System Architecture

6 Thread Switching

Yielding, General Switching

November 10 2008

Winter Term 2008/2009

Gerd Liefländer

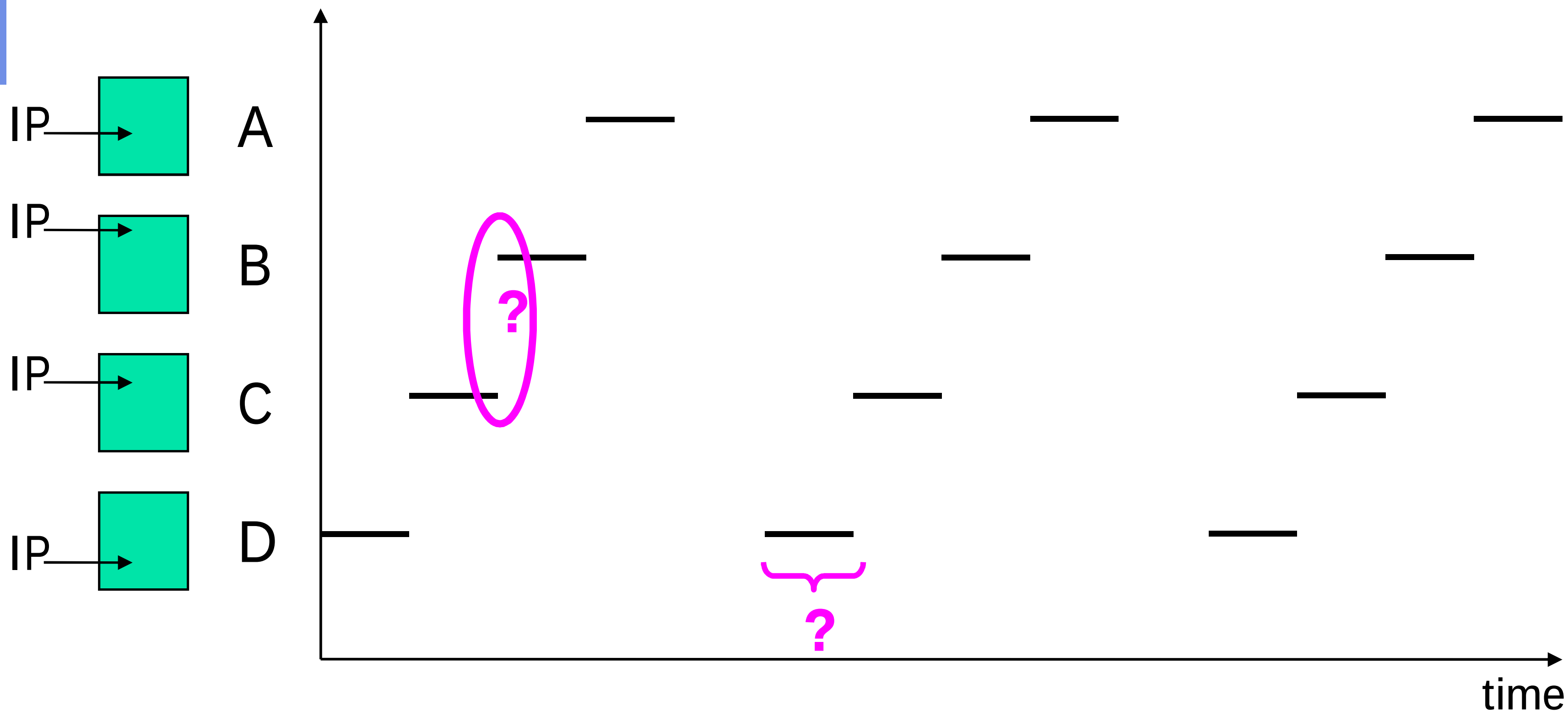
Agenda

- Review & Motivation
- Thread Switching Mechanisms
  - Cooperative PULT Scheduling + Thread Switch
  - Cooperative KLT Scheduling
  - KIT Thread Switch of KLTs
- Additional Design Parameters
  - User and Kernel Stack
  - Idle Thread
  - Initialization/Termination

Review & Motivation

Problems to Solve

- *How to design mechanism* **`thread_switch`**?



- *How to schedule threads, and for how long?*
- *Do we need time slices in every computer?*

Influence of CPU Switching

- CPU switching back and forth among threads:
  - Rate at which a thread performs its computation will not be uniform
  - Nor will it be **reproducible** if the same set of threads will run again
  - Its timing (e.g. waiting times) can depend on other application- or system-activities

⇒ Threads should never be programmed with built-in assumptions about timing

Conclusion

- **Never** accept a solution relying on timing conditions
- If you program portable application don’t rely on
  - specific scheduling policies
  - number of processors
  - ...
- In case, you can rely on a specific platform offering different scheduling policies, try to get the most promising one
- In each system the scheduling policies should be supported by a policy-free dispatching mechanism

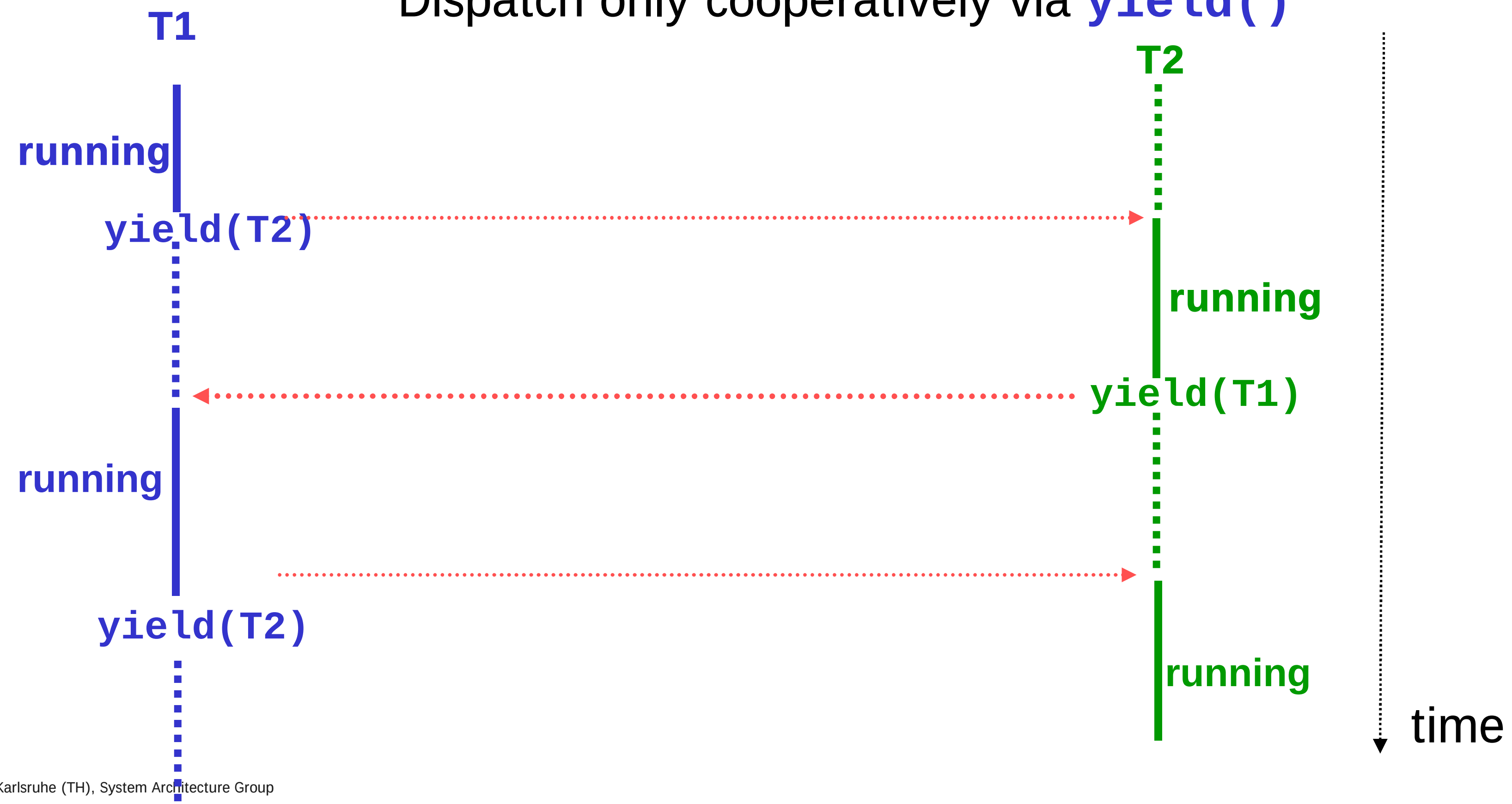
Pult Scheduling

Cooperative Scheduling

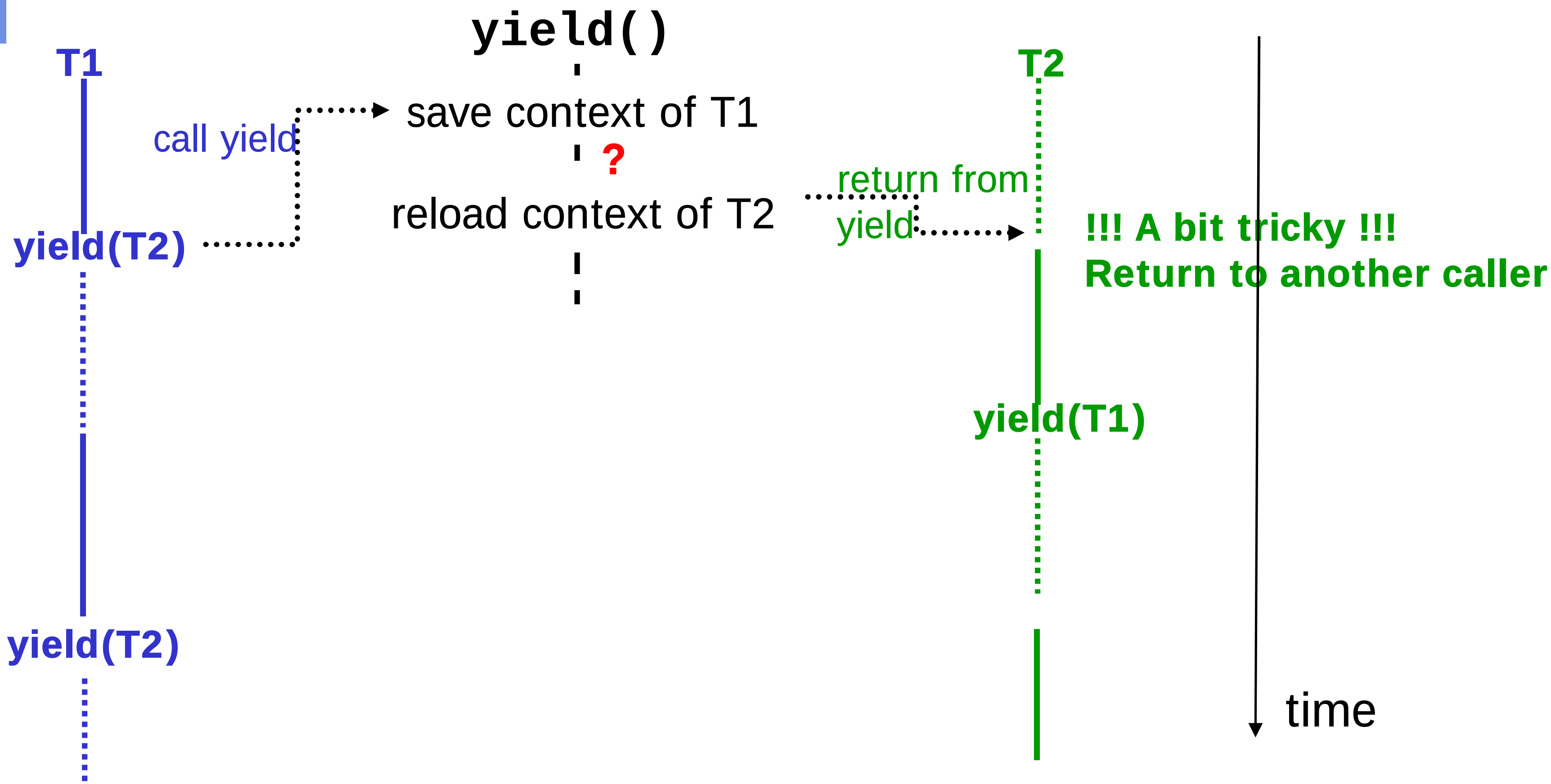
<u>Common research trick</u>:
simplify whenever possible

<u>Assumption:</u> Given 2 pure CPU- bound threads T1 and T2, **one CPU** and **no interference** with a device

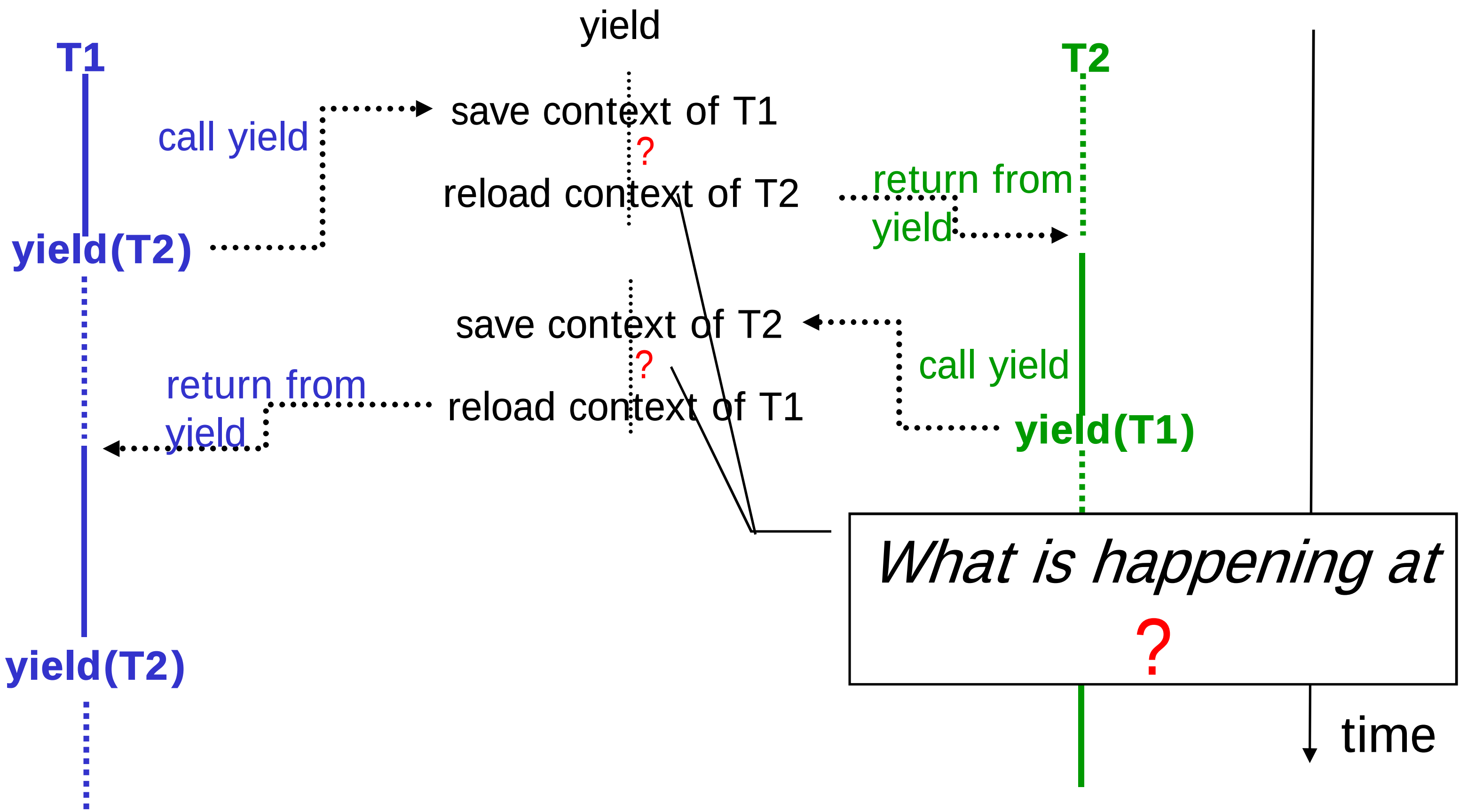
Dispatch only cooperatively via `yield()`



Simplified User Level Yield (1)



Simplified UL-Yield (2)



Simplified UL-Yield (3)

```
procedure yield(NT:thread)
{
…
save context of CT
… ? …
load context of NT
…
return;
}
```

Part of yield still runs under control of caller

How to solve this problem?

Part of yield will run under control of the next thread

Assumption: Both threads T1 and T2 already have called **yield()** once before

Corollary: Each thread gets and gives up control within the procedure **yield** at exactly the **same** (user land) **instruction**

Simplified UL-Yield (4)

```
procedure yield(NT:thread)
{
…
save context of CT
CT.sp := SP; SP := NT.sp;
load context of NT
…
return;
}
```

Part of yield still runs under control of caller

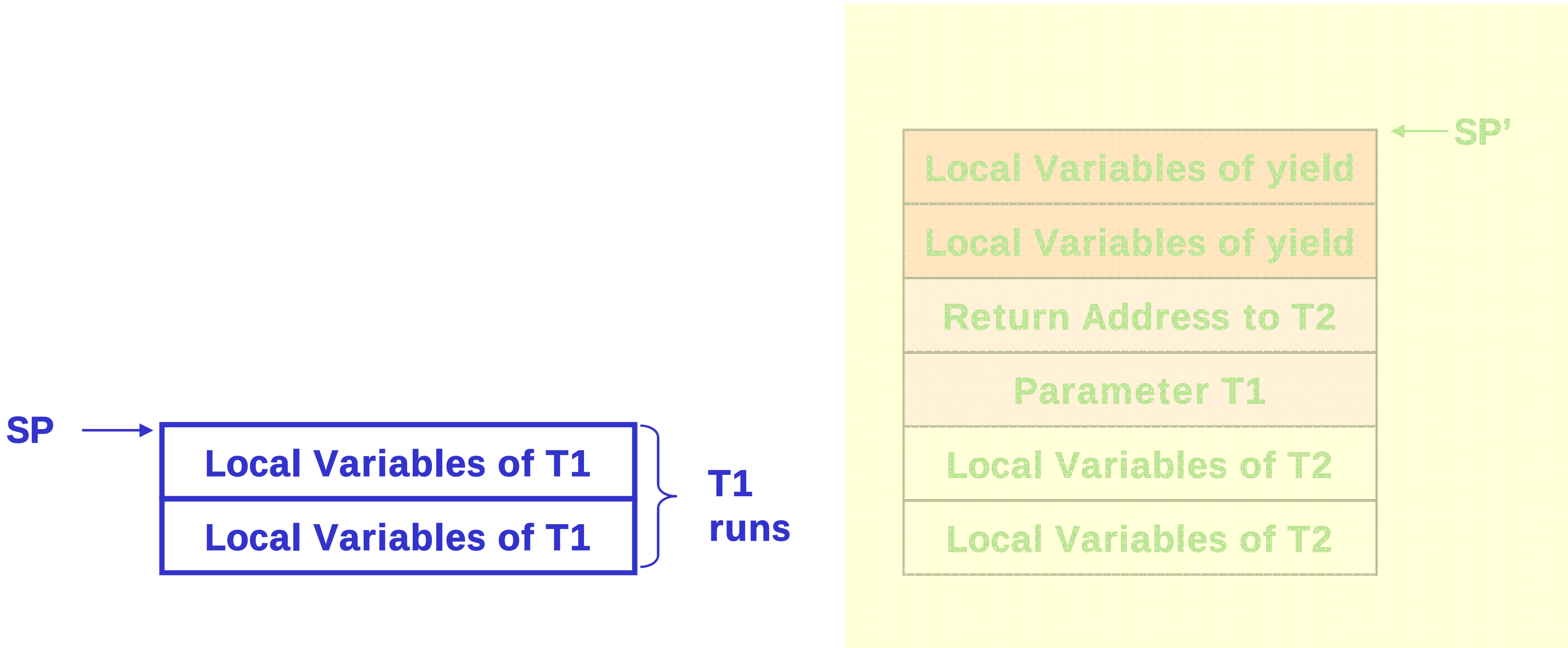
Change stack pointers

Part of yield will run under control of the next thread

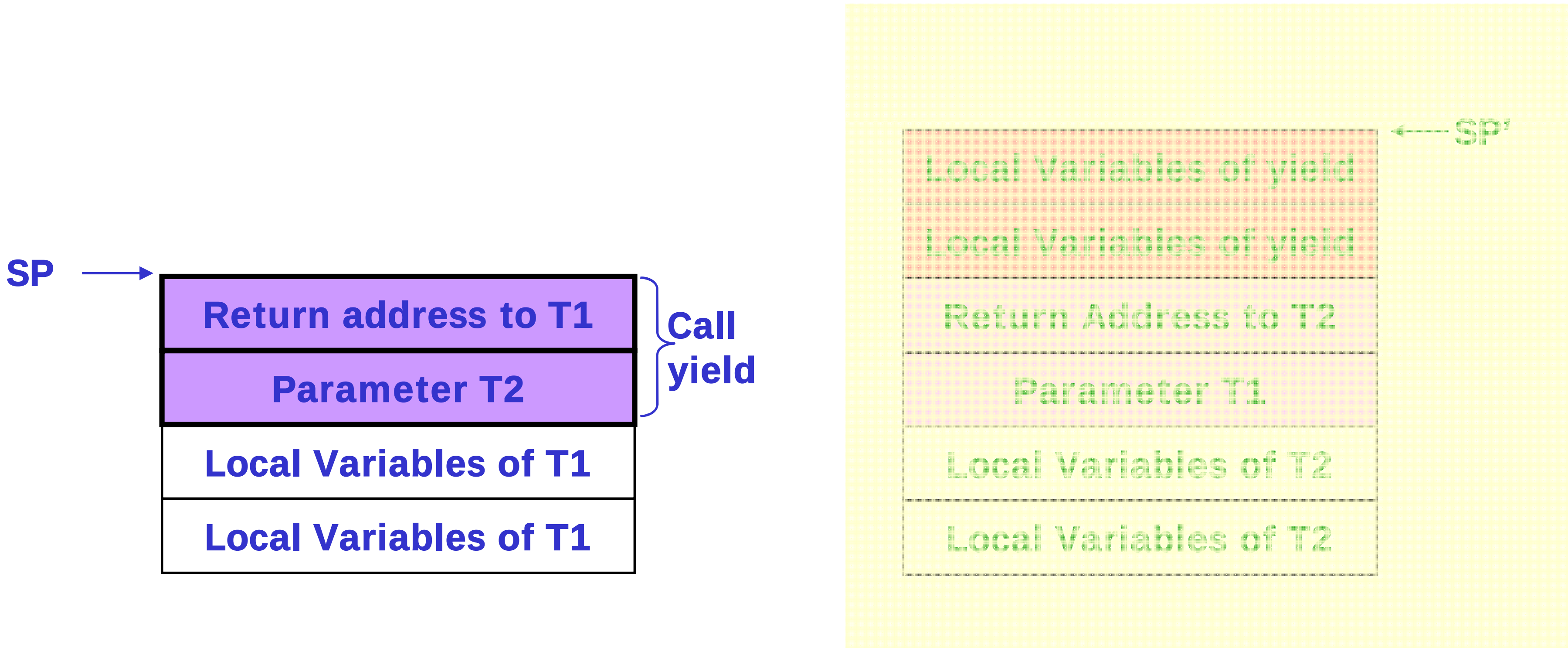
SP = stack pointer **register**

sp = **entry** in TCB

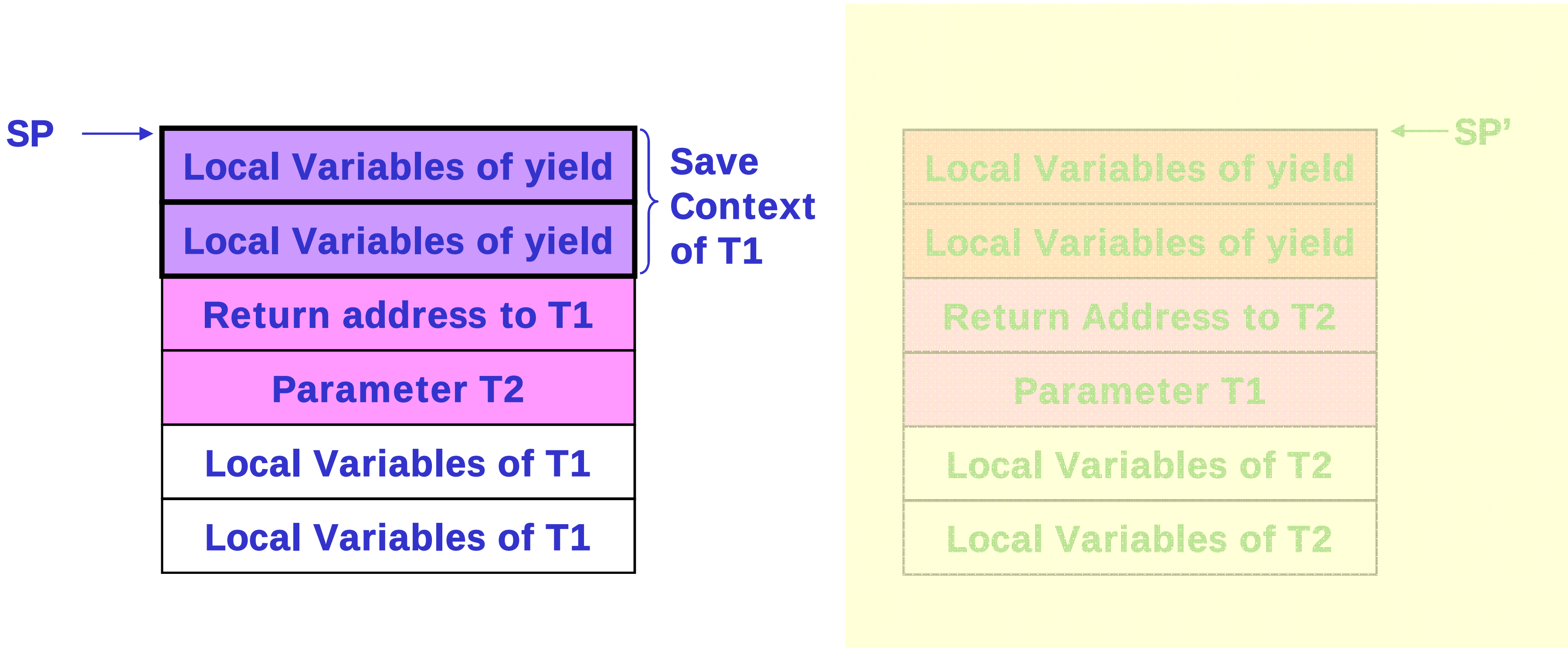
Stack Contents during UL-Yield (1)



Stack Contents during UL-Yield (2)



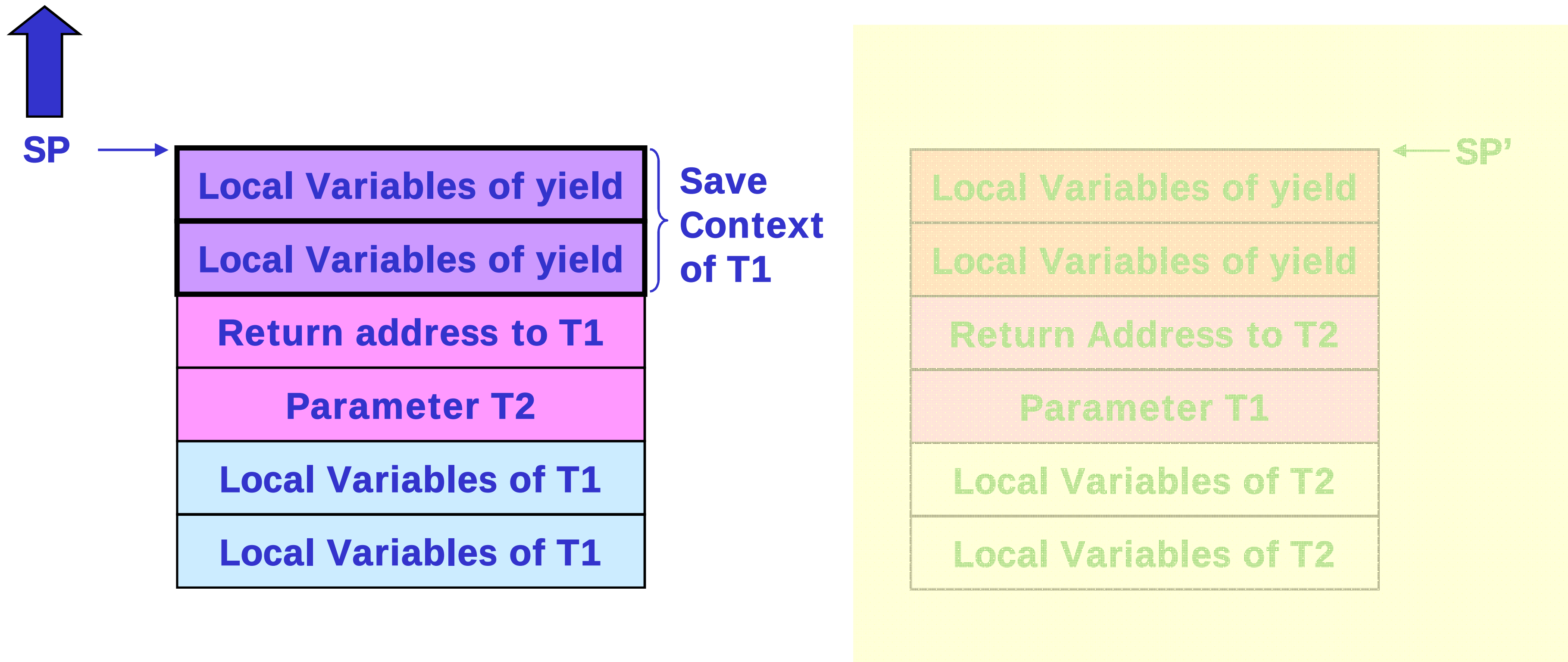
Stack Contents during UL-Yield (3)



Stack Contents during UL-Yield (4)

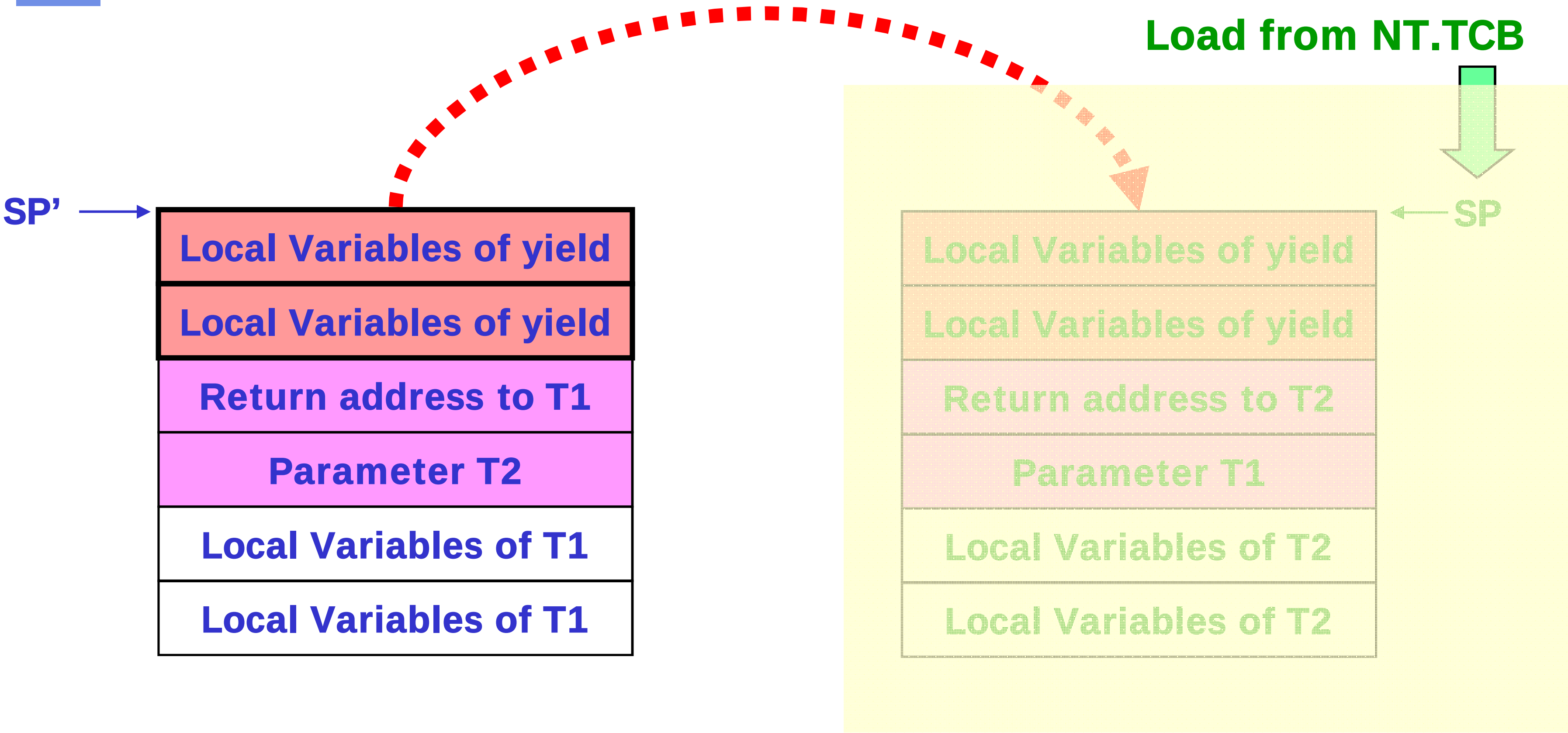
Switch Stack Pointer

Save to current TCB, i.e. to TCB of T1

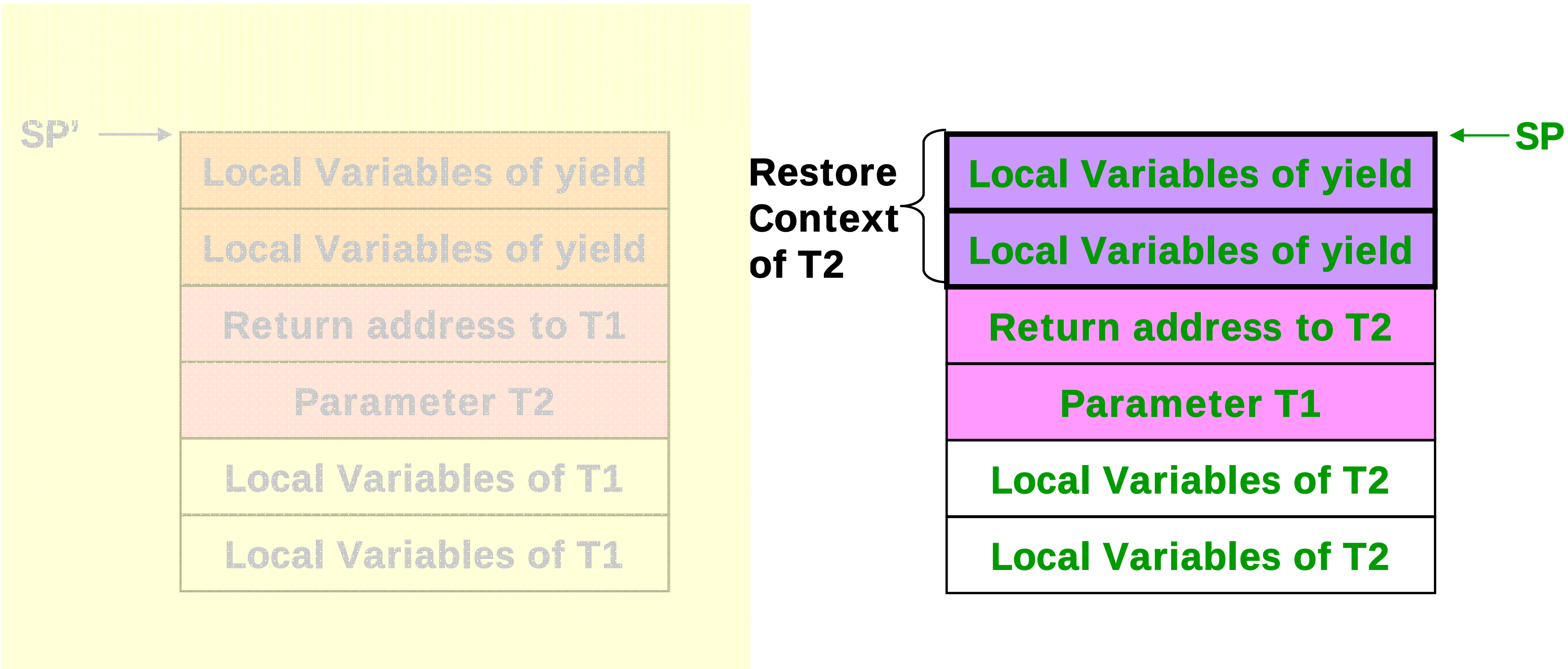


Stack Contents during UL-Yield (5)

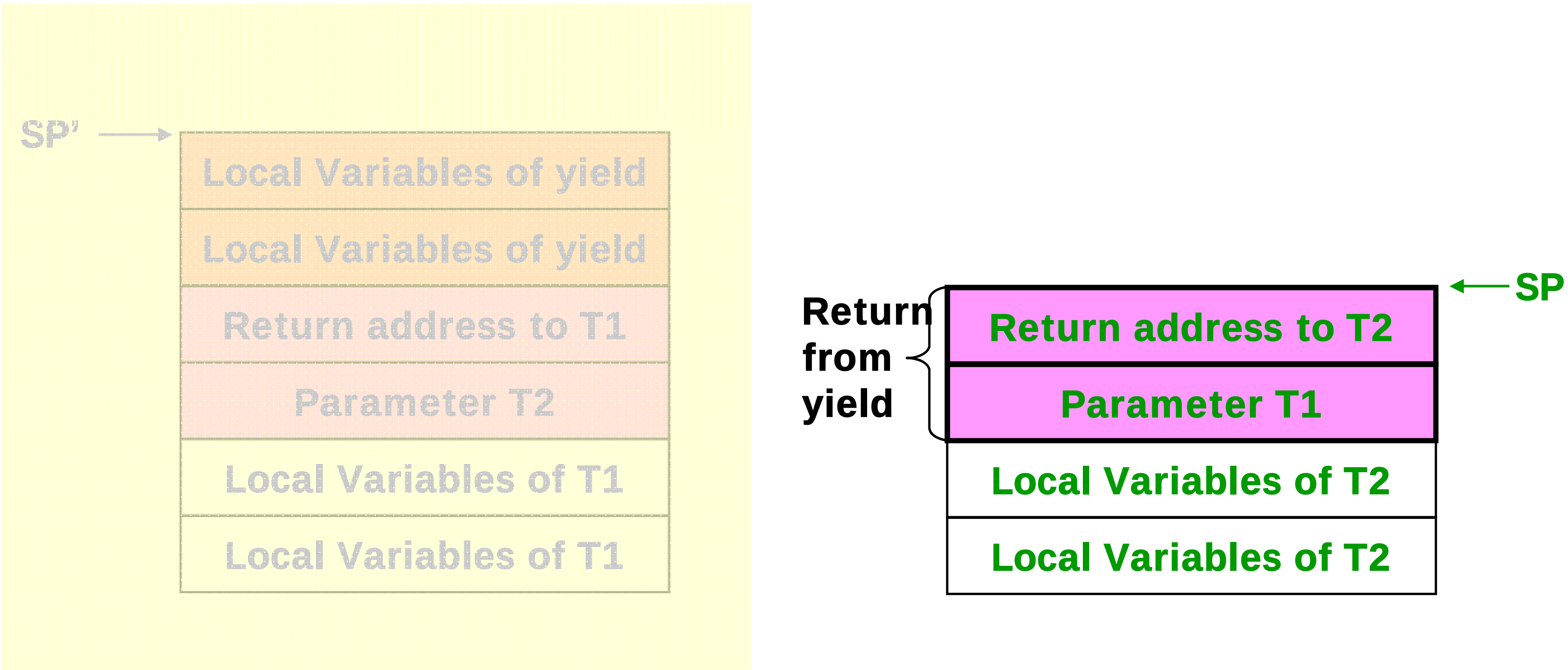
Switch Stack Pointer



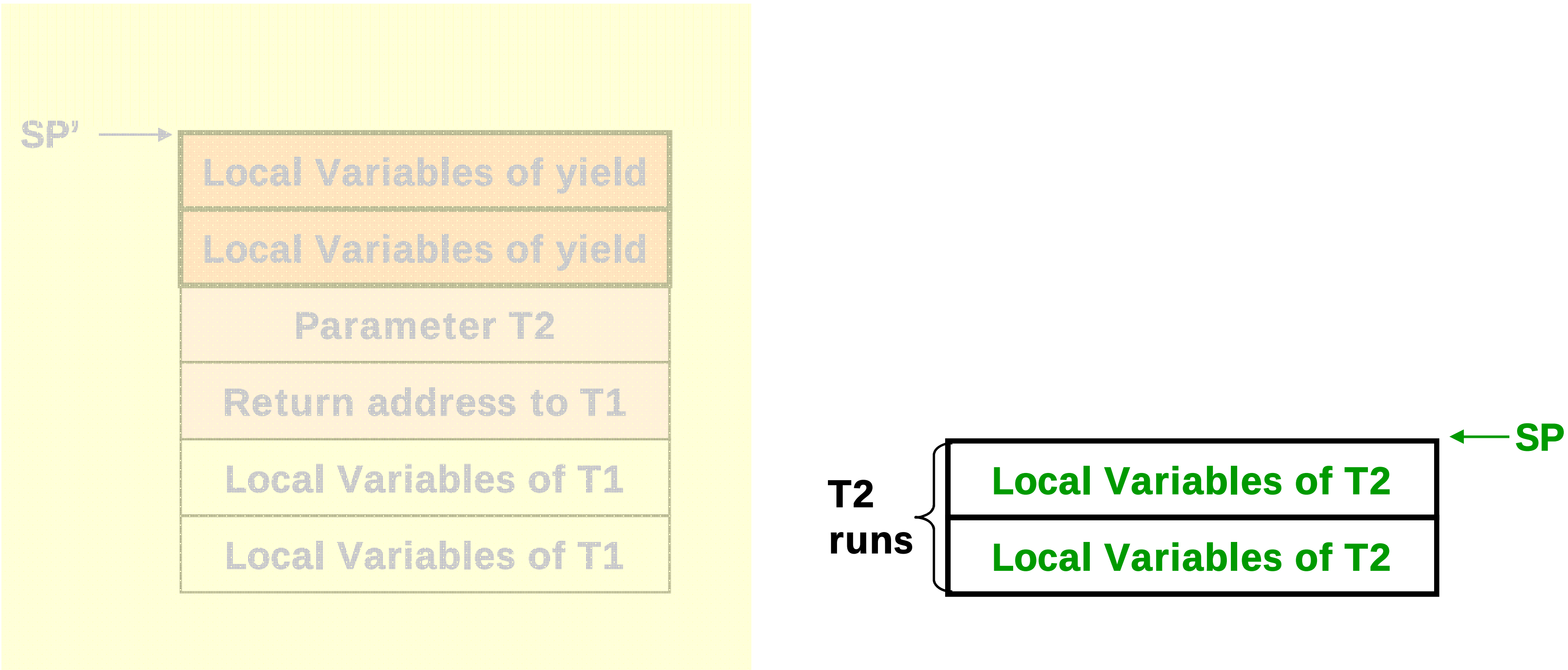
Stack Contents during UL-Yield (6)



Stack Contents during UL-Yield (7)



Stack Contents during UL-Yield (8)



Summary of a PULT-Yield

<u>Assumption:</u>

Suppose we have a single processor system, and yield is the only dispatching possibility $\Rightarrow$

- Only the stack of the running thread is “visible”
- Number of involved stack elements as well as their order is the same
- Content of involved stack elements differ a bit
- Of course, T_1 or T_2 can have different local variables

Thread Library Contents

- Can contain code for:
  - Creating and destroying PULTs
  - Passing messages between PULTs
  - Scheduling thread execution
  - Synchronizing with other PULTs
  - Saving/restoring context of a PULT

Potential Kernel Support for PULTs

Though the kernel is not aware of a PULT, it is still managing the activity of the task that hosts the PULT

Example:
When a “PULT” does a “blocking system call ” ⇒ kernel **blocks** its **whole task**

From the point of view of the **PULT scheduler** this PULT is still in the **PULT thread state running** !*

Thesis:
PULT thread states are independent of task states

Cooperative Scheduling of KLTs

Anthony D. Joseph

http://inst.eecs.berkeley.edu/~cs162

KIT Thread Switch of KLTs

Causes for a Thread Switch

Additional reasons for switching to another thread:

synchronous

- Current Thread (CT) terminates
- CT calls synchronous I/O, must wait for result
- CT waits for a message from another thread
- CT is cooperative, hands over CPU to another thread

asynchronous

- CT exceeds its time slice
- CT has lower priority than another ready thread:
  - CT interrupted by a device waking up another thread
  - A higher-priority thread's sleep time is exhausted
  - CT creates a new thread with higher priority
- CT gets a software interrupt from another thread

"preemption"

Needed: External Events

- *What might happen if a KLT never does any I/O, never waits for anything, and never calls* **`yield()`**?
  - Could the ComputePI program grab all resources and never release the processor?
    - What if it didn't print to console?
  - Must find a way that the kernel dispatcher regains control
- Answer: Utilize External Events
  - Interrupts: signals from hardware or software that stop the running code and jump to kernel
  - Timer: like an alarm clock that goes off every x milliseconds
  - If we ensure that external events occur frequently enough, the dispatcher can gain control again

Events triggering a Thread Switch

Exceptions (all synchronous events):

- Faulty event (reproducible)
  - Division by zero (during instruction)
  - Address violation (during instruction)
- Unpredictable event
  - Page fault (before instruction)
- Breakpoint
  - Data (after instruction)
  - Code (before instruction)
- System call
  - Trap

Events triggering a Thread Switch

Interrupts (all asynchronous events[1]):

- Clock
  - End of time slice
  - Wake up signal
- Printer
  - Missing paper
  - Paper jam, ...
- Network
  - Packet arrived, ...
- Another CPU[1]
  - Inter-Processor signal
  - Software Interrupt

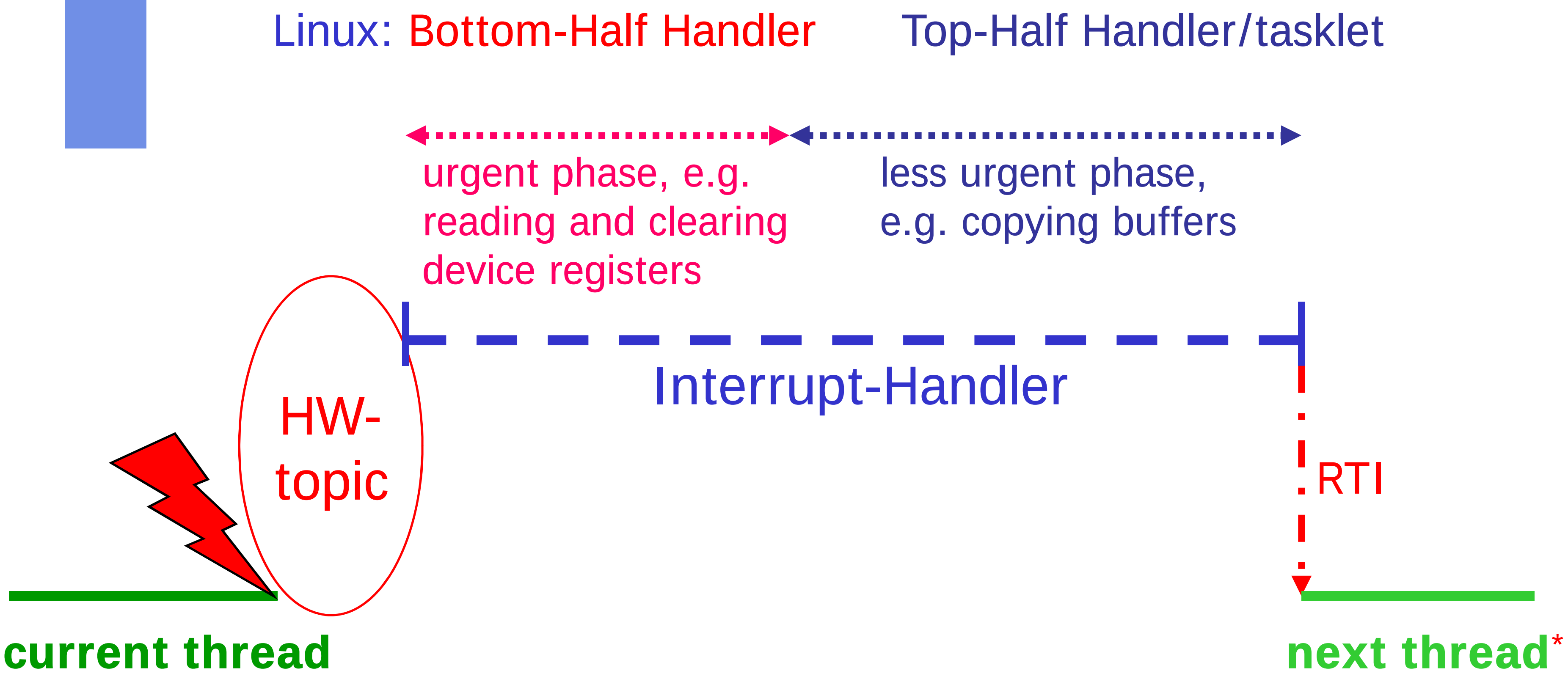
[1]From the point of view of the interrupted CPU

Nested Interrupt Handling (2)

APIC sits in between CPU and peripherals

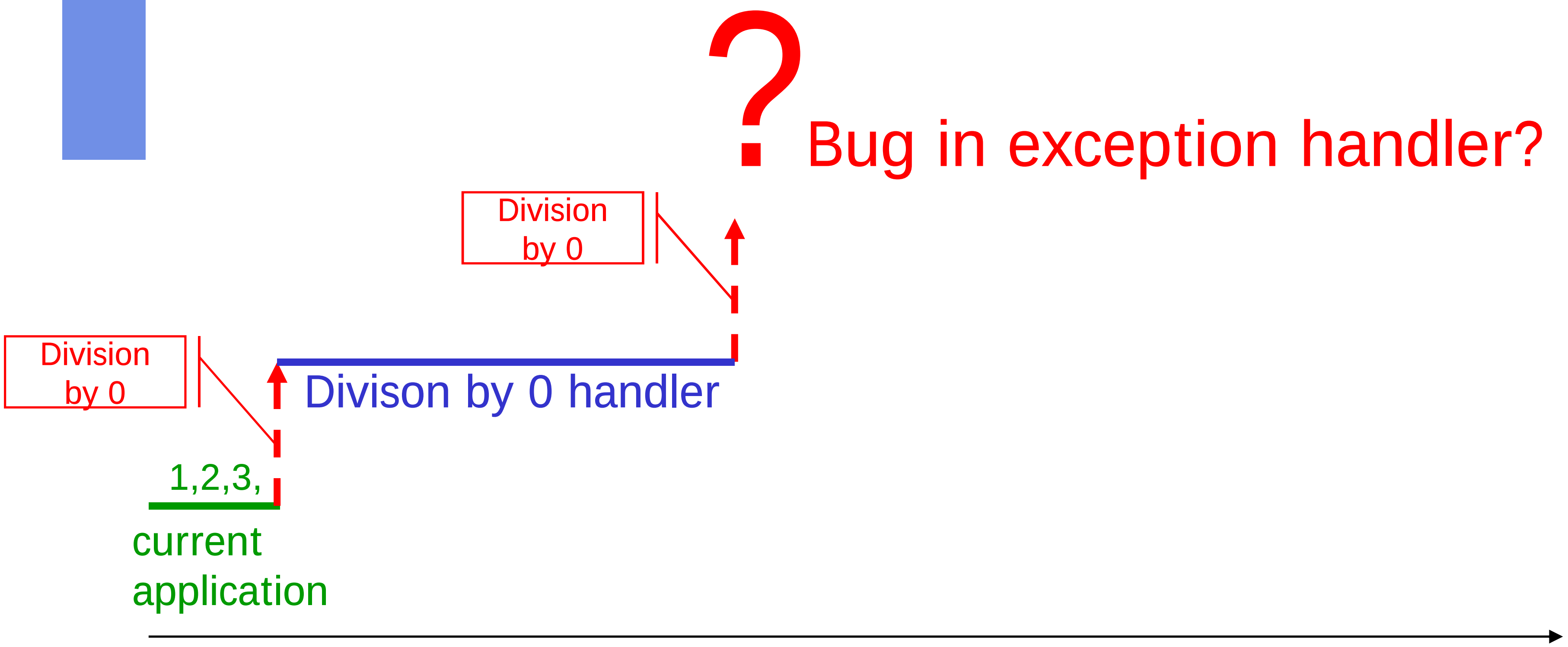
- IR-”Input” register
  - Pending interrupts are listed here (as “1” bits)
- MR-”Mask” register
  - Where IRs can be masked out
- IR-”Compare” register
  - Helps to decide whether interrupting the current interrupt handling is allowed
- Dynamic or static interrupt scheme
  - Rotating or fixed priorities

Linux Interrupt Handling



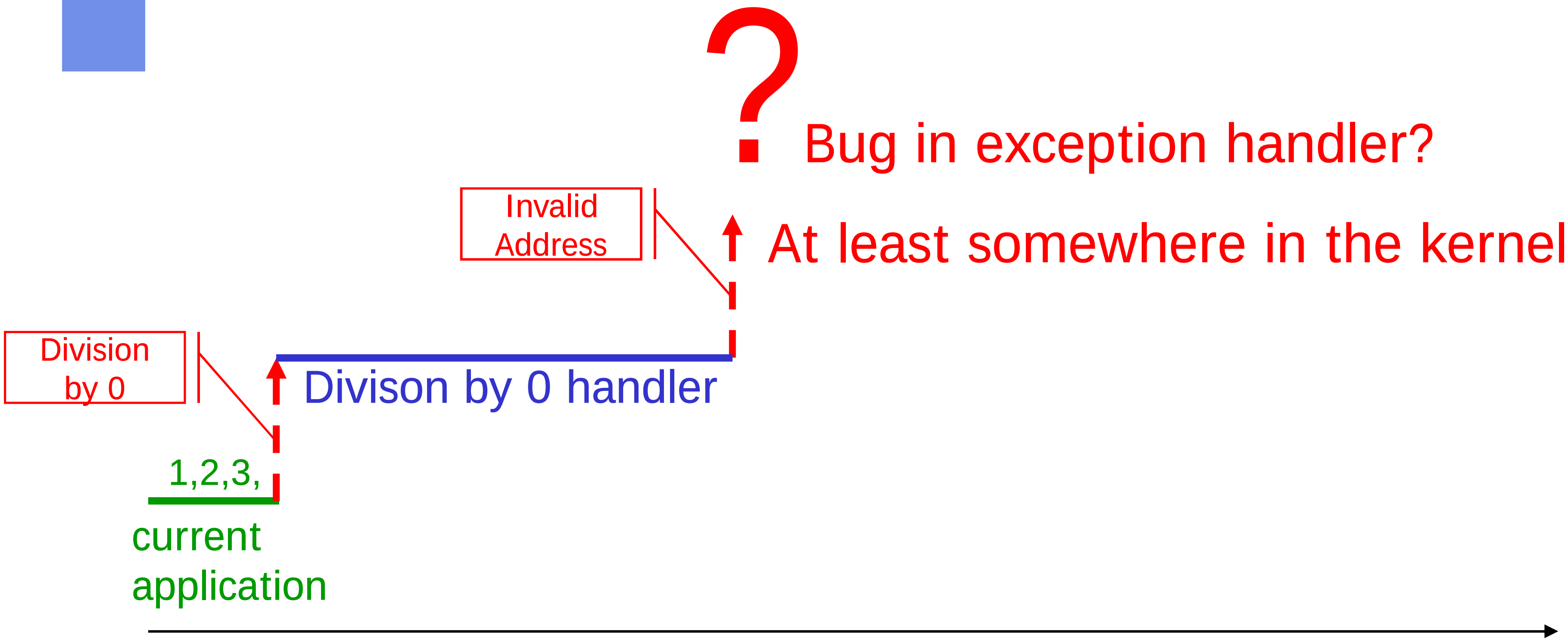
*Depending on system and/or interrupt, sometimes next thread = current thread

Nested Exception Handling (1)



Remark:
Some systems allow application-specific exception handlers.

Nested Exception Handling (2)



<u>Remark:</u>

Some systems allow two to three nested exceptions, but not more_

Construction Conclusion

Due to these events we need a centralized control instance in the

- Microkernel or
- Kernel

Due to the sensitivity of these events, thread switching and thread controlling need special protection:

- Kernel Mode
- Code and Data inside Kernel Address Space

Let’s study the case:

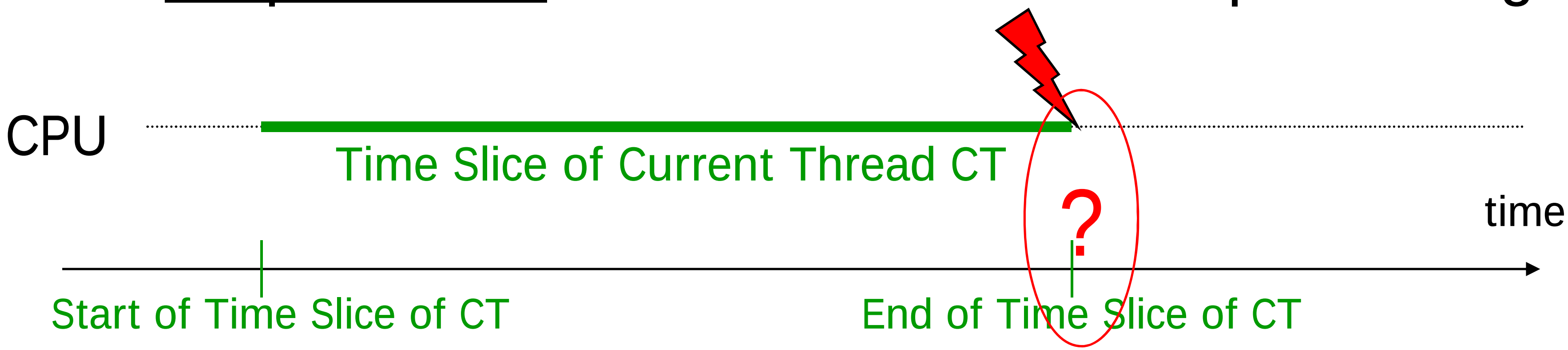
Current KLT **CT** has consumed its complete time slice

End of Time Slice ...

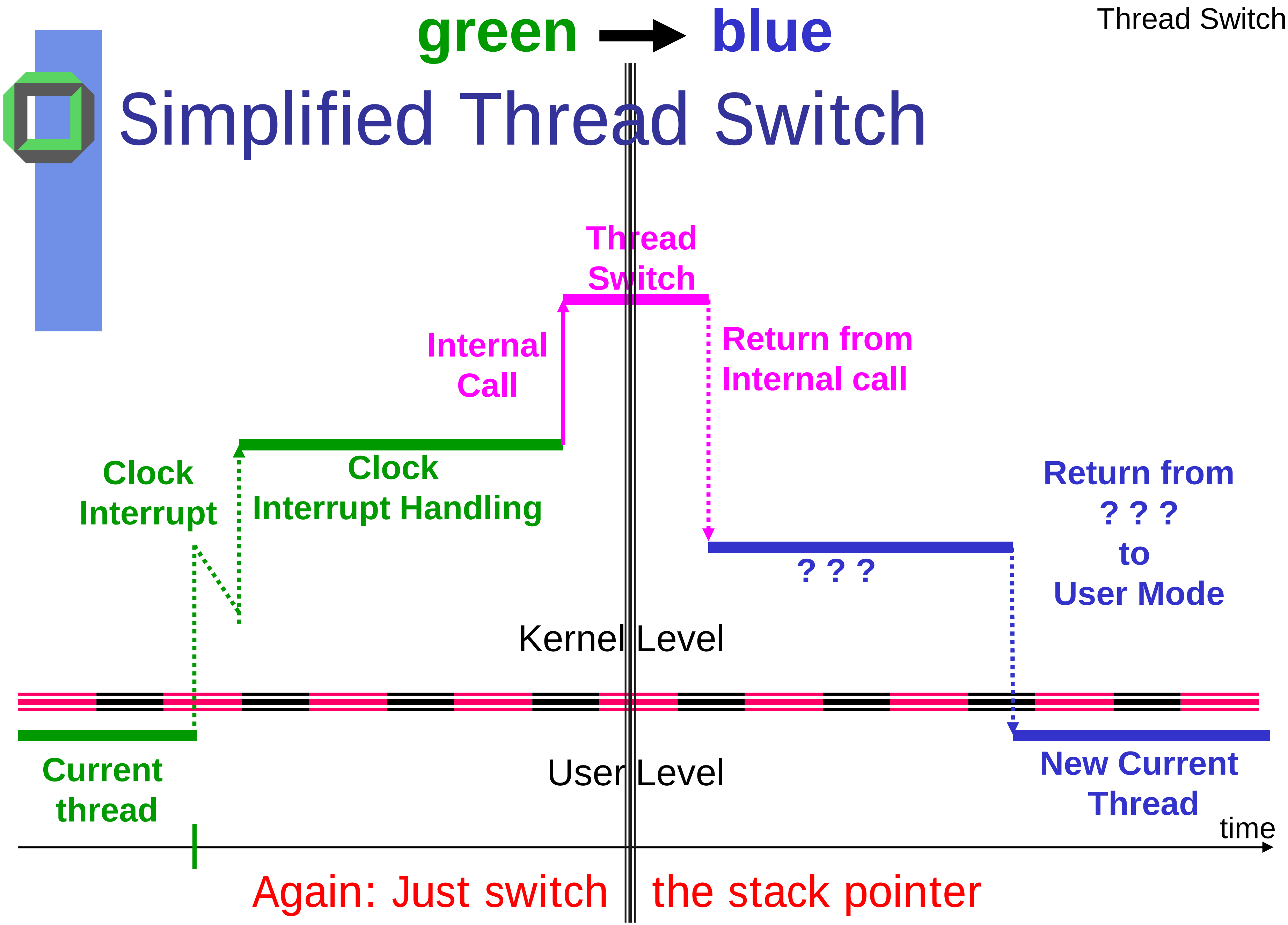
Objective: Establishing Fair Scheduling

Assumption: No other thread-switching events to be discussed in detail

Simplification: No detailed clock interrupt handling



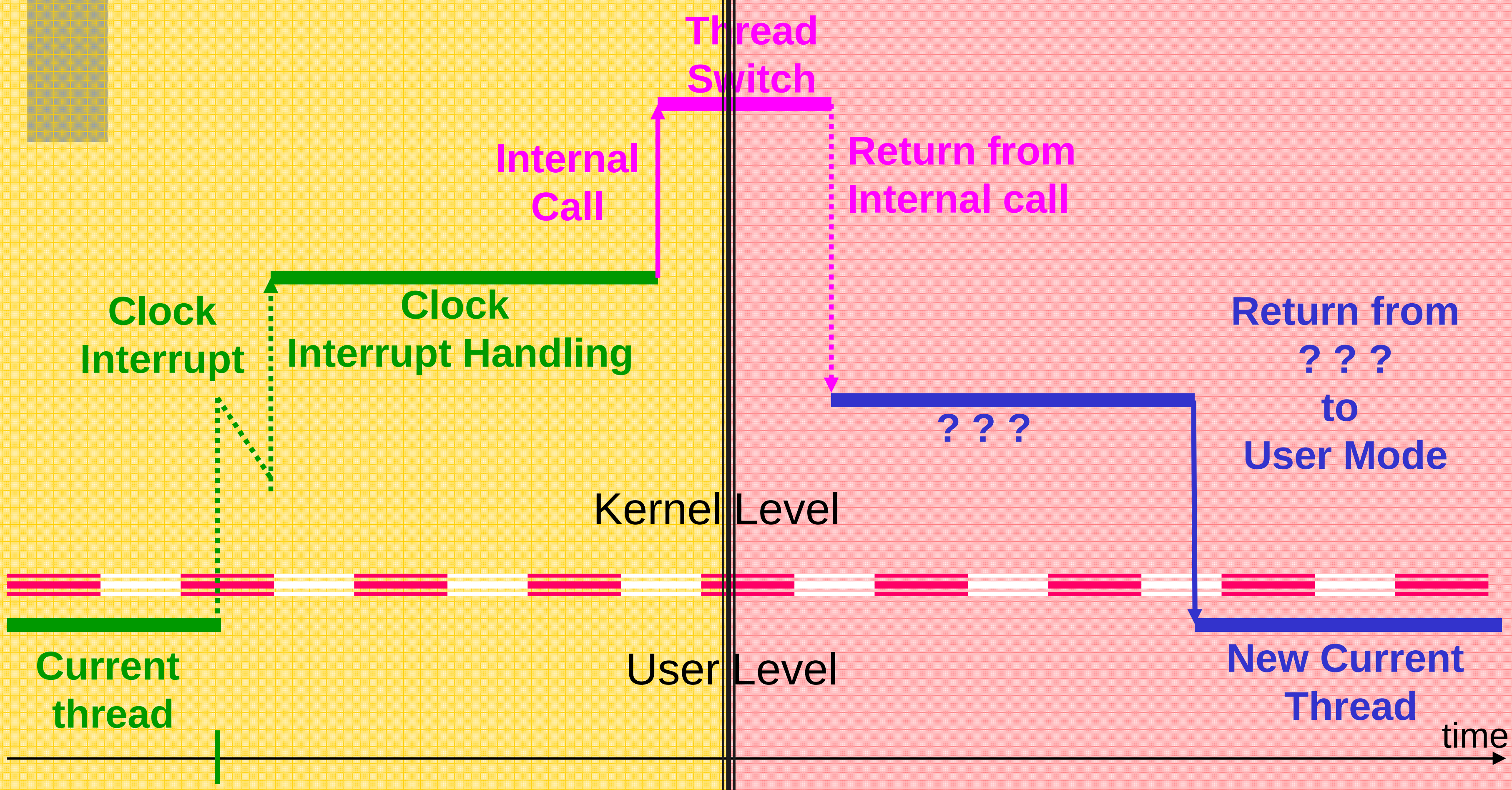
Simplified Thread Switch



Again: Just switch the stack pointer

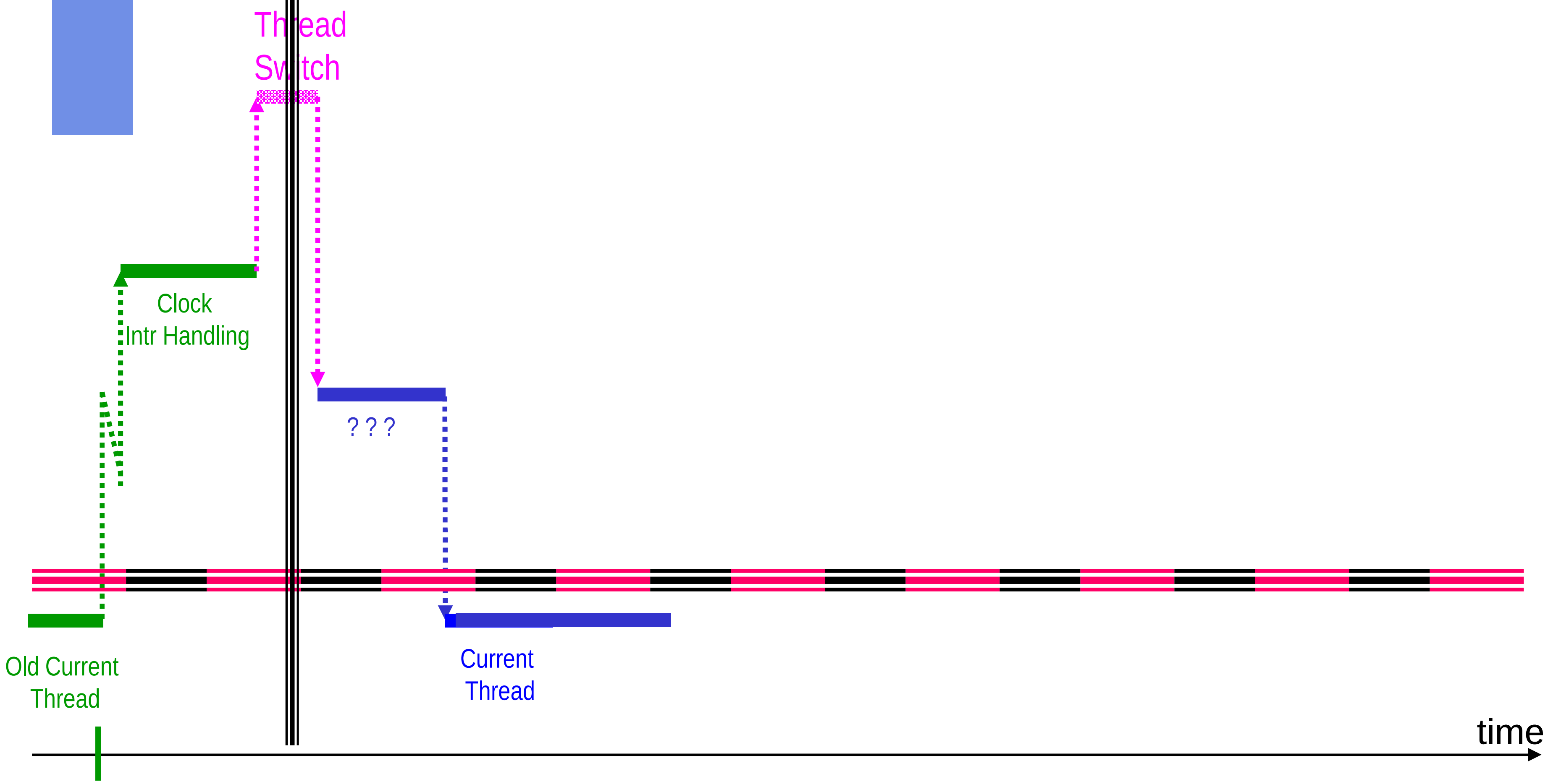
green → blue

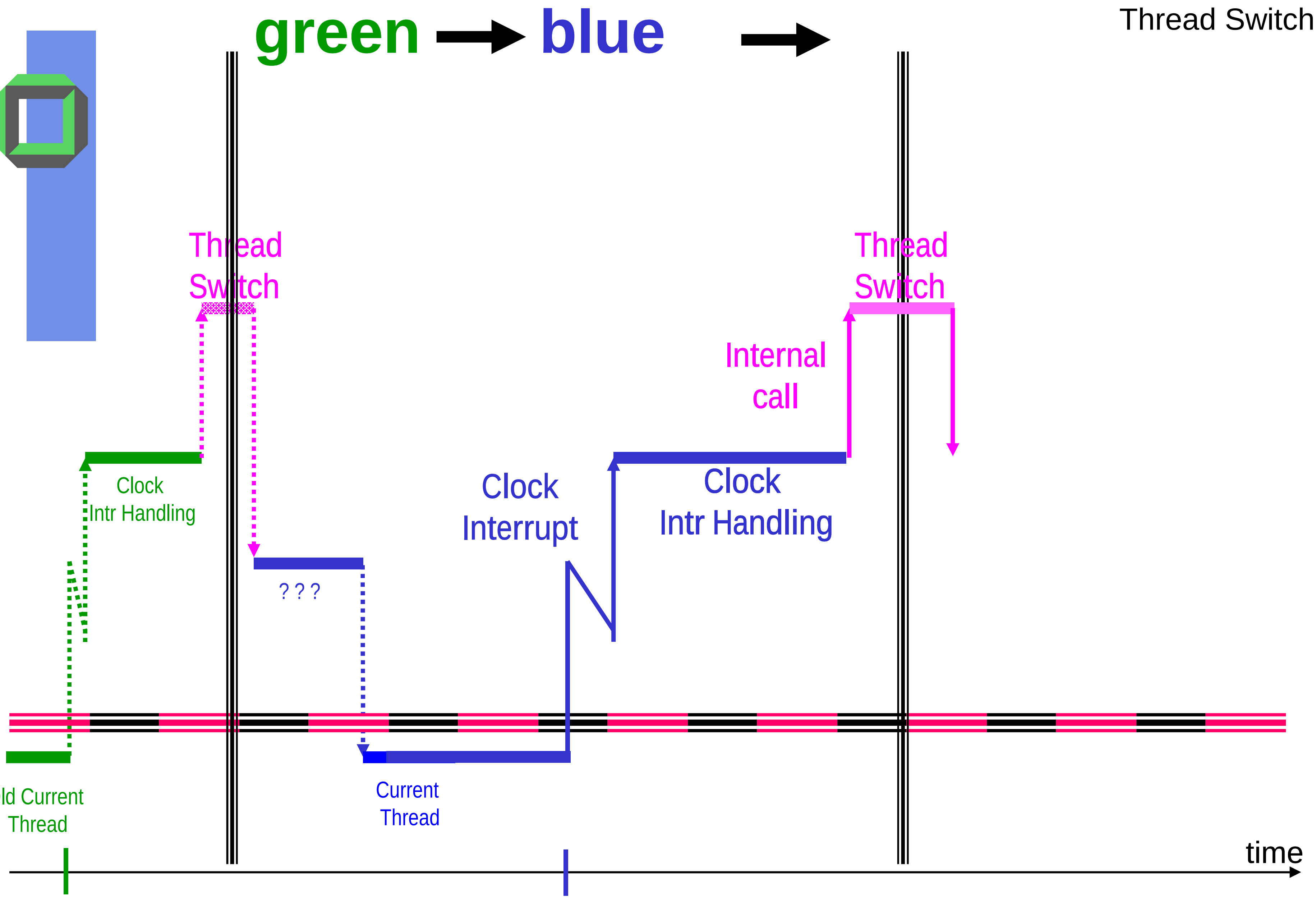
Simplified Thread Switch



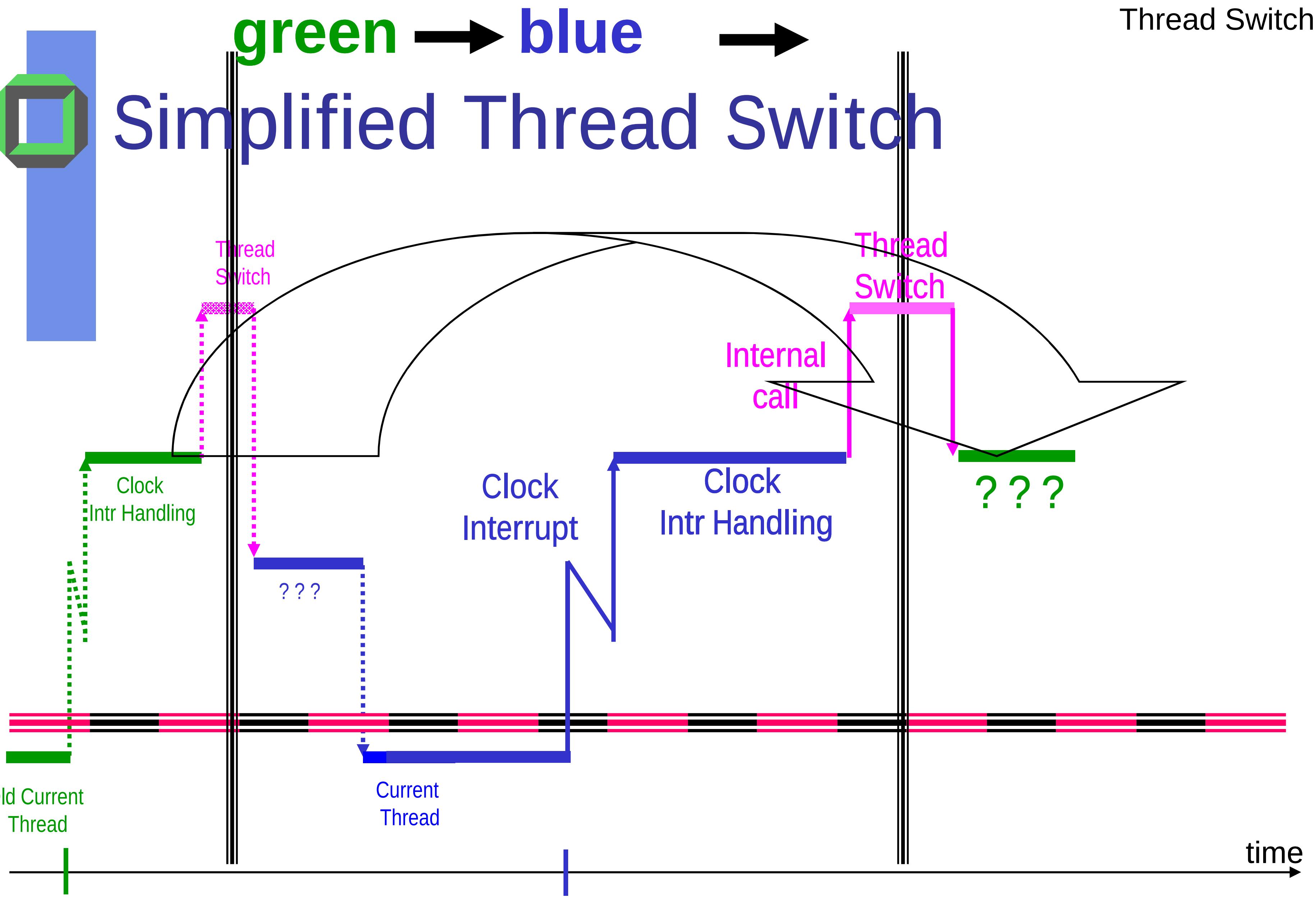
green → blue

Simplified Thread Switch



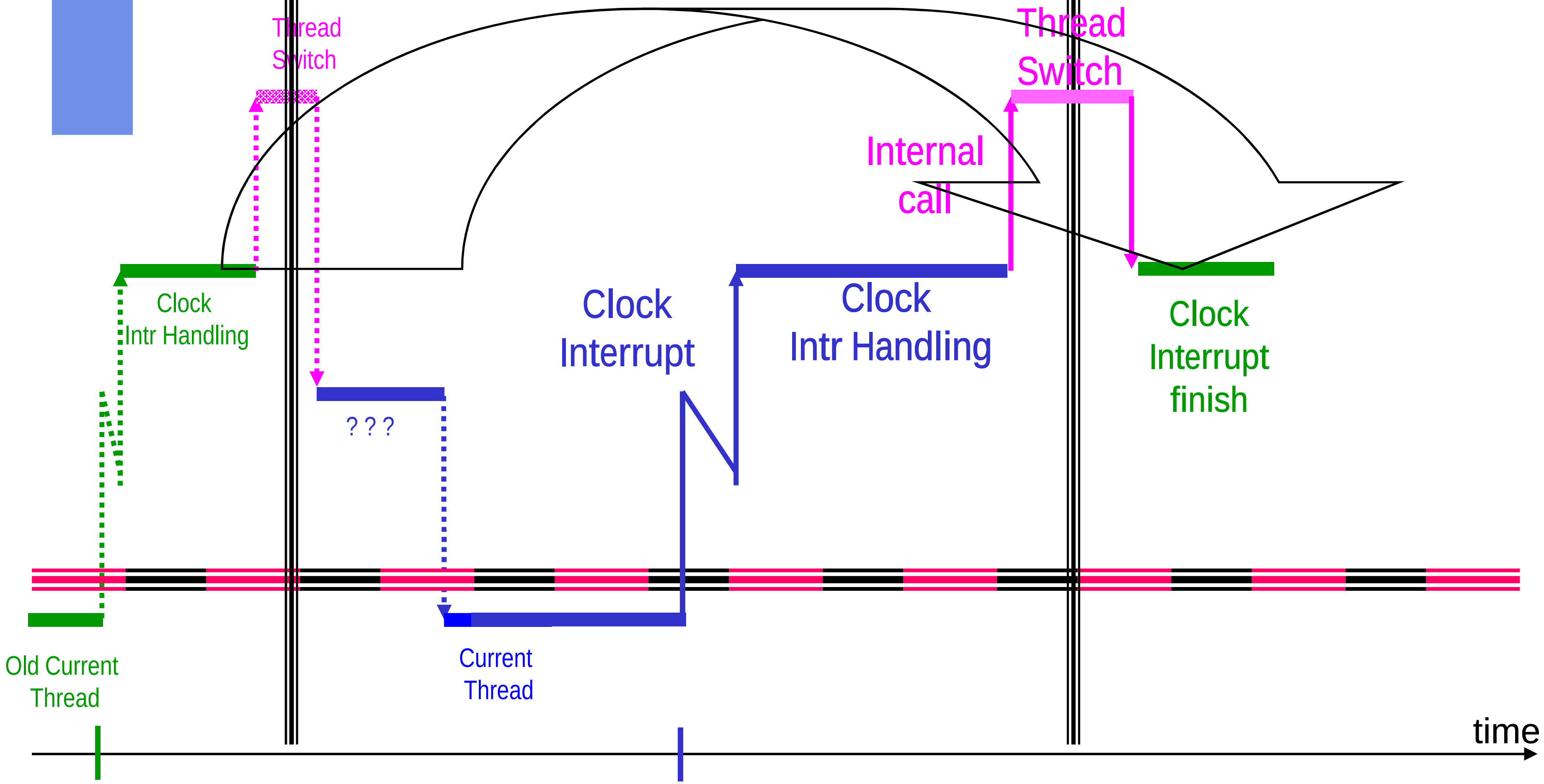

green → blue →
Thread Switch
Thread Switch
Internal call
Clock Intr Handling
Clock Interrupt
Clock Intr Handling
? ? ?
Old Current Thread
Current Thread
time

Simplified Thread Switch

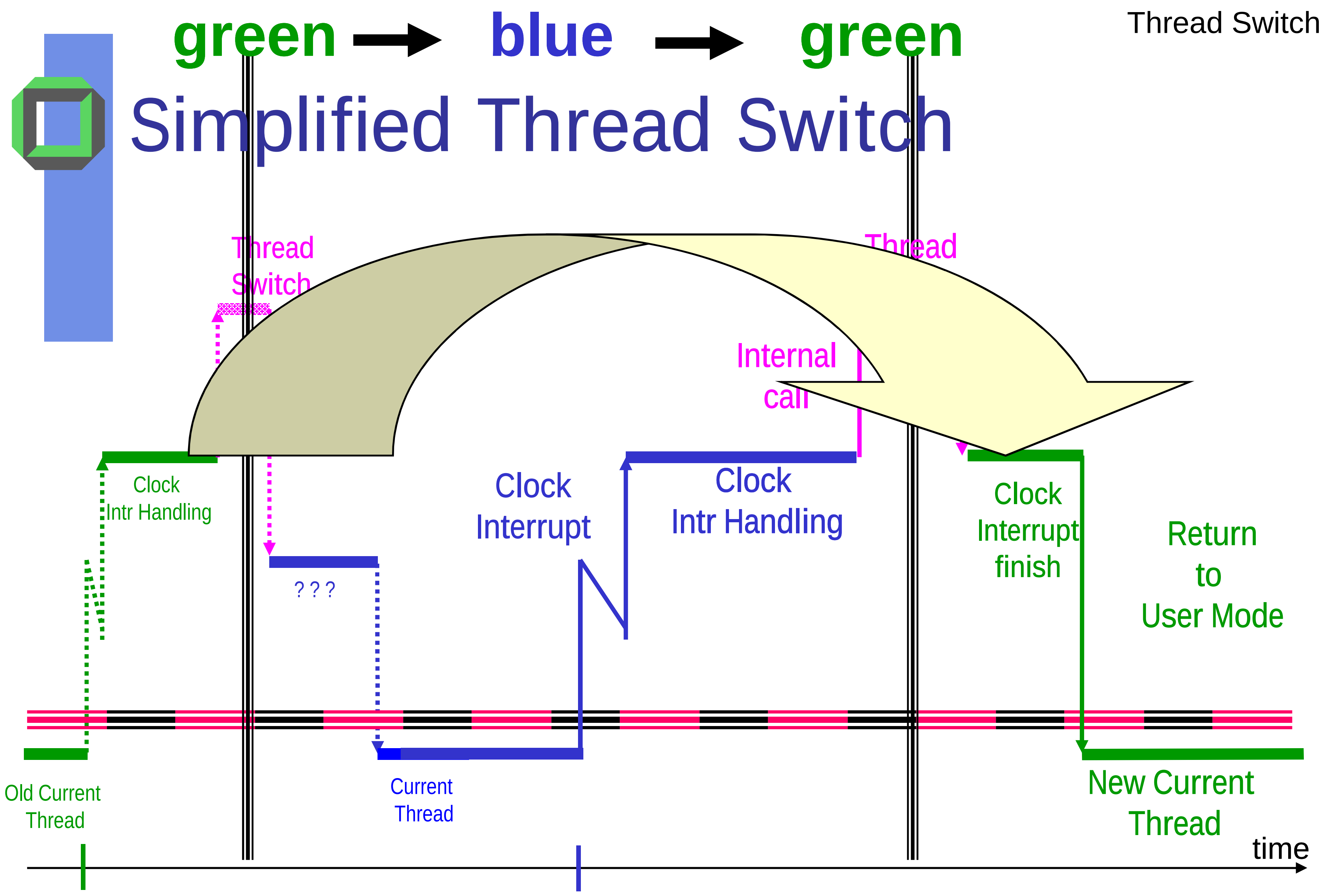


green → blue → green

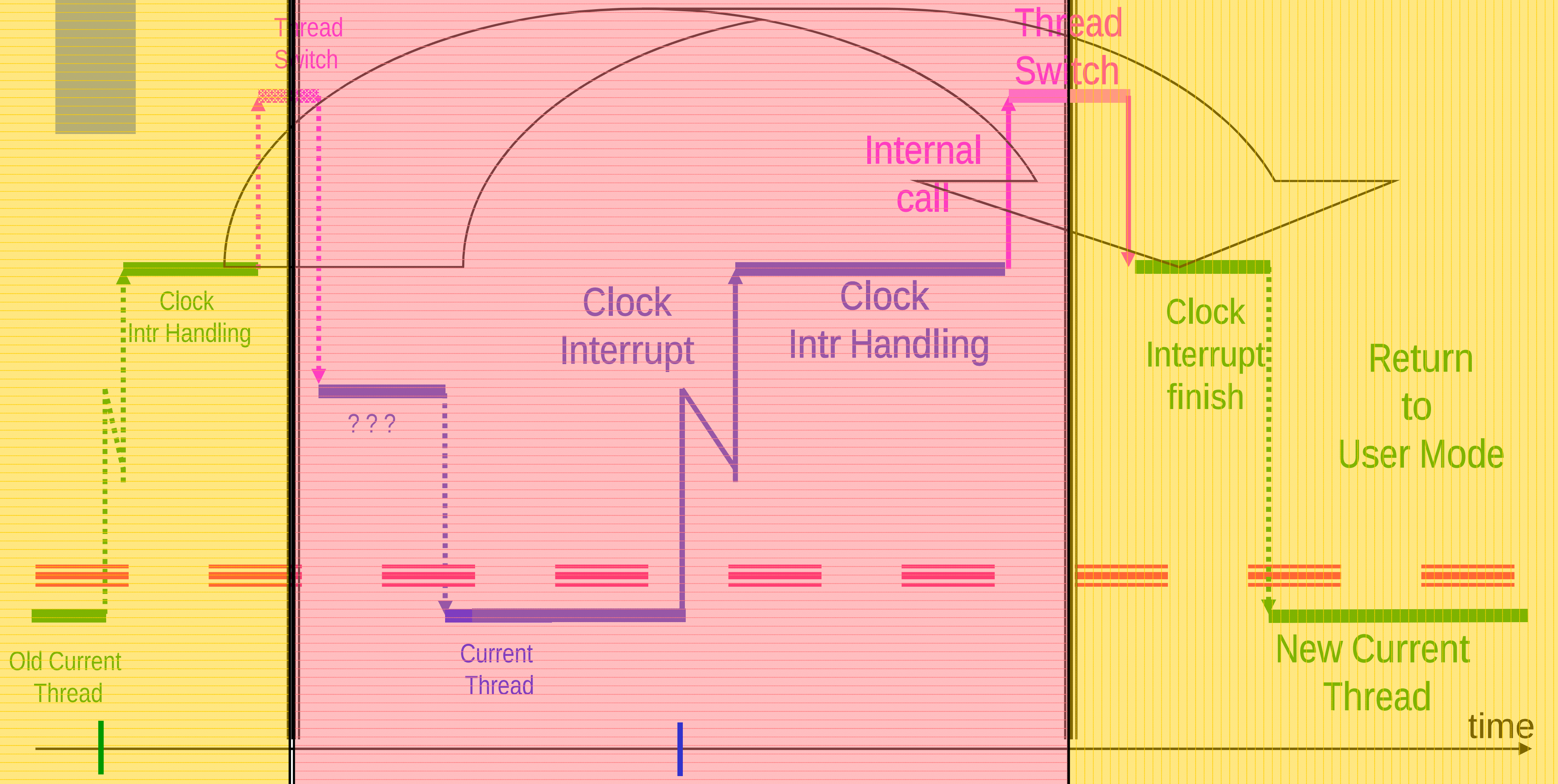
Simplified Thread Switch



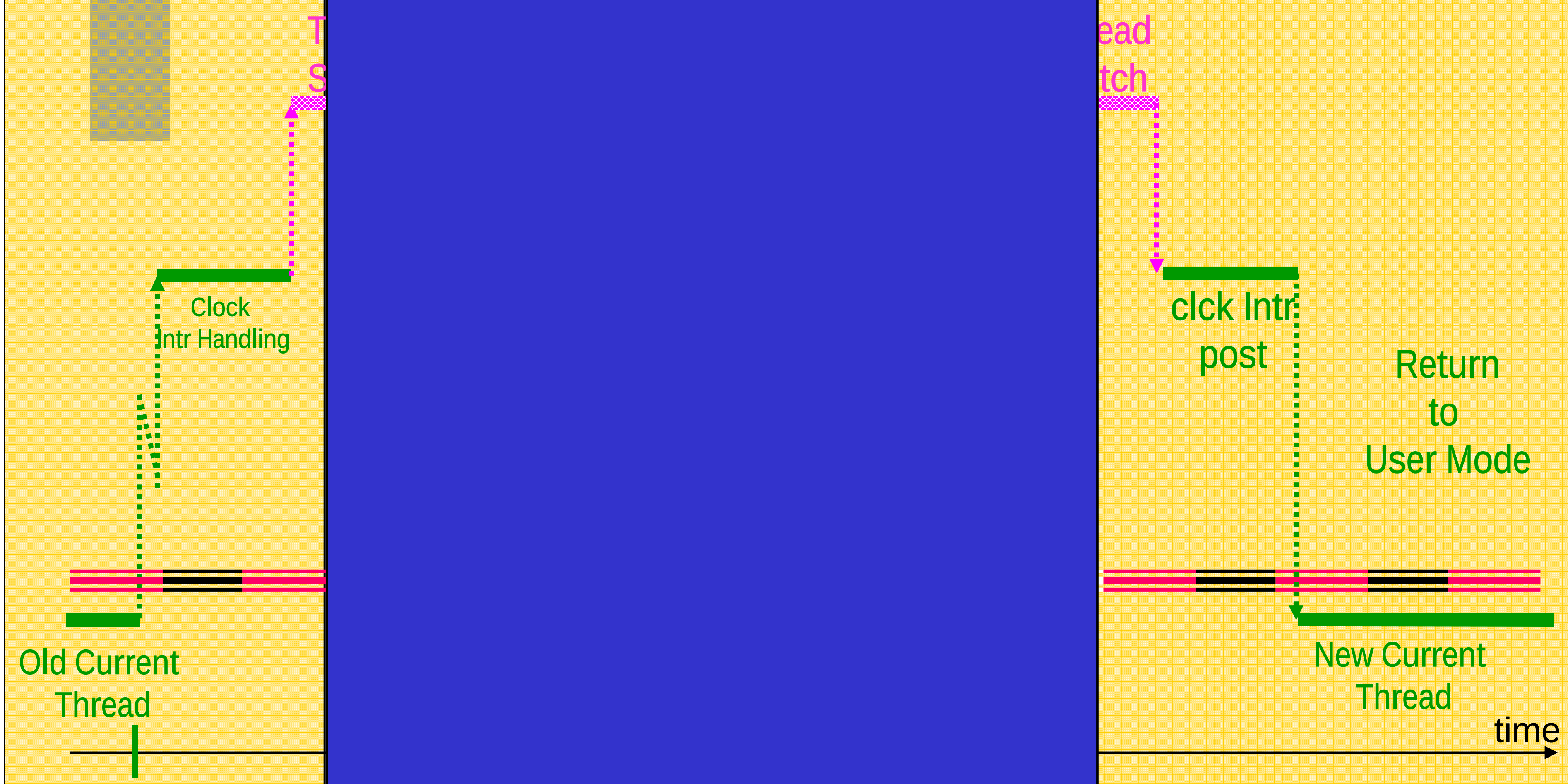
Simplified Thread Switch



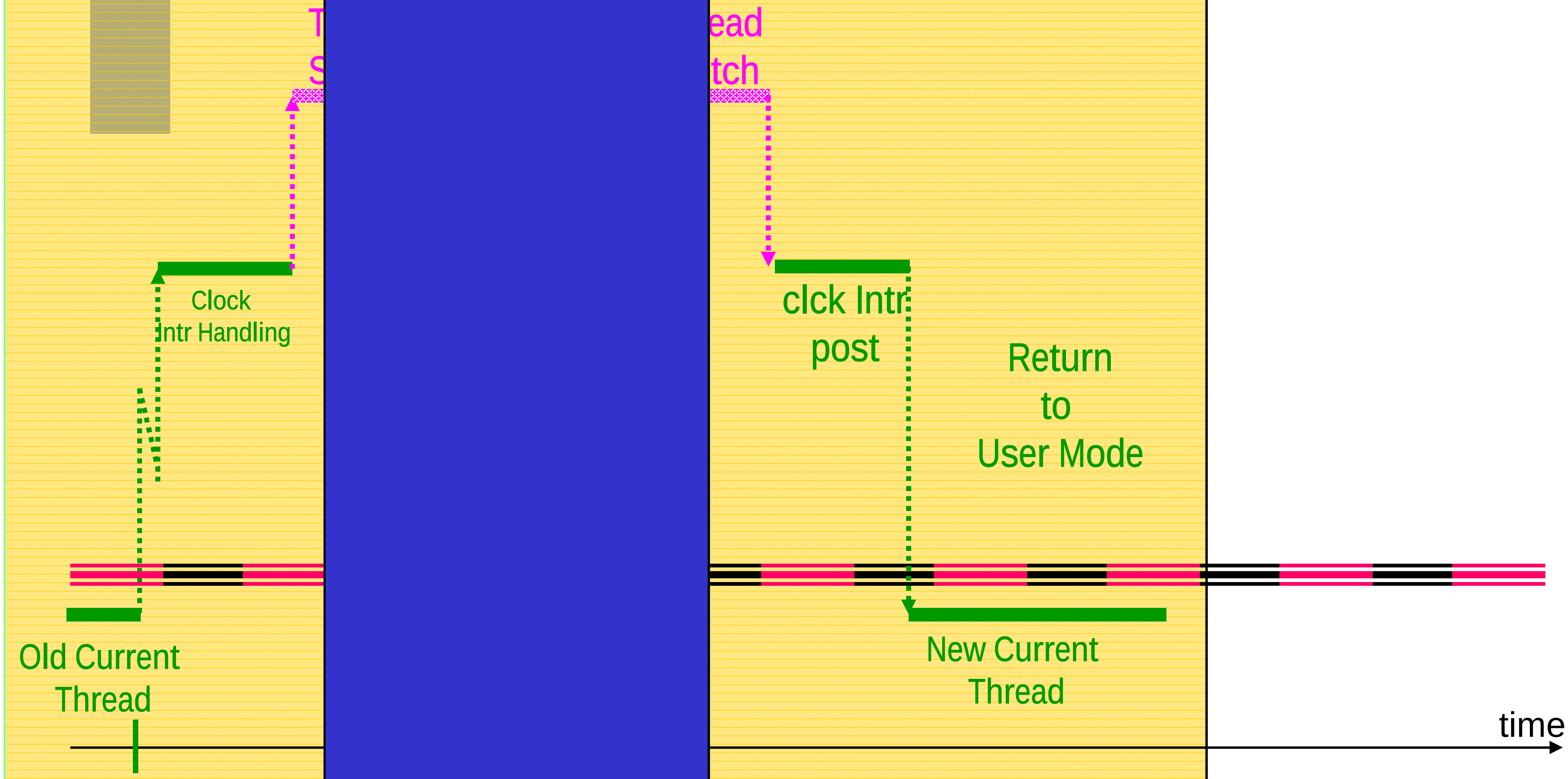
Simplified Thread Switch



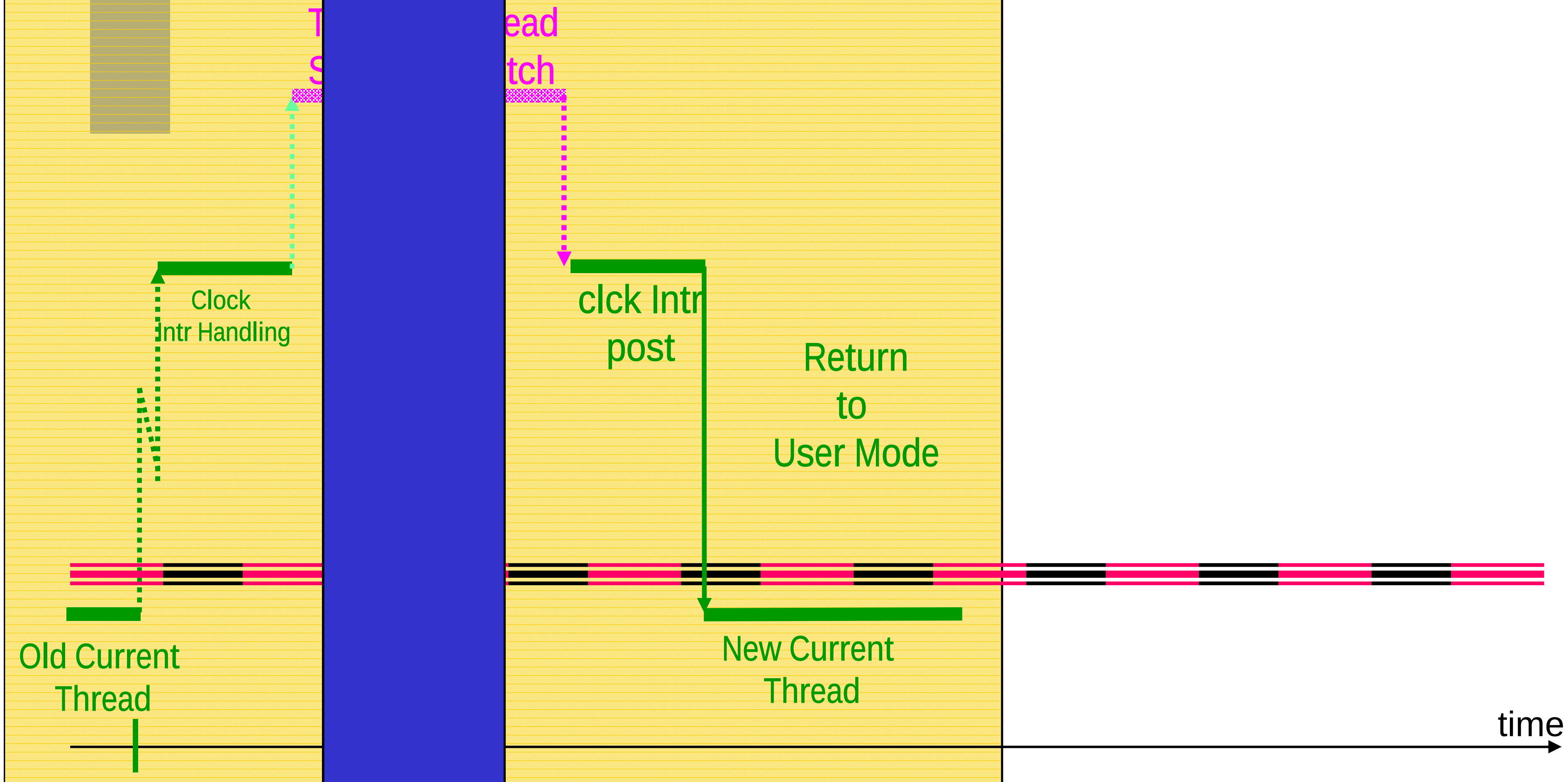
Simplified Thread Switch



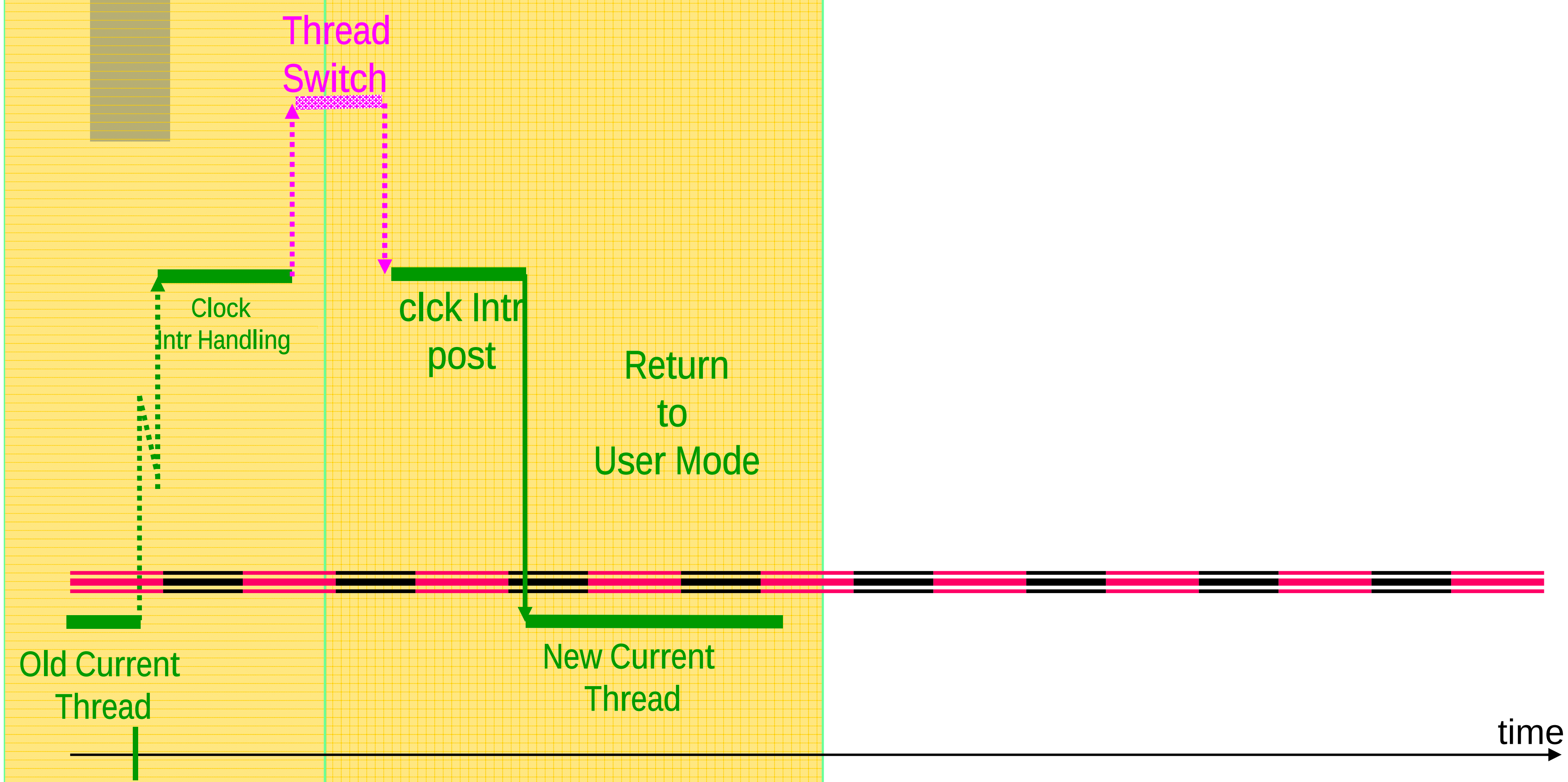
Simplified Thread Switch



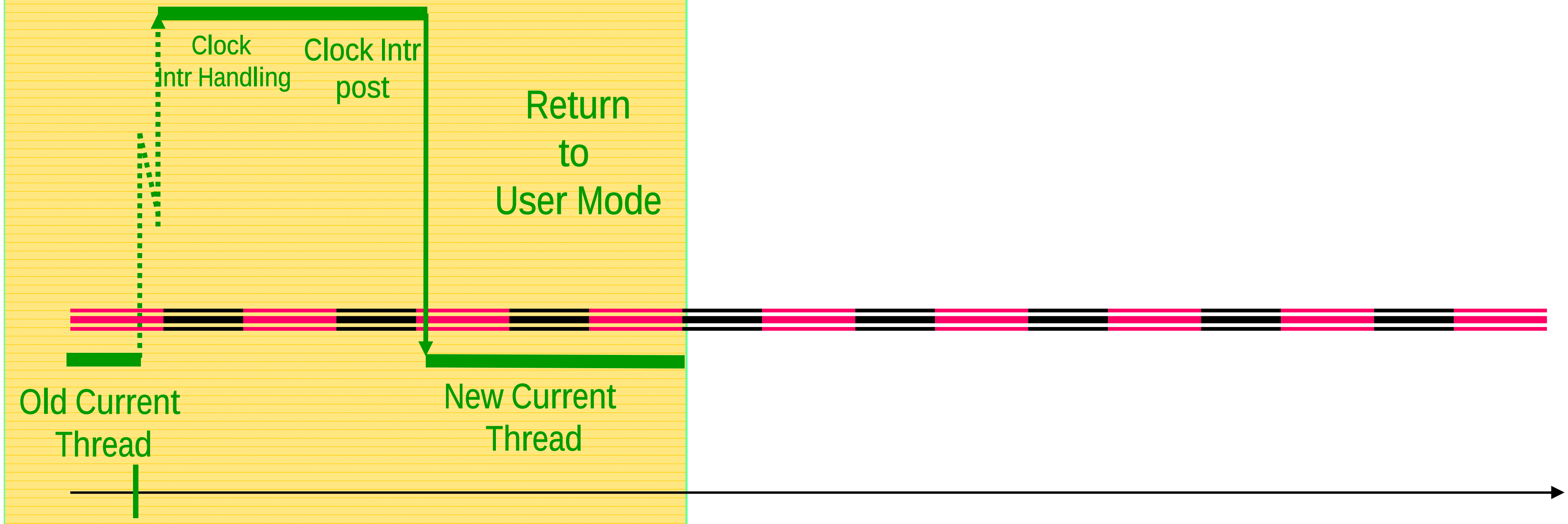
Simplified Thread Switch



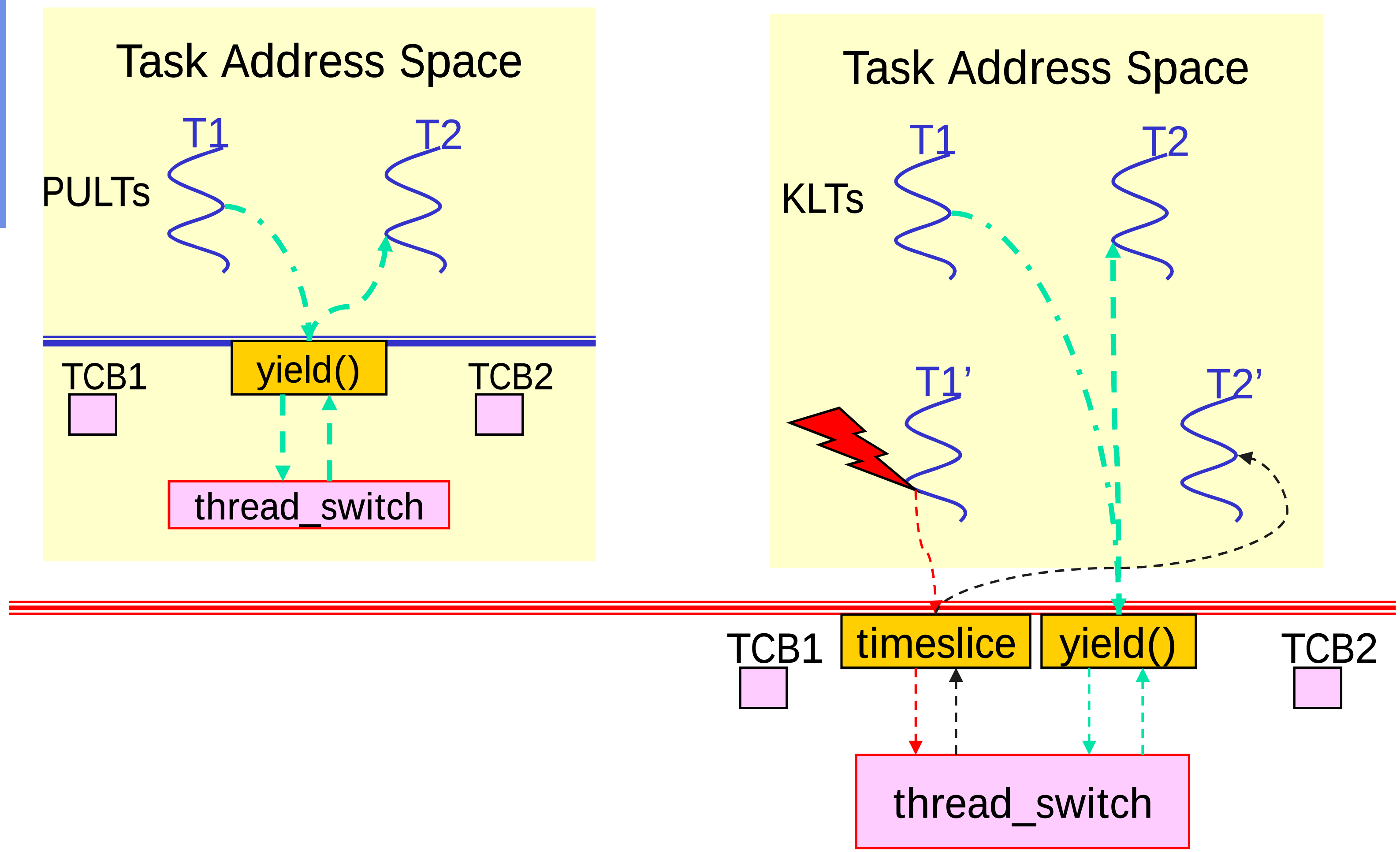
Simplified Thread Switch



Simplified Thread Switch



Review



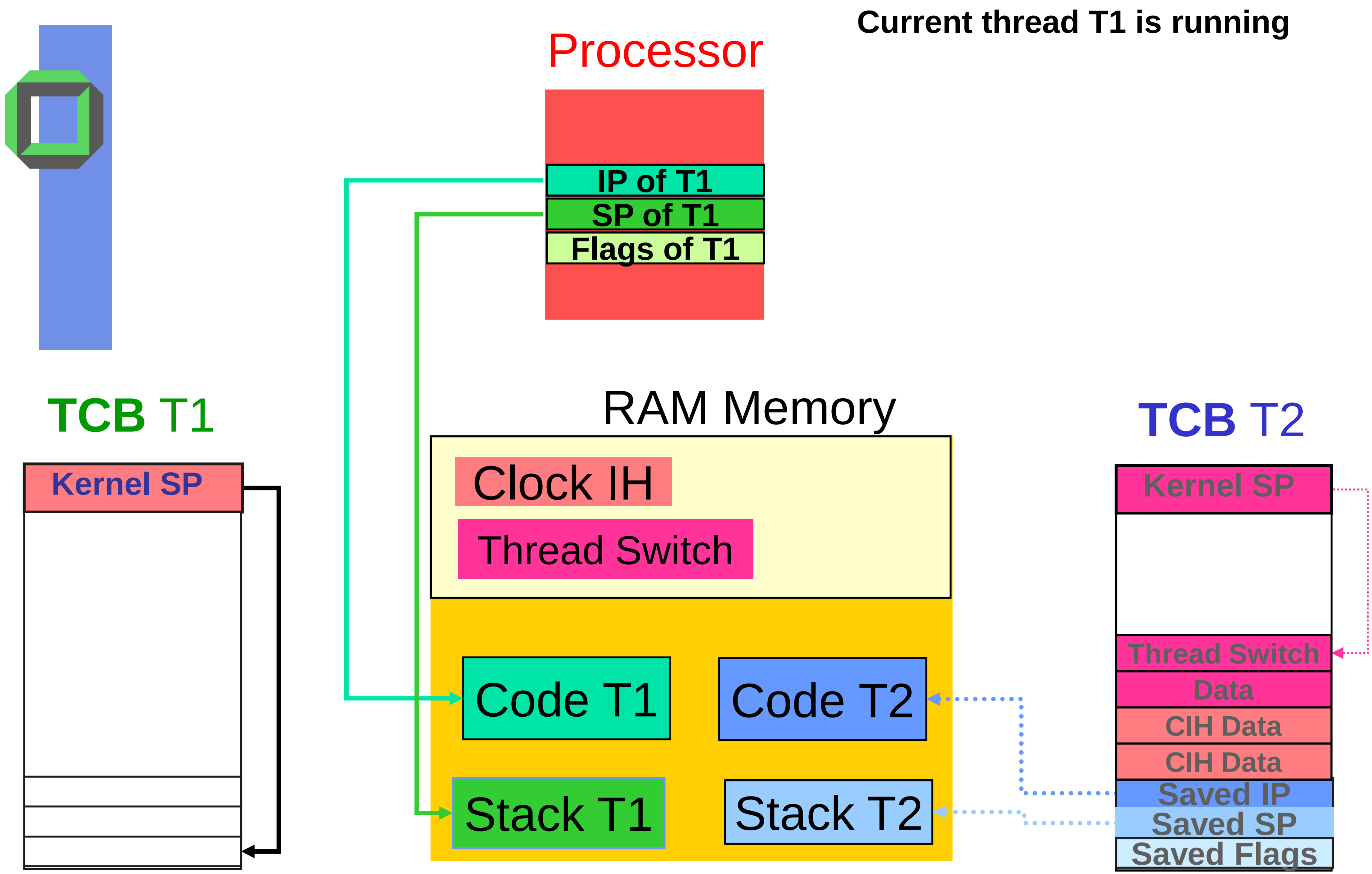
Task Address Space
PULTs
T1
T2
yield()
TCB1
TCB2
thread_switch
Task Address Space
KLTs
T1
T2
T1'
T2'
TCB1
timeslice
yield()
TCB2
thread_switch

Thread Switch Implementation

Assumption[1]:

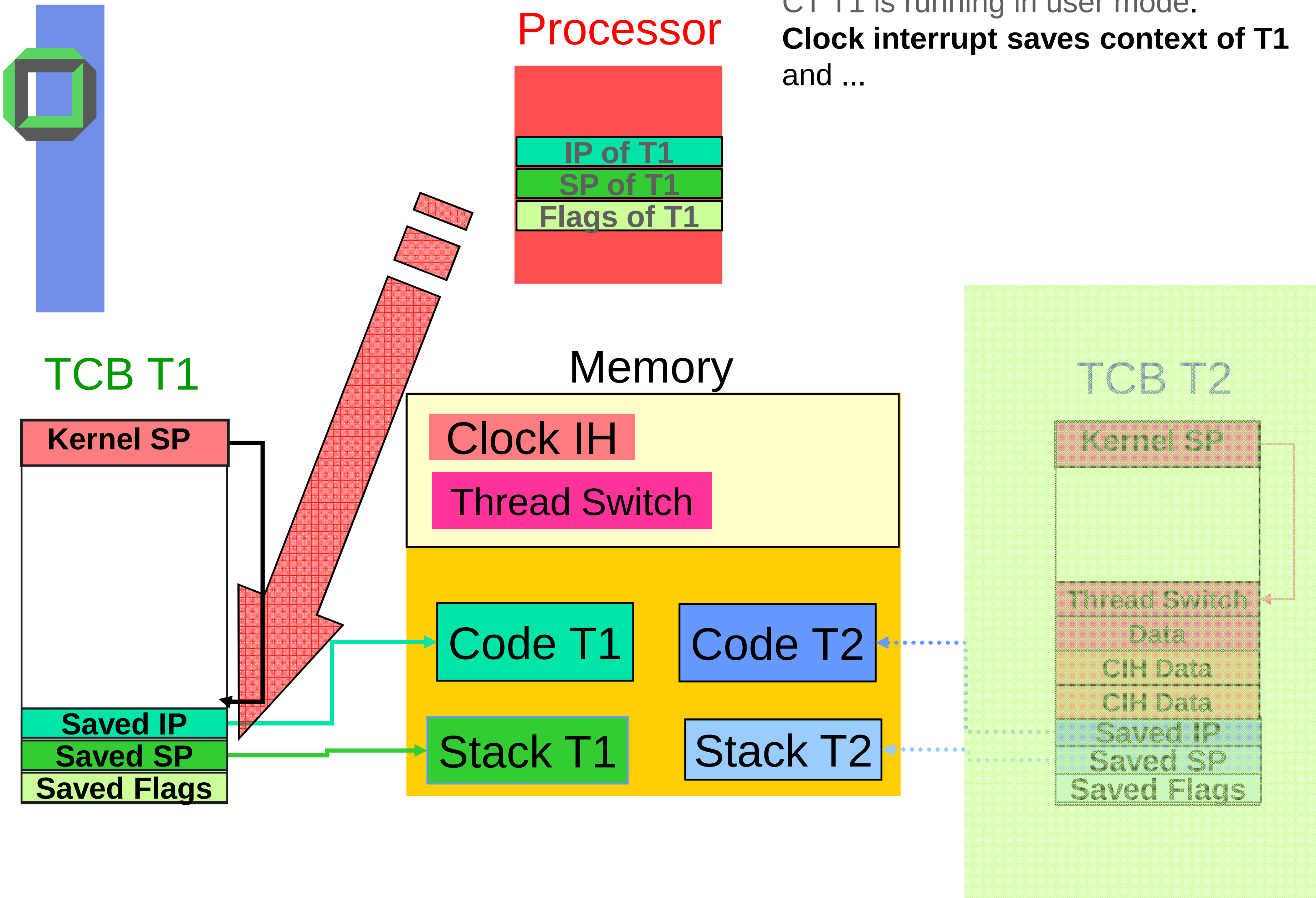
- Whenever entering the kernel, i.e. via
  - interrupt
  - exception
  - system call
- the HW automatically pushes SP, IP and status flags, e.g. the user-context of the current thread, e.g. CT = T1 onto the kernel stack (T1)
- Kernel stack is implemented in its related TCB, e.g. TCB1

[1]Some processors use shadow register instead of

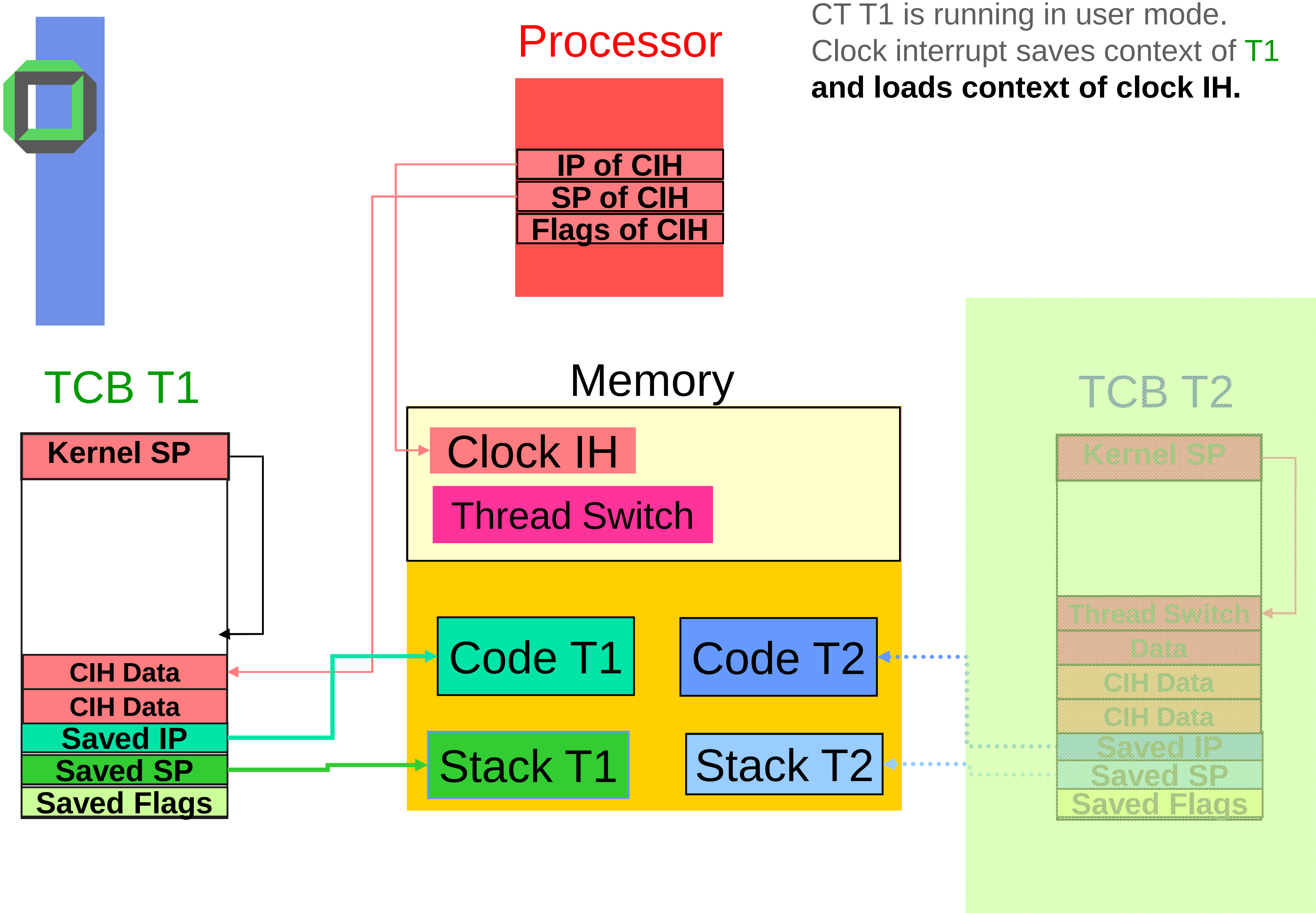


Note:
As long as T1 is running in user mode, the kernel stack is nearly empty,
However, at least the start address of the kernel stack is kept in TCBT1.SP

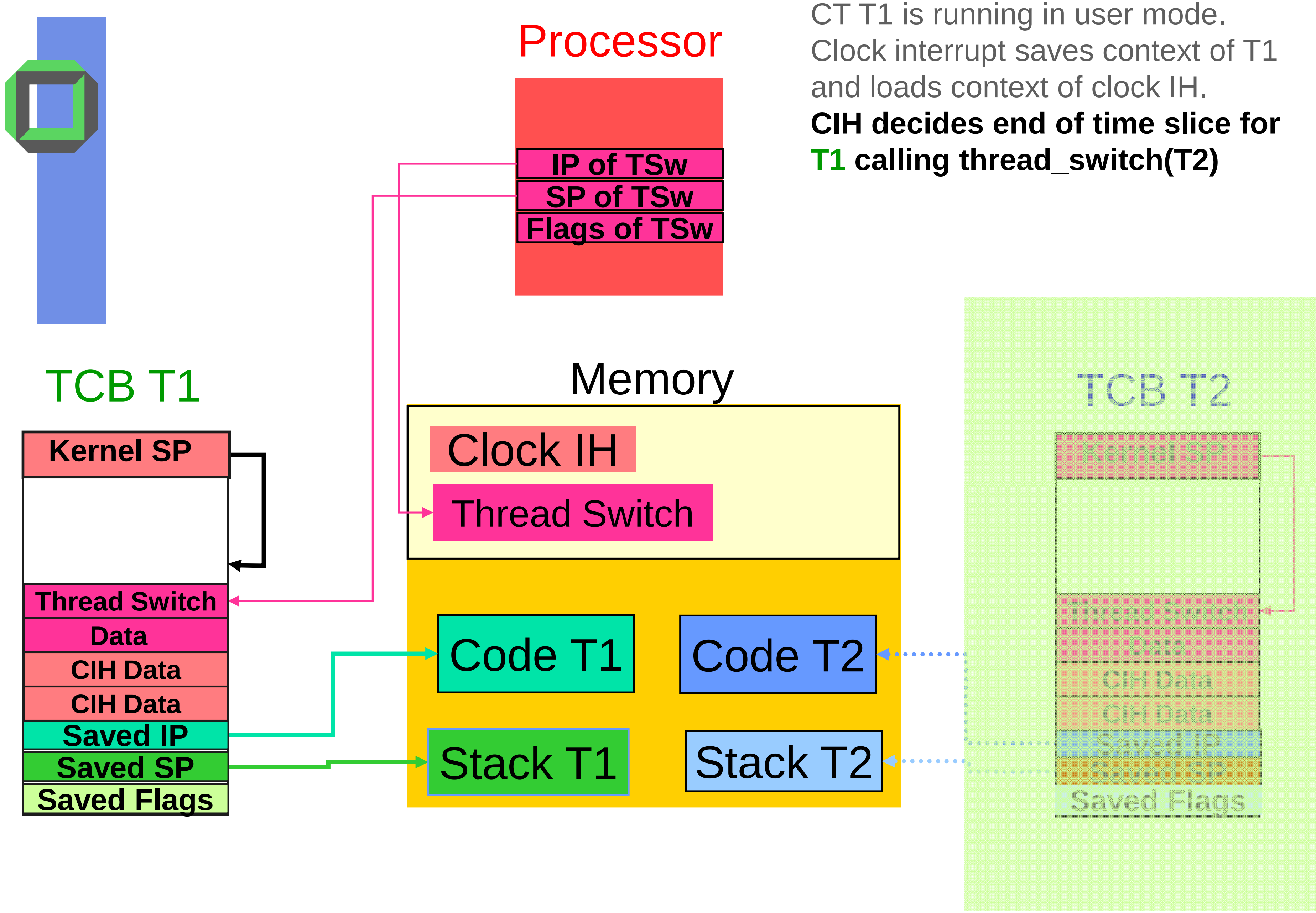
CT T1 is running in user mode.
Clock interrupt saves context of T1
and ...

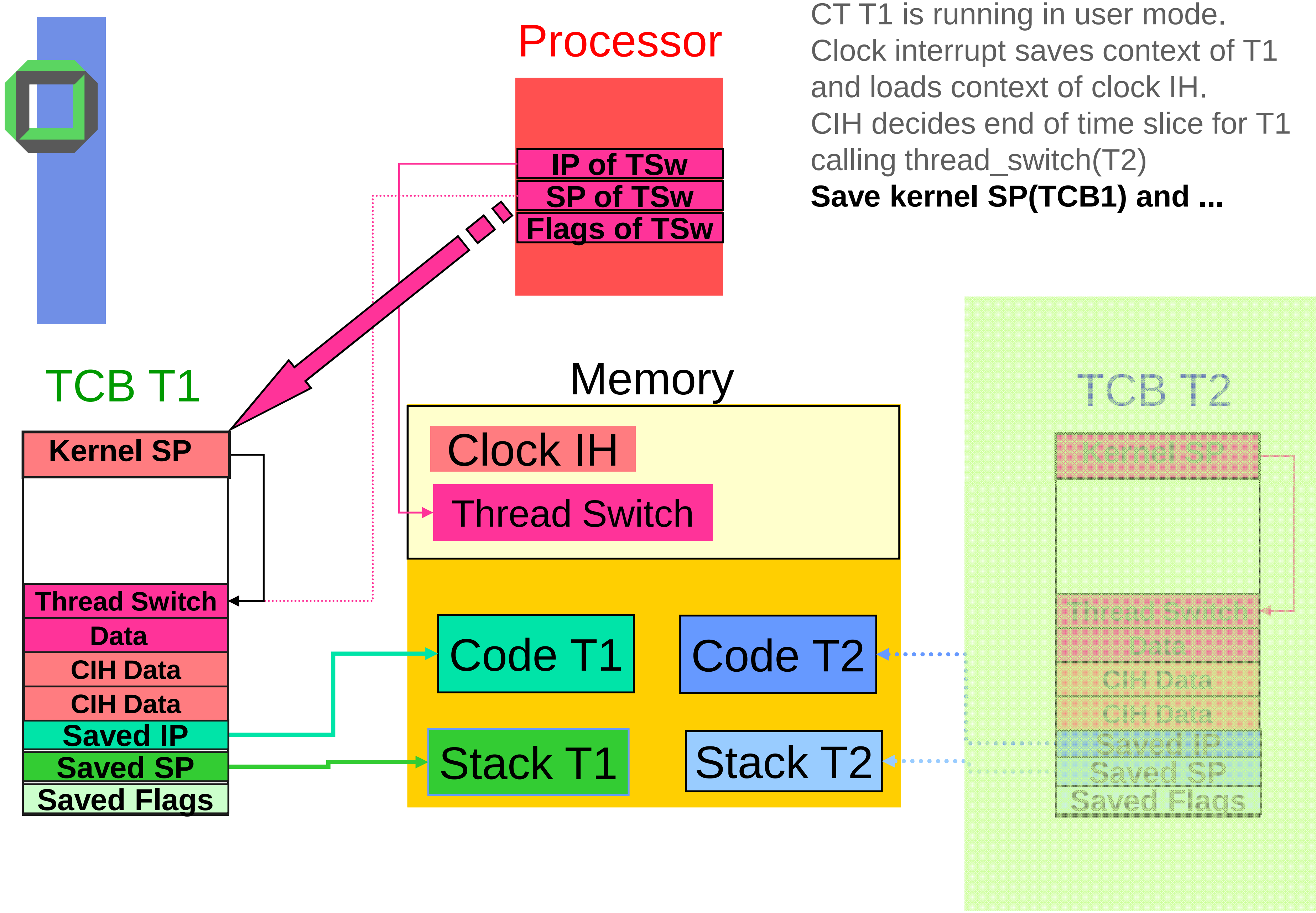


CT T1 is running in user mode.
Clock interrupt saves context of T1
and loads context of clock IH.



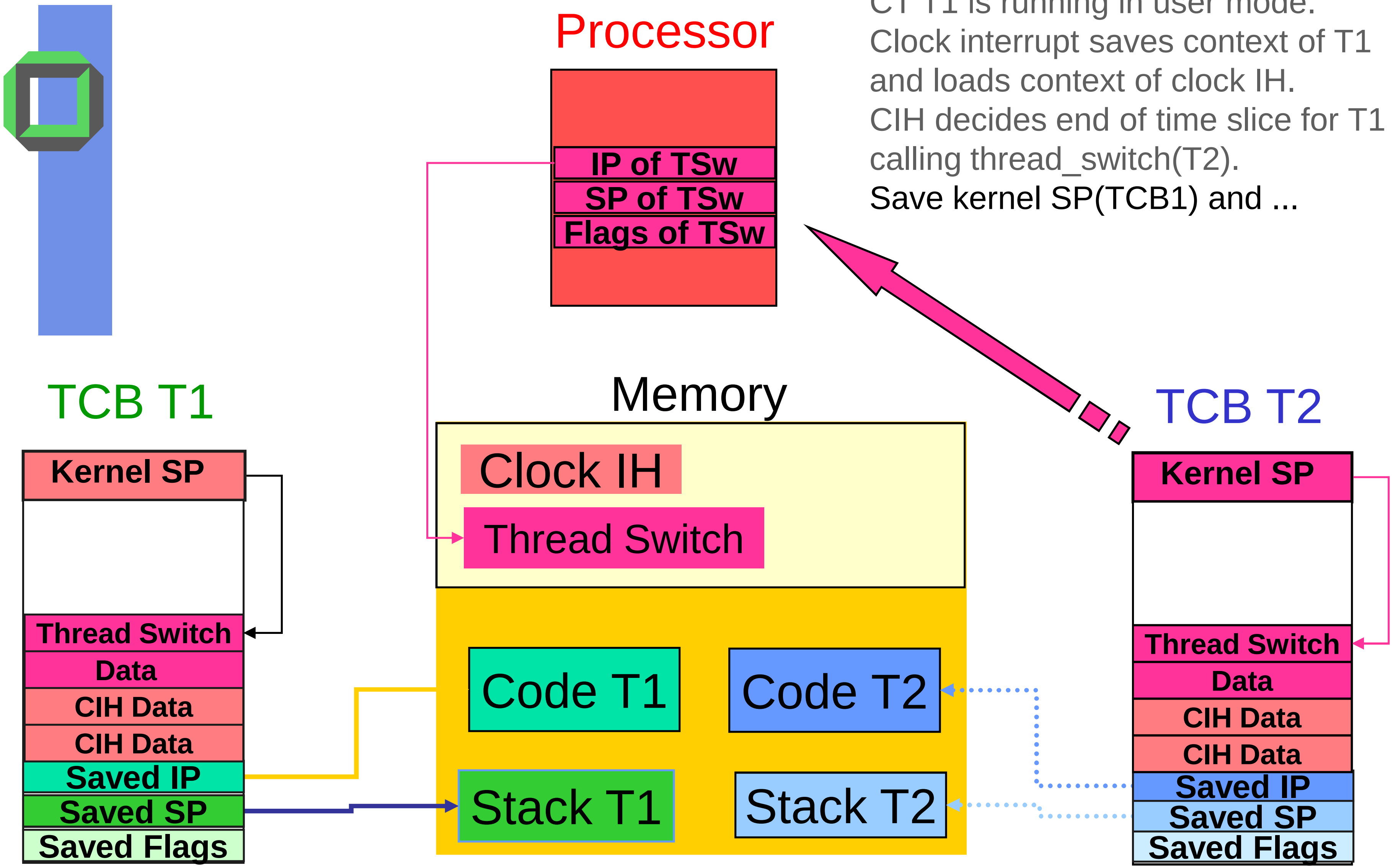
CT T1 is running in user mode.
Clock interrupt saves context of T1 and loads context of clock IH.
CIH decides end of time slice for T1 calling thread_switch(T2)



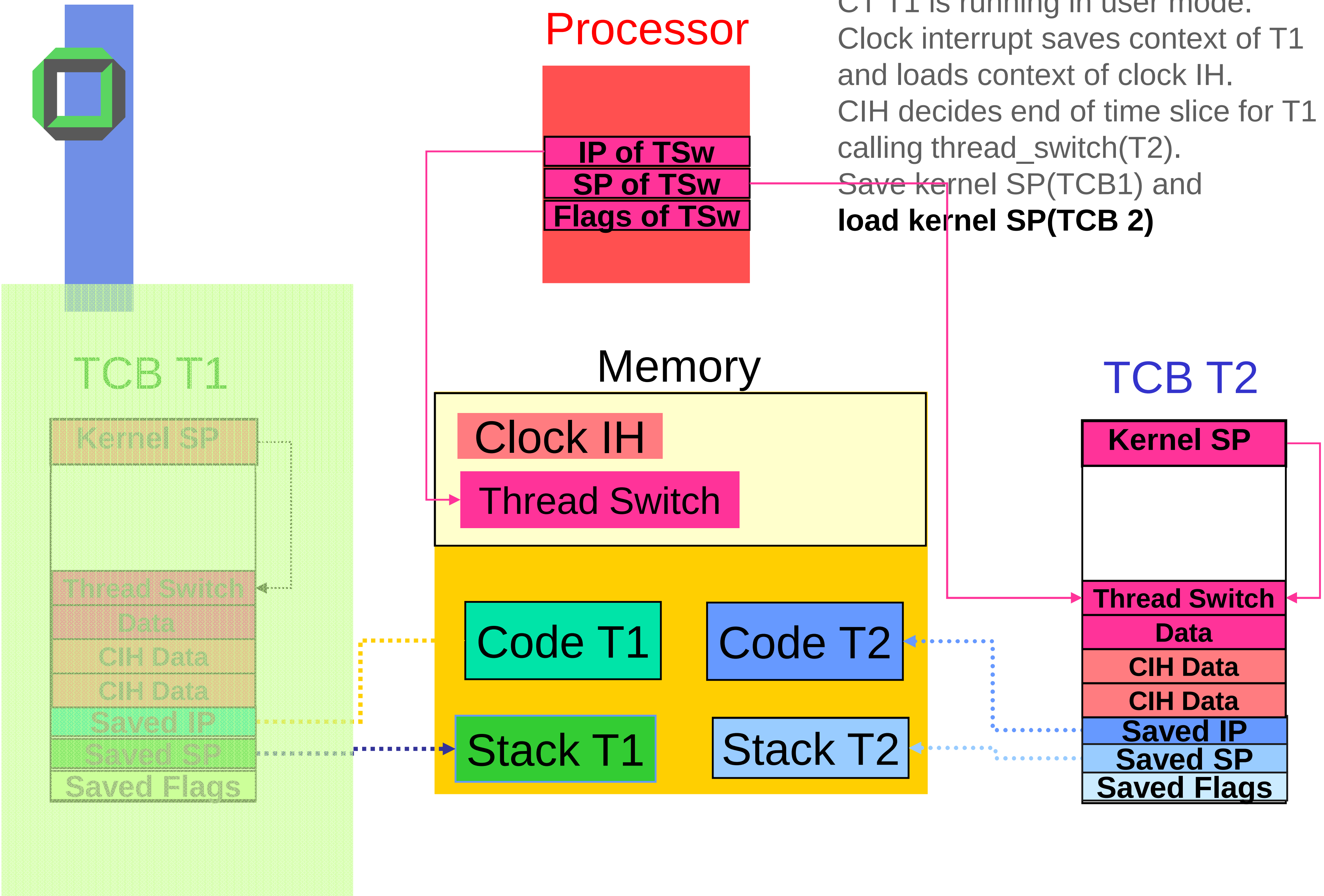


CT T1 is running in user mode.
Clock interrupt saves context of T1 and loads context of clock IH.
CIH decides end of time slice for T1 calling thread_switch(T2)
Save kernel SP(TCB1) and ...

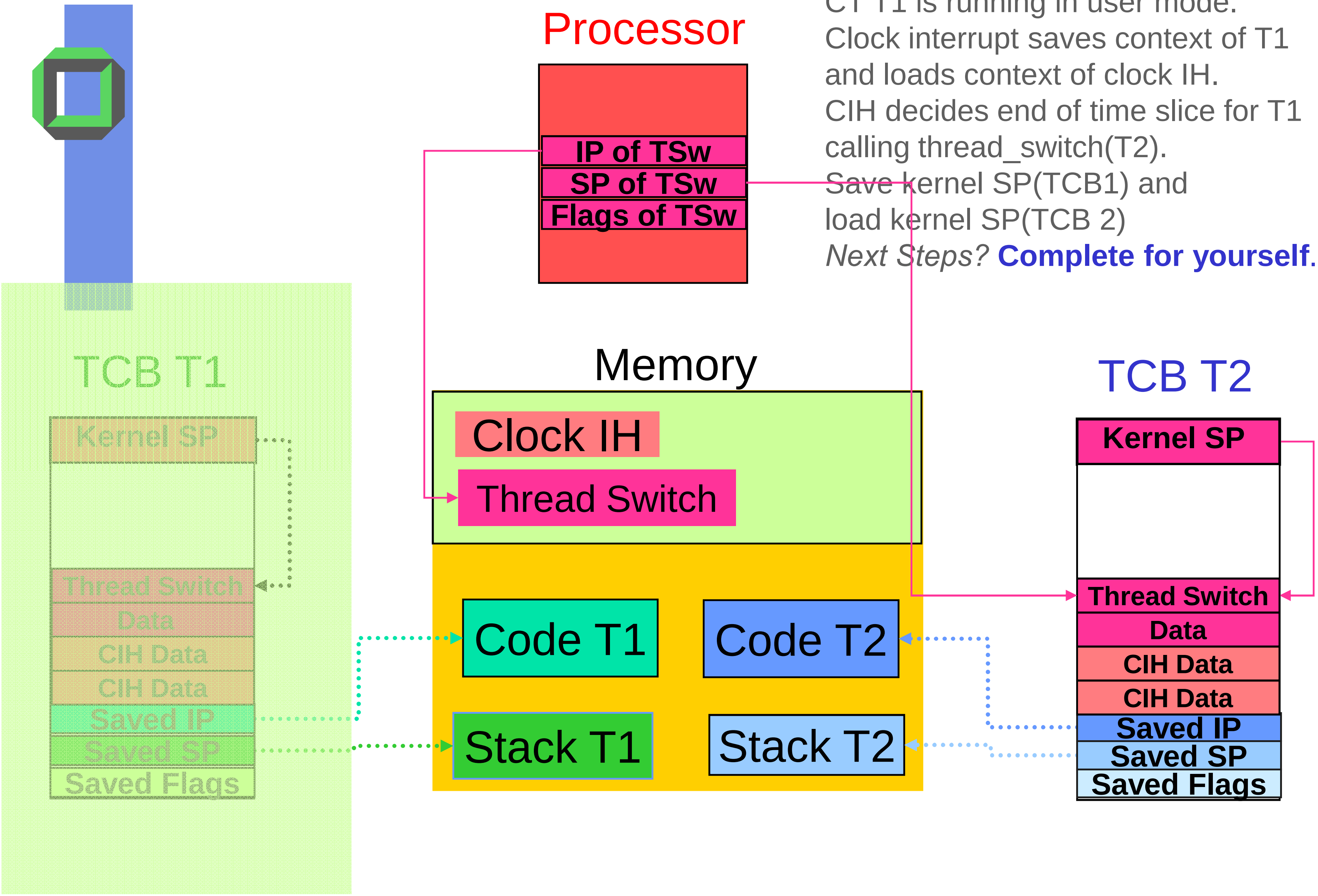
CT T1 is running in user mode.
Clock interrupt saves context of T1 and loads context of clock IH.
CIH decides end of time slice for T1 calling thread_switch(T2).
Save kernel SP(TCB1) and ...



CT T1 is running in user mode.
Clock interrupt saves context of T1 and loads context of clock IH.
CIH decides end of time slice for T1 calling thread_switch(T2).
Save kernel SP(TCB1) and
load kernel SP(TCB 2)



CT T1 is running in user mode.
Clock interrupt saves context of T1 and loads context of clock IH.
CIH decides end of time slice for T1 calling thread_switch(T2).
Save kernel SP(TCB1) and load kernel SP(TCB 2)
Next Steps? **Complete for yourself.**



Additional Design Parameter

Implementation Alternatives

Number of kernel stacks involved:

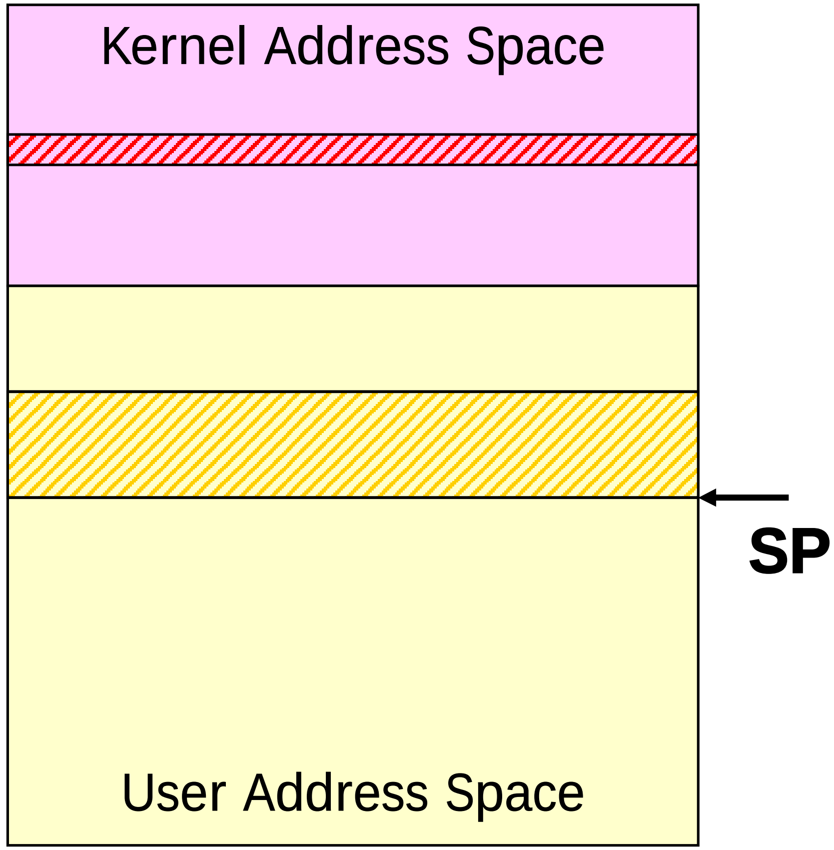
- 1 Kernel stack for all threads
- Each KLT/process has a kernel stack

Discuss carefully!!

Stack Management

- Each process/KLT has two stacks
  - Kernel stack
  - User stack
- Stack pointer changes when entering/exiting the kernel

Why is this necessary?



Open Questions

- *If thread T2 not known in advance ⇒ need for scheduling policy? (see later chapters)*
- *If we know thread T2, where do we get its TCB? (see exercise)*
- If kernel stack is part of TCB ⇒ danger of stack overflow?
- *How to handle thread initiation and termination?*

Remark:

Limitation on kernel stack size is no real problem in practice.

If your system suffers from a kernel stack overflow

⇒ obvious sign of a severe kernel bug

Examine your kernel design and implementation,

before playing around with increasing kernel stack sizes

Open Questions

How to handle thread initiation and termination?

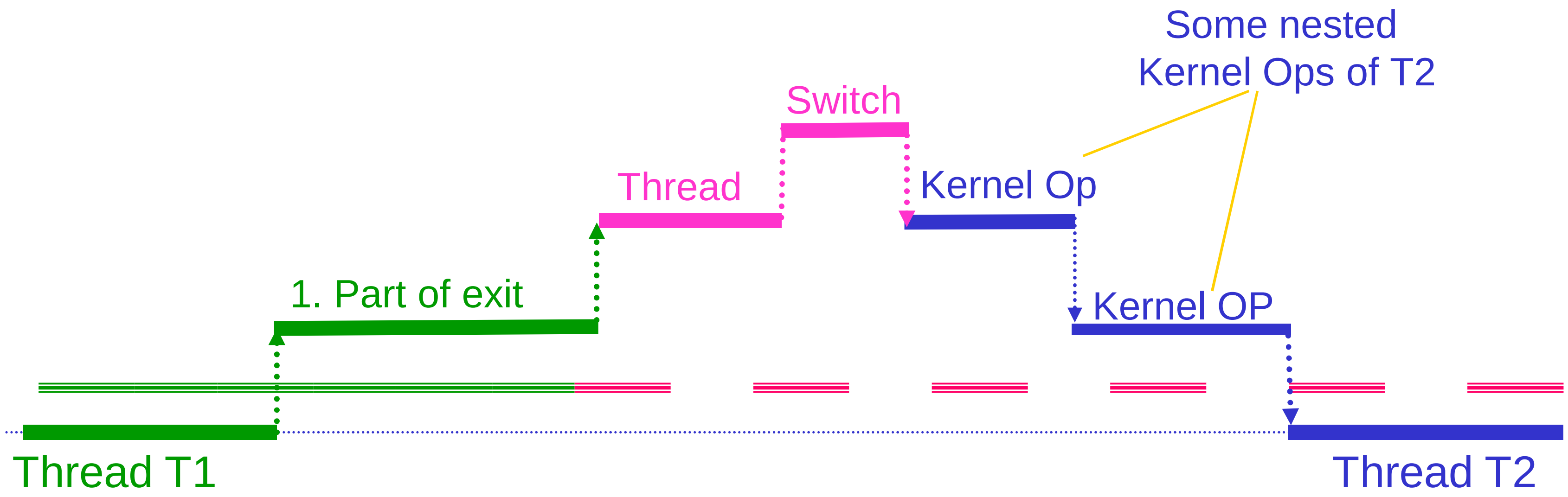
General remark (Principle of Construction):

“Solve special cases with the normal-case solution”

Thread Termination

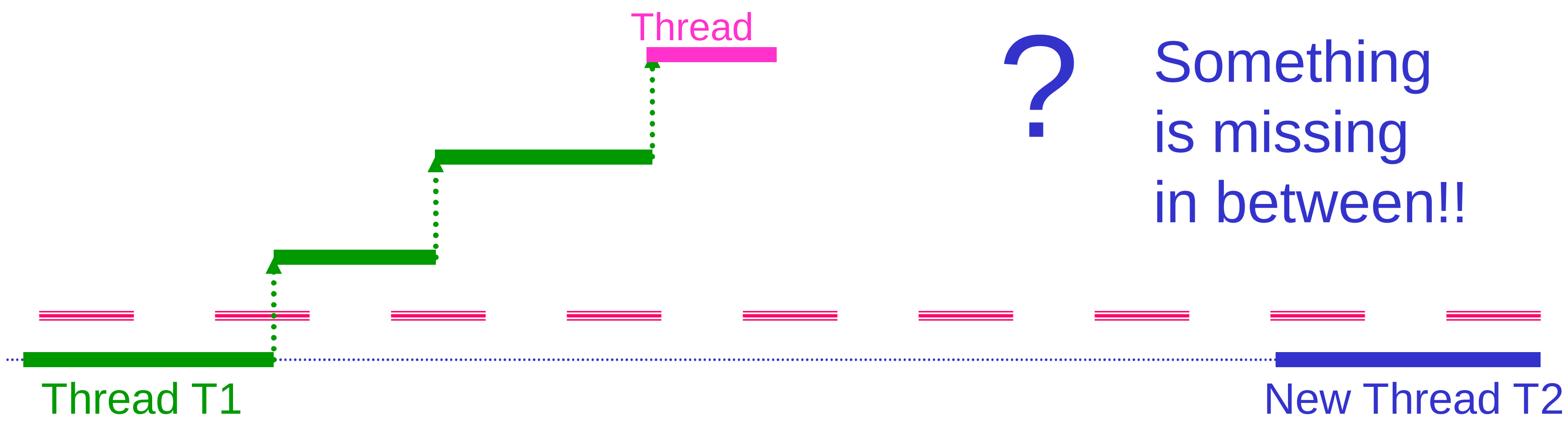
How to terminate a thread?

- Do all necessary work for cleaning up thread's environment
- Switch to another thread, never return to exiting thread
- No additional mechanisms required



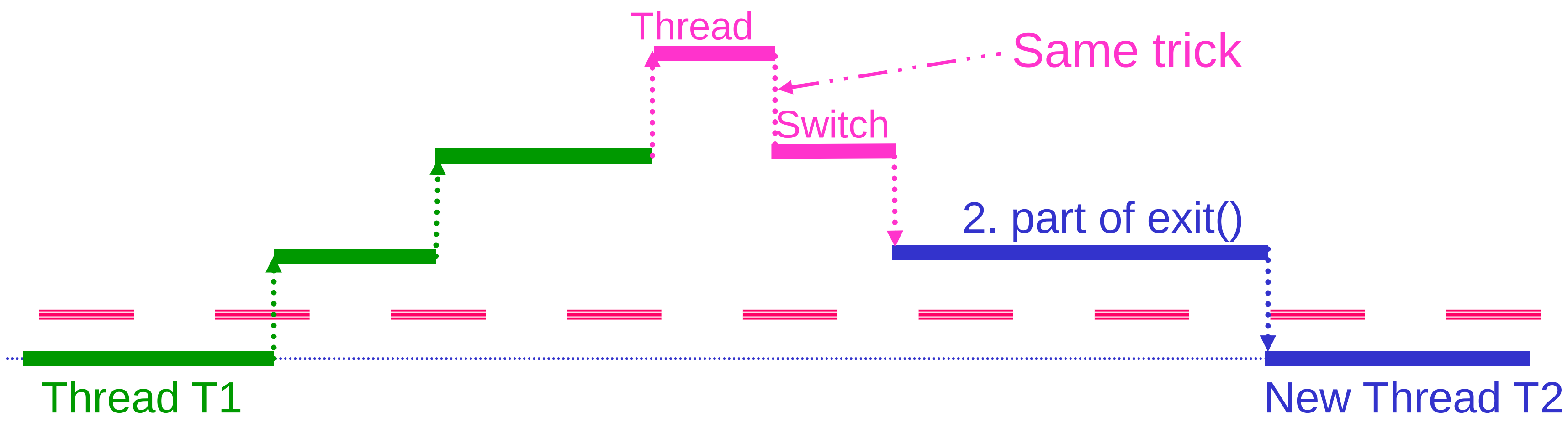
Thread Initialization

What to do, when switching to a brand new thread for the ***very first time****?*



Thread Initialization

- Initialize new thread's (T2) kernel stack with the second part of the thread_switch and the exit function
- Returning from thread_switch leads to second part of system call exit, ⇒
  - "return" to T2 in user mode, and
  - start with the first instruction of T2



Idle CPU Problem

- *What to do, when there is no thread to switch to?*

<u>Solution:</u>

Avoid that situation by introducing an idle thread that is always runnable

<u>Question</u>:

Major properties of an idle thread?

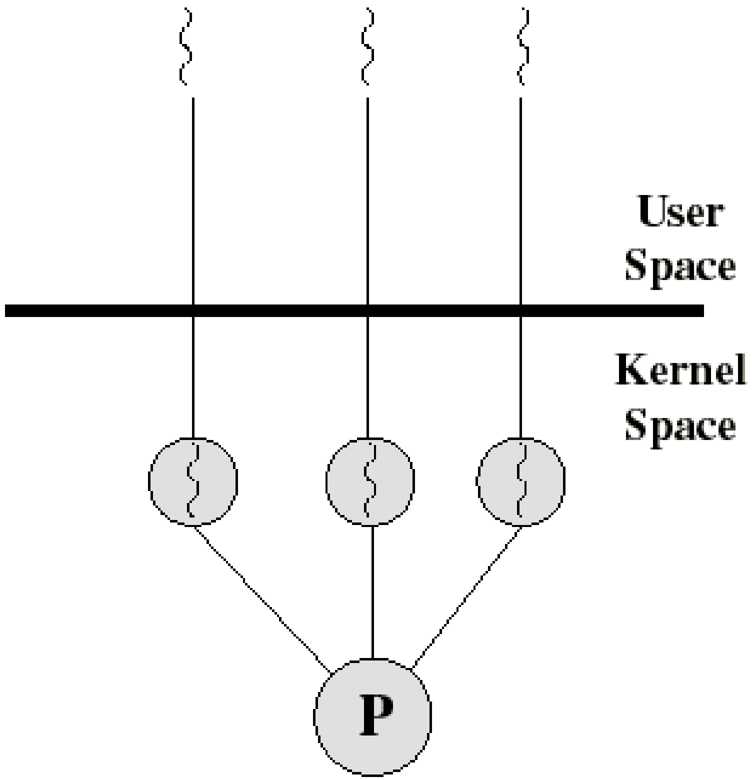
Idle Thread

- *When to install?*
  - *Before booting*
  - *While booting*
  - *After booting*

- *How to guarantee that idle thread is always runnable?*
  - *Avoid any wait events in the idle thread*
    - *...*
    - *...*
    - *...*

Summary: Kernel-Level Threads

- All thread management is done by the kernel
- No thread library, but API to kernel thread facility
- Kernel maintains TCBs for the task and threads.
- Switching between threads requires kernel.
- Scheduling on thread basis



Pros/Cons of KLTs

Advantages:

Kernel can simultaneously schedule threads of same task on different processors

A blocking system call only blocks the calling thread, but no other thread from the same application

Even “kernel” tasks can be multi-threaded

Disadvantages:

Thread switching within same task involves the kernel. We have two additional mode switches per thread switch!!

This can result in a significant slow down!!

Summary

<u>Thread-Switching Environment:</u>

- *Kernel Entry + Mode Switch (User → Kernel)*
- Changing Old Thread State
- Select New Thread (optional)
- **`Thread_Switch`** (context switch)
- Changing New Thread State
- *Kernel Exit + Mode Switch (Kernel → User)*

<u>Remark:</u> (Only needed for kernel-level threads)

Preview

- Thread Control
- Thread Representation
- Thread Switch
- Thread States orthogonal to Task States
- Dispatching of Threads