AAAAAAAAAA. Then, all of a sudden, it would switch to BBBBBBBBB. It’s a completely useless exercise from any practical standpoint, but it was a good way of showing that my task-switching worked. It took maybe a month to do this because I had to learn everything as I was going along.

Take it easy

Universität Bamberg

- Load programs 1 and 2 into memory at 0x8010_0000 one after the other



Control flow

- The *control flow* of the system switches between the kernel (function `exception`) and the `main` functions of our two programs

kernel.c

```
void exception(void) {
    copyprog(0);
    w_mepc(0x80100000);
    asm("mret");
}
```

user1.c

```
int main(void) {
    syscall(2, 'a');
    syscall(42, 0);
}
```

kernel.c

```
void exception(void) {
    // syscall nr = 2
    putachar((char)param);
    w_mepc(pc+4);
    asm("mret");
}
```

kernel.c

```
void exception(void) {
    // syscall nr = 42
    copyprog(1);
    w_mepc(0x80100000);
    asm("mret");
}
```

Control flow

Universität Bamberg

Ⓐ The kernel copies program **user1** (id = 0) to 0x8010_0000 and starts it by setting `mepc`

Ⓑ **user1** starts at the beginning (main at 0x8010_0000) and calls `syscall(2, 'a');`

Ⓒ The kernel is re-entered (using the `ecall` instruction via `ex.S`) and sees that syscall #2 was requested. It prints the char passed as parameter…

Ⓓ and returns to **user1** ***at the instruction after the ecall,*** which calls `syscall(42, 0);`

Ⓔ The kernel is re-entered and executes code for syscall #42 to copy program **user2**

Ⓕ Now the control flow repeats for program **user2**

Ⓐ

kernel.c

```
void exception(void) {
    copyprog(0);
    w_mepc(0x80100000);
    asm("mret");
}
```

Ⓑ

user1.c

```
int main(void) {
    syscall(2, 'a');
    syscall(42, 0);
}
```

Ⓒ

kernel.c

```
void exception(void) {
    // syscall nr = 2
    putachar((char)param);
    w_mepc(pc+4);
    asm("mret");
}
```

Ⓓ

Ⓔ

user1.c

```
void exception(void) {
    // syscall nr = 42
    copyprog(1);
    w_mepc(0x80100000);
    asm("mret");
}
```

Ⓕ

Wait, you said it happens at "`ecall`"!

- The *actual* switch from the user program to the kernel and back still happens in the syscall function (implemented in user mode):

identical code for user1.c and user2.c:

```
uint64 syscall(uint64 nr, uint64 param) {
    uint64 retval;
    asm volatile("mv a7, %0" : : "r" (nr));
    asm volatile("mv a0, %0" : : "r" (param));
    asm volatile("ecall");
    asm volatile("mv %0, a0" : "=r" (retval) );
    return retval;
}
```

m_epc

m_epc+4

kernel.c

```
void exception(void) {
    // decode syscall nr
    …m_epc + 4…
    asm("mret");
}
```

Chickens and eggs, again

Universität Bamberg

- All process switching is done in function `exception`
  - **How is the very first process started?**

- We need special handling here
  - The first process is currently "manually" started at the end of `setup`, as before:

```
void setup(void) {
  …
  copyprog(0);
  // set M Exception Program Counter to main, for mret
  w_mepc((uint64)0x80100000);

  // switch to user mode (configured in mstatus)
  // and jump to address in mepc CSR -> main().
  asm volatile("mret");
}
```

How does `copyprog` work?

Universität Bamberg

- It's simply a self-written version of memcpy that copies either from the `user1_bin` or the `user2_bin` array:

`copyprog` in `setup.c`

```
void copyprog(int process) {
  unsigned char* from;
  int user_bin_len;
  switch (process) {
    case 0: from = (unsigned char *)&user1_bin;
            user_bin_len = user1_bin_len; break;
    case 1: from = (unsigned char *)&user2_bin;
            user_bin_len = user2_bin_len; break;
    default: printstring("unknown process!\n");
             printhex(process); printstring("\n"); break;
  }

  unsigned char* to = (unsigned char *)0x80100000;
  for (int i=0; i<user_bin_len; i++) {
    *to++ = *from++;
  }
}
```

But does it work?

Universität Bamberg

- Yes, but…

```
$ qemu-system-riscv64 -nographic -machine virt -smp 1
-bios none -kernel kernel
abababababababababababababababababababababababbunknown
process!
0xffffffff8010106b

bbbababababababababababababababababababababaIn
exception process = 0x0000000000000001
 mcause = 0x0000000000000002
 mepc = 0x00000000801000fa
 mtval = 0x0000000000000000
```

…the system crashes after a number of process switches
(error message printing was added to exception.c)

- **Why?**

Let's enable PMP

- An effect like the one we see here is often an indication of ***memory corruption***, reasons for which could be:
  - a pointer is set to an incorrect value and dereferenced
  - a data structure extends to areas used by other code or data

- We can use **physical memory protection** in `setup` to check for problems:
  - Protect kernel memory against accesses from user mode

function `setup` in `setup.c`:

```
  w_pmpaddr0(0x80000000ull >> 2); // 0x0000_0000...0x7fff_ffff
  w_pmpaddr1(0x80100000ull >> 2); // 0x8000_0000...0x800f_ffff
  w_pmpaddr2(0xffffffffull >> 2); // 0x8010_0000...0xffff_ffff
  w_pmpcfg0(0x0f0000);            // full access for region 2
                                  // no access for regions 0,1
```

Crashing right away…

Universität Bamberg

After enabling PMP, program user1 crashes almost immediately:

```
$ qemu-system-riscv64 -nographic -machine virt -smp 1
  -bios none -kernel kernel
In exception process = 0x0000000000000000
 mcause = 0x0000000000000007
 mepc   = 0x0000000080101038
 mtval  = 0x0000000080004108
```

- Register `mepc` gives us an indication where the system crashed: `0x0000000080101038` is in the space of the user process!
- Can we find out more?
  - `mcause` gives the cause for the exception [2], a **store fault**:

0	7	Store/AMO access fault

  - We can now ***disassemble*** the user program (we know it is user1 since I added code to `exception.c` to print the current process number, which is 0)

Digging deeper…

Universität Bamberg

Let's try to disassemble user1 using the RISC-V `objdump` tool:

```
$ riscv64-unknown-elf-objdump -d user1

0000000080101036 <main>:
    80101036: 1101              addi   sp,sp,-32
    80101038: ec06              sd     ra,24(sp)
    8010103a: e822              sd     s0,16(sp)
    8010103c: 1000              addi   s0,sp,32
    8010103e: fe0407a3          sb     zero,-17(s0)
```

"-d" for "disassemble"

Function name and start address

address of the instruction

opcode of the instruction

disassembled human-readable instruction

The instruction at address `0x0000000080101038` is
`sd    ra,24(sp)`

The crashing instruction

The instruction at address `0x0000000080101038` is:

`sd ra,24(sp)`

- This instruction **stores** (`s`) a **double** value (`d`, 64 bit) which is in **register** `ra` into **memory at address** `24(sp)` **= (value of** `sp`**)**+24
  - Remember that the processor indicated a **store fault**
  - This `sd` instruction is part of the ***function prologue*** to save the return address, see lecture 3
- So ***writing to the stack fails***
  - Where is our stack right now? ***Look at*** `mtval`**:**
    `mtval = 0x0000000080004108`
  - This is the address which the processor tried to store (write) to…
    - ***It is in kernel space!***

Dar(li)n(g), I forgot about the stack!

Universität Bamberg

- We currently only have one processor stack
  - initialized at the very start in `boot.S` before jumping to setup:

boot.S

```
_entry:
        la      sp, kernelstack
        li      a0, 4096
        add     sp, sp, a0

        jal     setup
```

  - …and defined in `kernel.c` – so it's in kernel address space!

kernel.c

```
__attribute__ ((aligned (16))) char kernelstack[4096];
```

Solution

- Give each process its own stack, set up in `userentry.S` (this is identical for `user1` and `user2`)

`userentry.S`

```
_entry:
        la      sp, stack0
        li      a0, 4096
        add     sp, sp, a0

        jal     main
```

- Define a separate stack array in `user1` and `user2`, respectively:

identical code for user1.c and user2.c:

```
__attribute__ ((aligned (16))) char stack0[4096];
```

Next step: cooperative multitasking

Universität Bamberg

- We can switch processes now!
  - But each of the processes has to run to completion
- Can we switch processes ***while they are still running?***
- ***New system call:*** `yield()`
  - Give up control of the processor
  - Eventually return to instruction after yield when activated again

Problems:

- We now need to remember where to return to in a yielded process
  - Before, our processes ran to completion and started from main
- Both processes need to be in memory at the same time now…
  - Separate linker script for user1/2 and fixed address spaces
  - Optional: ***position-independent code*** *(relative addressing)*

Conclusion

Universität Bamberg

- We can switch processes now!
  - Per-process stacks required to completely separate the address spaces
  - All "magic" happens in the exception handler
    - Responsible not only for handling system calls, but also catches all other exceptions such as our store fault
  - PMP can be used to find (some) problems like this

- Cooperative multitasking uses a `yield()` system call to switch between processes ***without terminating the calling process***
    - This requires more saving of state
    - Demo and discussion in the lab session!

References

1. Linus Torvalds and David Diamond, *Just for Fun – the Story of an Accidental Revolutionary*, Harper 2001, ISBN 0-06-662073-2
2. Andrew Waterman, Krste Asanovic and John Hauser,
   The RISC-V Instruction Set Manual Volume II:
   Privileged Architecture Document, Version 20211203,
   Table 3.6: Machine cause register (mcause) values after trap