[...] the purpose of abstraction is not to be vague, but to create a new semantic level in which one can be absolutely precise.
-- *Edsger W. Dijkstra*, "The Humble Programmer" (1972)

Chapter 3

Threads

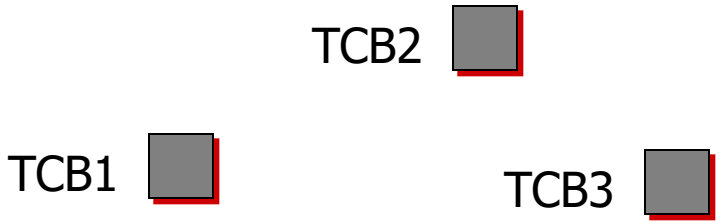
3.1 Thread description

- A process or thread (*more about the distinction later*) is represented by a special data structure, the thread control block (TCB), that contains all relevant information about the thread, e.g.:
  - Thread characteristics:
    - Thread ID, name of program
  - State information:
    - Instruction counter, stack pointer, register contents
  - Management data:
    - Priority, rights, statistics
- In larger systems thousands of threads are possible. That requires efficient management (i.e. suitable data structures).
- Depending on type and usage of OS different solutions are available.

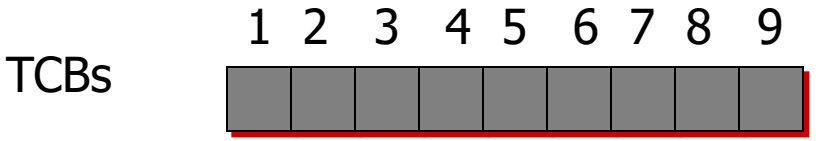
Management of thread control blocks

Freie Universität Berlin

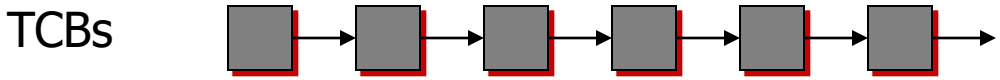
a) Single scalars



b) static long array

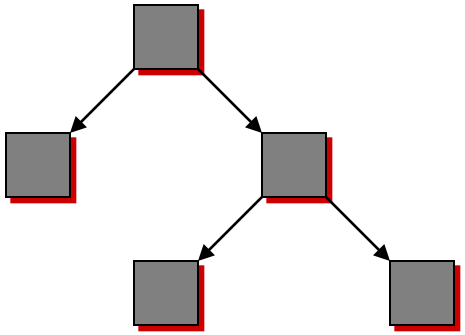


c) Variably long linked list

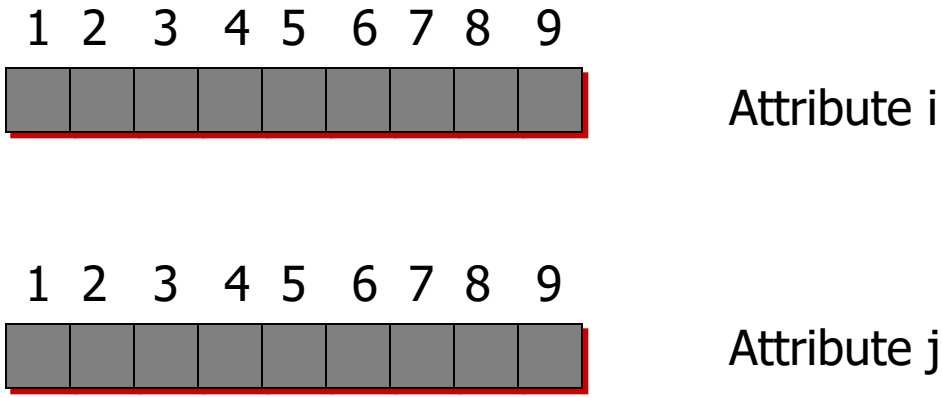


Management of thread control blocks

d) tree



e) inverted table

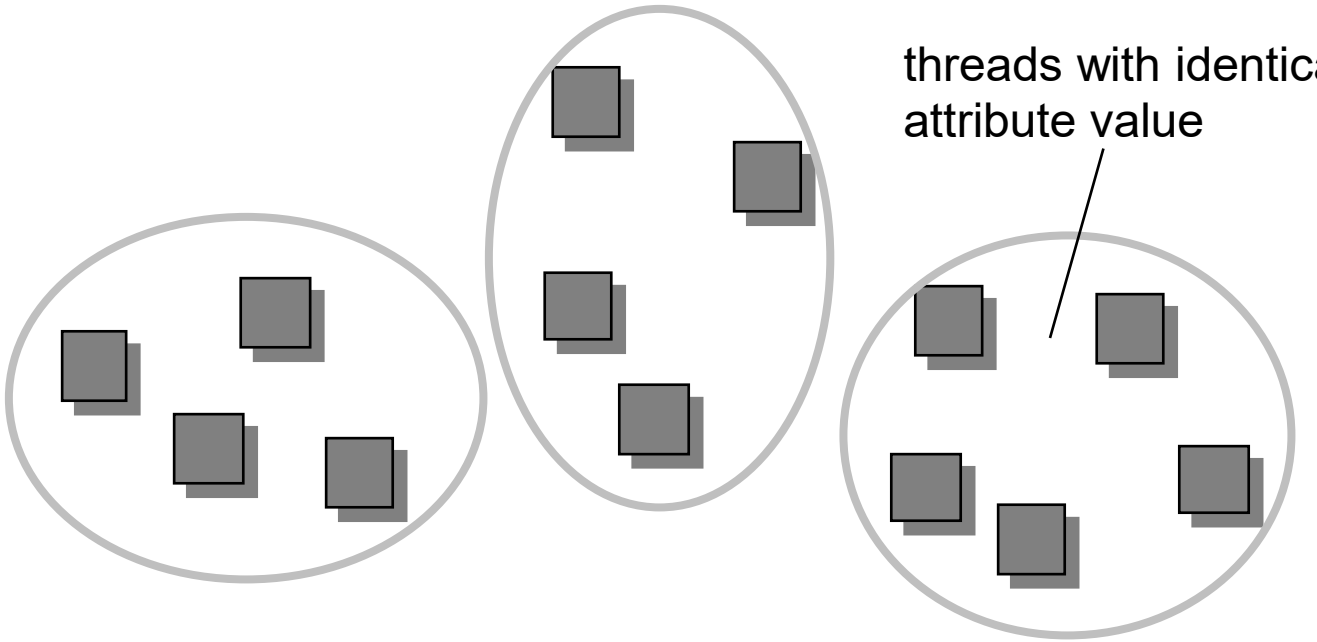


Management of thread control blocks

Freie Universität Berlin

To increase efficiency:

Forming subsets with regard to important attribute values (e.g. thread state, priority)



Static and dynamic systems

- **Static Operating Systems**

  All threads are known in advance and statically defined.
  - Threads are defined "at the desk". The TCBs are declared as program variables.
  - Threads are used for a specific application.
  - The TCBs are generated by a configuration program once.

- **Dynamic Operating Systems**

  The threads are created and deleted by kernel operations.

```
create_thread (id, initial values)
  // create thread control block
  // initialization of thread
delete_thread (id, final values)
  // return of final values
  // deletion of control block
```

Threads and Address Spaces

- A (logical) address space of a thread is the universe of its valid addresses, which it can access.
- Modern processors enable not only relative addressing (basis register), but also provide a memory management unit (MMU) for address translation.
- That allows to have an arbitrary number of *logical address spaces that* automatically can be mapped to the physical address space.
- That also leads to mutual protection of address spaces.
- Address spaces are independent (orthogonal) to threads.
- Each thread needs an address space at any time but several relations are possible:
  - A thread owns exactly one private address space (Unix process).
  - Several threads share an address space (Threads).
  - A thread switches from one address space to another.

Terminology

Concerning the terminology care must be taken in the relevant literature:

- A process (*task*) is mostly considered as a Unix-type process with a private address space.
- Most operating systems (including current UNIX variants) offer the possibility to run several processes in a shared address space.
- They are called *lightweight processes* or *threads*.
- Today's Unix variants (e.g. Linux, Solaris, …) offer the original Unix processes (tasks), that may consist of many threads.
- A Unix process is therefore an address space, that contains at least one *thread*.
- For Windows the same holds.
- A group of threads in a shared address space is sometimes also called *team* (System V) or *actor* (Chorus).
- **In this lecture course we use the term "thread".**

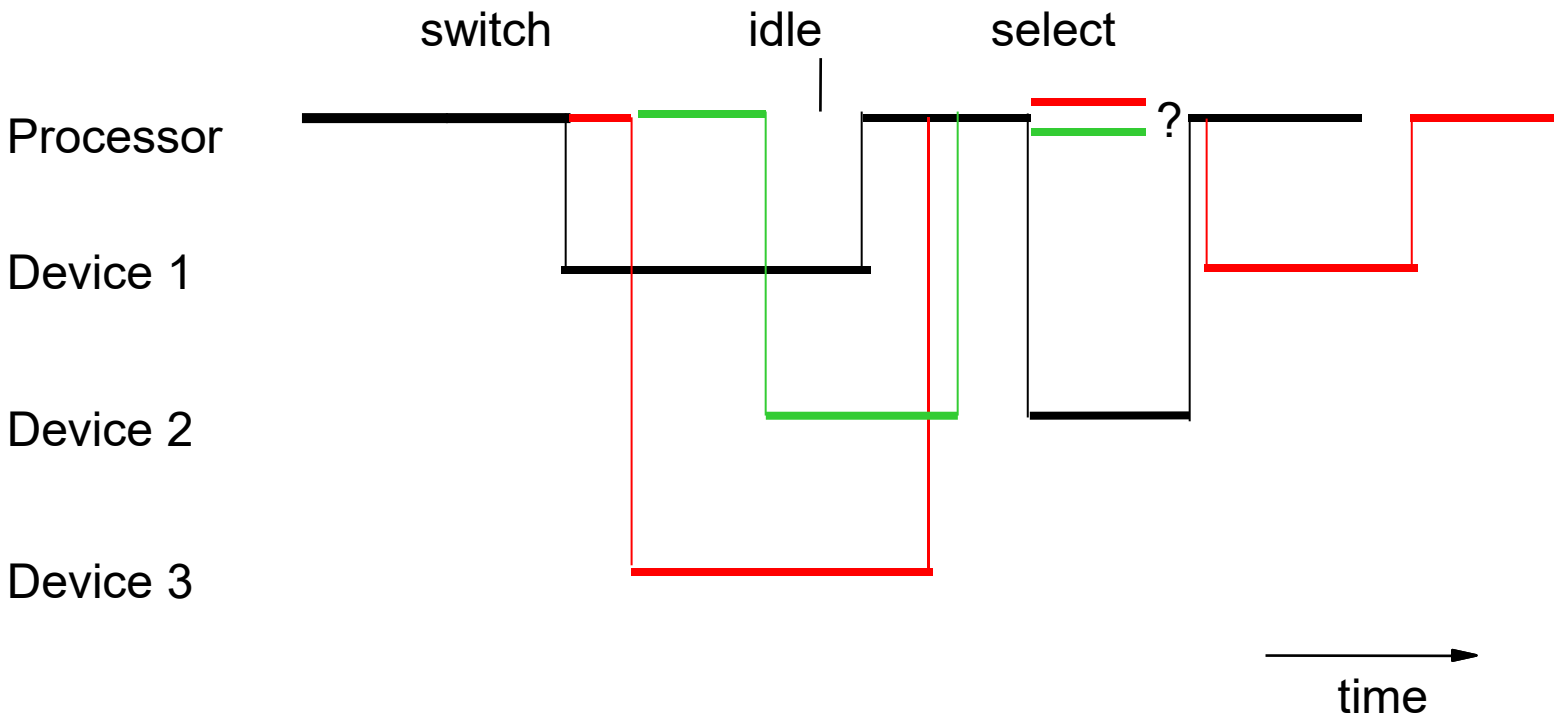
Overview of terms used

Operating system	Unit of execution	Superior unit
Mach	thread	task
Chorus	thread	actor
Mayflower	lightweight process	domain
V	thread	team
Amoeba	thread	cluster
Cosy	process	address space
Solaris	thread	process
NT	thread	process

3.2 Thread switch

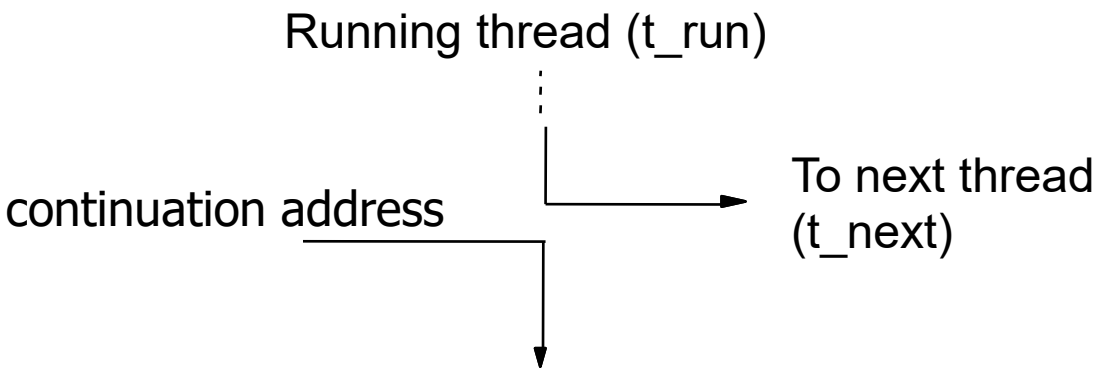
Thread switching

- Thread switch means that the processor stops the execution of the current thread and continues with the execution of another thread.
- Thread switch is the transition from one instruction sequence to another one.

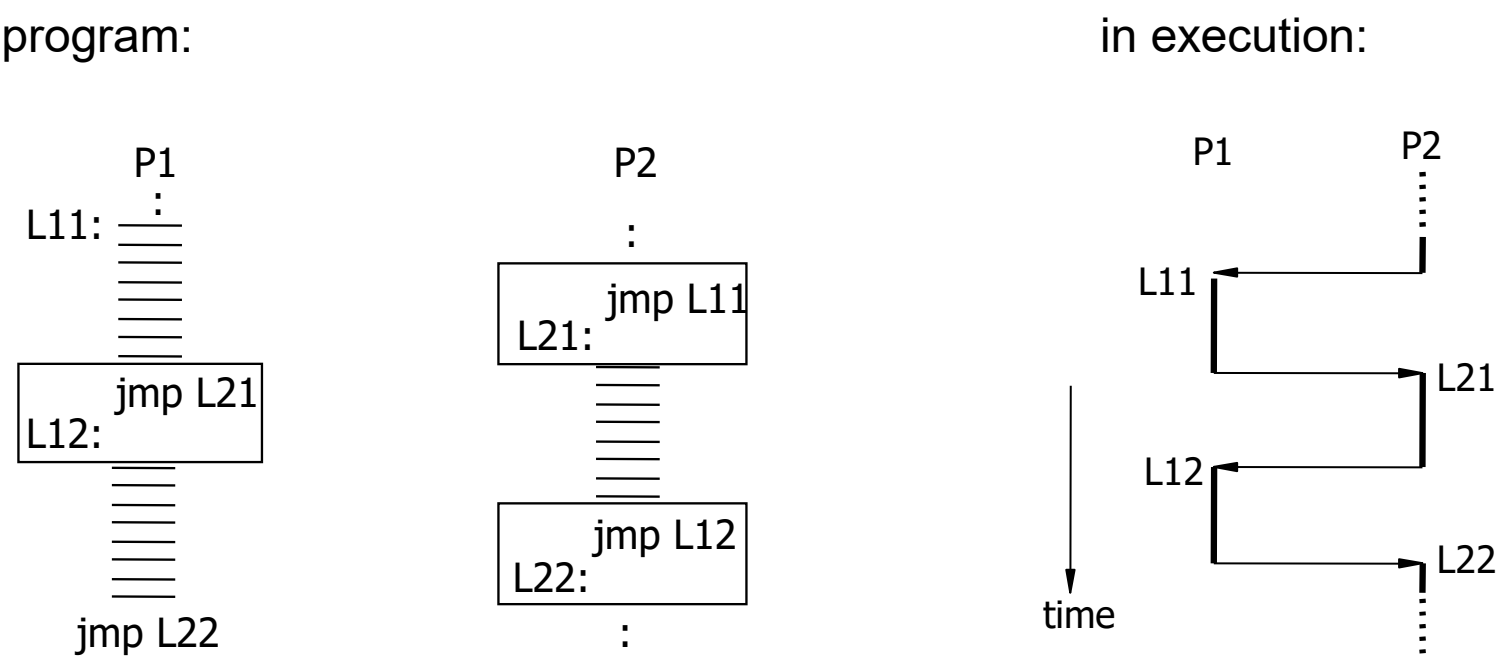


Switching by jumping

- In the most simple case the switch can be programmed statically and directly in the threads.
- It means the we insert a jump instruction that jumps into another thread.
- To continue the work at the very point where the thread was left, we have to memorize the position where we have to return.
- A switching point therefore consists of at least
  - continuation address (where did we interrupt the work)
  - jump instruction (where do we want to continue)



Switching by jumping



- For certain application areas (real-time systems) the time needed to switch from one thread to another is an important quality measure.
- The thread switch should be realized efficiently.
- The jump is the "minimal solution".

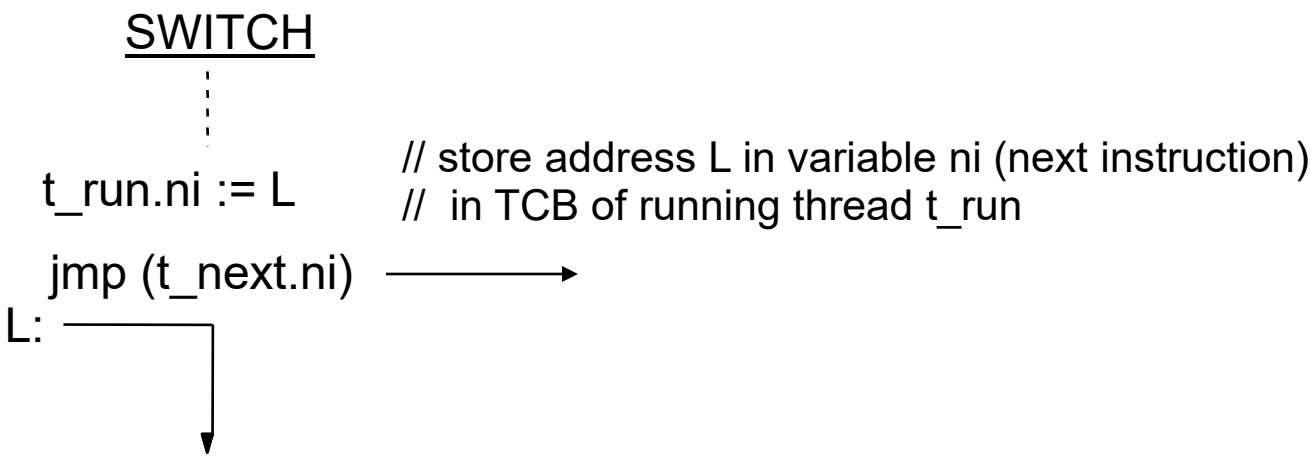
Switching more general

- Switching by direct jump is very inflexible and applicable only in very special cases.

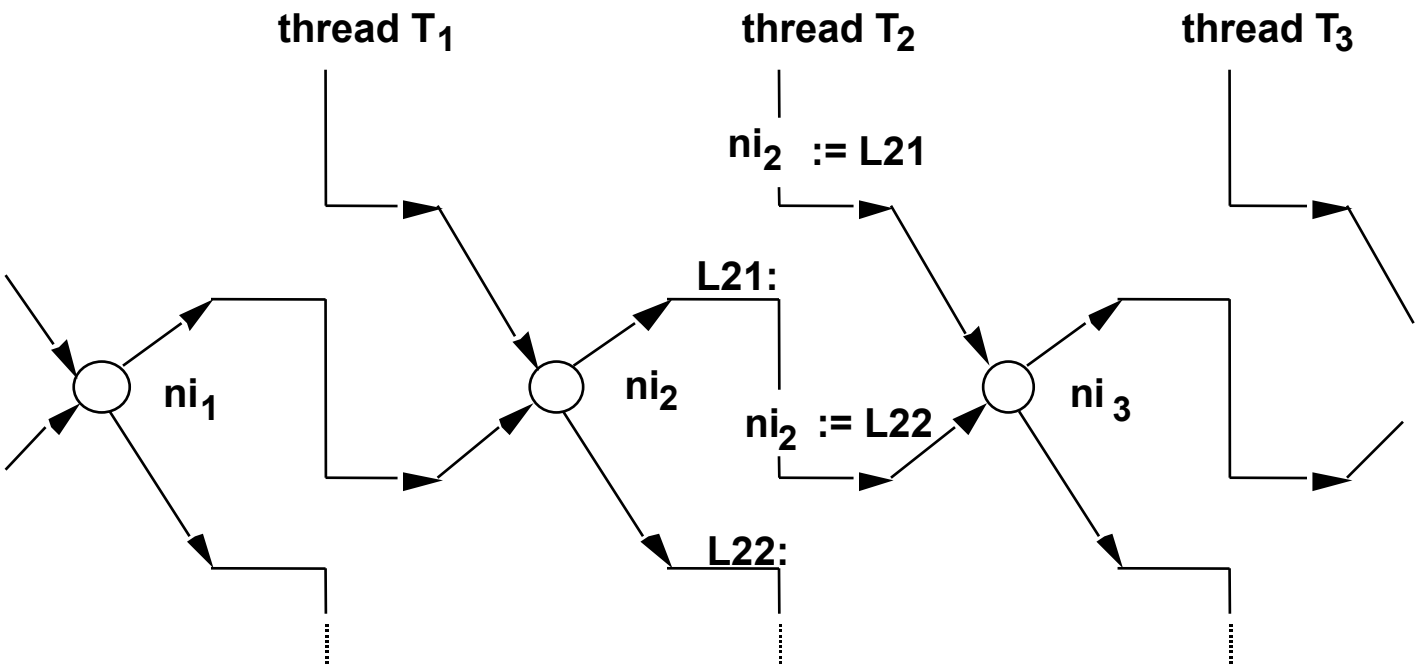
- In general the thread switch will be more costly since
  - we do not know, from where we return to the interrupted thread (memorizing the continuation address),
  - the next thread, to which we switch, is not always the same (selection of next thread),
  - the processor contains essential parts of the thread description that must not get lost (register reload).

Memorizing continuation address

Before switching to the new thread, we store the address of the next instruction to be executed in a dedicated variable ni (next instruction) of thread control block.



Switching with variable continuation addresses

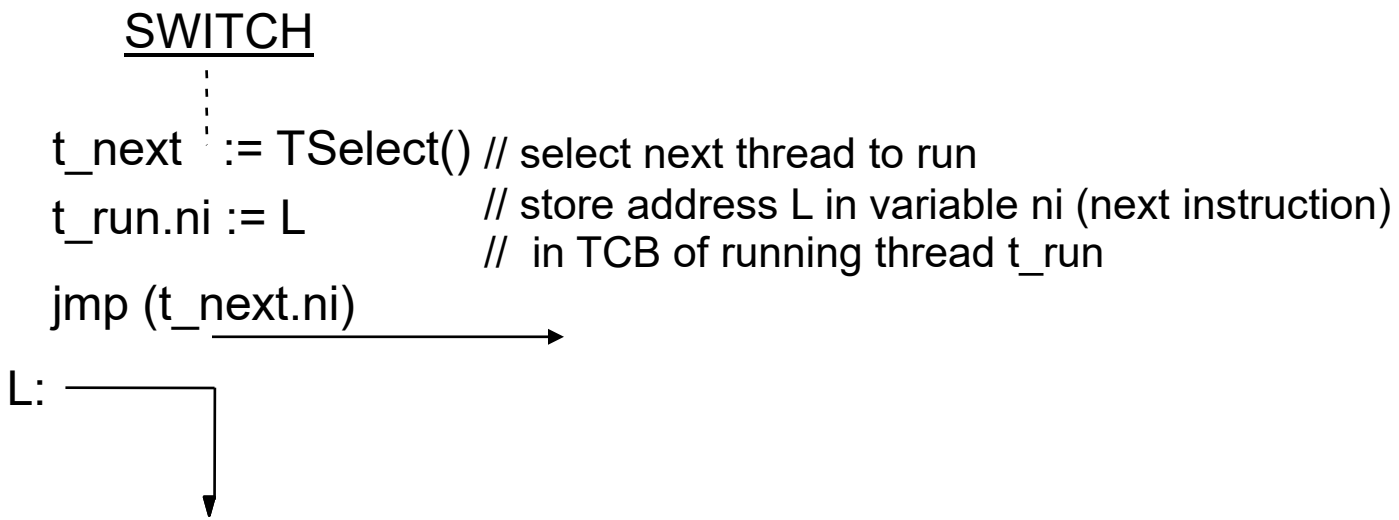


Selection of next thread

- Up to now we assumed that we know the next thread to which we want to switch.
- However, in most cases this target thread is not constant but will be determined at the very time of switching:
- Criteria for selection:
  - number of thread (cyclic switching)
  - order of arrival
  - priority (urgency)
    - constant
    - dynamic
- The selection of next thread influences the distribution of the processor's computing capacity to the threads.

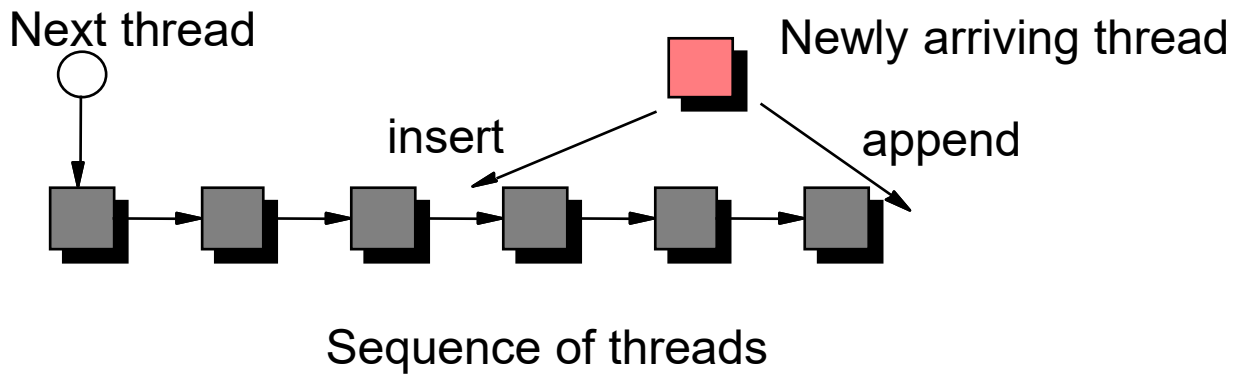
Selection of next thread

- Let TSelect() be a function that selects the next thread according to some criteria.



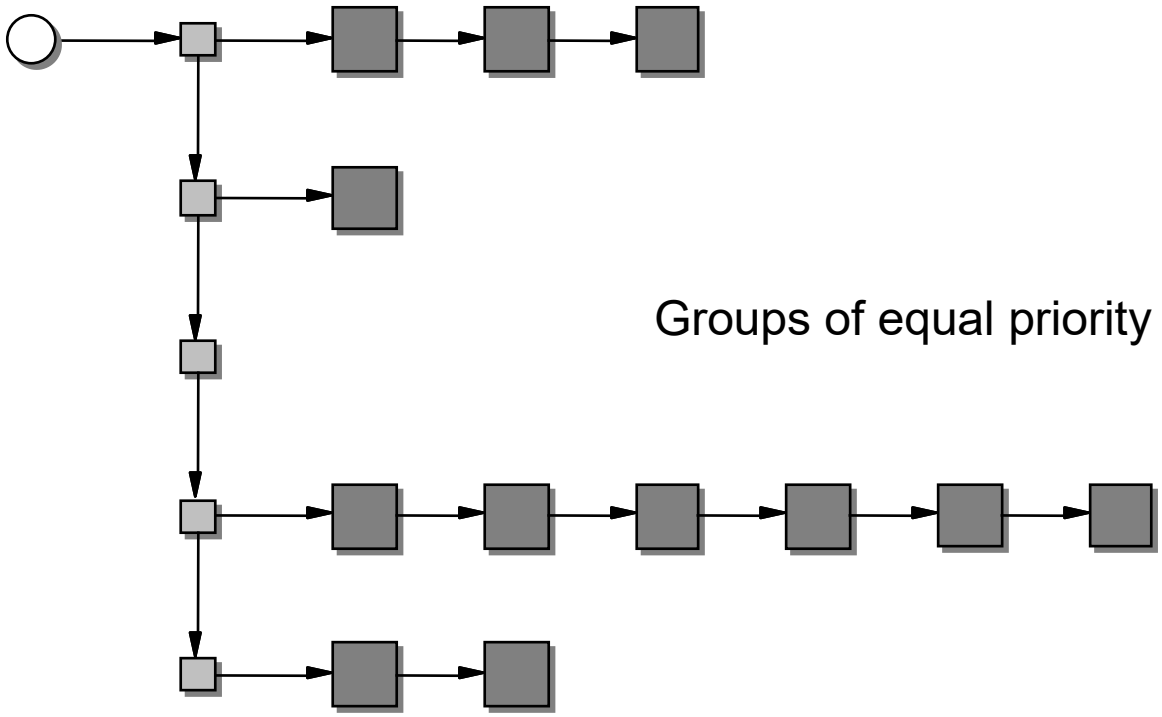
Selection of next threads

- The selection problem can be solved such that the threads are already totally ordered with regard to the execution order.
- The first thread (t_next) in this sequence is selected.
- New arriving threads will be inserted into that sequence according to the chosen order.



Selection of next threads

- When using priorities, the order can be organized in two dimensions



Processor registers

- Threads use arithmetic registers of the processor to store intermediate results.
- If we simply jump to a next thread, their content will be lost (overwritten).
- The "switch by jump" solution can only be used when
  - the contents of the registers will no longer be needed and
  - the new thread does not expect valid register contents

Thread context

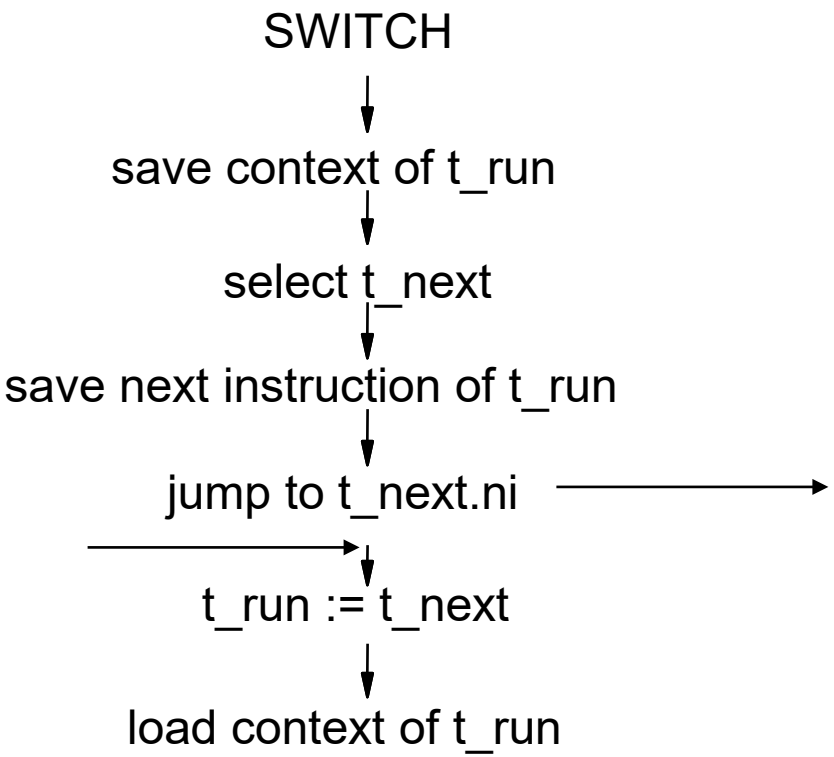
- Besides the instruction pointer the processor keeps much more thread specific information in its registers:
  - Contents of arithmetic registers, index registers, processor state etc., that represent the **state of the execution** of the program, i.e. of the thread.
  - Contents of address registers, segment tables, interrupt masks, access control information etc. that make up the thread‘s execution environment.
- Altogether, i.e. the complete thread specific information stored in the processor is called **thread context**.
- This thread context has to be saved as part of switching and restored when the thread is resumed.
- Data that is constant and available in the TCB, does not need to be saved.

Context switch

- Context switch is the most time consuming part of thread switch.
- To speed it up, the processor hardware can give some support:
  - by special instructions that allow storing the complete set of registers to the memory in one instruction (and also restoring).
  - by providing several sets of registers (e.g. 8) on the processor, such that at thread switch only the register needs to be saved that indicates the number of the currently valid register set.
- Thread switch can be comparably fast, if only the arithmetic registers need to saved and reloaded while the addressing environment remains constant (thread switch within an address space, lightweight threads, *threads*).

Switch
(as open instruction sequence)

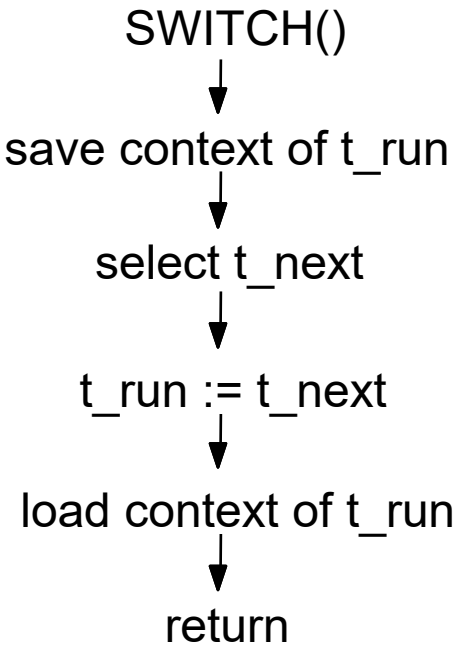
Switching now has the following form:



This sequence can be inserted in the thread‘s program at all places where a thread switch should to take place.

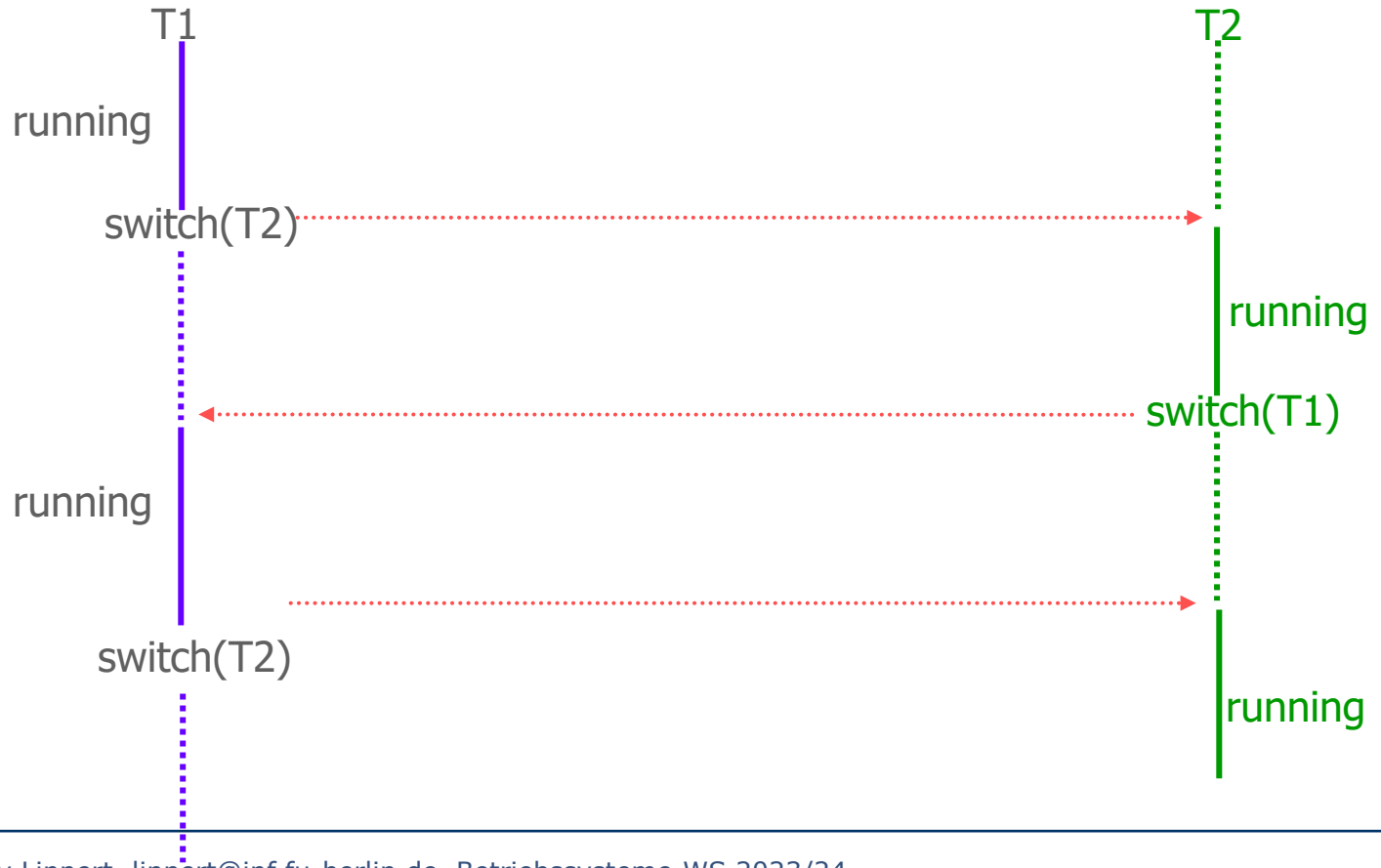
Switching as a subroutine

- If there are many places where switching takes place, it is worthwhile to organize switching as a subroutine.
- If all thread switches are carried out using the same switching subroutine, the saving of the continuation address (ni) and the jump to the next thread can be omitted.
- This is already done as part of the subroutine call and the return operation, respectively.
- This subroutine call is unusual insofar as one thread makes the call but another thread will return from that call.



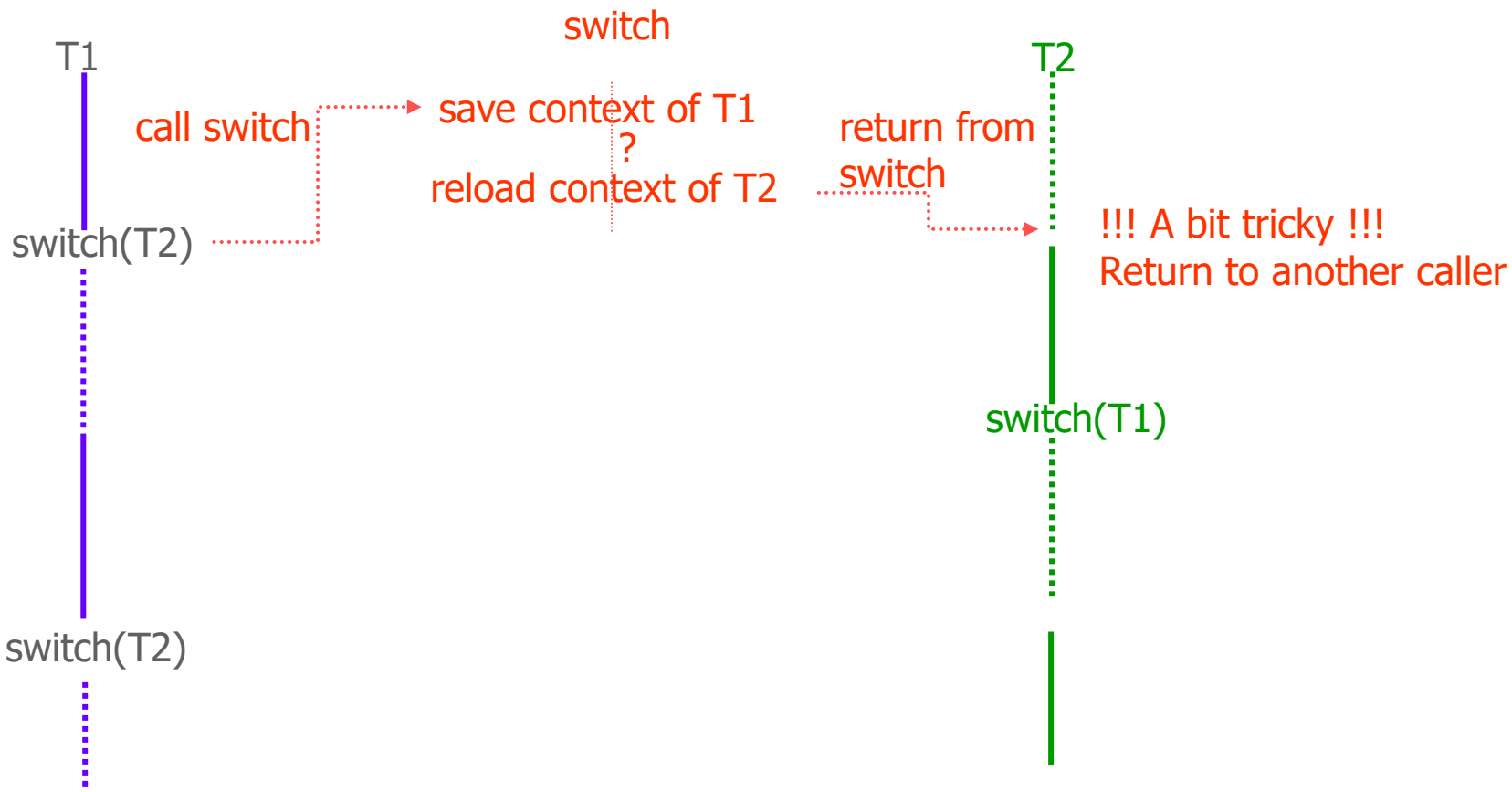
Thread Control: Cooperative Scheduling

Assumption: Given 2 CPU- bound threads T1 and T2,
1 CPU and no interference with peripherals.
Any dispatching is done cooperatively via switch()

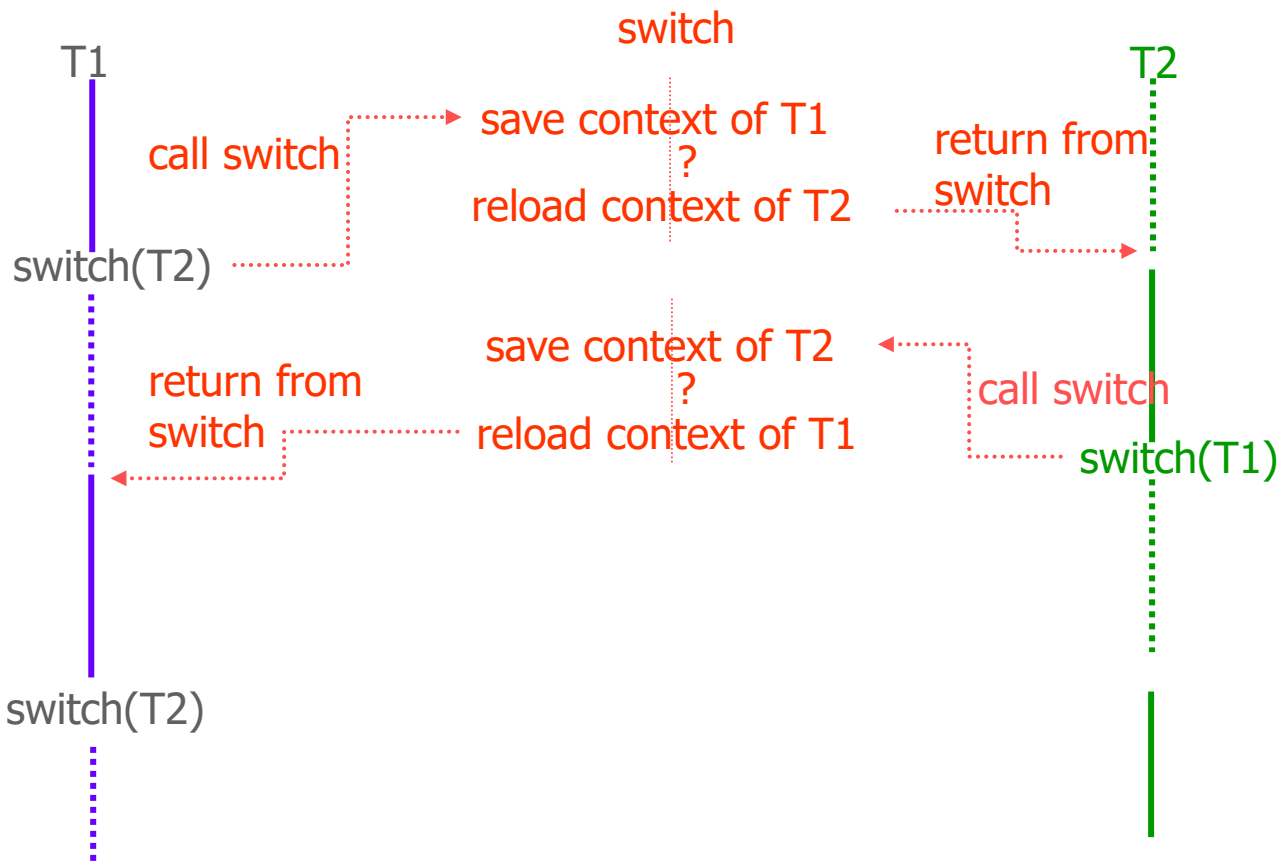


Freie Universität Berlin

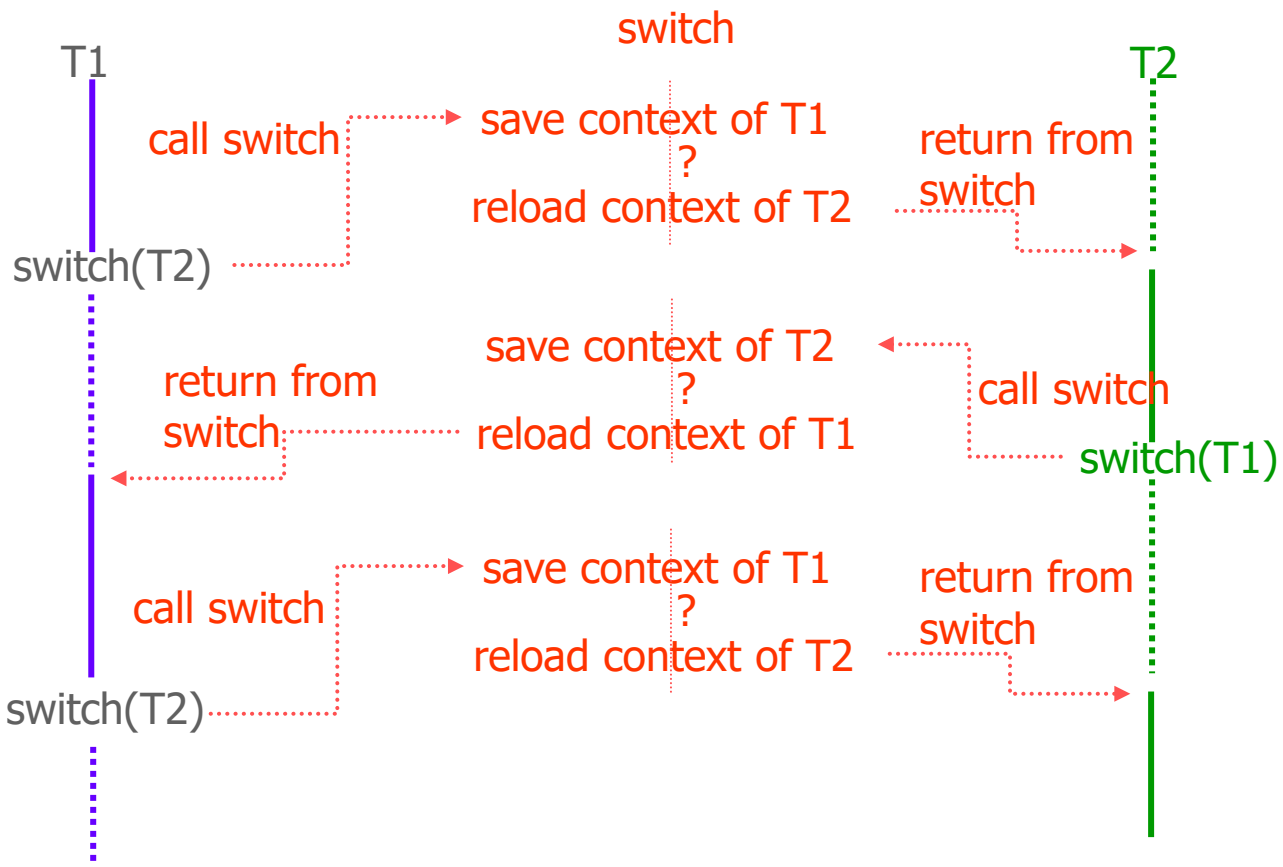
Simplified Calling of "Procedure" switch()



Simplified Calling of "Procedure" switch()



Simplified Calling of "Procedure" switch()



Freie Universität Berlin

Simplified Implementation of "Procedure" switch

```
procedure switch(NT:thread)
begin
…
save context of CT
… ? …
load context of NT
…
return to NT
end
```

This Part of switch still runs under control of CT

This Part of switch already runs under control of NT

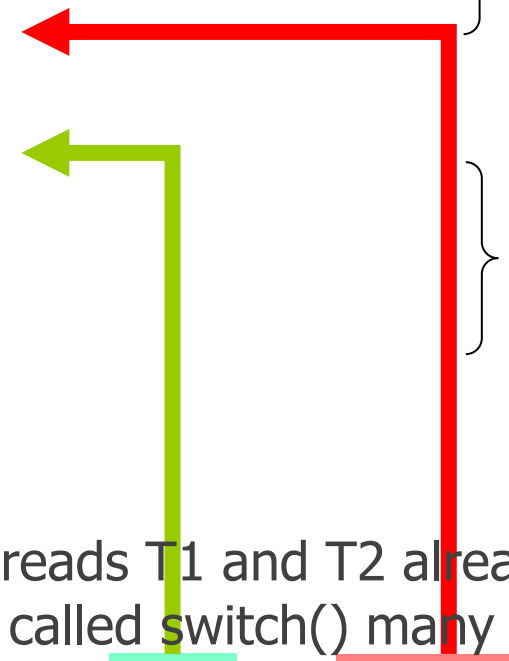
Assumption: Both threads T1 and T2 already have called switch() many times before.

Simplified Implementation of "Procedure" switch

```
procedure switch(NT:thread)
begin
…
save context of CT
… ? …
load context of NT
…
return to NT
end
```

This Part of switch still runs under control of CT

This Part of switch already runs under control of NT



Assumption: Both threads T1 and T2 already have called switch() many times before.

Corollary: Each thread *gets* and *gives up* control within `switch` code at exactly the same point.

Thread Control

Simplified Implementation of "Procedure" switch

```
procedure switch(NT:thread)
begin
…
save context of CT
… ? …
load context of NT
…
return to NT
end
```

This Part of switch still runs under control of CT

How to solve this problem?

This Part of switch already runs under control of NT

Assumption: Both threads T1 and T2 already have called switch() many times before.

Corollary: Each thread *gives up* and *gets* control within **`switch`** code at exactly the same point.

Thread Control

Simplified Implementation of "Procedure" switch

```
procedure switch(NT:thread)
begin
…
save context of CT
CT.sp := SP, SP := NT.sp
load context of NT
…
return to NT
end
```

This Part of switch still runs under control of CT

Change stack pointer

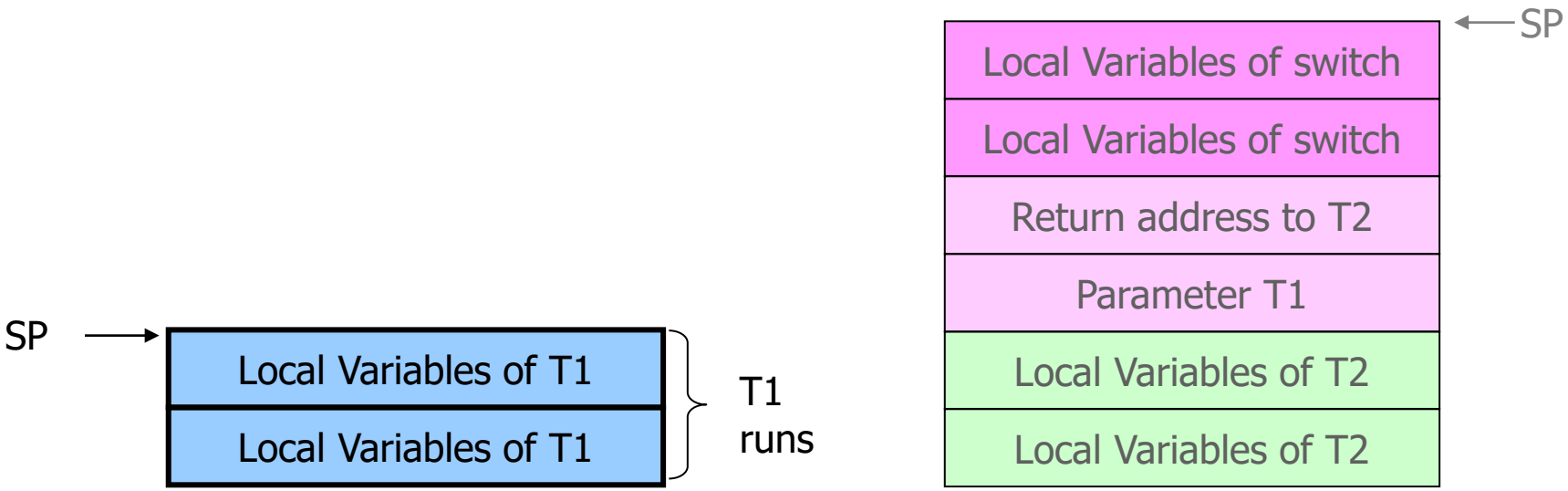
This Part of switch already runs under control of NT

Assumption: Both threads T1 and T2 already have called switch() many times.

Corollary: Each thread *gets* and *gives up* control within `switch` code at exactly the same point.

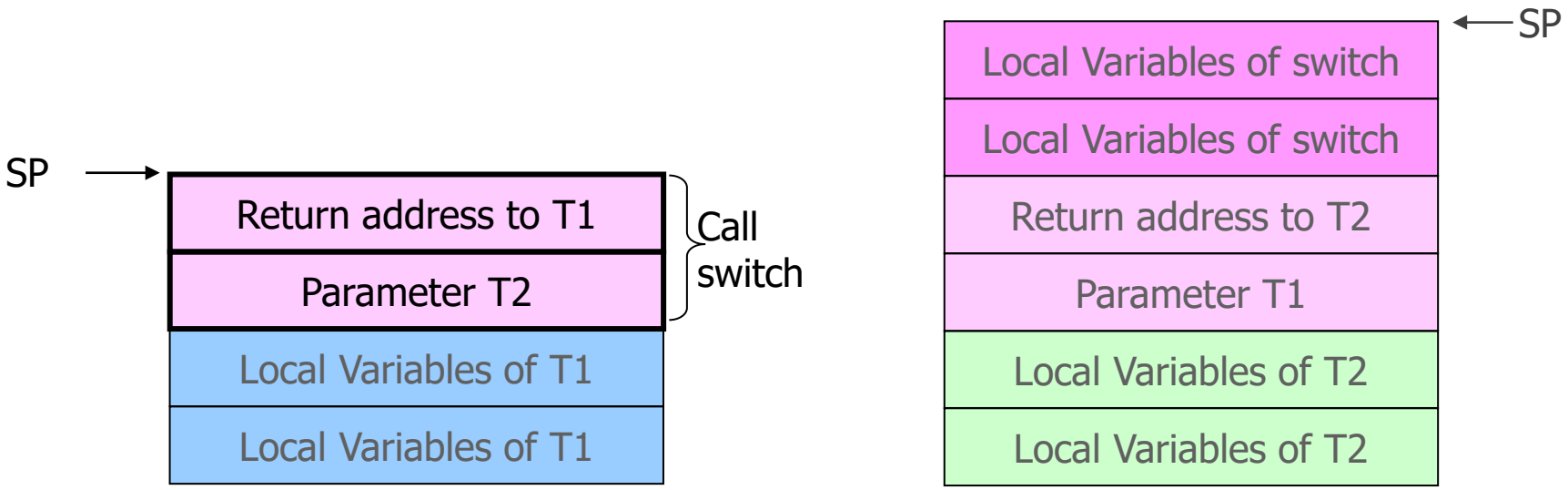
Freie Universität Berlin

Stack Contents during simplified switch

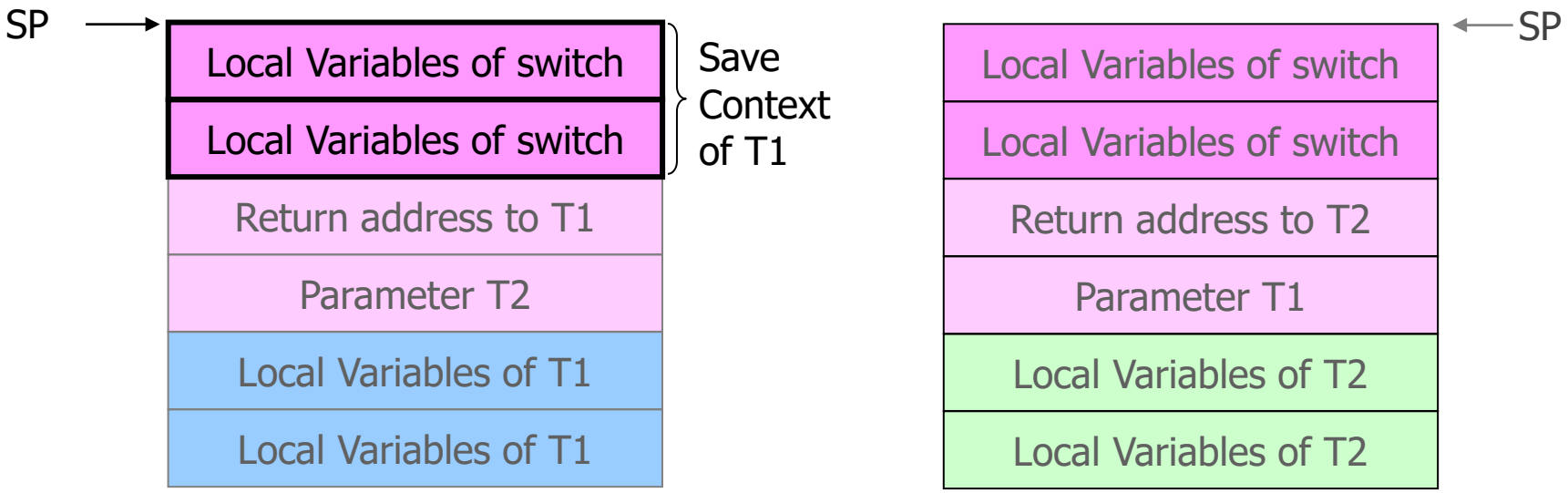


Freie Universität Berlin

Stack Contents during simplified switch



Stack Contents during simplified switch

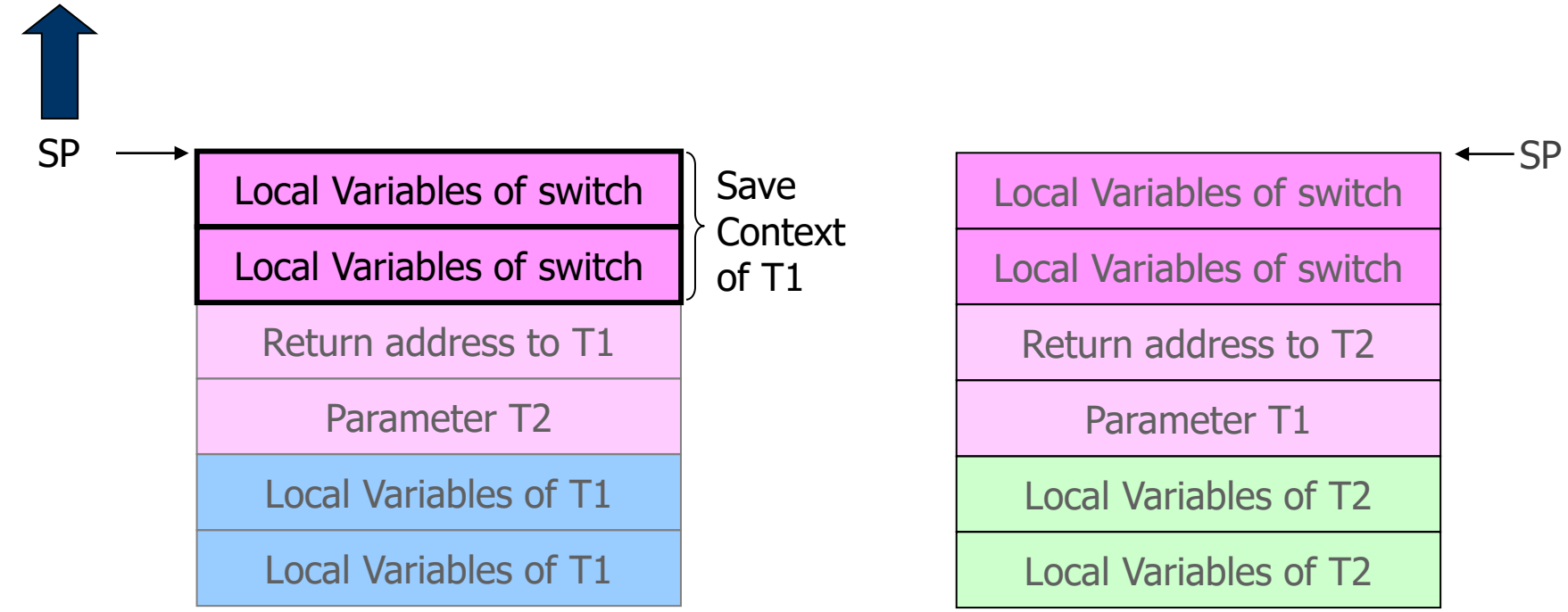


Thread Control

Stack Contents during simplified switch

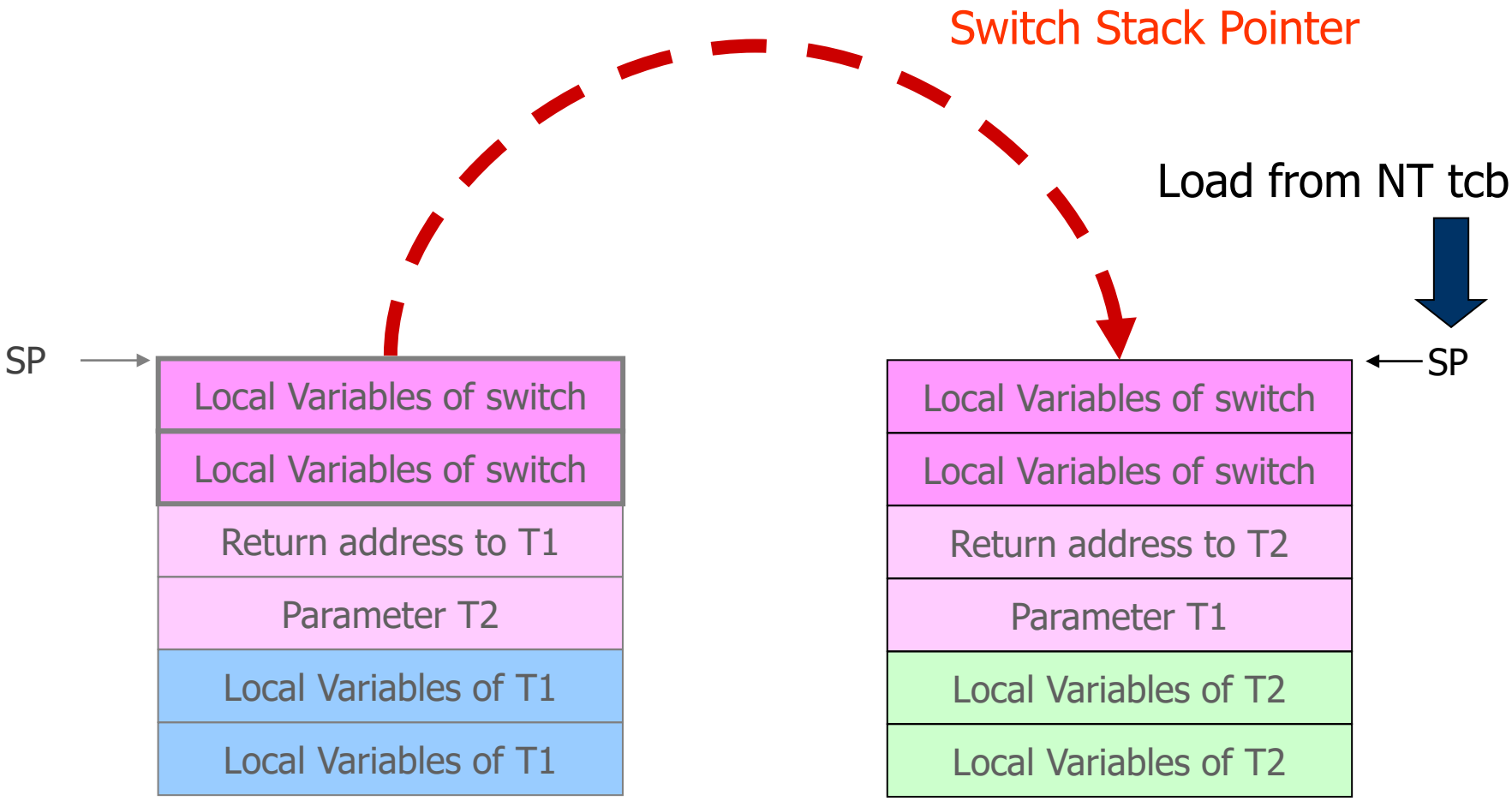
Switch Stack Pointer

Save to CT tcb

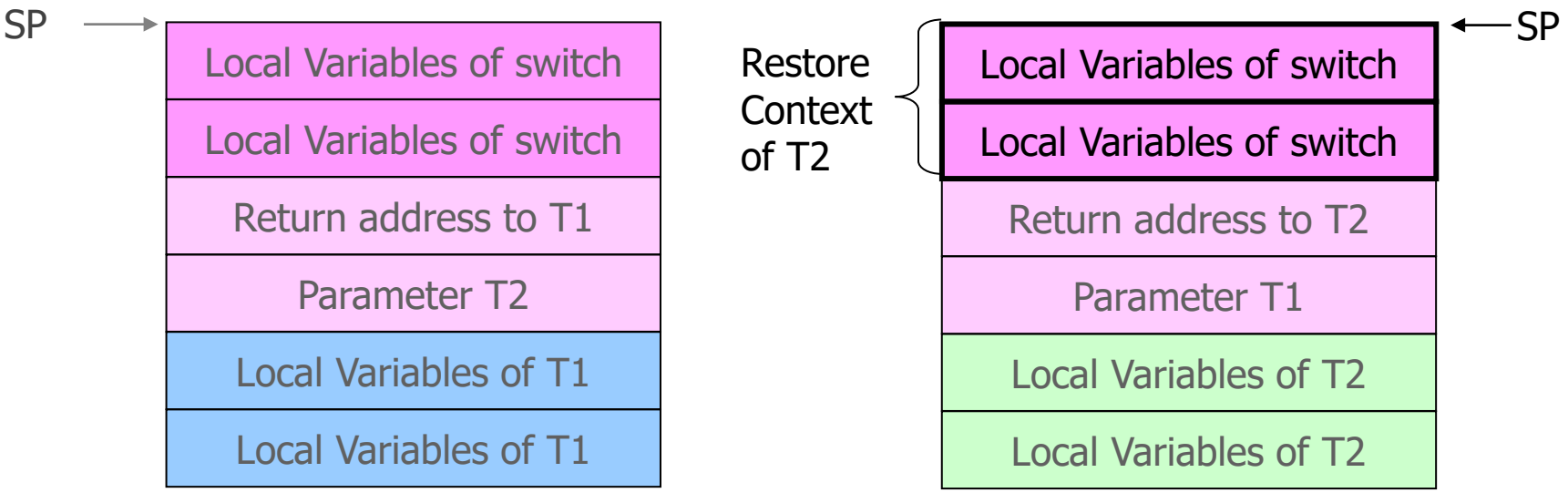


Thread Control

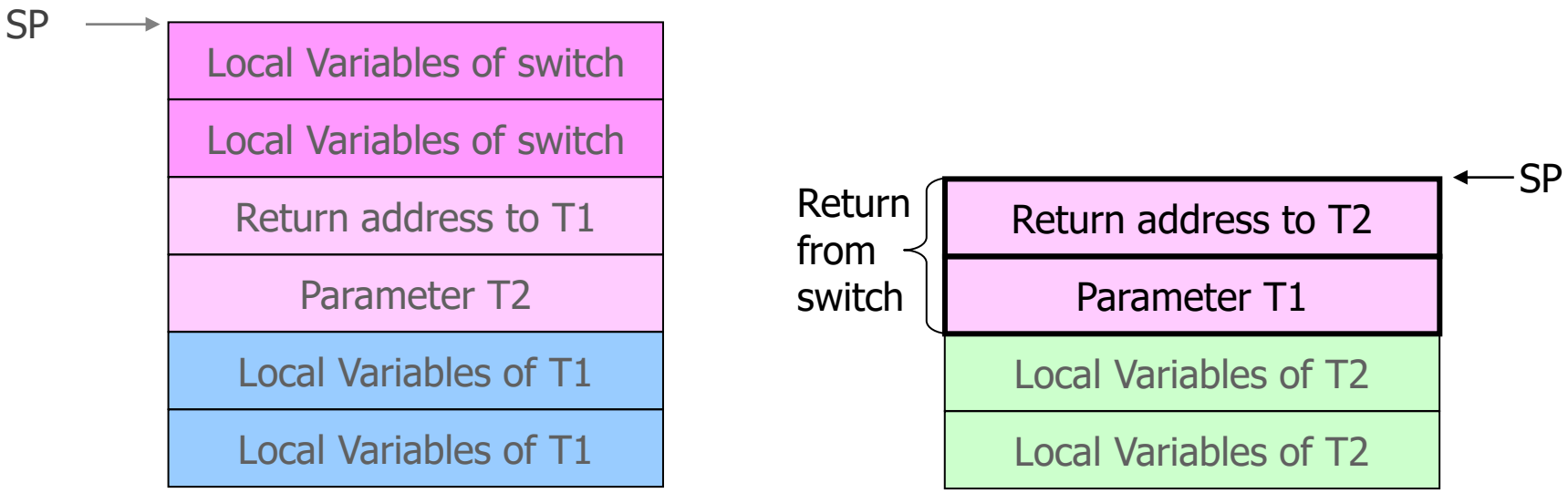
Stack Contents during simplified switch



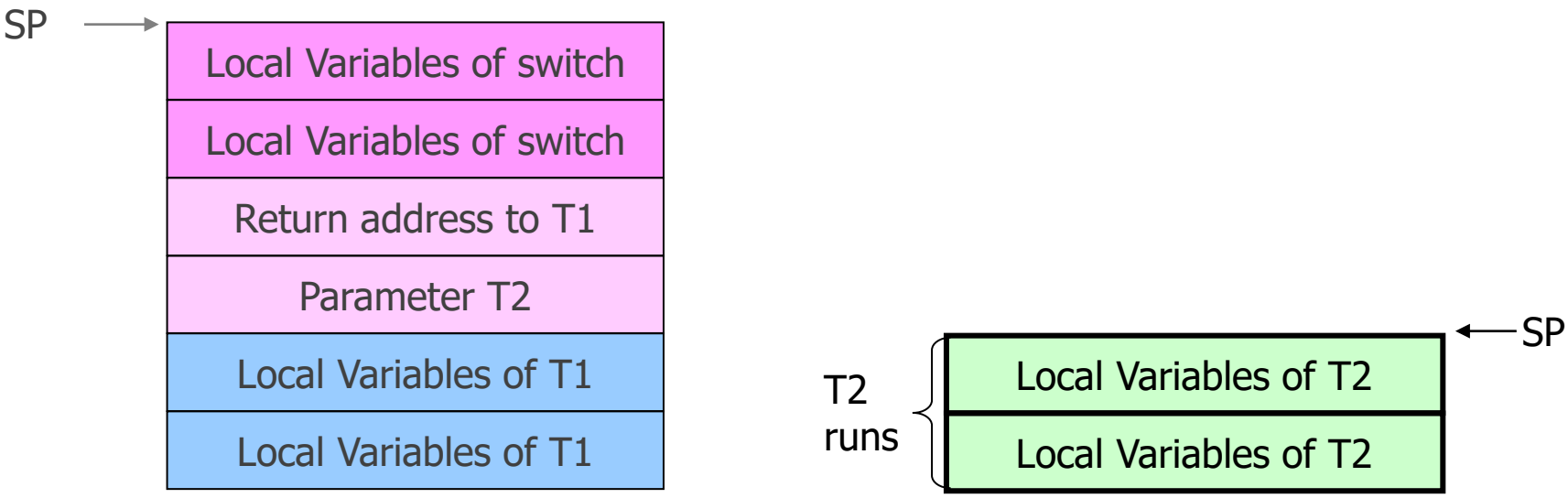
Stack Contents during simplified switch



Stack Contents during simplified switch

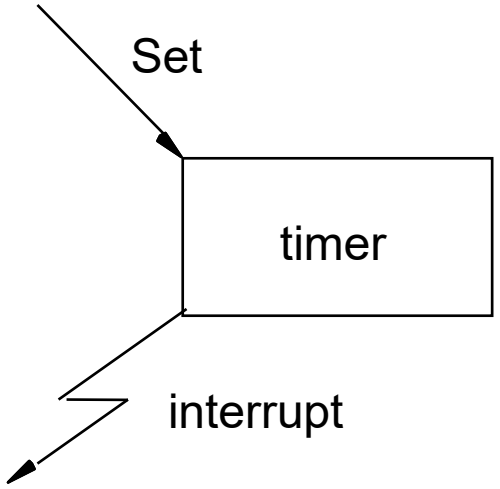


Stack Contents during simplified switch



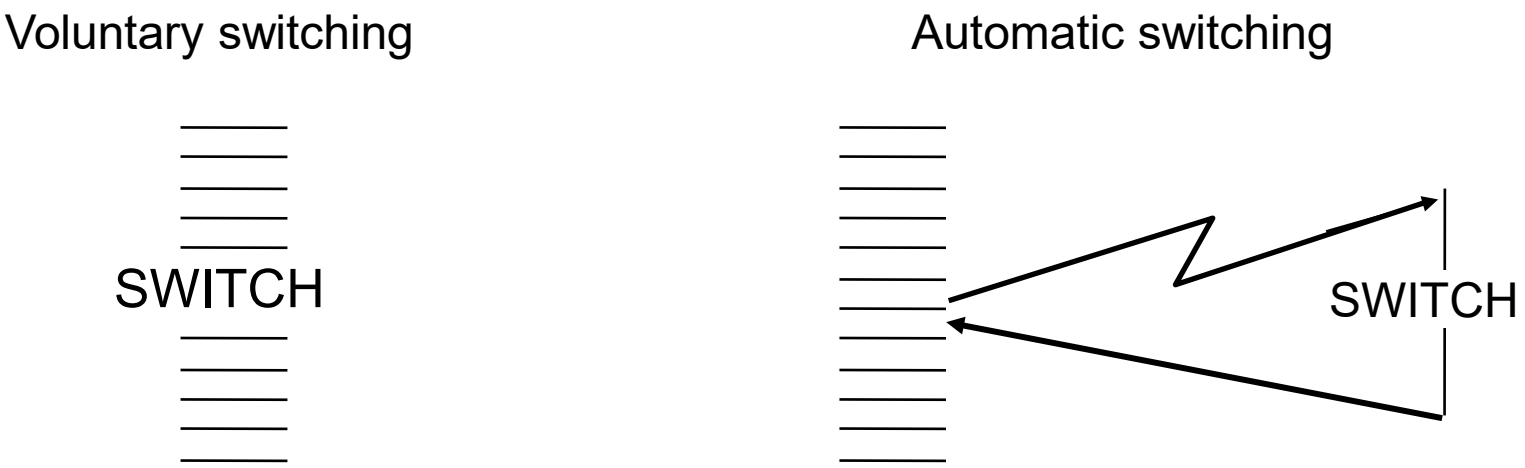
Automatic switching

- In many cases it is not possible or not reasonable to explicitly insert switching points into the threads.
- More desirable is **automatic** switching.
- To that end we need **a clock** (**timer**), i.e. a hardware device offering the following functions:
  - specifying a deadline (timer set)
  - interrupt on timeout



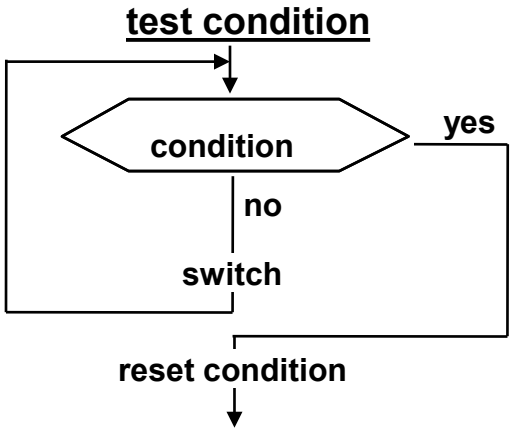
Automatic switching

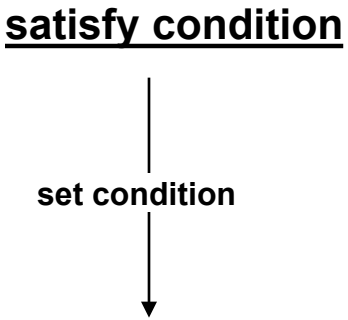
- With automatic switching programs can remain unchanged.
- The thread switch is triggered "from outside" and can happen at any arbitrary point in time. (Interrupts may not be switched off.)



Conditioned switching

- In the course of a thread situations can arise where a continuation of the processing is temporarily not possible, e.g. if the thread has to wait for input data.
- Instead of wasting time, the processor can do some other work.
- This is called **conditioned switching**, since the fact that switching takes place or not depends on some condition.
- Such a condition can be represented by a simple binary variable.





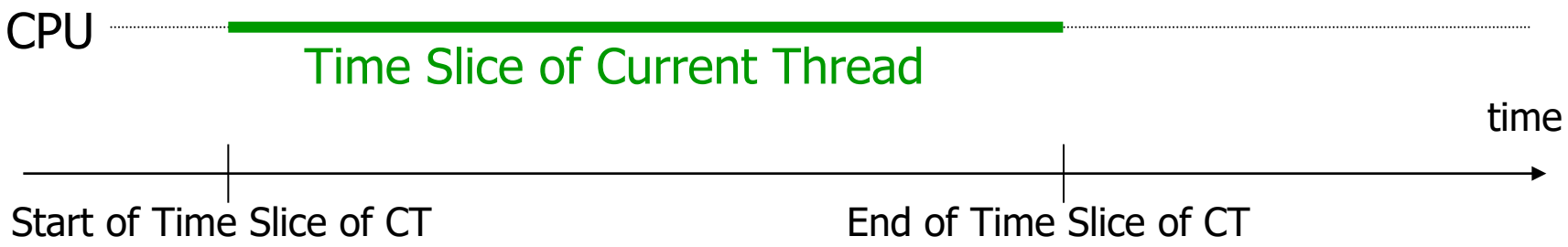
Thread Switching due to End of Time Slice

Freie Universität Berlin

Objective: Establishing Fair Scheduling

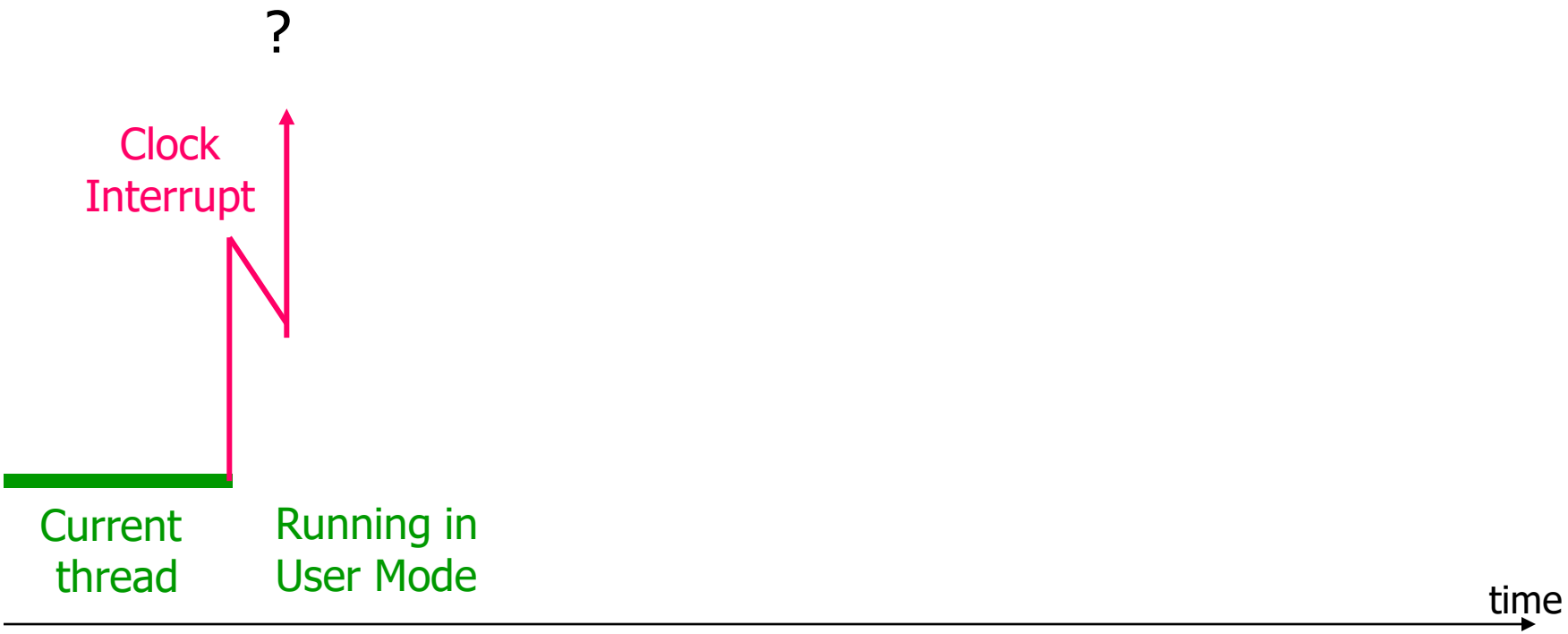
Assumption: No other thread switching events

Simplification: No detailed clock interrupt handling



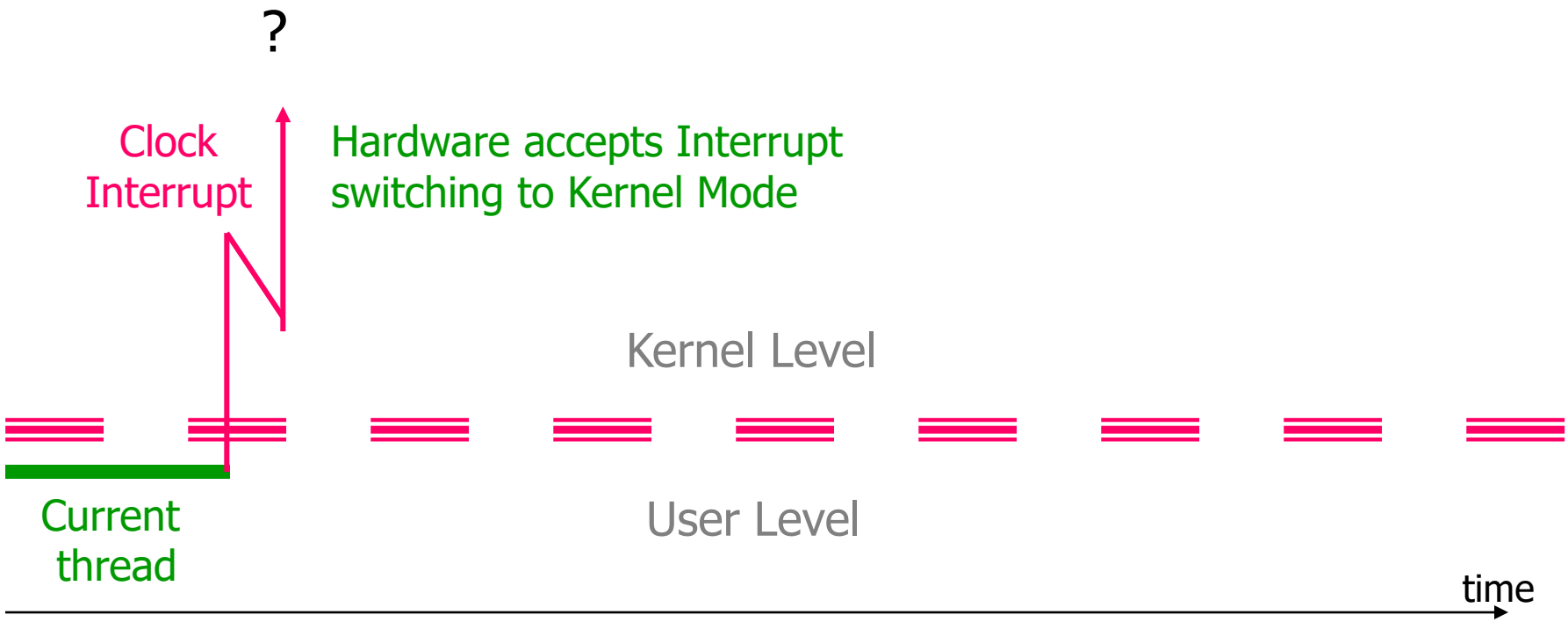
Simplified Thread Switch

green → blue



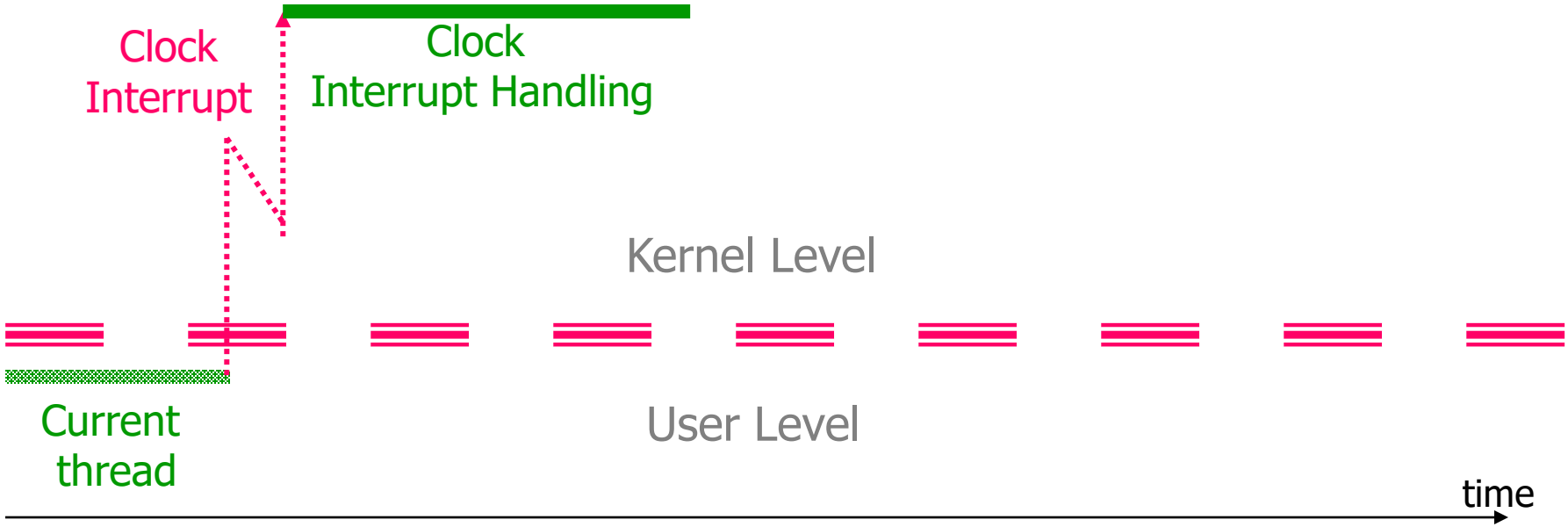
Simplified Thread Switch

green → blue

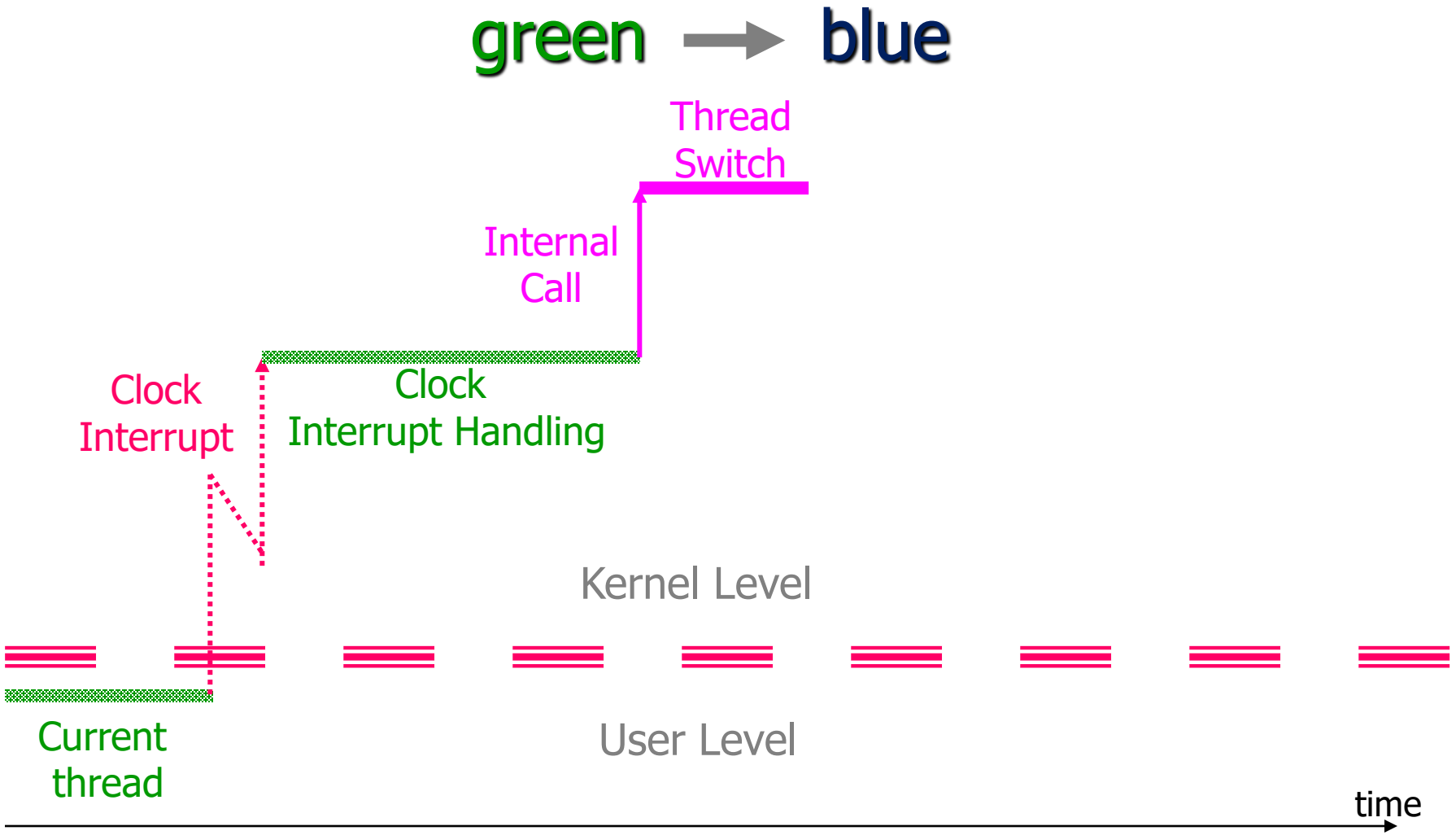


Simplified Thread Switch

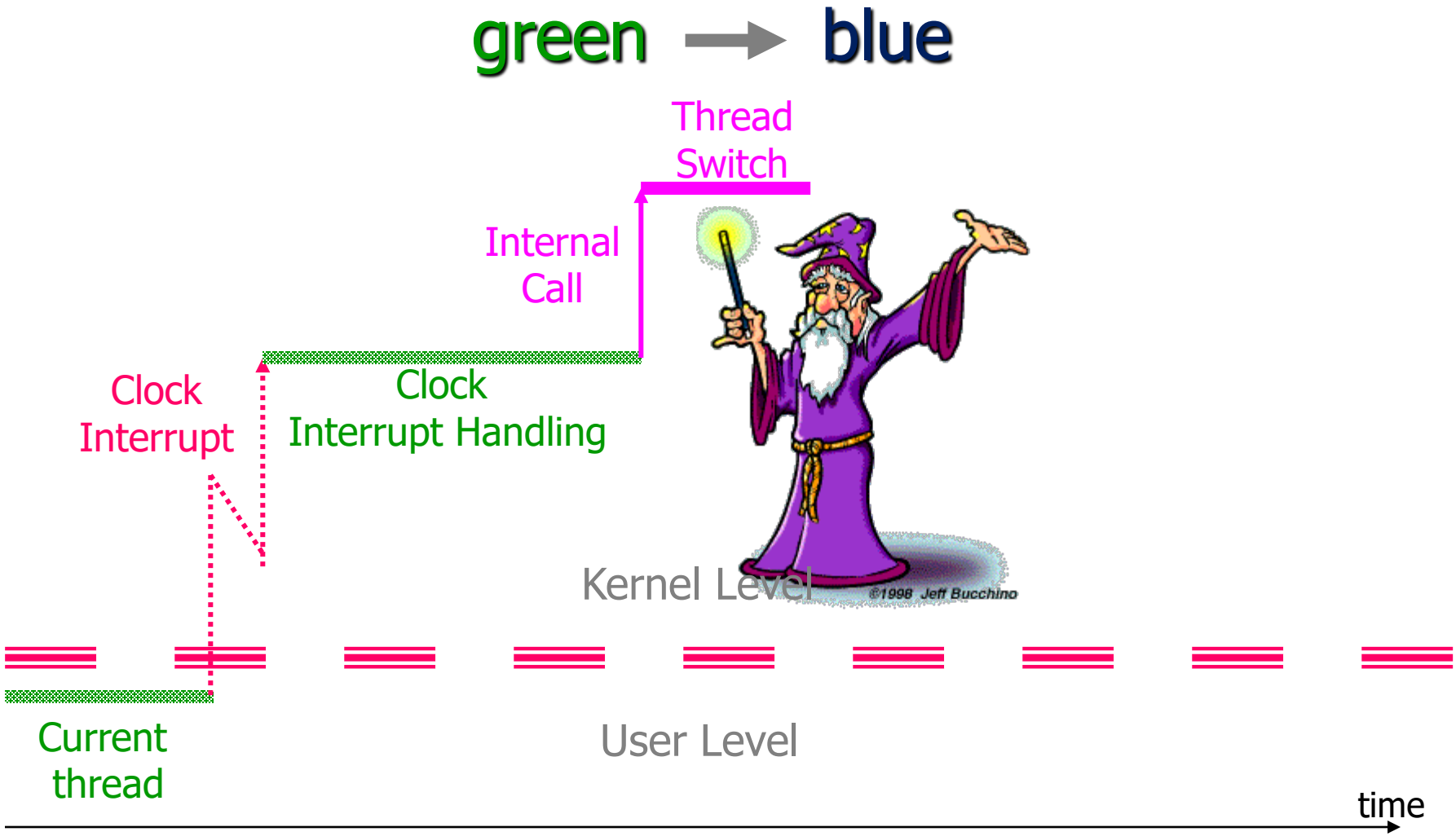
green → blue



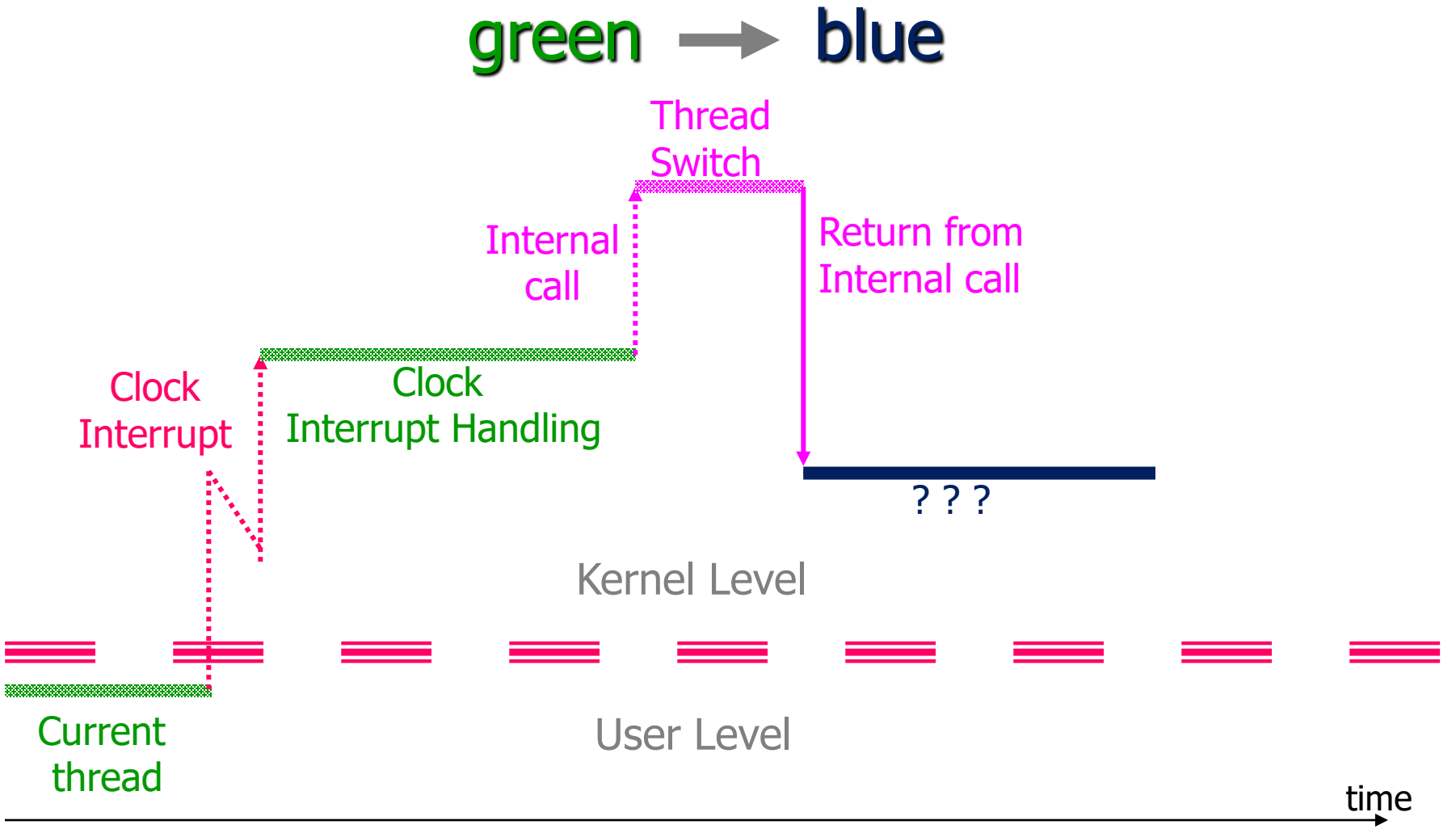
Simplified Thread Switch



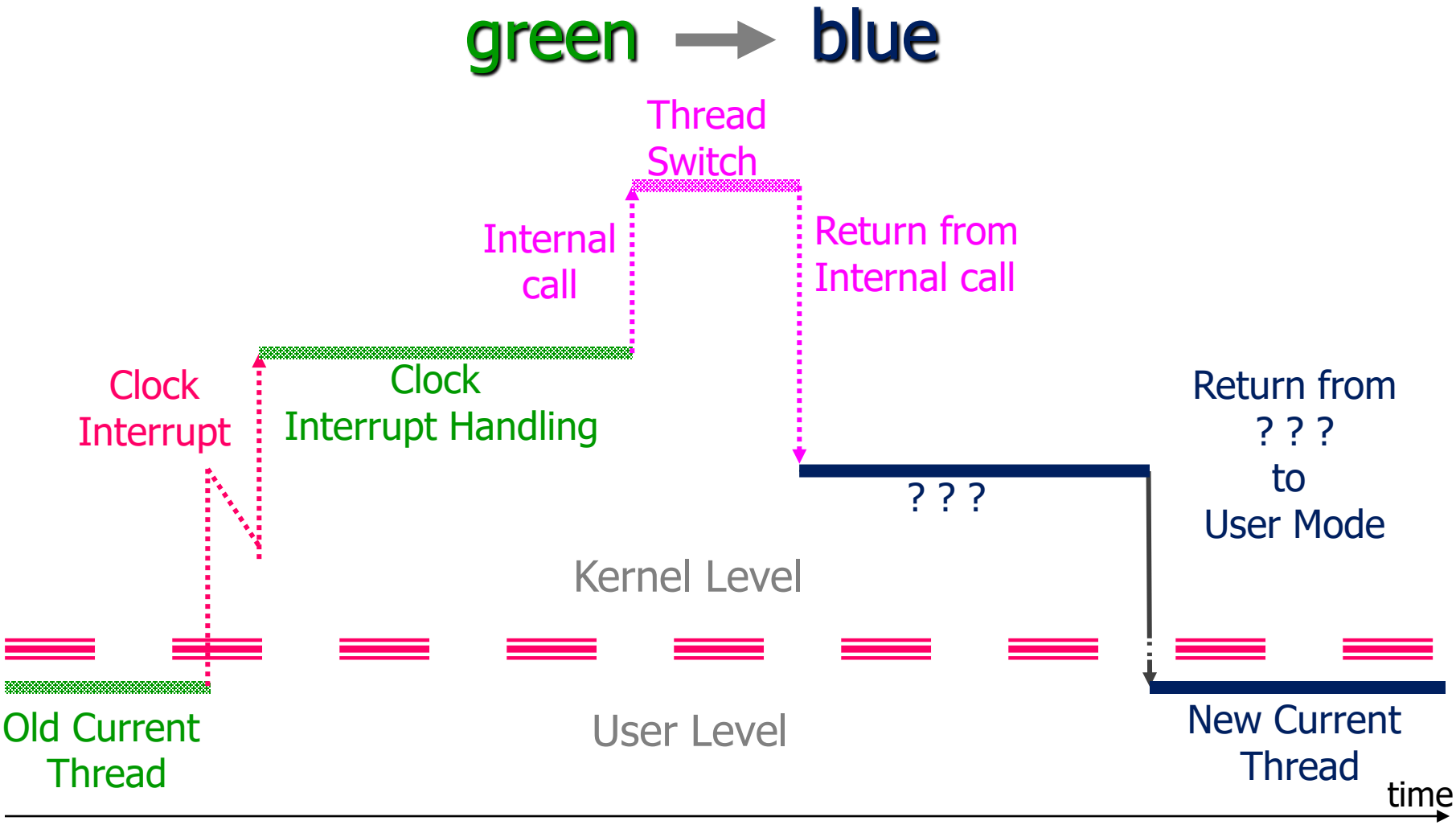
Simplified Thread Switch



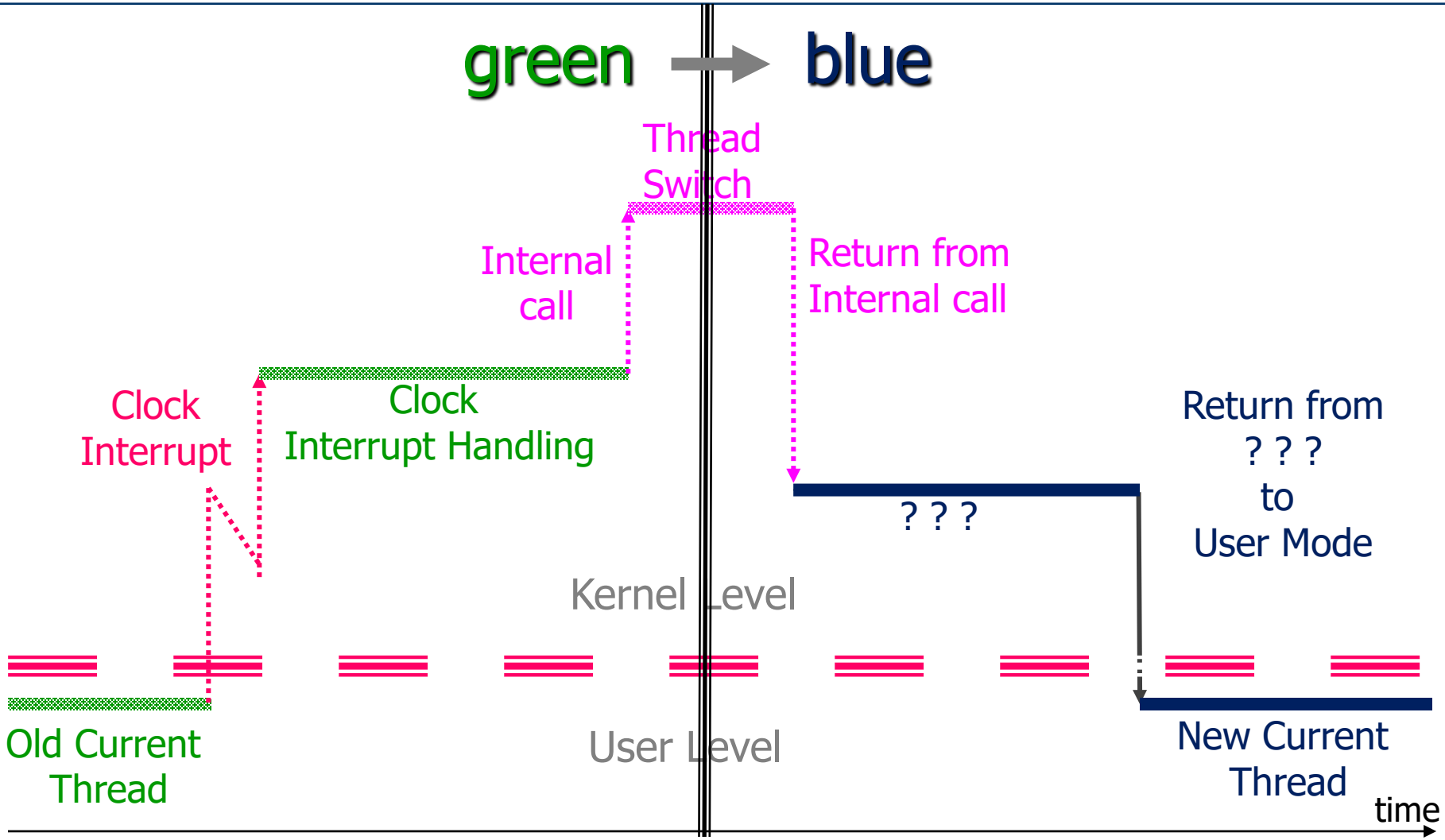
Simplified Thread Switch



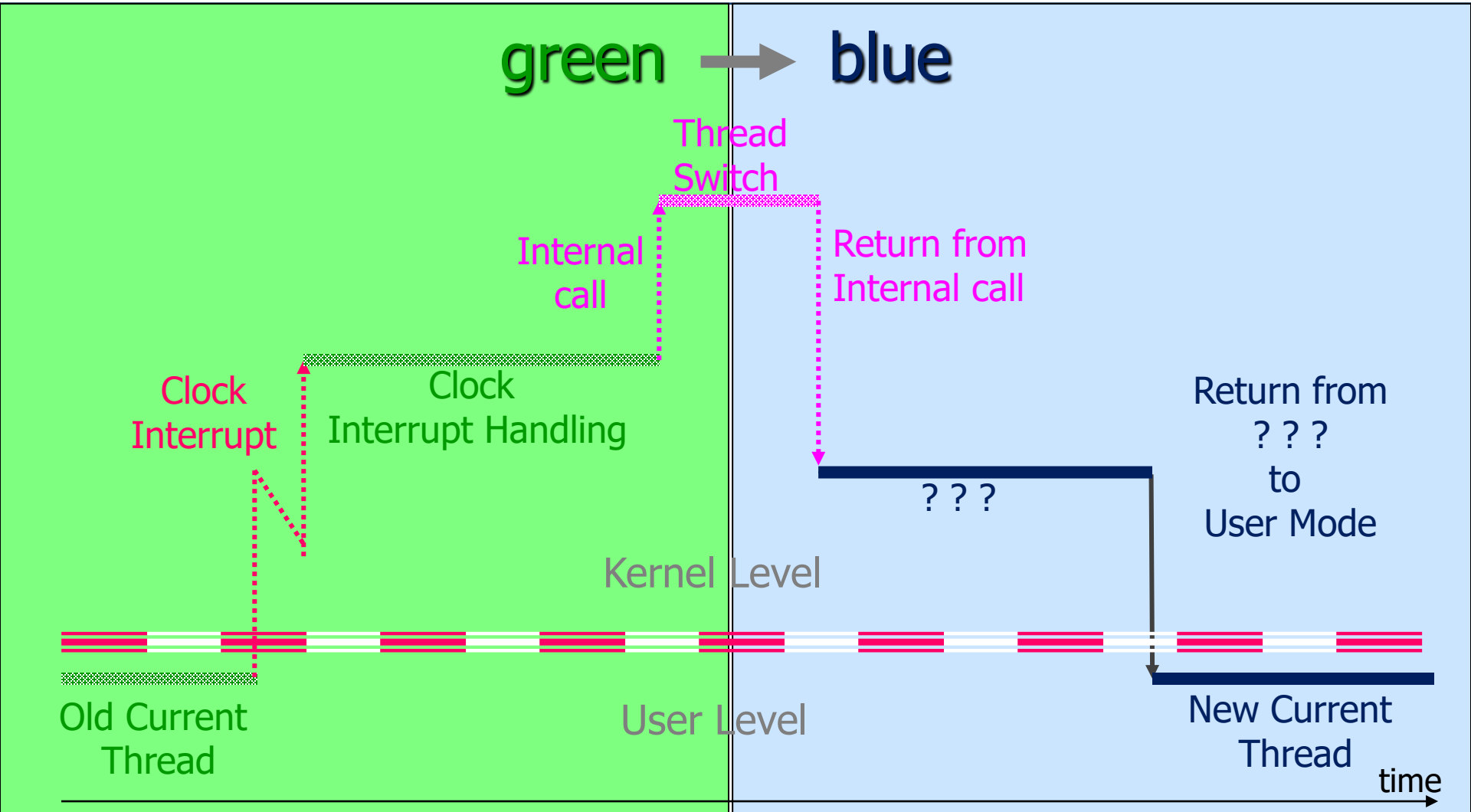
Simplified Thread Switch



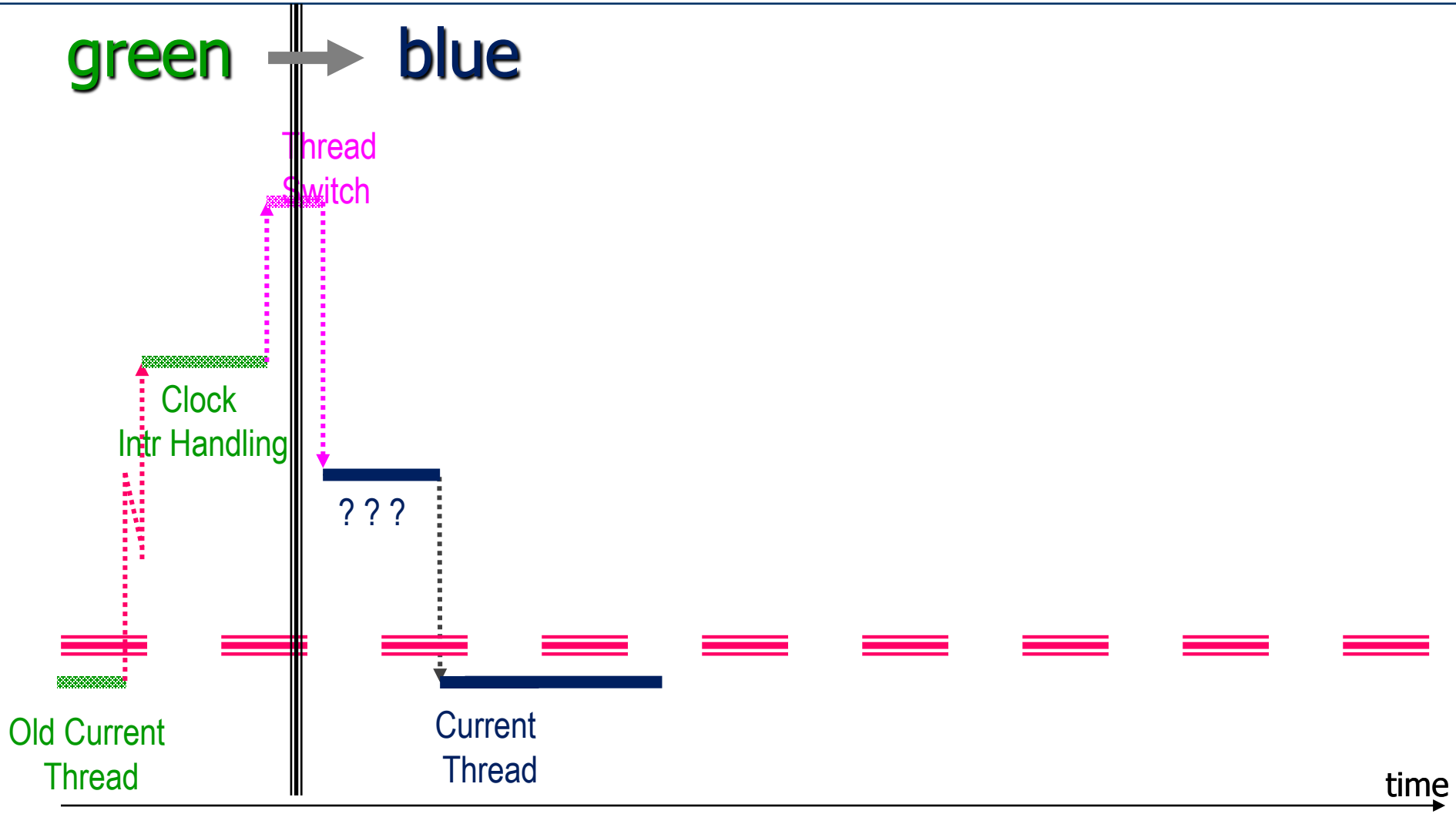
Simplified Thread Switch



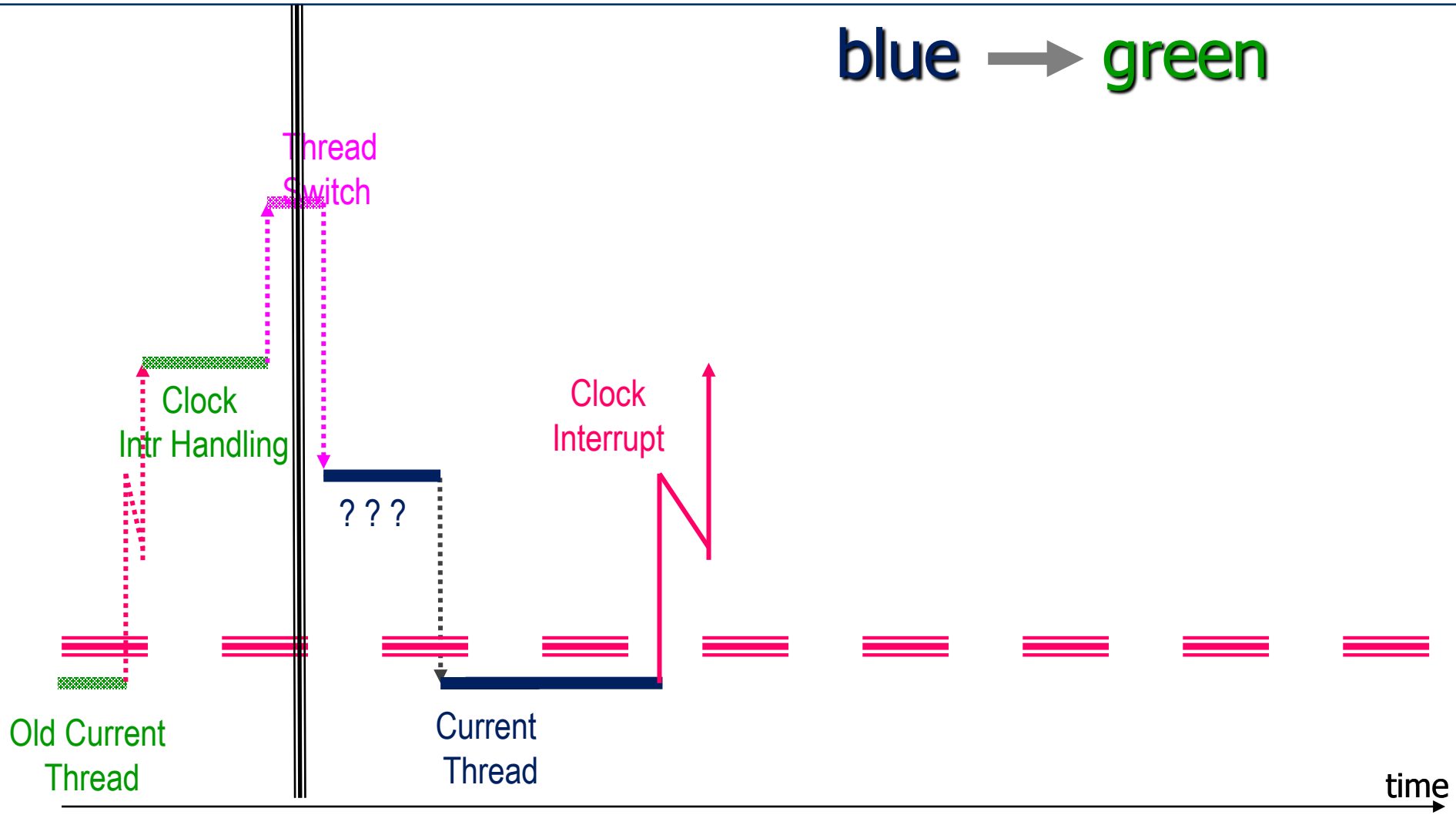
Simplified Thread Switch



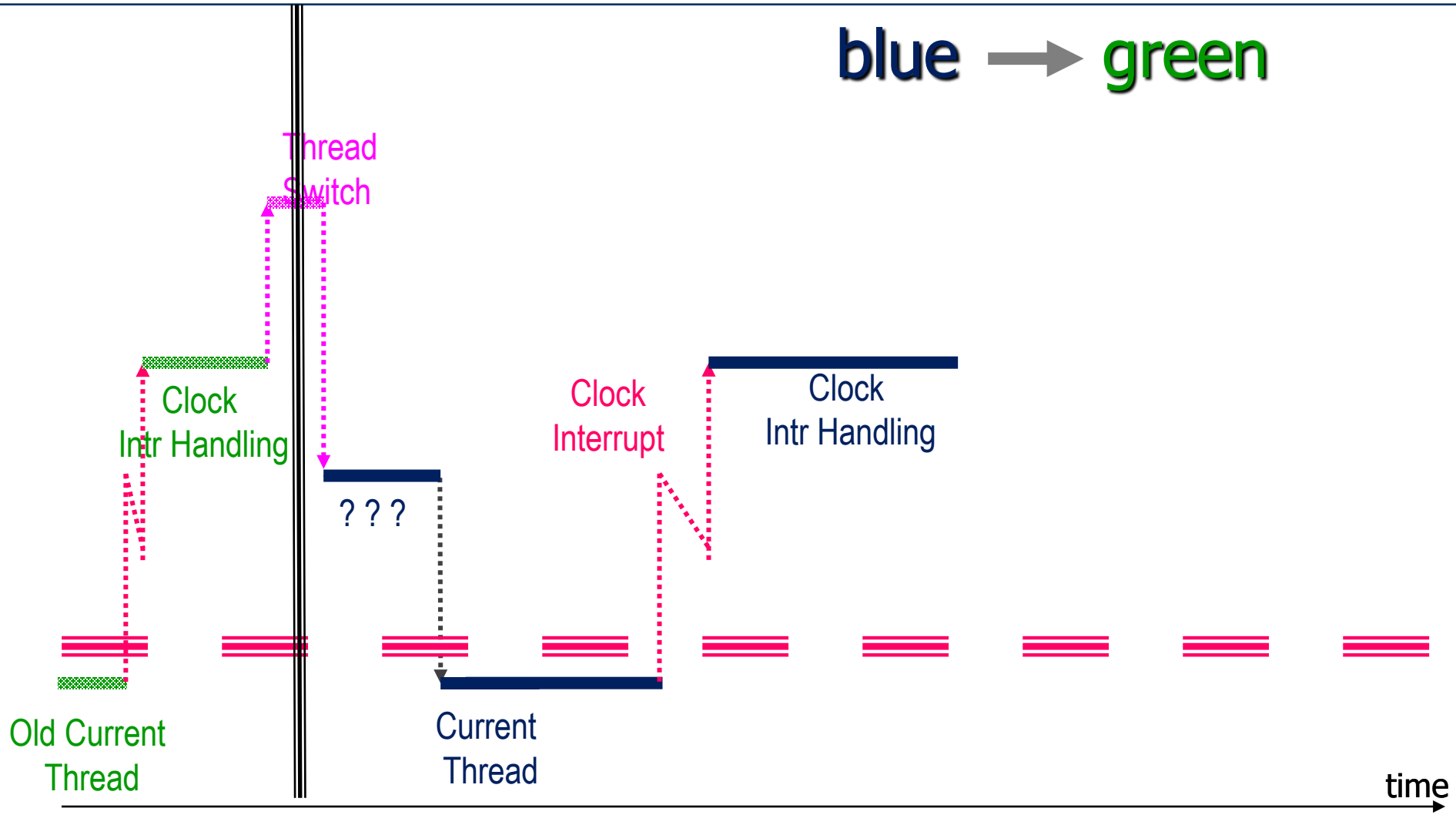
Simplified Thread Switch



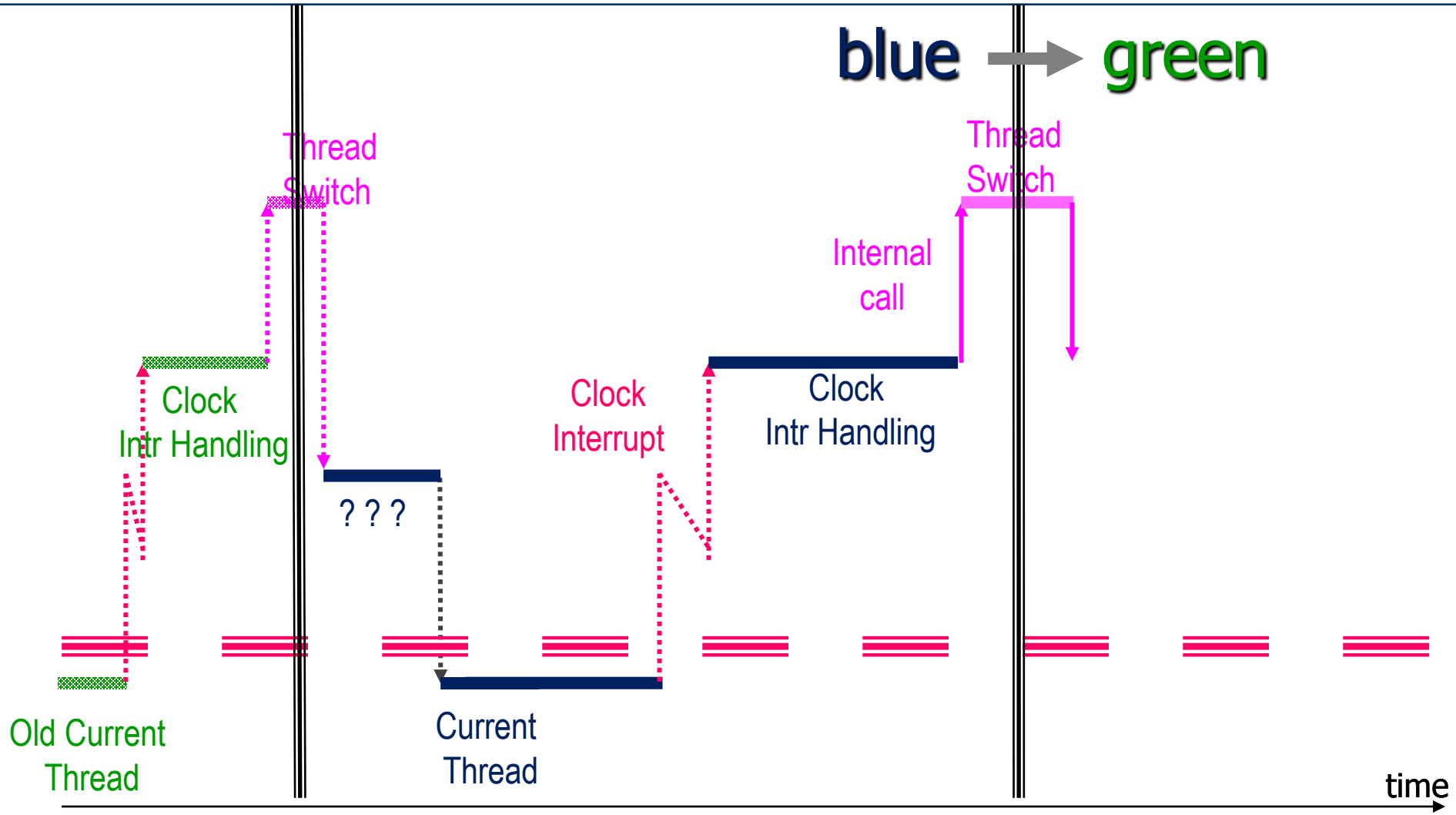
Simplified Thread Switch



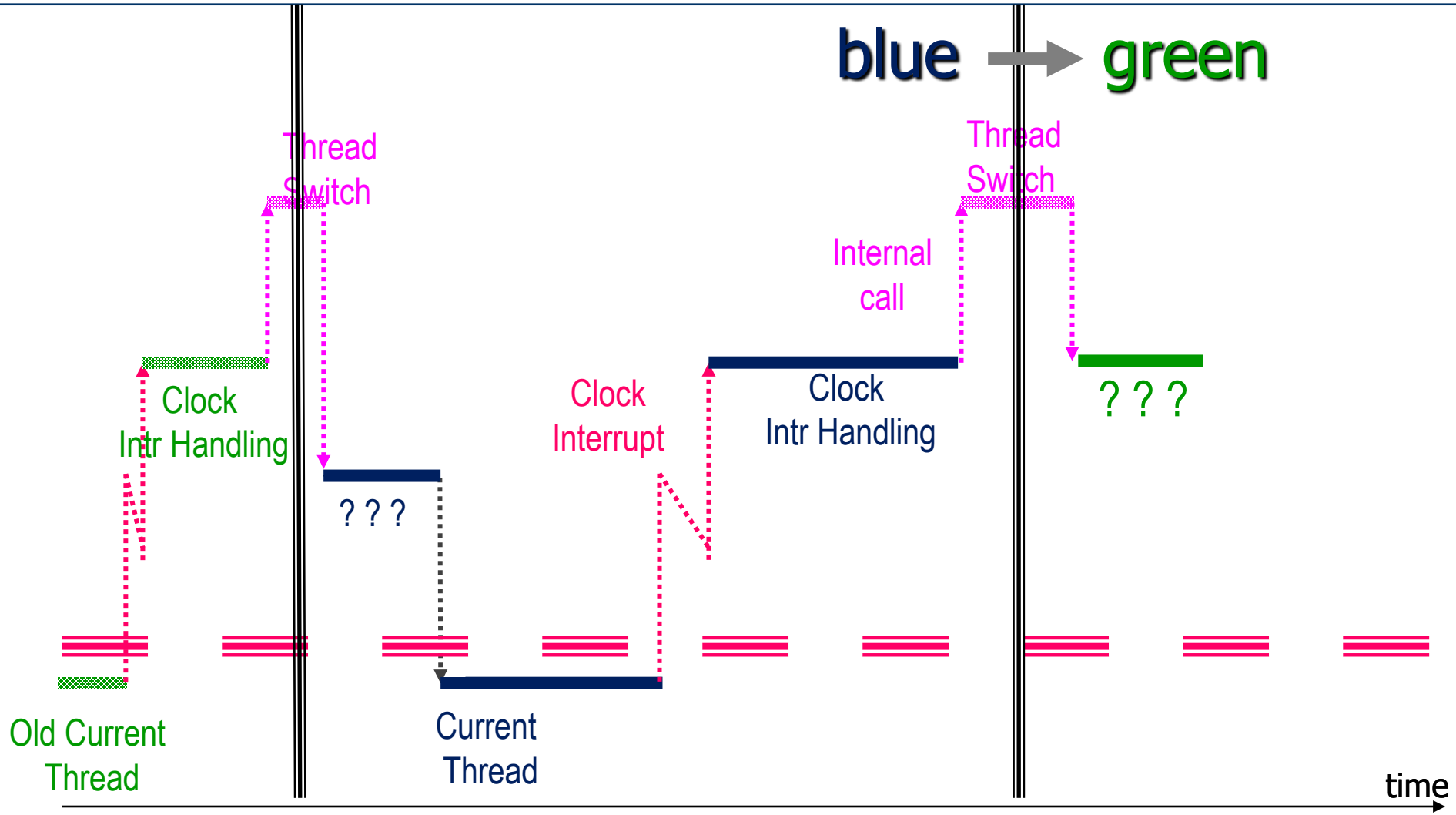
Simplified Thread Switch



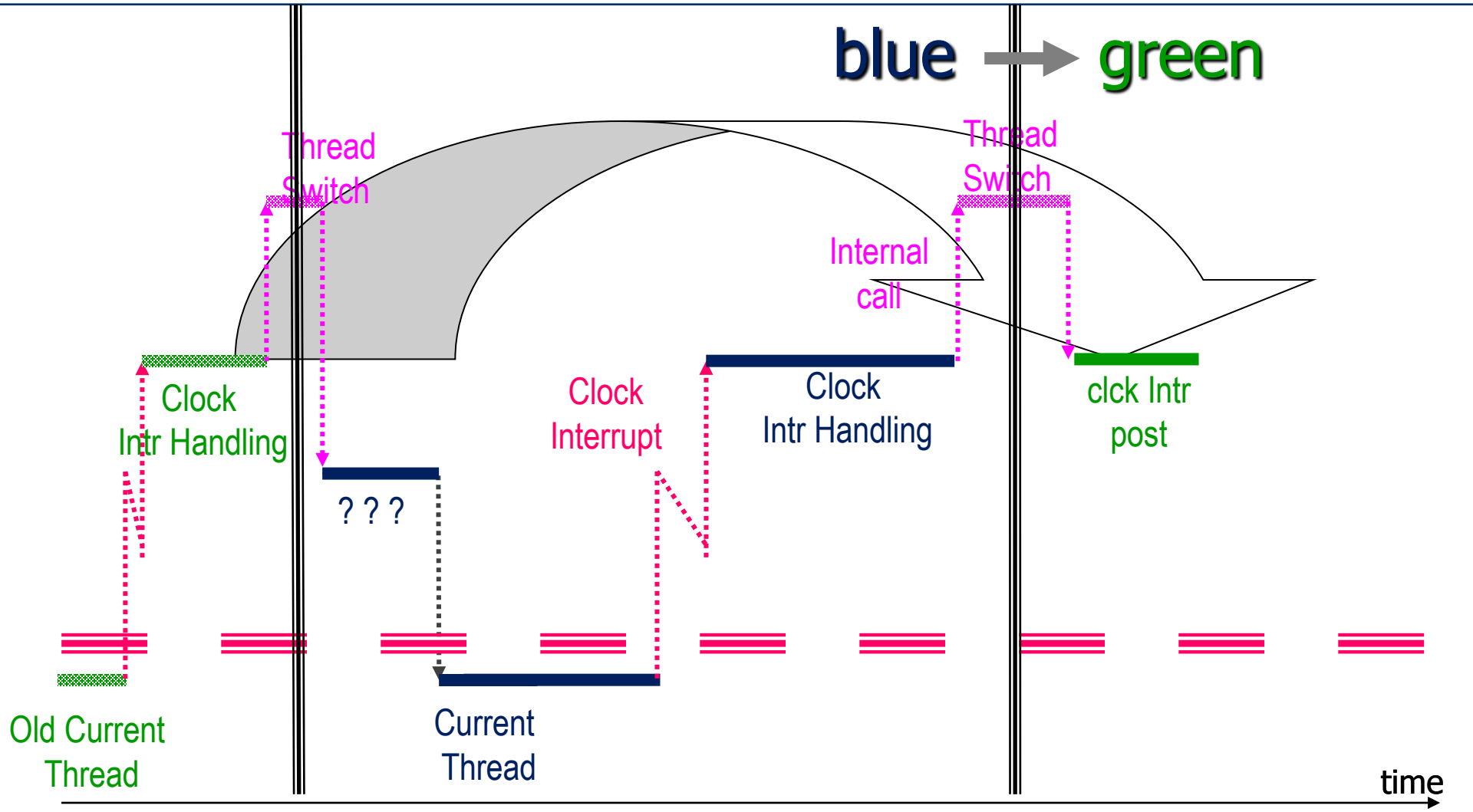
Simplified Thread Switch



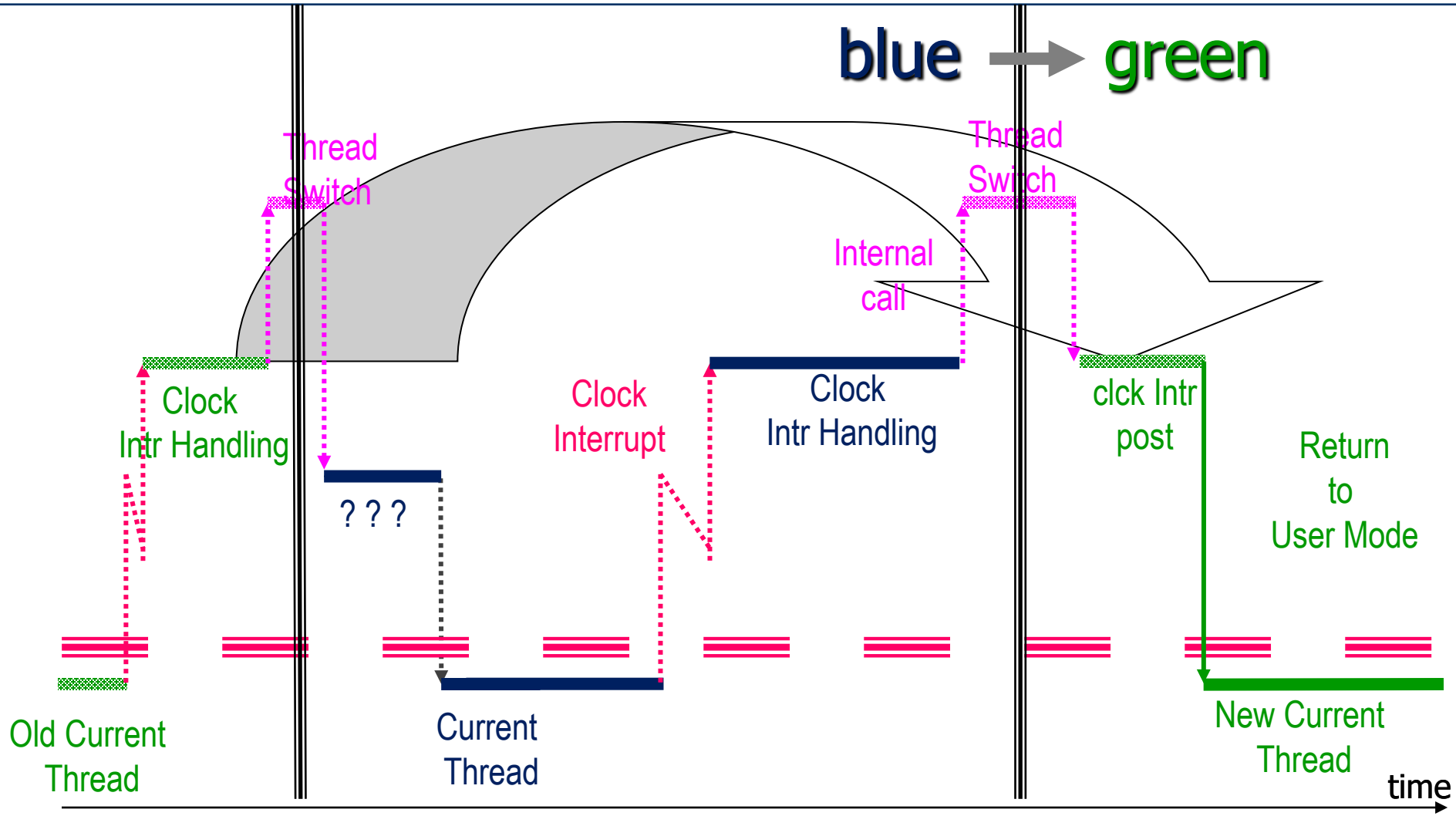
Simplified Thread Switch



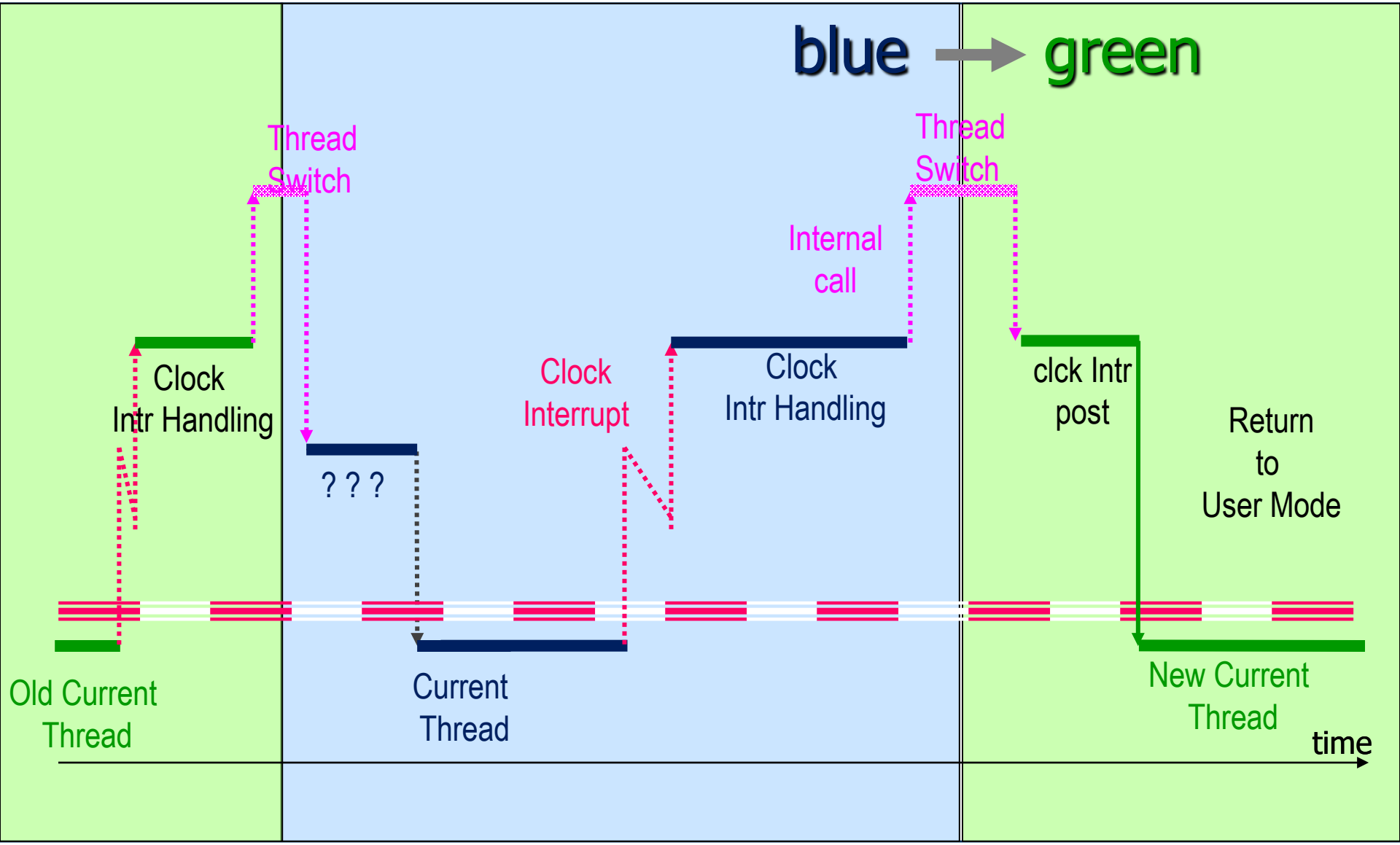
Simplified Thread Switch



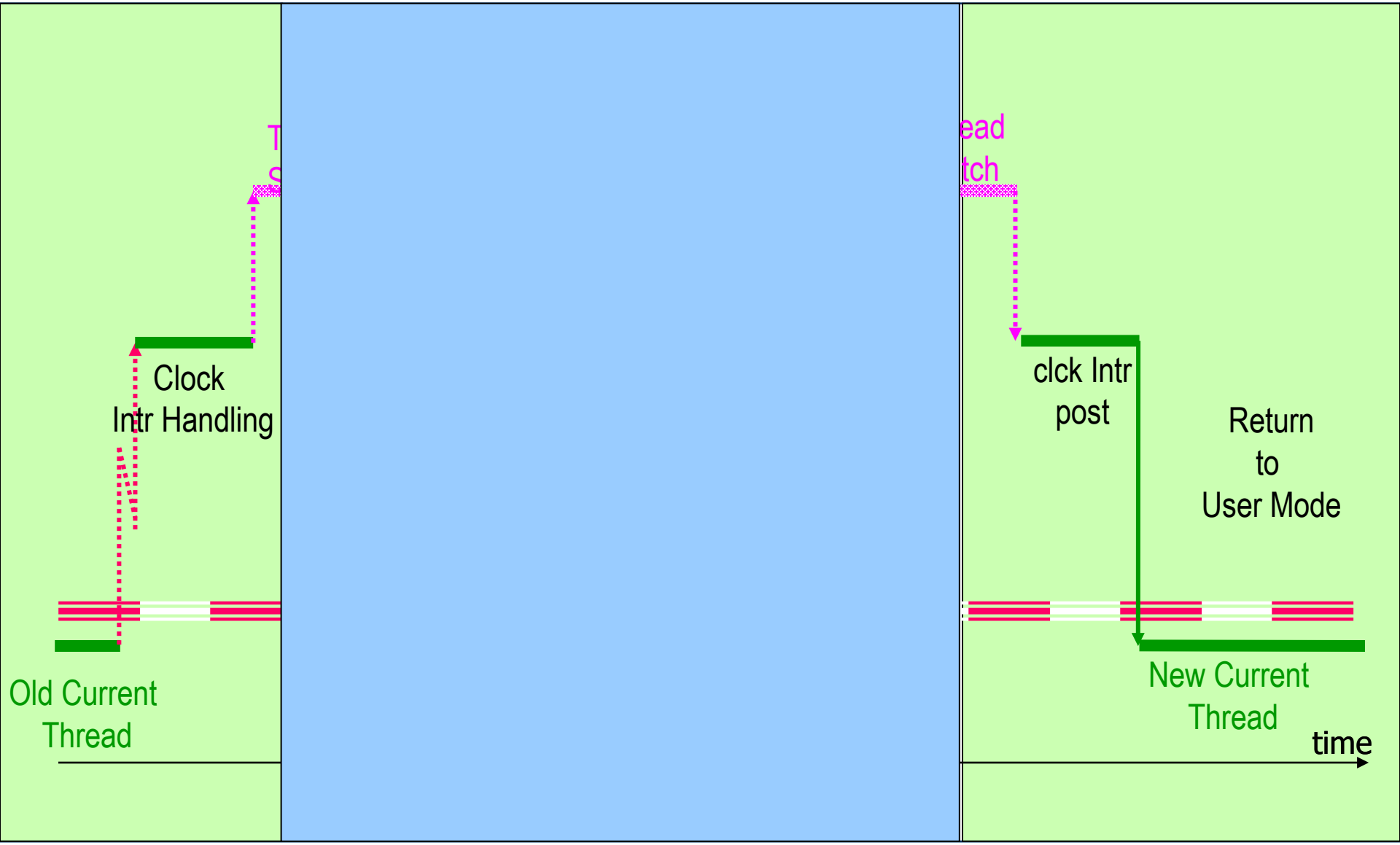
Simplified Thread Switch



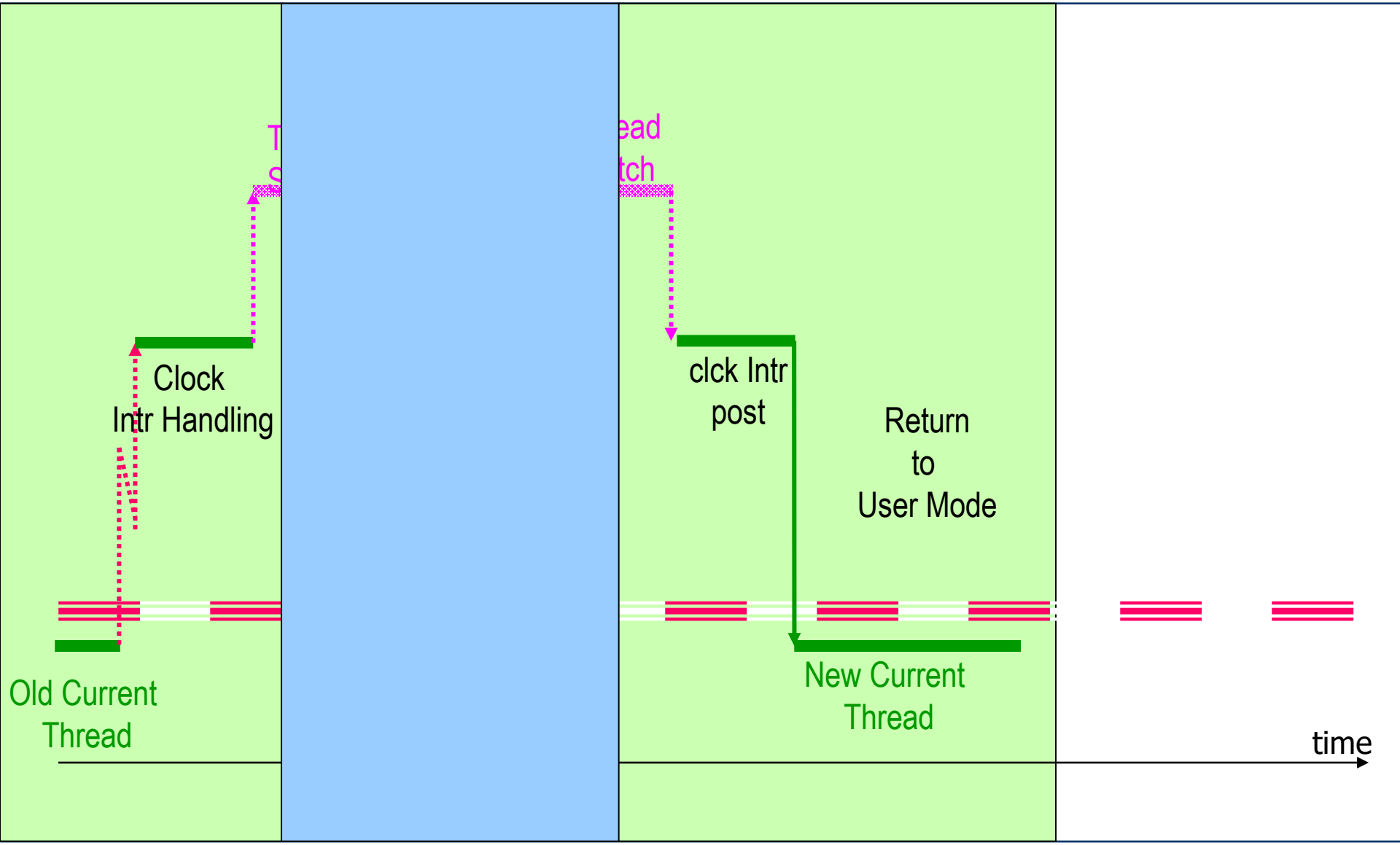
Simplified Thread Switch



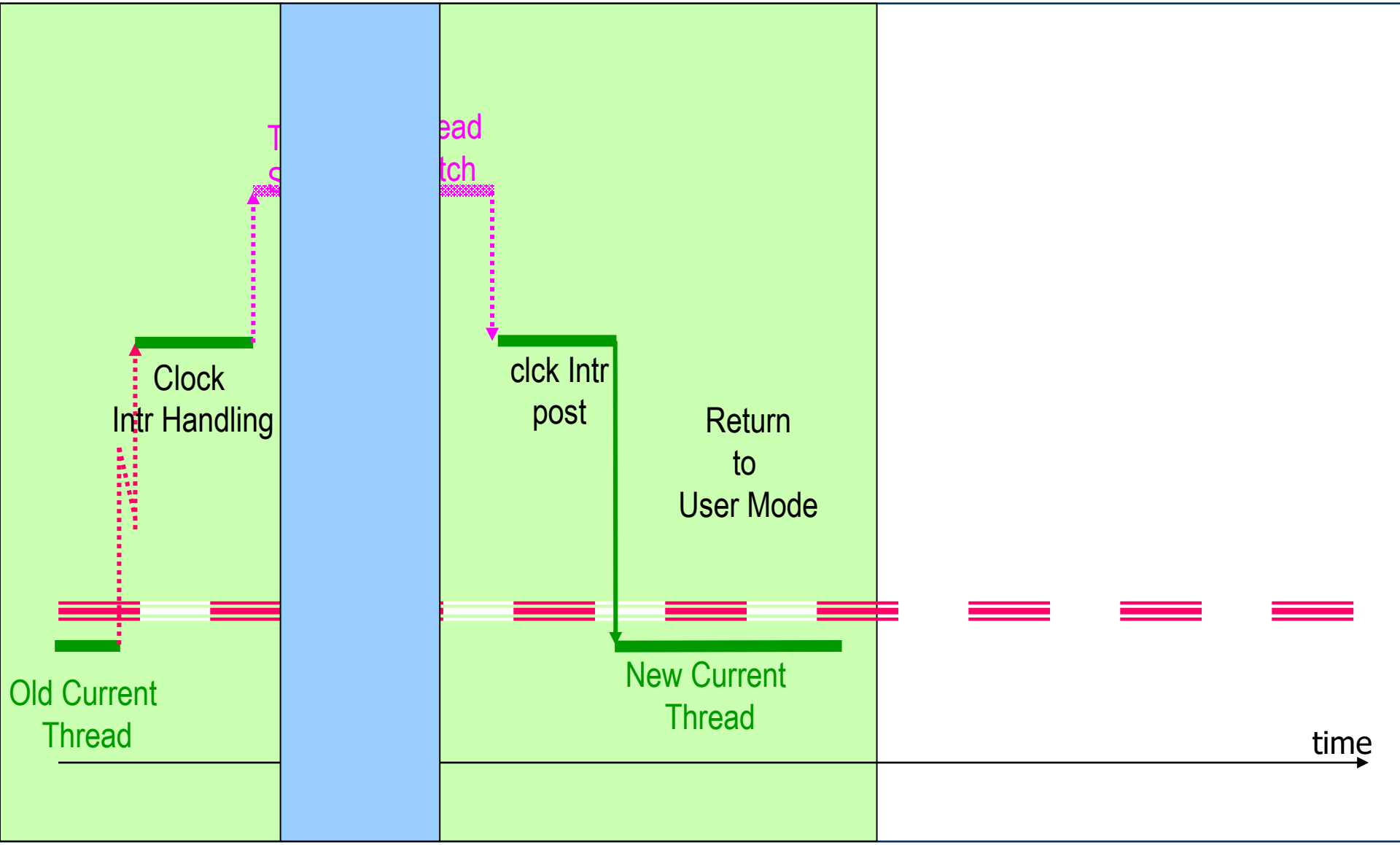
Simplified Thread Switch



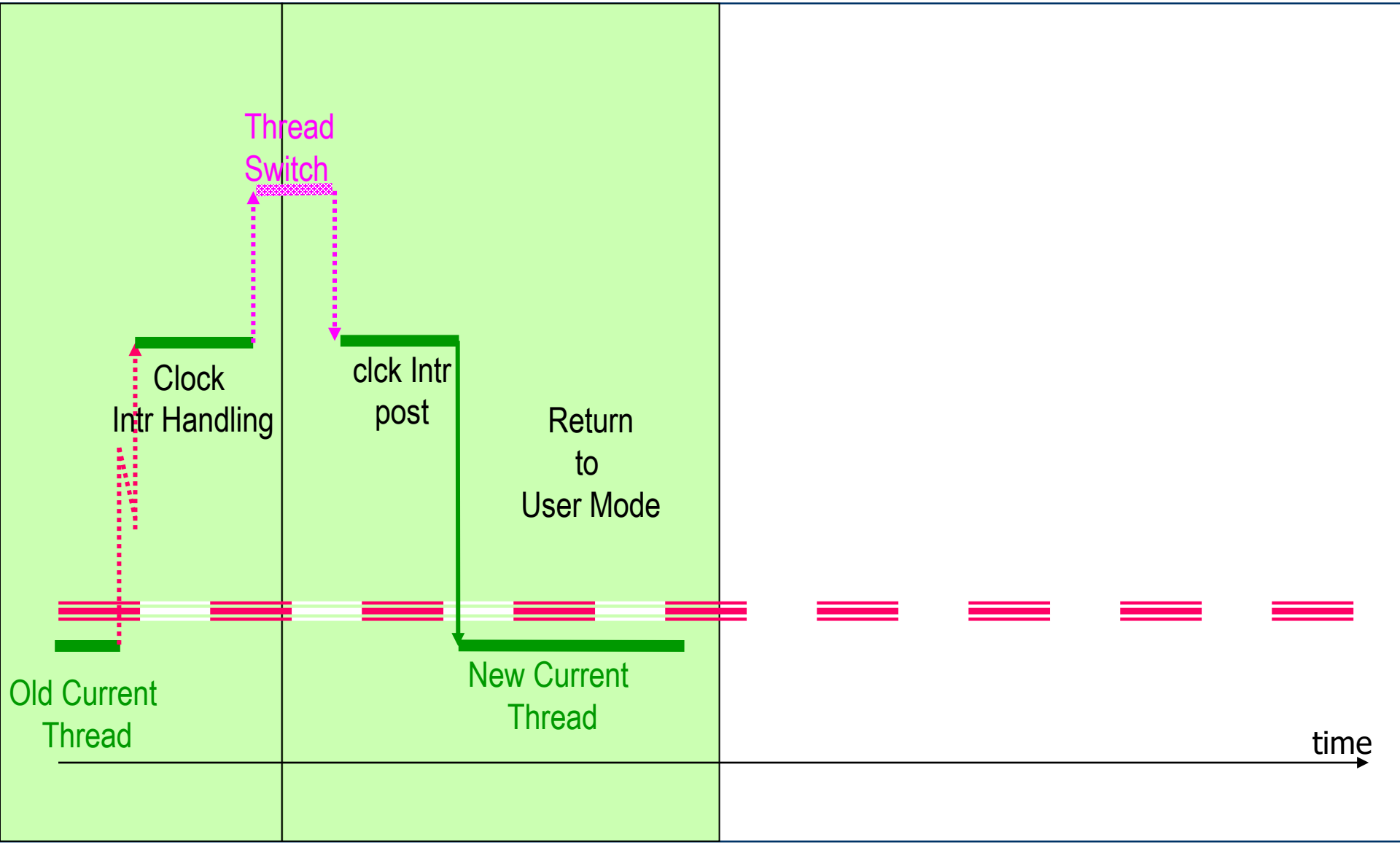
Simplified Thread Switch



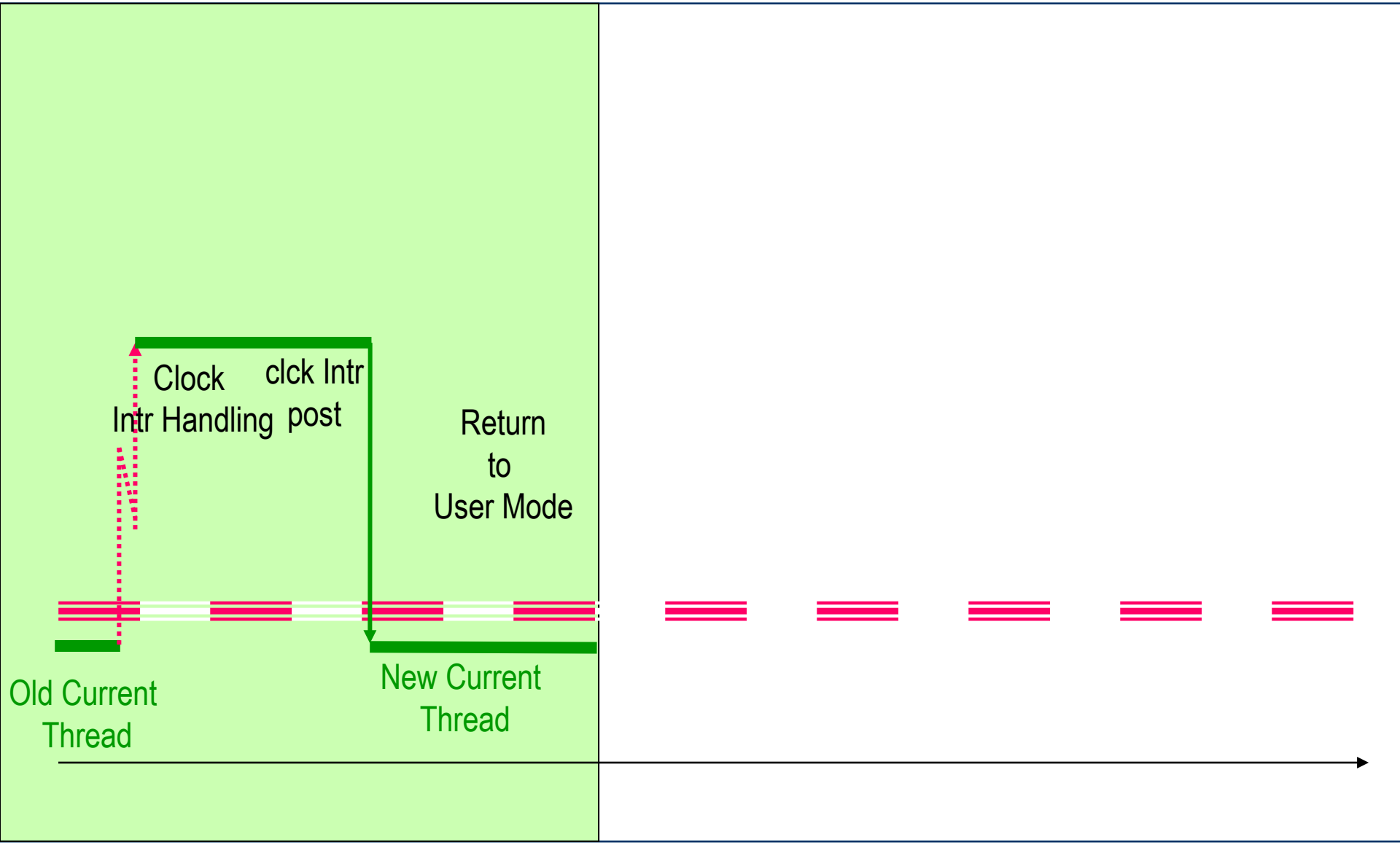
Simplified Thread Switch



Simplified Thread Switch



Simplified Thread Switch



Thread Switch

Freie Universität Berlin

Assumption:

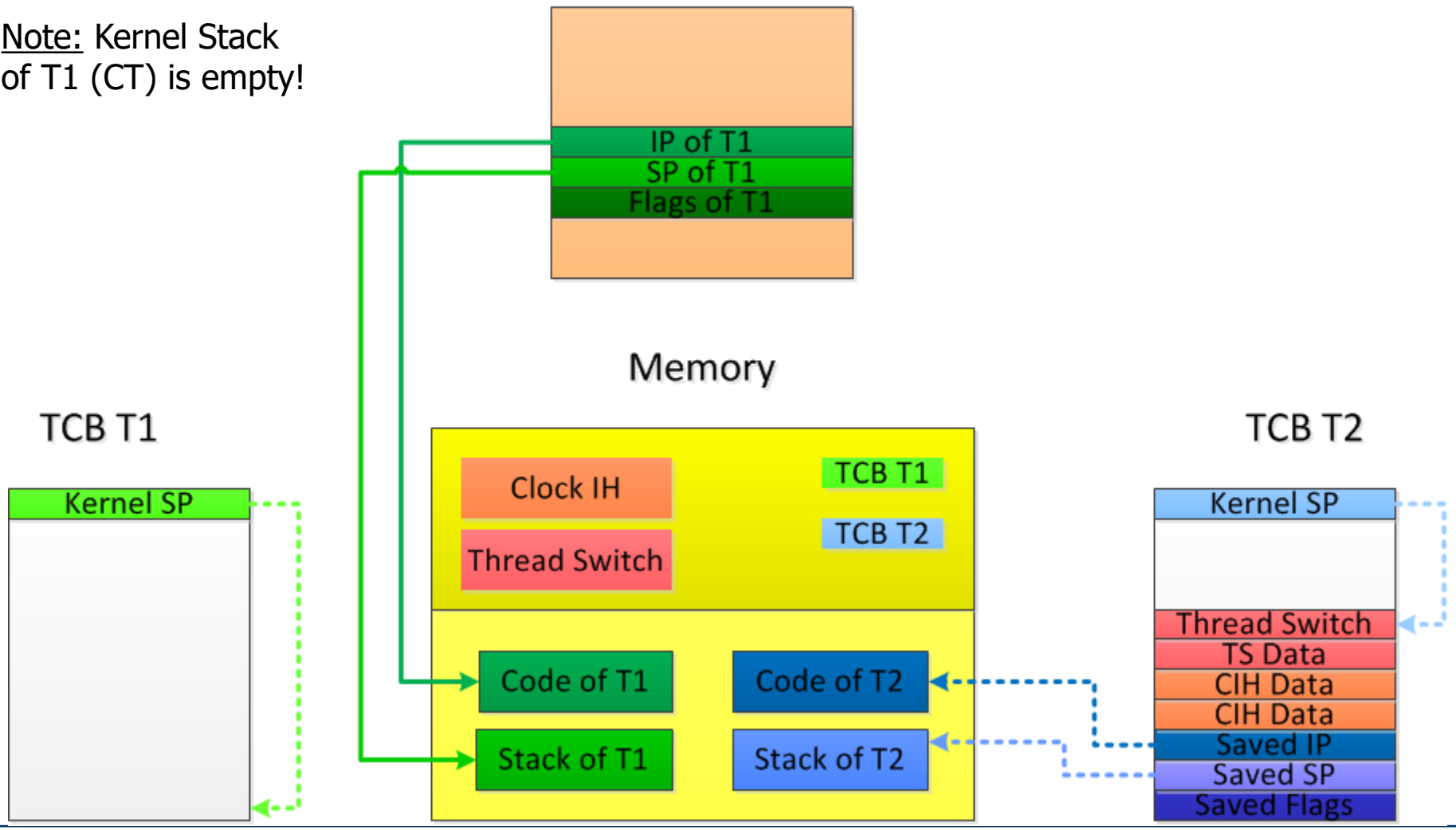
Hardware automatically pushes SP, IP and Flags of Current Thread T1 onto its Kernel Stack within TCB1!

Thread Switch

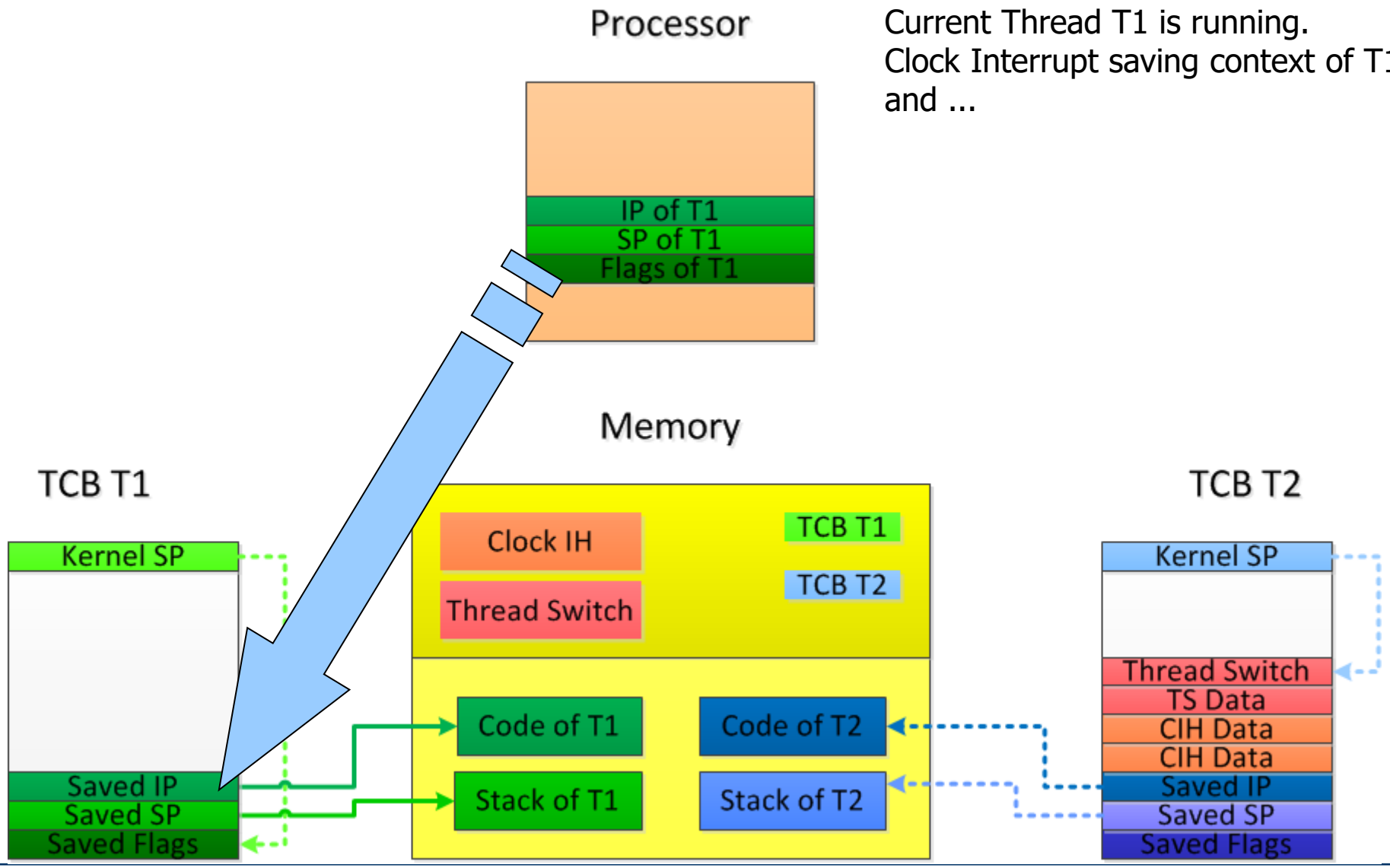
Freie Universität Berlin

Current Thread T1 is running.

Note: Kernel Stack of T1 (CT) is empty!

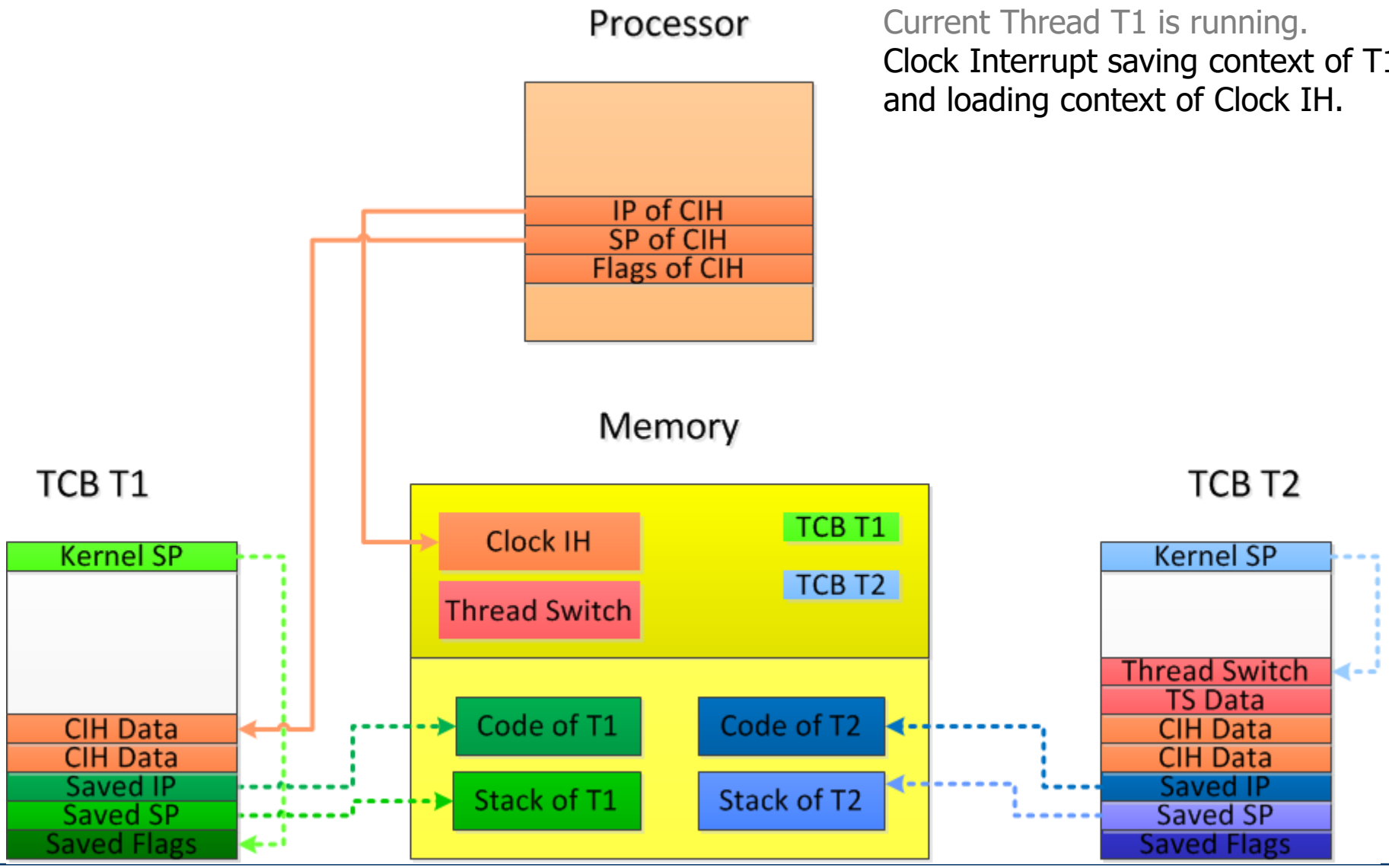


Thread Switch



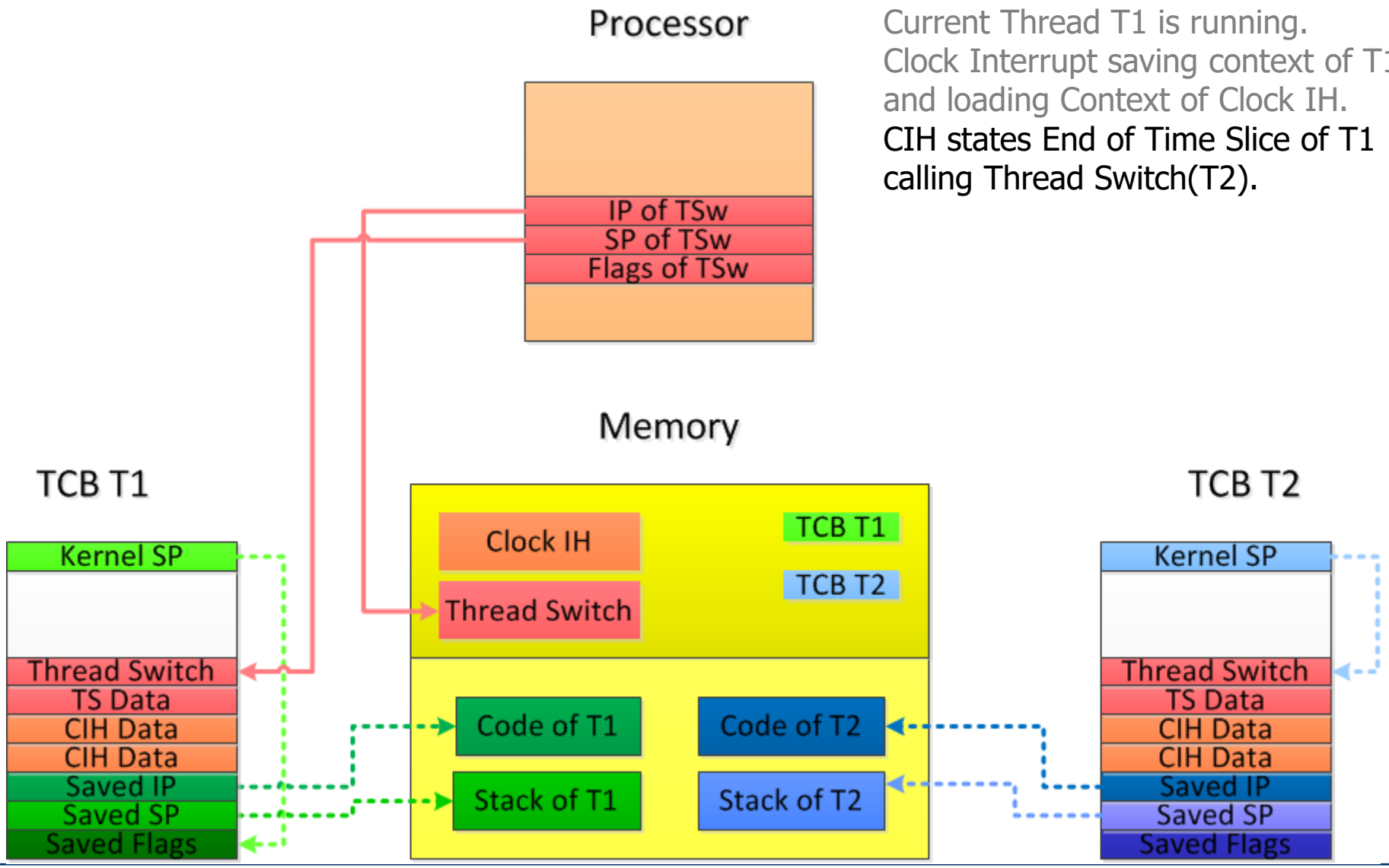
Current Thread T1 is running.
Clock Interrupt saving context of T1 and ...

Thread Switch



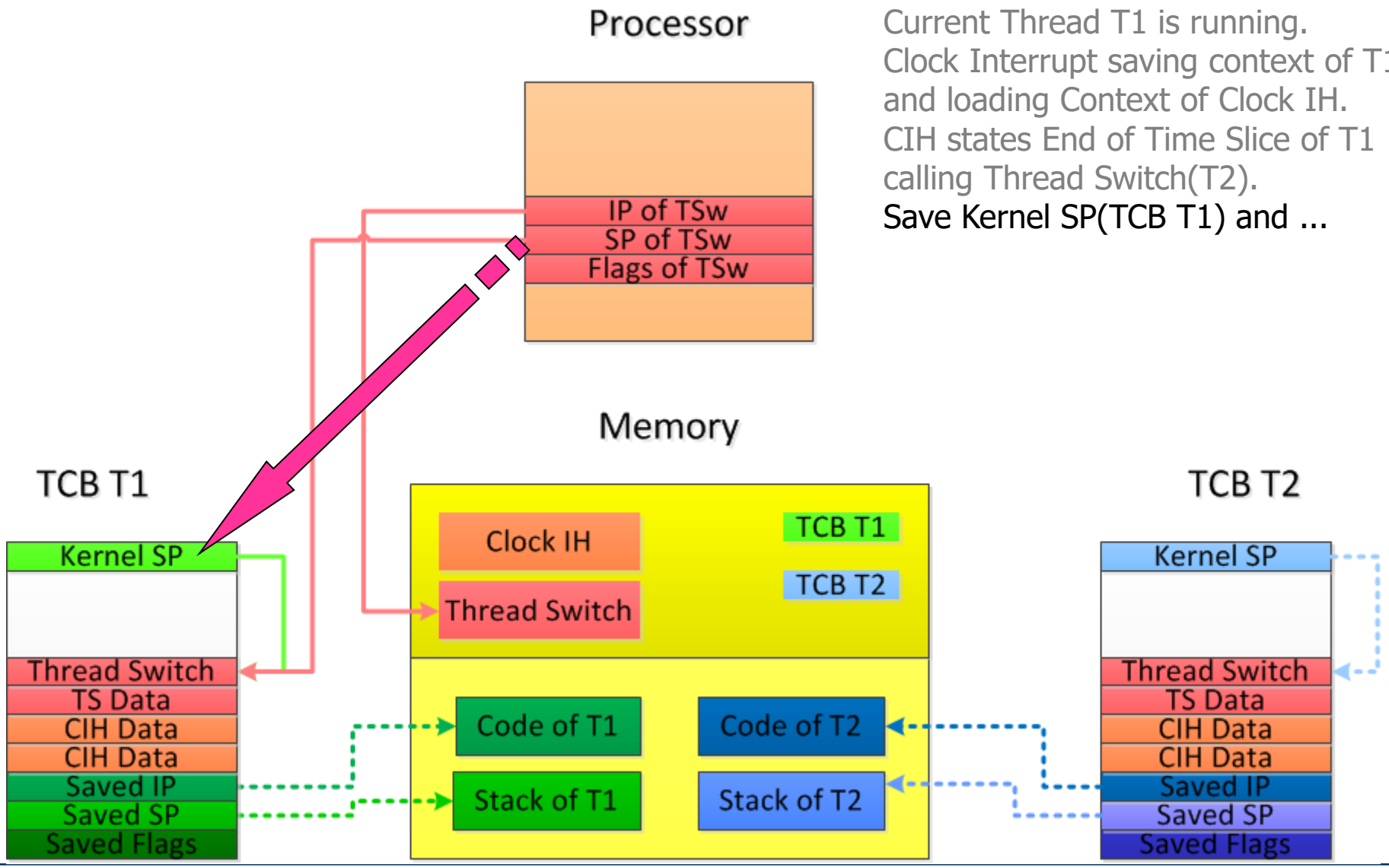
Current Thread T1 is running.
Clock Interrupt saving context of T1 and loading context of Clock IH.

Thread Switch



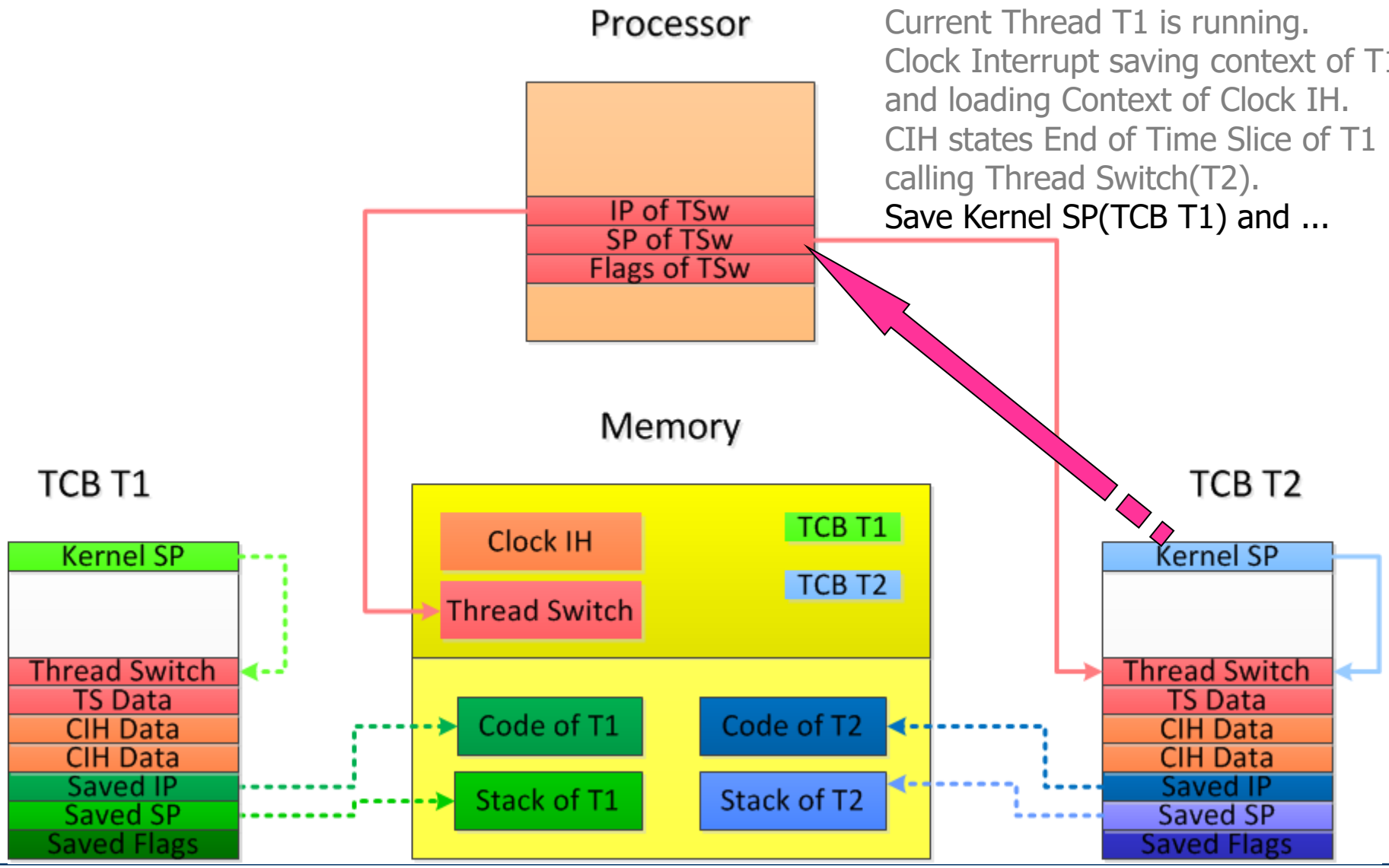
Current Thread T1 is running.
Clock Interrupt saving context of T1 and loading Context of Clock IH.
CIH states End of Time Slice of T1 calling Thread Switch(T2).

Thread Switch

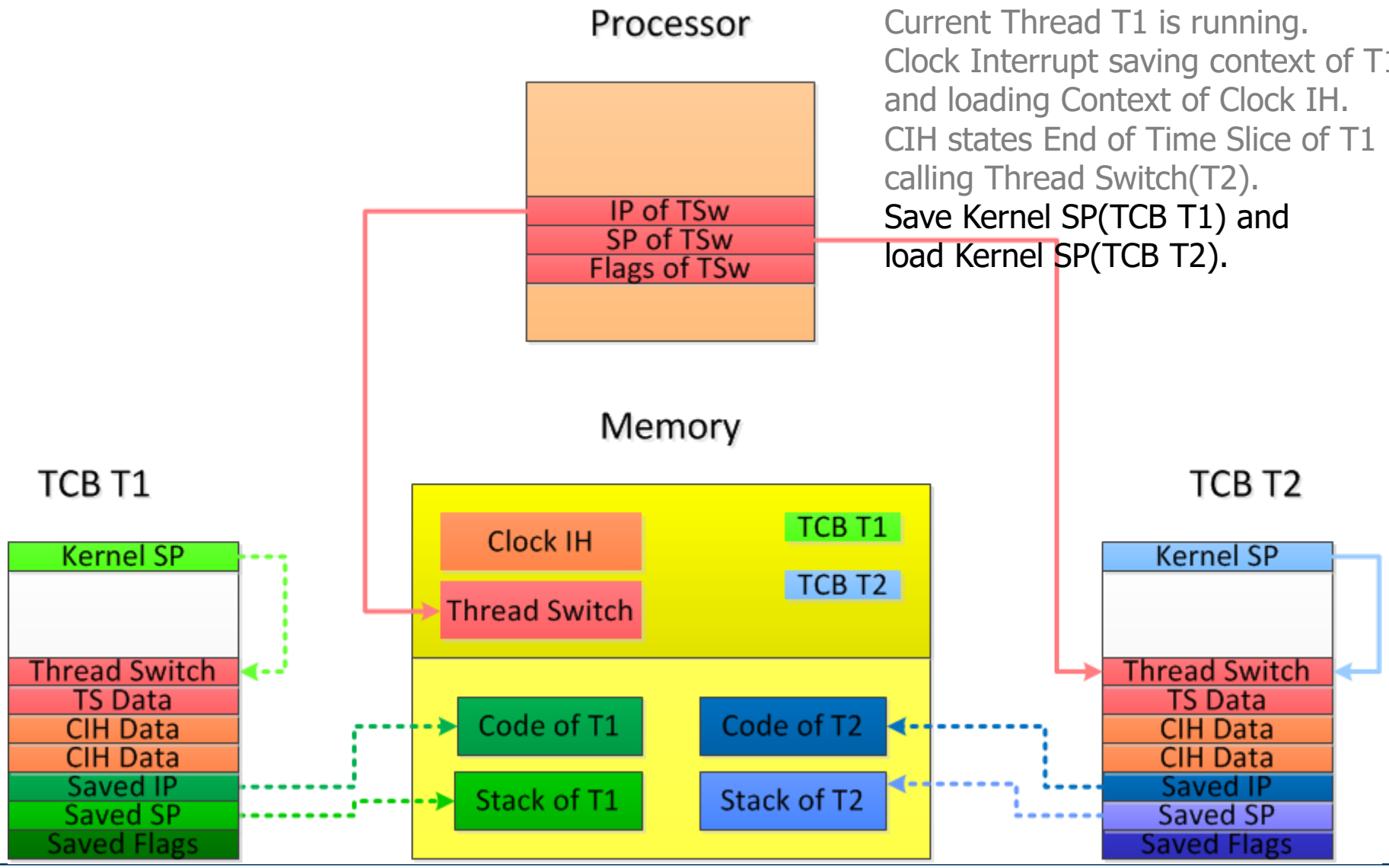


Current Thread T1 is running.
Clock Interrupt saving context of T1 and loading Context of Clock IH.
CIH states End of Time Slice of T1 calling Thread Switch(T2).
Save Kernel SP(TCB T1) and ...

Thread Switch

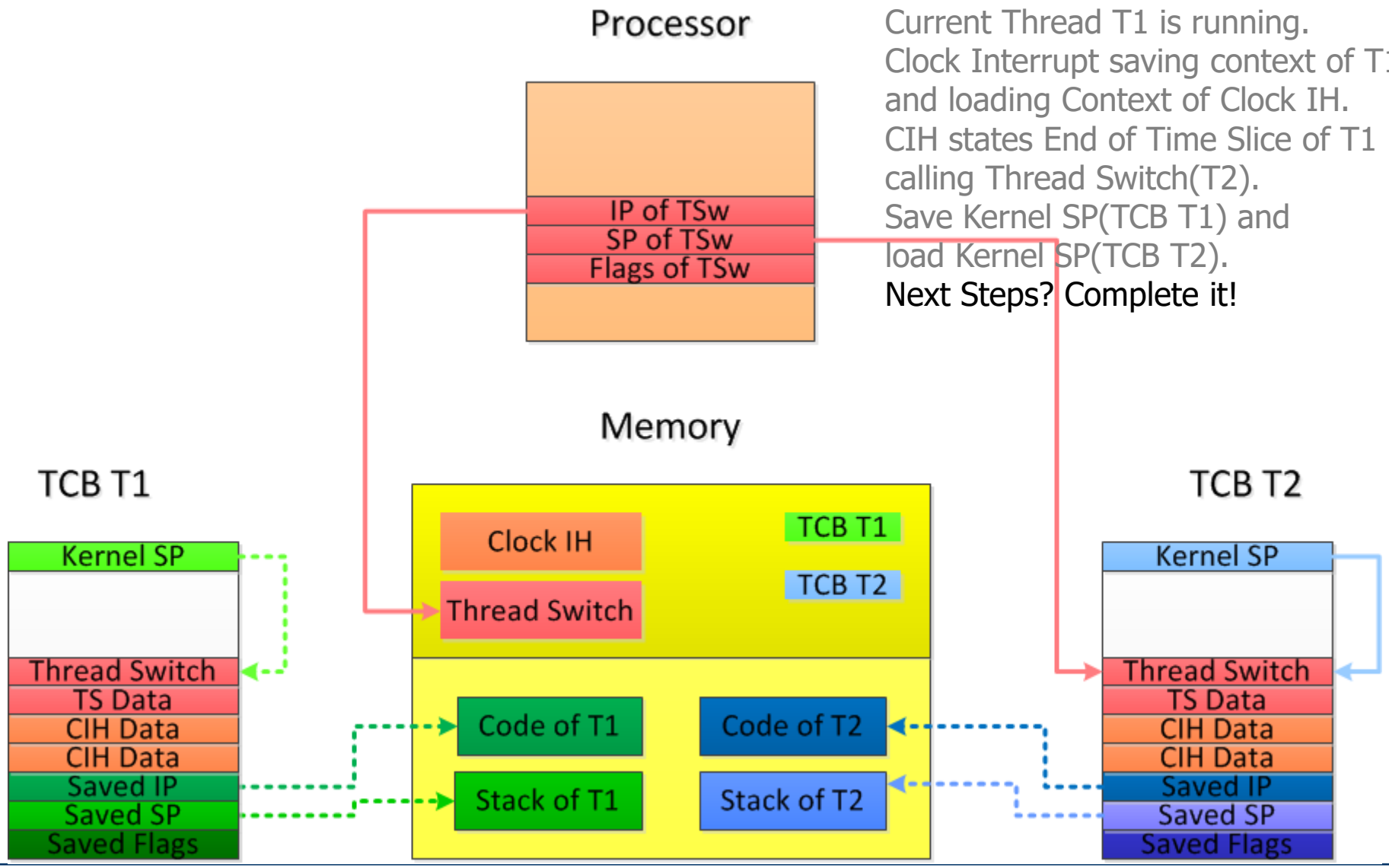


Thread Switch



Current Thread T1 is running.
Clock Interrupt saving context of T1 and loading Context of Clock IH.
CIH states End of Time Slice of T1 calling Thread Switch(T2).
Save Kernel SP(TCB T1) and load Kernel SP(TCB T2).

Thread Switch



Example: Simple Thread Switch on ARM

Freie Universität Berlin

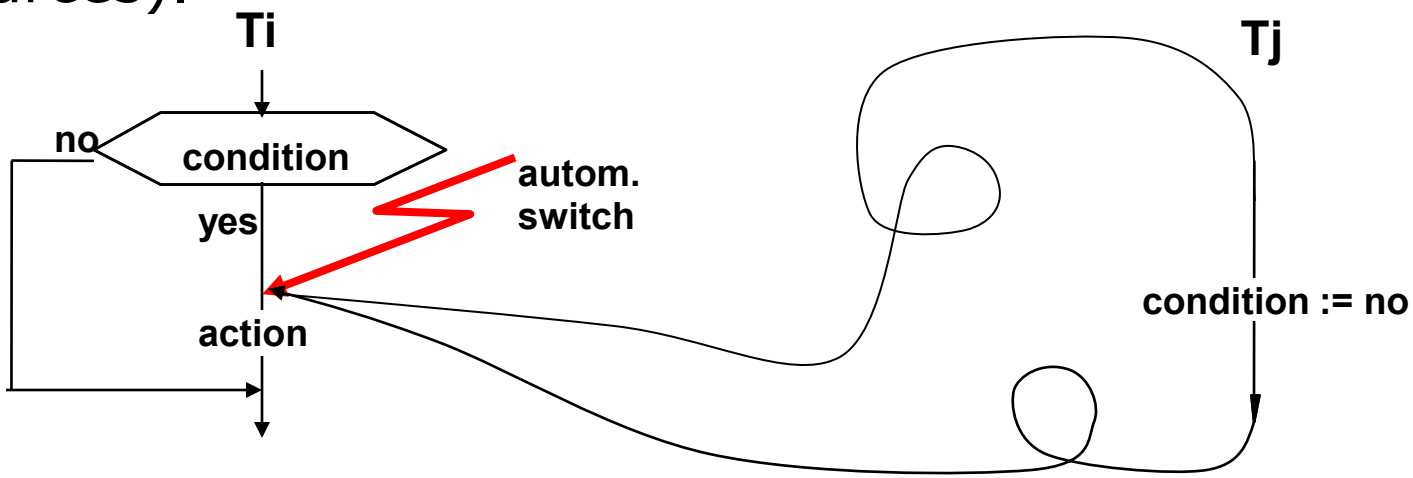
```
; save process state onto stack
STMFD   SP!, {r14}          ; link register for interrupt
STMFD   SP!, {r0-r14}^      ; user registers
MRS     r2, spsr            ; saved CPU state into R2
STMFD   SP!, {r2}           ; and then to stack
STR     SP, [r0]            ; pcb->cpu_state = SP

; switch to other process
LDR     SP, [r1]            ;SP = next_pcb->cpu_state

; restore context
LDMFD   SP!, {r2}           ; CPU state to R2
MSR     spsr, r2            ; and then into saved state
LDMFD   SP!, {r0-r14}^      ; user registers
LDMFD   SP!, {pc}           ; link register for return
                            ; from interrupt
```

3.3 Switching Prevention

- When using automatic switching, which is triggered by an external signal, we have no control of place and time of switching.
- It can happen that switching is triggered at exactly the time when a (voluntary) switching is just taking place. This can lead to unwanted behavior and errors.
- During switching we must make sure that no additional switching is triggered.
- Generally, it can result in errors, when a kernel operation is interrupted by another kernel operation, since kernel operations often work on shared data structures.
- If switching can take place at any point of time, then an arbitrary interleaving of threads and also kernel operations is possible.

Problems due to interleaved execution

- In kernel operations there are places, where depending on a condition some action is performed.
- It cannot be excluded that between the evaluation of the condition and the following action a thread switch takes place and before returning to this thread another threads changes the condition (Example: allocation of resources).



The action is based on wrong assumptions can result in faulty behavior.

Kernel as critical section

- Critical sections are safe if such an interleaving can be excluded.
- When a thread is within a critical section, no other thread is allowed to enter a critical section that is in conflict.
- This is called **mutual exclusion.**
- In OS kernels all possible places of conflict need to be identified and protected accordingly.
- Because in the kernel a large number of those critical sections can be found, we can do it the simple way and regard the entire kernel as a critical section.
- As a consequence we put the whole kernel under mutual exclusion.
- It must be ensured that kernel operations cannot be nested (executed in an interleaved fashion), but are executed until completion.

Realization of kernel exclusion

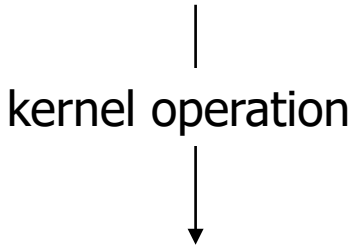
- If the processor does not provide an interrupt mechanism, no interrupt can occur.
- If the processor does provide an interrupt mechanism, we may disable interrupts for the duration of the kernel operation.
- By doing so, we have reduced the second case to the first case.
- But this is possible only with uniprocessor machines.

  In multiprocessor systems we may - even with interrupts disabled - experience a simultaneous execution of different kernel operations that access memory in an interleaved fashion.
- In this case we must enforce mutual exclusion for kernel operations explicitly by a central kernel lock.
- This is, however, no appropriate solution for many-core systems, since it would lead to a situation where parallel threads have to line up at the entry of the kernel.

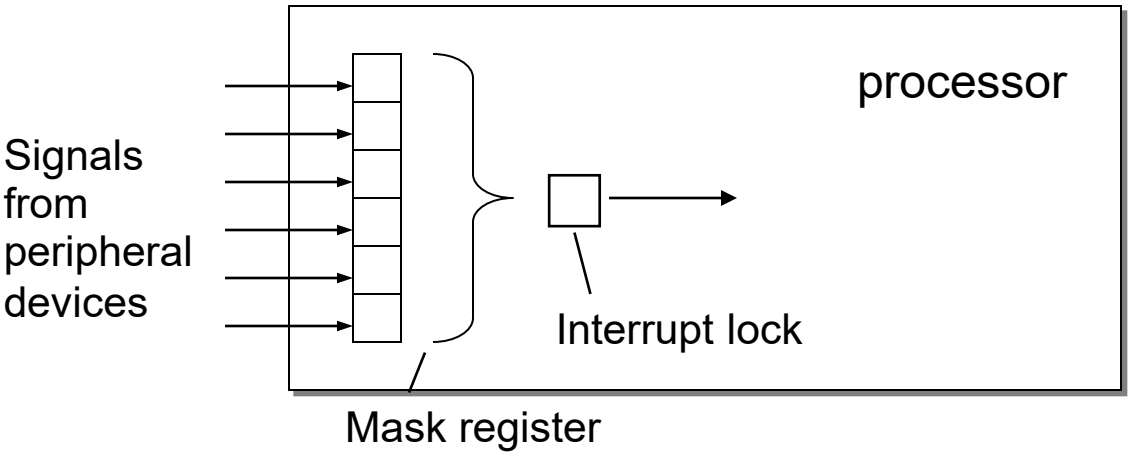
Kernel exclusion

- The realization of kernel exclusion obviously depends on
  - interrupts being possible or not
  - multiple processors or not
- Therefore we distinguish four cases :
  - Case 1: single processor system without interrupts
  - Case 2: single processor system with interrupts
  - Case 3: multiprocessor system without interrupts
  - Case 4: multiprocessor system with interrupts

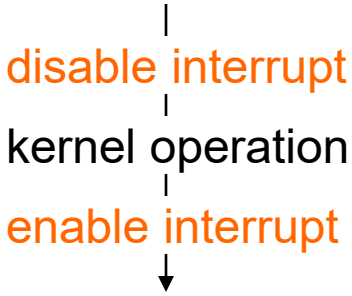
- Case 1 does not need any precautionary measures since there is no reason to leave a kernel operation before completion.



Case 2: single processor system with interrupts

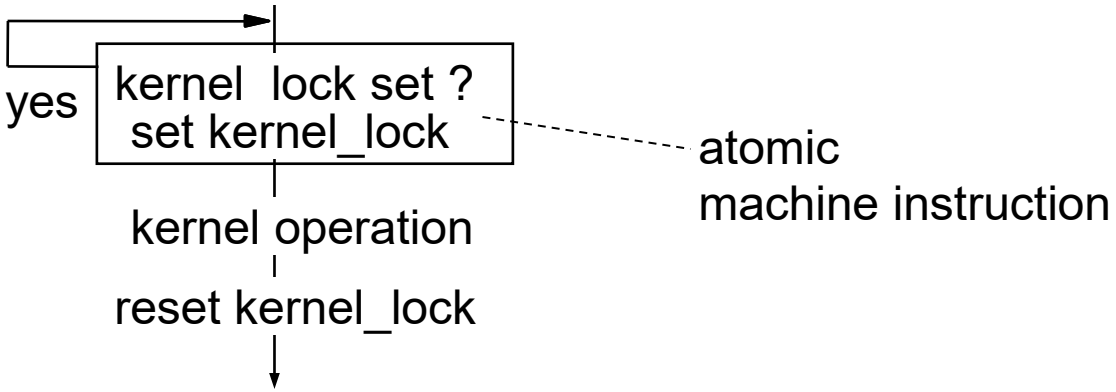


- The kernel operation is bracketed by an *disable interrupt* and an *enable interrupt*.



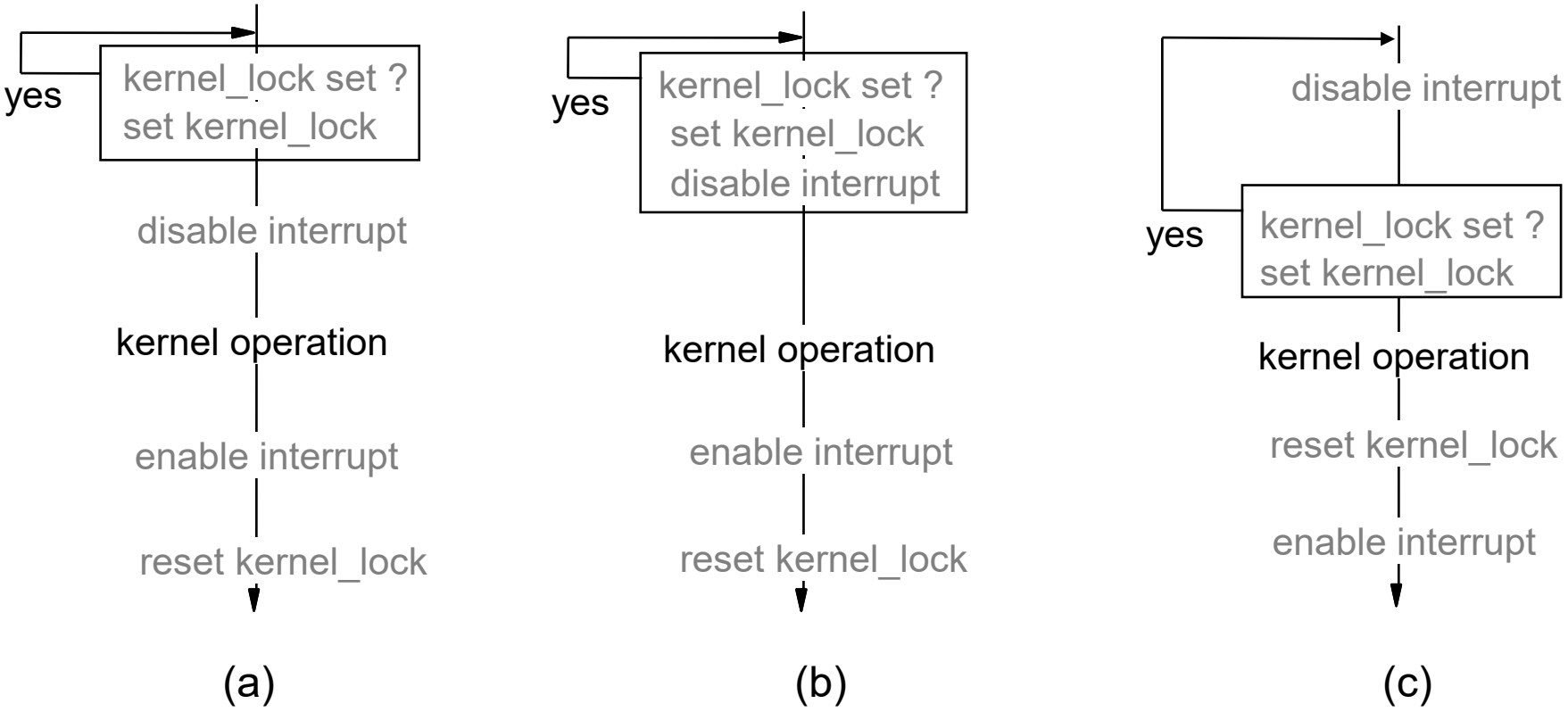
Case 3: Multiprocessor system without interrupts

- Using an atomic test-and-set instruction, we can come up with a simple solution:
  - If the kernel lock is busy, we repeatedly check its value in a mini loop.
  - This is called **busy waiting**.
  - Such a lock is called a **spin lock**.
  - Busy waiting means some waste of compute capacity that can be tolerated since kernel operation are usually short.



Case 4: Multiprocessor system with interrupts

- Here both techniques kernel lock and disabling interrupts must be employed.
- So we want to discuss the following three solutions:



Discussion of solutions

- Solution A:

  Here we have to consider that interrupt handling is also a kernel operation that also needs the kernel lock.

  If there is right after setting the kernel lock an interrupt, the interrupt processing would try to acquire the kernel lock in vain.

  The thread would be stuck at that point.

- Solution B

  An operation that acquires the kernel lock and disables the interrupts in one atomic operation would be ideal.

  Unfortunately, this is not offered by today‘s processors.

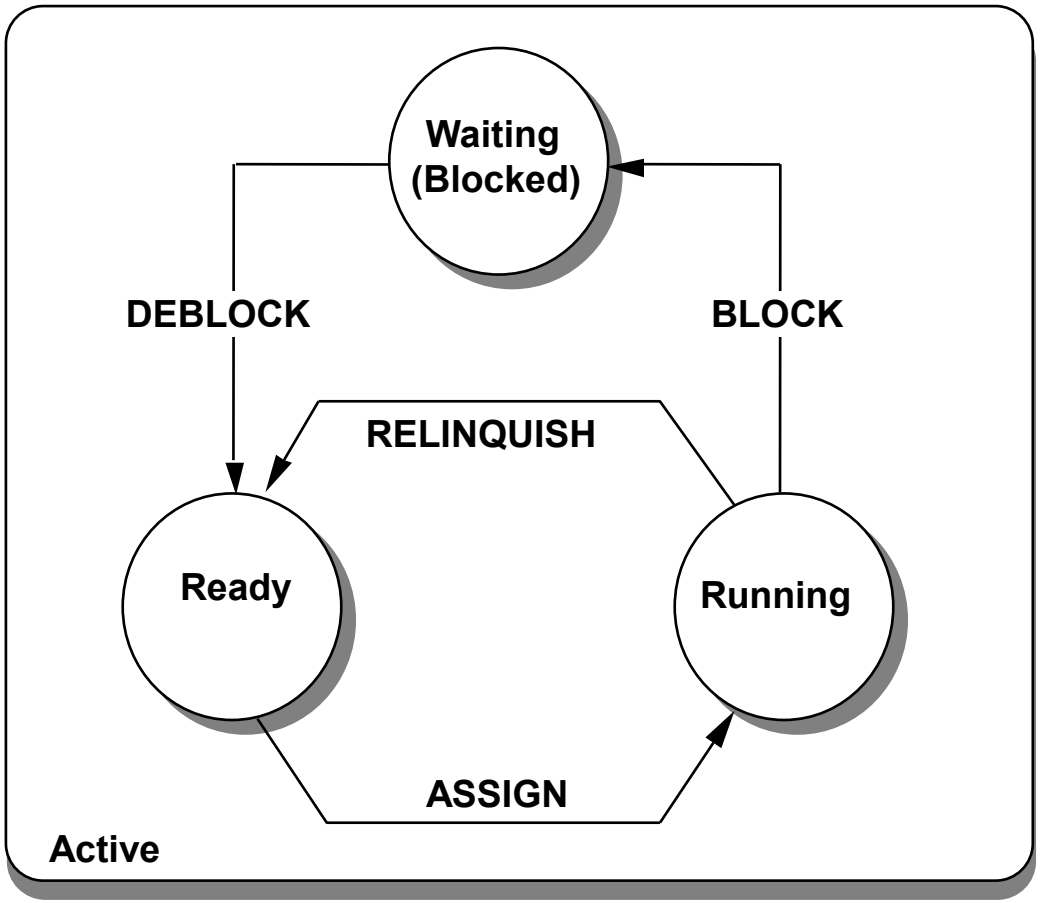
- Solution C

  Thus, we first have to disable the interrupts and then acquire the kernel lock. Solution C is the correct one.

3.4 Thread states

- We used the conditioned switch to release the processor, when the current thread could not continue execution for some reason.
- In this case we switch to another thread. This new thread, however, may also be blocked, e.g. because it waits for the completion of an I/O operation. If we switch to such a thread, the processor would be again immediately released.
- This way, we could try one thread after another until we finally may detect a thread that is ready to resume execution.
- To speed up the search for a ready thread, we combine threads according to their state (resumable, not resumable) to thread subsets.
- If we also consider the currently running thread, we can distinguish three different states:
  - State running: threads that are currently executed on the processor
  - State ready: threads that are ready to be executed but have to wait for the processor to become free.
  - State waiting: threads that are blocked since they wait for some (external) event

Thread states and their transitions



State change operations

For all state changes, the corresponding kernel operations are available:

- **Relinquish**

  Voluntary switching to another thread. The currently running thread remains executable, i.e. its state changes to "ready".

- **Assign**

  Taking the next thread from the ready set to resume its operation on the processor.

- **Block**

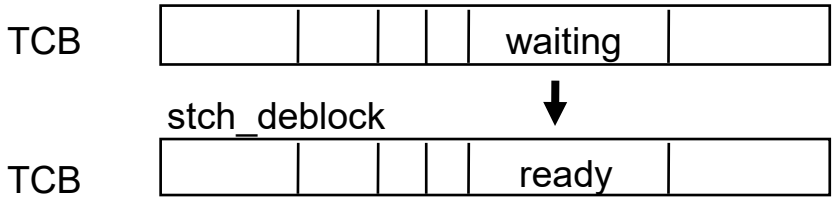
  Leave the processor since some condition does not hold (conditioned switch). Execution must not resumed until condition is met. Thread switches to state "waiting" or "blocked".

- **Deblock**

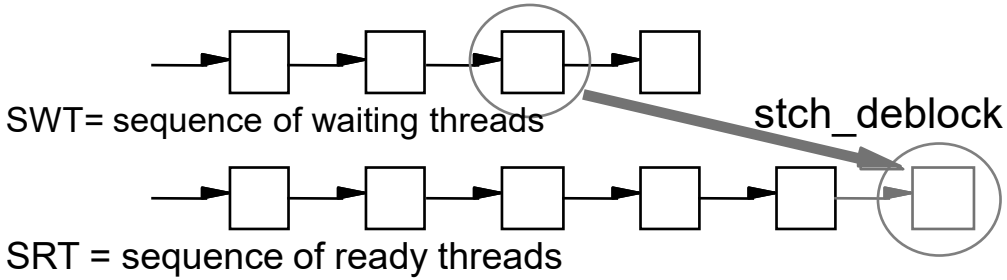
  If the event happened for which the blocked thread waited it changes its state from waiting to ready and is inserted into the set of ready threads.

State change operations

- When executing the state transitions we have to distinguish:
  - the state change operation themselves,
  - other actions that may be necessary related to the state change.
- The pure state change operations depend on how we implement the thread states. Example: stch_deblock ("stch" for "state change")
  - Thread state as attribute in the TCB

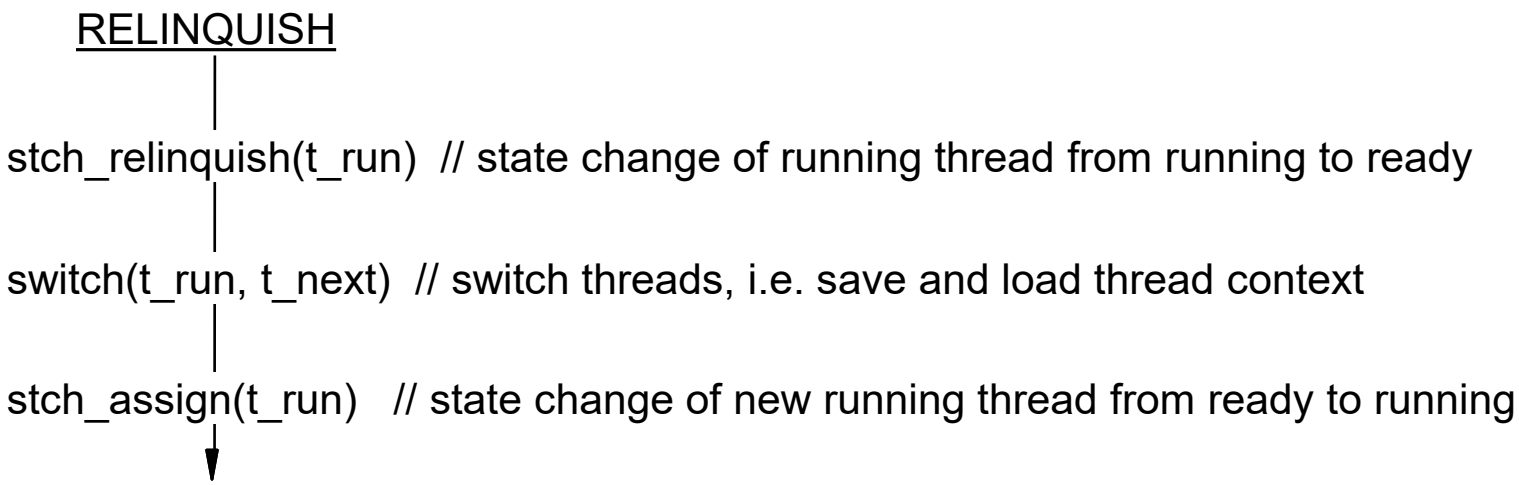


  - Thread state as membership in list



Thread switch as kernel operation

- Besides the state change as an operation at the TCB data structure we also perform the switch operation itself.
- "Relinquish" as a kernel operation may look like this:

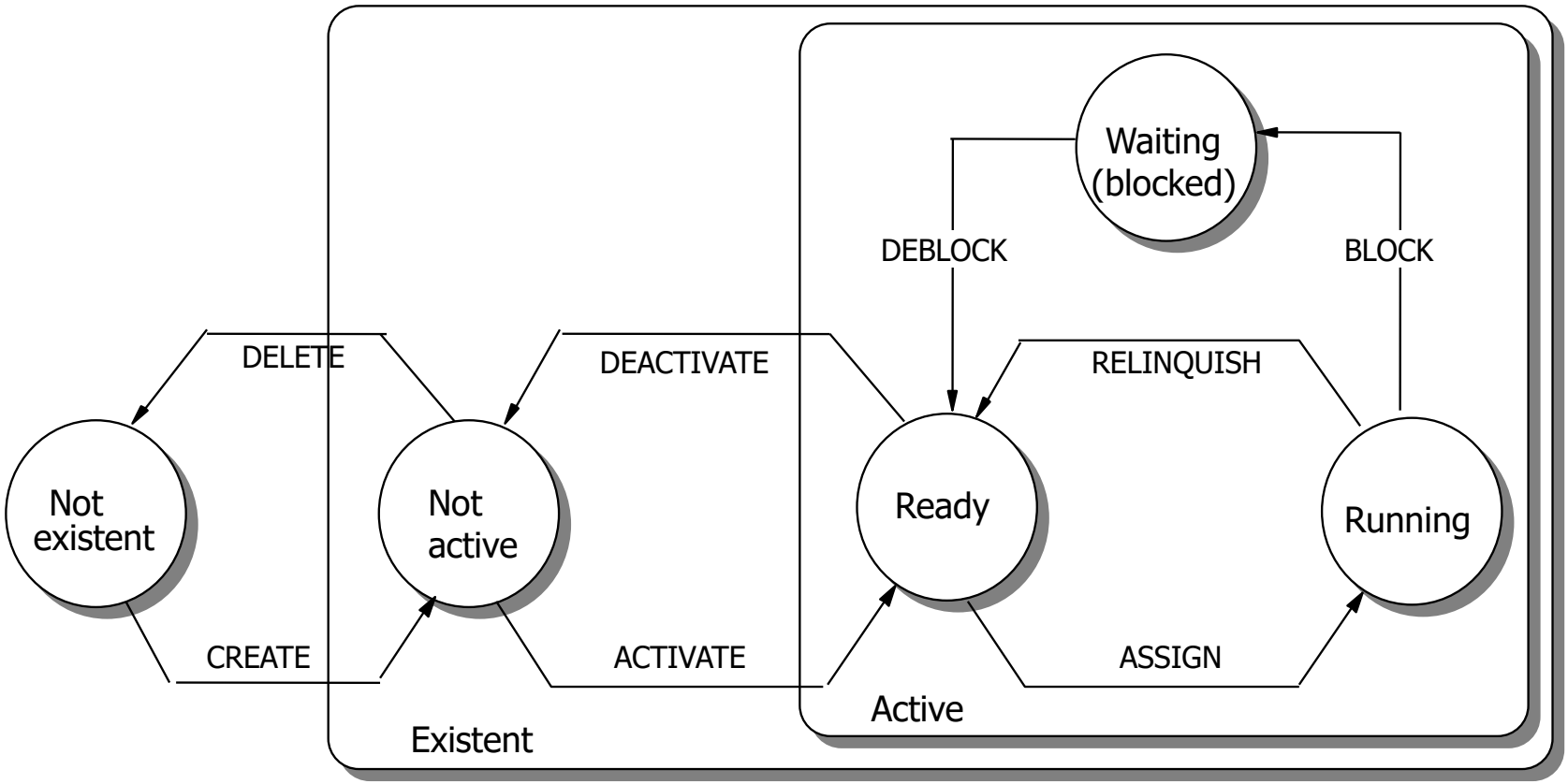


Dynamic System

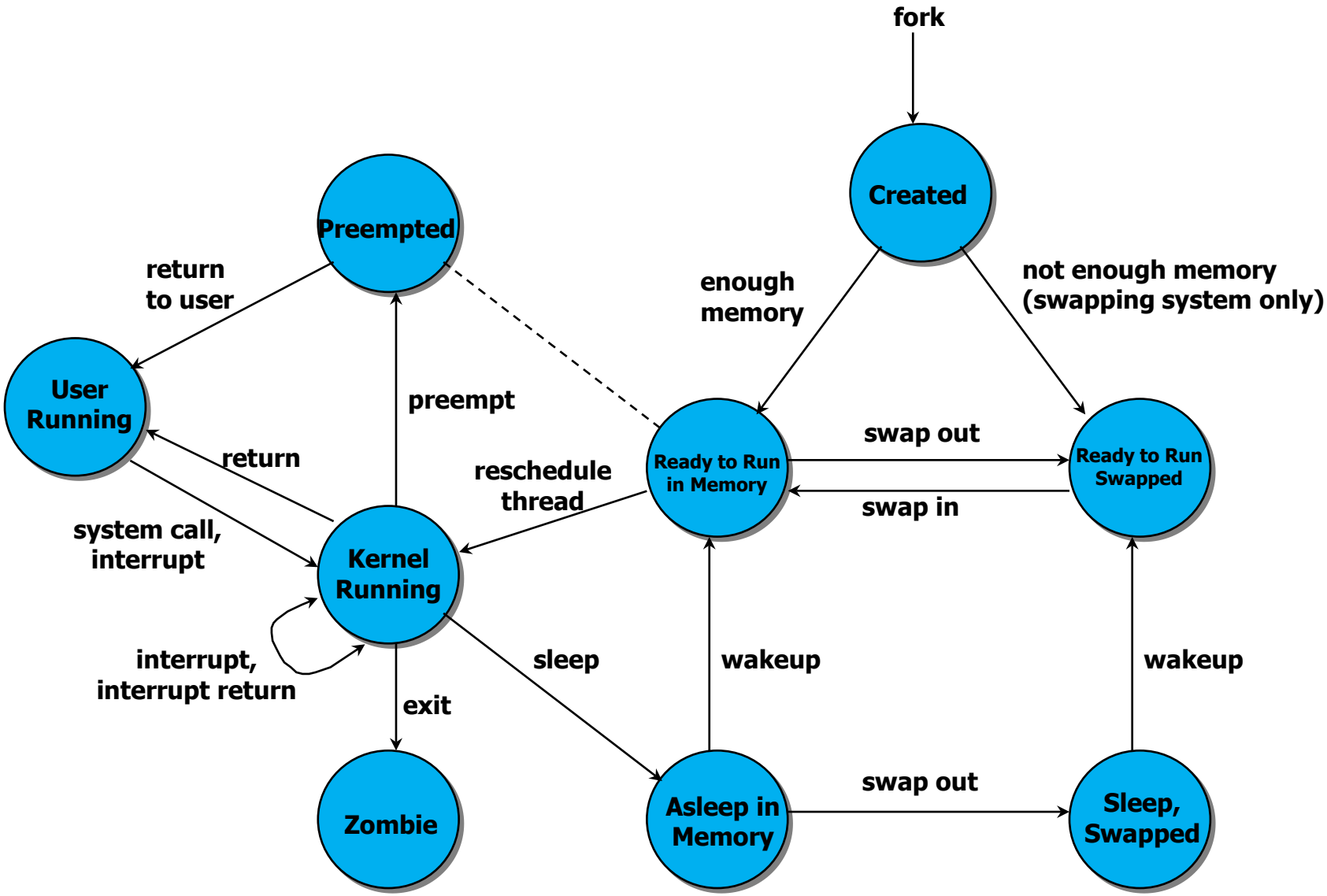
- In dynamic systems the set of threads is variable.

- For dynamic systems, we need the following operations
  - Activate / Deactivate
    - A thread may be defined (there exists a TCB), even code and data segment may be available, but the thread "rests" , i.e. it is not active.
    - We distinguish between *active* and *inactive* threads.
    - Transition between these states are possible by means of the operations *activate* and *deactivate*.
  - Create / Delete
    - In a second step we have to assume that threads are not yet available at system start and need to be created (and deleted) explicitly.
    - For that we provide the operations *create and delete.*

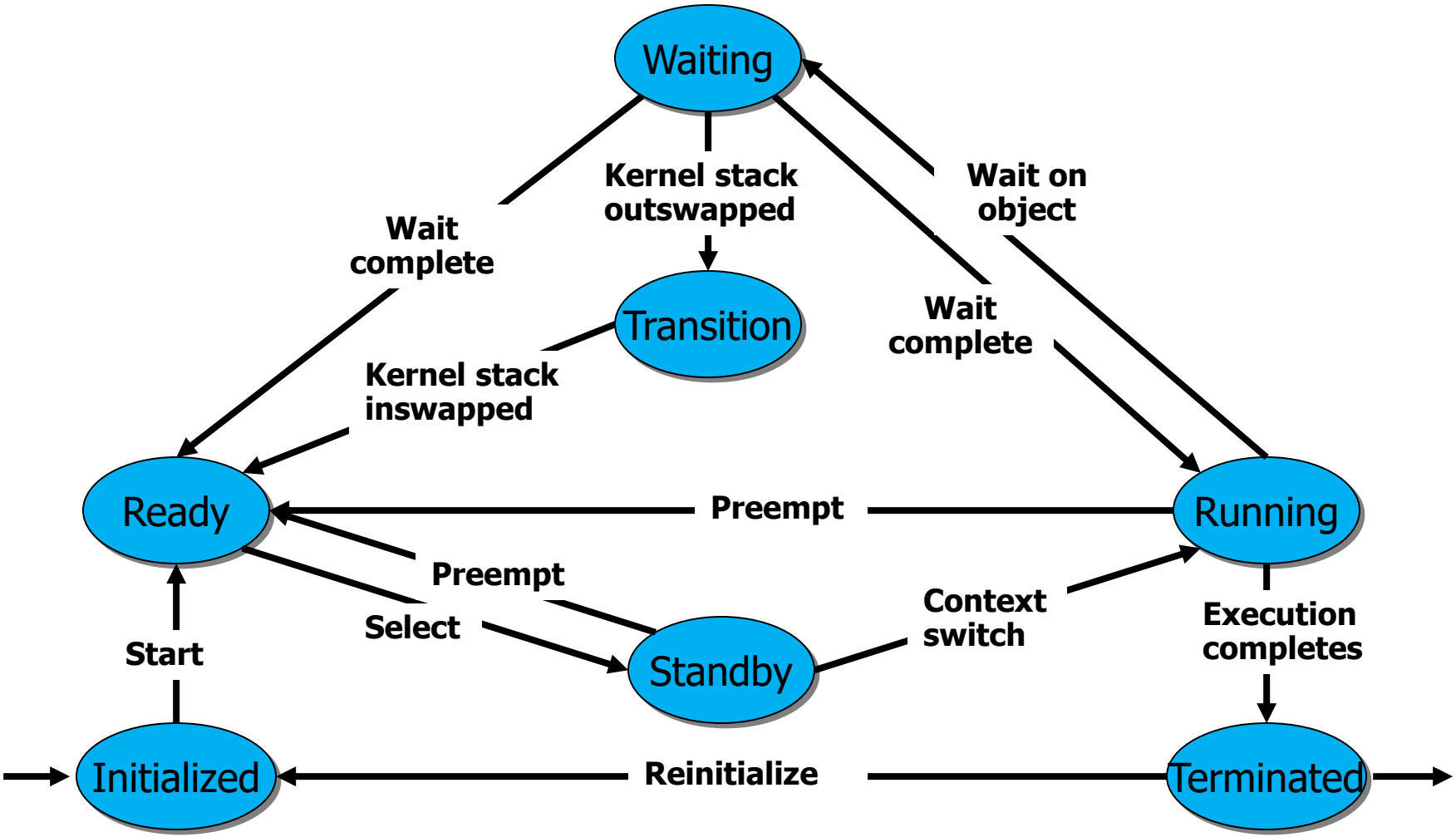
Complete state diagram



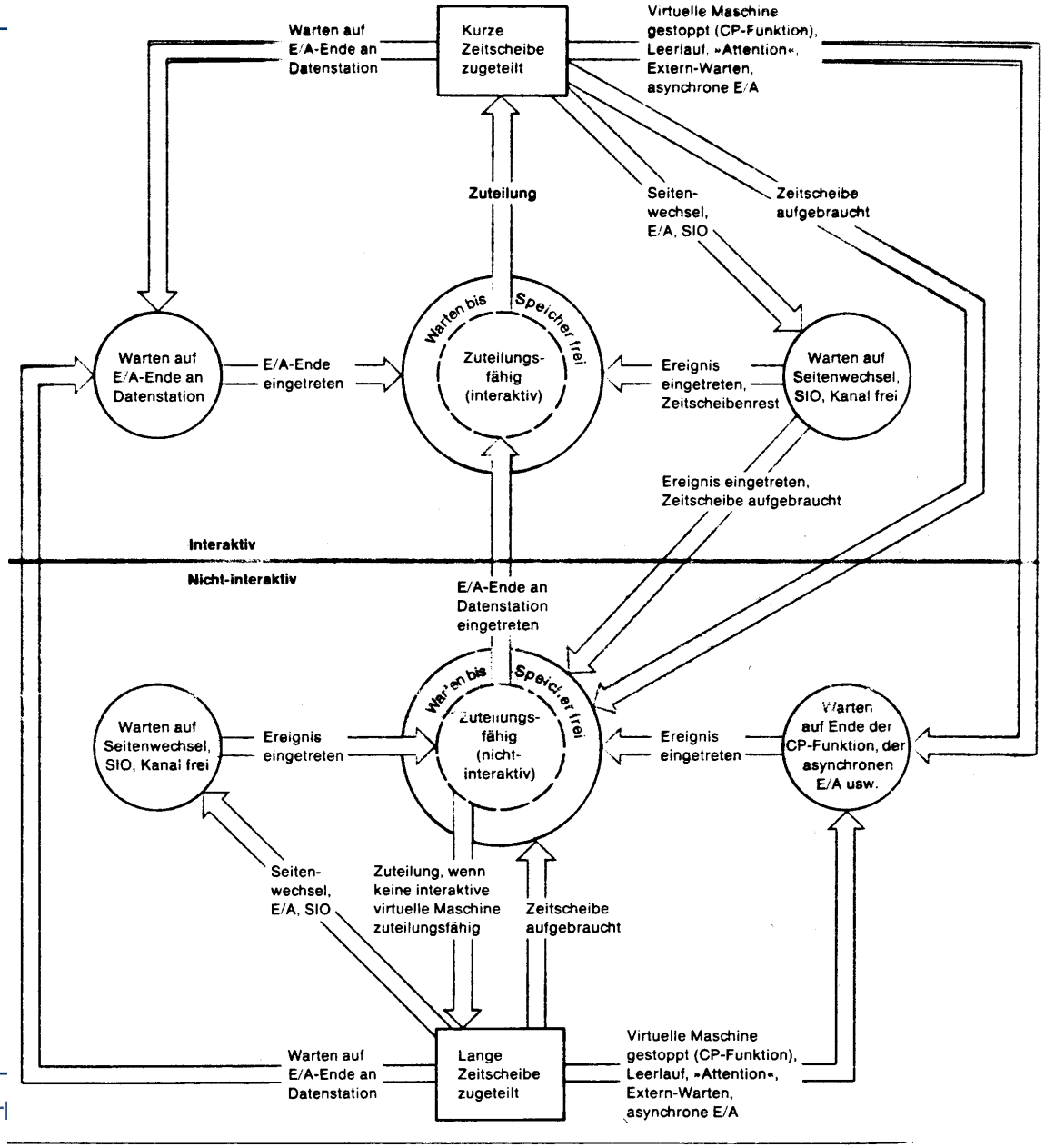
Thread states in Unix



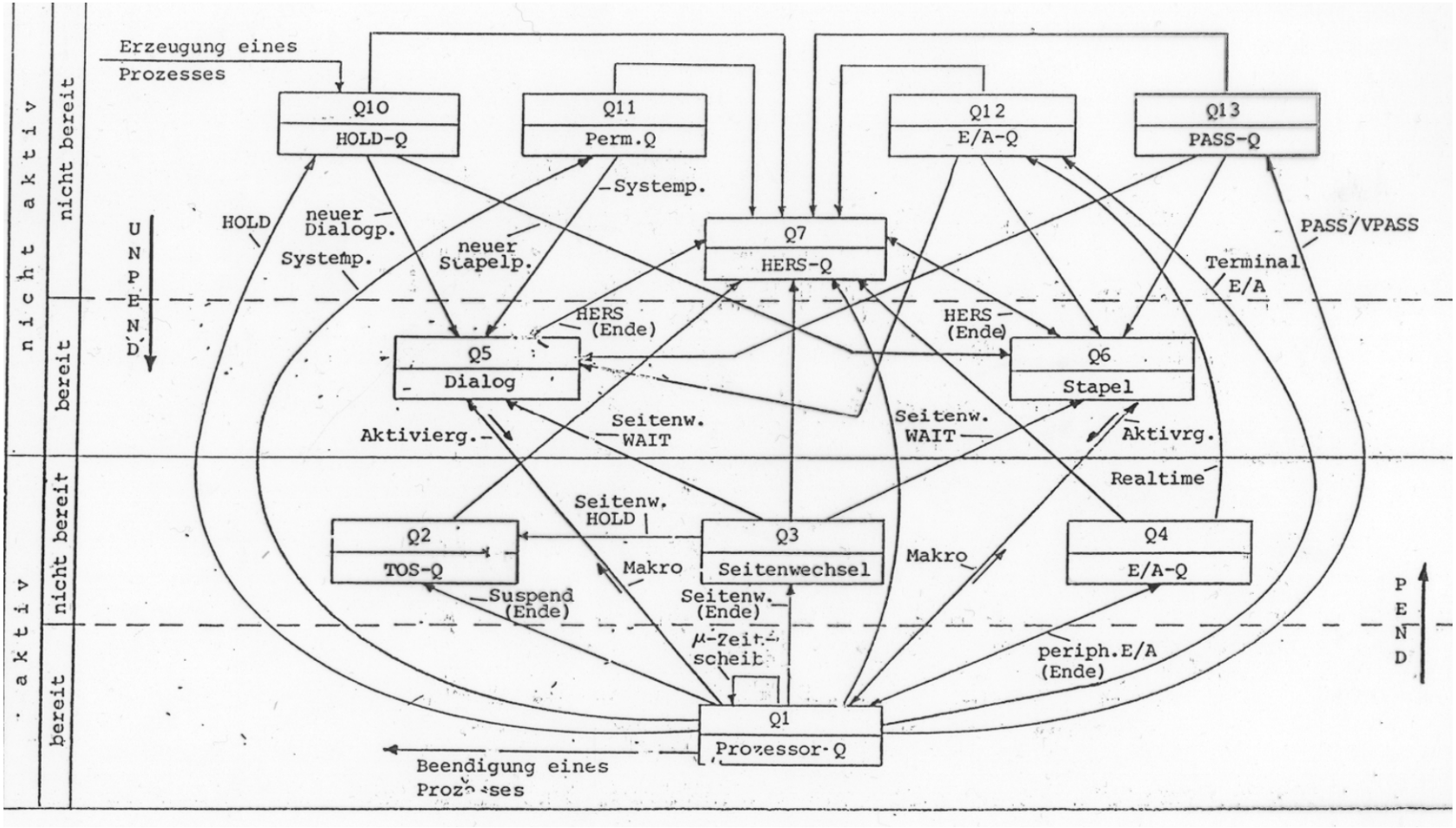
Windows thread states



Process states IBM VM/CMS



Process states Siemens BS 2000



3.5 Preemptions and Idling

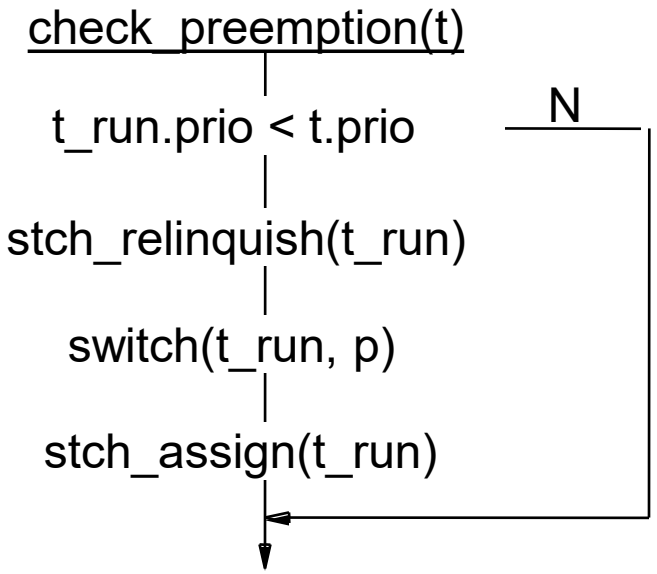
- Up to now, a thread remains being executed until
  - it voluntarily gives up the execution (relinquish),
  - it is forced to give up execution by a clock interrupt,
  - it cannot continue due to some condition it is waiting for.
- In many application areas not all activities (and the threads as their representatives) are of equal importance or urgency which leads to the concept of priorities.
- When using priorities, we want to make sure that at any time the thread with the highest priority is being executed.
- The consequence is that we do not wait until one of the above situations for thread switch occurs but immediately switch if a thread with a higher priority shows up, i.e. enters the ready queue.
- We say, the current thread is being **preempted** by the thread with the higher priority.

Check for preemption

- The priority rule requires that no ready thread may posses a higher priority than a running one.
- If we assume that this condition currently holds and the priorities are constant a violation of that rule can only happen when a new thread enters the ready queue.
- According to our state diagram there are exactly three transitions into the ready state.
  - relinquish
  - deblock
  - activate

Within these operations we have to check whether the thread performing the transition has a higher priority than the currently running ones.

If this is the case we switch to the more urgent one.

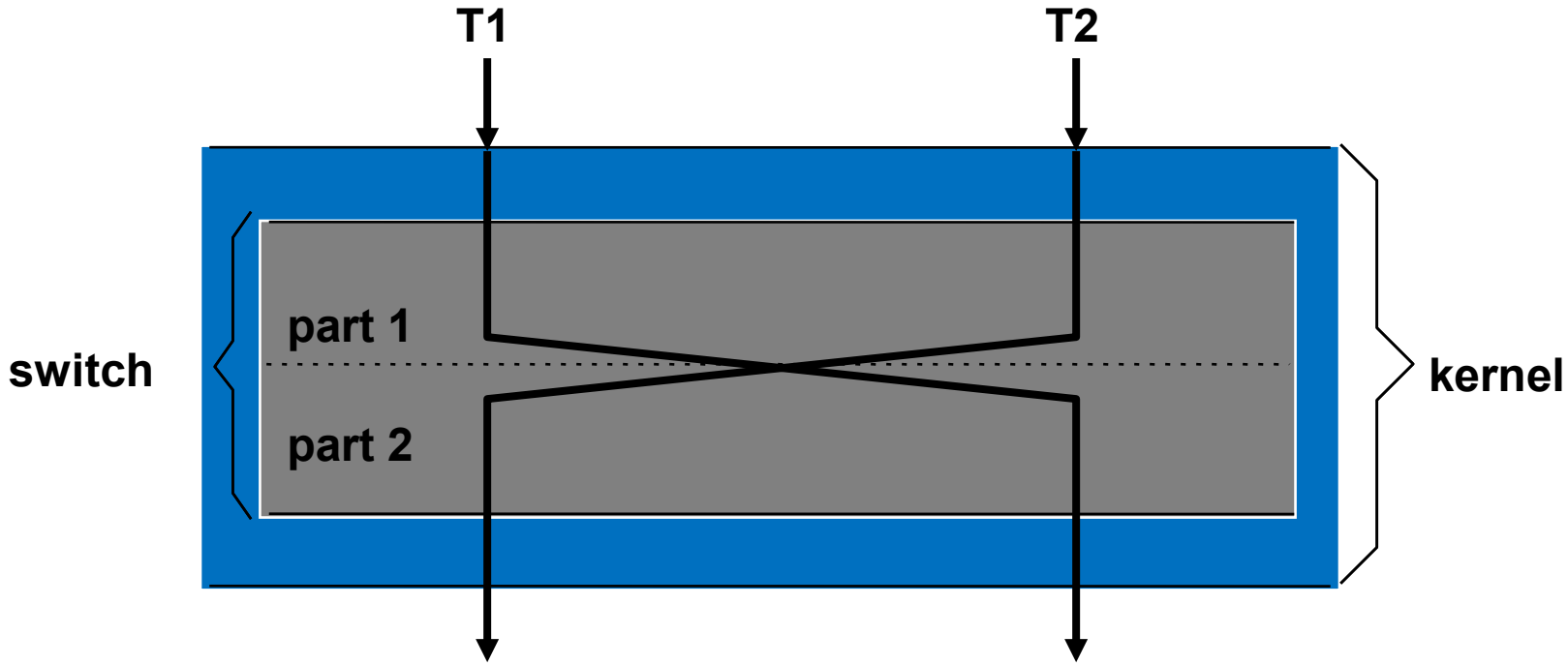


Idle problem

- In operating with waiting states it may happen that all threads are blocked since they are waiting for something. In this case the processor has nothing to do.
- To handle this situation in an elegant and consistent way we simply introduce an **idle thread.**
- It must have the following properties:
  - must not stop (cyclic thread, endless loop),
  - lowest priority (to be preempted by any real thread),
  - must be preemptable at any time.
- Examples
  - Empty loop — while true do; (usually wastes energy)
  - Dynamic Stop — If available: Special machine instruction that does not access memory but reacts to external signals (such as `halt` or entering C-states on x86, dynamically disables parts of processor)
  - Insertion of useful housekeeping tasks: Checks, reorganizations (e.g. garbage collection)

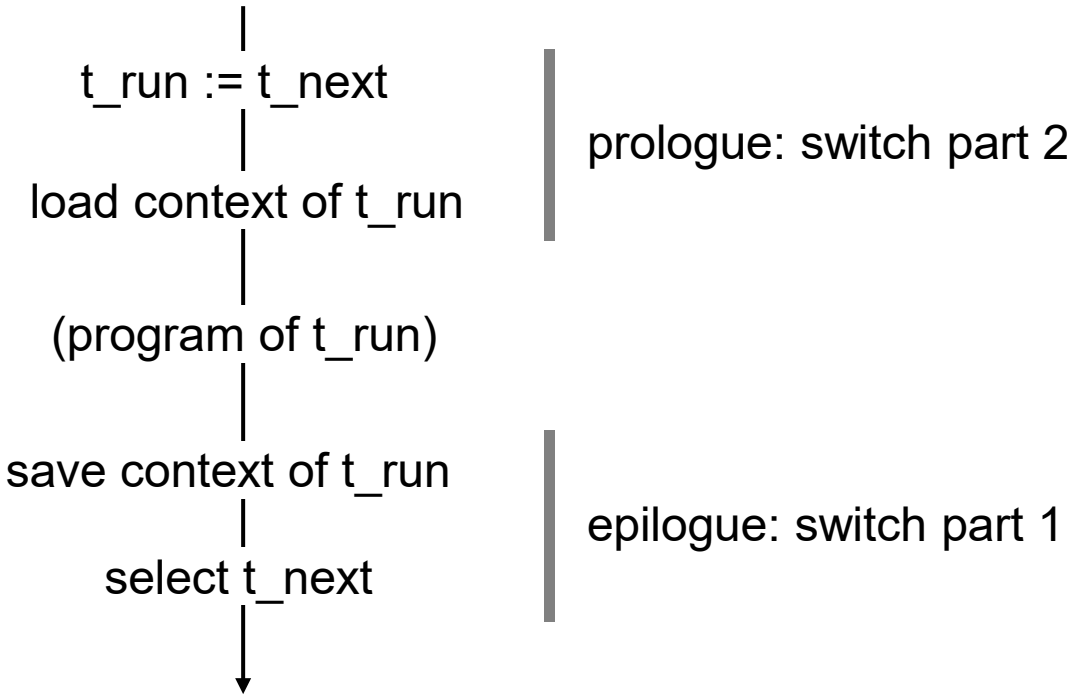
3.6 Initialization

- How can we switch to a thread for the first time?
- Each "entrance" to a thread takes place via the procedure "switch".



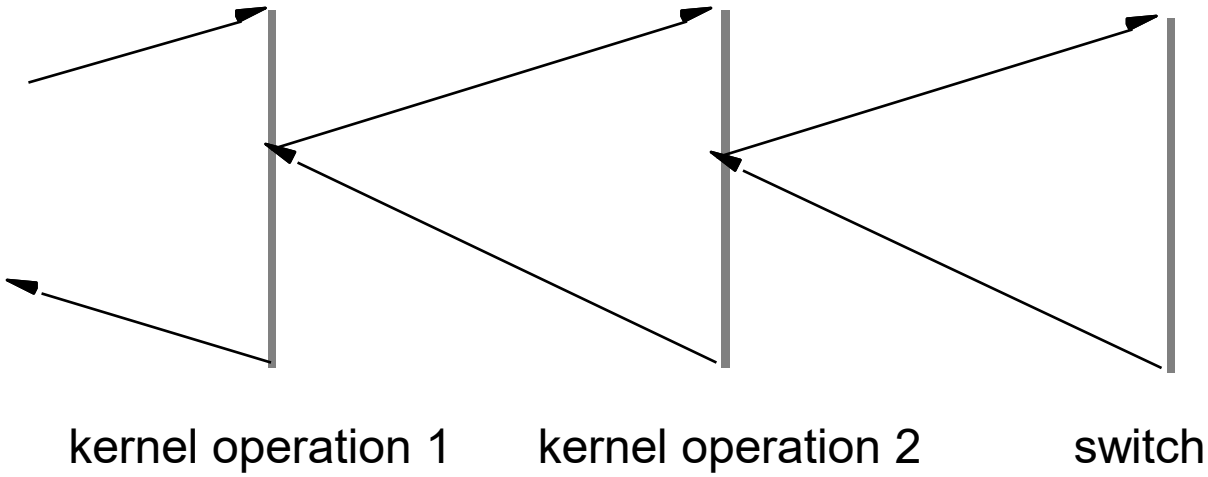
Initialization problem

- **The thread starts and ends in the "kernel of the kernel", in the procedure "switch".**

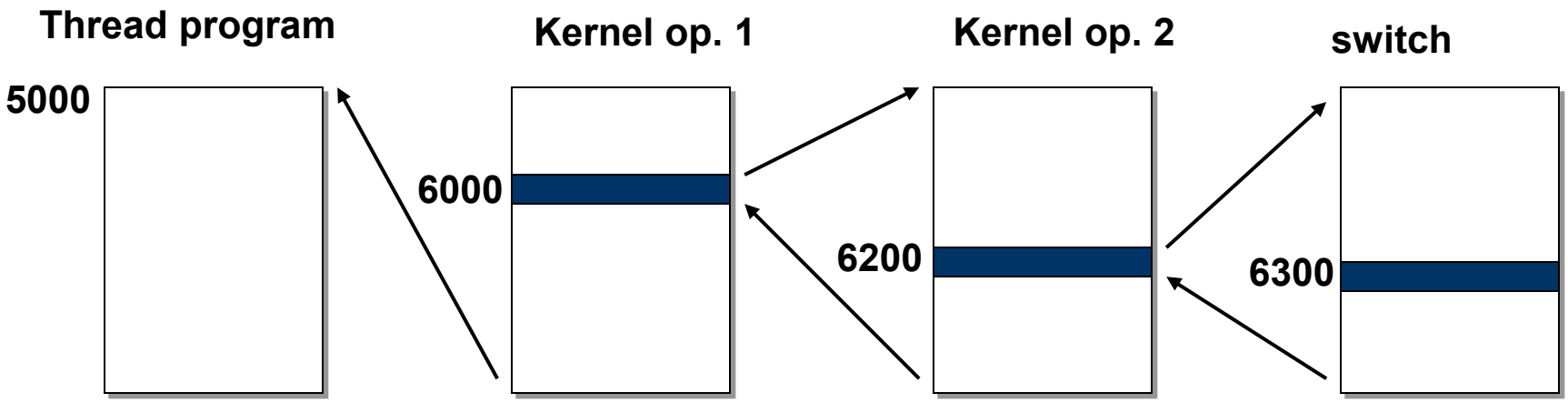


Initialization problem

- We have to initialize the thread (thread control block, stack) such that it looks as if it is just "in the middle" of the procedure switch.
- The switch, however, may be entered depending on the structure of the kernel after some other procedure calls.



Thread initialization (Example)



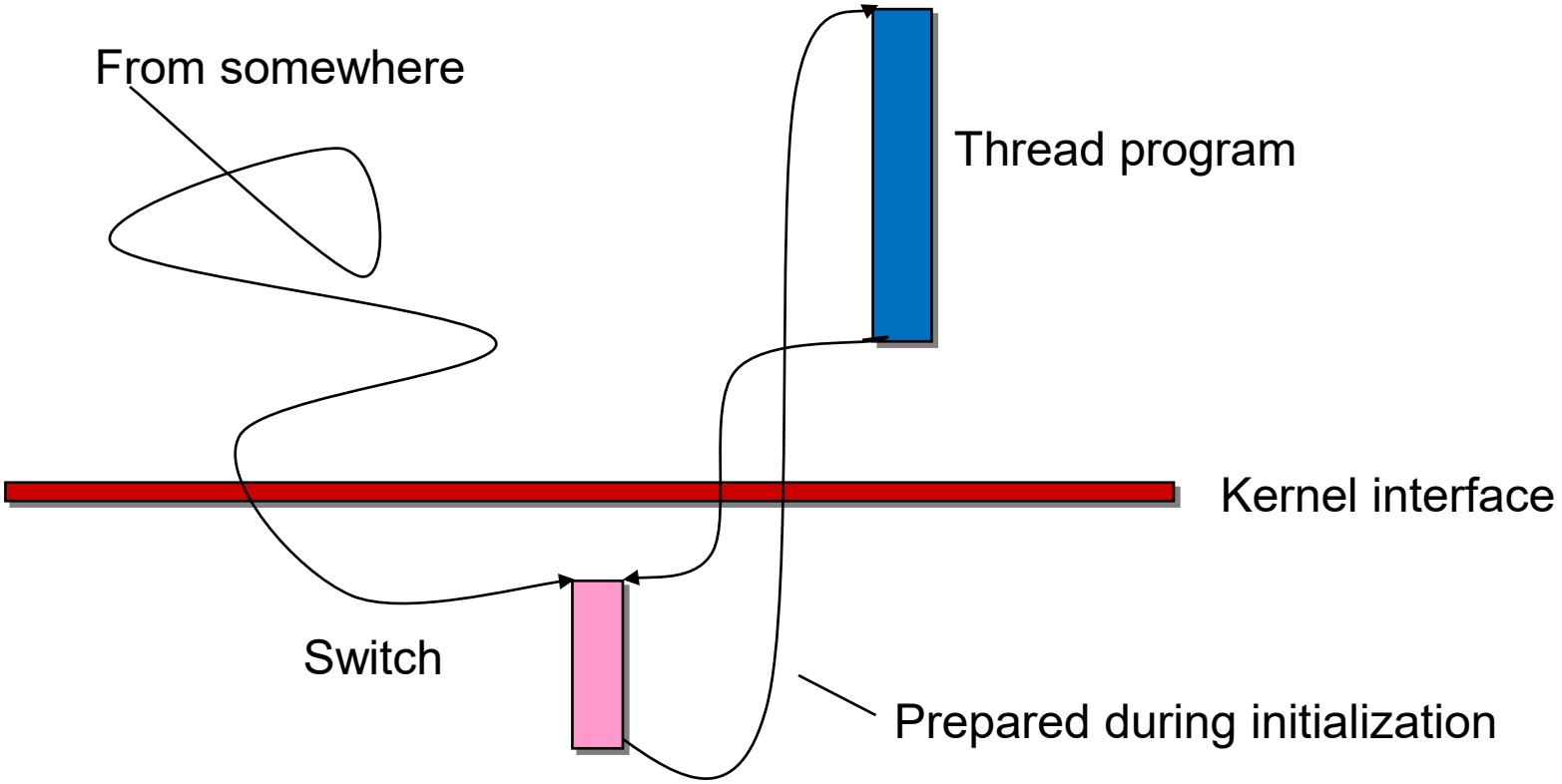
Thread control block (TCB)

	:
Instr. counter	6300
	:
Stack base	1000
Stack end	2000
Stack pointer	1003
	:

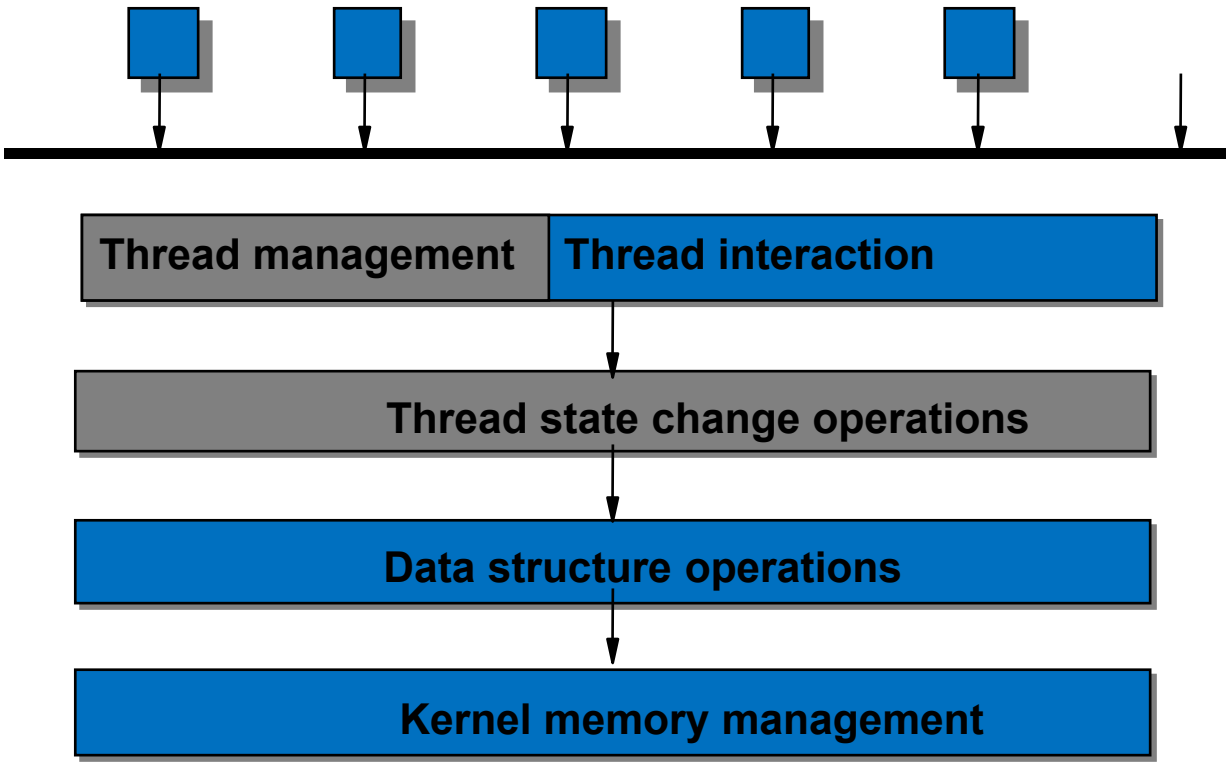
stack

1000	5000
1001	6000
1002	6200
1003	
	:

First entrance to a thread



3.7 Kernel operation for thread management



Example: Kernel operations for thread management I

Freie Universität Berlin

```
kernel module thread management;
   export <thread operations>;
   import <state change operations>;
   procedure CREATE_THREAD(P: thread);
      begin
      { create TCB for thread P;}
      STCH_CREATE(T)
      end;
   procedure DELETE_THREAD(P: thread);
      begin
      STCH_DELETE(T);
      { delete TCB of thread P;}
      end;
   procedure SET_ATTRIBUTE(P: thread; A: attribute; V: value);
      begin
      P.A := V
      end;
   procedure READ_ATTRIBUTE(P: thread; A: attribute; V: value);
      begin
      V := P.A
      end;
```

Example: Kernel operations for thread management II

Freie Universität Berlin

```
procedure RELINQUISH_THREAD(T: thread);
    begin
    STCH_RELINQUISH(T);
    SWITCH(P,T_NEXT);
    STCH_ASSIGN(T_RUN)
    end;
procedure BLOCK_THREAD(WT: sequence of thread; T: thread);
    begin
    if T = T_RUN then
        begin
        STCH_BLOCK(WT,T);
        SWITCH(T,T_NEXT);
        STCH_ASSIGN(T_RUN)
        end
    end;
procedure DEBLOCK_THREAD(WT: sequence of thread; T: thread);
    begin
    STCH_DEBLOCK(WT,T);
    { check for preemption}
    end;
```

Example: Kernel operations for thread management III

Freie Universität Berlin

```
procedure ACTIVATE_THREAD(T: thread);
   begin
   { initialize TCB and stack;}
   STCH_ACTIVATE(T);
   { check for preemption}
   end;
procedure DEACTIVATE_THREAD(T: thread);
   begin
   if T /= T_RUN
       then begin
            case T.STATE
               waiting: STCH_DEBLOCK(waiting queue of T,T);
               ready:
               end;
            { delete objects created by thread T}
            { finish all activities of T}
            STCH_DEACTIVATE(T);
            end
       else begin
            STCH_RELINQUISH(T);
            { delete objects created by thread T}
            { finish all activities of T}
            STCH_DEACTIVATE(T);
            SWITCH(T,T_NEXT);
            STCH_ASSIGN(T_RUN)  // !!!
            end
   end;
end thread management.
```

3.8 Threads in Programming Languages

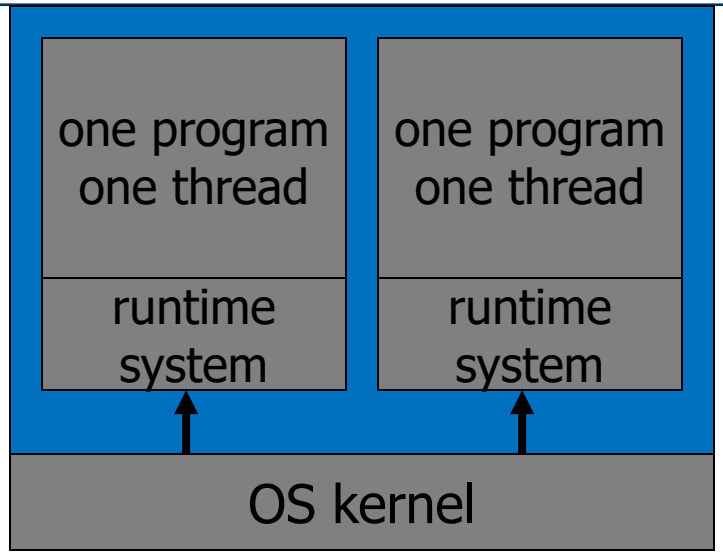
- Many modern programming languages contain a thread concept to formulate concurrent activities within programs (e.g. Java Threads).
- Or there are programming libraries, that extend a programming language by a thread concept.

- What is the relation of these threads to OS threads?
- How does the OS support those threads?

Threads in OS and Programming Languages

Classic multiprogramming:

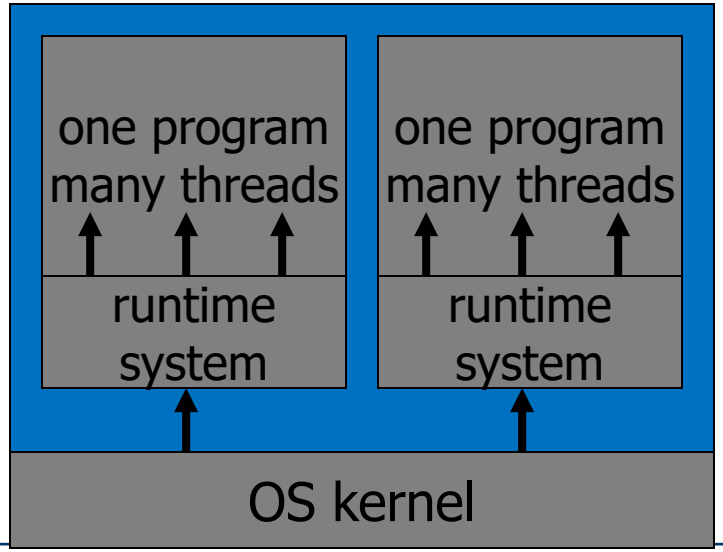
Independent threads in private address spaces.



Threads in a programming language without OS support:

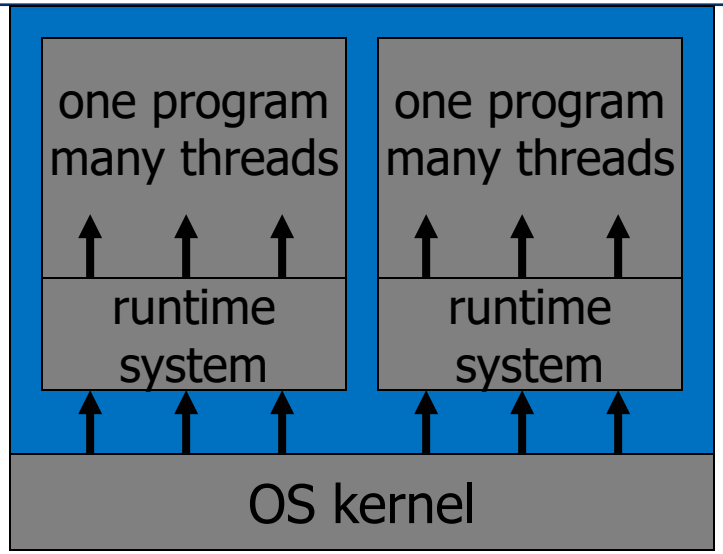
For the OS these threads are not visible.

The whole program is one thread to the OS.



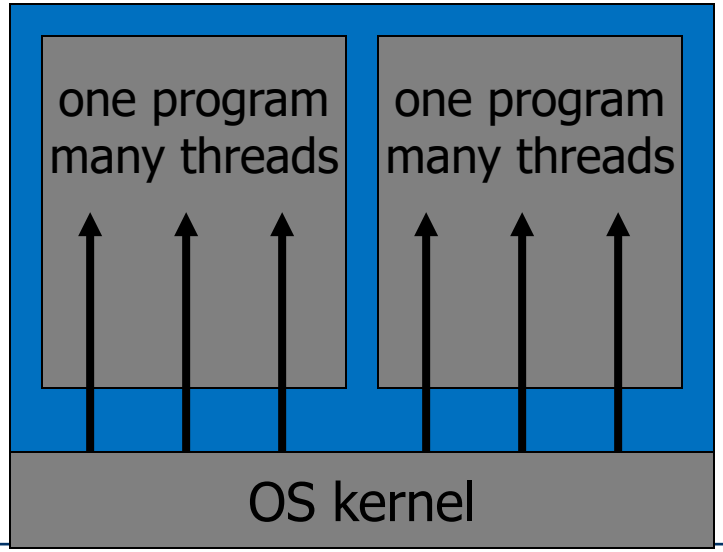
Threads in OS and Programming Languages

Programming language threads are mapped 1:1 to OS-threads.



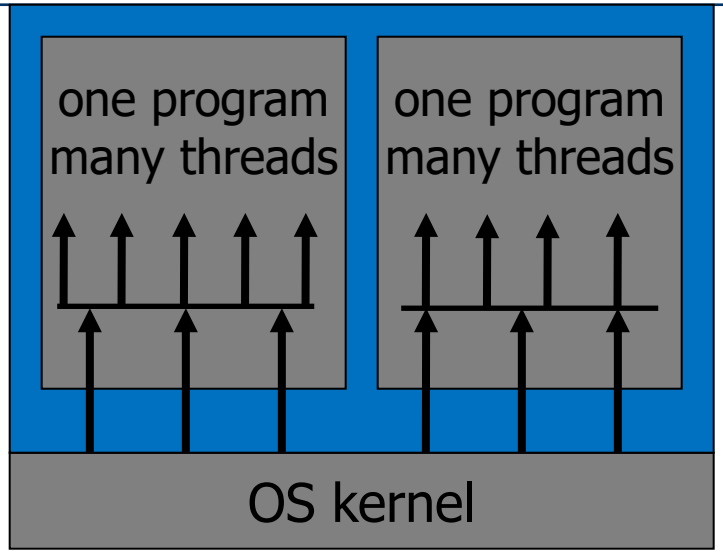
The programming language does not support threads.

The parallel program consists of OS-threads, that share an address space (e.g. Pthreads)



Threads in OS and Programming Languages

User-level threads are mapped m:n to OS-threads.



Further References

- Stallings,W.: Operating Systems 6th ed., Prentice Hall, Chapter 3