Karlsruhe Institute of Technology

# Microkernel Construction
## I.10 – Local IPC (Optimization for Multi-Threaded Applications)

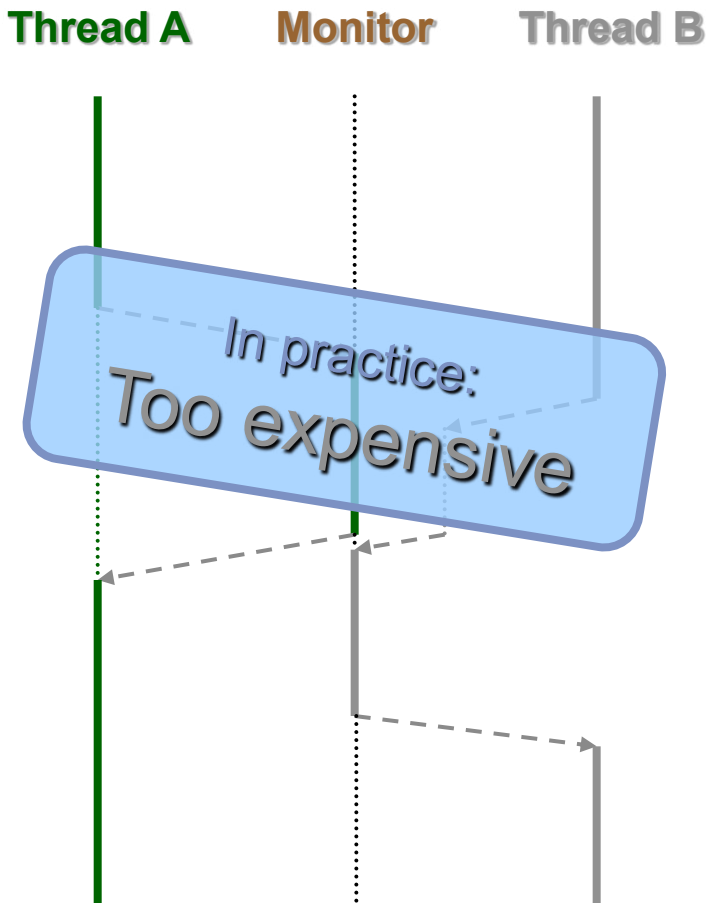
**Lecture Summer Term 2017**
**Wednesday 15:45-17:15 R 131, 50.34 (INFO)**

Jens Kehne | Marius Hillenbrand
Operating Systems Group, Department of Computer Science

# Tutoren für Betriebssysteme gesucht

- Was?
  - Betreuung von Tutorium
  - Korrektur von Abgaben (keine Notenvergabe!)
  - HiWi-Job über je 40 Stunden von Oktober bis Februar
- Warum?
  - Präsentationstechnik üben!
  - Zugang zu Tutorenschulung (SQ, 4 ECTS)
- Interesse?
  - Mathias Gottschlag (R161)
  - mathias.gottschlag@kit.edu

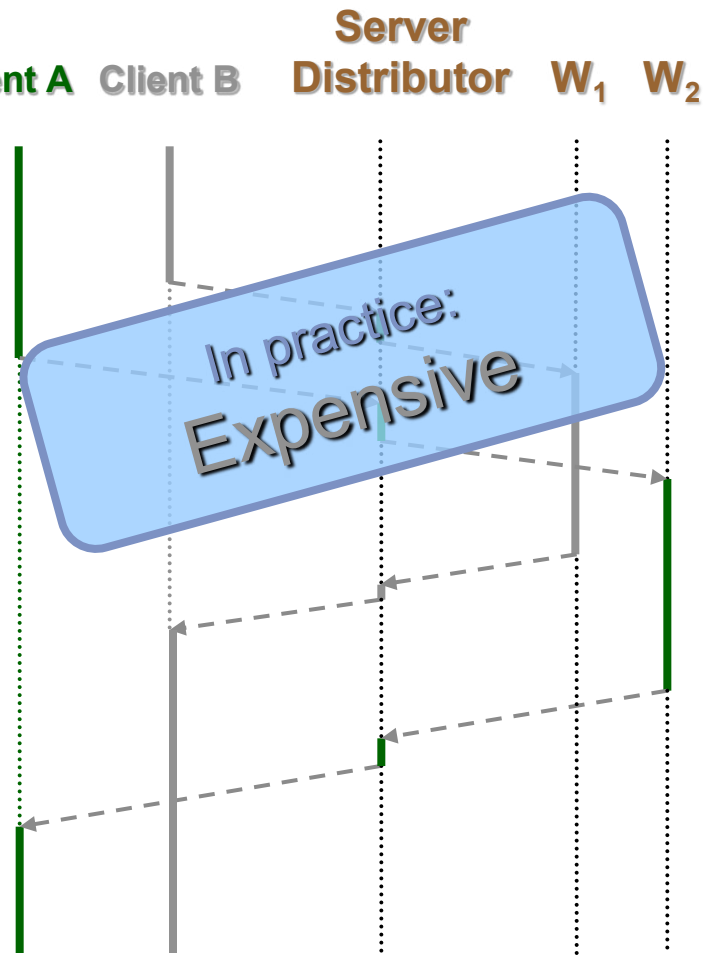
# Synchronization via IPC



# AS-local IPC in Practice

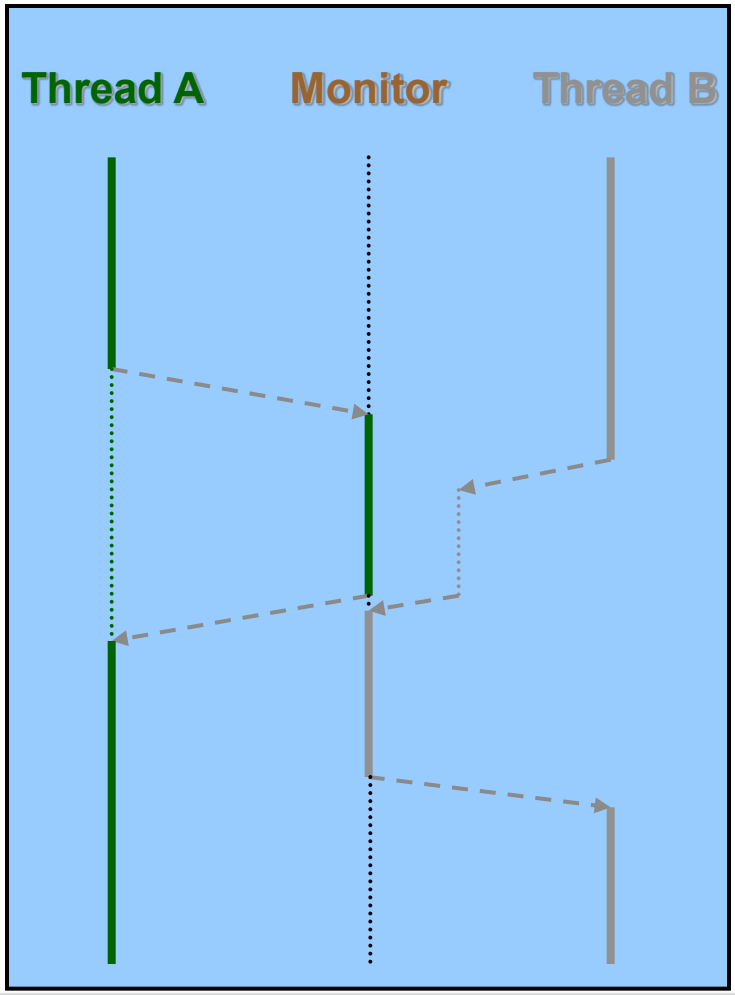
## Synchronization


## Load Distribution

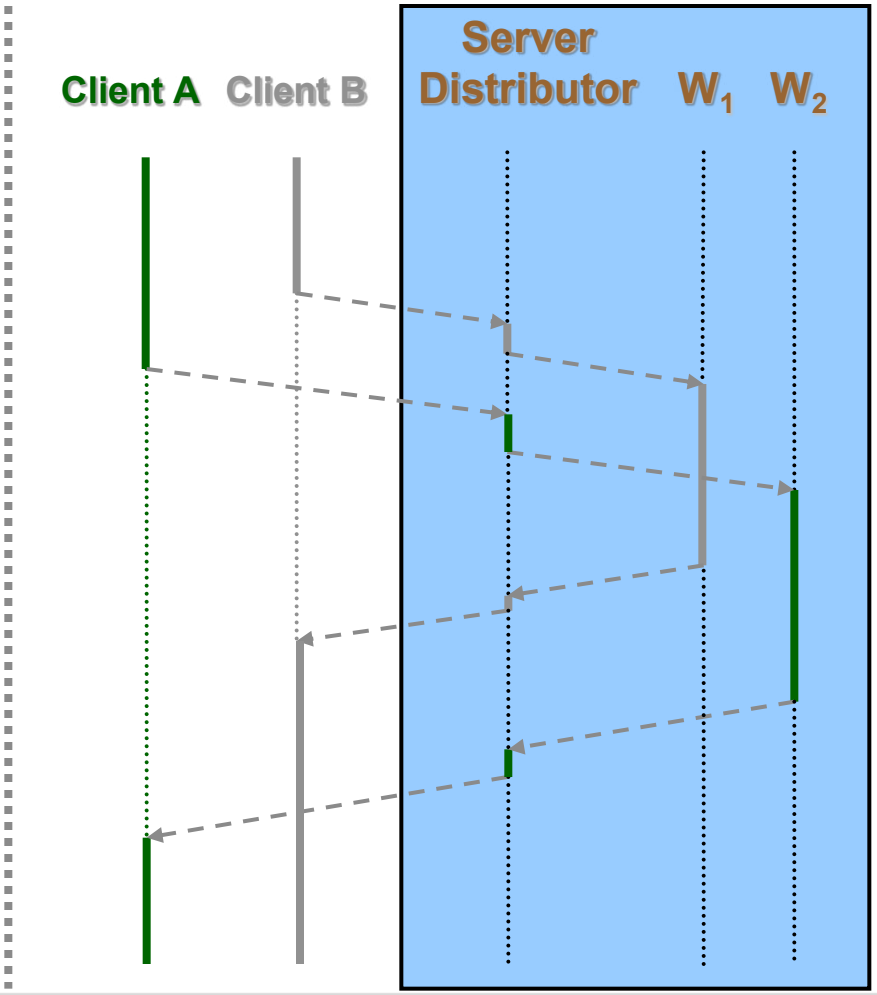


# AS-local IPC in Practice

Synchronization




Load Distribution



# AS-local IPC in Practice

Synchronization

Load Distribution

## Observations

- IPC operations are within same address space
- IPC operations have both blocking send and receive phases

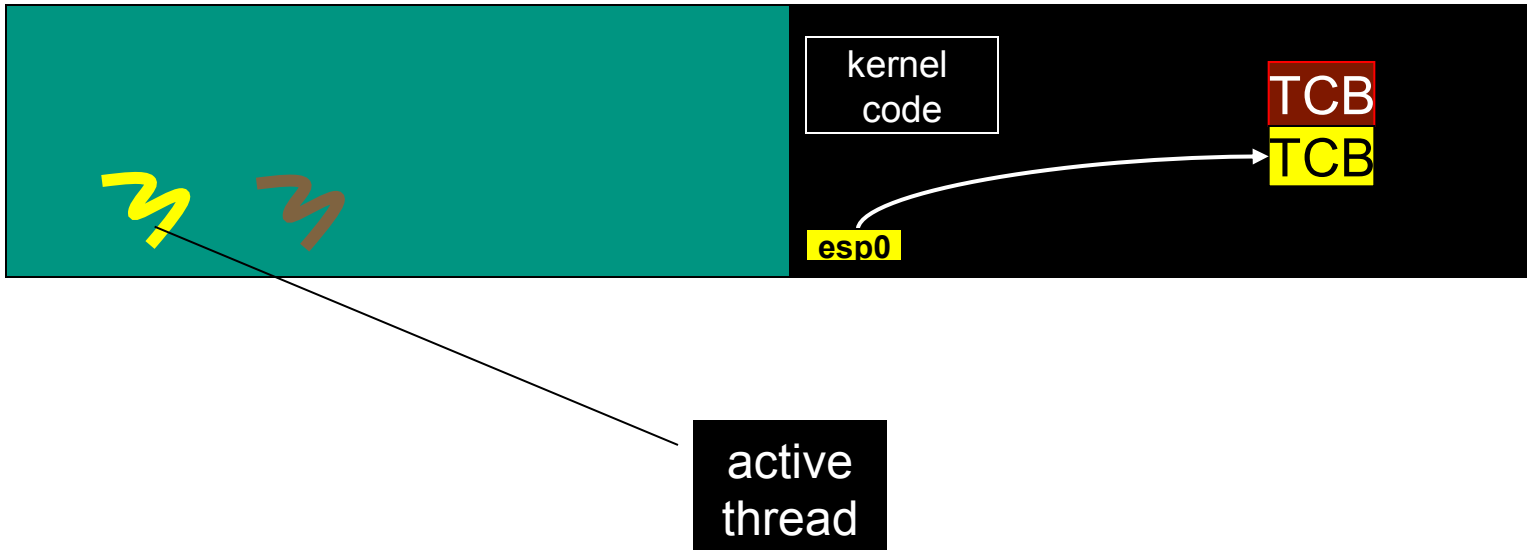
## -- Introduce special Local IPC --

- Restrictions
  - Same address space
  - Must have both blocking send and receive phase
- Can execute entirely at user-level
- LIPC executes in **~20 cycles!**

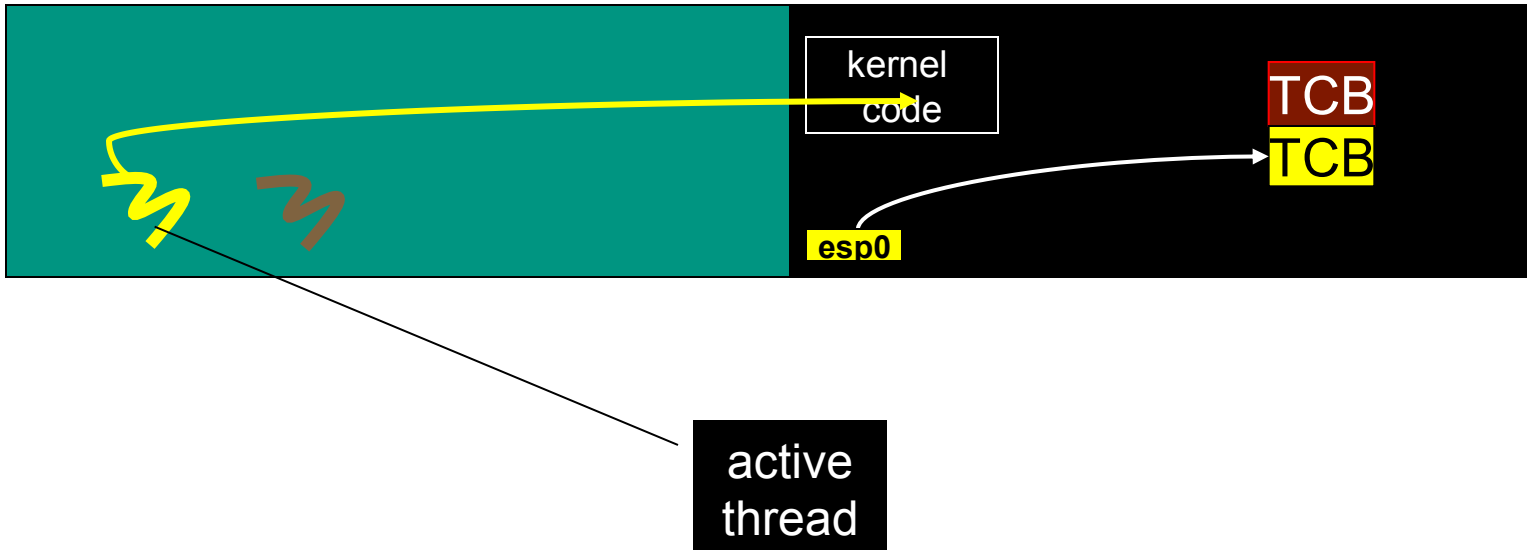
# User-Level Threads?

- Would achieve required speed
- But …
  - Not known to the kernel
  - Execute in a single thread's context
  - Making them kernel-schedulable does not pay (SDI: scheduler activations)
  - Two concepts – inelegant, contradicts minimality
- We want …
  - Kernel-level threads
  - The speed of user-level threads

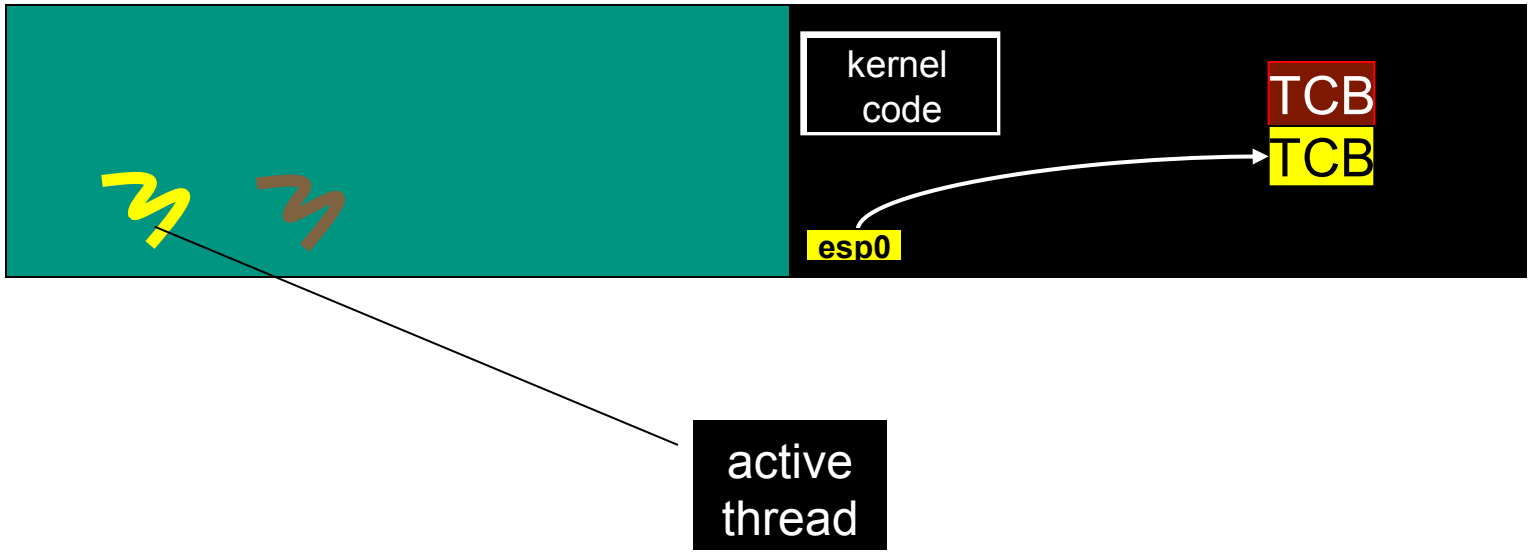
# Strict Switching



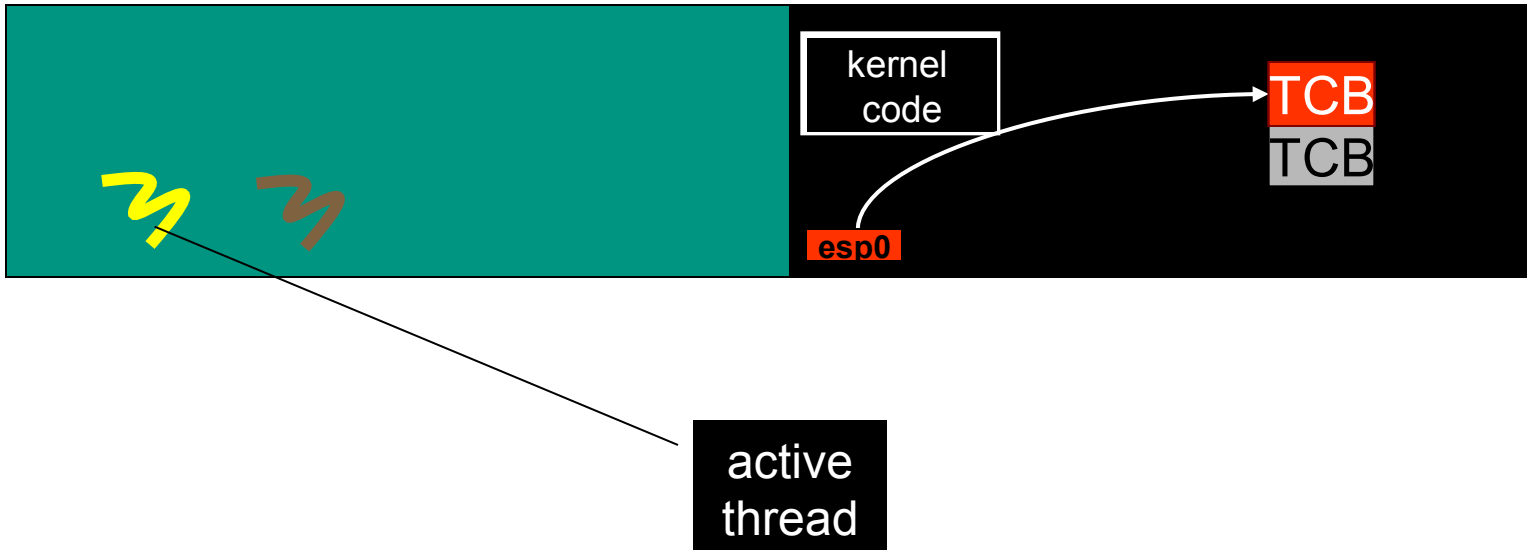
# Strict Switching



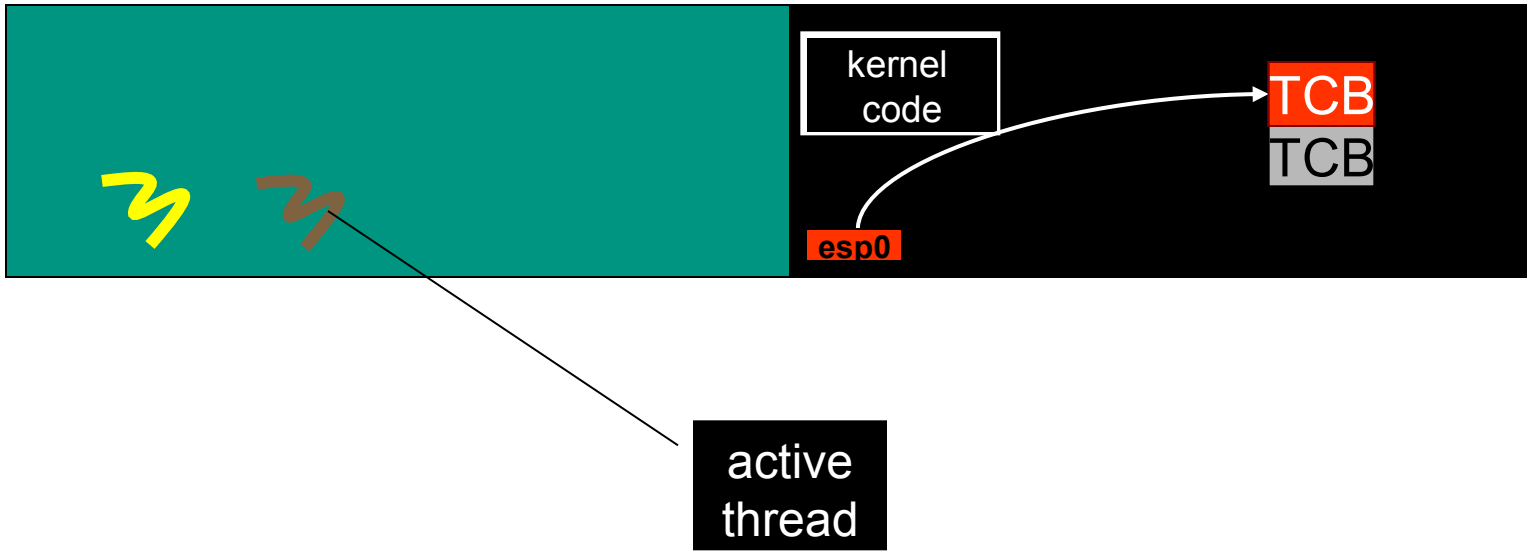
# Strict Switching



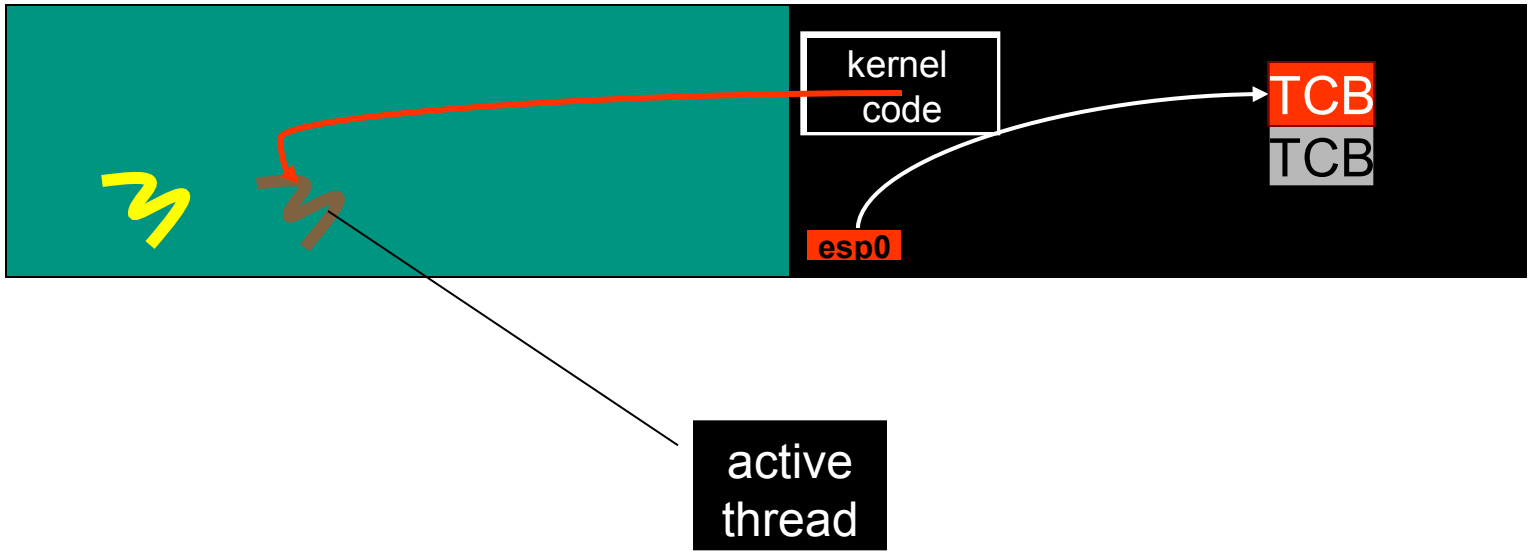
# Strict Switching



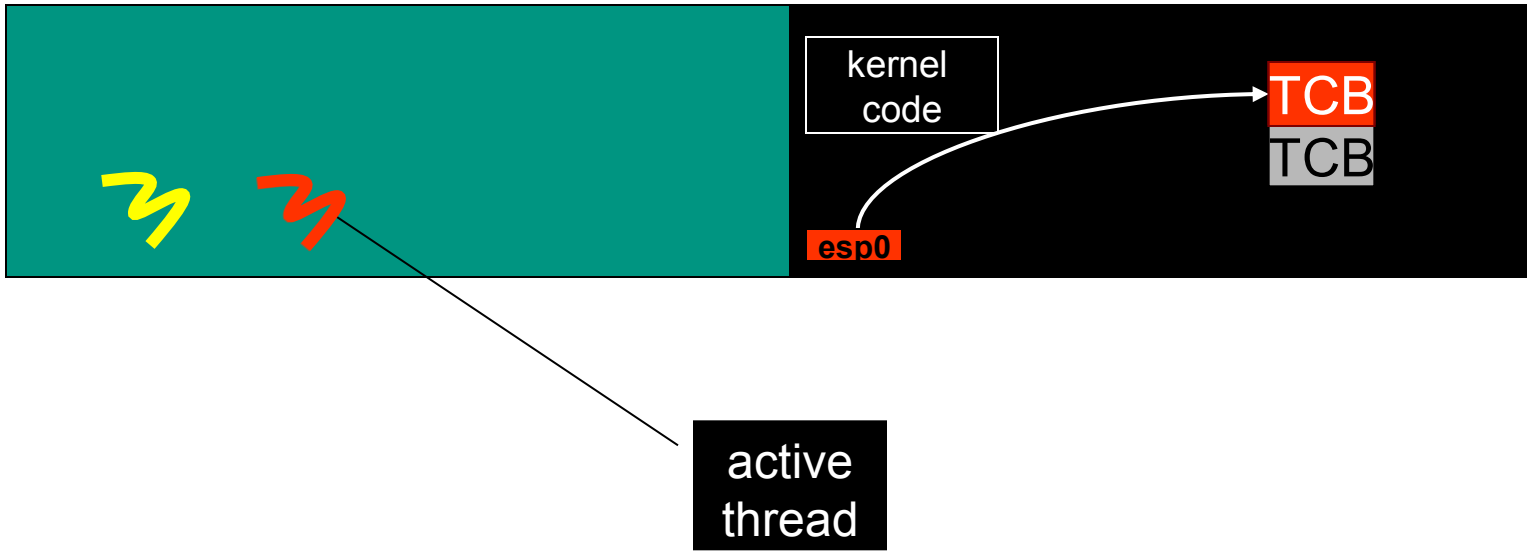
# Strict Switching



# Strict Switching

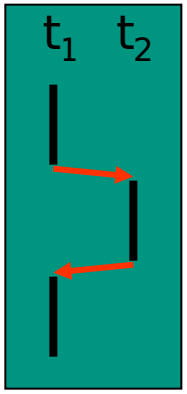


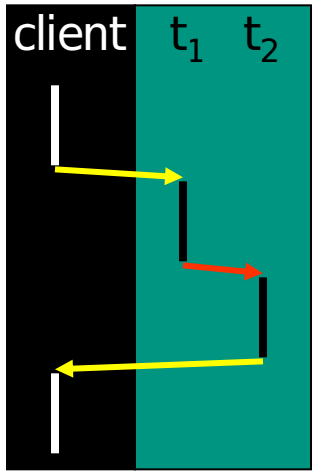
# Strict Switching



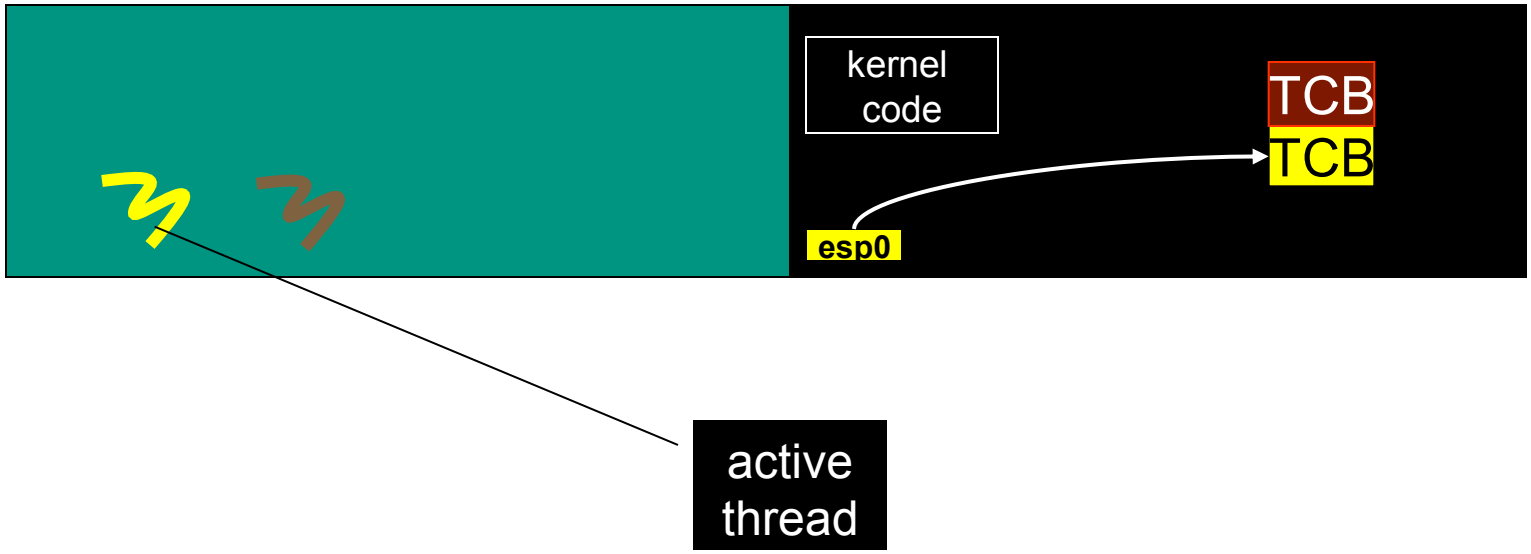
# Basic Idea

- Assume IPC $t_1 \rightarrow t_2$, same address space
- Let $t_1$ execute $t_2$-code
- Postpone real switch until the kernel is activated
- Pays if multiple lazy switches occur before first kernel activation, e.g.:
  - $t_1 \rightarrow t_2$, work, $t_2 \rightarrow t_1$
    - Costs 0 kernel-level IPC
  - client $\rightarrow t_1 \rightarrow t_2 \rightarrow$ client
    - Costs 2 kernel-level IPCs

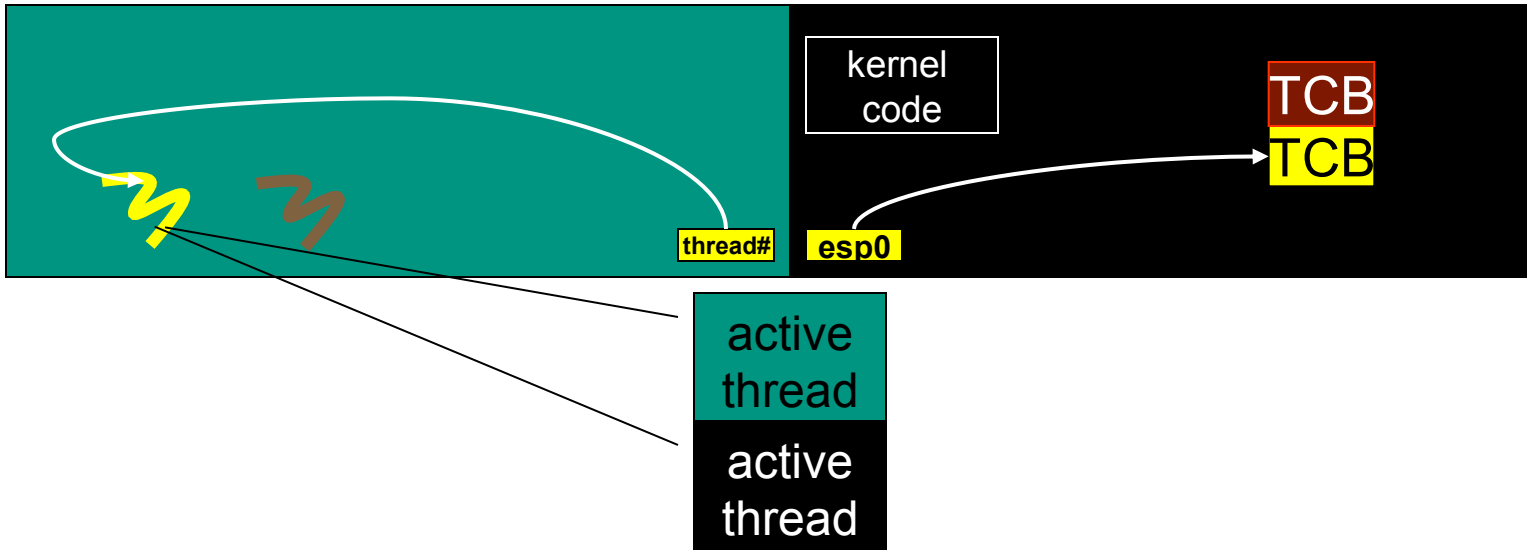


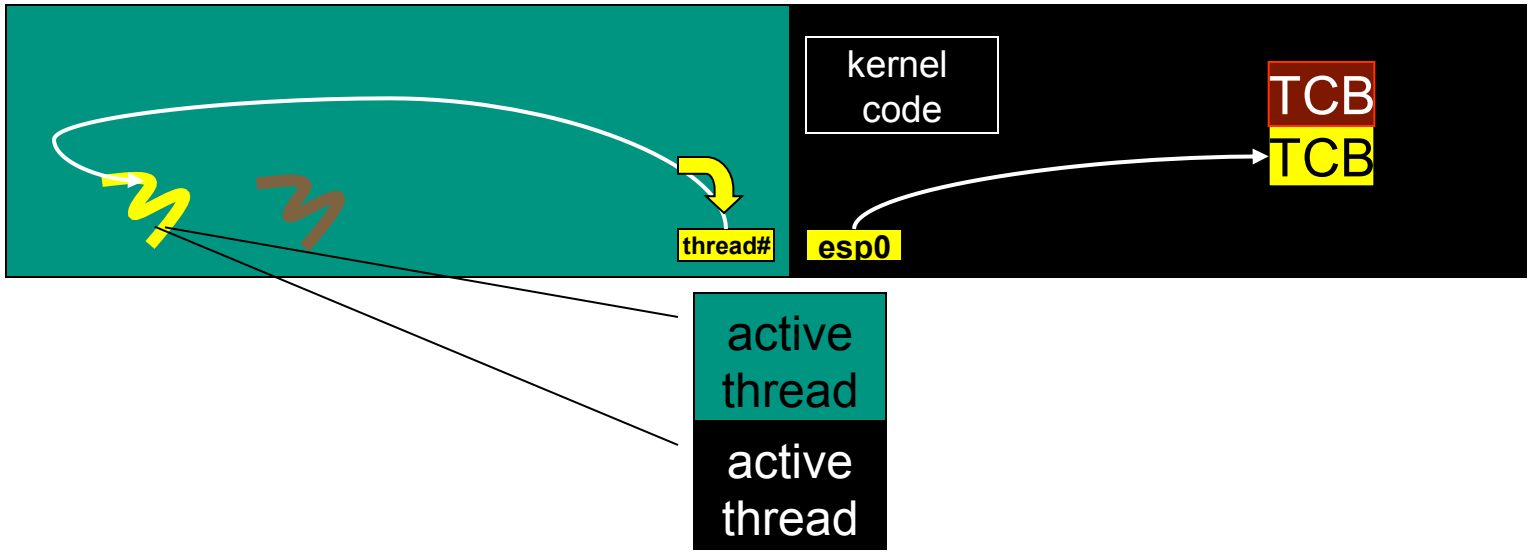
# Lazy Switching



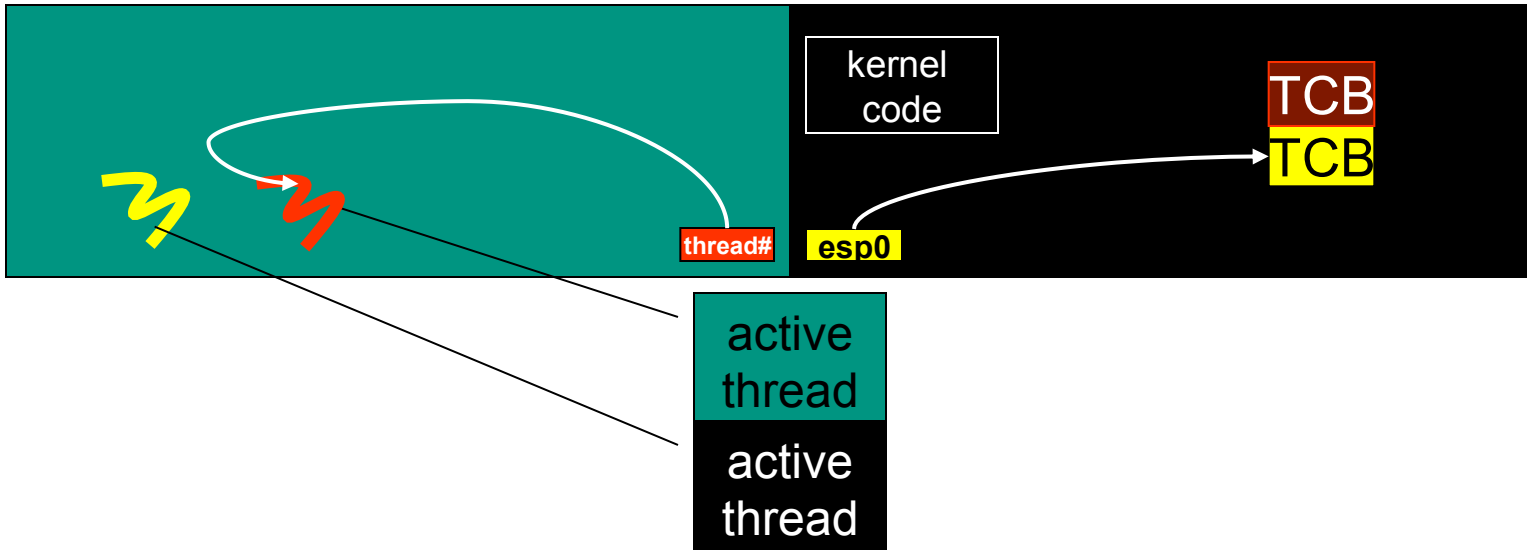
# Lazy Switching



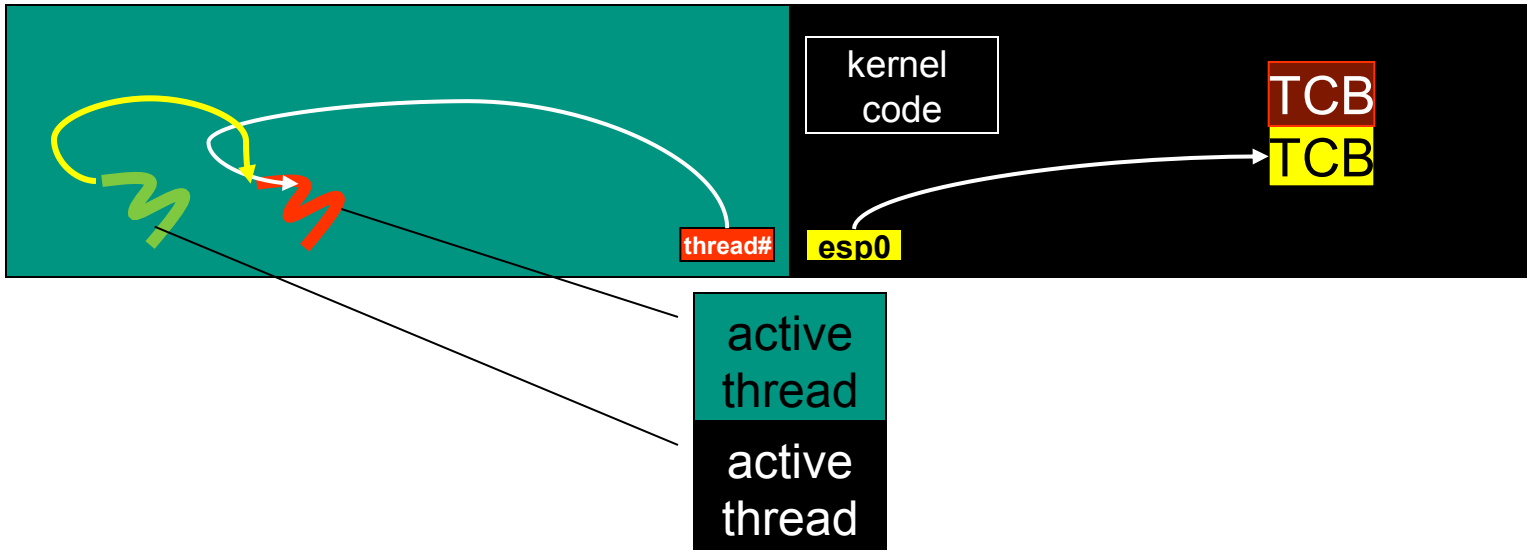
# Lazy Switching



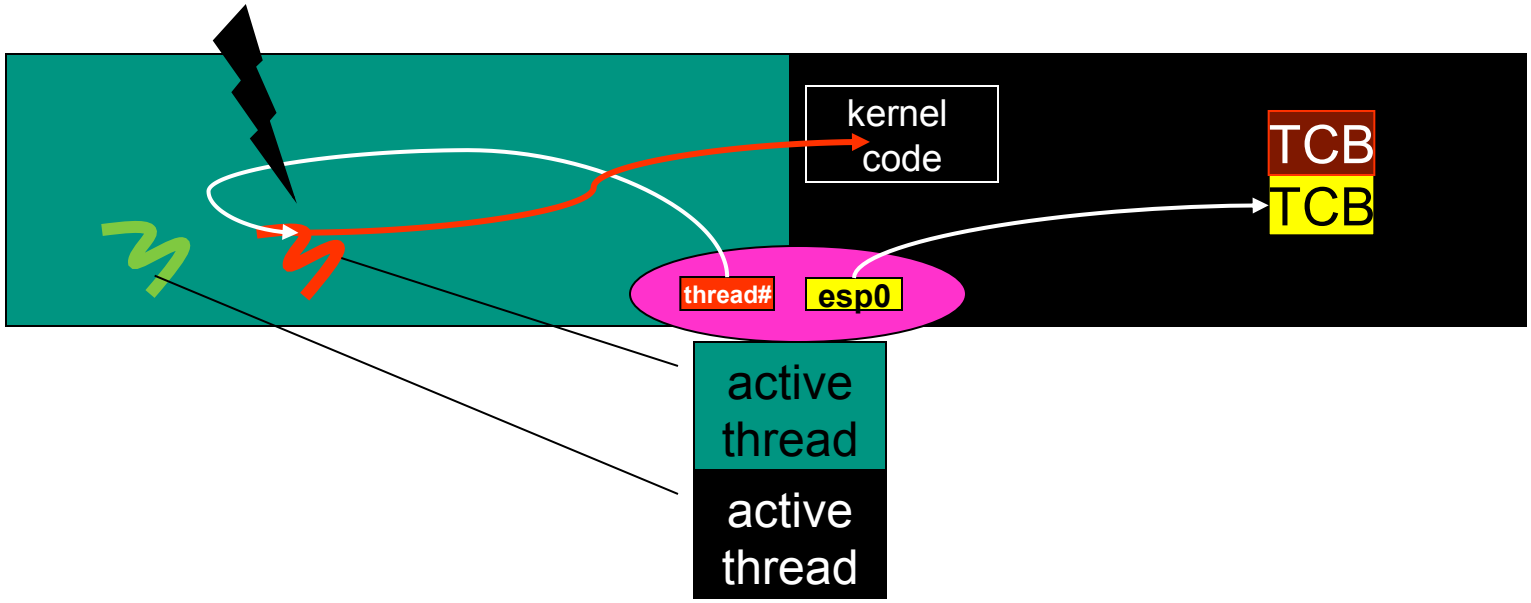
# Lazy Switching



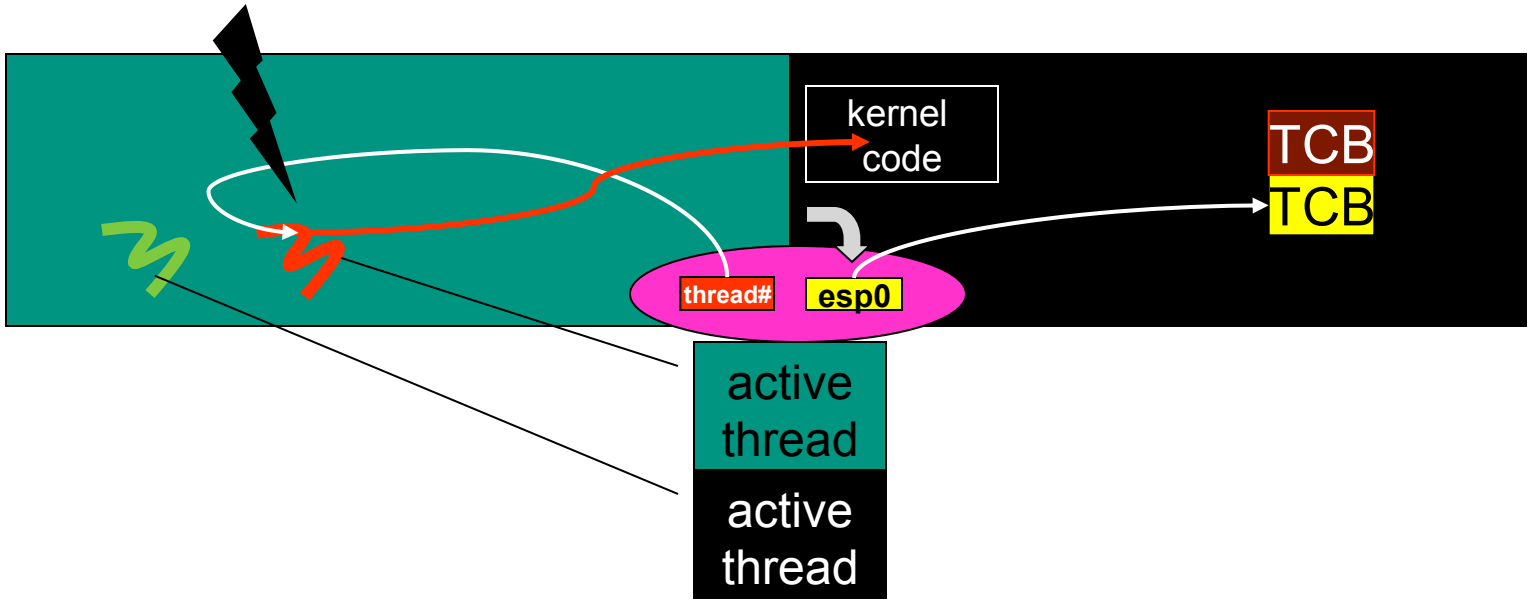
# Lazy Switching



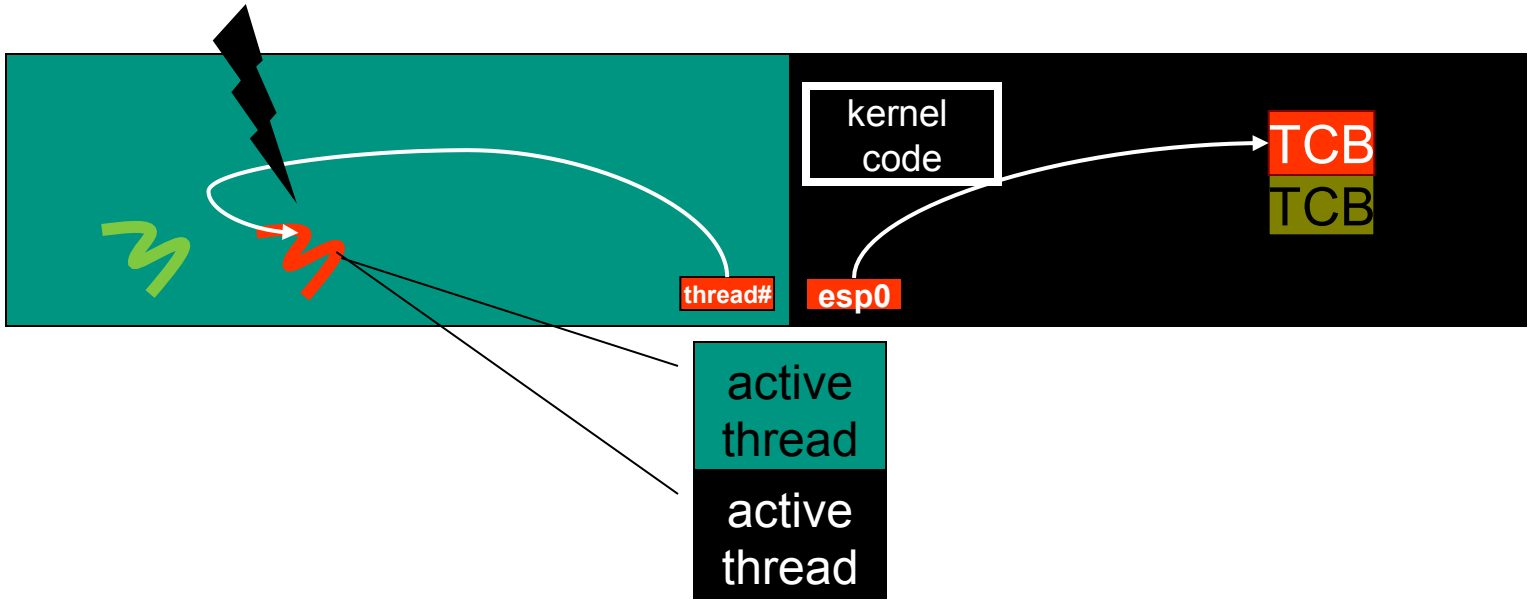
# Lazy Switching



# Lazy Switching



# Lazy Switching



# IPC Revisited

A → B: SendAndWaitForReply in user-mode

```
    call IPC function, i.e. push A's instruction pointer ;
    save A's stack pointer ;

    if B is valid thread ID and thread B waits for thread A
    then

        set A's status to "wait for B" ;
        set B's status to "run" ;
        load B's stack pointer ;
        current thread := B ;

        return, i.e. pop B's instruction pointer
    else
        more complicated IPC handling
    fi .
```

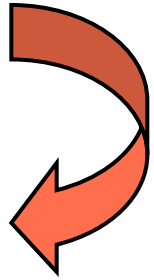
Atomicity?

Kernel Data?

# Atomicity

A → B: SendAndWaitForReply in user-mode

```
call IPC function, i.e. push A's instruction pointer ;
save A's stack pointer ;
– restart point –
if B is valid thread ID and thread B waits for thread A
then
    – forward point –
    set A's status to "wait for B" ;
    set B's status to "run" ;
    load B's stack pointer ;
    current thread := B ;
    – completion point –
    return, i.e. pop B's instruction pointer
else
    more complicated IPC handling
fi .
```

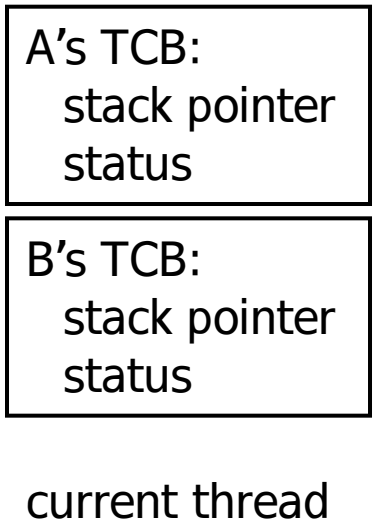


# Atomicity (2)

Interruption between forward point and completion point:

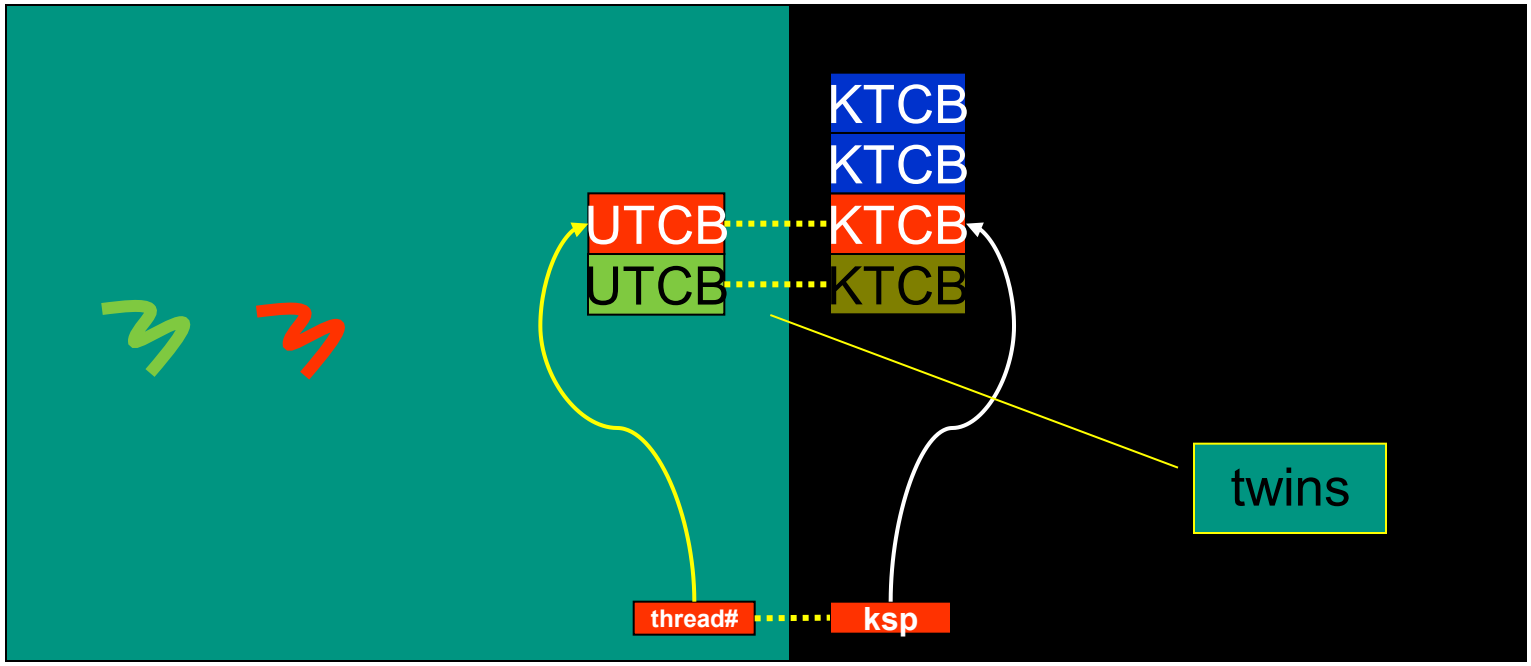
```
if is page fault
then
    kill thread A
else
    set A's status to "wait for B" ;
    set B's status to "run" ;
    load B's stack pointer ;
    current thread := B ;
    set interrupted instruction pointer to completion point
fi .
```

# Kernel Data

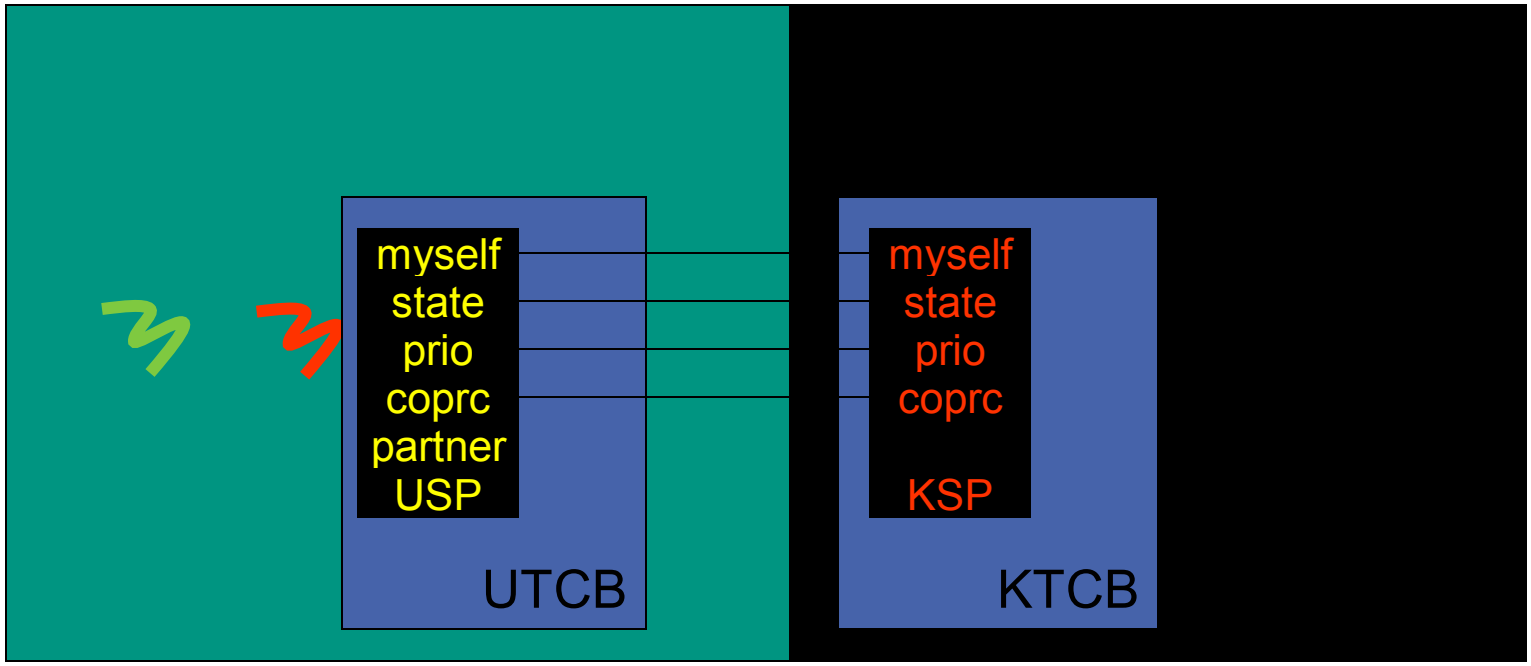



- Stack pointer
  - Can be user accessible
- Status
  - User-level effects
    - Local to A's task can be ignored
    - Indirect effects on other tasks can be ignored
  - System-level effects
    - Must be avoided
    - Validate values on change
    - Maintain twin variable in kernel

# UTCB – KTCB



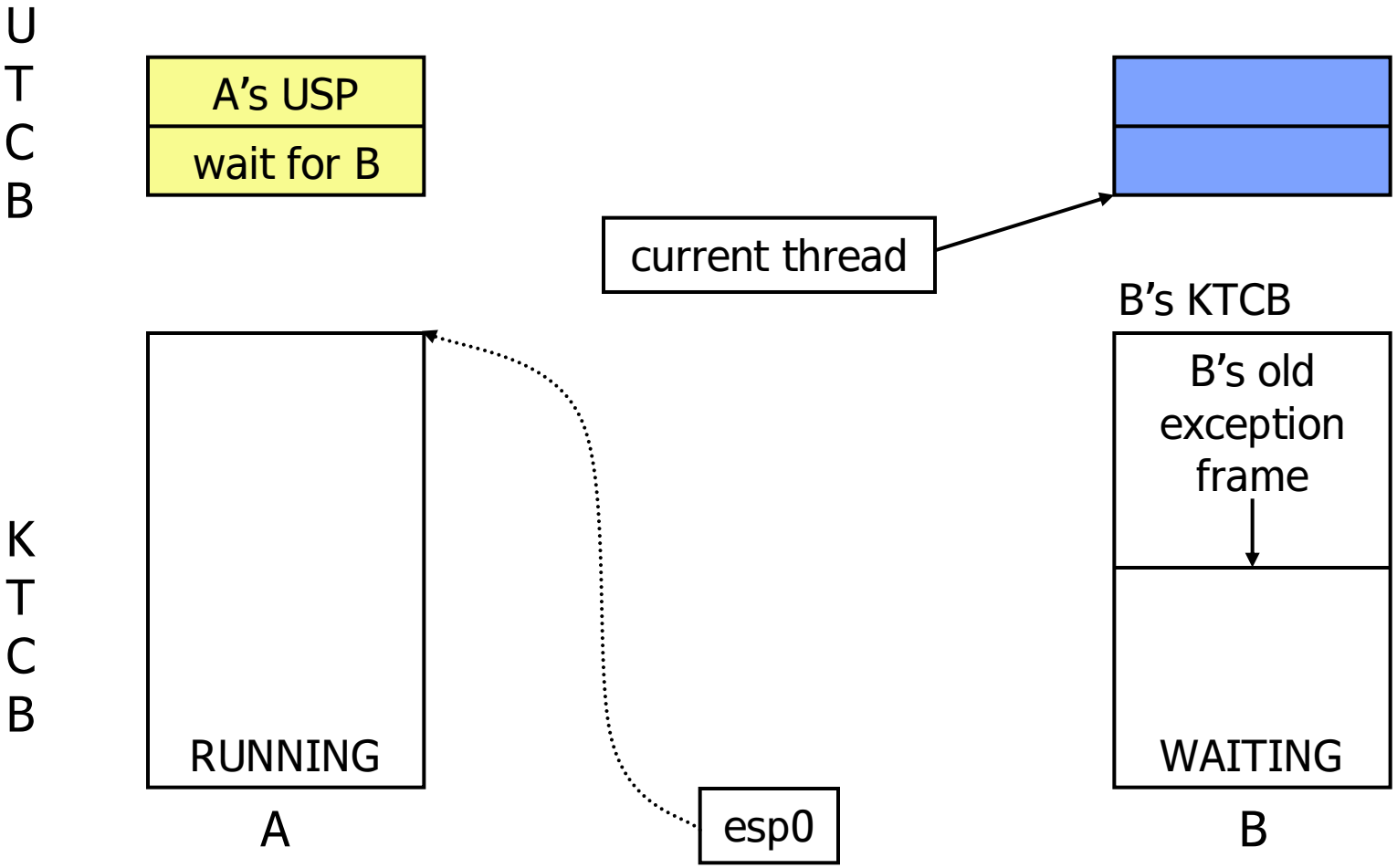
# UTCB – KTCB


myself
state
prio
coprc
partner
USP
UTCB
myself
state
prio
coprc
KSP
KTCB

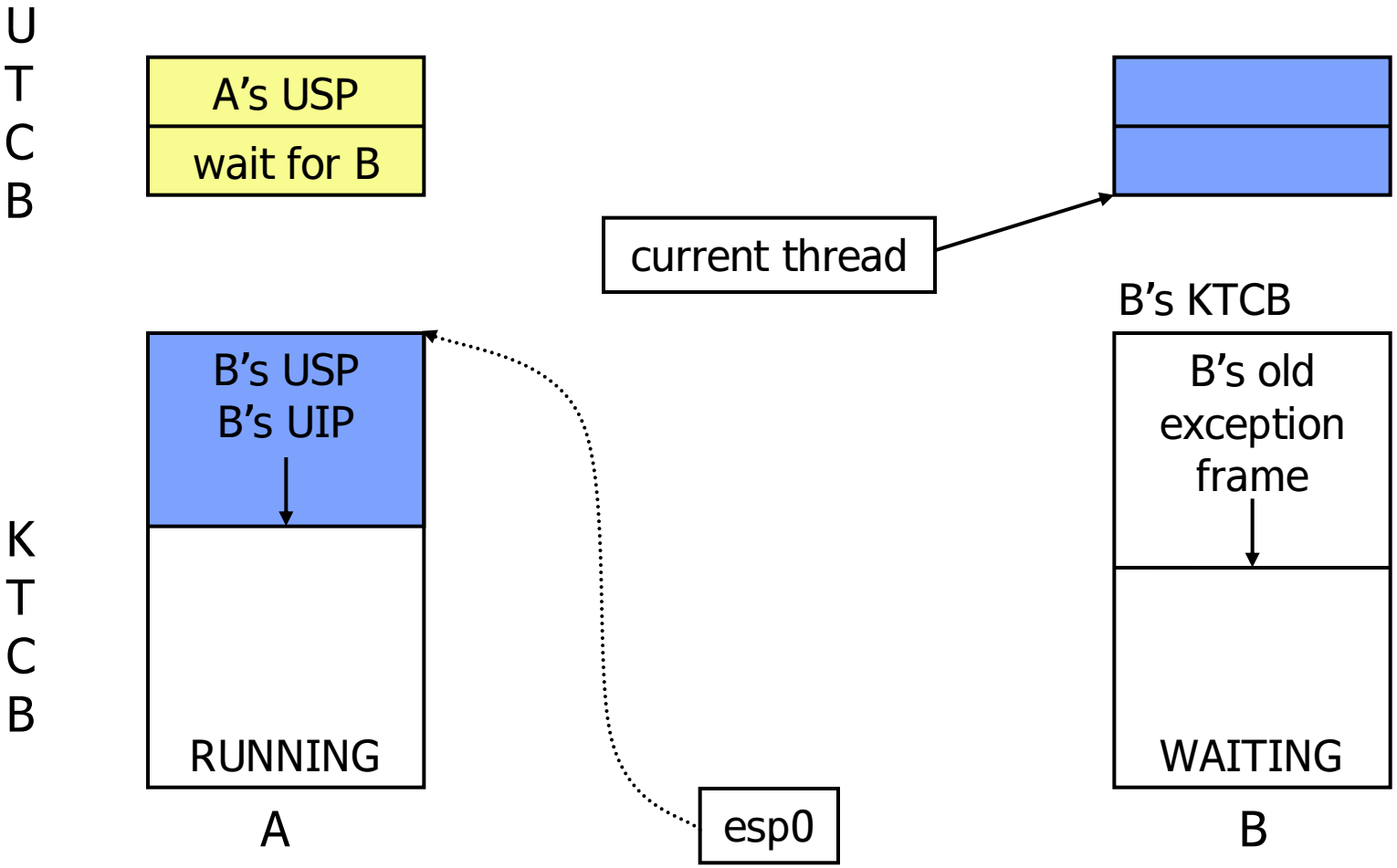
# Current_thread Inconsistency

```
if   CurrentKTCB.utcb != CurrentUTCB
then
    /* Inconsistency found – check validity of user-level thread switch. */
    NewKTCB := getKTCB(CurrentUTCB.myself) ;
    if   NewKTCB.myself = CurrentUTCB.myself  and
         NewKTCB.space = CurrentKTCB.space  and
         NewKTCB.utcb = CurrentUTCB
    then
        /* Valid user-level switch to valid thread in same address space. */
        update kernel state ;
        CurrentKTCB := NewKTCB ;
    else
        kill thread(CurrentKTCB)
    fi
fi .
```

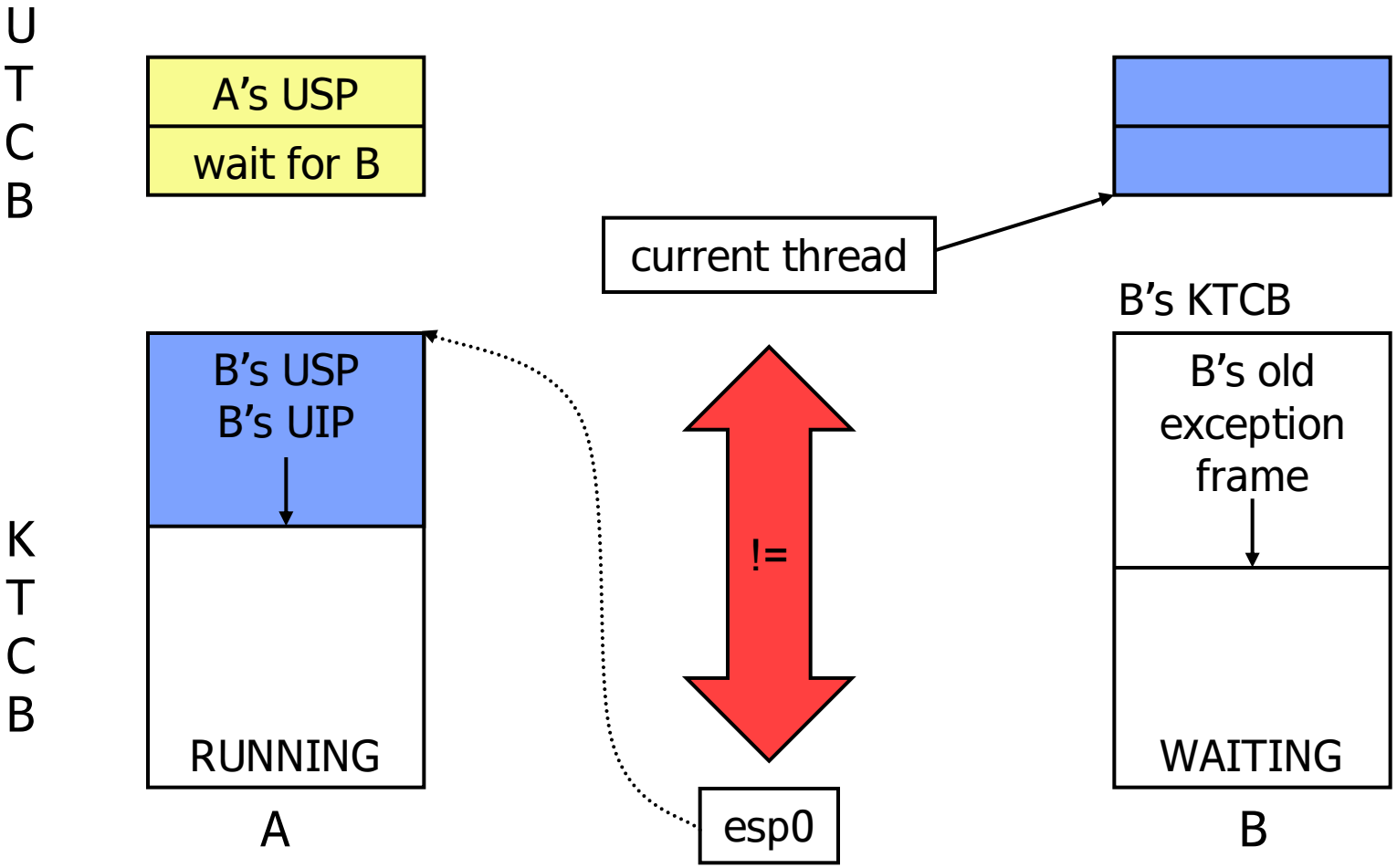
# Kernel State Fixup – A → B



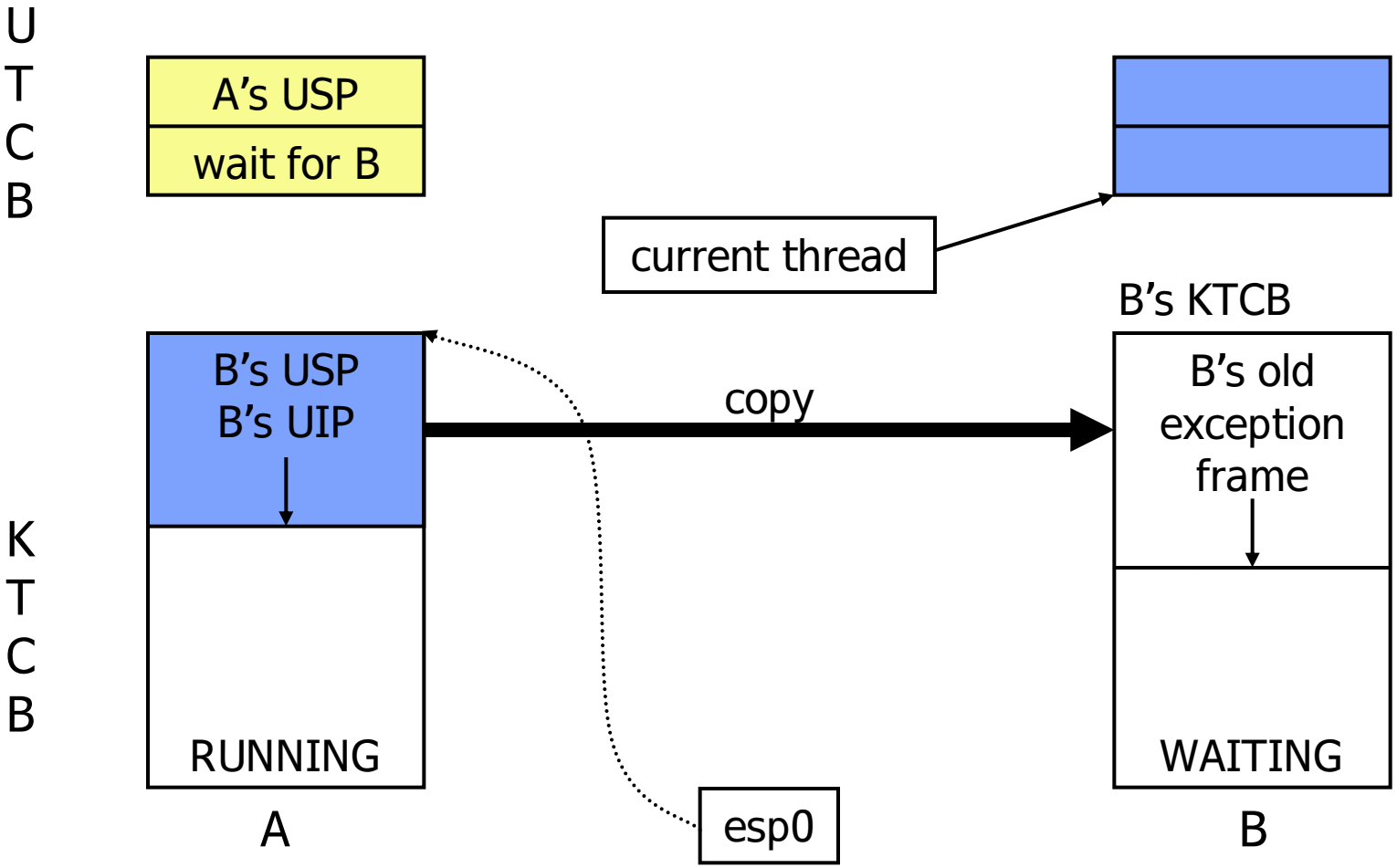
# Kernel State Fixup – A → B



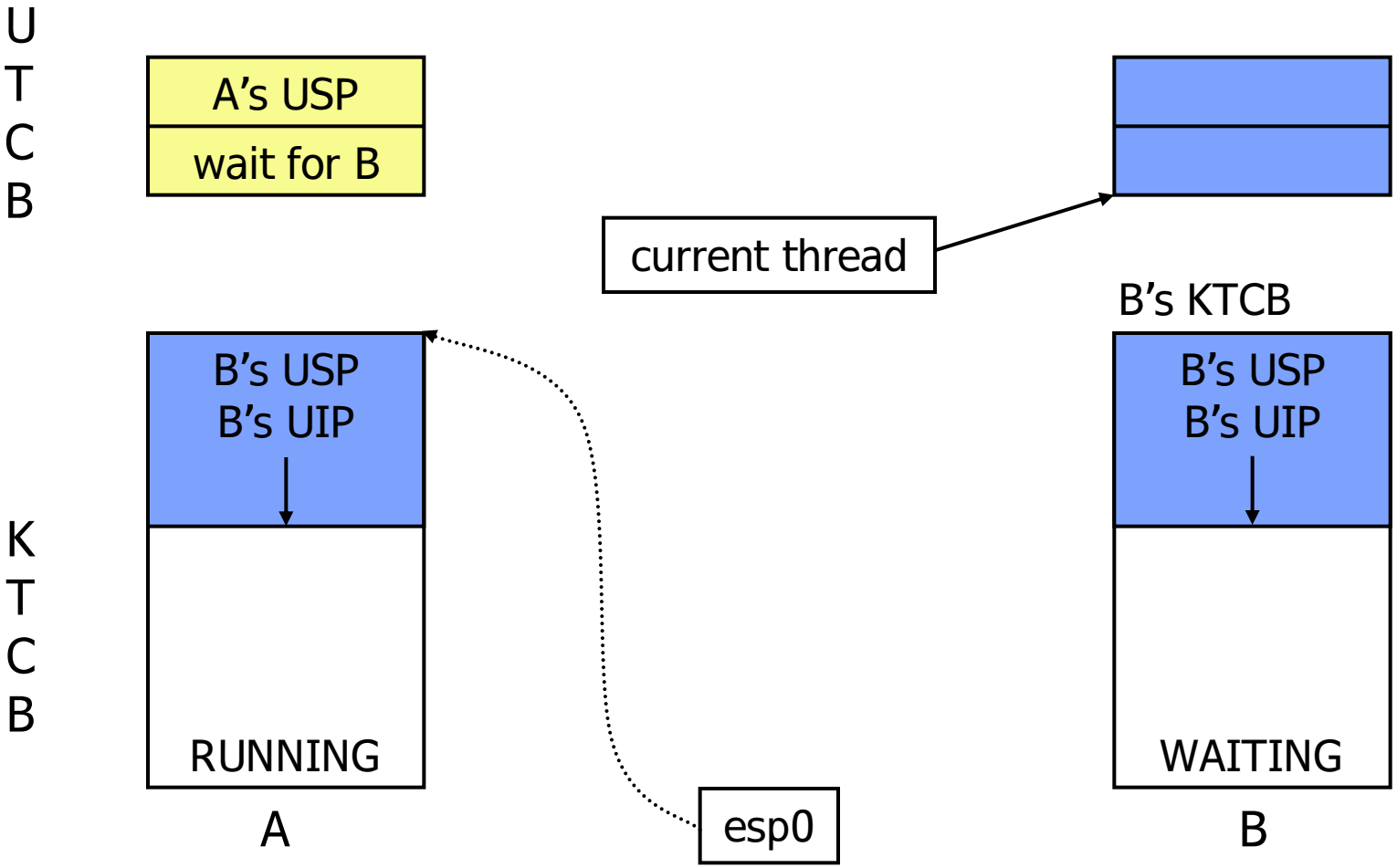
# Kernel State Fixup – A → B



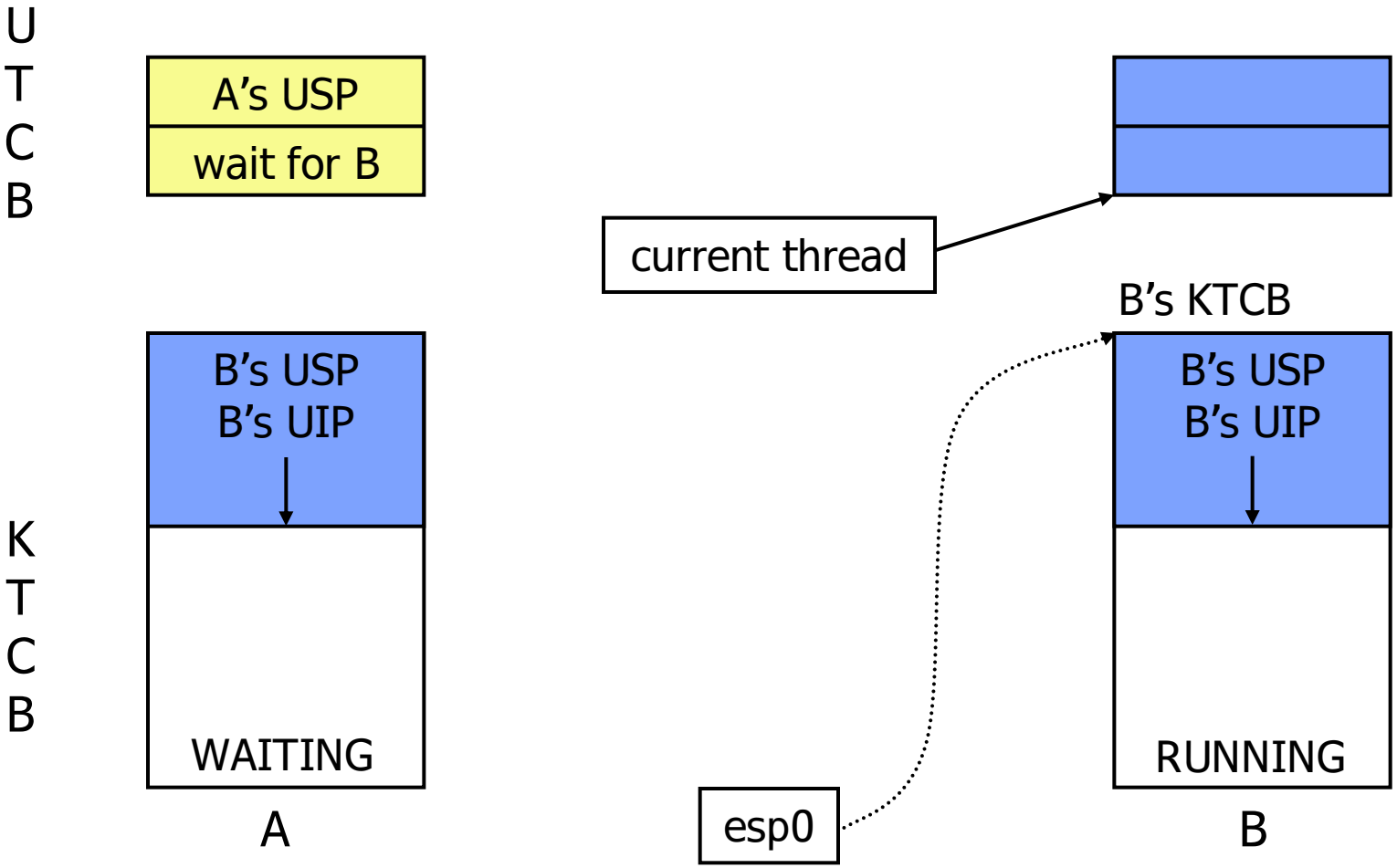
# Kernel State Fixup – A → B



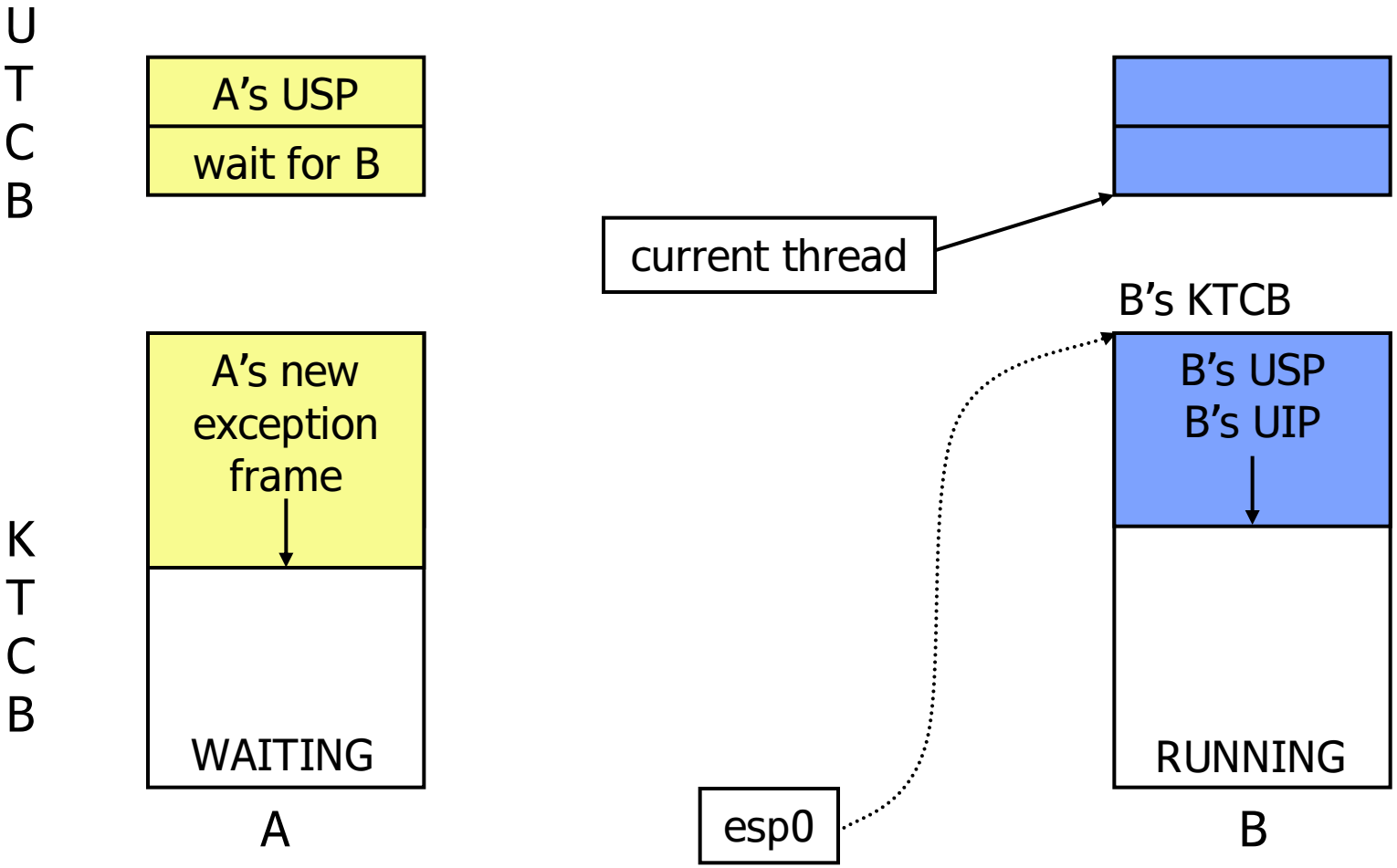
# Kernel State Fixup – A → B



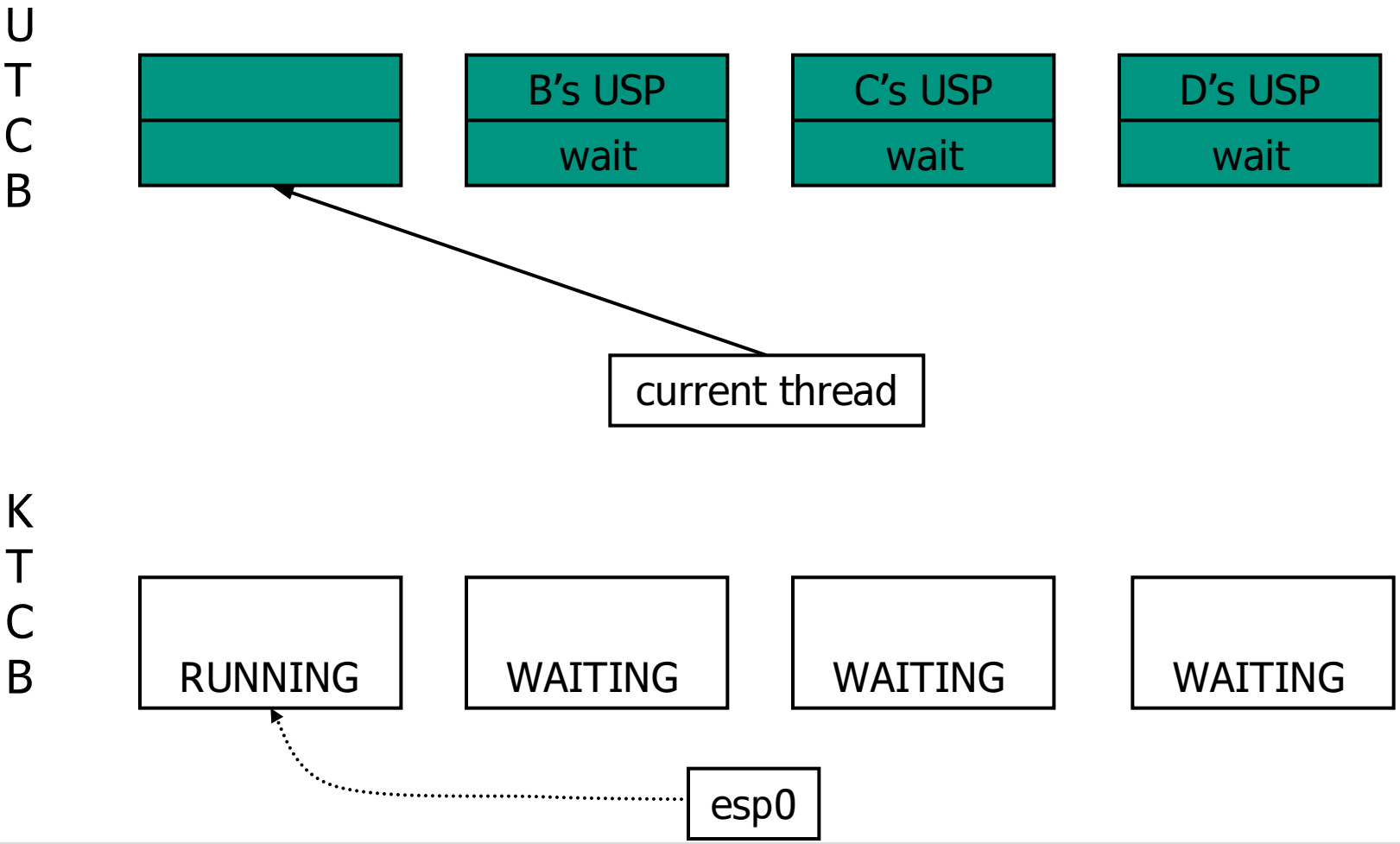
# Kernel State Fixup – A → B



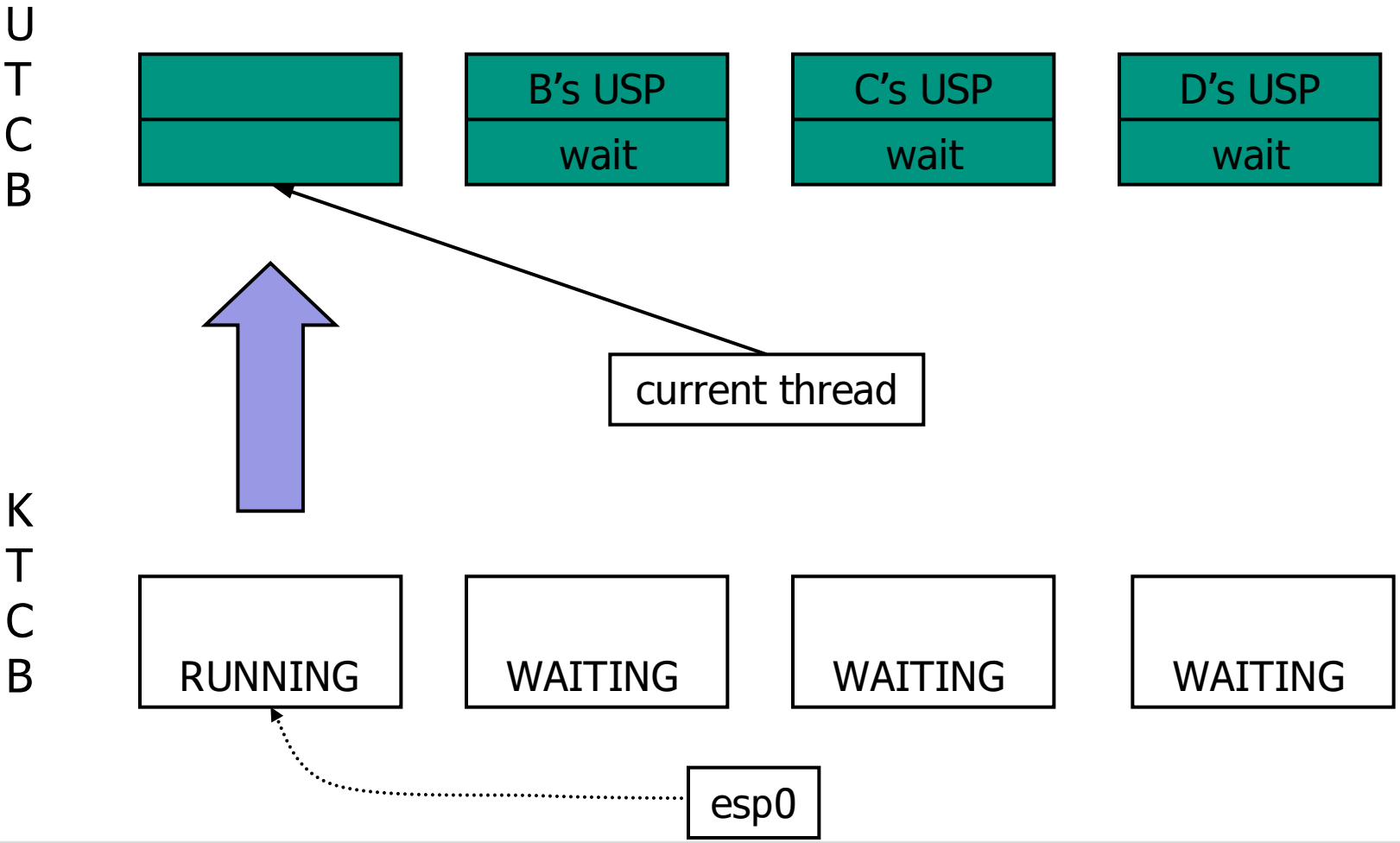
# Kernel State Fixup – A → B



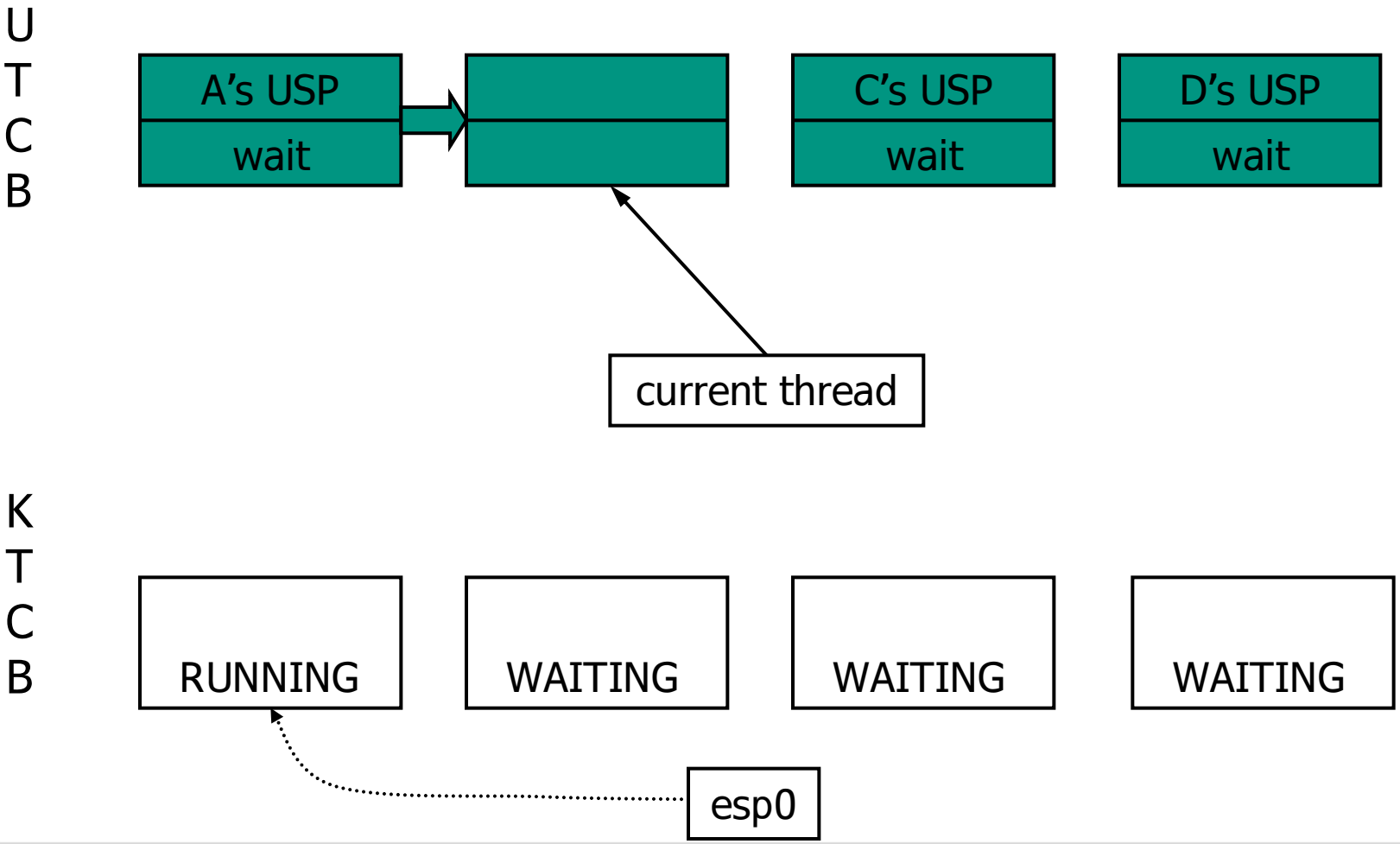
# LIPC Chains


UTCB
B's USP
wait
C's USP
wait
D's USP
wait
current thread
KTCB
RUNNING
WAITING
WAITING
WAITING
esp0

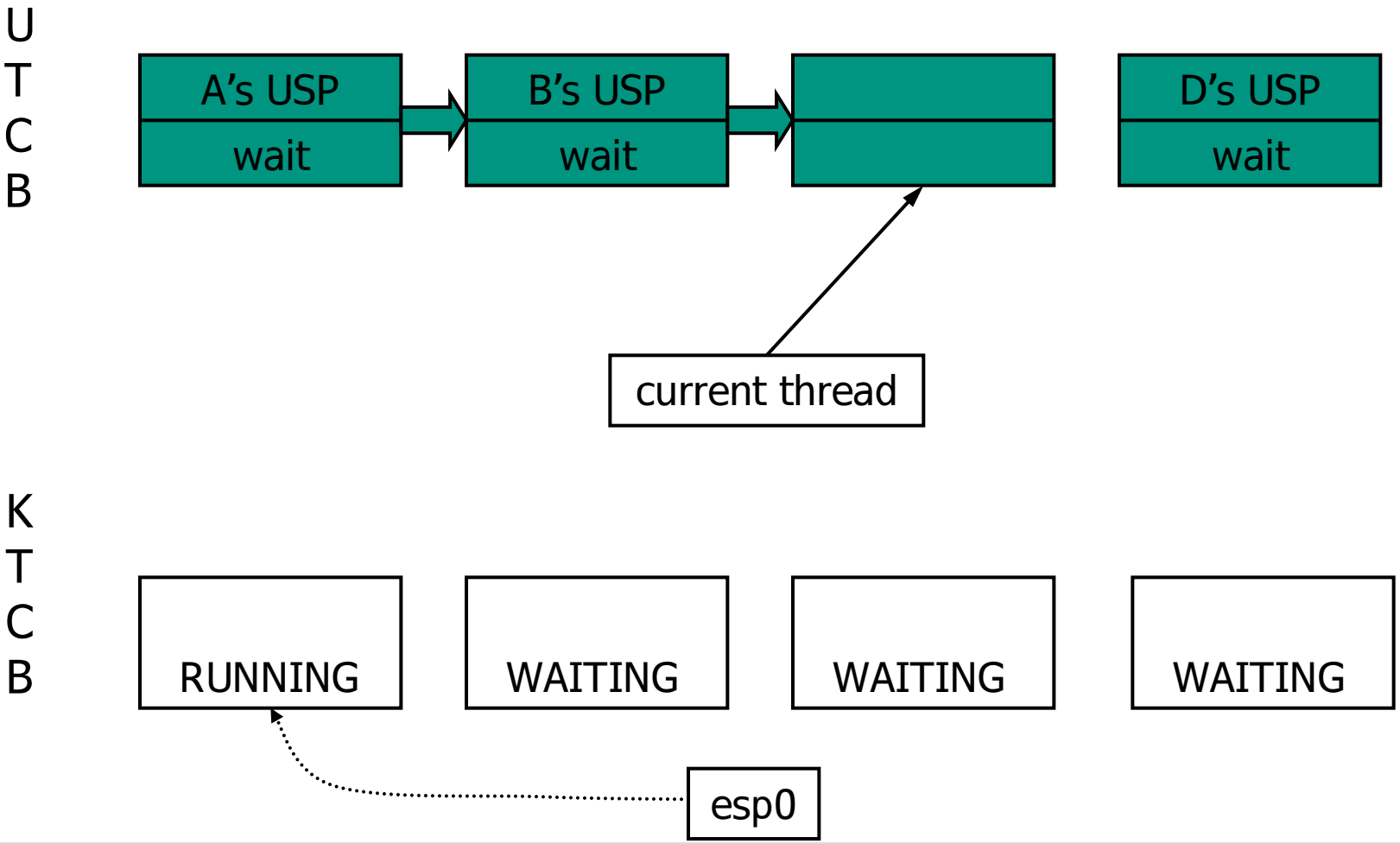
# LIPC Chains


UTCB
B's USP
wait
C's USP
wait
D's USP
wait
current thread
KTCB
RUNNING
WAITING
WAITING
WAITING
esp0

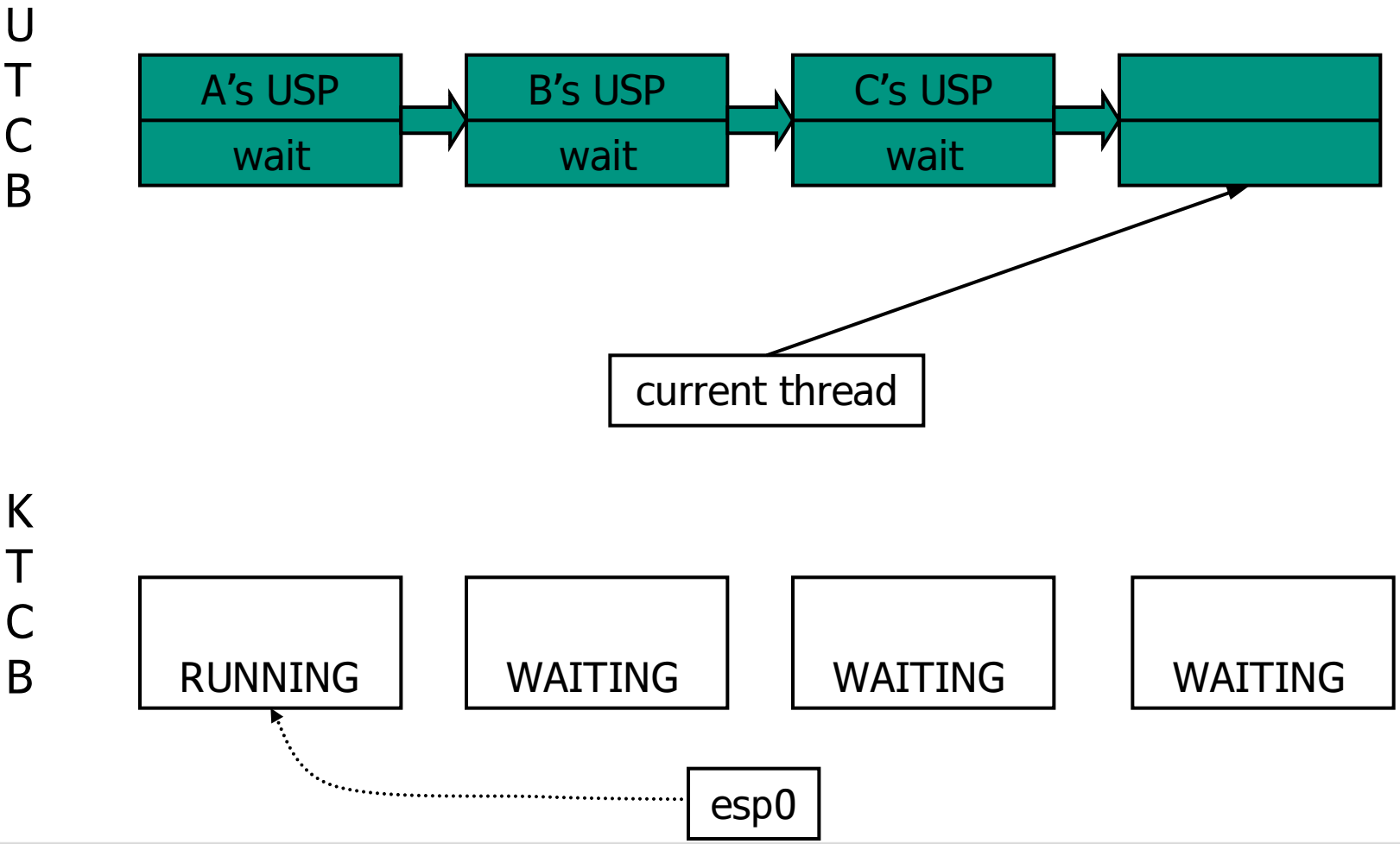
# LIPC Chains


UTCB
A's USP
wait
C's USP
wait
D's USP
wait
current thread
KTCB
RUNNING
WAITING
WAITING
WAITING
esp0

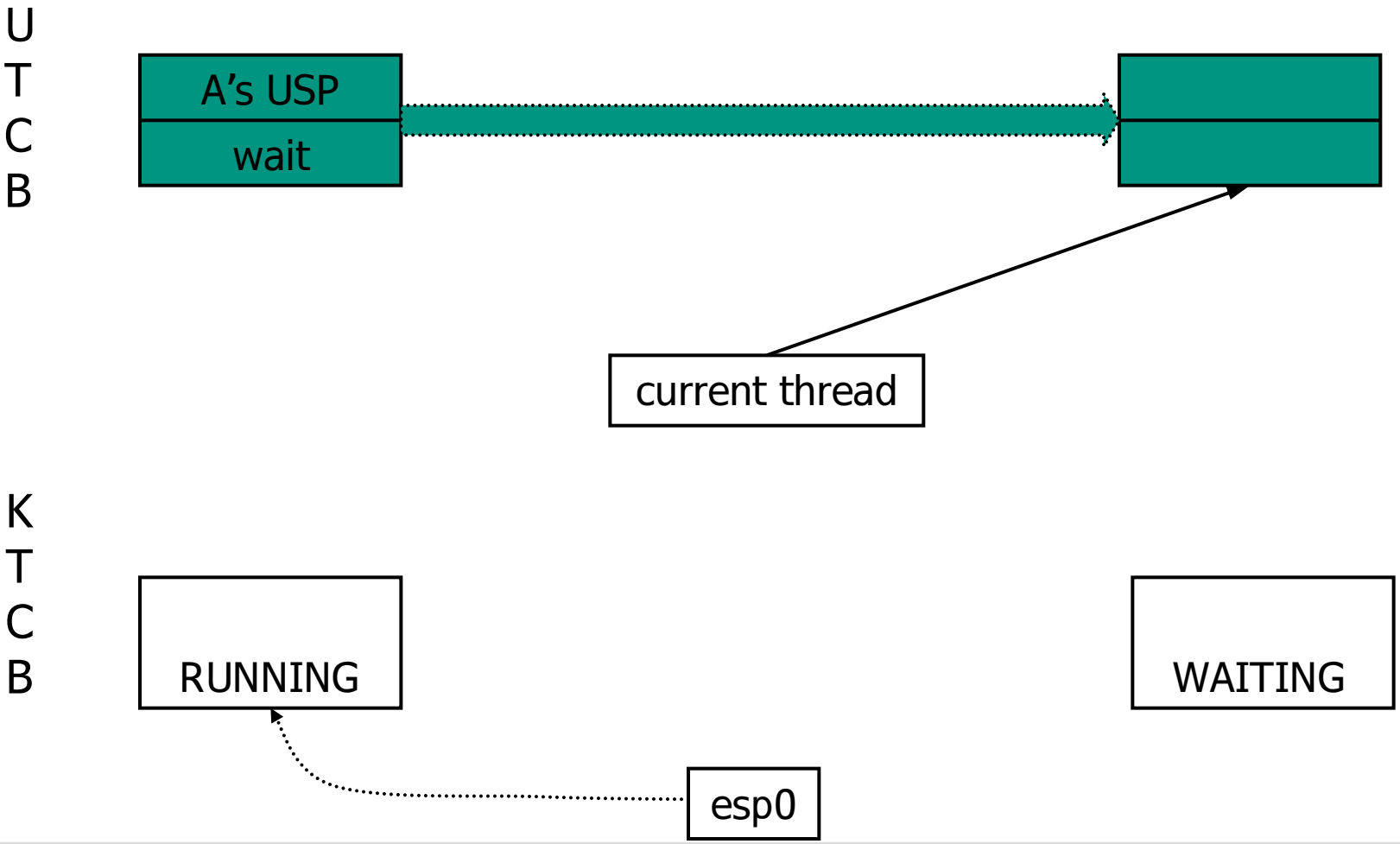
# LIPC Chains


UTCB
A's USP
wait
B's USP
wait
D's USP
wait
current thread
KTCB
RUNNING
WAITING
WAITING
WAITING
esp0

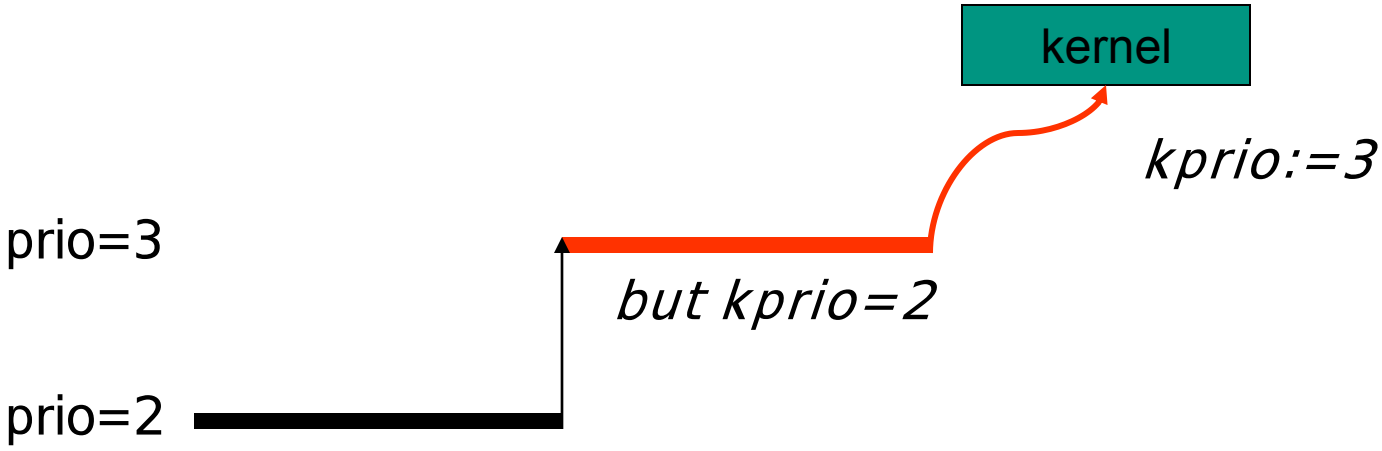
# LIPC Chains


UTCB
A's USP
wait
B's USP
wait
C's USP
wait
current thread
KTCB
RUNNING
WAITING
WAITING
WAITING
esp0

# LIPC Chains



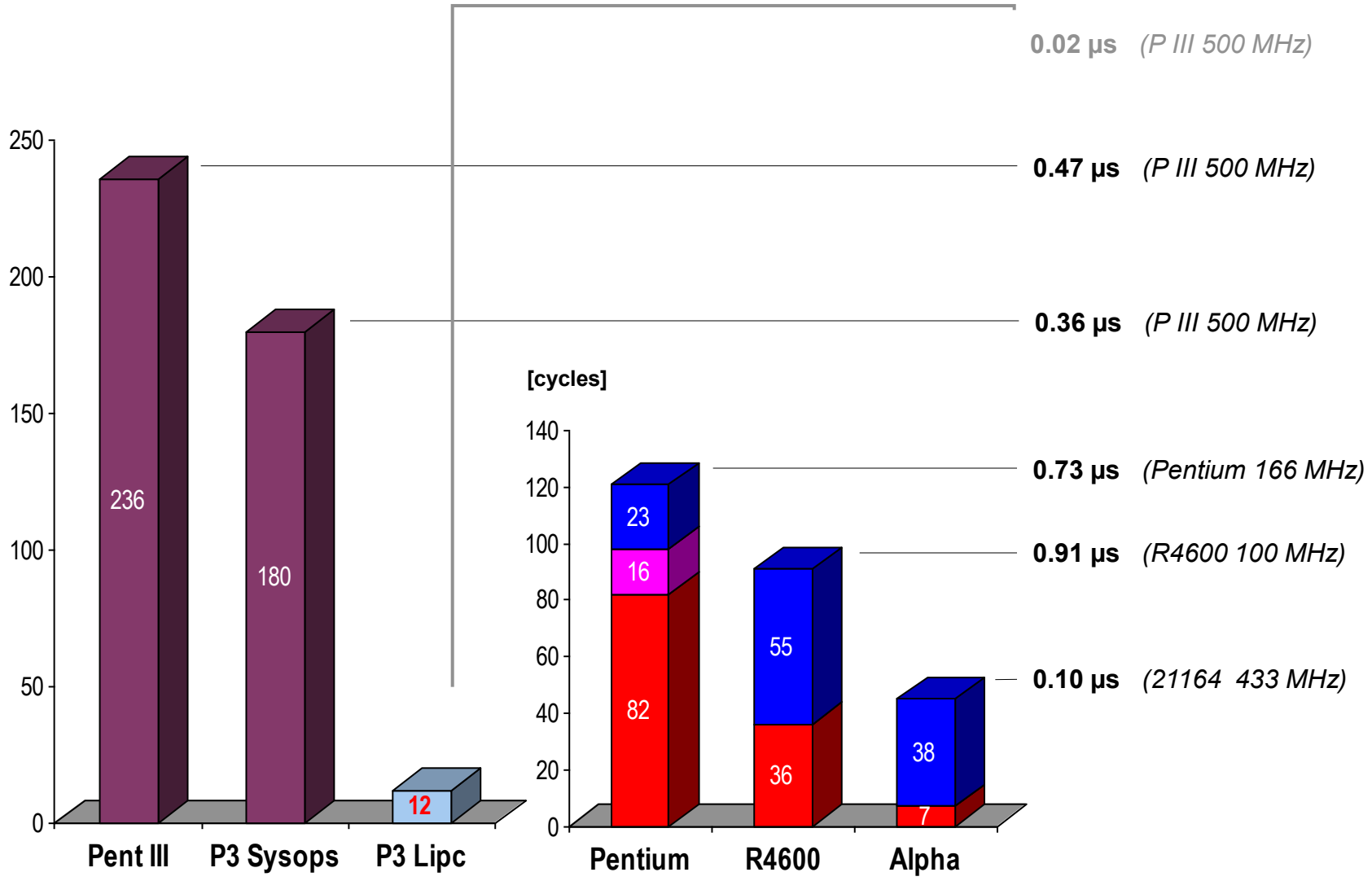
# What About Priorities?



# Safety & Security

- Threads can only destroy their own task.
  - Possible even without lazy switching.
- Threads can only cheat about their identity within their own task.
  - Affects only own task.
- Threads cannot modify their effective priority, uid, etc.

# IPC Performance Promise – May 2001



# IPC Performance – Prototype

- LIPC: 23 cycles
  - 1/15th of regular IPC (no sysops, no fastpath)
- Overhead on IPC due to LIPC extensions
  - 43 cycles intra-AS IPC
  - 146 cycles inter-AS IPC
    - UTCB synchronization

Too much for real-world systems:
P3 inter-AS IPC was only 236 cycles w/o LIPC support!

- Overhead due to kernel fixup
  - ???

# Limitations of LIPC

- Intra address space only
- Register-only IPC, no map/grant/string
- Always send and receive phase
- Infinite receive timeout

- Tricky
  - Change from Wait_for_X to Wait_for_Any