Betriebssysteme

Vorlesung im Herbstsemester 2008
Universität Mannheim

Kapitel 5b: Symmetric Multiprocessing in ULIX

Felix C. Freiling

Lehrstuhl für Praktische Informatik 1

Universität Mannheim

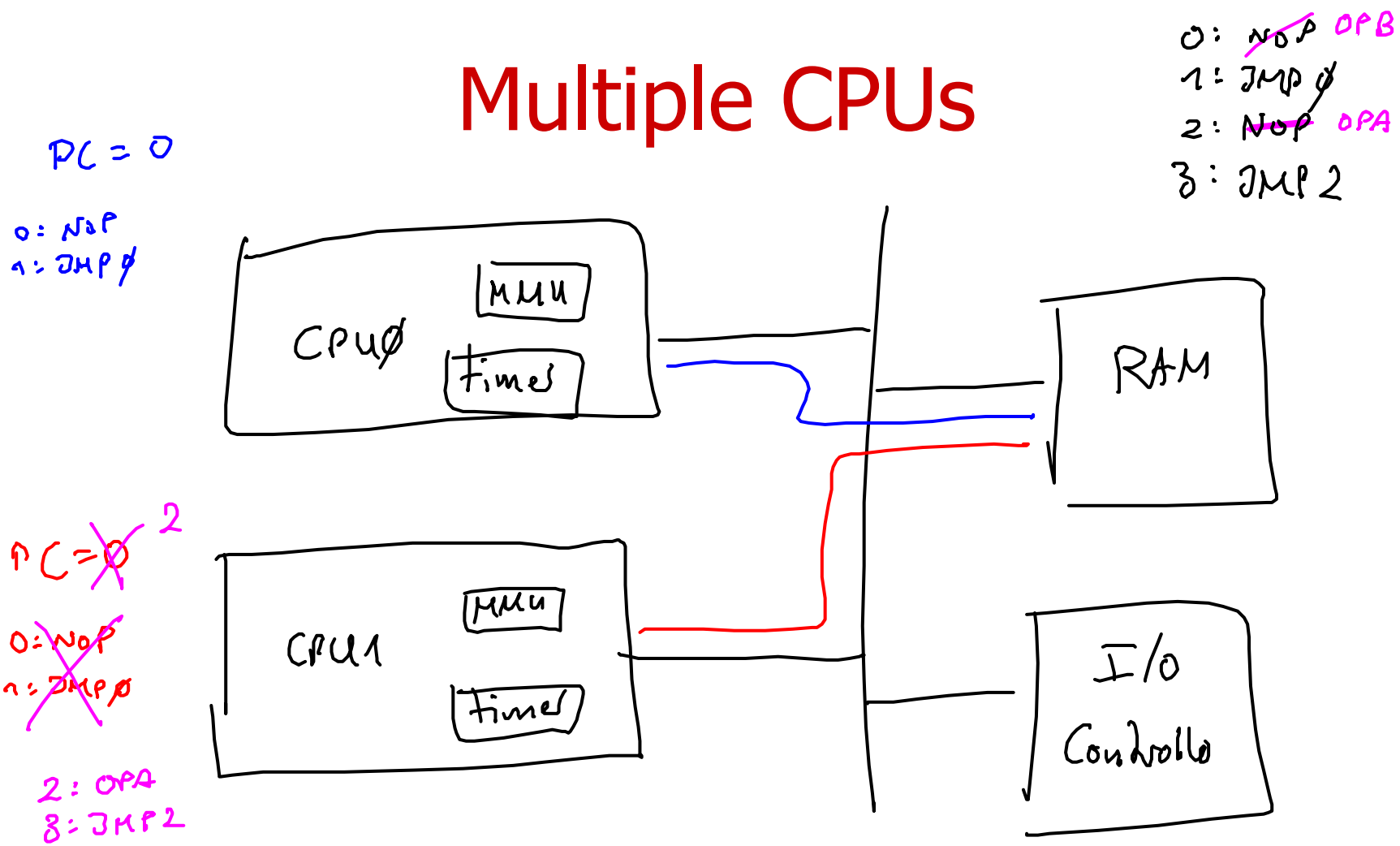
(Diese Folien sind nicht Teil des Basiskurses)

Overview

- Symmetric multiprocessing and load sharing

- Running programs on multicomputers
- Separating control flows
- Using explicit CPUID
- Multiprocessing without CPUIDs

Multiple CPUs

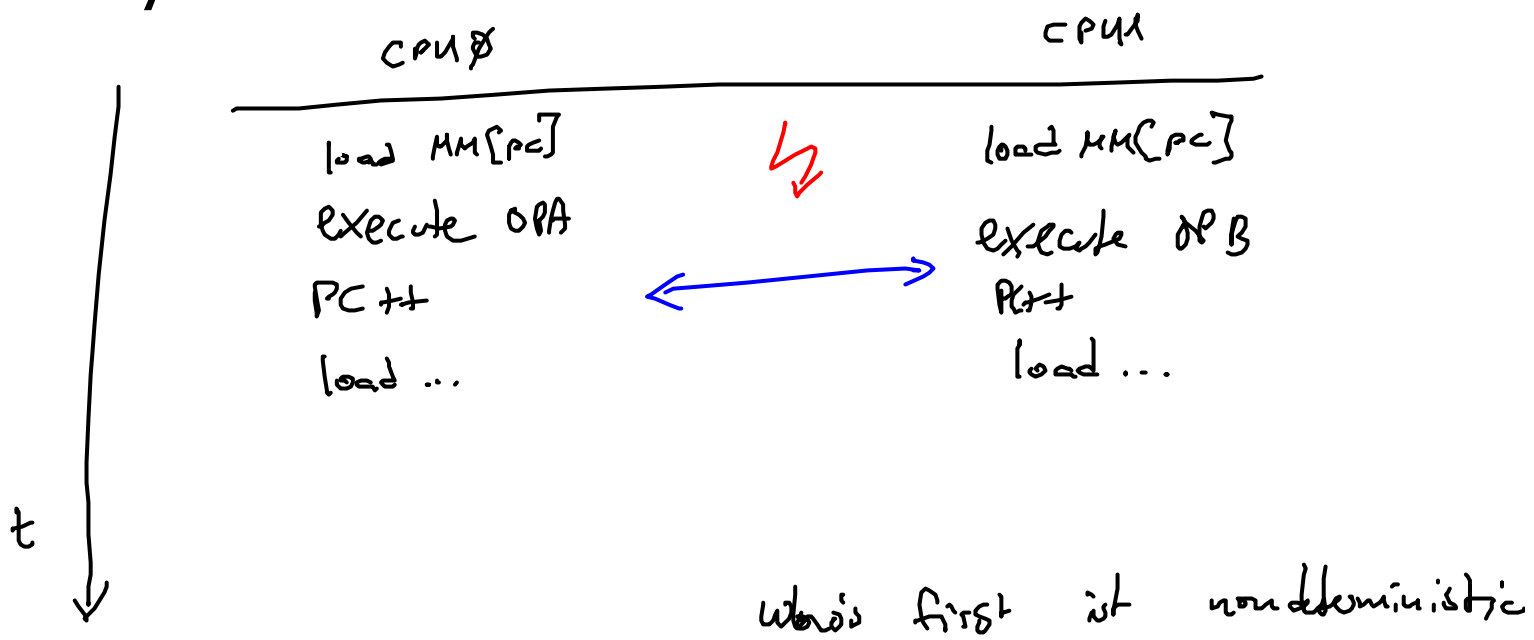


Details

- Two CPUs, both have their own registers etc.
  - In particular they have their own PC
  - They execute their own instruction cycle (concurrently)
  - Use the same memory
- Memory:
  - 0: OPA; 1: JMP 0; 2: OPB; 3: JMP 0;
- Assume initially PC1=PC2=0
  - Then both CPUs execute the same program
- Assume initially PC1=0 and PC2=2
  - Both CPUs execute different programs

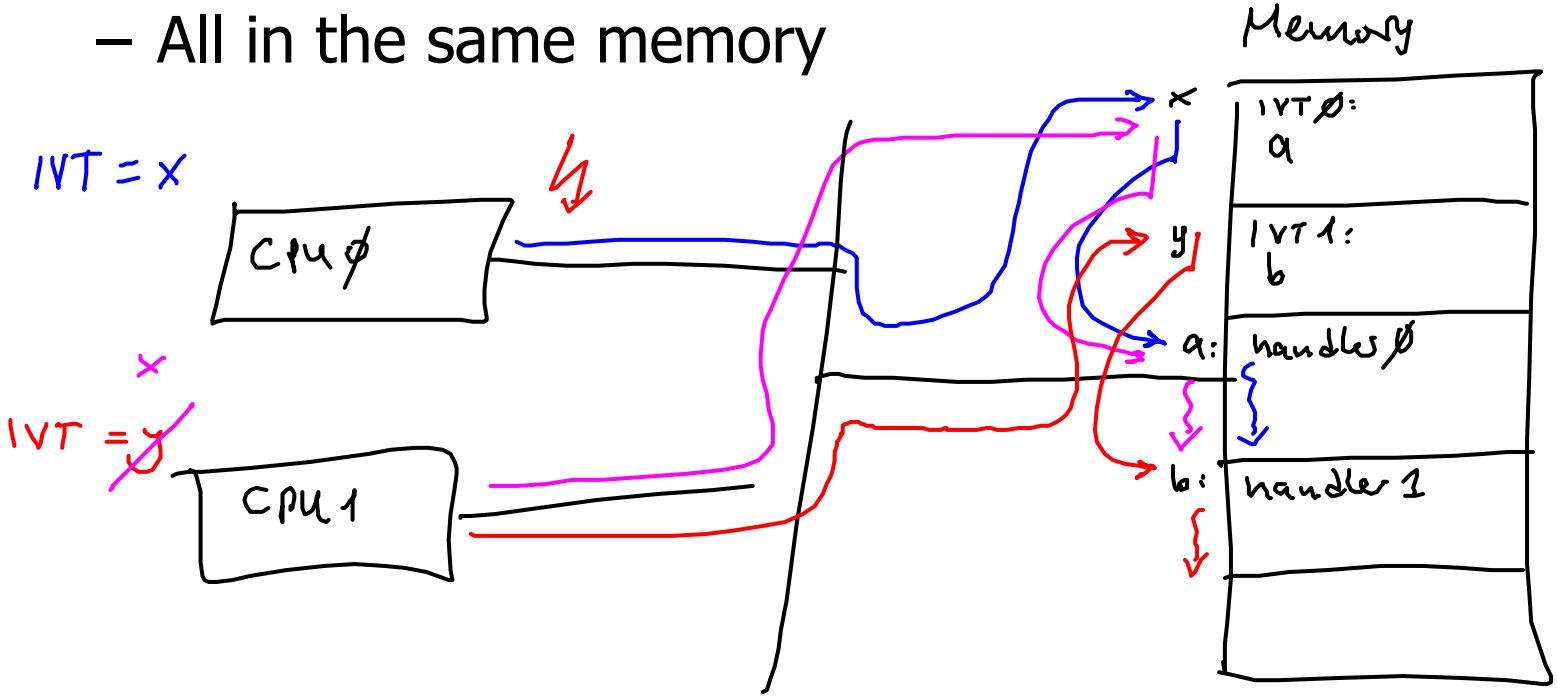
Real Concurrency

- Conflicting access to memory are arbitrated by hardware



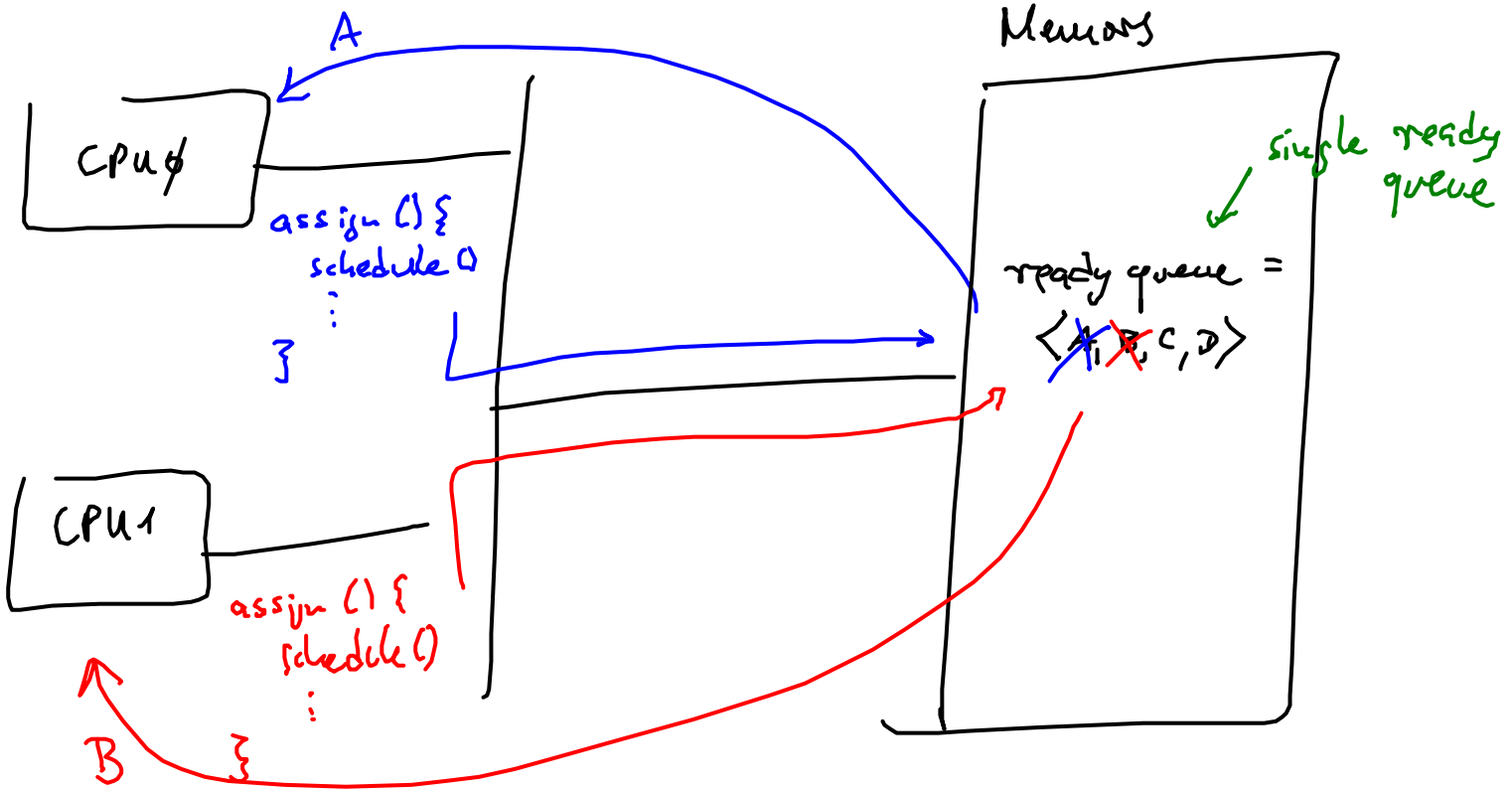
Interrupts

- Every CPU has its own IVT register
  - Every CPU can have "own" interrupt vectors
  - Every CPU can have "own" interrupt handlers
  - All in the same memory



Global Ready Queue

- Symmetric multiprocessing with load sharing



Protection of Global List

- We need to protect the global list from concurrent accesses
  - Single load/store accesses are usually no problem, but complex list manipulations need to be executed atomically
  - See Chapter 6
- Assume for now: List manipulations are atomic

Who is Running Where?

```
⟨kernel global variables 108c⟩+≡
  thread_id running[NUM_CPUS];
```

- If CPU0 wants to know which thread it is executing, it inspects running[0]
- ...
- If CPUn wants to know which thread it is executing, it inspects running[n]

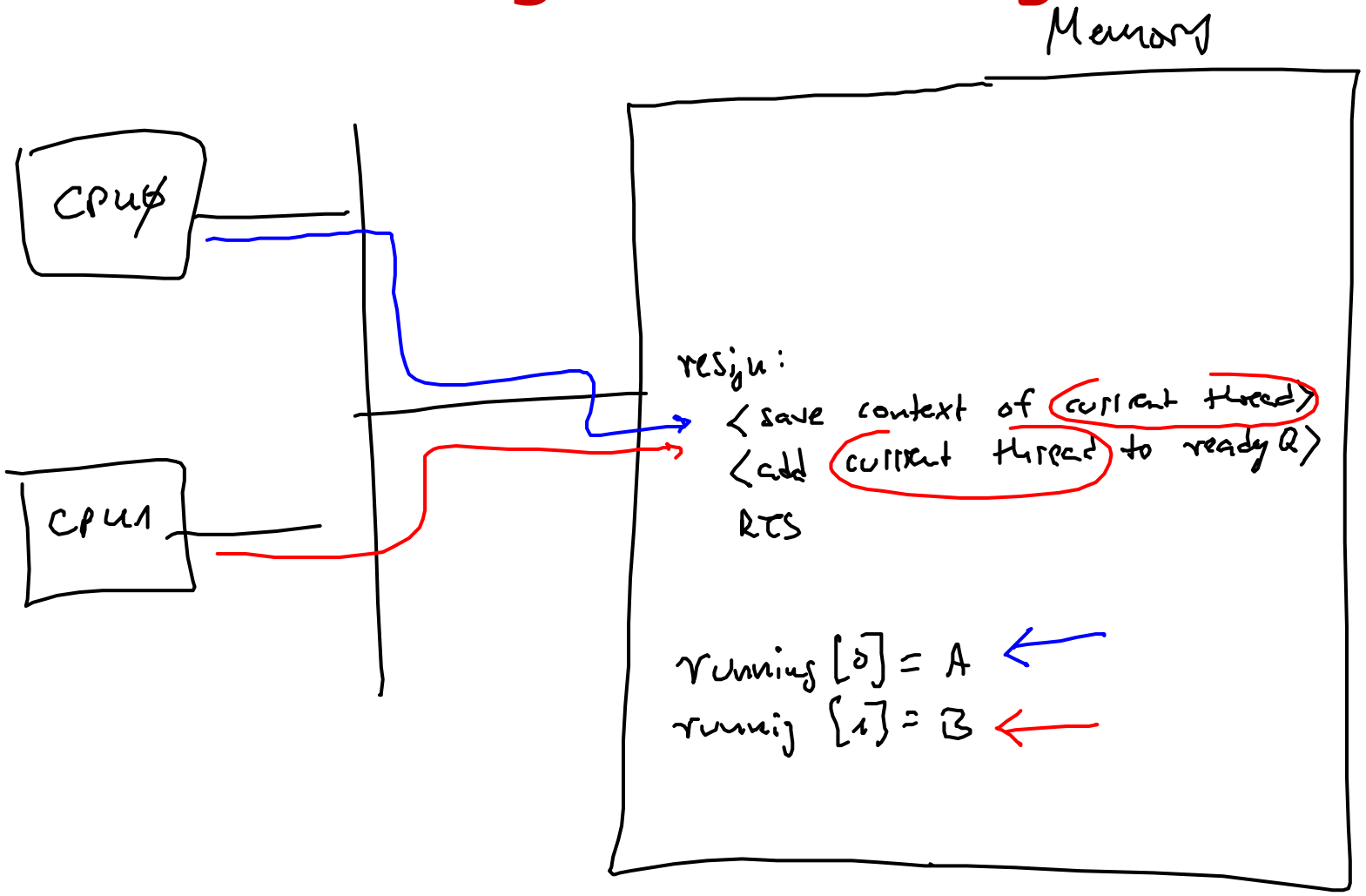
Resign for CPU 0

- Taken from last lecture:

```
⟨kernel functions 110a⟩+≡
  void resign() {
    ⟨save processor context of running thread 132b⟩
    add_to_ready_queue(running[0]);
  }
```

- Only works for monoprocessor system
  - **`running[0]`** hard coded
- How do you program **`resign`** for multiprocessor?

Using One `resign`



Who am I?

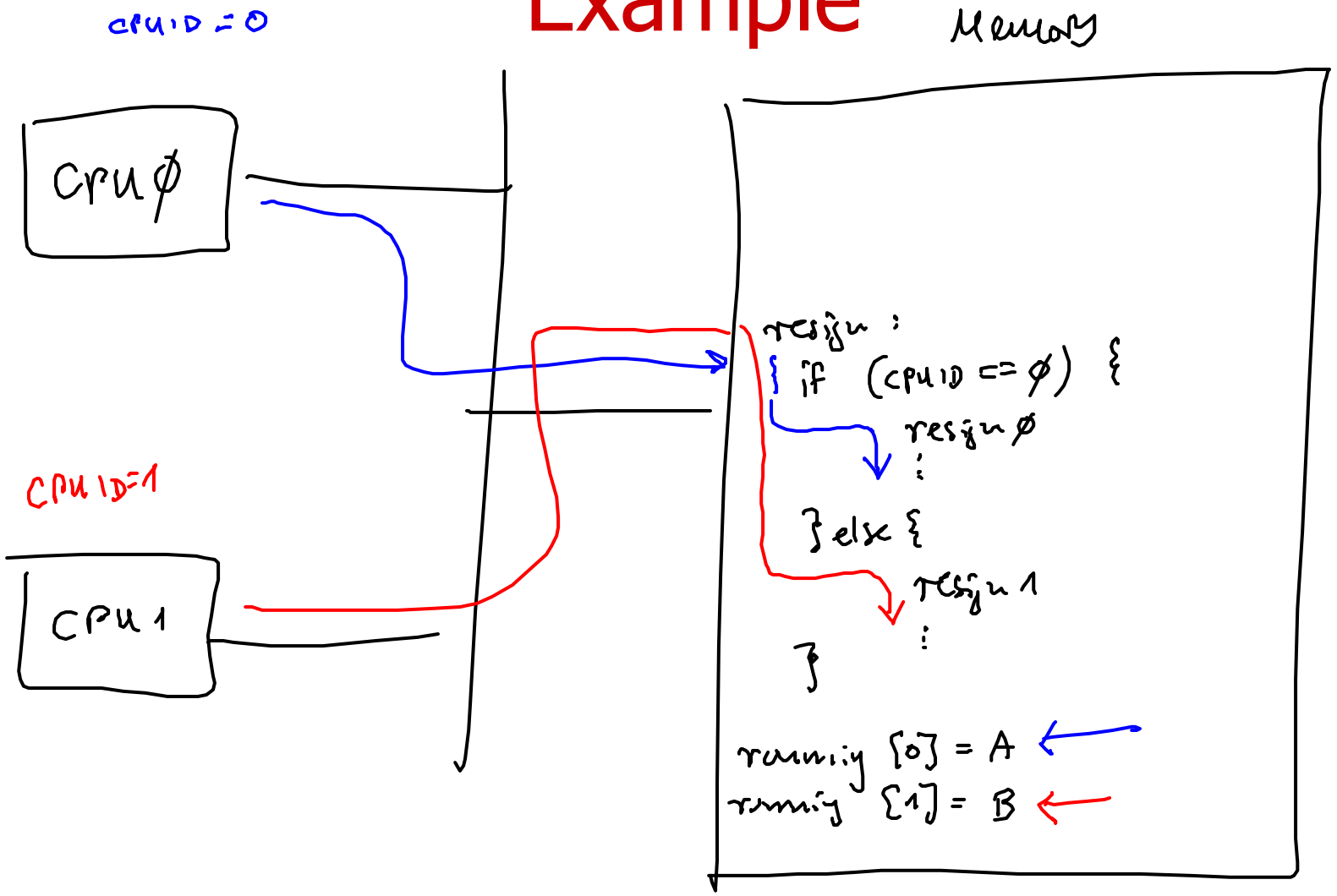
- All CPUs are the same (symmetric)
- All CPUs use the same memory
- How can any CPU know its ID?
  - Am I CPU0 or CPU1?

- First solution: Assume a special hardware register CPUID
  - Hardcoded during manufacturing of computer
  - CPUID at CPU0 is 0, at CPU1 is 1, etc.

Resign with CPUID

```
resign() {
  if (CPUID == 0) {
    // run resign for CPU0
    <store processor context of running[0]>
    <add running[0] to ready queue>
  } else {
    // run resign for CPU1
    <store processor context of running[1]>
    <add running[1] to ready queue>
  }
}
```

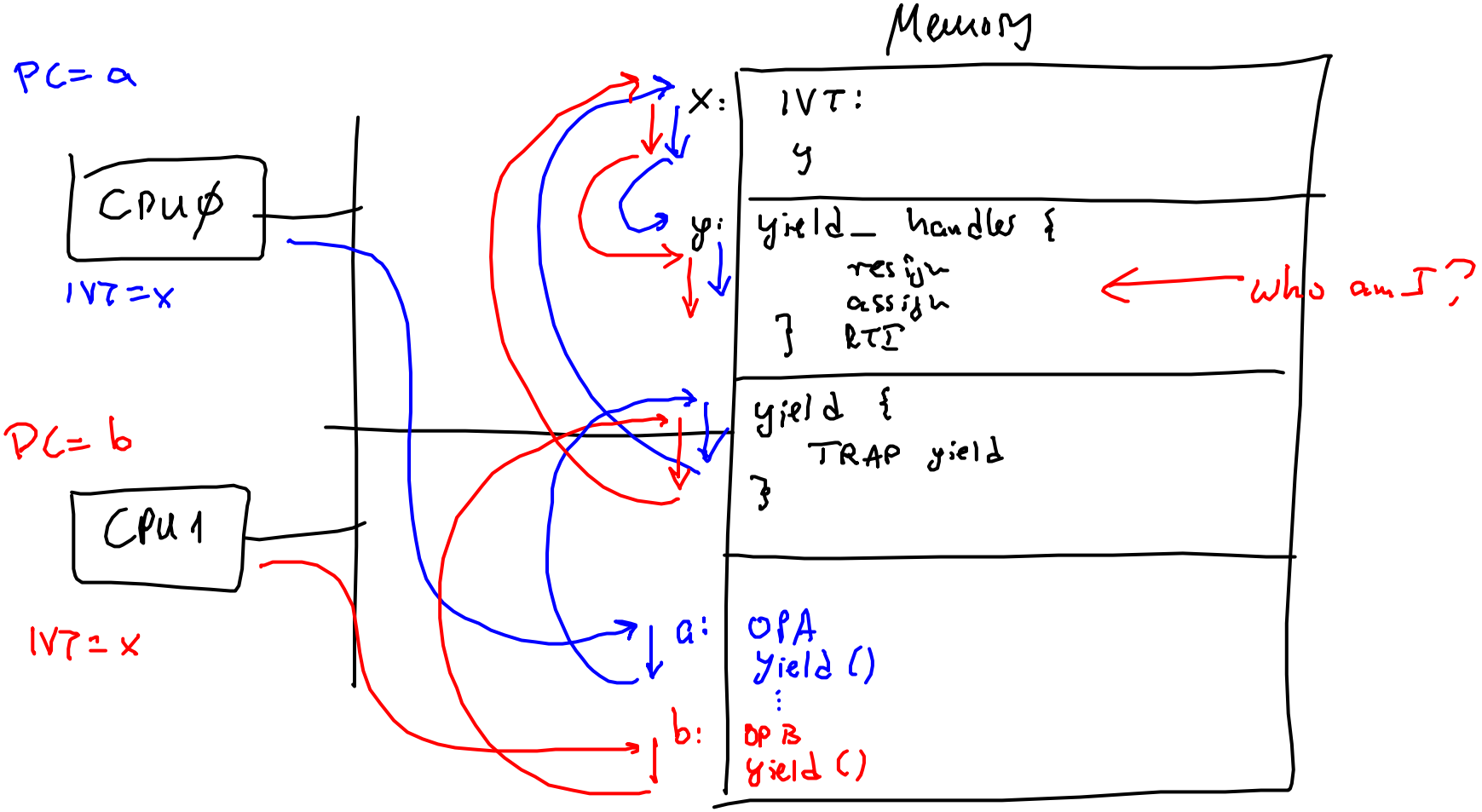
Example


cpuID = 0
Memory
CPU 0
CPU 1
cpuID=1
resign :
{ if (cpuID == 0) {
resign0
:
} else {
resign1
:
}
running [0] = A
running [1] = B

CPUID needed?

- We can implement the same functionality without a register like CPUID
- Idea: run "separate programs" on all CPUs
  - Remember slide 3
  - Knowledge of id of CPU is implicitly contained within control flow
- Important: Control flows are not allowed to merge!

Example: TRAP

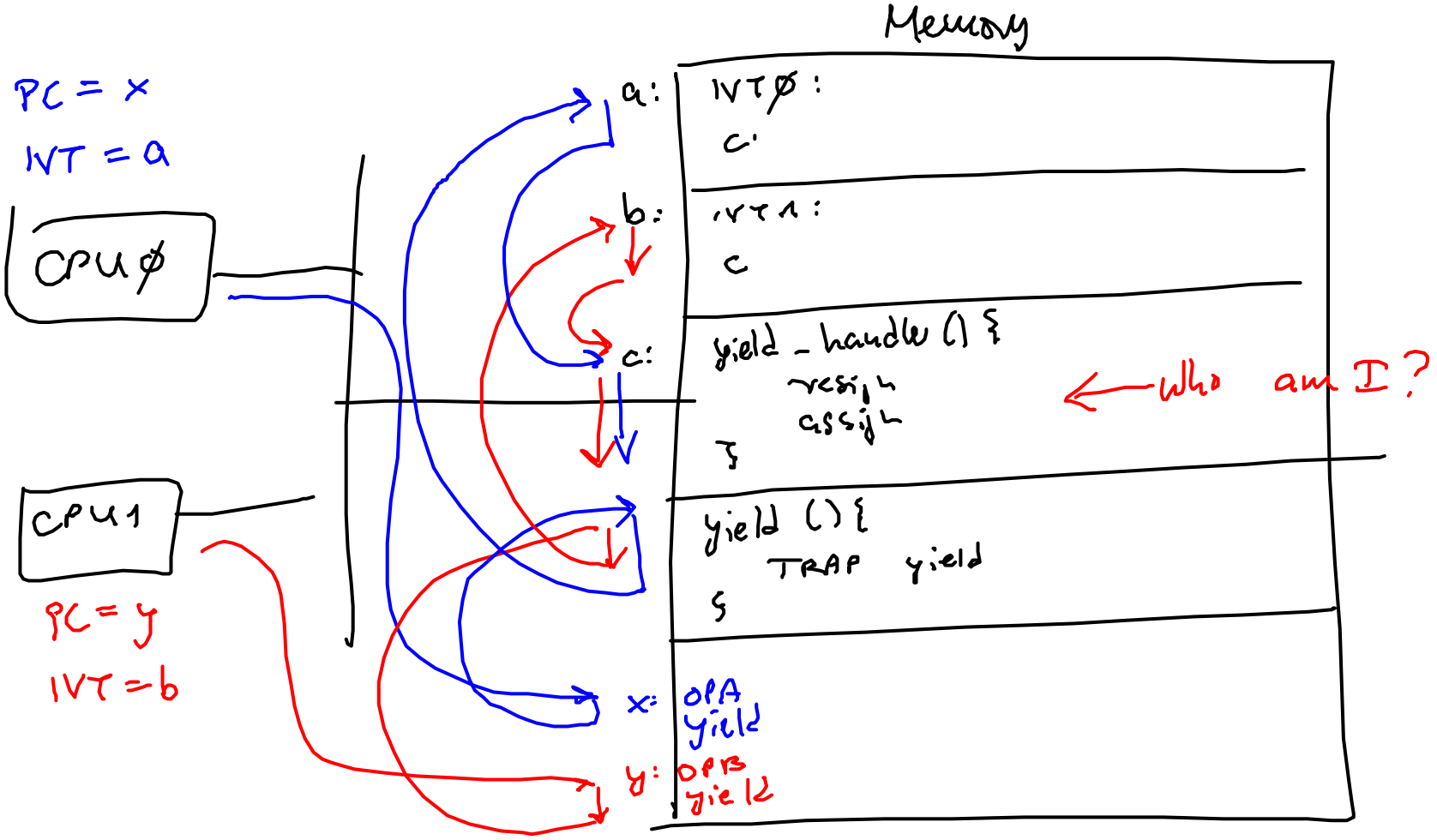


Details

- System call yield implemented via TRAP in local function call (library)
- Use same interrupt vector table
  - => use same interrupt handler
  - => control flows merge
  - => need CPUID to distinguish again!

- Idea: Use two different interrupt vector tables

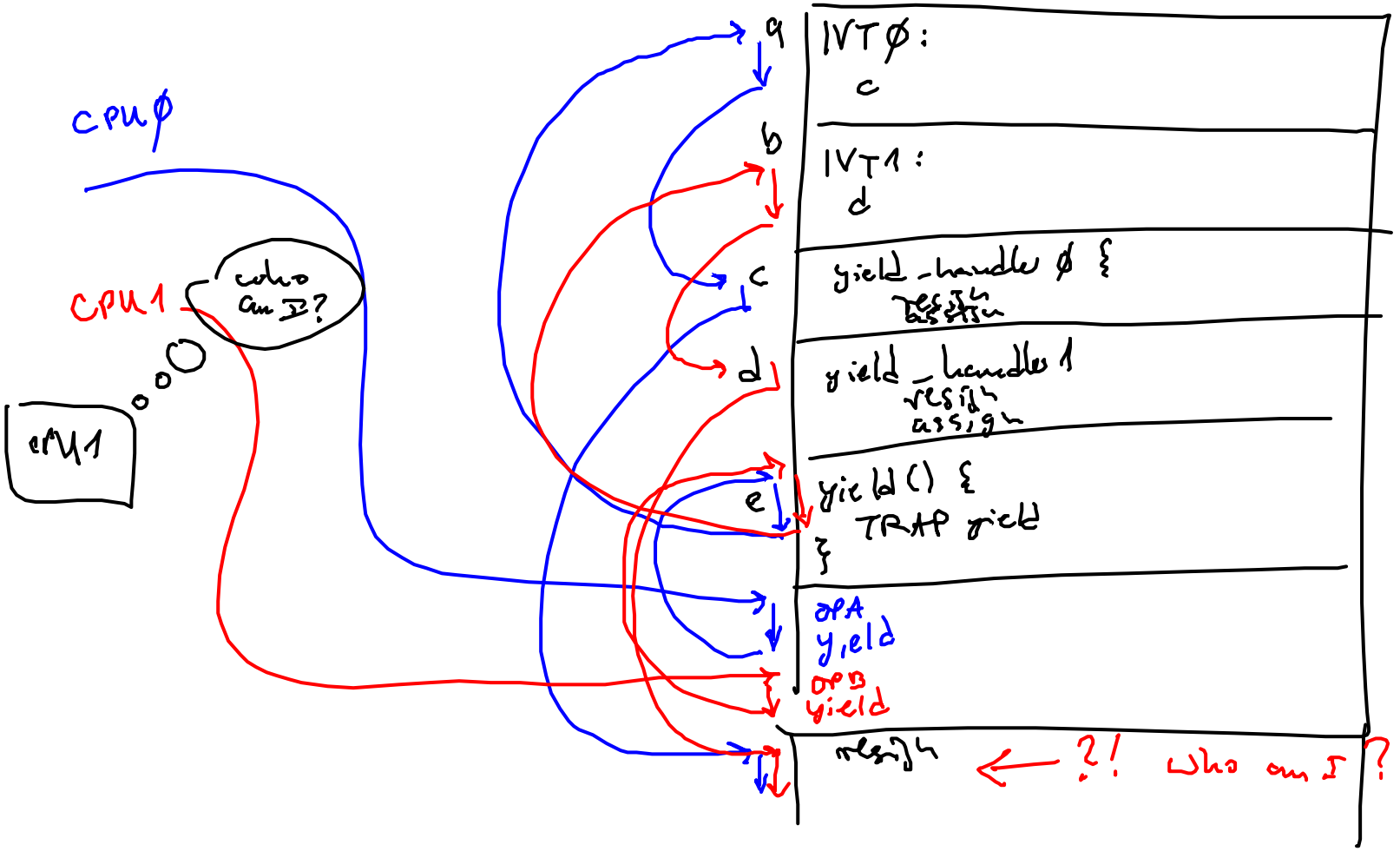
Example: TRAP with two IVTs



Details

- Different interrupt vector tables, but same handler
  - => control flows merge
  - => need CPUID again
- Idea: use separate interrupt handlers

Example:
Separate Interrupt Handlers



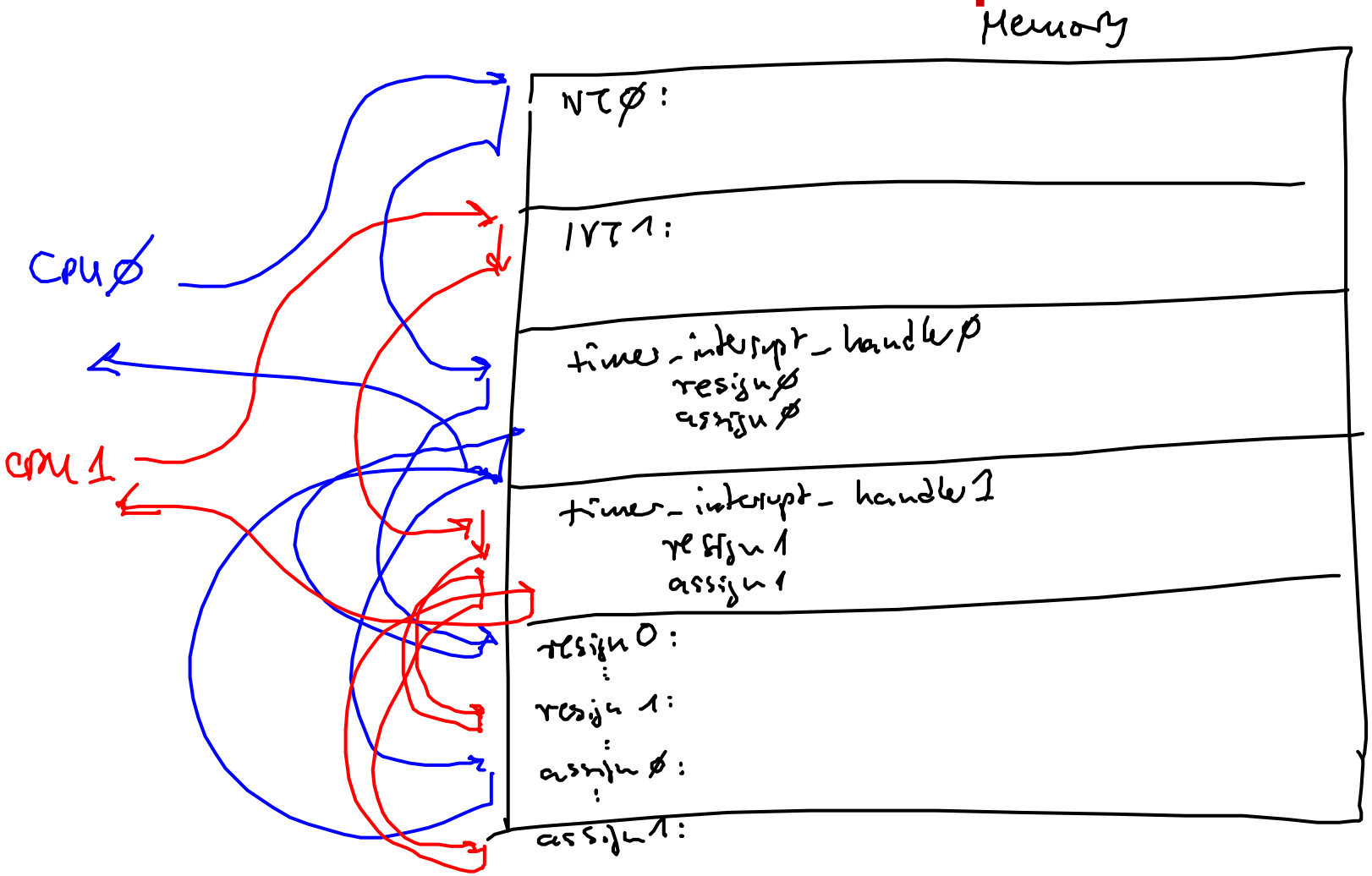
Separate Dispatcher Functions

- For every CPU we need separate
  - interrupt vector table
  - interrupt handlers
  - dispatcher functions assign, resign, block
- Take care to always "use the right version" of
  - the interrupt handler
  - the dispatcher function

External Interrupts

- Up to now only synchronous interrupts (TRAP) considered
- Idea works equally for asynchronous interrupts
- Example: timer interrupt
  - Timer unit associated to each CPU
  - Raises interrupt at the associated CPU (and nowhere else)

Timer Interrupt



Simplification

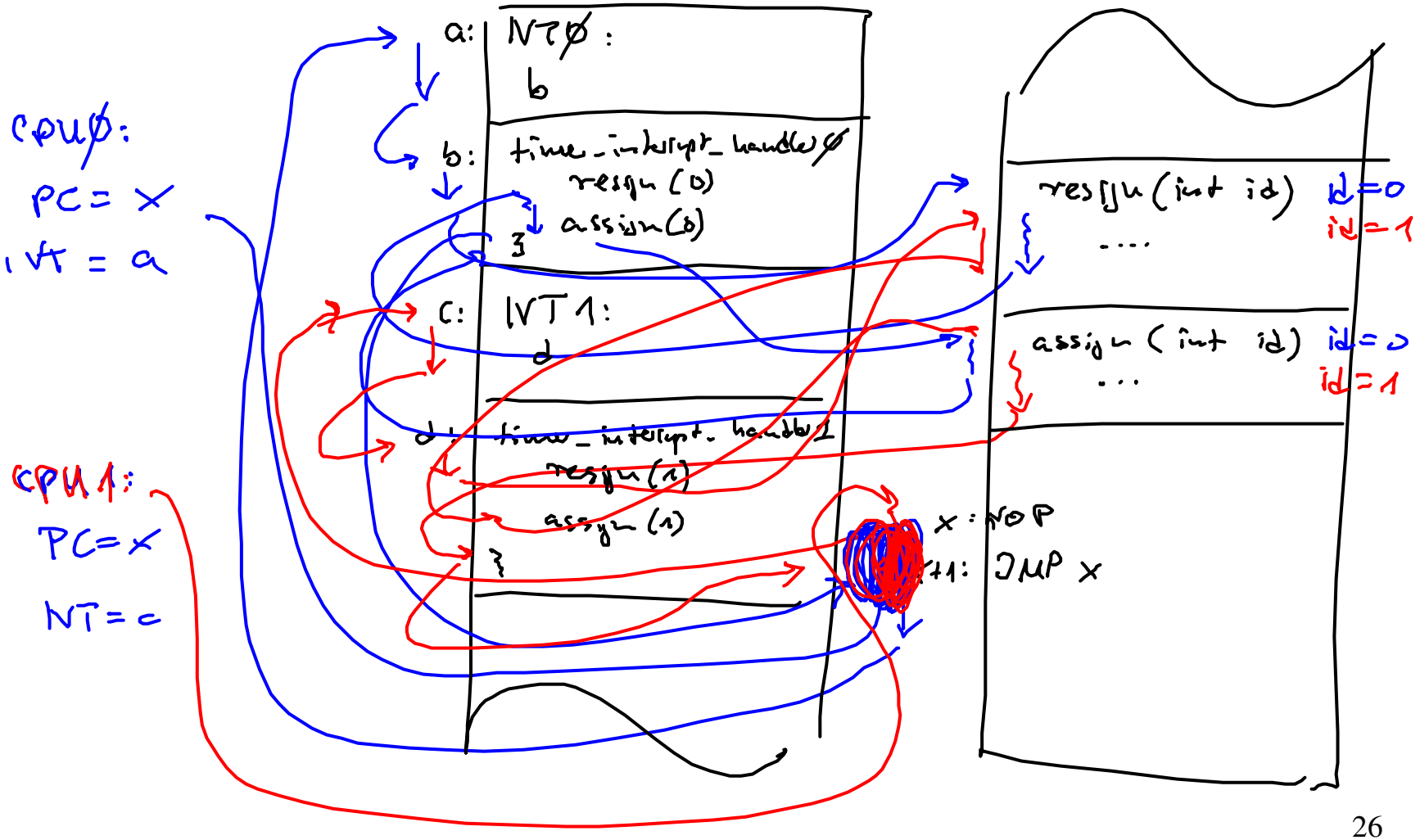
- Use id as parameter for all software artifacts
  - Does not (always) work for interrupt handlers

```
⟨kernel functions 110a⟩+≡
  void resign(int cpuid) {
    ⟨save processor context of running thread 132c⟩
    add_to_ready_queue(running[cpuid]);
  }
```

```
⟨kernel functions 110a⟩+≡
  void assign(int cpuid) {
    running[cpuid] = schedule(); // invoke the scheduler
    ⟨load processor context of running thread 134b⟩
  }
```

Now both CPUs
can even run the same program!

Same Program...



Bootstrapping

- How can two symmetric CPUs separate their control flows initially?
  - Both run same program
- Idea: break symmetry
  - Both CPUs "race" for a global bit
  - Use "test and set" atomic operation (see Chapter 2)
  - The CPU winning the race is CPU0, the loser is CPU1

Outlook

- Easy and elegant symmetric multiprocessing with load sharing
- Needs separate code for every CPU
  - Kernel must be configured at compile time
- Needs synchronization (see Chapter 6)